HIGH TECH

ДМИТРИЙ ОСИПОВ

ГРАФИКА В ПРОЕКТАХ DELPHI

СИМВОЛ®

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-122-3, название «Программирование в ASP.NET AJAX» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

HIGH TECH

Графика в проектах Delphi

Дмитрий Осипов

Санкт-Петербург — Москва
2008

Серия «High Tech»

Дмитрий Осипов

Графика в проектах Delphi

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Выпускающий редактор	*П. Щеголев*
Редактор	*Ю. Бочина*
Художник	*В. Гренда*
Корректор	*Л. Минина*
Верстка	*Д. Орлова*

Осипов Д.

Графика в проектах Delphi. – СПб: Символ-Плюс, 2008. – 648 с.: цв. ил.

ISBN-13: 978-5-93286-134-9
ISBN-10: 5-93286-134-7

В книге Дмитрия Осипова «Графика в проектах Delphi» представлен уникальный материал, посвященный программированию деловой графики для современных версий Windows. Рассмотрены графический механизм системы, функции прикладного интерфейса программирования GDI (Graphics Device Interface), методы работы с графикой средствами визуальной библиотеки Delphi и тонкости современной графической библиотеки Windows GDI+. Обсуждаются особенности управления цветом и вывода текста, рисование примитивов, страничные и мировые преобразования, форматы растровых и векторных рисунков, организация работы с печатающим устройством, обработка метаданных в современной цифровой фотографии и приемы улучшения качества изображений, цветовая коррекция и многое другое, без чего нельзя создать интерфейс современного программного продукта.

Книгу отличает глубина и ясность изложения материала, поэтому она будет полезна как начинающему программисту, так и профессионалу, который сможет использовать ее как справочник по функциям и методам среды разработки Delphi.

ISBN–13: 978–5–93286–134–9
ISBN–10: 5–93286–134–7

© Дмитрий Осипов, 2008
© Издательство Символ-Плюс, 2008

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, www.symbol.ru. Лицензия ЛП N 000054 от 25.12.98.

Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 29.08.2008. Формат 70×100^{1}/$_{16}$. Печать офсетная.
Объем 40,5 печ. л. Тираж 2000 экз. Заказ №

Отпечатано с готовых диапозитивов в ГУП «Типография «Наука»
199034, Санкт-Петербург, 9 линия, 12.

Оглавление

Введение 11

I. Graphics Device Interface (GDI) 15

1. Программирование на Windows API 17

Дескрипторы, или особенность доступа к объектам Windows 18
Описание класса главного окна 19
Регистрация класса окна 21
Создание экземпляра окна 22
Отображение окна 23
Организация цикла обработки сообщений 24
Оконная процедура 28
Отправка сообщений 28
Листинг программы 29
Дочерние окна 32
Субклассирование 33

2. Контекст графического устройства 38

Дескриптор контекста для экрана и печатающих устройств 40
Контекст окна приложения 41
Освобождение дескриптора контекста 46
Контекст в памяти (совместимый контекст) 47
Доступ к стандартным объектам контекста 48
Доступ к текущему объекту контекста 49
Информационный контекст 50
Восстановление состояния контекста 51
Управление объектами GDI 52
Разработка хранителя экрана 54
Приложение. Функция GetDeviceCaps() 56

3. Управление устройствами видеовывода 61

Сбор информации об устройствах видеовывода 62
Изменение настроек экрана 65

4. Кисти ... 68

Стандартные кисти ... 68
Атрибуты кисти, структура TLogBrush ... 70
Создание логической кисти ... 73

5. Перья ... 77

Стандартные перья GDI ... 77
Косметическое перо ... 79
Геометрическое перо ... 81

6. Линии и кривые ... 86

Позиционирование пера ... 86
Линии ... 87
Дуги ... 92
Кривые ... 95

7. Простейшие геометрические фигуры и заливка областей ... 100

Функции определения прямоугольных областей ... 101
Заливка прямоугольной области ... 105
Простейшие геометрические фигуры ... 108
Режим заполнения сложной области ... 112
Имитация элементов управления ... 112
Имитация фокуса ввода ... 115
Вывод заголовка окна ... 116
Пассивное состояние элемента управления ... 117

8. Траектории ... 120

Создание траектории ... 121
Вывод траектории ... 122
Преобразование траектории в набор отрезков ... 125
Анализ траектории ... 126
Преобразование траектории в регион ... 128

9. Регионы ... 130

Создание региона ... 130
Вывод региона ... 133
Операции с регионами ... 135
Объединение регионов ... 136
Преобразование региона в прямоугольный регион ... 137
Получение информации о регионе ... 137

10. Отсечение и регионы контекста устройства ... 141

Окно нестандартной формы ... 144
Обращение к регионам контекста устройства ... 145

Определение региона отсечения ... 146
Проверка вхождения в регион отсечения ... 149
Определение метарегиона ... 149
Перерисовка региона ... 151

11. Системы координат и режимы отображения ... 153

Взаимные преобразования координат ... 155
Общие характеристики режимов отображения ... 157
Настройка страничных координат и координат устройства ... 160
Метрические режимы отображения ... 165
Пользовательские режимы отображения ... 166

12. Мировые координаты и аффинные преобразования ... 175

Перевод контекста в мировые координаты ... 175
Аффинные преобразования на плоскости ... 176
Пример «Стрелочные часы» ... 182

13. Представление цвета в RGB-модели ... 186

Хранение данных о цвете в памяти ... 187
Определение характеристик цвета контекста устройства ... 189
Макросы ... 189
Работа с отдельным пикселом ... 190
Системные цвета ... 193
Альтернативные цветовые модели ... 195

14. Цветовые палитры ... 196

Перевод дисплея в 8-битный режим ... 197
Структура палитры ... 198
Макросы для работы с палитрой ... 199
Системная палитра ... 199
Логическая палитра ... 202
Сообщения, связанные с изменением палитры ... 206
Поиск цвета в палитре ... 207

15. Аппаратно-зависимые растры ... 209

Представление монохромного DDB в памяти ... 210
Создание DDB ... 211
Загрузка растра из ресурса ... 216
Универсальная функция отображения битового образа ... 218

16. Аппаратно-независимые растры ... 220

Формат DIB-растра и файла BMP ... 220
Отображение DIB-растра ... 224
Пример загрузки образа DIB из файла BMP ... 226

Перенос пикселов между DIB и DDB . . . 228
DIB-секция . . . 229
Пример загрузки образа DIB из файла BMP с помощью DIB-секции . . . 232

17. Растровые операции . . . 235

Участники растровых операций . . . 236
Бинарные растровые операции, ROP2 . . . 237
Тернарные растровые операции . . . 239
Кватернарные операции, функция MaskBlt . . . 246
Прозрачность . . . 246

18. Расширенный формат метафайла, EMF . . . 251

Структура метафайла . . . 252
Загрузка метафайла из файла . . . 256
Создание расширенного метафайла . . . 257
Размещение метафайла в памяти . . . 259
Копирование метафайла . . . 259
Построчное воспроизведение метафайла . . . 260
Комментарий к метафайлу . . . 263

19. Шрифты . . . 265

Набор символов . . . 267
Ключевые метрики логического шрифта . . . 269
Описание логического шрифта, структура TLogFont . . . 271
Сбор информации об установленных шрифтах . . . 271
Доступ к системным шрифтам . . . 274
Логический шрифт . . . 274
Инсталляция и удаление шрифта . . . 280

20. Вывод текста . . . 283

Простейшие приемы вывода текста . . . 283
Управление выводом текста средствами контекста . . . 286
Приемы форматирования текста . . . 289
Дополнительные эффекты при выводе текста . . . 294

21. Альфа-наложение и градиентная заливка . . . 299

Альфа-наложение . . . 299
Градиентная заливка области . . . 305

II. Графика в VCL . . . 309

22. Визуальная библиотека компонентов Delphi . . . 311

Концепция ООП и опорные классы VCL . . . 312
Простейшие графические объекты VCL . . . 317

Глобальный объект «экран» – класс TScreen . . . 327
Глобальный объект «монитор» – класс TMonitor . . . 329

23. Холст VCL – класс TCanvas . . . 331

Линии и кривые . . . 333
Простейшие геометрические фигуры . . . 335
Заливка области . . . 337
Вывод текста . . . 338
Работа с холстом в многопоточном режиме . . . 340

24. Растровые и векторные изображения в VCL . . . 341

Класс TGraphic . . . 341
Иконка – класс TIcon . . . 344
Растровое изображение – класс TBitmap . . . 345
Метафайл – класс TMetafile . . . 350
Класс TJPEGImage . . . 352
Хранилище изображения – класс TPicture . . . 354

25. Коллекционируем изображения . . . 356

Контейнер изображений, класс TImageList . . . 357
Экспорт пиктограмм из контейнера . . . 364
Взаимодействие с элементами управления . . . 365

26. Графические элементы управления VCL . . . 367

Класс TGraphicControl . . . 367
Изображение, компонент TImage . . . 376
Фигура, компонент TShape . . . 377
Область для рисования, компонент TPaintBox . . . 378
Разделитель, компонент TSplitter . . . 378
Рельефная панель, компонент TBevel . . . 379
Быстрая кнопка, компонент TSpeedButton . . . 380
Метка, компонент TLabel . . . 381

27. Организация работы с принтером . . . 383

Работа с принтером средствами Windows . . . 384
Технические характеристики принтера . . . 386
Описание принтера в Delphi, класс TPrinter . . . 386
Печать многострочного текста . . . 391
Печать изображений . . . 392
Окно предварительного просмотра . . . 393
Преобразование цветного изображения
для печати на монохромном принтере . . . 396
Диалог с принтером . . . 397

III. GDI+ ... 403

28. Введение в GDI+ ... 405

Мифы и реальность ... 406
Подготовка к работе ... 408
Соглашение об именовании классов GDI+ в проектах Delphi ... 409
Основной объект – холст GDI+ ... 409
Представление цвета в GDI+ ... 411
Структуры определения координат и размеров ... 413
Отладка проектов GDI+ ... 414

29. Кисти GDI+ ... 425

Сплошная кисть TGPSolidBrush ... 426
Узорная кисть TGPHatchBrush ... 427
Текстурная кисть TGPTextureBrush ... 430
Аффинные преобразования кистей ... 434
Кисть с линейной градиентной заливкой TGPLinearGradientBrush ... 436
Градиентная кисть сложной формы TGPPathGradientBrush ... 442

30. Перья GDI+ ... 447

Создание пера ... 448
Цвет и толщина пера ... 449
Стиль пера ... 450
Наконечники пера ... 454
Стык линий ... 457
Расслоение пера ... 459

31. Траектории GDI+ ... 461

Траектория, класс TGPGraphicsPath ... 462
Последовательный просмотр траектории, класс TGraphicsPathIterator ... 480

32. Регионы GDI+ ... 484

Регион, класс TGPRegion ... 484
Холст GDI+ и регион отсечения ... 495

33. Графические примитивы и заливка областей в GDI+ ... 499

Прямые и ломаные линии ... 500
Кривые ... 502
Простейшие фигуры ... 506

34. Координатные системы и преобразования в GDI+ ... 510

Страничная система координат в GDI+ ... 511
Мировые координаты, матрица TGPMatrix ... 513
Мировые преобразования, класс TGPGraphics ... 528

35. Изображения в GDI+ ... 530

Кодеры и декодеры изображений ... 531
Класс TGPImage ... 533
Роль хоста при выводе рисунков, метод DrawImage ... 551

36. Метаданные EXIF ... 557

Чтение метаданных EXIF ... 560
Редактирование метаданных ... 562
Миниатюра изображения ... 563

37. Особенности работы с битовыми образами и метафайлами ... 566

Битовые образы, класс TGPBitmap ... 567
Метафайл, класс TGPMetafile ... 574

38. Работа со шрифтами в GDI+ ... 588

Шрифт, класс TGPFont ... 589
Семейство шрифтов, класс TGPFontFamily ... 592
Коллекции шрифтов ... 593

39. Операции с текстом в GDI+ ... 597

Методы TGPGraphics по выводу текста ... 597
Форматирование текстовой строки, класс TGPStringFormat ... 600
Вывод символа в точной позиции ... 607
Исследование строки ... 609

40. Качество вывода и коррекция цвета ... 616

Сглаживание ... 616
Порядок наложения ... 618
Интерполяция растра ... 619
Повышение качества вывода текста ... 621
Сохранение состояния холста GDI+ ... 623
Коррекция цвета, класс TGPImageAttributes ... 626

Заключение ... 637

Литература ... 639

Алфавитный указатель ... 641

Введение

Говорят, что когда в начале 70-х годов XX века Николауса Вирта попросили дать определение программы, он предложил краткую и изящную формулировку:

программа = структуры + алгоритмы

Интересно, что бы ответил разработчик языка программирования Pascal на этот же вопрос, если бы он был задан сегодня? Не берусь говорить от имени этого блестящего ученого, профессора Цюрихского университета, обладателя высшей награды в области информатики – премии им. Тьюринга. Со времен создания языка Pascal минуло четыре десятилетия. За это время электронные технологии достигли невиданных для тех времен высот. Сегодня в любой квартире или офисе мы обнаружим компьютер с процессором, производительность которого на порядки превосходит возможности самых передовых ЭВМ 1970-х годов. Вместе с развитием электроники совершенствовались и операционные системы (ОС) – от простейших DOS до современных многозадачных систем. Одним из значимых этапов в развитии операционных систем стало появление у них доброжелательных графических оболочек. Первопроходцем в этой области считается Apple Computer с разработанной в январе 1983 года ОС «Liza». Ровно год спустя Apple анонсировала знаменитый Macintosh. Первая же версия Windows (в те времена еще не столь широко известной компании Microsoft) увидела свет только в 1985 году.

Полагаю, что сегодня в определение современной программы кроме структур и алгоритмов необходимо добавить третий ингредиент – графический интерфейс. В подтверждение этих слов можно привести небывалый успех Windows. Во многом благодаря продвинутым графическим возможностям вышедшая в 1990 году операционная среда Windows 3.0, а несколько позднее (в 1995 г.) – полноценная операционная система Windows 95 практически полностью захватили нишу домашних и офисных систем для компьютеров, совместимых с IBM PC. Причем со сцены были сметены не только невзрачные DOS различных версий, но и весьма удачные ОС, например OS/2 Warp фирмы IBM. На сегодняшний день можно говорить если не о тотальном господстве Windows (существует общепризнанная система Unix, ряд энтузиастов работает в Linux, QNX, FreeBSD), то, по крайней мере, о существенном распространении современных версий операционных систем Microsoft (Windows 2000, Windows XP, Windows 2003, Windows XP x64 и сверхновой Windows Vista).

Настоящая книга посвящена программированию двумерной графики для 32- и 64-разрядных версий Microsoft® Windows® в среде Delphi. Подавляющее большинство приведенных примеров сохранят работоспособность практически во всех версиях среды, начиная с Borland Delphi 2.0 и заканчивая современной Delphi от CodeGear. Ареал нашего исследования отображен на рис. 1.

Что такое Windows GDI (Graphics Device Interface)? Это набор стандартных функций двухмерной графики, которые обеспечивают приложения возможностью общаться с любым установленным в Windows графическим устройством. Прикладной программе совсем не обязательно владеть всеми техническими тонкостями эксплуатации принтера или монитора! Библиотека GDI безукоризненно выполнит обязанности посредника между запросами приложения и драйверами графических устройств.

Группа функций Windows GDI – это существенный подкласс функций, входящих в состав прикладного интерфейса программирования Windows API. Основная часть этих функций сосредоточена в системной библиотеке gdi32.dll. Из этой библиотеки они экспортируются модулем windows – доминирующим элементом из состава библиотеки времени выполнения (Run Time Library, RTL) среды программирования Delphi. За ним следует царство визуальной библиотеки компонентов (Visual Components Library, VCL). Обращения к функциям GDI инкапсулируются в графических классах Delphi, ключевые классы объединены в модуле graphics.

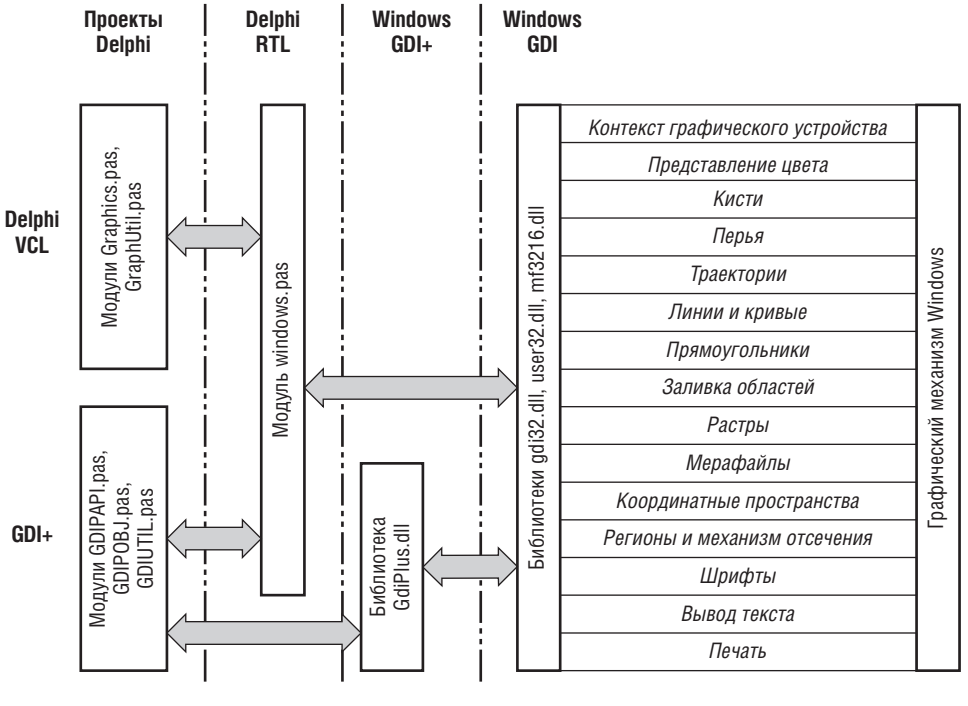


Рис. 1. Взаимодействие Delphi и графических библиотек Windows

Технология GDI+ представляет собой дальнейшее развитие графического механизма операционной системы Windows. Это объектно-ориентированное воплощение библиотеки GDI, значительно расширяющее ее возможности, реализованное в виде новой библиотеки GdiPlus.dll, появившейся на свет с выходом Windows XP.

Излагаемый в книге материал разделен на три части. В первой части подробно рассматриваются функции прикладного интерфейса программирования Windows API, посвященные разработке графического интерфейса приложений. По ходу изучения глав первой части мы научимся:

- создавать приложения без помощи VCL – нашей опорой станут функции операционной системы Windows;
- работать с графическим контекстом устройства;
- создавать перья, кисти, регионы, траектории;
- рисовать линии, кривые и простейшие геометрические фигуры;
- управлять режимами отображения;
- осуществлять страничные и мировые преобразования;
- проводить сложные растровые операции;
- представлять изображения в растровых (BMP, DIB) и векторных форматах (WMF и EMF);
- выводить простейшие и форматированные текстовые надписи.

Во второй части книги рассматриваются особенности программирования графики средствами визуальной библиотеки компонентов Delphi. Здесь мы обсудим:

- особенности инкапсуляции функций GDI в компонентах Delphi;
- графические классы и элементы управления среды;
- диалоговые окна, обеспечивающие настройку принтера;
- организацию работы с печатающими устройствами.

В третьей части книги изучается современная библиотека GDI+. Материал посвящен:

- изучению новейших возможностей объектов GDI+;
- работе с графическими форматами JPEG, TIFF, GIF, EMF+;
- обработке метаданных современной цифровой фотографии EXIF;
- методам улучшения качества изображений и цветовой коррекции.

Соглашения

Для акцентирования внимания читателя на ключевых частях излагаемого материала такой текст отмечен особым форматированием и пиктограммами. Речь идет о трех следующих замечаниях:

«На заметку»
Таким способом в тексте книги выделен материал, который вы должны принять к сведению. Обычно это советы, комментарии или замечания.

«Внимание!»
Текст, отмеченный восклицательным знаком, определяет однозначные правила действий программиста в той или иной ситуации.

«Стоп!»
Однозначный запрет, заостряет внимание на характерных ошибках. Короче говоря – никогда так не делайте.

Описываемые в книге функции могут быть отмечены пиктограммами «Windows API» или «Windows GDI».

Первая из пиктограмм говорит о том, что обсуждаемая функция относится к функциям прикладного интерфейса программирования Microsoft Windows и отражена в справочной системе Windows SDK (Platform Software Development Kit). Вторая пиктограмма конкретизирует, что указанная функция относится к подклассу функций графического интерфейса Windows API (соответствует ветви «Graphics and Multimedia» в SDK). Синтаксические конструкции всех функций API «переведены» с языка Си на язык Delphi. Если функция или процедура не снабжена пиктограммой, то это означает, что она входит в состав функций Borland Delphi.

Кроме того:

- Впервые встречающиеся термины и определения выделены курсивом.
- Впервые встречающиеся аббревиатуры комментируются.
- Все таблицы и рисунки в книге пронумерованы по принципу: номер главы, точка-разделитель, порядковый номер в главе.
- Код примеров, синтаксические конструкции даны моноширинным шрифтом, зарезервированные слова выделены полужирным шрифтом.
- Помимо содержания книга включает подробный предметный указатель, позволяющий найти в тексте интересующую читателя функцию, класс или структуру по ее названию.

В тексте можно обнаружить ссылки на примеры, исходные тексты программ, и графические файлы, доступные для скачивания на сайте издательства по адресу: *www.symbol.ru/library/Delphi_graphic*.

I

Graphics Device Interface (GDI)

1

Программирование на Windows API

Раз наша книга посвящена исследованию неотъемлемой части прикладного интерфейса программирования – функций GDI и GDI+, то наиболее подходящим практическим способом их изучения станет разработка программ на базе функций Windows API. Такой подход позволяет заглянуть в само сердце графической подсистемы и исключает необходимость отвлекаться на частности. Поэтому подавляющее число примеров написано без использования визуальной библиотеки компонентов (VCL) – на «голом» API. Исключение составили лишь листинги для глав, посвященных графике VCL, и примеры повышенной сложности. Именно поэтому первая глава книги посвящена не сколько графике, сколько основам программирования на Windows API.

Библиотека визуальных компонентов Delphi значительно упрощает процесс создания программы (скрывая от нас механизм взаимодействия приложения с операционной системой), но за это мы расплачиваемся относительно неэффективным результирующим кодом. Для устранения этого недостатка следует работать на более низком уровне – уровне Windows API. Программирование без опоры на библиотеку визуальных компонентов обязывает программиста в совершенстве знать операционную систему Microsoft Windows, особенности взаимодействия программы и ОС, возможности, предоставляемые функциями Windows API.

На протяжении всего периода работы приложения под управлением Windows оно находится под неусыпным контролем со стороны операционной системы. Для этого с момента создания до момента разрушения *главного окна* приложения операционная система поддерживает с ним активный диалог. Беседа ведется посредством отправки сообщений *оконной процедуре* (*window procedure*) главного окна приложения. Что понимается под словосочетанием «отправить сообщение»? Это значит, что операционная система осуществит запись о каком-то событии в область памяти, доступную оконной процедуре. Указанная область памяти называется *очередью программы*, или *очередью сообщений*. Очередь может хранить несколько сообщений, она устроена по принципу стека «первым пришел – первым вышел» (*FIFO, First Input First Output*).

Программа также вправе отправить сообщение Windows. В этом случае сообщение от программы помещается в другую, общесистемную очередь Windows. Заметьте, что общесистемная очередь существует в одном-единственном экземпляре и в нее помещаются сообщения от всех окон.

Для того чтобы приложение не пропустило предназначенное ему послание, оно периодически просматривает свой «почтовый ящик» на предмет новых поступлений. Для этого в любой программе (кстати, и в самой Windows) организуется *цикл обработки сообщений* (*message loop*). И если по какой-то причине у приложения остановится этот цикл, то оно безнадежно «зависнет», а если остановится цикл операционной системы, то нас спасет только перезагрузка.

После того как мы выяснили самую страшную тайну Microsoft о том, что идея жизнедеятельности Windows позаимствована у обычного почтового отделения, а программы – большие специалисты в эпистолярном жанре (просто-напросто посылают друг-другу письма), обозначим этапы, через которые надо пройти программисту для того, чтобы создать простейшее жизнеспособное приложение для Windows. Для этого необходимо:

- Описать макет будущего главного окна приложения. Для этого заполняются поля специализированной структуры, называемой классом окна.
- Зарегистрировать этот класс в Windows.
- Создать экземпляр главного окна приложения.
- Отобразить окно на экране компьютера.
- Организовать цикл обработки сообщений Windows.
- Описать оконную процедуру, в рамках которой наша программа станет обслуживать все поступающие в ее адрес послания.

Дескрипторы, или особенность доступа к объектам Windows

На протяжении всех глав книги, посвященных GDI и GDI+, для обращения к тому или иному объекту мы будем пользоваться *дескрипторами* объектов. Для начинающего программиста, работающего исключительно с компонентами среды программирования Delphi, это не вполне понятный термин. В Delphi каждый объект гордо несет уникальное имя, что значительно упрощает обращение к его свойствам и методам. Допустим, нам необходимо изменить заголовок кнопки. В стандартном проекте Delphi для этого достаточно указать имя компонента, через точку – соответствующее свойство и присвоить ему новое значение:

```
Button1.Caption:='Я - кнопка!';
```

При работе на Windows API все несколько усложняется. Операционная система Windows тщательно охраняет свои ключевые объекты, она не допускает получение к ним непосредственного доступа. При создании объекта ему назначается уникальное целочисленное значение – его идентификатор. Работа с такими объектами возможна только при передаче указателя на них

в соответствующие функции. В терминах Windows такие указатели называются дескрипторами. Хотя физически дескриптор представляет собой обычное целое число, но для того, чтобы исходный код программы стал более читабельным, установлен ряд правил об именах дескрипторов. Название типа дескриптора всегда начинается с символа «H» (первый символ английского слова *handle*). Кроме того, по существующим соглашениям для ключевых дескрипторов закреплены постоянные имена, так, например, при чтении документации SDK можно быть твердо уверенным, что:

```
Instance:HINSTANCE; //дескриптор экземпляра приложения
Wnd: HWND;          //дескриптор окна
DC:HDC;             //дескриптор контекста устройства
Brush:HBRUSH;       //дескриптор кисти
Pen:HPEN;           //дескриптор пера
Font:HFONT;         //дескриптор шрифта
GDIObj:HGDIOBJ;     //дескриптор GDI объекта
```

Это далеко не полный перечень всех существующих дескрипторов, но думаю, что вы уже заметили созвучие их с именами классов из комплекта VCL Delphi и, столкнувшись с новым названием дескриптора, легко соотнесете его со знакомым объектом.

Описание класса главного окна

В классическом приложении для Windows всегда существует главное окно, в котором осуществляется обработка сообщений, поступающих приложению. Главное окно всегда создается на основе специальной структуры, называемой *классом окна*. В заголовочных файлах C++ из пакета Microsoft Visual Studio (и, соответственно, в SDK) упомянутая структура называется `WNDCLASSEX`. В Delphi имеется аналог указанной структуры, это запись `TWndClass`. Запись содержит десять полей:

```
type TWndClass = packed record
  style: UINT;              //стиль (комбинация стилей) окна
  lpfnWndProc: TFNWndProc;  //адрес оконной процедуры
  cbClsExtra: Integer;      //выделенная память
  cbWndExtra: Integer;      //выделенная память
  hInstance: HINST;         //дескриптор экземпляра программы
  hIcon: HICON;             //дескриптор иконки окна
  hCursor: HCURSOR;         //курсор окна
  hbrBackground: HBRUSH;    //кисть, закрашивающая окно
  lpszMenuName: PAnsiChar;  //имя меню
  lpszClassName: PAnsiChar; //имя класса окна
end;
```

Стиль создаваемого окна определяется *первым* по счету полем структуры `Style`. Сюда может быть передано одно значение или комбинация значений, определяющих особенности внешнего вида и поведения окна. Так как большинство стилей окна оказывают весьма существенное влияние на порядок вывода изображений, рекомендую изучить табл. 1.1.

Таблица 1.1. Стили окна, поле style структуры TWndClass

Стиль	Описание
CS_BYTEALIGNCLIENT	Автоматическое выравнивание клиентской области окна по горизонтали по границе байта. Этот флаг позволяет несколько повысить производительность вывода изображения.
CS_BYTEALIGNWINDOW	То же самое, что и CS_BYTEALIGNCLIENT, только теперь речь идет обо всем окне.
CS_CLASSDC	Позволяет сформировать единый стиль для вывода в многооконном приложении, так как при создании нескольких окон одного класса они станут разделять один единственный контекст графического устройства.
CS_DBLCLKS	Позволяет окну реагировать на двойной щелчок.
CS_DROPSHADOW	При перетаскивании окна включается эффект тени. Стиль появился в Windows XP.
CS_GLOBALCLASS	Создаваемый класс окна станет доступен всем приложениям. Этот флаг часто применяется при хранении окон в динамических библиотеках.
CS_HREDRAW	При изменении горизонтального размера окна автоматически поступает команда на перерисовку этого окна.
CS_NOCLOSE	Отключается системная кнопка закрытия окна в его заголовке.
CS_OWNDC	Для каждого из окон одного класса создается уникальный контекст графического устройства.
CS_PARENTDC	Каждое дочернее окно, созданное на базе этого класса, будет использовать контекст устройства родительского окна.
CS_SAVEBITS	Позволяет быстро восстанавливать часть окна, скрытую под другими окнами. Для этого затененная область сохраняется в виде растрового образа и копируется в окно в момент его перерисовки.
CS_VREDRAW	При изменении вертикального размера окна автоматически поступает команда на перерисовку этого окна.

Особо подчеркну то, что если программист рассчитывает использовать комбинацию стилей, то между стилями необходимо ставить оператор OR (логическое «ИЛИ»). Если приложение не будет обладать дочерними окнами, то, в самом общем случае, стилем окна объявляют комбинацию: CS_VREDRAW и CS_HREDRAW. Это значит, что окно должно перерисовываться при изменении вертикального или горизонтального размера.

Возвращаемся к структуре TWndClass. Ее *второе* поле (lpfnWndProc) хранит адрес оконной процедуры, в которой описывается реакция приложения на поступающие извне сообщения. *Третье* и *четвертое* поля определяют количество байт памяти, дополнительно выделяемых для хранения структуры окна и экземпляра окна в памяти. Всегда устанавливайте эти поля в ноль, тогда задачу распределения памяти решит Windows. *Пятое* поле hInstance желает знать, какое приложение готовит к регистрации класс окна. *Шестое*, *седь-*

мое и *восьмое* поля соответственно хранят дескрипторы иконки, курсора и кисти (заливающей клиентскую часть) окна. *Девятое* поле lpszMenuName задает меню класса окна. И последнее, *десятое* поле содержит имя класса.

В случае построения простейшего главного окна приложения потребуется следующее описание класса:

```
with WindowClass do
begin
 Style := CS_HREDRAW or CS_VREDRAW; {Стиль окна класса}
 lpfnWndProc := @WindowProc;        {Указатель на оконную процедуру}
 cbClsExtra := 0; {Выделенная память, самостоятельно используемая программой}
 cbWndExtra := 0; {Выделенная память, самостоятельно используемая программой}
 hInstance := 0;  {Ссылка на экземпляр программы}
 hIcon := LoadIcon (0, idi_Application); {Дескриптор маленькой иконки окна}
 hCursor := LoadCursor (0, idc_Arrow);   {Дескриптор курсора}
 hbrBackground := GetStockObject(White_Brush);{Дескриптор кисти окна}
 lpszMenuName := '';       {Ссылка на строку имени меню}
 lpszClassName := AppName; {Имя класса}
end;
```

В Windows существует достаточно большой спектр готовых описаний классов окон. Это кнопки, переключатели, строки ввода и другие стандартные элементы управления.

Регистрация класса окна

Справившись со всеми десятью полями класса окна, переходим к следующему этапу – регистрации класса будущего окна в Windows. В качестве крупного специалиста по данному вопросу рекомендую функцию:


```
Function RegisterClass(WndClass:TWndClass):Integer;
```

В единственный параметр функции направляется заполненная ранее структура TWndClass. В случае успеха функция возвращает уникальный идентификатор зарегистрированной структуры (в терминах Windows – атомарное значение, ATOM). Если регистрация завершилась неудачей, то получаем ноль. На такое неуважительное к себе отношение программа просто обязана отреагировать, например, следующим образом:

```
If RegisterClass(WindowClass) = 0 then Halt;
```

Если попытка регистрации нового класса завершается неудачно, то выполнение программы прекращается – выполняется суровая директива Halt(), моментально останавливающая приложение.

Если на момент обращения к функции RegisterClass() *в системе уже был зарегистрирован (например, другим приложением) идентичный класс окна, то функция возвратит атомарное значение, оставшееся после предыдущей регистрации.*

Создание экземпляра окна

Факт успешной регистрации класса окна говорит о том, что операционная система не против того, чтобы на основе зарегистрированного класса был создан экземпляр окна. Теперь наступает черед вводить в бой тяжелую кавалерию – очередную системную функцию Windows:

```
Function CreateWindow(
         ClassName : PChar;     // имя зарегистрированного класса
         lpWindowName : PChar; // заголовок окна
         Style : Longword;      // стиль окна
         x : Integer;           // горизонтальная позиция окна
         y : Integer;           // вертикальная позиция окна
         Width : Integer;       // ширина окна
         Height : Integer;      // высота окна
         WndParent :  HWND;     // дескриптор родительского окна
         Menu : HMENU;          // дескриптор меню окна
         Instance : HINSTANCE; // дескриптор экземпляра приложения
          lpParam : pointer     // указатель на дополнительную информацию
         ) : HWND;
```

Первый параметр `ClassName` представляет собой имя только что зарегистрированного класса окна. В Windows предусмотрен целых ряд заранее описанных заготовок классов. Так, для создания стандартной кнопки – имя класса «BUTTON», для создания комбинированного списка – «COMBOBOX», строки ввода – «EDIT», метки текста – «STATIC», и т. д. В нашем случае мы намерены создать эксклюзивное главное окно приложения на основе описанного и зарегистрированного именно нами класса окна. Поэтому значение параметра `ClassName` должно совпадать со значением последнего поля в классе окна (структуре `TWndClass`). Второй параметр соответствует тексту заголовка окна. Индивидуальные характеристики или, как чаще говорят, стиль создаваемого экземпляра окна следует указывать в третьем параметре `Style`. Перечень стилей настолько богат, что я лучше отправлю вас к справочной системе. Только отмечу, что для описания стандартного окна вполне приемлемо значение `WS_OVERLAPPEDWINDOW`. Это комбинация стилей: `WS_OVERLAPPED`, `WS_CAPTION`, `WS_SYSMENU`, `WS_THICKFRAME`, `WS_MINIMIZEBOX` и `WS_MAXIMIZEBOX`, указывающая функции на то, что мы рассчитываем получить окно с заголовком, рамкой и системными кнопками в правой части заголовка. Следующие четыре аргумента связаны с местом вывода и размерами будущего окна и поэтому в особых комментариях не нуждаются, а вот восьмой параметр `WndParent` весьма важен. Это дескриптор родительского окна или элемента управления. У главного окна программы он нулевой. Но если вы создаете дочерние окна (не забывайте, что кнопка, панель, строка ввода и много чего другого также являются окнами), то в этом параметре следует указывать ссылку на окно-владелец. Следующий параметр `Instance` является дескриптором приложения, в котором создается окно. Это глобальная переменная, за ее объявление отвечает среда программирования. Последний необязательный параметр `lpParam` – указатель на дополнительные данные окна. При удачном выполнении функция возвратит уникальный дескриптор созданного окна.

Для работы с дескриптором приложения не стоит создавать свою собственную переменную `HInstance: LongWord`*, так как она уже объявлена в модуле SysInit среды Delphi.*

```
var Wnd : HWND;
…
Wnd := CreateWindow          // создает окно и возвращает его дескриптор
     (AppName,               // имя класса
      'Demo',                // заголовок окна
      ws_OverlappedWindow,   // стиль окна
      cw_UseDefault,         // позиция верхнего левого угла окна по горизонтали
      cw_UseDefault,         // позиция верхнего левого угла окна по вертикали
      cw_UseDefault,         // ширина окна
      cw_UseDefault,         // высота окна
      0,                     // родительское окно
      0,                     // меню
      HInstance,             // указывает на экземпляр программы
      nil);
```

В распоряжении программистов имеется еще более совершенная функция `CreateWindowEx()`*. Она позволяет создавать окна с расширенными возможностями.*

Отображение окна

После создания экземпляра окна осуществляем его вывод на экран. Эту почетную миссию выполнит функция:

Win API

```
Function ShowWindow(Wnd : HWND; CmdShow : Integer;) : Boolean;
```

С первым параметром все просто, это дескриптор только что созданного окна (результат работы функции `CreateWindow`). Второй параметр устанавливает режим отображения окна. Например, для того чтобы вывести окно, нужно использовать константу `SW_SHOW`. Наиболее часто применяемые константы перечислены в табл. 1.2.

Таблица 1.2. Некоторые значения параметра CmdShow

Значение	Описание
SW_HIDE	Скрыть окно
SW_MAXIMIZE	Развернуть окно
SW_MINIMIZE	Свернуть окно
SW_RESTORE	Активизировать и восстановить размеры окна
SW_SHOW	Показать окно

Обновление заданной области окна

Для перерисовки всей клиентской области окна обычно используют функцию:

Win API

```
Function UpdateWindow(Wnd : HWND) :Boolean;
```

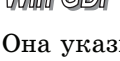
Единственный аргумент функции – дескриптор обновляемого окна. Функция информирует Windows о необходимости отправки окну сообщения WM_PAINT, заставляющего перерисовать клиентскую область окна.

Зачастую при формировании графического интерфейса приложения возникает необходимость в обновлении не всего окна, а только его части (например, области, над которой находилось окно другого приложения). Для этих целей предназначена функция:

```
function InvalidateRect(Wnd: HWND; Rct: PRect; Erase: Boolean): Boolean;
```

Здесь Wnd – дескриптор обслуживаемого окна; Rct – координаты прямоугольника, нуждающегося в обновлении; Erase – в состоянии true указывает GDI на то, что перед повторной прорисовкой области Rct она должна быть полностью стерта. Обращение к функции из кода программы вынуждает систему отправить окну Wnd сообщение WM_PAINT. Если вместо дескриптора окна функция получит неопределенный указатель, то обновятся все имеющиеся в данный момент окна (будет осуществлена широковещательная рассылка сообщений WM_ERASEBKGND и WM_NCPAINT, инициирующих перерисовку рамки, заголовка и фона окна).

Обратная задача поставлена перед функцией:



```
function ValidateRect(Wnd: HWND; Rct: PRect): Boolean;
```

Она указывает на то, что при очередной перерисовке окна Wnd прямоугольник Rct не должен быть затронут, вследствие чего этот прямоугольник удаляется из обновляемой области окна.

Функции InvalidateRect() *и* ValidateRect() *пользуются услугами функций обновления регионов –* InvalidateRgn() *и* ValidateRgn().

Организация цикла обработки сообщений

Работа приложений в операционной системе Microsoft Windows основывается на обработке сообщений. Сообщение – это реакция Windows на события, происходящие в системе. События подразделяются на аппаратные (нажатие клавиш на клавиатуре, движение мышки по коврику и т. п.) и программные (передаваемые приложениями с помощью функций SendMessage и PostMessage). Ни одно сообщение не должно (да, в общем-то, и не может) отправляться в обход операционной системы. Сформированное сообщение помещается в *очередь сообщений* (*message queue*), затем планировщик системы распределяет их в очереди соответствующих процессов. Задача программиста – научить свое приложение обрабатывать сообщения, отправляемые ему операционной системой. Задача приложения – извлекать сообщения из своей очереди, осуществлять их обработку и при этом (желательно) не «зависнуть». Общепринятое название блока команд, извлекающего сообщения из очереди, – *цикл обработки сообщений*.

В цикле обработки сообщений применяется пара функций Windows API: GetMessage() и DispatchMessage(). Определение первой функции выглядит следующим образом:

```
Function GetMessage(
     Msg : TMsg;      // указатель на структуру сообщения
     Wnd : HWND;      // дескриптор окна, созданного программой
     wMsgFilterMin: Cardinal;// нижняя граница фильтра
     wMsgFilterMax : Cardinal// верхняя граница фильтра
     ) : boolean;
```

Функция должна осуществить выбор из очереди сообщений сообщения, предназначенного окну, идентифицируемому дескриптором Wnd. С помощью GetMessage() допускается обрабатывать сообщения для всех созданных программой окон, но в этом случае вместо дескриптора окна необходимо передавать нулевое значение. Третий и четвертый параметры играют роль фильтров, ограничивающих поток сообщений. Минимальный номер принимаемого сообщения указываем в параметре wMsgFilterMin, максимальный – в wMsgFilterMax. Если оба параметра установлены в ноль, то фильтры отключаются и обслуживаются все сообщения. Ключевым аргументом выступает запись, содержащая принимаемое сообщение, – Msg. Эта запись соответствует структуре Windows tagMSG. В Delphi предусмотрен аналог данной структуры – запись TMsg.

```
type
  TMsg = packed record
    Wnd: HWND;
    message: Cardinal;
    wParam: WPARAM;
    lParam: LPARAM;
    time: DWORD;
    pt: TPoint;
  end;
```

Здесь:

Wnd – дескриптор окна, для которого предназначено сообщение, окном может быть любой экранный объект (кнопки, группы переключателей, строки ввода и т. д.);

message – число, описывающее сообщение. Для удобства в Windows для каждого числа объявлен идентификатор, начинающийся с префикса WM (*window message*). Если перевести дословно – *оконное сообщение*. Например: при перерисовке окна поступает сообщение WM_PAINT, при закрытии окна – WM_CLOSE, при нажатии клавиши на клавиатуре – WM_KEYDOWN;

wParam – в зависимости от типа сообщения, в поле окажется дополнительная информация. Кроме того, здесь может находиться идентификатор окна или элемента управления, связанного с данным сообщением;

lParam – параметр хранит индекс или указатель на некоторые вспомогательные данные в памяти;

time – время помещения сообщения в очередь;

pt – экранные координаты курсора мыши в момент помещения сообщения в очередь.

Возвращаемся к GetMessage(). Цикл обработки сообщений будет продолжаться до тех пор, пока функция не возвратит значение false, это знаменательное событие произойдет после получения сообщения WM_QUIT – завершение работы программы.

Программист имеет право регистрировать в системе свои собственные пользовательские сообщения. Для этого применяется функция RegisterWindowMessage().

В программах Delphi классический цикл обработки сообщения выглядит примерно так:

```
while GetMessage(Msg, 0, 0, 0)=true do
begin
 TranslateMessage(Msg);
 DispatchMessage(Msg);
end;
```

Здесь инструкции TranslateMessage() и DispatchMessage() передают сообщение Msg обратно операционной системе. Функция TranslateMessage() предназначена для преобразования сообщений виртуальных клавиш в соответствующий код WM_CHAR. Функция DispatchMessage() при посредничестве Windows отправляет сообщение соответствующей оконной процедуре. В табл. 1.3 приведены некоторые типы оконных сообщений Windows, они пригодятся в процессе описания оконной процедуры.

Таблица 1.3. Ключевые оконные сообщения Windows

Идентификатор	Описание
WM_ACTIVATE	Активизация (деактивизация) окна.
WM_CLOSE	Закрытие окна.
WM_KEYDOWN	Нажатие клавиши на клавиатуре.
WM_CHAR	Для некоторой клавиши было отправлено сообщение WM_KEYDOWN или WM_KEYUP.
WM_KEYUP	Клавиша на клавиатуре отпущена.
WM_LBUTTONDOWN	Нажата левая кнопка мыши.
WM_MOUSEMOVE	Переместился указатель мыши.
WM_PAINT	Команда на перерисовку клиентской области окна.
WM_TIMER	Произошло событие таймера.
WM_COMMAND	Щелчок по пункту меню, кнопке.
WM_QUIT	Программа должна быть завершена.

Сообщение WM_PAINT *– ключевой, но далеко не единственный представитель набора системных сообщений, связанных с перерисовкой. Список могут дополнить сообщения:* WM_NCPAINT, WM_NCCLASSSIZE *и* WM_SIZE.

В визуальной библиотеке компонентов Delphi обработка сообщения WM_PAINT *закамуфлирована в рамках события* OnPaint(). *В частности, указанное событие имеется у ключевого элемента проектов Delphi – формы (экземпляра класса* TForm*).*

Сообщение WM_PAINT

По сути, 90 процентов материала этой книги посвящено организации обработки сообщения WM_PAINT, связанного с необходимостью перерисовки окна или его части. Указанное сообщение может быть отправлено как операционной системой, так и приложением. В первом случае можно быть уверенным, что сообщение не было сгенерировано зря – Windows никогда не отправит WM_PAINT в адрес не нуждающегося в перерисовке окна. ОС предварительно проверит состояние региона окна, подлежащего обновлению. Если в регионе имеется участок с изменившимся с момента последней перерисовки содержимым, то тогда система поместит в очередь даже не одно, а целых три сообщения: WM_NCPAINT, WM_ERASEBKGROUND и WM_PAINT. Первое из сообщений укажет окну, что требуется перерисовать рамку и заголовок окна. Сообщение WM_ERASEBKGROUND даст команду о необходимости восстановления фона окна, для этих целей будет задействована кисть из поля hbrBackground структуры TWnd, используемой при регистрации класса окна. Наконец, WM_PAINT выполнит основную работу – обновит ту часть клиентской области окна, которая нуждается в восстановлении.

Как правило, без особой на то необходимости программисты явным образом не описывают обработку сообщений WM_NCPAINT *и* WM_ERASEBKGROUND. *В этом случае сообщения обслуживаются процедурой по умолчанию.*

Если в ходе выполнения программы возникает необходимость внеочередной перерисовки, то для генерации WM_PAINT из кода программы программисты обычно применяют функции UpdateWindow() и RedrawWindow(). Позднее мы еще не раз обратимся к этим функциям, а сейчас рассмотрим фрагмент кода, демонстрирующий порядок обработки сообщения WM_PAINT.

```
WM_PAINT   :
 begin
 DC:=BeginPaint(Window,PAINTSTRUCT);
 {другие команды}
 EndPaint(Window,PAINTSTRUCT);
 end;
```

Стандартная обработка сообщения перерисовки окна включает три этапа:

1. Получение дескриптора контекста окна путем вызова функции BeginPaint().
2. Собственно процесс перерисовки.
3. Обращение к функции EndPaint() для освобождения дескриптора контекста.

Программисту Delphi не мешает помнить, что в библиотеке VCL сообщение WM_PAINT *обслуживается на уровне класса* TWinControl. *Немного покопавшись в секции частных объявлений класса, вы обнаружите заголовок соответствующей процедуры:*

```
procedure WMPaint(var Message: TWMPaint); message WM_PAINT; _
```

По ходу книги для нас наибольший интерес будет представлять сообщение `WM_PAINT`. *Цель сообщения – уведомление окна о том, что оно или его часть нуждается в перерисовке. Приложение также в состоянии стать инициатором команды перерисовки, но для этого ему надо обратиться к функциям* `UpdateWindow()` *или* `RedrawWindow()`. *В следующей главе, посвященной контексту графического устройства, мы вновь вернемся к вопросу обслуживания сообщения* `WM_PAINT`.

Оконная процедура

Основная задача оконной процедуры – организация обработки сообщений, поступающих в адрес окна, поэтому *оконная процедура вызывается в момент получения окном сообщения*. К счастью, это не вынуждает программиста запереться на пару лет в келье, чтобы описать поведение окна при поступлении всех мыслимых и немыслимых сообщений, начинающихся с приставки «`WM_`». В Windows предусмотрена обработка сообщений по умолчанию при помощи процедуры `DefWindowProc()`. Но если при получении определенного сообщения поведение окна должно чем-то отличаться от стандартного, то будьте добры – садитесь за клавиатуру. Мы об этом еще поговорим, в частности, в подразделе, посвященном субклассированию.

Заголовок классической оконной процедуры определяется описанной в Windows функцией `WindowProc()`:

```
Function WindowProc(
                Wnd : HWND;       // дескриптор окна
                uMsg : UINT;      // идентификатор сообщения
                wParam : WPARAM; // первый (старший) параметр сообщения
                lParam : LPARAM  // второй (младший) параметр сообщения
                    ) : longint;
```

Возвращаемое значение зависит от обработанного сообщения.

Для дочерних окон (окон, созданных с флагом `WS_CHILD`*) сообщения отправляются индивидуально (в обход главного окна). Это замечание следует учитывать при разработке приложений, содержащих «цепочки» дочерних окон, например, когда на поверхности окна находится панель группировки, а ей в свою очередь принадлежат другие дочерние окна (списки, кнопки и т. д.).*

Отправка сообщений

Приложение должно уметь не только получать, но и отправлять сообщения. Для этого применяются функции `SendMessge()` и `PostMessage()`. Они обладают идентичным набором параметров:

```
Function SendMessage(
      Wnd : HWND;        // дескриптор окна-получателя
      Msg : UINT;        // отправляемое сообщение
      wParam : WPARAM;   // дополнительный старший параметр сообщения
      lParam : LPARAM    // дополнительный младший параметр сообщения
           ) : LRESULT;
```

Win API

```
Function PostMessage(
      Wnd : HWND;         // дескриптор окна-получателя
      Msg : UINT;         // отправляемое сообщение
      wParam : WPARAM;    // дополнительный старший параметр сообщения
      lParam : LPARAM     // дополнительный младший параметр сообщения
           ) : boolean;
```

Однако, несмотря на внешнюю схожесть, поведение функций существенно отличается друг от друга. Функция `SendMessage()` работает в синхронном режиме, она отправляет сообщение непосредственно процедуре окна и ожидает окончания его обработки. В отличие от нее, асинхронная функция `PostMessage()` стандартным образом помещает сообщение в очередь Windows и сразу возвращает управление вызвавшей ее программе, не дожидаясь результатов обработки сообщения.

Синхронная функция `SendMessage()` относится к разряду потенциально опасных, так как если окно-получатель по каким-то причинам не сможет ответить на сообщение, то вызвавшее функцию приложение может даже потерять работоспособность. Поэтому вместо `SendMessage()` зачастую используют `SendMessageTimeout()`. Указанная функция ожидает ответ в течение установленного таймаута и, если за этот промежуток времени ответа получено не было, она гарантированно возвращает управление своей программе.

Исходя из разницы в работе функций, различаются и возвращаемые ими значения. `SendMessage()` *возвращает значение, полученное в результате обработки сообщения,* `PostMessage()` *всего лишь информирует, удалось ей поставить сообщение в очередь (значение* `true`*) или нет (*`false`*).*

Листинг программы

Поскольку мы решили поработать без VCL, то уже самый первый шаг, связанный с созданием нового проекта, будет иметь свои особенности. Нам надо создать пустой проект – проект без формы. Самый простой путь для этого – сгенерировать шаблон для нового консольного приложения. Для этого находим пункт меню File→New...→Other и в выпорхнувшем на ваш рабочий стол окне «New Items» выбираем иконку «Console Application». В появившемся листинге удаляем ненужную директиву компилятора `{$APPTYPE CONSOLE}`. В строке `Uses` вместо модуля `SysUtils` устанавливаем ссылки на файлы `Windows` и `Messages`. Сохраняем проект под именем ex01_01.dpr. Если вы внимательно следовали инструкциям, то в редакторе кода останутся лишь эти строки:

```
program ex01_01;
uses Windows, Messages;

begin

end.
```

Еще раз обращаю внимание читателя на то, что мы явным образом задействовали лишь две библиотеки подпрограмм Delphi. Модуль `Windows` содержит вызовы ключевых функций прикладного интерфейса Windows API, а модуль

Messages хранит описания основных сообщений. За бортом нашего проекта остались такие монстры визуальной библиотеки компонентов Delphi, как Forms, Classes, Graphics, Controls, без которых не обходится ни одно из стандартных приложений VCL.

Теперь подготовим ряд переменных и записей. Для этого перед словом begin набираем следующие строки:

```
const AppName = ' ex01_01'; //константа с именем приложения

var
 Wnd : HWND;                {Дескриптор окна}
 Msg : TMsg;                {Структура для хранения сообщений}
 WindowClass : TWndClass;   {Структура класса окна}
 DC:HDC;                    {Дескриптор контекста графического устройства}
 PAINTSTRUCT : TPaintStruct;{Структура для рисования}
```

Сразу после объявления переменных, но до слова begin описываем оконную процедуру программы.

```
function WindowProc (Window: HWND; Msg: UINT;
                     wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
  WindowProc := 0;
  case Msg of
   {действия при поступлении сообщения WM_DESTROY}
    WM_DESTROY : begin
                 PostQuitMessage (0); {завершение выполнения программы}
                 Exit;
                 end;
    {действия при поступлении сообщения WM_PAINT}
    WM_PAINT   : begin {управление прорисовкой окна}
                 DC:=BeginPaint(Window,PAINTSTRUCT);
                 DrawText(DC,
                          PChar('Hello, Word!'),-1,
                          PAINTSTRUCT.rcPaint,
                          DT_SINGLELINE or DT_CENTER or DT_VCENTER);
                 EndPaint(Window,PAINTSTRUCT);
                 end;
  end;
{обработка сообщений по умолчанию осуществляется функцией Windows DefWindowProc}
  WindowProc := DefWindowProc(Window, Msg, WParam, LParam);
end;
```

Обратите внимание на последние строки оконной процедуры. В ней мы обращаемся за помощью к функции DefWindowProc() — это функция Windows API, осуществляющая обработку сообщений по умолчанию, т. е. всех тех сообщений, которые не были явно описаны в нашем, не постесняемся этого слова, выдающемся коде.

Нам осталось заполнить пустое пространство между ключевыми словами begin и end.

```
Begin
//описание класса создаваемого окна
```

```
with WindowClass do
begin
    Style := cs_HRedraw or cs_VRedraw;
    lpfnWndProc := @WindowProc;
    cbClsExtra := 0;
    cbWndExtra := 0;
    hInstance := 0;
    hIcon := LoadIcon (0, idi_Application);
    hCursor := LoadCursor (0, idc_Arrow);
    hbrBackground := GetStockObject(WHITE_BRUSH);
    lpszMenuName := '';
    lpszClassName := AppName;
  end;
//регистрация класса окна
  If RegisterClass (WindowClass) = 0 then Halt;
//создание окна
  Wnd := CreateWindow(AppName, 'Demo', ws_OverlappedWindow,
                      cw_UseDefault, cw_UseDefault,
                      cw_UseDefault, cw_UseDefault,0,0, HInstance, nil);

  ShowWindow (Wnd, SW_SHOW); //показ окна
  UpdateWindow (Wnd);        //перерисовка окна
//цикл обработки сообщений
  while GetMessage (Msg, 0, 0, 0) do
  begin
    TranslateMessage (Msg);
    DispatchMessage (Msg);
  end;
Halt (Msg.wParam); //Завершение работы программы
end.
```

Теперь простейшее приложение готово – проверяем его работоспособность (рис. 1.1). Предвижу возмущение читателя: «Столько работы для того, чтобы создать элементарный проект!» Предлагаю взглянуть на это с другой стороны. Например, размер полученного исполняемого файла минимален – не более 16 Кбайт! Создание аналогичной программы на Delphi с применением VCL действительно занимает не более минуты нашего драгоценного времени, но из-за этого размер exe-файла составит не менее 300 Кбайт, вот такая

Рис. 1.1. Окно простейшей программы

арифметика! Но это далеко не все преимущества нашего проекта. В сравнении со стандартным приложением Delphi только что написанная программа менее требовательна к оперативной памяти, задействует вдвое меньшее число объектов GDI, обладает большей производительностью. Подтверждением моих слов может стать рис. 1.2, на котором представлен экранный снимок диспетчера задач Windows. Диспетчер задач контролирует перечисленные выше показатели проекта, написанного на Windows API, и проекта, построенного при поддержке VCL. Процесс ex01_01.exe по всем параметрам превосходит своего неповоротливого стандартного собрата Project1.exe.

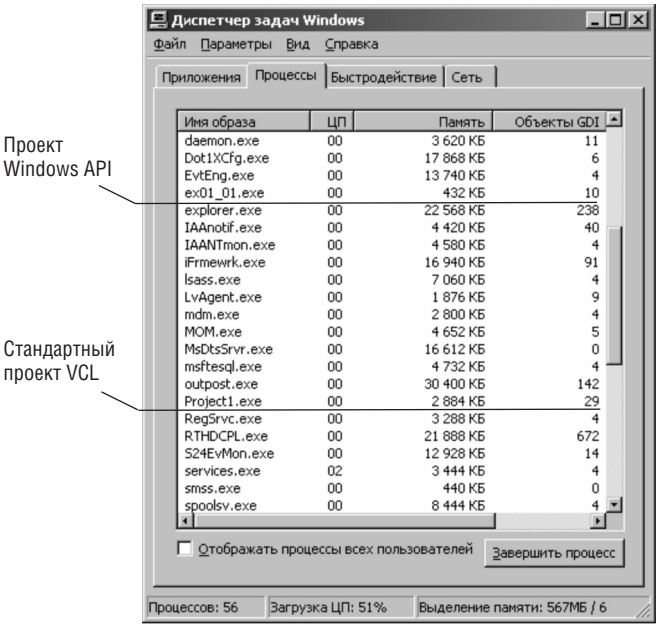


Рис. 1.2. Экранный снимок диспетчера задач

Дочерние окна

Для уяснения еще некоторых особенностей программирования на Windows API предлагаю продолжить работу над проектом. Несколько дополнив наш код, мы получим опыт размещения на окне элемента управления – кнопки.

1. В разделе объявления переменных добавьте дескриптор будущей кнопки `btn`:

```
var Wnd, Btn : HWnd;
```

2. Сразу после создания экземпляра окна `Wnd` создайте экземпляр кнопки:

```
Btn:=CreateWindow('BUTTON',
                  'Закрыть',
```

```
                    WS_CHILDWINDOW,
                    10,
                    10,
                    75,
                    22,
                    Wnd, //определяем дескриптор владельца кнопки
                    0,
                    HInstance,
                    nil);
```

3. Покажите экземпляр кнопки:

```
ShowWindow(Wnd, SW_SHOW);
UpdateWindow(Wnd);
ShowWindow(BTN, SW_SHOW); //показ экземпляра кнопки
```

4. Дополните Case-селектор оконной процедуры еще одним сообщением:

```
CASE MESSAGE OF
. .
  {Поступление сообщения WM_COMMAND}
  WM_COMMAND: BEGIN
       if lParam=Btn then {если источник – наша кнопка}
       begin
         if MessageBox(WND,
            PChar('Закрыть окно?'),
            PChar('Подтвердите'),
            MB_OKCANCEL+MB_ICONINFORMATION)=idOK
        then SendMessage(Wnd,WM_DESTROY,0,0); {сами себе отправляем WM_DESTROY}
       END;
END;
```

Субклассирование

Читатель, внимательно изучивший предыдущий материал, уяснил порядок обслуживания сообщений в Windows. Когда операционная система получает сообщение для какого-то из окон, она выясняет адрес оконной процедуры окна-получателя и вызывает ее, передавая окну-получателю все содержание сообщения. В теории все просто. Но это поверхностное впечатление, как всегда на практике возникают проблемы. Допустим, что наше приложение содержит цепочку окон, в которой одно окно принадлежит другому. Например, программная логика программы предполагает размещение на главном окне проекта двух групп переключателей. В таком случае при формировании интерфейса проекта разумно применить две группирующие панели с принадлежащими им кнопками (рис. 1.3).

Для воплощения нашей идеи на практике целесообразно воспользоваться сообщением WM_CREATE, вызываемым сразу после создания главного окна приложения. В рамках обработки сообщения создаем панели группировки (описываемые дескрипторами grBox1 и grBox2) и шесть переключателей (rBtn1... rBtn6), причем первые три кнопки принадлежат первой, а вторые три – второй

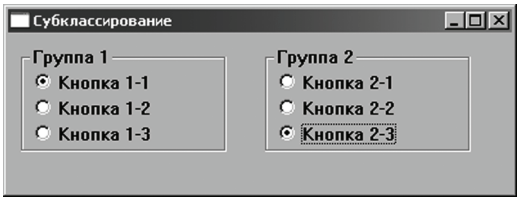


Рис. 1.3. Внешний вид окна с двумя группами переключателей

панели. Ниже представлен фрагмент листинга главной оконной процедуры, претворяющий в жизнь нашу идею.

```
var
  grBox1, grBox2, {дескрипторы панелей группировки}
  rBtn1,rBtn2,rBtn3, rBtn4,rBtn5,rBtn6 : HWnd; // дескрипторы кнопок
//…
function WindowProc (Window: HWND; Msg: UINT;
                     wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
  case Msg of
  WM_CREATE :
  begin
//=============== Группа переключателей 1 ==================
{создание группирующей панели № 1}
  grBox1:= CreateWindow('BUTTON', 'Группа 1',
            WS_CHILD OR WS_VISIBLE OR BS_GROUPBOX,
            10, 10, 160, 82, Window, 0, HInstance, nil);
{первая кнопка, принадлежит группирующей панели № 1}
  rBtn1 := CreateWindow('BUTTON','Кнопка 1-1',
            WS_CHILD OR WS_VISIBLE OR BS_AUTORADIOBUTTON OR BS_NOTIFY
            OR WS_GROUP, 10, 20, 120, 16, grBox1, 0, HInstance, nil);
//… аналогичный код для кнопок rBtn2 и rBtn3
//=============== Группа переключателей 2 ==================
  grBox2:= CreateWindow('BUTTON','Группа 2',
            WS_CHILD OR WS_VISIBLE OR BS_GROUPBOX,
            200, 10, 160, 82, Window, 0, HInstance, nil);
  rBtn4 := CreateWindow('BUTTON','Кнопка 2-1',
            WS_CHILD OR WS_VISIBLE OR BS_AUTORADIOBUTTON OR BS_NOTIFY
            OR WS_GROUP, 10, 20, 120, 16, grBox2, 0, HInstance, nil);
//… аналогичный код для кнопок rBtn5 и rBtn6

  end;
//… обработка остальных сообщений
end;
```

После старта проекта мы столкнемся с определенной сложностью – мы не сможем оперативно реагировать на действия пользователя, выбирающего ту или иную кнопку. Все наши попытки «отловить» сообщения кнопок в разделе секции WM_COMMAND главной оконной процедуры обречены на провал. Это объясняется тем, что владельцами кнопок выступают панели группировки,

а не главное окно приложения. Поэтому все исходящие от кнопок сигналы Windows станет направлять в адрес оконных процедур панелей grBox2 и grBox2, а не в процедуру WindowProc. Проблема в том, что в проекте нет оконных процедур для панелей группировок кнопок-переключателей...

Выйти из создавшегося затруднительного положения нам поможет *субклассирование* (*subclassing*). Идея субклассирования заключается в замене оконной процедуры по умолчанию другой – нестандартной.

Хакерский прием подмены оконной процедуры без особых проблем осуществляется функцией:

```
function SetWindowLong(Wnd: HWND; Index: Integer; NewLong: Longint):
Longint;
```

Нам лишь требуется передать в функцию три параметра. Первый из них (параметр Wnd) – дескриптор окна, которое мы собираемся обмануть, в нашем случае это панели группировки. Второй параметр Index уточняет задачу функции, например, для смены адреса оконной процедуры в него направляется константа GWL_WNDPROC. Кроме фокуса с субклассированием функция владеет еще рядом силовых приемов, например она позволяет изменить стиль окна (табл. 1.4). Третий параметр NewLong определяет подменяемое значение. В случае успешного выполнения функция возвращает предыдущее 32-разрядное значение (если речь идет о субклассировании, то мы получим адрес старой оконной процедуры). Это значение стоит запомнить в отдельной переменной, оно может пригодиться для возврата окна к родной оконной процедуре, но это уже сфера задач функции CallWindowProc(), о которой мы поговорим позднее.

Таблица 1.4. Значения параметра Index функции SetWindowLong( )

Константа	Описание
GWL_EXSTYLE	Установка расширенного стиля окна (совместимого с процедурой CreateWindowEx).
GWL_STYLE	Установка нового стиля окна.
GWL_WNDPROC	Установка адреса процедуры окна.
GWL_HINSTANCE	Установка нового дескриптора приложения.
GWL_ID	Смена идентификатора окна.
GWL_USERDATA	Установка данных, связанных с окном.
DWL_DLGPROC	Новый адрес процедуры диалогового окна.
DWL_MSGRESULT	Изменение значения, возвращаемого диалоговым окном.
DWL_USER	Работа с дополнительной информацией.

Допустим, что в нашем проекте уже объявлена нестандартная оконная процедура для панели группировки GroupBoxProc(), тогда ее вызов будет выглядеть следующим образом:

```
{переменная для хранения адреса стандартной оконной процедуры}
```

```
Var OldProc : TFNWndProc;
//…
OldProc:=TFNWndProc(SetWindowLong(grBox1, GWL_WNDPROC, Longint(@GroupBoxProc)));
```

Последнюю строку примера можно разместить сразу после создания окна панели группировки `grBox1`. С этого момента все сообщения, предназначенные для панели, станут направляться в адрес оконной процедуры `GroupBoxProc()`.

По умолчанию Delphi использует классическое для языка Паскаль соглашение о вызовах `pascal`. *Такой подход недопустим для функций Windows API, они работают в рамках стандартного соглашения* `stdcall`. *Одно из коренных отличий между соглашениями – порядок обработки параметров функций. Стандартное соглашение обслуживает параметры справа налево, а соглашение Паскаль наоборот – слева направо.*

К структуре подставной оконной процедуры предъявляется ряд жестких требований. Перечень ее параметров обязан полностью соответствовать списку параметров обычной оконной процедуры. Также нельзя путать последовательность аргументов. На первом месте всегда идет дескриптор окна, затем сообщение, старший и младший параметр. После описания заголовка процедуры необходимо указать, что функция опирается на стандартное соглашение о вызовах `stdcall`.

```
function GroupBoxProc (Window: HWND; Msg: UINT;
                       wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
  case Msg of
    WM_COMMAND : begin
                 //…
                 end;
  end;
{вызов стандартной процедуры окна}
CallWindowProc(OldProc, Window, Msg, wParam, lParam);
end;
```

Как правило, код нестандартной оконной процедуры завершается обращением к функции:

```
function CallWindowProc(PrevWndFunc: TFNWndProc; Wnd: HWND;
                        Msg: UINT; wParam: WPARAM; lParam: LPARAM): LRESULT;
```

Функция позволяет вернуть обработку сообщений окна в стандартное русло – в «объятия» родной оконной процедуры. Указатель на эту процедуру передается в первый параметр функции, в нашем примере это не что иное, как возвращаемое функцией `SetWindowLong()` значение.

РЕЗЮМЕ

Прикладной программный интерфейс Windows API позволяет создавать эффективные программные продукты, по ряду параметров существенно превосходящие приложения, построенные в классическом для Delphi стиле.

Именно поэтому большая часть примеров в этой книге написана на API. Такой подход позволит нам не только глубже понять особенности графического механизма Windows, но и получить необходимые для настоящего профессионала навыки системного программирования.

Вместе с тем опора на Windows API ни в коем случае не отвергает применение в наших проектах визуальной библиотеки компонентов Delphi. Ведь корни VCL также опираются на функции API, а компоненты инкапсулирует основной функционал Windows. Но это не обычное подражание. Инженеры Borland значительно упростили и, одновременно, усовершенствовали процесс программирования. Хорошим тому подтверждением стал графический класс `TCanvas`, не только впитавший в себя ключевые функции Windows GDI, но и приобретший преимущества классического класса VCL. Именно поэтому программист Delphi должен уметь разумно сочетать в своих программах преимущества Windows API и VCL.

2

Контекст графического устройства

Операционная система Windows своим успехом во многом обязана великолепной инженерной идее, реализованной в виде *контекста графического устройства* (*Device Context*). Контекст представляет собой центр визуального интерфейса системы – в его руках хранятся все нити управления выводом информации на экран или на печатающее устройство.

В чем его заслуги перед Windows? Вряд ли получится ответить на этот вопрос в двух словах. Начнем с того, что контекст позволяет отделить аппаратную часть компьютера (устройство графического вывода) от программной. Именно поэтому все функционирующие под управлением операционной системы компоненты чудесным образом превращаются в аппаратно-независимые. Забота о преобразовании команд GDI на вывод графических примитивов в команды, понятные аппаратной части компьютера, возложена на плечи операционной системы и драйвера устройства, но ни в коем случае не на программиста. В итоге существенно облегчается труд разработчика как прикладного, так и системного программного обеспечения, ведь теперь при написании кода практически нет нужды задумываться о технических особенностях построения того или иного устройства графического вывода. Различия между устройствами сглаживаются до такой степени, что с точки зрения программирования процесс вывода изображения в окне приложения практически не отличается от печати рисунка на принтере – графическое устройство превращается в некую весьма удобную для нас абстракцию. Вторая примечательная особенность контекста устройства в том, что он содержит данные о текущих настройках графического устройства, на котором мы планируем что-то отобразить. Например, в перечень атрибутов контекста входят описание логической кисти и пера, шрифта, режим отображения, палитра и многое другое. Поэтому в простейшем случае при вызове функций GDI достаточно указать координаты, определяющие место вывода графических примитивов, а их цвет, толщину пера, стиль, используемый шрифт, высоту символов и многое другое функции GDI выяснят у контекста.

С точки зрения Windows контекст графического устройства представляет собой иерархию взаимосвязанных структур и объектов, находящихся в адресном пространстве пользовательского режима приложения и в адресном пространстве ядра системы. С одной стороны контекст взаимодействует с драйвером графического устройства, с другой – с нашими приложениями (рис. 2.1).

Разработчики Windows не раскрывают нам тайну реализации контекста и оставляют за собой право изменять его в новых операционных системах, но вне зависимости от версии ОС в роли основной нити, связывающей наши программы и устройства графического вывода, выступает *дескриптор контекста устройства*:

```
HDC = type LongWord;
```

Технически это 32-разрядный идентификатор, позволяющий программисту отправлять команды GDI в адрес конкретного устройства вывода, будь это окно программы, графический элемент управления, принтер, плоттер, факс или виртуальное устройство.

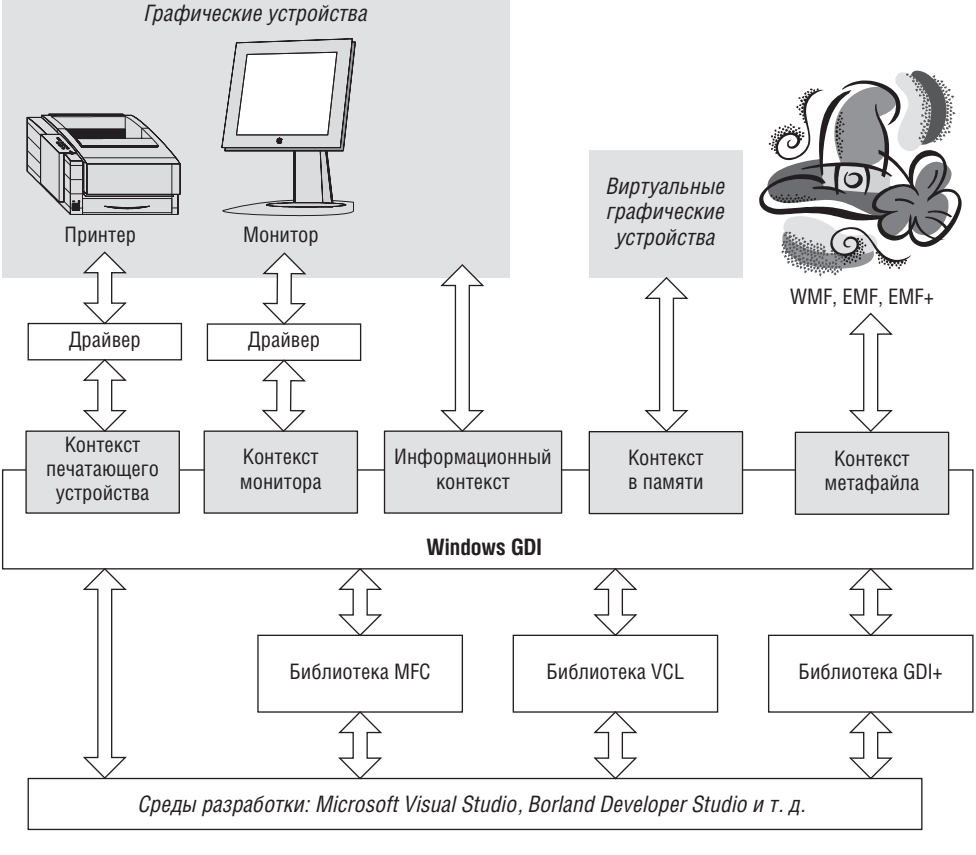


Рис. 2.1. Место контекста устройства в GDI

Дескриптор контекста графического устройства является обязательным параметром для подавляющего большинства функций GDI.

Документация SDK выделяет четыре типа контекстов: контекст дисплея, контекст печатающего устройства, контекст в памяти и информационный контекст. Если проявить принципиальность, то стоит отметить факт существования пятого типа контекста – метафайлового (см. рис. 2.1), но о нем мы поговорим далее в главе 18, посвященной расширенным метафайлам Windows.

Контекст графического устройства, как конгломерат сложных структур и объектов, содержит некоторый перечень полей, определяющих наиболее общие характеристики контекста. Список весьма значителен, но в нем можно выделить атрибуты, хранящие следующие данные:

- регионы контекста, определяющие порядок отсечения;
- установленная система координат и текущий режим отображения;
- параметры двумерных аффинных преобразований в мировых координатах;
- параметры заливки областей;
- параметры черчения линий;
- цветовые характеристики графического устройства;
- выбранный шрифт и особенности вывода текста.
- сложение цветов или, говоря научным языком, установленная бинарная растровая операция.
- параметры траектории.

Список можно продолжить, но пока переведем дыхание, тем более, что со всеми ключевыми атрибутами контекста мы повстречаемся в следующих главах книги. А теперь настал черед знакомства со способами получения дескриптора контекста устройства.

Основная задача контекста устройства заключается в предоставлении доступа к графической поверхности, состоящей из двумерного массива пикселов. Как правило, каждый из элементов массива доступен как для операций чтения, так и для записи.

Дескриптор контекста для экрана и печатающих устройств

Итак, с точки зрения программиста контекст графического устройства представляет собой мост между любым аппаратным графическим устройством и нацеленным на рисование исходным кодом. В роли графического устройства способен выступить не только экран монитора, но принтеры, плоттеры и другие аппаратные устройства, связанные с выводом или вводом графической информации. В системе Windows они такие же полноправные обладатели контекста. Для доступа к дескриптору контекста такого рода требуется познакомиться с функцией `CreateDC()`:

```
Function CreateDC(Driver, Device, Output : PAnsiChar;
                  Init : pDeviceModeA ):HDC;
```

Здесь `Driver` – имя драйвера устройства, в качестве аргумента могут выступать только три значения: строка «DISPLAY» (для работы с экраном), строка «WINSPOOL» (для работы с печатающим устройством) и неопределенный указатель `nil`. Второй параметр `Device` – это системное имя устройства. В современных 32- и 64-разрядных версиях Windows параметр `Output` не применяется. Параметр `Init` представляет собой указатель на структуру `PDeviceModeA`, благодаря которой мы сможем инициализировать контекст. Все четыре аргумента одновременно практически никогда не используются. Например, для обращения к контексту дисплея достаточно только первого параметра.

```
Procedure SCREEN_PAINT;
begin
   DC:=CreateDC(PChar('DISPLAY'),nil, nil, nil);
     SetBkMode(DC,TRANSPARENT);
     TextOut(DC, 10, 10, PChar('Контекст экрана'), 15);
   DeleteDC(DC);
end;
```

Благодаря `CreateDC()` мы получаем право рисовать на поверхности рабочего стола Windows, что может пригодиться приложениям, выполняющим свои задачи в фоновом режиме или в свернутом виде, например для оповещения пользователя о получении почты или окончании записи компакт-диска.

Для доступа к принтеру необходим второй параметр функции:

```
DC:= CreateDC(nil, PChar('Canon CP660PS'),nil, nil);
```

Функция `CreateDC()` превосходно подходит для обращения к контекстам печатающих устройств или экрана компьютера, но она не позволит программисту работать с контекстом окна приложения. Для решения этой задачи предназначены другие функции.

По умолчанию при создании любой контекст устанавливается в текстовый режим отображения (`MM_TEXT`). Позднее, в главе 11 «Системы координат и режимы отображения», мы обсудим все применяемые в Windows режимы отображения. А пока достаточно знать, что в текстовом режиме:

– начало координат располагается в левом верхнем углу;

– горизонтальная ось X направлена слева направо, а вертикальная ось Y – сверху вниз;

– в качестве единицы измерения выступает пиксел.

Контекст окна приложения

Если программа научена откликаться на сообщение `WM_PAINT` (знакомой нам по предыдущей главе команды на перерисовку окна), то наиболее рациональным способом получения дескриптора контекста устройства станет обращение к функции `BeginPaint()`. Эту функцию мы уже активно использовали в примерах первой главы, но незаслуженно мало времени уделили ее описанию.

Настало время устранить эту оплошность, и, как всегда, начнем с обсуждения синтаксической конструкции:

```
Function BeginPaint(WND : HWND; var PaintStruct : TPaintStruct) : HDC;
```

Первый параметр функции – описатель окна, дескриптор контекста которого мы и рассчитываем получить. Второй параметр – ссылка на заполняемую автоматически информационную структуру TPaintStruct.

```
type TPaintStruct = packed record
    HDC: HDC;         //дескриптор контекста
    fErase: BOOL;     //true - требует очистки области перед началом рисования
    rcPaint: TRect;   //координаты прямоугольной области для рисования
    fRestore: BOOL;   //не используется
    fIncUpdate: BOOL;//не используется
    rgbReserved: array[0..31] of Byte; //не используется
  end;
```

В записи используются только три первых поля, из них для нас наиболее полезно поле rcPaint, в котором мы обнаружим клиентские координаты обслуживаемого окна. Вторая половина полей записи TPaintStruct оставлена без дела, по крайней мере, это утверждает документация SDK.

В случае успешного выполнения функция порадует программиста дескриптором контекста, в противном случае нам будет возвращен неопределенный указатель.

Функция BeginPaint() *может быть вызвана только в блоке кода, предназначенного для обслуживания сообщения перерисовки окна* WM_PAINT. *Во всех остальных случаях обращение к этой функции бессмысленно.*

Обращение к функции BeginPaint() всегда должно дополняться вызовом парной ей функции EndPaint():

```
Function EndPaint(WND : HWND; const PaintStruct : TPaintStruct) :
boolean;
```

Функция устанавливает атрибуты контекста в исходное состояние и сигнализирует о завершении обработки события перерисовки окна, поэтому обычно ее вызов осуществляется в последних строках кода обслуживания сообщения WM_PAINT.

Функции EndPaint() *и* ReleaseDC() *не уничтожают контекст графического устройства, так как последний принадлежит окну и может еще неоднократно понадобиться. Поэтому эти функции скорее нужны для демонстрации хорошего стиля программирования.*

Сразу после получения дескриптора контекста мы можем приступать к выводу графической информации. Ниже представлен уже знакомый по первой главе книги листинг оконной процедуры, контролирующей сообщения, направляемые операционной системой к окну Window через параметр Msg. В примере мы ограничились обслуживанием только сообщения WM_PAINT.

```
var PAINTSTRUCT : TPaintStruct;
    DC : HDC;
//…
function WindowProc(Wnd: HWND; Msg: UINT;
                    wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
  case Msg of
       WM_PAINT   : begin {обработка сообщения прорисовки окна}
                    DC:=BeginPaint(Wnd, PAINTSTRUCT);
                    TextOut(DC, 10, 10, PChar('Hello, Word!'), 12);
                    EndPaint(Wnd, PAINTSTRUCT);
                    end;
//… другие сообщения
  end;
  WindowProc := DefWindowProc (Wnd, Msg, WParam, LParam);
end;
```

С целью сокращения размера листингов и повышения их читабельности в подавляющем большинстве остальных примеров основной графический код будет помещаться не в раздел обработки сообщения WM_PAINT, а в отдельную процедуру GDI_PAINT(). В таком случае предыдущий листинг станет выглядеть примерно так:

```
Procedure GDI_PAINT(DC:HDC);
begin
TextOut(DC, 10, 10, PChar('Hello, Word!'), 12);
end;
```

Процедура GDI_PAINT() вызывается из оконной процедуры:

```
function WindowProc (Wnd: HWND; Msg: UINT; wParam: WPARAM; lParam: LPARAM)
                     : LongInt; stdcall;
begin
  case Msg of
       WM_PAINT   : begin
                    DC:=BeginPaint(Wnd, PAINTSTRUCT);
         {!!!}      GDI_PAINT(DC); //ВЫЗОВ ПРОЦЕДУРЫ ГРАФИЧЕСКОГО ВЫВОДА
                    EndPaint(Wnd, PAINTSTRUCT);
                    end;
//… другие сообщения
  end;
  WindowProc := DefWindowProc (Window, Msg, WParam, LParam);
end;
```

Предусмотрен еще один способ, позволяющий «ухватить за хвост» дескриптор контекста клиентской части окна. Для этого требуется вызвать функцию GetDC():

```
Function GetDC(WND : HWND) : HDC;
```

Единственное, в чем нуждается функция, – указание дескриптора окна, на поверхности которого мы намерены что-то нарисовать. Функция GetDC() умеет работать только с контекстом клиентской области окна, а что делать,

когда требуется обратиться к на первый взгляд недоступной области заголовка или рамки окна? Для этого существует функция

Win GDI
```
Function GetWindowDC(WND : HWND) : HDC;
```

Функция GetWindowDC() предоставит в наше распоряжение 100% поверхности окна приложения, включая заголовок, область, отведенную под главное меню, и статусную панель. Таким образом можно изрядно озадачить пользователя, в качестве шутки предложив ему мягко говоря нестандартный заголовок окна:

```
Procedure Humorize(Window: HWND);
var DC:HDC;
    Rct:TRect;
begin
DC:=GetWindowDC(Window);

SetRect(Rct,5,5,20,20); DrawFrameControl(DC,rct,DFC_CAPTION,DFCS_CAPTIONMIN);
OffsetRect(Rct,15,0);   DrawFrameControl(DC,Rct,DFC_CAPTION,DFCS_CAPTIONMAX);
OffsetRect(Rct,17,0);   DrawFrameControl(DC,Rct,DFC_CAPTION,DFCS_CAPTIONCLOSE);

ReleaseDC(Window,DC);
end;
```

Розыгрыш заключается в том, что мы сымитировали в левой части заголовка окна кнопки управления приложением (рис. 2.2). Для реализации столь коварного плана нам потребовались услуги только что изученной функции GetWindowDC (которая предоставила в наше распоряжение всю поверхность окна) и функции DrawFrameControl, чья специализация заключается в рисовании основных элементов управления Windows (см. главу 7 «Простейшие геометрические фигуры и заливка областей»).

Можно еще немножко «поколдовать» и оживить псевдокнопки, но это тема для другой книги про Delphi и Windows... Мы же вновь вернемся к дескриптору контекста устройства. В операционной системе предусмотрена и усложненная версия функции GetDC().

Win GDI
```
Function GetDCEx(WND : HWND; RgnClip : HRGN; flags : Cardinal) : HDC;
```

Ключевая особенность функции в том, что она предоставляет программисту больше возможностей по управлению контекстом в тех случаях, когда поверхность окна перекрывается другими окнами. Для работы с GetDCEx() во втором параметре RgnClip надо указать на регион отсечения, который может быть объединен с видимой областью окна. В третий параметр передаются флаги из табл. 2.1, они определяют особенности создаваемого контекста.

Рис. 2.2. Графический вывод на заголовке окна

Если вместо указателя на окно в функции `GetDC()` *или* `GetDCEx()` *направить нулевое значение, то они возвратят дескриптор контекста экрана.*

Таблица 2.1. Возможные значения флагов функции GetDCEx( )

Значение	Описание
`DCX_WINDOW`	Возвращает дескриптор контекста, соответствующий не клиентскому, а полному прямоугольнику окна.
`DCX_CACHE`	Возвратит контекст из кэша диспетчера окон. Эта операция выполнится, даже если класс окна создавался с флагами `CS_OWNDC` и `CS_CLASSDC`.
`DCX_PARENTCLIP`	Использует видимый регион родительского окна. Даже если родительское окно создавалось (функцией `CreateWindow`) на основе стилей `WS_CLIPCHILDREN` и `CS_PARENTDC`, эти ограничения не учитываются. За начальную точку принимается левый верхний угол окна, описываемый параметром `Wnd`.
`DCX_CLIPSIBLINGS`	Исключает видимые области всех родственных окон (окон этого же процесса), расположенных выше окна, идентифицированного параметром `Wnd`.
`DCX_CLIPCHILDREN`	Исключает видимые области всех дочерних окон, расположенных под окном `Wnd`.
`DCX_NORESETATTRS`	Не сбрасывает атрибуты контекста в значения по умолчанию после освобождения дескриптора.
`DCX_LOCKWINDOWUPDATE`	Обычно флаг устанавливается в тех случаях, когда надо перерисовывать содержимое окна во время его перемещения по экрану. Если этот флаг включен, то игнорируется запрет на перерисовку окна, установленный с помощью функции `LockWindowUpdate()`.
`DCX_EXCLUDERGN`	Область отсечения `RgnClip` исключается из видимой области, описываемой дескриптором контекста устройства `DC`.
`DCX_INTERSECTRGN`	Область отсечения `RgnClip` объединяется с видимой областью, описываемой дескриптором контекста устройства `DC`. Возвращается контекст нового региона.
`DCX_INTERSECTUPDATE`	Видимая область контекста пересекается с областью обновления.
`DCX_VALIDATE`	Объявит содержимое окна действительным. При совместном использовании с флагом `DCX_INTERSECTUPDATE` поведение функции становится тождественным функции `BeginPaint()`.

Для получения дескриптора контекста графического устройства в проектах `Delphi`, *построенных на основе визуальной библиотеки компонентов, проще всего обращаться к свойству* `Handle` *холста (класс* `TCanvas`*), например:*

```
DC:=Form1.Canvas.Handle;
```

Если описываемый дескриптором `DC` контекст графического устройства связан с окном, то можно выяснить и дескриптор этого окна. Для этого предназначена функция:

```
function WindowFromDC(DC: HDC): HWND;
```

Кстати, используя термин «окно», не следует замыкаться лишь на окне приложения. Читатель, внимательно изучивший предшествующую главу, уже знает, что в Windows под окном следует понимать любой объект, появившийся на свет посредством вызова функции `CreateWindow()`, а это может быть и кнопка, и полоса прокрутки, и строка ввода, и многие другие элементы управления. Получив дескриптор окна, мы в большинстве случаев приобретаем право рисовать на его поверхности. Таким образом, в наше распоряжение попадает достаточно широкий спектр элементов управления с палитры компонентов Delphi, которые строили из себя недотрог. Единственное условие – элементы должны состоять в родстве с классом `TWinControl` (в литературе, посвященной Delphi, потомков указанного класса часто называют оконными элементами управления). Советую вспомнить это замечание, когда вдруг потребуется осуществить вывод на поверхности компонентов Delphi, не обладающих холстом (свойство `Canvas`).

Освобождение дескриптора контекста

К сожалению, визитной карточкой современного программиста среднего уровня, особенно взращенного в «райских садах» ультрановых сред разработки (старательно собирающих и уничтожающих брошенные объекты – технология сборки мусора), считается пренебрежительное отношение к такому ключевому ресурсу операционной системы, как виртуальная память. Можно долго полемизировать о причинах и следствиях такого стиля программирования. Не станем останавливаться на этом, скажу лишь то, что с ними нам не по пути. Мы постараемся самым строжайшим образом следить за тем, чтобы выходящий из-под нашего «пера» код был максимально рационален.

Говоря о рациональности, я имею в виду то, что каждый вновь созданный экземпляр контекста не прочь «подкрепиться», проглотив пару-другую лишних килобайт памяти. Кроме того, Windows не в состоянии поддерживать неограниченное число дескрипторов контекста устройства. Не берусь утверждать, что несколько оставленных без внимания дескрипторов нарушат работоспособность приложения или, тем более, операционной системы, но ведь утечки памяти нам вроде бы тоже ни к чему. Поэтому, завершив работу с контекстом, обязательно отпустите на волю захваченный ресурс. Для этого потребуется обращение к функции:

```
Function DeleteDC(DC : HDC): integer;
```

Здесь необходимо указать дескриптор контекста. Если контекст был успешно освобожден, то функция вернет единичку, в противном случае – ноль.

Еще одна функция просто сигнализирует системе, что принадлежащий определенному окну контекст графического устройства в ближайшее время нам не потребуется:

```
Function ReleaseDC(WND  :  HWND; DC : HDC): integer;
```

Эту функция не столько из разряда обязательных к применению, сколько демонстрационных. Она показывает качественный стиль программирования – в данном случае уважение к системным ресурсам. Другими словами, обращение к `ReleaseDC()` является признаком хорошего тона, мы говорим Windows, что благодарим за предоставленный ресурс GDI и больше в нем не нуждаемся. Практически аналогичное назначение у рассмотренной несколькими страницами ранее функции `EndPaint()`. Обе функции лишь сбрасывают атрибуты контекста в состояние по умолчанию. Скорее всего, роль `ReleaseDC()` и `EndPaint()` значительно возрастает в ситуации, когда один и тот же контекст разделяется между двумя и более потоками или процессами. Представляете, какая чехарда начнется, когда два приложения попытаются одновременно что-нибудь нарисовать в одном и том же окне? В этом случае указанные выше функции могут снимать признак захвата контекста, уменьшать счетчик захватов, переключать семафор, мьютекс, триггер или выполнять другую схожую задачу по управлению доступом к общему ресурсу. Но это не более чем догадка, так как в SDK по этому поводу ничего не сказано.

Функция `DeleteDC()` *обычно применяется в паре с* `CreateDC()`. *Функция* `ReleaseDC()` – *напарник для* `GetDC()`, `GetDCEx()` *и* `GetWindowDC()`.

Контекст в памяти (совместимый контекст)

Вывод простейшего изображения на графическом устройстве сопряжен с определенными ресурсными затратами, ведь в этот процесс кроме функций GDI вовлекаются посредники: электронные устройства (например, дисплей) и программное обеспечение – драйверы. А если речь идет о сложных графических операциях, требующих значительных ресурсов, компьютер может не успеть прорисовать результирующее изображение за отведенный такт времени. В этом случае экран может мерцать или даже отобразить незавершенный рисунок, что наверняка ввергнет пользователя в глубокую печаль и прежде времени погонит в магазин за новым видеоадаптером. Одним из способов борьбы с подобным недостатком является формирование картинки в памяти компьютера и отправка ее на видеокарту только после завершения ее создания.

Для имитации реального контекста в памяти создается совместимый контекст:

```
Function CreateCompatibleDC(DC : HDC) : HDC;
```

Функция требует передачи единственного аргумента – действительно существующего контекста `DC`. Это необходимо для того, чтобы создаваемый в памяти контекст приобрел все черты реального устройства. Или, как говорят настоящие программисты, контекст в памяти стал *совместимым* с реальным контекстом, например с контекстом окна приложения.

Если вместо DC *в функцию* CreateCompatibleDC() *мы передадим ноль, то функция возвратит дескриптор, совместимый с установками текущего экрана.*

В случае успешного выполнения функция вернет описатель одного-единственного монохромного пиксела. Чтобы по-настоящему развернуться, настоящей творческой личности таких площадей вряд ли хватит. Посему для расширения доступной поверхности в контекст выбирается битовый образ. Это действие устанавливает размер поверхности для рисования в соответствии с размерами растрового рисунка.

```
var DDB:HBitmap;
    memDC:HDC;
begin
memDC:=CreateCompatibleDC(GetDC(0));  //совместимый контекст в памяти
{создание битового образа размером 100×100}
DDB:=CreateCompatibleBitmap(memDC,100,100);
SelectObject(memDC,DDB);
//рисуем в памяти
//…
//переносим данные из памяти на поверхность реального контекста
BitBlt(DC,0,0,100,100, memDC,0,0,SRCCOPY);
DeleteDC(memDC);   //освобождаем ресурсы контекста
DeleteObject(DDB); //удаляем битовый образ
end;
```

Не забывайте пример работы с совместимым контекстом и битовым образом; он нам в дальнейшем очень пригодится. В терминах специалистов по графическому выводу этот прием называется двойной буферизацией, он позволяет устранить эффект мерцания при выводе сложных рисунков.

Доступ к стандартным объектам контекста

В момент создания контекста устройства с ним связывается минимальный ряд объектов GDI, необходимых для его нормального функционирования. В перечень джентльменского набора GDI входят логические перья, кисти, шрифты и палитра. Для обращения к любому из этих объектов стоит воспользоваться услугами функции:

Win GDI
```
Function GetStockObject(fnObject : Integer) : HGDIOBJ;
```

Единственный параметр функции требует передачи константы требуемого логического объекта (табл. 2.2).

Представленные в табл. 2.2. объекты принадлежат всей системе в целом, а не только конкретному экземпляру контекста. Нам позволено пользоваться услугами системных перьев, кистей, шрифтов, но мы не сможем ни модифицировать (за редким исключением), ни, тем более, уничтожить системный объект GDI.

Таблица 2.2. Названия констант стандартных объектов контекста устройства

Константа	Описание
BLACK_BRUSH	Черная кисть.
DKGRAY_BRUSH	Темно-серая кисть.
DC_BRUSH	Сплошная кисть, позволяющая изменять свой цвет посредством вызова функции SetDCBrushColor().
GRAY_BRUSH	Серая кисть.
HOLLOW_BRUSH, NULL_BRUSH	Пустая кисть.
LTGRAY_BRUSH	Светло-серая кисть.
WHITE_BRUSH	Белая кисть.
BLACK_PEN	Черное перо.
DC_PEN	Сплошное перо, способное изменять свой цвет. Для изменения цвета предназначена функция SetDCPenColor().
WHITE_PEN	Белое перо.
ANSI_FIXED_FONT	Моноширинный системный шрифт.
ANSI_VAR_FONT	Системный шрифт пропорциональной ширины.
DEVICE_DEFAULT_FONT	Шрифт устройства по умолчанию. (Только Windows NT/2000/XP.)
DEFAULT_GUI_FONT	Шрифт по умолчанию пользовательского графического интерфейса. Применяется для вывода текстовых сообщений в диалоговых окнах и меню.
OEM_FIXED_FONT	Моноширинный шрифт OEM.
SYSTEM_FONT	Системный шрифт, используемый по умолчанию для вывода текстовых сообщений, надписей на кнопках и элементах меню.
SYSTEM_FIXED_FONT	Устаревшая константа моноширинного шрифта. Оставлена для обеспечения совместимости с 16-разрядными версиями Windows.
DEFAULT_PALETTE	Логическая палитра. По умолчанию указанная палитра содержит только статические цвета системной палитры.

Доступ к текущему объекту контекста

Одновременно в системе могут существовать сотни различных объектов GDI, однако в одно и то же время конкретный контекст способен пользоваться услугами только ограниченного числа помощников. Если ему требуется провести несколько линий различными перьями или закрасить несколько областей различными кистями, контекст должен для каждой линии выбирать новое перо, а для каждой области свою кисть. Для обращения к текущему объекту контекста стоит воспользоваться функцией:

```
Function GetCurrentObject(DC : HDC;  ObjectType : UINT) : HGDIOBJ;
```

Кроме дескриптора контекста устройства функция потребует указать тип востребованного объекта – параметр `ObjectType` (табл. 2.3).

Таблица 2.3. Возможные значения параметра ObjectType

Константа	Описание
`OBJ_BITMAP`	Возвращает выбранный в контекст текущий растровый рисунок.
`OBJ_BRUSH`	Текущая кисть.
`OBJ_COLORSPACE`	Текущее цветовое пространство.
`OBJ_FONT`	Текущий шрифт.
`OBJ_PAL`	Текущая логическая палитра.
`OBJ_PEN`	Текущее перо.

Информационный контекст

Если задача приложения заключается в исследовании атрибутов контекста устройства, то вместо тяжеловесной функции `CreateDC()` обычно применяют ее упрощенный аналог:

```
Function CreateIC(Driver, Device, Output : PAnsiChar;
                  Init : pDeviceModeA):HDC;
```

Перечень параметров `CreateIC()` аналогичен параметрам `CreateDC()`. Отличие в том, что функция `CreateIC()` не предназначена для рисования, и все попытки отправки графических команд в адрес полученного с ее помощью дескриптора не приведут к появлению изображения. Причина в том, что возвращаемый ею дескриптор контекста устройства значительно упрощен и его единственное назначение – информирование нас об атрибутах контекста. После завершения работы с информационным контекстом не забудьте вызвать функцию `DeleteDC()` и освободить захваченные ресурсы.

Технические характеристики контекста, функция GetDeviceCaps()

В Windows имеется универсальная функция, способная проинформировать нас о технических характеристиках графических устройств вывода информации. Ее имя – `GetDeviceCaps()`.

```
Function GetDeviceCaps(DC : HDC; Index : Integer) : Integer;
```

Наша новая знакомая способна рассказать о трех десятках ключевых характеристиках устройства вывода (принтера, плоттера или экрана дисплея). Для этого в `GetDeviceCaps()` передается дескриптор контекста устройства `DC`. Второй параметр `Index` представляет собой один из почти трех десятков идентификаторов (см. таблицу в приложении к главе). В зависимости от его значения функция возвратит то или иное описание устройства.

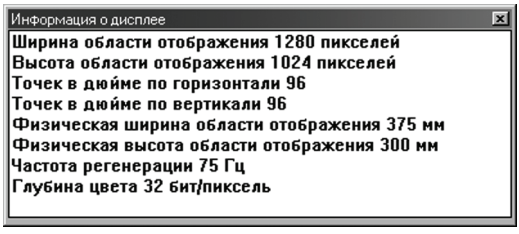


Рис. 2.3. Информация о дисплее

Ниже предложен вариант процедуры, информирующей пользователя об основных характеристиках устройства вывода. Результаты выполнения возвращаются в список, идентифицируемый дескриптором LISTBOX. На рис. 2.3 представлены результаты исследования 19-дюймового жидкокристаллического дисплея.

```
Procedure GetDCInfo(DeviceName : string; LISTBOX: HWnd);
var IC:HDC;
    S:String;
    i : integer;
begin
IC:=CreateIC(PChar(DeviceName),Nil, Nil, Nil);

i:=GetDeviceCaps(IC,HORZRES);
S:=Format('Ширина области отображения %d пикселов',[i]);
SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));

i:=GetDeviceCaps(IC,VERTRES);
S:=Format('Высота области отображения %d пикселов',[i]);
SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));

//ДРУГИЕ ПОКАЗАТЕЛИ

DeleteDC(IC);
end;
```

Восстановление состояния контекста

Предоставляющий столь потрясающие возможности контекст устройства – весьма тонкий и хрупкий механизм, работоспособность которого зависит от огромного множества внутренних и внешних факторов. Поэтому при работе с контекстом надо руководствоваться правилом – не навреди! Как ни удивительно, но в этом плане нашим первым помощником станет сам контекст. Он обладает парой функций, позволяющих сохранять текущее состояние контекста и, если возникнет такая необходимость, возвращаться к этому состоянию.

Первую часть работы выполняет функция:

Win GDI

```
Function SaveDC(DC:HDC) : Integer;
```

Она попросит GDI запомнить текущий графический режим и принадлежащие контексту объекты (перо, кисть, регион отсечения и т. д.). Функция возвратит идентификатор архивной копии контекста. Он нам пригодится в момент, когда потребуется вернуть контекст к предыдущему состоянию. Идентификатор передается во второй параметр функции RestoreDC:

Win GDI
```
Function RestoreDC(DC : HDC; SavedDC   : Integer) : Boolean;
```

Восстановив контекст, функция RestoreDC() очистит сохраненные данные, идентификатор SavedDC станет бесполезен.

Процедура сохранения контекста может вызываться многократно. Но при этом надо помнить, что информация о состоянии заносится в стек, организованный по принципу LIFO (Last In First Out – «последним пришел – первым вышел»). Поэтому, получив команду на восстановление контекста, данные о котором лежат внизу стека, функция RestoreDC() *очистит всю информацию, находящуюся в стеке выше.*

Управление объектами GDI

В момент создания контекста устройства его атрибуты заполняются значениями по умолчанию. Например, в поле, отвечающее за характеристики текущего пера, помещаются данные о пере, способном провести черную сплошную линию толщиной в один пиксел. Вместе с тем в запасе Windows GDI имеется достаточно обширный перечень функций, позволяющих создавать различные объекты GDI – в частности, перьев разнообразных цветов и стилей. Для того чтобы контекст устройства смог воспользоваться услугами вновь созданного объекта, требуется выбрать его в контекст устройства. Для этой цели предназначена функция:

Win GDI
```
function SelectObject(DC:HDC; GDIObj: HGDIOBJ):HGDIOBJ;
```

Здесь DC – дескриптор контекста устройства, GDIObj – дескриптор объекта GDI. С помощью функции SelectObject() в контекст устройства выбираются перья, кисти, шрифты, регионы и растры. В случае успешного выполнения функция возвращает дескриптор предыдущего (только что замененного) объекта. Исключение составляет случай, когда заменяется регион. Тогда SelectObject() проинформирует нас о виде региона: SIMPLEREGION, COMPLEXREGION или NULLREGION. Если в момент замещения возникла исключительная ситуация, мы получим ошибку с кодом HGDI_ERROR.

Вновь напомню беспощадное правило программирования для Windows GDI. Оно очень простое – не забывать уничтожать объекты GDI, в услугах которых мы больше не нуждаемся. Поэтому, завершив работу с объектом GDIObj, без сожаления отдавайте его на растерзание функции:

Win GDI
```
function DeleteObject(GDIObj: HGDIOBJ): BOOL;
```

Но перед вызовом функции обязательно убедитесь, чтобы перо, кисть или другой утилизируемый объект не были выбраны в контекст: иначе DeleteOb-

ject() не справится со своими обязанностями. Признаком того, что объект успешно отправлен к праотцам, станет значение true, возвращенное функцией.

Исследование объектов GDI

Кроме создания различных объектов GDI программист зачастую сталкивается и с обратной задачей – получением данных об объектах GDI. С ролью исследователей лучше всего справятся функции GetObjectType() и GetObject(). Первая функция предназначена для выяснения типа объекта.

Win GDI
```
function GetObjectType(GDIOBJ: HGDIOBJ): DWORD;
```

Для «вычисления личности» объекта функции достаточно познакомиться с его дескриптором. Возвращаемые функцией константы представлены в табл. 2.4.

Таблица 2.4. Константы объектов GDI

Константа	Описание
OBJ_BITMAP	Битовый образ
OBJ_BRUSH	Кисть
OBJ_COLORSPACE	Цветовое пространство
OBJ_DC	Контекст устройства
OBJ_ENHMETADC	Расширенный метафайловый контекст
OBJ_ENHMETAFILE	Расширенный метафайл
OBJ_EXTPEN	Перо с расширенными возможностями
OBJ_FONT	Шрифт
OBJ_MEMDC	Контекст в памяти
OBJ_METAFILE	Метафайл старой версии
OBJ_METADC	Метафайловый контекст старой версии
OBJ_PAL	Палитра
OBJ_PEN	Перо
OBJ_REGION	Регион

Вторая функция препарирует объект на более серьезном уровне.

Win GDI
```
function GetObject(GDIOBJ: HGDIOBJ; Count: Integer; pBuff: Pointer):
Integer;
```

Здесь первый параметр рассчитывает получить дескриптор изучаемого объекта. Параметр pBuff – указатель на буфер в памяти, в который функция загрузит Count байт информации об объекте GDI. Размер буфера и способ доступа к полученным данным определяется типом объекта. Для пера это структура TLogPen, для кисти – TLogBrush, битовый образ описывается структурой TBitmap, шрифт – структурой TLogFont.

Разработка хранителя экрана

Наших знаний о контексте графического устройства почти достаточно, чтобы реализовать простейший хранитель экрана, осталось лишь уточнить некоторые нюансы – и за клавиатуру!

Порывшись в недрах папки Windows, вы обнаружите несколько файлов с расширением «scr». Это и есть программы хранителей, еще раз подчеркну слово «программы», ведь файл хранителя – это по сути тот же исполняемый файл «exe», с той лишь разницей, что он был откомпилирован с директивой:

```
{$D SCRNSAVE Название хранителя}
```

Указанная команда размещается сразу за строкой `uses` в головном модуле проекта Delphi – dpr. Если вы не сильны в английском языке, то вправе использовать русскоязычное или любое другое название хранителя, компилятор возражать не станет.

Сердцем будущего хранителя экрана станет процедура `ScreenSaver()`. В ней вы не обнаружите команд на прорисовку трехмерных моделей, переливающихся всеми цветами радуги. Для начала достаточно и того, что наша программа станет выводить дежурную фразу «Hello, World!» в случайных местах экрана, а более сложные картинки мы научимся создавать позднее.

```
Procedure ScreenSaver(Window: HWND; DesctopRect:TRect);
var DC:HDC;
    Clr:COLORREF;
const S='Hello, World!';
begin
DC:=GetWindowDC(Window); //получаем дескриптор всей поверхности окна
FillRect(DC,DesctopRect,GetStockObject(BLACK_BRUSH));
Clr:=RGB(RANDOM(255),RANDOM(255),RANDOM(255)); //случайный цвет текста
Clr:=GetNearestColor(DC,Clr); //учитываем возможность неполноцветного режима
SetTextColor(DC,Clr);         //устанавливаем цвет текста
SetBkMode(DC,TRANSPARENT);    //прозрачный фон
TextOut(DC,Random(DesctopRect.Right), {выводим надпись в случайной позиции}
           Random(DesctopRect.Bottom),S,Length(s));
ReleaseDC(Window,DC); //освобождаем дескриптор
end;
```

Листинг достаточно хорошо снабжен комментариями, поэтому на описании его работы мы останавливаться не станем, а лишь отметим тот факт, что процедура должна вызываться не по факту поступления в адрес окна сообщения `WM_PAINT`, а из отдельного потока или по команде таймера. Мы направимся по второму пути и объявим константу – идентификатор таймера:

```
const TimerID = 1;
```

Жизнь в таймер мы вдохнем в оконной процедуре в момент поступления сообщения `WM_CREATE`.

При реализации оконной процедуры будущего хранителя экрана учтем некоторые правила хорошего тона для подобных программ. Во-первых, при

выводе хранителя необходимо скрывать указатель мыши. Во-вторых, завершая работу с хранителем, возвращаем указатель на место. В-третьих, сигналом на прекращение работы хранителя станут поступление в адрес окна сообщений о нажатии клавиш, кнопок мыши или перемещении указателя мыши.

```
function WindowProc (Window: HWND; Msg: UINT;
                     wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
  case Msg of
   WM_CREATE :
      begin
      SetTimer(Window, TimerId, 1000, nil); //создание таймера
      ShowCursor(FALSE); //прячем курсор
      end;
       //нажатие любой клавиши и движение мыши - выход из хранителя
       WM_KEYDOWN, WM_LBUTTONDOWN, WM_MBUTTONDOWN, WM_RBUTTONDOWN,WM_MOUSEMOVE :
                  PostQuitMessage (0);
       WM_TIMER : begin
                   GetWindowRect(Window,DesctopRect);//размеры окна
                   ScreenSaver(Window,DesctopRect);  //процедура перерисовки
                  end;
       WM_DESTROY : begin
                   KillTimer(Window, TimerId);  //"убиваем" таймер
                   ShowCursor(TRUE);            //восстанавливаем курсор
                   PostQuitMessage (0);
                   Exit;
                  end;
  end;
  WindowProc := DefWindowProc (Window, Msg, WParam, LParam);
end;
```

Мы практически подошли к финалу. Нам осталось лишь позаботиться, чтобы после создания окно принимало максимальные размеры. С этой целью функция `ShowWindow()` вызывается с константой `SW_MAXIMIZE`. Протестировав полученный проект, поменяйте расширение исполняемого файла приложения с «exe» на «scr» и скопируйте его в системный каталог Windows.

В заключение отмечу, что представленный код, хотя и вполне работоспособен, но представляет собой лишь каркас для полноценного хранителя экрана. Настоящий хранитель для Windows (кроме собственно оберегания дисплея, с чем наша программа успешно справляется) должен уметь отображать окно настроек опций хранителя и выводить окно предварительного просмотра. Для этого хранитель проверяет параметры, с которыми он был вызван. Существует соглашение, что окно настроек опций вызывается при поступлении команды «`/c`*», окно предварительного просмотра – «*`/p`*».*

РЕЗЮМЕ

Контекст графического устройства – это стержень, на котором держится графическая система Windows. Он предоставляет универсальное, независимое от устройства описание поверхности, на которую будет осуществляться графический вывод. Различают четыре основных вида контекста: контекст дисплея, контекст печатающего устройства, информационный контекст и контекст в памяти. Непосредственный доступ к контексту получить невозможно, вместо этого GDI предлагает дескриптор контекста – 32-битный идентификатор. Дескриптор контекста необходим для нормальной жизнедеятельности большей части функций GDI.

Приложение. Функция GetDeviceCaps()

Таблица. Значения параметра Index функции GetDeviceCaps( )

Значение Index	Возвращаемая характеристика	
DRIVERVERSION	Версия драйвера устройства.	
TECHNOLOGY	Технология драйвера. Может принимать одно из следующих значений:	
	DT_PLOTTER	Векторный плоттер.
	DT_RASDISPLAY	Растровый дисплей.
	DT_RASPRINTER	Растровый принтер.
	DT_RASCAMERA	Растровая камера.
	DT_CHARSTREAM	Поток символов.
	DT_METAFILE	Метафайл.
	DT_DISPFILE	Файл дисплея.
HORZSIZE	Физическая ширина области отображения в миллиметрах.	
VERTSIZE	Физическая высота области отображения в миллиметрах	
HORZRES	Ширина области отображения в пикселах.	
VERTRES	Высота области отображения в пикселах.	
LOGPIXELSX	Число пикселов в логическом дюйме по ширине области отображения.	
LOGPIXELSY	Число пикселов в логическом дюйме по высоте.	
BITSPIXEL	Число битов цвета для представления одного пиксела.	
PLANES	Число цветовых плоскостей.	
NUMBRUSHES	Количество кистей устройства.	
NUMPENS	Количество перьев устройства.	
NUMFONTS	Количество шрифтов устройства.	

Значение Index	Возвращаемая характеристика	
NUMCOLORS	Число зарезервированных цветов. Имеет смысл, если Windows работает с глубиной цвета, не превышающей 8 бит. В противном случае функция возвратит −1.	
ASPECTX	Относительная ширина пиксела устройства по горизонтали.	
ASPECTY	Относительная ширина пиксела устройства по вертикали.	
ASPECTXY	Относительная ширина пиксела устройства по диагонали.	
CLIPCAPS	Указывает способность отсечения устройства. Если устройство умеет отсекать вывод в прямоугольную область, то функция возвратит 1 (значение CP_RECTANGLE). Если устройство не умеет отсекать вывод, то в результате мы получим 0 (значение CP_NONE).	
SIZEPALETTE	Число элементов в системной палитре.	Индекс может применяться только при условии, что драйвер устройства устанавливает бит RC_PALETTE в индексе RASTERCAPS.
NUMRESERVED	Число резервных элементов в системной палитре.	
COLORRES	Разрешающая способность устройства по цвету (бит на пиксел).	
PHYSICALWIDTH	Ширина физической страницы в пикселах.	Для печатающих устройств.
PHYSICALHEIGHT	Высота физической страницы в пикселах.	
PHYSICALOFFSETX	Расстояние от левого края физической страницы до левой границы области печати в единицах устройства.	
PHYSICALOFFSETY	Расстояние от верхнего края физической страницы до верхней границы области печати в единицах устройства.	
VREFRESH	Частота регенерации кадров в секунду (Гц) устройства отображения. Индекс доступен в Windows версии NT/2000/2003/XP.	
SCALINGFACTORX	Коэффициент масштабирования для оси X принтера.	
SCALINGFACTORY	Коэффициент масштабирования для оси Y принтера.	
BLTALIGNMENT	Желательное выравнивание вывода по горизонтали, выраженное целым числом пикселов. Индекс доступен в Windows версии NT/2000/2003/XP.	
SHADEBLENDCAPS	Возможности устройства по затемнению и микшированию.	
	SB_CONST_ALPHA	Анализирует поле SourceConstantAlpha (альфа-прозрачность) структуры BLENDFUNCTION функции AlphaBlend().
	SB_GRAD_RECT	Возможность создавать прямоугольник с градиентной заливкой.
	SB_GRAD_TRI	Возможность создавать треугольник с градиентной заливкой.
	SB_NONE	Устройство не поддерживает ни одну из возможностей.

Значение Index	Возвращаемая характеристика	
	SB_PIXEL_ALPHA	Способность выполнять альфа-микширование.
	SB_PREMULT_ALPHA	Способность выполнять альфа-микширование после предварительной подготовки.
RASTERCAPS	Растровые возможности устройства. Функция возвращает комбинацию флагов:	
	RC_BANDING	Необходимо разбиение на полосы.
	RC_BITBLT	Устройство способно передавать растровые изображения.
	RC_BITMAP64	Устройство способно передавать растровые изображения размером более 64 Кбайт.
	RC_DI_BITMAP	Поддерживает функции SetDIBits() и GetDIBits().
	RC_DIBTODEV	Поддерживает функцию SetDIBitsToDevice().
	RC_FLOODFILL	Способно выполнять заливку области.
	RC_PALETTE	Устройство действует на основе палитры.
	RC_SCALING	Поддержка масштабирования.
	RC_STRETCHBLT	Поддерживает функцию StretchBlt().
	RC_STRETCHDIB	Поддерживает функцию StretchDIBits().
CURVECAPS	Поддержка вывода кривых. Функция возвращает комбинацию флагов:	
	CC_NONE	Устройство не поддерживает вывод кривых.
	CC_CHORD	Устройство способно выводить хорды.
	CC_CIRCLES	Поддержка вывода окружностей.
	CC_ELLIPSES	Способность рисования эллипсов.
	CC_INTERIORS	Умение закрашивать внутренние области.
	CC_PIE	Способность выводить сектора круга.
	CC_ROUNDRECT	Вывод прямоугольников со скругленными углами.
	CC_STYLED	Рисование стилизованных рамок.
	CC_WIDE	Рисование утолщенных рамок.
	CC_WIDESTYLED	Поддержка вывода утолщенных стилизованных рамок.
LINECAPS	Возможность вывода линий устройством. Функция возвращает комбинацию флагов:	
	LC_NONE	Устройство не способно выводить линии.
	LC_INTERIORS	Устройство способно рисовать внутренние области.
	LC_MARKER	Вывод маркеров.
	LC_POLYLINE	Вывод ломаных линий.

Значение Index	Возвращаемая характеристика	
	LC_POLYMARKER	Рисование нескольких маркеров.
	LC_STYLED	Вывод стилизованных линий.
	LC_WIDE	Вывод утолщенных линий.
	LC_WIDESTYLED	Поддержка рисования утолщенных стилизованных линий.
POLYGONALCAPS	Возможность вывода устройством многоугольников. Функция возвращает комбинацию флагов:	
	PC_NONE	Устройство не способно выводить многоугольники.
	PC_INTERIORS	Устройство способно рисовать внутренние области.
	PC_POLYGON	Поддержка вывода многоугольников.
	PC_RECTANGLE	Умение рисовать прямоугольники.
	PC_SCANLINE	Поддержка вывода отдельной строки развертки.
	PC_STYLED	Вывод стилизованных границ.
	PC_WIDE	Вывод утолщенных границ.
	PC_WIDESTYLED	Поддержка утолщенных стилизованных границ.
	PC_WINDPOLYGON	Умение рисовать многоугольник с заполнением с учетом числа кручения.
TEXTCAPS	Возможности устройства по выводу текста. Функция возвращает комбинацию флагов:	
	TC_OP_CHARACTER	Устройство обеспечит точный вывод на уровне отдельного символа.
	TC_OP_STROKE	Устройство обеспечит точный вывод на уровне отдельного штриха.
	TC_CP_STROKE	Устройство обеспечит точную вырезку на уровне отдельного штриха.
	TC_CR_90	Поворот символов на угол 90 градусов.
	TC_CR_ANY	Поворот символов на любой угол.
	TC_SF_X_YINDEP	Отдельное масштабирование по вертикали и горизонтали.
	TC_SA_DOUBLE	Способность удваивать символы при масштабировании.
	TC_SA_INTEGER	Для масштабирования используются только целочисленные коэффициенты.
	TC_SA_CONTIN	Для масштабирования могут использоваться любые коэффициенты.
	TC_EA_DOUBLE	Вывод символов с двойной толщиной.
	TC_IA_ABLE	Поддержка курсива.

Значение Index	Возвращаемая характеристика	
	TC_UA_ABLE	Поддержка подчеркивания символа.
	TC_SO_ABLE	Поддержка перечеркивания символа.
	TC_RA_ABLE	Поддержка растровых шрифтов.
	TC_VA_ABLE	Поддержка векторных шрифтов.
	TC_RESERVED	Зарезервировано.
	TC_SCROLLBLT	Устройство не поддерживает прокрутку с применением передачи двоичных блоков.
COLORMGMTCAPS	Способность устройства по управлению цветом. Индекс доступен в Windows версии NT/2000/2003/XP. Функция возвращает комбинацию флагов:	
	CM_CMYK_COLOR	Устройство воспринимает профиль цвета ICC цветового пространства CMYK.
	CM_DEVICE_ICM	Устройство способно согласовывать цвета изображения либо в драйвере, либо непосредственно в устройстве.
	CM_GAMMA_RAMP	Устройство поддерживает функции GetDeviceGammaRamp() и SetDeviceGammaRamp().
	CM_NONE	Устройство не поддерживает согласование цветов ICM (*Image color matching*).

3

Управление устройствами видеовывода

За недолгий период существования электронно-вычислительных машин способ общения человека и компьютера претерпел ряд принципиальных изменений. Совершенствовались как способы ввода, так и способы отображения информации. Первые ЭВМ вообще не имели дисплеев, вывод данных осуществлялся на печатающие устройства и (в лучшем случае) на световые табло. Из-за ограниченности возможностей столь архаичных устройств трудно говорить о высокой степени эффективности диалога оператора и старой вычислительной машины. Но научно-технический прогресс не стоял на месте. Это можно подтвердить небольшим экскурсом в историю развития видеоадаптеров, тем более что это может оказаться весьма полезной прелюдией к материалу, посвященному вопросу управления устройствами видеовывода.

Первый персональный компьютер фирмы IBM появился на свет в 1981 году, на его вооружении стоял простейший *монохромный адаптер* (*Monochrome Display Adapter*, MDA), нацеленный исключительно на текстовый режим работы. Все, что умел MDA, так это вывести на черно-белый экран два с половиной десятка строк по 80 символов в каждой. Столь скромные возможности адаптера, хотя и компенсировались относительно невысокой ценой, не могли удовлетворить запросы пользователей. Поэтому спустя пару лет IBM разработала первый цветной видеоадаптер, широко известный по аббревиатуре *CGA* (*Color Graphic Adapter*). Основное достоинство CGA заключалось в возможности работы не только в текстовом, но и в графическом режиме. Адаптер поддерживал разрешение 320×200 пикселов с глубиной цвета 2 бита на пиксел (4 цвета). Хотя цветовые возможности CGA были, мягко говоря, слабоваты, но главное было сделать первый шаг. К середине 1984 года на рынке появился цветной *адаптер с расширенными возможностями* (*Enhanced Graphic Adapter*, EGA). Новый адаптер взирал на своего предшественника свысока! Еще бы, он обладал 16-цветовым режимом работы с размером растра 640×350 пикселов! Следующей знаменательной вехой на пути совершенствования графической составляющей персонального компьютера стал видеоадаптер *VGA* (*Video Graphics Array*). Он работал с 256 цветами при разрешении 320×200, кроме

того, VGA был способен обслуживать растр размерностью 640×480 пикселов для 16 цветов. Пусть не ухмыляются владельцы современных мобильных телефонов с полноцветными дисплеями – для 1987 года это существенное достижение. Дальнейшее наращивание возможностей видеоадаптеров было направлено на повышение количества цветов и разрешающей способности. На рынке появились десятки различных фирм, предлагающих все более и более совершенные модели устройств. Развитие технологий постепенно привело к появлению высококачественных устройств отображения. В первую очередь – это построенные на основе электронно-лучевых трубок мониторы и жидкокристаллические дисплеи. Они характеризуются высоким качеством передачи цветов, малым размером пиксела и большой разрешающей способностью, так что на сегодняшний день мало кого можно удивить разрешением 1600×1200 с 32-битным цветом.

Сбор информации об устройствах видеовывода

Любая нацеленная на работу с современной графикой программа просто обязана знать, с какими устройствами отображения ей придется сотрудничать. Стоит ли говорить, что нет смысла выводить цветную картинку на монохромный экран или пытаться втиснуть огромный рисунок в видеоадаптер с доисторическим форматом буфера. Именно поэтому одной из наиболее часто применяемых функций, способных собрать данные о доступных в системе устройствах видеовывода, является `EnumDisplayDevices()`:

```
Function EnumDisplayDevices(Device : Pointer; DevNum : Cardinal;
                            var DD : TDisplay_DeviceA;
                             Flags : Cardinal) : boolean;
```

Здесь `Device` – указатель на название конкретного устройства. Зачастую вместо имени в параметр направляют неопределенный указатель `nil`. В таком случае функцией собираются данные о доступных видеоадаптерах компьютера, индексы которых передаются в параметр `DevNum`. Для построения списка всех адаптеров необходимо организовать цикл вызовов `EnumDisplayDevices()`, последовательно (начиная с нуля) увеличивая значение `DevNum` на единицу. Как только устройства закончатся, функция `EnumDisplayDevices` отрапортует об этом значением `false`. Результаты трудов функции мы обнаружим в третьем по счету параметре `DD`, это структура `TDisplay_DeviceA`. Последний аргумент функции программистами Microsoft оставлен для будущих поколений разработчиков и пока не используется, поэтому `EnumDisplayDevices()` не обидится, получив в параметр `Flags` нулевое значение.

```
type TDisplay_DeviceA = packed record
  cb: DWORD;                                  //размер структуры
  DeviceName: array[0..31] of AnsiChar;       //имя устройства
  DeviceString: array[0..127] of AnsiChar;  //описание устройства
  StateFlags: DWORD;   //флаги состояния (см. табл. 3.1)
end;
```

Таблица 3.1. Флаги состояния записи TDisplay_DeviceA

Константа	Значение	Описание
DISPLAY_DEVICE_ATTACHED_TO_DESKTOP	h01	Устройство отображает рабочий стол Windows.
DISPLAY_DEVICE_MIRRORING_DRIVER	h02	Виртуальное устройство.
DISPLAY_DEVICE_MODESPRUNED		Набор видеорежимов устройства превышает возможности оконечного устройства. Другими словами, дисплей не способен поддержать все режимы видеоадаптера.
DISPLAY_DEVICE_PRIMARY_DEVICE	h04	Основное устройство видеовывода.
DISPLAY_DEVICE_REMOVABLE	h08	Съемное устройство, которое не может являться основным дисплеем.
DISPLAY_DEVICE_VGA_COMPATIBLE	h10	VGA-совместимое устройство.

Перед первым обращением к функции EnumDisplayDevices() *необходимо проинициализировать предназначенную для сбора данных структуру* TDisplay_DeviceA*; информация об этом помещается в поле* cb.

Ниже представлен пример, демонстрирующий способ сбора данных о видеоадаптерах в поле со списком. Информация собирается в оконной процедуре в рамках обработки сообщения WM_CREATE.

```
var  DD:TDisplay_DeviceA; DevNum :cardinal;
//…
function WindowProc(Window: HWND; Msg: UINT;
                    wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
case Msg of
WM_CREATE : begin
       ComboBox:= CreateWindow('COMBOBOX','',
                        WS_CHILD OR WS_BORDER OR BS_DROPDOWNLIST,
                        1, 1, 80, 17, Window, 0, HInstance, nil);
       DevNum:=0;
       DD.cb:=SizeOf(TDisplay_DeviceA);  // !!! Обязательная инициализация!!!
       while EnumDisplayDevices(nil,DevNum,DD,0)=true do
        begin
          SendMessage(ComboBox,CB_ADDSTRING,0,
            INTEGER(pChar(DD.DeviceName+' => '+DD.DeviceString+#0)));
            INC(DevNum);
        end;
//…
end;
```

В результате выполнения кода комбинированный список с дескриптором ComboBox заполняется описаниями доступных адаптеров.

Умение получить информацию о находящихся в нашем распоряжении устройствах видеовывода представляет собой необходимое, но далеко не достаточное условие для обеспечения корректной работы программы с графикой, ведь кроме названий адаптеров ей необходимо знать их технические характеристики. Поэтому нам потребуется помощь функции, занимающейся профессиональным сбором информации о поддерживаемых устройством режимах работы:

```
Function EnumDisplaySettings(DeviceName : pANSIChar;
         ModeNum  : Cardinal; var DevMode: TDeviceMode) : boolean;
```

Параметр DeviceName представляет собой указатель на заканчивающуюся нулем строку с именем устройства отображения (первичный адаптер обычно называется «\\.\DISPLAY1»). Впрочем, если логика программы предполагает работу только с основным видеоадаптером, то вместо имени в параметр передавайте nil. Кроме того, DeviceName вправе ссылаться на строку, полученную при посредничестве функции EnumDisplayDevices(). Второй по счету параметр ModeNum определяет порядковый номер интересующего нас режима. Для сбора всех имеющихся режимов отображения инициируйте ModeNum нулевым значением и (осуществляя пошаговое приращение) вызывайте функцию до тех пор, пока она не возвратит false. Как показывает опыт, для создания полной картины потребуется 3–4 сотни обращений к EnumDisplaySettings(). Вся информация о режиме накапливается в параметре DevMode, описываемом структурой TDeviceMode. В Microsoft не поленились и снабдили TDeviceMode более чем тремя десятками полей (многие из них предназначены для описания принтера), поэтому все их мы перечислять не станем, а отметим лишь наиболее полезные для нас.

```
type TDeviceMode= packed record
 dmDeviceName: array[0..CCHDEVICENAME - 1] of AnsiChar; //имя устройства
 dmBitsPerPel: DWORD;  //глубина цветов в битах
 dmPelsWidth: DWORD;   //ширина
 dmPelsHeight: DWORD;  //высота
 dmDisplayFrequency: DWORD;  //частота кадровой развертки
 dmPosition: DWORD; //позиция устройства в случае работы с несколькими мониторами
 //… другие поля записи
end;
```

Ниже предложен пример, демонстрирующий способ построения списка режимов.

```
Procedure DisplaySettingsList(DeviceName : pANSIChar);
var ModeNum :cardinal;
    DM:TDeviceMode;
    s:string;
begin
ModeNum :=0;
While EnumDisplaySettings(DeviceName,ModeNum ,DM)=true do
begin
 s:=Format('%d x %d пикс. %d бит %d Гц',
        [DM.dmPelsWidth,DM.dmPelsHeight,DM.dmBitsPerPel,DM.dmDisplayFrequency]);
```

```
    SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
    INC(ModeNum);
  end;
  end;
```

На этот раз информация передается в список с дескриптором ListBox. Для этого в адрес списка отправляется сообщение SendMessage() с текстовыми данными о разрешении, глубине цвета и частоте обновления.

Изменение настроек экрана

Имея поистине бесценные знания о поддерживаемых видеоадаптером режимах работы, мы без особых проблем сможем перевести его в наиболее подходящее для нашей программы состояние. Для управления режимом работы основного устройства видеовывода компьютера предназначена функция:

```
Function ChangeDisplaySettings(DevMode: TDeviceMode;
                               Flags : Cardinal) : Integer;
```

Функция изменяет настройки дисплея, переводя его в указанный (с помощью первого аргумента) режим работы. Значения описываются в формате структуры TDeviceMode, уже знакомой нам по функции EnumDisplaySettings(). Особо важен параметр Flags, он представляет собой комбинацию флагов, представленных в табл. 3.2.

Таблица 3.2. Значения флагов функции ChangeDisplaySettings( )

Значение	Описание
0	Дисплей динамически переходит в новый графический режим.
CDS_FULLSCREEN	Временный режим. После завершения работы приложения, сменившего графический режим, система постарается возвратиться в исходное состояние.
CDS_GLOBAL	Новые установки будут применены для всех пользователей компьютера. Флаг должен использоваться совместно с флагом CDS_UPDATEREGISTRY.
CDS_NORESET	Данные о предлагаемом графическом режиме сохраняются в реестре, но экран продолжит работу в старом режиме до перезагрузки. Флаг используется только совместно с CDS_UPDATEREGISTRY.
CDS_RESET	Новый режим применяется принудительно.
CDS_SET_PRIMARY	Выбранное устройство назначается первичным.
CDS_TEST	Система только тестирует корректность графического режима без изменений работы видеоадаптера.
CDS_UPDATEREGISTRY	Предложенный режим применяется немедленно с записью пользовательских настроек в реестре Windows.

Если работа функции завершилась корректно, то она возвратит значение DISP_CHANGE_SUCCESSFUL. Например, следующие строки кода позволяют протестировать корректность будущих настроек экрана.

```
if  ChangeDisplaySettings (dm,CDS_TEST)=DISP_CHANGE_SUCCESSFUL
 then EnableWindow(btnApplay,True)
 else EnableWindow(btnApplay,False);
```

Предложенный код переведет кнопку с дескриптором `btnApplay` в пассивное состояние, если выбранный графический режим не поддерживается устройством видеовывода.

Предположим (рис. 3.1), что наш список `ListBox` заполнен всеми доступными для устройства режимами (в соответствии с предыдущим примером работы с функцией `EnumDisplaySettings`). В этом случае код, позволяющий изменять текущие настройки видеоадаптера, может выглядеть примерно так:

```
Procedure ApplayDisplaySettings;
var ModeNum :cardinal;
    DM:TDeviceMode;
begin
ModeNum:=SendMessage(ListBox,LB_GETCURSEL,0,0); //узнаем индекс выбранного режима
EnumDisplaySettings(pAnsiChar(CB_Text),ModeNum ,DM);
  s:=Format('Установить режим:'+#13+'%d x %d пикс. %d бит %d Гц ?',
        [DM.dmPelsWidth,DM.dmPelsHeight, DM.dmBitsPerPel, DM.dmDisplayFrequency]);
  If MessageBox(Wnd,PChar(s),
      PChar('Применение режима'),MB_YESNO+MB_ICONQUESTION)=idYes then
      ChangeDisplaySettings(DM,0);
end;
```

Хорошо воспитанное приложение должно уметь реагировать на изменение графического режима, даже инициированного другой программой. Признаком перестройки режима работы станет поступление сообщения `WM_DISPLAYCHANGE`.

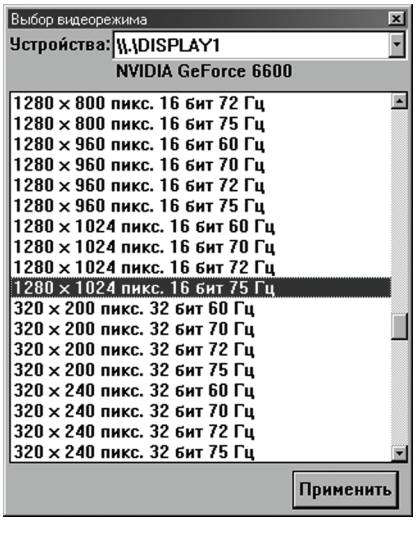


Рис. 3.1. Окно программы

Поддержка телевизионного выхода

Начиная с Windows 98 (ОС, потенциально способная поддерживать работу с несколькими устройствами отображения), появилась усовершенствованная версия функции ChangeDisplaySettings():

```
function ChangeDisplaySettingsEx(DeviceName: PChar;
                  var DevMode: TDeviceMode;
                  WND: HWND; Flags: DWORD; lParam: Pointer): Longint;
```

В первую очередь функция нацелена на работу с телевизором. У функции заметно добавилось параметров. Теперь для адресации видеоустройства в параметре DeviceName можно указать его имя. Если мы оставим первый параметр без внимания (указатель в никуда nil), функция станет работать с первичным устройством. Назначение аргумента DevMode осталось прежним – определение параметров графического режима. Параметр WND не используется и удовлетворится нулевым значением. Функция ChangeDisplaySettingsEx() поддерживает те же самые флаги, что и ее прототип, плюс флаг CDS_VIDEOPARAMETERS. Наличие дополнительного флага свидетельствует о том, что последний аргумент функции (lParam) содержит указатель на специализированную структуру, содержащую информацию о дополнительных параметрах устройства. В Windows API эта структура объявлена под названием _VIDEOPARAMETERS (заголовочный файл tvout.h). В Delphi (к сожалению) аналог _VIDEOPARAMETERS не описан; предполагаю, что в Borland решили, что если программист добрался до функции ChangeDisplaySettingsEx(), то ему не составит особого труда описать свою собственную версию структуры, например под названием TVideoParameters.

Информацию о _VIDEOPARAMETERS *вы найдете в SDK в разделе, посвященном структурам, работающим с контекстом устройства (*Device Context Structures*).*

РЕЗЮМЕ

Мы познакомились с функциями, предназначенными для сбора информации о зарегистрированных в системе устройствах видеовывода, и доступными для них режимами работы. Кроме того, в главе рассмотрены способы управления такими важнейшими параметрами видеоустройств, как разрешающая способность, глубина цветов и частота кадровой развертки.

4

Кисти

Кисть (англ. *brush*) – один из ключевых объектов Windows GDI. Она предназначена не столько для рисования (это зона ответственности главного героя следующей главы – пера), сколько для закраски замкнутых областей. Изначальные возможности кисти небогаты, у нее всего три атрибута – цвет, штриховка и узор. Впрочем, функционал кисти определяется стоящими перед ней задачами, а ведь «малярные работы» вряд ли можно назвать рисованием. Это легко доказать – вспомните детские книжки-раскраски, в которых мы закрашивали контуры заранее подготовленных рисунков всевозможными цветами.

Физически кисть создается из битового образа. В момент заливки области образ кисти размножается во всех направлениях до тех пор, пока не накроет всю поверхность.

В GDI различают два типа объектов кистей: стандартные и логические. Первые автоматически создаются Windows. Представители стандартных кистей на нас смотрят свысока – они находятся под защитой операционной системы, так что программисту не позволяется их удалять или модифицировать (за небольшим исключением). В противоположность своим высокомерным товаркам логические кисти готовы разбиться в лепешку, лишь бы порадовать программиста своими возможностями. Такая самоотдача обусловлена тем, что у логической кисти нет дипломатической неприкосновенности и экземпляр кисти находятся в полной власти ее создателя. В момент формирования объекта программист может наделить экземпляр кисти требуемыми параметрами, а после того как надобность в кисти отпадает, безжалостно ее уничтожить.

В Windows GDI кисти идентифицируются дескрипторами типа `HBRUSH`.

Стандартные кисти

Стандартные кисти являются заранее подготовленными объектами GDI. Единственное преимущество таких кистей над их логическими коллегами –

в высокой скорости доступа. Приложение не тратит время на создание объекта такого рода, ведь он уже существует и готов к работе. Для обращения к заготовке достаточно воспользоваться функцией `GetStockObject()`.

Функция `GetStockObject()` *специализируется не только на обращении к кистям. Существует около двух десятков других, заранее подготовленных стандартных объектов GDI. В их числе перья, шрифты и палитры.*

На сегодняшний день зарезервировано 8 именованных констант, позволяющих получить дескриптор семи стандартных кистей Windows (см. табл. 4.1).

Таблица 4.1. Константы стандартных кистей

Константа	Значение	Описание
WHITE_BRUSH	0	Белая кисть.
LTGRAY_BRUSH	1	Светло-серая кисть
GRAY_BRUSH	2	Серая кисть.
DKGRAY_BRUSH	3	Темно-серая кисть
BLACK_BRUSH	4	Черная кисть.
HOLLOW_BRUSH NULL_BRUSH	5	Пустая кисть.
DC_BRUSH	18	Кисть, позволяющая настраивать свой цвет. Доступна только в Windows NT/2000/XP.

Внимательный читатель наверняка заметил, что содержимое табл. 4.1 представляет собой выписку из перечня стандартных объектов GDI (см. табл. 2.2).

Стандартные кисти не обладают ни узором, ни штриховкой, они однородные (рис. 4.1). Ниже представлены строки кода, демонстрирующие порядок обращения к стандартным кистям, идентифицируемым значениями 0,1, ..., 5.

```
procedure GDI_Paint(DC:HDC);
VAR Brush  : HBRUSH;
    B:Byte;
    Rct:TRect;
begin
SetRect(Rct,10,10,50,50);
for B:=0 to 5 do
  begin
  Brush:=GetStockObject(B);
```

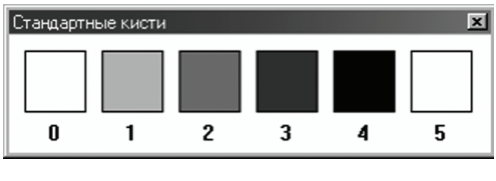


Рис. 4.1. Стандартные кисти Windows

```
    SelectObject(DC,Brush);
    FillRect(DC,Rct,Brush);
    TextOut(DC,Rct.Left+(Rct.Right-Rct.Left) DIV 2 -5 ,
              Rct.Bottom+5,PChar(IntToStr(Br)),1);
    OffsetRect(Rct,50,0); //сдвиг прямоугольной области вправо
    end;
  end;
```

Как и все заранее подготовленные объекты GDI стандартные кисти доступны только для чтения, поэтому если нас не устраивают их атрибуты, то придется создавать свой собственный, более сговорчивый экземпляр. Впрочем, в спектре заранее подготовленных кистей имеется одно счастливое исключение. Это кисть DC_BRUSH, она не страдает звездной болезнью, посему позволяет изменять свой цвет; для этого предназначена функция:

Win GDI
```
Function SetDCBrushColor(DC:HDC; Color :  COLORREFF) : COLORREFF;
```

Здесь DC – дескриптор контекста устройства; Color – новый цвет кисти. Ниже представлен фрагмент кода, демонстрирующий порядок работы со стандартной кистью DC_BRUSH. Не забывайте, что кисть такого рода существует только в ОС класса Windows NT, 2000, XP и Vista.

Для выяснения текущей «боевой раскраски» кисти DC_BRUSH применяют функцию:

Win GDI
```
Function GetDCBrushColor(DC:HDC) : COLORREFF;
```

Перед обращением к функции не забудьте выбрать кисть в контекст, иначе вызов окажется безрезультатным.

Все стандартные кисти находятся в единоличной собственности графического ядра Windows, поэтому пользовательские приложения не могут уничтожить эти кисти или даже модифицировать их. Единственное исключение из правил – кисть DC_BRUSH*, допускающая изменение цвета.*

Атрибуты кисти, структура TLogBrush

Как уже упоминалось во вступлении к этой главе, любая кисть характеризуется тремя атрибутами: цветом, стилем и узором. Для описания свойств кисти в Windows GDI введена структура TLogBrush:

```
type TLOGBRUSH = packed record
  lbStyle: UINT;     //поле стиля кисти
  lbColor: COLORREF; //поле цвета
  lbHatch: Longint;  //поле узора
end;
```

Первое поле структуры определяет стиль кисти. Предусмотрено девять констант стиля (табл. 4.2).

Для определения цвета кисти обращаемся ко второму полю структуры. Но при этом учитываем несколько особенностей. Во-первых: если выбран стиль

кисти BS_HOLLOW, BS_NULL, BS_PATTERN или BS_PATTERN8X8, то установленный в поле lbColor цвет кисти игнорируется. Во-вторых: если применен стиль BS_DIBPATTERN или BS_DIBPATTERNPT, то вместо значения цвета в lbColor передается константа DIB_PAL_COLORS или DIB_RGB_COLORS. Первая константа говорит, что таблица цветов DIB-образа содержит 16-разрядные индексы цветов в логической палитре. В свою очередь константа DIB_RGB_COLORS указывает на то, что изображение DIB содержит литеральные коды цветов RGB.

Таблица 4.2. Константы стиля кисти

Константа	Описание
BS_DIBPATTERN BS_DIBPATTERN8X8	Узорная кисть на основе независимого от устройства битового образа (*device-independent bitmap*, DIB). В этом случае в поле lbHatch структуры TLogBrush передают дескриптор изображения DIB. При выборе рисунка помните, что ОС Windows 95 не поддерживает кисти размером более чем 8×8 пикселов, у остальных современных версий Windows такого ограничения нет.
BS_DIBPATTERNPT	То же, что и BS_DIBPATTERN, но теперь в поле lbHatch должен находиться указатель на DIB-образ.
BS_HATCHED	Штриховая кисть.
BS_HOLLOW BS_NULL	Пустая кисть.
BS_PATTERN BS_PATTERN8X8	Узорная кисть, определенная битовым образом в памяти.
BS_SOLID	Сплошная кисть.

Поле lbHatch многофункционально. В случае, когда явным образом определено, что кисть штриховая (стиль кисти BS_HATCHED), то в это поле передается константа штриховки (табл. 4.3). Если установлен стиль BS_DIBPATTERN или BS_DIBPATTERN8X8, то поле должно содержать дескриптор изображения DIB. Стиль BS_DIBPATTERNPT требует, чтобы в поле lbHatch оказался указатель на упакованное в памяти растровое изображение.

Таблица 4.3. Константы штриховки кисти

Константа	Значение	Описание
HS_HORIZONTAL	0	Горизонтальная штриховка.
HS_VERTICAL	1	Вертикальная штриховка.
HS_FDIAGONAL	2	Штриховка с наклоном 45 градусов. Линии направлены из левого верхнего угла в нижний правый угол закрашиваемой области.
HS_BDIAGONAL	3	Штриховка с углом 45 градусов. Линии направлены из правого верхнего угла в нижний левый угол.
HS_CROSS	4	Штриховка из комбинации горизонтальных и вертикальных линий.
HS_DIAGCROSS	5	Диагональная штриховка из наклонных линий.

Все представленные в таблице варианты штриховки вы позднее увидите на рис. 4.2.

Согласно SDK в Windows для описания кисти кроме структуры TLOGBRUSH *может использоваться* LOGBRUSH32. *Фактически записи идентичны, единственное отличие – в применении более поздней структуры* LOGBRUSH32. *Если в коде проекта используется структура* EMRCREATEBRUSHINDIRECT *(частный случай применения функции* CreateBrushIndirect*) и, ко всему прочему, приложение предназначено для работы как в 32-разрядной, так и в 64-разрядной Windows, то при описании кисти Microsoft рекомендует отдать предпочтения* LOGBRUSH32.

При необходимости структура TLOGBRUSH позволит исследовать объект кисти. Для этого потребуется небольшая помощь функции GetObject().

```
var Brush:HBRUSH; LogBrush:TLogBrush;
//…
GetObject(Brush,SizeOf(TLogBrush),@LogBrush);
```

Эти строки кода демонстрируют способ сбора информации о кисти с дескриптором Brush в запись LogBrush.

Кисти системных цветов

Окна и элементы управления большинства выполняющихся под управлением Windows приложений окрашиваются в гамму заранее подготовленных системных цветов. Цвета заголовка и поверхности окна, полосы прокрутки, меню, кнопки и отбрасываемые ими тени гармонируют друг с другом и делают интерфейс привлекательным. Поэтому, если художественное решение нашей программы должно соответствовать текущему цветовому дизайну, то приложению лучше всего воспользоваться однородной кистью с цветом, соответствующим системному цвету операционной системы. Для получения дескриптора экземпляра кисти системных цветов предназначена функция:

Win GDI

```
Function GetSysColorBrush(Index: Integer) : HBRUSH;
```

Единственный параметр кисти – константа системного цвета, например COLOR_INACTIVECAPTION или COLOR_WINDOW.

Хотя кисти системных цветов не относятся к разряду стандартных объектов GDI, но за их создание и уничтожение также отвечает операционная система. Вероятнее всего, кисти системных цветов формируются Windows по мере обращения к функции GetSysColorBrush() *из приложений. В таком случае GDI находит дескриптор запрашиваемой кисти в кэше и отдает его во временное пользование приложению.*

В русскоязычных версиях Windows системные цвета настраиваются в панели управления компьютером с помощью апплета «Экран», кроме того, системные цвета могут изменяться на программном уровне, путем вызова функции SetSysColor().

Создание логической кисти

Классифицируя логические кисти, можно выделить однородные, штриховые и растровые кисти. В свою очередь, растровые кисти подразделяются на кисти, сформированные на основе аппаратно-зависимых (DDB) и аппаратно-независимых битовых образов (DIB).

Простейшая из кистей – однородная. Для создания экземпляра такой кисти предназначена функция:

```
Function CreateSolidBrush(Color: COLORREFF):HBRUSH;
```

В результате мы получим кисть с цветом, определенным в параметре `Color`. Если контекст устройства (в который выбирается цветная кисть) не в состоянии работать в полноцветном режиме и, как следствие, использует палитру, то GDI постарается подыскать соответствующий заказанному цвету индекс в списке красок палитры. Если полного совпадения добиться не удается, то контекст сымитирует требуемый цвет, комбинируя доступные оттенки.

Все созданные программистом логические объекты GDI должны быть удалены после выполнения своего функционала. Для удаления логических кистей применяется метод `DeleteObject()`. *Будьте внимательны: перед уничтожением кисти надо удостовериться, что она не выбрана в контексте устройства.*

Следующая по степени возрастания сложности кисть называется штриховой. Визуально она представляет собой рисунок, состоящий из повторяющихся вертикальных, горизонтальных или диагональных линий. Специалистом по созданию подобной кисти считается функция:

```
Function CreateHatchBrush(Hatch : Integer; Color : COLORREF) : HBRUSH;
```

Полученная кисть включает пикселы двух цветов. Цвет пикселов линий определяется параметром `Color`, цвет фоновых пикселов зависит от того, какое значение было установлено функцией `SetBkColor()`. Можно вообще отказаться от вывода фоновых пикселов, для этого необходимо воспользоваться функцией `SetBkMode()`, установив прозрачный режим заполнения фона контекста (`TRANSPARENT`). Кроме цветового аргумента в функции имеется параметр `Hatch`, определяющий вид штриховки. В него передаются константы из табл. 4.3. Ниже предложен простейший пример, демонстрирующий процесс создания кистей всех шести стандартных типов штриховки.

```
procedure GDI_Paint(DC:HDC);
VAR Brush  : HBRUSH;
    HS:Byte;
    Rct:TRect;
begin
SetBKMode(DC,TRANSPARENT); //отказ от вывода фоновых пикселов
SetRect(Rct,10,10,60,60);
for HS:=0 to 5 do
  begin
```

```
    Brush:=CreateHatchBrush(HS,$000000);
    FillRect(DC,Rct,brush);
    TextOut(DC,Rct.Left,Rct.Bottom+2,PChar(IntToStr(HS)),1);
    SelectObject(DC, GetStockObject(NULL_BRUSH));
    DeleteObject(Brush);
    OffsetRect(Rct,60,0);
    end;
end;
```

Фоновые пикселы штриховой кисти отображаются, только когда с помощью функции `SetBkMode()` *установлен непрозрачный режим заполнения фона* `OPAQUE`.

Результат работы процедуры представлен на рис. 4.2.

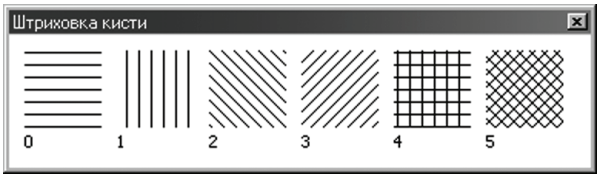


Рис. 4.2. Варианты штриховки

Специалисты не рекомендуют применять стандартные штриховые кисти в документах, предназначенных для печати. На экране с разрешением до 120 точек на дюйм подобный документ смотрится неплохо, но при выводе на печатающее устройство (особенно с высоким разрешением) штриховка сливается. Причина в том, что размер узора стандартной логической кисти GDI определяется не в логических единицах, а в единицах устройства – пикселах. При уменьшении масштаба узор «съедается» принтером.

Настала очередь познакомиться с наиболее сложной растровой кистью. Рисунок такой кисти создается на основе битового аппаратно-зависимого (DDB) или аппаратно-независимого образа (DIB). Для работы с растром DDB предназначена функция:

Win GDI

```
Function CreatePatternBrush(BITMAP:HBITMAP): HBRUSH;
```

Единственный параметр функции – дескриптор растровой картинки. Ниже представлен фрагмент кода, демонстрирующий работу этой функции. Опорные растровые картинки создаются динамически во время работы программы. Для этого объявлены три массива `Pattern1`, `Pattern2` и `Pattern3`.

```
const
  Pattern1 : array[0..7] of WORD=($F0,$F0,$F0,$F0,$0F,$0F,$0F,$0F);
  Pattern2 : array[0..7] of WORD=($7F,$7F,$00,$00,$7F,$7F,$00,$00);
  Pattern3 : array[0..7] of WORD=($FF,$00,$FF,$FF,$00,$00,$00,$00);
```

Для создания битового образа применяется функция `CreateBitmap()`. В листинге формируется растровая картинка размером 8×8 пикселов, именно та-

ков максимальный размер узорной кисти в Windows 95. Во всех более поздних версиях Windows это ограничение снято.

```
Procedure GDI_PAINT(DC:HDC);
var   Brush  : HBRUSH;
      BMP:HBITMAP;
      Rect:TRect;
begin
SetRect(Rect, 10, 10, 60, 60);

BMP:=CreateBitmap(8, 8, 1, 1, @Pattern1);
 Brush:=CreatePatternBrush(BMP);
 FillRect(dc, rect, brush);
 SelectObject(DC, GetStockObject(NULL_BRUSH));
 DeleteObject(BMP); DeleteObject(Brush);
//и т. д.
end;
```

Результаты работы листинга представлены на рис. 4.3.

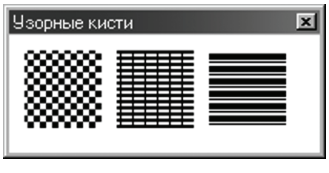


Рис. 4.3. Создание узорной кисти

Функция `CreatePatternBrush()` создает растровую кисть на основе аппаратно-зависимого растра. Если логика программы предполагает использование аппаратно-независимого растра, то нам на помощь придет функция `CreateDIBPatternBrushPt()`:

Win GDI

```
Function CreateDIBPatternBrushPt(DIBPacked:Pointer;
                                 ColorSpec : Cardinal):HBITMAP;
```

Для поддержки совместимости с 16-разрядными версиями Windows в API сохранена функция `CreateDIBPatternBrush()`.

Параметр `DIBPacked` рассчитывает получить указатель на упакованный DIB-образ. Параметр `ColorSpec` может принимать одно из следующих значений: `DIB_PAL_COLORS` или `DIB_RGB_COLORS`. Первая константа указывает на то, что при построении кисти задействуется цветовая палитра, вторая константа говорит, что таблица цветов растрового образа содержит значения цветов модели RGB.

Наиболее универсальной из функций, позволяющих создавать логическую кисть, является:

Win GDI

```
Function CreateBrushIndirect(const LOGBRUSH: TLogBrush):HBRUSH;
```

Функция формирует кисть на основе данных структуры `TLOGBRUSH`.

В визуальной библиотеке компонентов Delphi кисть описывается классом `TBrush`. *Здесь для создания экземпляра кисти GDI используется функция* `CreateBrushIndirect()`.

Выравнивание штриховых кистей

По умолчанию при заливке областей штриховой кистью левый верхний пиксел узора ложится в верхнюю левую точку закрашиваемой области. Все остальные пикселы кисти следуют за базовой точкой узора, последовательно складываясь в мозаичную картину. В ряде случаев (например, при прокрутке закрашиваемой области, соединении нескольких областей) возникает потребность в смещении базовой точки узора. Для решения такой задачи предназначена функция:

```
Function SetBrushOrgEx(DC:HDC; XOrg, YOrg : Integer; PrevPt:pPoint) :
Boolean;
```

Смещение левого верхнего пиксела штриховой кисти осуществляется параметрами `XOrg` и `YOrg`. Если ведется речь о кисти размером 8×8 пикселов, то в параметры смещения допускается передавать значения в диапазоне от 0 до 7. При необходимости сохранить предыдущие координаты базовой точки обработайте параметр `PrevPt`; это указатель на структуру `TPoint`.

В GDI предусмотрена функция, способная проинформировать программиста о базовой точке.

```
Function GetBrushOrgEx(DC : HDC; Pt : pPoint) : Boolean;
```

В результате мы получим указатель на структуру `TPoint`.

Сброс базовой точки в исходное состояние осуществляет функция `UnrealizeObject()`.

РЕЗЮМЕ

Основное назначение кисти – упростить и ускорить процесс закраски какой-то области или фигуры. В Windows предусмотрены несколько методов создания экземпляра кисти, начиная от простейших однотонных до цветных кистей с особой штриховкой. Кроме применения предопределенных стилей штриховки программисту разрешено создавать свои стили, основанные на растровых рисунках. В главе 21 мы познакомимся с еще одним способом закраски области – градиентной заливкой с помощью функции `GradientFill()`.

5

Перья

Если вдруг нам понадобится провести тонкую линию, то мы возьмем в руки карандаш, ручку или фломастер, а не малярную кисть. Поэтому *перо* (англ. *Pen*) принадлежит к перечню наиболее популярных объектов в GDI. Это тот инструмент, от характеристик которого зависит внешний вид всех рисуемых линий и кривых. Программисты различают два вида перьев – *косметические* и *геометрические*. В обычных приложениях наиболее часто применяются косметические перья. Это простейший вид пера, описываемый тремя параметрами: цветом, стилем линии и толщиной. Косметическое перо как нельзя лучше подходит для черчения тонких линий и кривых. Геометрическое перо развивает возможности своего коллеги – оно дополнено еще четырьмя атрибутами: узором, типом наконечника, типом соединения (стыком) и штриховкой. Одна из существенных особенностей геометрического пера в том, что проведенные им линии чувствительны к двумерным аффинным преобразованиям, поэтому при работе приложения в мировых координатах геометрические перья превосходят своих косметических коллег.

В GDI предусмотрено три способа создания экземпляра пера. Функции `CreatePen()` и `CreatePenInderect()` создают простые косметические перья. Это достаточно древние методы, существующие со времен 16-разрядных версий Windows. С выходом в свет 32-разрядной Windows появилась функция `ExtCreatePen()`, позволяющая создавать косметическое и геометрическое перо с расширенными возможностями. Все перечисленные функции возвращают дескриптор пера `HPEN`. Несмотря на поддержку `ExtCreatePen()` в домашних операционных системах класса Windows 95/98/Me возможности функции полностью раскрываются лишь начиная с Windows NT 4.0 SP4.

Стандартные перья GDI

Любому контексту устройства разрешено пользоваться услугами целого трио заранее подготовленных стандартных перьев. Дабы контекст не растерял подарки, стандартные перья не передаются ему в единоличное пользование,

корректнее сказать, что контекст арендует перья у операционной системы. Стандартные перья прочно «вмонтированы» в GDI и ни при каких обстоятельствах не собираются с ним расставаться. Попытки программистов-новаторов извести стандартные перья со света с помощью классических (подобных DeleteObject()) и не очень классических функций не производят на Windows ни малейшего впечатления, так что даже не тратьте на это время. Для каждого из стандартных перьев объявлена соответствующая константа. Черное перо откликается на BLACK_PEN, белое – на WHITE_PEN (см. табл. 2.2). Оба пера способны провести непрерывную линию толщиной в один пиксел. Что касается третьего пера, то это длинная история. Видимо, в Microsoft читали русские народные сказки про то, как жили-были три брата, два умных, а третий... сами знаете. Наверное, поэтому последнее стандартное «перышко» рисовать в принципе не умеет, за это оно названо NULL_PEN.

Надо полагать, что пустое перо предназначено для корректного проведения операции DeleteObject(). Допустим, мы заканчиваем обработку сообщения WM_PAINT, до этого в контекст было выбрано какое-то созданное нами перо с дескриптором Pen.

```
Pen = CreatePen(... //создали перо Pen
SelectObject(DC, GetStockObject(PEN)); //выбрали Pen в контекст
//операции с использованием пера Pen
SelectObject(DC, GetStockObject(NULL_PEN)); //выбрали в контекст пустое перо
DeleteObject(Pen); //теперь мы можем удалить перо Pen
```

Операционная система не позволит удалить объект, если он выбран в атрибут контекста графического устройства, поэтому мы замещаем перо Pen пустышкой и только после этого освобождаем ресурс.

В современной графической библиотеке GDI+ отсутствуют понятия пустого пера и пустой кисти.

Несмотря на то, что во власти GDI находится сразу тройка перьев, контекст в одно и то же время способен чертить только одним из них, держа оставшиеся перья наготове. По умолчанию в руке контекста зажато черное перо. Чтобы получить дескриптор альтернативного пера, подставьте его константу в единственный параметр функции GetStockObject().

Получив дескриптор объекта, нам остается передать его в руки контекста, вызвав функцию SelectObject().

```
SelectObject(DC, GetStockObject(WHITE_PEN)); //выбираем белое перо
MoveToEX(DC,0,0,nil); LineTo(DC,100,100);    //проводим наклонную черту
```

В современных версиях Windows к перечню трех заранее подготовленных перьев добавлено четвертое. Ему соответствует константа DC_PEN. Это сплошное перо единичной толщины, его ключевая особенность – возможность изменять цвет.

Модификация цвета пера

Говоря о недостатках работы с объектами GDI, разработчики программного обеспечения зачастую отмечают сложность модификации их параметров. После создания объекта мы его обязаны воспринимать таким, какой он есть, он монументален подобно памятнику и практически не позволяет настраивать свои характеристики. Если вести речь о стандартных перьях GDI, то они (за исключением DC_PEN) доступны только для чтения.

Для изменения цвета пера DC_PEN предназначена функция:

```
Function SetDCPenColor(DC:HDC; Color : COLORREF) : COLORREF;
```

Благодарите и за эту милость. С остальными характеристиками еще сложнее – если вам потребуется поправить стиль или толщину, то будьте добры, создавайте новый экземпляр пера.

Косметическое перо

Итак. Если характеристики стандартных перьев GDI вас не устраивают, то следует «заточить» свое собственное логическое перо. Простейший способ создания экземпляра пера с покладистым характером предоставит функция CreatePen():

```
Function CreatePen(PenStyle : Integer;
                   Width : Integer;
                   Color : COLORREF):HPEN;
```

В результате формируется перо косметического типа. Оно обладает тремя параметрами: PenStyle – стиль пера, Width – толщина и Color – цвет. Функция CreatePen() может воспользоваться первыми семью стилями пера, представленными в табл. 5.1 и на рис. 5.1.

Таблица 5.1. Стили пера

Константа	Значение	Описание	Примечание
PS_SOLID	0	Сплошная линия.	Стили доступны всем спектру 32- и 64-разрядных ОС Windows.
PS_DASH	1	Линия, состоящая из отрезков.	
PS_DOT	2	Линия точек.	
PS_DASHDOT	3	Линия из отрезков и точек.	
PS_DASHDOTDOT	4	Отрезок + две точки.	
PS_NULL	5	Невидимая линия.	
PS_INSIDEFRAME	6	Сплошная линия.	
PS_USERSTYLE	7	Пользовательский стиль.	Стили могут применяться только в функции ExtCreatePen().
PS_ALTERNATE	8	Стиль используется только косметическим пером, он определяется набором пикселов.	

При работе с функциями CreatePen() и CreatePenIndirect() перечисленные в таблице стили (за исключением PS_SOLID и PS_INSIDEFRAME) имеют смысл лишь в том случае, когда толщина пера равна единице. Если это значение превышено, то вне зависимости от назначенного стиля пера GDI на экран выводится сплошная линия. Надо отметить, что в аналогичной ситуации перо GDI+ ведет себя более покладисто и сохраняет свой стиль при любой толщине.

Применение стилей PS_DASH, PS_DOT, PS_DASHDOT, PS_DASHDOTDOT *в GDI при выводе линий и кривых целесообразно только при создании перьев с толщиной, не превышающей единицу.*

При рисовании утолщенных линий GDI старается выводить дополнительные пикселы симметрично основной оси линии. Так, если мы проводим горизонтальную линию толщиной 3 пиксела из точки (0,10) в точку (100,10) она займет точки с координатой Y, равной 9, 10 и 11. Заметьте, что я недаром применил слово «старается». В страничном режиме отображения корректная центровка утолщенной линии относительно ее оси возможна только в случае, когда толщина пера описывается нечетным числом пикселов. Кроме того, у Windows возникают сложности с центровкой линий, проходящих под углом к осям координат. Если толщина пера превышает единицу и назначен стиль PS_INSIDEFRAME, то при выводе всех графических примитивов, вписанных в прямоугольник (прямоугольник, эллипс, сектор окружности, и т. п.), результирующая линия не центруется относительно главной оси, а смещается вовнутрь фигуры.

Ниже представлен листинг, демонстрирующий способ создания косметических перьев семи стандартных стилей.

```
Procedure GDI_PAINT(DC:HDC);
var Pen:HPen;
    PenStyle:Byte;
begin
 for PenStyle:=0 to 6 do //индекс стиля пера
 begin
  Pen:=CreatePen(PenStyle,1,RGB(0,0,0)); //создали перо
  SelectObject(DC,Pen); //выбираем перо в контекст устройства
  TextOut(DC,0,20*PenStyle,PChar(IntToStr(PenStyle)),1);
  MoveToEx(DC,15,10+20*PenStyle,nil); //начальная позиция пера
  LineTo(DC,200,10+20*PenStyle); //команда на вывод линии
  SelectObject(DC,GetStockObject(NULL_PEN)); //выбрали в контекст системное перо
  DeleteObject(Pen); //удалили использованное перо
  end;
end;
```

В рамках обработки сообщения WM_PAINT мы последовательно создаем семь перьев и твердой рукой чертим горизонтальные линии. Для вывода линии применяется пара функций MoveToEx() и LineTo(). Для наглядности все линии пронумерованы. Результат нашей работы вы видели чуть раньше, на рис. 5.1.

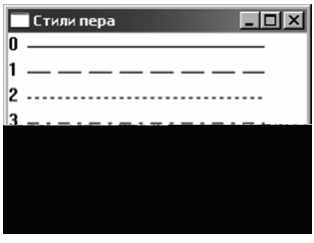


Рис. 5.1. Стили пера

Не забывайте удалять все созданные вашими руками объекты GDI. Это правило в полной мере относится и к перьям. Для удаления пера применяется функция `DeleteObject()`. *Перед ее вызовом надо убедиться, что удаляемое перо не было выбрано в контекст устройства. Именно поэтому в представленном выше примере перед удалением логического пера его место в контексте замещается стандартным пером* `NULL_PEN`.

Второй способ создания пера с одновременной настройкой его параметров предоставит функция:

Win GDI
```
Function CreatePenIndirect(const LogPen : TLogPen):HPEN;
```

Фактически это аналог функции `CreatePen()`, с той лишь разницей, что для определения характеристик пера в функцию передается не три параметра, а единственная структура `TLogPen`. Впрочем, поля структуры целиком и полностью соответствуют параметрам функции `CreatePen()`, так что эти функции – близнецы-братья:

```
Type TLOGPEN = packed record
 lopnStyle: UINT;     //стиль пера
 lopnWidth: TPoint;//толщина пера передается в поле x, поле y не используется
 lopnColor: COLORREF; //цвет пера
end;
```

В операционных системах Windows NT/2000/XP/Vista окончания утолщенных линий по умолчанию имеют полукруглую форму.

В среде Delphi перо инкапсулируется в классе `TPen`. *Это аналог косметического пера, для его создания Delphi пользуется услугами функции* `CreatePenIndirect()`.

Геометрическое перо

Третий способ создания экземпляра пера наиболее интересен. Благодаря функции `ExtCreatePen()` мы получаем возможность конструировать перья с расширенным спектром возможностей. Более того, функция позволяет вызывать к жизни не только банальное косметическое, но и более совершенное – геометрическое перо.

```
SelectObject(DC,Pen);
MoveToEx(DC,15,10,nil); LineTo(DC,200,10);
SelectObject(DC,GetStockObject(NULL_PEN));
DeleteObject(pen);

//другие варианты окончания линий
end;
```

Обратите внимание на снимок окна этой программы (рис. 5.2), на нем проявляется отличие линии, нарисованной пером с флагом PS_ENDCAP_FLAT, от линий PS_ENDCAP_ROUND и PS_ENDCAP_SQUARE. Хотя в коде программы протяженности по горизонтали всех линий задаются равными, сразу заметно, что линия PS_ENDCAP_FLAT несколько короче. Причина кроется в том, что отрезки PS_ENDCAP_ROUND и PS_ENDCAP_SQUARE получили приращение за счет добавления к их концам полукруга и квадрата, а окончания линии PS_ENDCAP_FLAT просто отсечены.

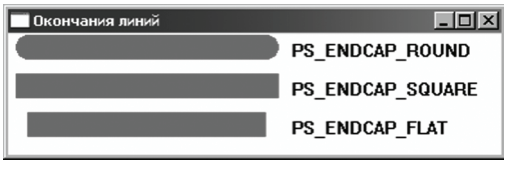


Рис. 5.2. Три варианта окончания утолщенных линий

Степень влияния констант PS_JOIN_BEVEL, PS_JOIN_MITER и PS_JOIN_ROUND на соединение линий наглядно продемонстрирована на рис. 5.3. Это экранный снимок окна с девятью (напоминающими букву «М») фигурами, нарисованными функцией PolyLine(). Каждая из фигур прочерчена пером с уникальной комбинацией констант, определяющих типы окончаний и соединений линий.

Иногда при соединении линий, нарисованных пером PS_JOIN_MITER, возникает необходимость принудительного ограничения протяженности заостренного стыка. Подобное развитие событий вполне вероятно при соединении отрезков, проходящих под малыми углами друг к другу, в результате чего получается неоправданно вытянутое соединение. Специалистом по ограничению прав «распоясавшихся» стыков считается функция:

Win GDI

```
Function SetMiterLimit(DC:HDC; NewLimit:single;
                                OldLimit:pSingle):Boolean;
```

Функция определяет максимальное отношение длины заострения к толщине линии – угловой предел. По умолчанию он равен 10 единицам.

Продолжим изучать функцию ExtCreatePen(). Ее второй параметр Width несет персональную ответственность за толщину пера. Если геометрическое перо создается в мировых координатах, то в качестве единицы измерения толщины выступает не обычный пиксел, а логическая единица. Так что мы приобретаем право чертить линии, измеряемые в долях миллиметров, дюймах, метрах или в чем-нибудь еще, – все зависит от текущих настроек контекста устройства (см. главу 11, «Системы координат и режимы отображения Windows»).

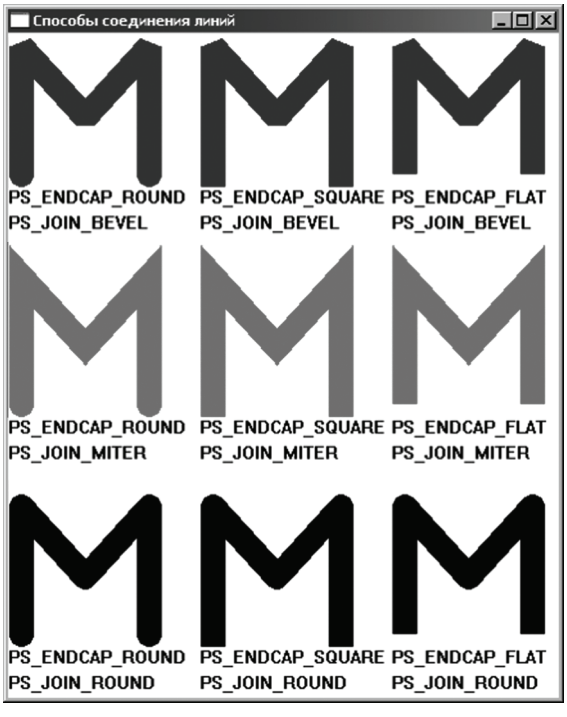


Рис. 5.3. Варианты соединения линий

Через третий параметр функция рассчитывает получить запись `TLogBrush`. Указанная структура взята напрокат у другого объекта GDI – кисти. Она содержит поля цвета, стиля и штриховки кисти. При создании косметического пера единственным значимым полем структуры является поле `lbColor`, определяющее цвет пера. Если мы формируем геометрическое перо, то задействуются все поля `TLogBrush`. В таком случае совсем не сложно создать некий симбиоз пера и кисти – узорное перо. В следующих строках кода мы так и поступаем.

```
LogBrush.lbColor:=RGB(0,0,0);    {черный цвет}
LogBrush.lbStyle:=BS_HATCHED;    {стиль - узор}
LogBrush.lbHatch:=HS_DIAGCROSS; {вид узора - диагональные линии}
pen:=ExtCreatePen(PS_GEOMETRIC, 19, LogBrush, 0, nil);
```

В результате получилось перо с диагональным узором. На рис. 5.4 предложены шесть основных узоров для пера.

Совместное использование параметров `StyleCount` и `Style` позволяет нам определить свою собственную последовательность чередования отрезков. Таким образом, мы не только сможем повторить любой из классических стилей пера GDI, но и создать свой собственный. Алгоритм чередования закрашенных и пустых пикселов описывается в массиве `Style`. Первый элемент массива со-

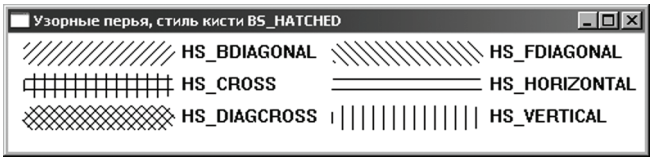


Рис. 5.4. Узорные перья

ответствует длине отрезка, второй элемент – длине пустого промежутка, третий – длине следующего отрезка и т. д. В `StyleCount` передается общее количество элементов массива. Ниже предложен отрывок кода, создающий перо нестандартного стиля.

```
const STYLE : array[0..3] of CARDINAL =(12,10,5,5);
//…
pen:=ExtCreatePen(PS_GEOMETRIC OR PS_USERSTYLE,1,LogBrush, 4, @STYLE);
```

Проведенная таким пером черта будет представлять чередование: 12 закрашенных пикселов, 10 пустых пикселов, 5 закрашенных и 5 пустых. Заметьте, что при формировании своего собственного алгоритма чередования закрашенных и пустых пикселов перо должно создаваться с константой `PS_USERSTYLE`.

При построении приложений учитывайте, что скорость вывода линии и кривых находится в прямой зависимости от вида пера. За счет простоты реализации косметическое перо быстрее своего геометрического коллеги!

РЕЗЮМЕ

Перо – один из фундаментальных объектов Windows GDI. От его характеристик зависит внешний вид выводимых на экран или другое устройство линий и кривых. К основным параметрам пера относятся цвет, стиль и толщина. Различают два вида перьев: косметические и геометрические. Косметические перья предназначены для черчения тонких (толщиной в один пиксел) отрезков. Напротив, геометрические перья специализируются на рисовании утолщенных линий. Контекст устройства может воспользоваться тремя заранее подготовленными перьями либо перьями, созданными в ваших проектах.

6

Линии и кривые

Что может быть проще, чем провести линию? Мы с вами делали это сотни раз: в детстве, старательно выводя лучи от солнца, на уроках в школе, на скучных лекциях в институтах, на бесполезных совещаниях и семинарах во взрослой жизни. Что нужно для того, чтобы начертить линию на практике? В простейшем случае достаточно листа бумаги и карандаша. Если уж идти дальше, то было бы неплохо еще обзавестись линейкой, а для черчения кривых воспользоваться лекалом. Не такие уж серьезные пожелания...

Для рисования линий и кривых в GDI предусмотрено немногим более десятка функций. Они относительно просты в обращении, хотя и опираются на весьма серьезные алгоритмы. Несмотря на внешнюю незатейливость, кривые и линии позволяют сформировать графический образ практически любой степени сложности. Ведь именно из них складываются контуры более серьезных геометрических фигур, а они в свою очередь формируют изображения элементов управления, рисуют таблицы, диаграммы и аналитические графики.

Основной инструмент, благодаря которому на поверхности контекста появляется линия, это не что иное, как рассмотренное в прошлой главе перо. Однако просто факт существования экземпляра пера недостаточен для того, чтобы графический механизм Windows смог им воспользоваться. Перо должно быть выбрано в контекст устройства, иначе это не более чем карандаш, лежащий в коробке на верхней полке шкафа. Для того чтобы контекст «взял карандаш в руку», потребуется вызов функции `SelectObject()`. Стоит учитывать, что контекст способен рисовать только одним пером. Поэтому, если характеристики очередной проводимой линии отличаются от предыдущей, то для ее черчения потребуется смена пера.

Позиционирование пера

Как мы проводим линию на листе бумаги? Первым делом устанавливаем карандаш в ее начальной точке. Примерно этого же и ожидает контекст графи-

ческого устройства – он рассчитывает узнать от нас, из какой именно координаты на его поверхности необходимо начать закрашивать пикселы. Если мы забудем это сделать, то вывод изображения начнется из точки с логическими координатами (0,0) – это положение пера по умолчанию. Есть еще один вариант развития событий – процесс графического вывода стартует из точки, в которой перо оказалось после выполнения предыдущей операции (например, вывода другой линии). Для перемещения пера в намеченную позицию применяют функцию:

```
Function MoveToEx(DC:HDC; X,Y : Integer; PriorPos : pPoint) : Boolean;
```

Функция ничего не рисует, она просто позиционирует перо в точке с логическими координатами (X,Y). Последний параметр – указатель на запись TPoint, в которую сохранится предыдущее положение пера. Если такой необходимости нет, то MoveToEx() не станет возражать, если в этот параметр мы передадим неопределенный указатель nil.

Вполне естественно, что в наборе функций Windows GDI предусмотрен способ информирования о текущем местоположении пера. Ведь вероятно, что вместо того, чтобы двигать мебель по квартире (читай, устанавливать перо в новую точку), стоит проверить – может быть, текущее состояние дел нас вполне устраивает:

```
Function GetCurrentPositionEx(DC : HDC; CurrentPos : pPoint):Boolean;
```

Результаты мы обнаружим в выходном параметре CurrentPos, который (как вы уже догадались) представляет собой указатель на переменную TPoint.

Текущая позиция пера интересует далеко не все функции GDI. В части, касающейся рисования линий и кривых, текущая позиция пера используется функциями: LineTo(), PolyLineTo(), PolyBezierTo(), ArcTo(), AngleArc() *и* PolyDraw(). *В свою очередь, функциям* PolyLine(), PolyBezier(), PolyPolyline(), Arc() *положение пера безразлично.*

Линии

Подготовив перо и выбрав его (посредством вызова SelectObject) в контекст устройства, мы получаем возможность начертить линию. Наиболее просто желаемый результат достигается с помощью функции:

```
Function LineTo(DC : HDC; XEnd, YEnd : Integer);
```

Отрезок проводится из текущей позиции пера в точку с координатами (XEnd, YEnd). Обязательно запомните, что последний пиксел с координатой (XEnd,YEnd) не закрашивается – он не входит в состав отрезка. После того как линия отобразится на экране, перо переводится в эту точку и отдыхает до поступления очередной команды. Внешний вид получившейся линии в первую очередь определяется настройками пера, но, кроме этого, на представление линии на графическом устройстве оказывают влияние очень многие факторы:

- установленный режим отображения (см. главу 11 «Системы координат и режимы отображения Windows»);
- мировые преобразования (см. главу 12 «Мировые координаты»);
- выбранная растровая операция (см. главу 17 «Растровые операции»);
- цвет и режим заполнения фона.

Кстати, это замечание справедливо для всех функций GDI, умеющих рисовать линии и кривые.

Столь большое число ингредиентов, связанных с выводом простейшей линии, предназначено не столько для запугивания начинающего программиста, сколько для предоставления ему дополнительного сервиса при решении на первый взгляд элементарной задачи – закрашивания пикселов. Поэтому (пока мы относимся не к клану гуру, а лишь к скромному отряду исследователей) при выводе обычной линии не станем отвлекаться на шаманские заклинания типа «двумерное аффинное преобразование» или «тернарная операция» и заострим внимание на самой сути вывода линий и кривых. А к заклинаниям мы вернемся позднее...

Практически ни одна из функций GDI при рисовании линии, кривой или другой, более сложной, фигуры не использует ее последний пиксел. Система полагает, что оставшаяся точка предназначена только для позиционирования пера. Благодаря такому решению Windows снижает вероятность повторной закраски одного и того же пиксела. В таком алгоритме работы в первую очередь заинтересованы принтеры и плоттеры, так как многократная заливка одной и той же точки листа бумаги краской может неблагоприятно сказаться на результате печати.

Анализ линии, функция LineDDA

Каким образом GDI принимает решение на закрашивание того или иного пиксела при выводе линии? Поверьте, это совсем не праздный вопрос. Судите сами. Если линия строго горизонтальна или вертикальна, то все ясно. Проводя отрезок из точки (1,10) в точку (1,100) пером единичной толщины, Windows, не притрагиваясь к координате X, дает последовательное приращение координате Y. Задействуются точки: (1,10), (1,11), (1,12), ..., (1,99). Но какие пикселы потребуются для прорисовки наклонной линии, допустим, проходящей из точки (10,10) в точку (140,180)? Со 100% точностью можно утверждать только то, что координаты первого используемого пиксела равны (10,10). Определить координаты следующей точки «на глаз» уже сложнее. Ими, с какой-то степенью вероятности, могут оказаться пиксел (11,11) или пиксел (10,11). А о координатах 50-й по счету точки вообще допустимо говорить лишь приблизительно. Взгляните на рис. 6.1, на нем показано, что для вывода казалось идентичных линий могут быть задействованы разные группы пикселов.

На сегодняшний день имеется достаточно большой перечень алгоритмов, посвященных «элементарному» вопросу вывода прямой. В современных версиях Windows линии рисуются на основе алгоритма GIQ (Grid Intersection Quantization). В соответствии с GIQ каждый пиксел окружается виртуальным ромбом со сторонами 1×1 пиксел. Если проводимая линия соприкасается с любой из сторон ромба, то пиксел закрашивается.

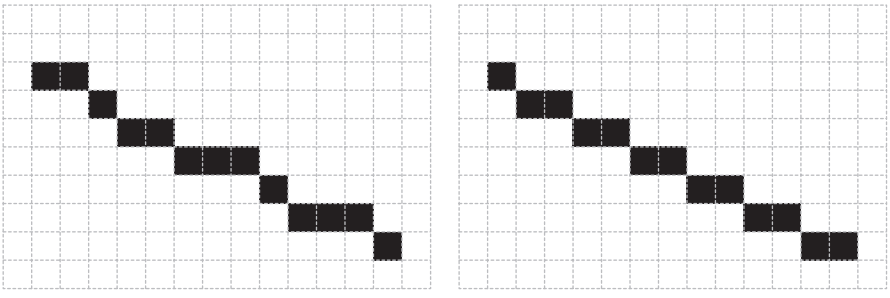


Рис. 6.1. Варианты вывода линии

Не вдаваясь в душещипательные подробности функционирования алгоритма GIQ, поговорим об одном важном следствии его работы. В состав Windows GDI включена функция LineDDA(), позволяющая проводить анализ прямой линии на интересующую нас тему – какой пиксел подвергнется экзекуции (будет закрашен).

Win GDI

```
Function LineDDA(XStart, YStart, XEnd, YEnd : Integer;
                 LineFunc : TFNLineDDAProc; Data : LPARAM) : boolean;
```

Функция ничего не рисует (как видите, среди ее аргументов даже нет намека на контекст устройства). Первые четыре параметра LineDDA() соответствуют координатам начальной и конечной точек исследуемой линии. Параметр LineFunc представляет собой ссылку на функцию косвенного вызова. Эта функция вызывается ровно столько раз, сколько элементарных точек задействуется для прорисовки линии. В Delphi функция обратного вызова типизируется следующим образом:

```
TFNLineDDAProc = procedure (p1, p2: Integer; Data : LPARAM); stdcall;
```

Здесь p1 и p2 как раз то, что нас интересует, – координаты X и Y очередной точки анализируемой линии. Параметр Data отдан во власть программисту и используется по его усмотрению.

Работу функции иллюстрирует следующий фрагмент кода:

```
Var Wnd, ListBox, Button : HWnd;
//…
{функция обратного вызова}
Procedure LineFunc(p1, p2: Integer; lParam: LPARAM); stdcall;
begin
  INC(Counter); //приращение счетчика точек
  s:=Format('№ %d - (%d,%d)',[Counter,p1,p2]);
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
  DC:=GetDC(Wnd);
  SetPixel(DC,p1,p2,$FF0000); //закрашиваем пиксел синим цветом
  ReleaseDC(Wnd,DC);
end;

{оконная процедура}
```

```
function WindowProc (Window : HWnd; Message, WParam : WPARAM;
         LParam : LPARAM) : LongInt; stdcall;
begin
  case Message of
  WM_CREATE : {создаем список и кнопку}
   begin
    GetClientRect(Window,Rct);
    LISTBOX:= CreateWindow('LISTBOX', '',
              WS_CHILD OR WS_BORDER OR (LBS_STANDARD - LBS_SORT),
              0, 48, 80, 30, Window, 0, HInstance, nil);

    Button:= CreateWindow('BUTTON', 'Анализ',
             WS_CHILD OR WS_BORDER OR WS_VISIBLE OR BS_PUSHBUTTON OR BS_LEFT,
             Rct.Right-90, rct.Bottom-32, 83, 30, Window, 0, HInstance, nil);
   end;
WM_COMMAND :
begin
  if (LParam=Button) then //щелчок по кнопке
   begin
    Counter:=0;
    //очистка списка
    while SendMessage(LISTBOX,LB_GETCOUNT,0,0)>0 do
          SendMessage(LISTBOX,LB_DELETESTRING,0,0);
    {анализ отрезка (10,10)-(140,180)}
    LineDDA(10,10, 140, 180,@LineFunc,DC);
    end;
 end;
```

Программка невелика. По нажатию кнопки `Button` очищается список и вызывается функция `LineDDA()`. В рамках функции обратного вызова заполняются строки списка `ListBox`, в них вы обнаружите порядковый номер и координаты задействованных пикселов (рис. 6.2). Кроме того, в функции обратного вызова с помощью `SetPixel()` мы закрашиваем пикселы, входящие в состав исследуемого отрезка.

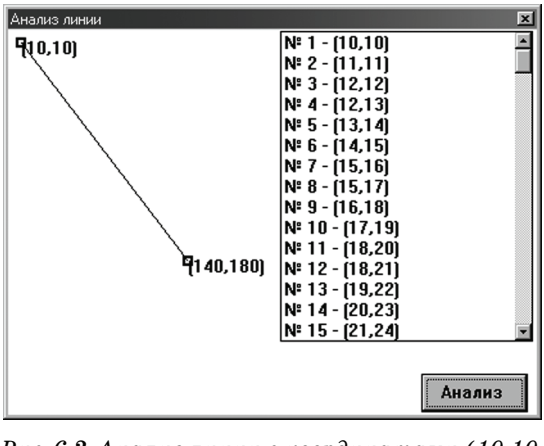


Рис. 6.2. Анализ линии с координатами (10,10)–(140,180)

*В библиотеке на сайте **www.symbol.ru** вы обнаружите немного усложненный исходный код программы. В нем пользователю позволено самостоятельно выбирать координаты анализируемого отрезка, для этого мы создаем еще два элемента управления – метки, которые разрешено перемещать мышью.*

Ломаная линия

Для того чтобы на поверхности устройства появилось изображение нескольких соединенных линий, допустимо остановиться на многократном вызове функций `LineTo()`, однако более эффективным решением станет обращение к специализированной функции `PolyLine()`:

```
Function Polyline(DC:HDC; var Points; Count : Integer) : Boolean;
```

Функция готова нарисовать ломаную линию на основе массива точек `Points`, число выводимых точек ограничивает параметр `Count`. `Polyline` не чувствительна к текущей позиции пера – вывод ломаной начнется из точки, на которую укажет младший элемента массива `Points`.

```
var Points : Array[0..2] of TPoint;
begin
Points[0]:=Point(50,50); Points[1]:=Point(75,75); Points[2]:=Point(25,75);
Polyline(Form1.Canvas.Handle,Points,3);
end;
```

Поговорим о ближайшем сподвижнике функции `Polyline()` – функции `PolyLineTo()`.

```
Function PolylineTo(DC:HDC; var Points; Count : Integer) : Boolean;
```

Функция предназначена для решения аналогичной задачи – черчения ломаной линии, но с той лишь разницей, что `PolylineTo()` рисовать начинает не из точки, описанной в младшем элементе массива `Points`, а из текущей позиции пера. После завершения работы функции текущая позиция пера перемещается в точку с координатами, соответствующими последнему элементу массива.

Хотя функции `Polyline()` и `PolylineTo()` не нацелены на рисование замкнутых контуров, но ничто не запрещает нам сымитировать такой контур путем ввода в массив еще одной точки, координаты которой будут совпадать с координатами начальной.

Если логика программы предполагает рисование группы сложных фигур, то вместо многократного вызова `Polyline()` или `PolylineTo()` стоит воспользоваться функцией:

```
Function PolyPolyline(DC : HDC; const Points;
                      const Vert; Count : cardinal) : Boolean;
```

Функция специализируется на выводе нескольких ломаных линий за один вызов. Координаты вершин всех линий хранятся в массиве `Points`. Сколько именно элементов массива `Points` отведено для описания точек той или иной фигуры, определяется в массиве `Vert`. Последний параметр функции содержит

не число вершин, а количество ломаных. Порядок использования функции предложен в следующем фрагменте кода:

```
Procedure GDI_PAINT(DC:HDC);
var  Points: array[0..8] of TPoint;
     Vert  : array[0..1] of CARDINAL;
begin
// Квадрат
Vert[0] := 5;
{для рисования квадрата задействовано 5 элементов массива Points}
Points[0].x := 10;  Points[0].y := 10;
Points[1].x := 100; Points[1].y := 10;
Points[2].x := 100; Points[2].y := 100;
Points[3].x := 10;  Points[3].y := 100;
Points[4].x := 10;  Points[4].y := 10;
// Треугольник
Vert[1] := 4;
{для рисования треугольника задействовано 4 элемента массива Points}
Points[5].x := 140; Points[5].y := 50;
Points[6].x := 140; Points[6].y := 110;
Points[7].x := 20;  Points[7].y := 110;
Points[8].x := 140; Points[8].y := 50;

PolyPolyline( DC, Points, Vert, 2);
end;
```

Выбирая между многократным вызовом функции `LineTo()` и обращением за услугой к функции `Polyline()` или `PolylineTo()`, лучше отдавать предпочтение нашим новым знакомым. Такое утверждение основывается на том, что:

1. Однократный вызов функции всегда выполняется быстрее, чем многократное обращение к одной и той же, пусть даже весьма расторопной функции.
2. Подготовленная с помощью `Polyline()` и `PolylineTo()` ломаная линия не содержит пикселов, прорисованных дважды. Это особенно важно во время печати и при проведении бинарных растровых операций.
3. При выводе ломаной линии стилевым пером каждый новый отрезок продолжает цикл чередования рисунка пикселов, а рисунок линии, проведенной функцией `LineTo()`, всегда будет начинаться заново.
4. Функции вывода ломаных линий позволяют легко управлять внешним видом стыков.

Дуги

Концепция построения всех дуг в GDI основана на уравнении эллипса. Взглянув в наш горячо любимый (надеюсь, еще со средней школы) учебник геометрии, мы обнаружим нехитрое математическое выражение:

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

Немного подумав, можно прийти к потрясающему околонаучному выводу, что эллипс – это фигура, чьи габариты определяются размерами его осей (*a* и *b*), а место вывода – значениями *x* и *y*. Адаптировав уравнение эллипса к реалиям Windows GDI, получим усовершенствованную «двухэтажную» формулу:

$$\frac{(x-x_0)^2}{a^2}+\frac{(y-y_0)^2}{b^2}=1$$

Здесь (x_0, y_0) – координата центра эллипса. Вы еще не устали? Нет? Тогда продолжу. Исходя из предположения, что программист – весьма ленивое создание, в Microsoft справедливо постановили, что даже такое математическое выражение вгонит нашего брата в сон. Поэтому корпорация вновь пошла программистам навстречу и подготовила до такой степени элементарные функции GDI, что они позволят рисовать эллипсы и его дуги не только нам с вами, но даже *Homo Sapiens*, действительно ведущему свой славный род от обезьяны.

С практической точки зрения дуга представляет собой часть периметра эллипса, который, в свою очередь, описывается координатами ограничивающей эллипс прямоугольной области. Взгляните на параметры (предназначенных для рисования дуг) функций `Arc()` и `ArcTo()`:

Win GDI
```
Function Arc(DC:HDC; Left, Top, Right, Bottom,
             StartX, StartY, EndX, EndY : Integer):Boolean;
```

и

Win GDI
```
Function ArcTo(DC:HDC; Left, Top, Right, Bottom,
               StartX, StartY, EndX, EndY : Integer):Boolean;
```

Определив координаты левой верхней (`Left`, `Top`) и правой нижней вершин (`Right`, `Bottom`) прямоугольника, мы завершим полдела – укажем местоположение и размеры эллипса. Ответственность за внедрение эллипса внутрь прямоугольника возложена на GDI, об этом не стоит беспокоиться. Но раз мы рисуем дугу, то надо уточнить – какую часть периметра эллипса требуется отобразить. Для этого из центра эллипса мысленно проводим два луча (рис. 6.3).

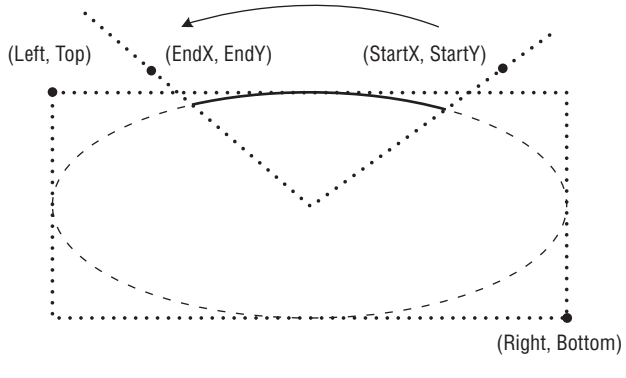


Рис. 6.3. Дуга эллипса – функция Arc( )

Дуга берет свое начало в месте пересечения периметра эллипса с лучом, проходящим через точку (StartX, StartY), вывод дуги завершается в месте пересечения периметра эллипса с лучом (EndX, EndY).

Единственная разница между функциями Arc() и ArcTo() в том, что Arc() не задействует текущую позицию пера, а ArcTo() перед рисованием дуги проводит линию из текущего положения пера в начальную точку дуги. По умолчанию дуга рисуется против часовой стрелки. В операционных системах, базирующихся на технологии NT, это направление настраиваемое. Для этого предназначена функция:

Win GDI
```
Function SetArcDirection(DC:HDC; ArcDirection : Integer) : Boolean;
```

Направлению против часовой стрелки соответствует константа AD_CLOCKWISE, по часовой – AD_COUNTERCLOCKWISE. Существует и просто информационная функция:

Win GDI
```
Function GetArcDirection(DC:HDC) : Integer;
```

Она расскажет о текущих настройках системы.

В ОС класса Windows NT предусмотрен способ определения дуги не только координатами точек, но и в градусах. Но в этом случае речь идет не об универсальном эллипсе, а о его частном случае – круге (рис. 6.4):

Win GDI
```
Function AngleArc(DC:HDC; X,Y :Integer;
                  Radius : Cardinal;
                  StartAngle, SweepAngle: Single) : Boolean;
```

Здесь (X,Y) – центр круга; Radius – радиус круга в логических единицах; StartAngle – величина угла от оси X до луча, определяющего начало дуги; SweepAngle – величина угла между лучами, определяющими начало и конец дуги. Функция чувствительна к текущей позиции пера.

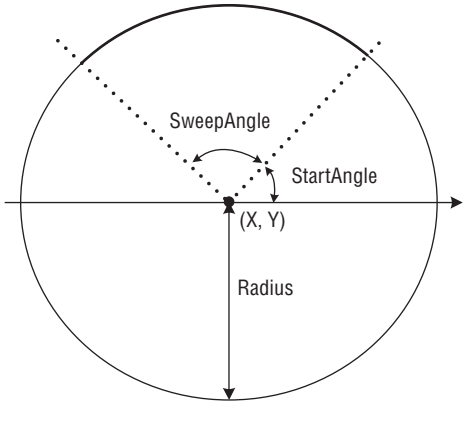


Рис. 6.4. Дуга эллипса – функция AngleArc( )

Кривые

Программы инженерной графики, системы автоматизированного проектирования, издательские пакеты, приложения мультимедиа, игры, одним словом – современное программное обеспечение не в состоянии решить свои задачи за счет ограниченного перечня функций, рисующих прямые линии и дуги. Попробуйте начертить коленчатый вал или контуры истребителя средствами пусть даже очень хорошей функции построения линий! Как говорится, нет ничего невозможного, и, в конце концов, у вас что-нибудь получится, но сколько уйдет на это времени! Понимая проблемы инженеров, Microsoft внедрила в GDI пару функций, позволяющих рисовать кривые. В основу идеи был положен математический аппарат, разработанный на стыке 1950–1960-х годов П. Безье (P. Bezier) и П. де Кастело (P. De Casteljau). Эти математики (кстати, независимо друг от друга) создали эффективный и одновременно простой способ описания кривых линий. Большая часть славы досталась ученому Безье – впоследствии эти кривые в честь него были названы *сплайнами Безье*, а имя господина Кастело, к сожалению, известно лишь специалистам.

Технически для описания кривой Безье достаточно определить всего четыре точки[1]. Это две точки концов кривой и две точки управления, они отклоняют кривую от центральной оси. Для рисования одной или нескольких кривых предназначена пара функций:

```
Function PolyBezier(DC:HDC; const Points; Count : Cardinal) :Boolean;
```

и


```
Function PolyBezierTo(DC:HDC; const Points; Count : Cardinal) :Boolean;
```

Здесь `Points` – массив, содержащий минимум 4 точки; `Count` – определяет, сколько точек описано в массиве. Имеет значение порядок следования точек в массиве. Первая точка характеризует начало кривой, вторая и третья – отклоняющие точки, четвертая точка – конец кривой и т. д. Единственное различие между функциями в том, что `PolyBezierTo()` чувствительна к текущей позиции пера, а `PolyBezier()` – нет.

Подготовим небольшой проект, который позволит нам «потрогать» кривую Безье руками (точнее, указателем мыши). В библиотеке на сайте издательства есть пример создания кривых Безье, написанный на Windows API, но сейчас (для ускорения нашей работы) мы разработаем проект на основе визуальной библиотеки компонентов. Для этого на главной форме (которая называется `frmMain`) расположите 4 метки (компоненты класса `TLabel`). Метки станут олицетворять опорные точки сплайна: точки начала и конца кривой и две отклоняющие точки. Переименуйте метки: `lb1`, `lb2`, `lb3` и `lb4`. В свойство `Caption`

[1] Сплайн, траектория которого определяется по четырем точкам, называется кубическим.

меток поместите любой символ, например «o». В секции частных объявлений опишите две переменные:

```
var OffsetX, OffsetY : byte;
```

Эти переменные позднее помогут нам расположить отклоняющие точки сплайна Безье ровно в центре меток. Заполнение переменных значениями и первоначальное позиционирование меток происходит в момент нажатия кнопки мыши над любым компонентом TLabel.

```
procedure TfrmMain.lb1MouseDown(Sender: TObject; Button: TMouseButton;
                                Shift: TShiftState; X, Y: Integer);
begin
With Sender as TLabel do
begin
  OffsetX:=(Width div 2);   Left := Left+X-OffsetX;
  OffsetY:=(Height div 2);  Top := Top+Y-OffsetY;
end;
end;
```

Назначьте это событие общим для всех четырех компонентов TLabel.

Настала очередь описать событие, связанное с перемещением опорных точек сплайна. Оно возникает в момент движения над компонентами-метками указателя мыши с удерживаемой в нажатом состоянии левой кнопкой.

```
procedure TfrmMain.lb1MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
if (ssLeft IN Shift) then //если нажата левая кнопка мыши
    With (Sender as TLabel) do
    begin
      Left:=Left+X-OffsetX; Top:=Top+Y-OffsetY; //изменяем координаты компонента
      frmMain.Repaint; //перерисовка формы
    end;
end;
```

Прежде чем мы перейдем к сердцу программы – событию перерисовки OnPaint(), – назначьте событие OnMouseMove() общим для всех четырех меток.

```
procedure TfrmMain.FormPaint(Sender: TObject);
var DC:HDC; Pen:HPEN; Brush:HBRUSH; LogBrush:TLogBrush;
    A : Array[1..4] of TPoint;
begin
 DC:=frmMain.Canvas.Handle;
 Brush:=CreateSolidBrush(RGB(255,255,255));
 FillRect(DC,frmMain.ClientRect,Brush);

    A[1].X:=lb1.Left+OffsetX; A[1].Y:=lb1.Top+OffsetY;
    A[2].X:=lb2.Left+OffsetX; A[2].Y:=lb2.Top+OffsetY;
    A[3].X:=lb3.Left+OffsetX; A[3].Y:=lb3.Top+OffsetY;
    A[4].X:=lb4.Left+OffsetX; A[4].Y:=lb4.Top+OffsetY;

    SelectObject(DC, GetStockObject(BLACK_PEN));
    PolyBezier(DC,A,4); {вывод сплайна Безье}
```

```
    LogBrush.lbColor:=RGB(100,100,100); LogBrush.lbStyle:=BS_SOLID;
    Pen:=ExtCreatePen(PS_COSMETIC OR PS_DOT,1,LogBrush,0,nil);
    SelectObject(DC, Pen);
    PolyLine(DC,A,4); {вывод вспомогательных линий}

    SelectObject(DC, GetStockObject(NULL_PEN));
    SelectObject(DC, GetStockObject(NULL_BRUSH));
    DeleteObject(Pen); DeleteObject(Brush);
end;
```

Программа закончена. Для повышения наглядности отклоняющие точки подчеркнуты ломаной линией, нарисованной штриховым пером. Управляя местоположением меток, вы сможете создать любую из простейших представленных на рис. 6.5 кривых.

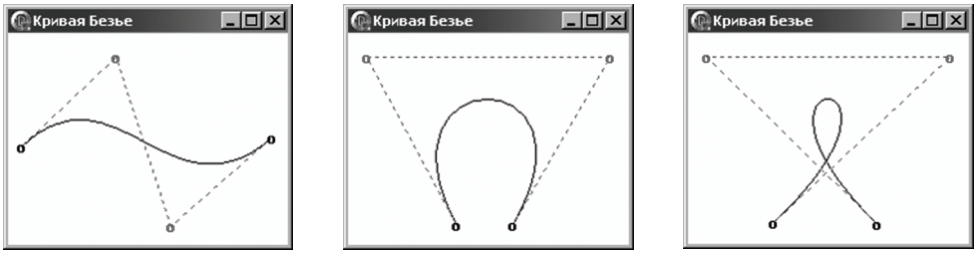


Рис. 6.5. Экранные снимки программы «Кривая Безье»

В библиотеке к книге среди примеров к главе 33 вы обнаружите еще одну программу, рисующую кривую Безье. Данная версия программы умеет выводить не только кривую Безье, но и канонический сплайн GDI+.

Попрактиковавшись со сплайнами, наблюдательный человек обязательно заметит две отличительные особенности кривых Безье:

- Во-первых, сплайн всегда находится внутри фигуры с вершинами в определяющих точках кривой.
- Во-вторых, прямая, проведенная из конечной в ближайшую отклоняющую точку, всегда представляет собой касательную к сплайну в его конечной точке.

К этому добавлю третью особенность сплайнов. При аффинных преобразованиях (см. главу 12 «Мировые координаты») кривые Безье преобразуются в кривые Безье. Это упрощает работу механизма GDI – ему достаточно трансформировать координаты определяющих точек из мировой системы координат в страничную и провести кривую.

*В GDI+ появилась еще одна разновидность кривых – **канонические сплайны**. Их отклоняющие точки не оттягивают сплайн от прямолинейного маршрута, а явным образом определяют его траекторию (это весьма удобно для пользователя). Кроме того, в GDI+ возможно управлять степенью напряжения сплайна (придавая ему некоторую схожесть с поведением натянутой струны).*

В группе ОС Windows NT/2000/XP/Vista для вывода группы ломаных линий и разъединенных сплайнов может быть применена функция:

```
Function PolyDraw(DC:HDC; const Points; const Types;
                  Count : Integer) : Boolean;
```

В сравнении с перечнем параметров функций PolyBezier() и PolyBezier() наша новая знакомая дополнена параметром Types (табл. 6.1). Это массив, каждый элемент которого характеризует соответствующий тип точки из массива Points.

Таблица 6.1. Константы типов точек функции PolyDraw( )

Константа	Описание
PT_MOVETO	Указывает на то, что это первая точка новой фигуры. Вывод фигуры начинается с новой позиции пера.
PT_LINETO	Проводимая к этой точке линия должна брать начало из текущей позиции пера. После этого текущая позиция обновляется. Константа может использоваться совместно с константой PT_CLOSEFIGURE. В этом случае заданная точка замыкается с последней точкой PT_MOVETO.
PT_BEZIERTO	В сплайне Безье данная точка играет роль отклоняющей либо последней точки. Константа может использоваться совместно с константой PT_CLOSEFIGURE. В этом случае заданная точка замыкается с последней точкой PT_MOVETO.
PT_CLOSEFIGURE	Флаг завершения фигуры, применяется только совместно с PT_BEZIERTO или PT_LINETO.

Для корректной работы функции PolyDraw() в массиве Type должен строго соблюдаться порядок следования констант. Так, при описании сплайна Безье после появления значения PT_MOVETO необходимо объявить подряд не менее трех констант PT_BEZIERTO. Первые две константы PT_BEZIERTO определяют отклоняющие точки кривой, третья – ее последнюю точку.

```
procedure GDI_PAINT(DC:HDC);
Var A:Array[0..7] of TPoint;
    T:Array[0..7] of byte;
begin
//кривая Безье
A[0].X:=150; A[0].Y:=150; T[0]:=PT_MOVETO;
A[1].X:=0;   A[1].Y:=10;  T[1]:=PT_BEZIERTO;
A[2].X:=200; A[2].Y:=10;  T[2]:=PT_BEZIERTO;
A[3].X:=50;  A[3].Y:=150; T[3]:=PT_BEZIERTO OR PT_CLOSEFIGURE;
//квадрат
A[4].X:=25;  A[4].Y:=25;  T[4]:=PT_MOVETO;
A[5].X:=175; A[5].Y:=25;  T[5]:=PT_LINETO;
A[6].X:=175; A[6].Y:=175; T[6]:=PT_LINETO;
A[7].X:=25;  A[7].Y:=175; T[7]:=PT_LINETO OR PT_CLOSEFIGURE;
PolyDraw(DC,A,T,8);
end;
```

Представленный фрагмент кода иллюстрирует способ применения функции `PolyDraw()`. За один вызов функции были воспроизведены две фигуры. Первая из них – кривая Безье (точки 0–3). Вторая фигура – изображение квадрата (точки 4–7). Снимок окна программы представлен на рис. 6.6.

При выводе группы кривых Безье разработчик часто сталкивается с проблемой обеспечения плавного перехода одной кривой в другую. Для решения этой проблемы следует запомнить простое правило. Для обеспечения гладкого стыка между кривыми на одной линии должны находиться три точки массива: вторая отклоняющая точка первого сплайна, конечная точка первого сплайна (она же является первой точкой второго сплайна) и первая отклоняющая точка второго сплайна. Такие точки называют *коллинеарными*.

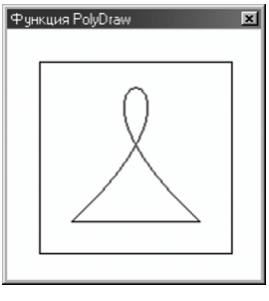


Рис. 6.6. Демонстрация возможностей PolyDraw( )

РЕЗЮМЕ

В рамках библиотеки GDI современных версий Windows описано около полутора десятка функций, специализирующихся на рисовании простейших графических примитивов – линий, ломаных, дуг и кривых. Наиболее универсальные из них – функции, позволяющие строить кубические сплайны Безье, с их помощью можно начертить контур фигуры любой степени сложности.

7

Простейшие геометрические фигуры и заливка областей

Угадайте, какая из геометрических фигур в Windows GDI самая заслуженная? Прав окажется тот, кто скажет: прямоугольник. Именно Его Величество Прямоугольник и есть центр мироздания всего графического интерфейса Windows. Своей победой над маститыми коллегами прямоугольник обязан только своей простоте. Судите сами – для описания прямоугольника достаточно определиться с координатами всего двух расположенных друг от друга по диагонали вершин, ни одной другой фигуре этого недостаточно. Поэтому все, даже очень вычурные, окна Windows и подавляющее число визуальных элементов управления берут начало от прямоугольника.

Исследование порядка работы с геометрическими фигурами мы начнем с изучения способа хранения координат прямоугольника. Мы уже выяснили, что для определения любой прямоугольной области в Windows GDI требуется указать как минимум две граничные точки. Как правило, это координаты левого верхнего и нижнего правого углов. Говорю «как правило», потому что иногда вместо координат правого нижнего угла используют значения ширины и высоты области (с таким способом описания мы столкнемся в GDI+)[1]. Для хранения координат прямоугольной области в модуле `types` среды Delphi объявлена запись `TRect`. В сравнении с классической структурой `Rect` из состава Windows API она несколько усовершенствована, так как допускает два способа назначения координат.

```
Type pRect = ^TRect;
TRect = packed record
 case Integer of
 0: (Left, Top, Right, Bottom : Longint);{координаты в стиле Windows GDI}
```

[1] В GDI+ координаты прямоугольника определяются местоположением верхней левой точки и протяженностью сторон. Поэтому прямоугольные структуры GDI и GDI+ несовместимы.

```
1: (TopLeft, BottomRight: TPoint);      {координаты в стиле Delphi}
end;
```

Не стоит волноваться, если при заполнении записи TRect *вы перепутаете местами точки и передадите координаты левой верхней точки в полях (*Right, Bottom*), а координаты правой нижней точки в полях (*Left, Top*). Операционная система все равно проявит сообразительность и нарисует правильный прямоугольник.*

Альтернативный способ определения координат базируется на применении структуры TPoint. Ее задачи более тривиальны, она описывает одну-единственную точку на поверхности устройства. В Delphi описание TPoint обнаружим в том же самом модуле типов types.

```
Type PPoint = ^TPoint;
TPoint = packed record
  X: Longint; //горизонтальная координата X
  Y: Longint; //вертикальная координата Y
end;
```

Специально для экономных натур разработана структура:

```
Type TSmallRect= packed record
    Left: SHORT;
    Top: SHORT;
    Right: SHORT;
    Bottom: SHORT;
end;
```

Ее отличие – в сокращенной размерности полей: вместо 4-байтных полей записи TRect здесь для определения координат применяются 2-байтные.

Функции определения прямоугольных областей

Для обслуживания структуры TRect в Windows GDI предусмотрен обширный перечень вспомогательных функций. Начнем с простейшей из них – функции обнуления координат области.

Win GDI

```
Function SetRectEmpty(var Rect : TRect):Boolean;
```

После того как запись Rect освободится из «объятий» функции SetRectEmpty(), во всех ее полях окажутся нулевые значения. В помощь SetRectEmpty() приставлен тайный наблюдатель:

Win GDI

```
Function IsRectEmpty(const Rect : TRect):Boolean;
```

Функция проверяет, пуста ли область Rect. Положительное решение принимается в случае совпадения координат левой верхней и правой нижней точек записи. В этом случае результатом выполнения функции станет значение true.

Инициализацию структуры новыми аргументами выполняет функция:

Win GDI

```
Function SetRect(var Rect : TRect;
                 Left, Top, Right, Bottom : Integer):boolean;
```

Функция заполнит запись Rect значениями точек (Left, Top) и (Right, Bottom). Конечно, разрешено заполнять структуру данными по старинке (поочередно передавая значения в каждое из полей); это особо не отразится на скорости выполнения откомпилированного приложения, но применение SetRect() в первую очередь благотворно сказывается на читабельности исходного кода наших программ.

Очередная сервисная функция предназначена для проверки факта совпадения прямоугольных областей:

Win GDI
```
Function EqualRect(const Rect1 : TRect; const Rect2 : TRect ) : Boolean;
```

Она сравнивает координаты Rect1 и Rect2 и возвращает true в случае их идентичности.

Для клонирования прямоугольных областей стоит воспользоваться функцией:

Win GDI
```
Function CopyRect(var DstRect:TRect; const SrcRect : TRect) : Boolean;
```

Функция предназначена для копирования координат из SrcRect в DstRect.

Языки C и Pascal позволяют передавать данные между записями с помощью элементарной операции присваивания, так что обращение к CopyRect() *вряд ли потребуется.*

Для оперативного изменения геометрических размеров прямоугольника предназначена функция:

Win GDI
```
Function InflateRect(var Rect : TRect; X, Y : Integer) : Boolean;
```

Параметры X и Y предназначены для организации приращения (положительные значения) или уменьшения (отрицательные значения) ширины и высоты прямоугольной области, переданной в параметр Rect. Способ применения функции представлен в следующем листинге:

```
Procedure GDI_PAINT(DC:hDC);
var X:Byte;
    Rect:TRect;
begin
SetRect(Rect, 10, 10, 210, 210);

for x:=4 downto 0 do
 begin
 InflateRect(Rect, -10, -10);
 FillRect(DC, Rect, GetStockObject(X));
end;
end;
```

В цикле мы перебираем стандартные кисти (см. табл. 4.1) Windows GDI и с их помощью закрашиваем прямоугольную область Rect. Заметьте, что функция одновременно воздействует на обе точки прямоугольника. Поэтому с каждой итерацией цикла протяженность области сокращается (как по горизон-

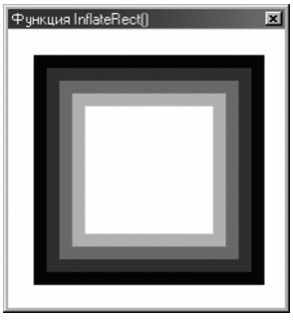


Рис. 7.1. Демонстрация возможностей функции InflateRect( )

тали, так и по вертикали). В результате выполнения кода мы получим окно (рис. 7.1) с изображениями пяти вложенных друг в друга квадратов.

Сдвиг прямоугольной области в любом из направлений по горизонтальной и вертикальной осям обеспечивает функция:

Win GDI
```
Function OffsetRect(var Rect : TRect; X,Y : Integer):Boolean;
```

Направление и величина сдвига определяется параметрами `X` и `Y`. На габариты области функция не влияет.

Проверку вхождения в прямоугольную область `Rect` точки с координатами `P` осуществит функция:

Win GDI
```
Function PtInRect(const Rect : TRect; P : TPoint) : Boolean;
```

Если это действительно так, то функция возвращает значение `true`. При этом, если точка `P` лежит точно на линии нижней или правой стороны прямоугольника, функция посчитает, что точка не входит в область.

Если требуется провести проверку на факт пересечения двух прямоугольных областей, воспользуйтесь функцией:

Win GDI
```
Function IntersectRect(var DstRect : TRect;
                       const SrcRect1:TRect;
                       const SrcRect2:TRect) : Boolean;
```

Функция определяет область пересечения двух прямоугольных областей `SrcRect1` и `SrcRect1` и помещает координаты пересечения в параметр `DstRect`. Если прямоугольники не соприкасаются, то функция возвратит значение `false` и обнулит поля структуры `DstRect`. В следующих строках кода представлен пример анализа двух прямоугольников.

```
Procedure GDI_PAINT(DC:hDC);
var  DstRect,SrcRect1,SrcRect2:TRect;
begin
 SetRect(SrcRect1,10,10,100,100);
 FillRect(DC,SrcRect1,GetStockObject(GRAY_BRUSH));
```

```
 SetRect(SrcRect2,55,55,145,145);
 FillRect(DC,SrcRect2,GetStockObject(GRAY_BRUSH));

 IntersectRect(DstRect, SrcRect1,SrcRect2);
 FillRect(DC,DstRect,GetStockObject(DKGRAY_BRUSH));
end;
```

Область пересечения закрашена кистью темно-серого оттенка (рис. 7.2).

Для объединения двух областей в одно целое предназначена функция:

```
Function UnionRect(var DstRect : TRect;
                   const SrcRect1:TRect;
                   const SrcRect2:TRect) : Boolean;
```

Здесь SrcRect1 и SrcRect2 — объединяемые прямоугольники, результат объединения мы обнаружим в параметре DstRect. Алгоритм работы функции таков: сравниваются координаты всех точек и выявляется, какая из них расположена выше и левее и какая из точек расположена ниже и правее остальных точек Победительницы конкурса заполняют структуру DstRect, а проигравшие в слезах расходятся по домам. Демонстрация возможностей функции представлена в следующем фрагменте кода:

```
Procedure GDI_PAINT(DC:hDC);
var DstRect, SrcRect1,SrcRect2:TRect;
begin
 SetRect(SrcRect1,10,10,80,30);
 SetRect(SrcRect2,80,100,175,145);

 UnionRect( DstRect, SrcRect1,SrcRect2);
 FillRect(DC,DstRect,GetStockObject(DKGRAY_BRUSH));

 FillRect(DC,SrcRect1,GetStockObject(LTGRAY_BRUSH));
 TextOut(DC, SrcRect1.Left,SrcRect1.Top,'SrcRect1',8);
 FillRect(DC, SrcRect2,GetStockObject(LTGRAY_BRUSH));
 TextOut(DC, SrcRect2.Left,SrcRect2.Top,'SrcRect2',8);
end;
```

Результат работы кода представлен на рис. 7.3.

После того как мы научились складывать прямоугольные области, логично было бы научиться их вычитать, благо для этого создана отдельная функция:

```
Function SubtractRect(var DstRect : TRect;
                      const SrcRect1:TRect;
                      const SrcRect2:TRect) : Boolean;
```

Функция вычитает прямоугольник SrcRect2 из прямоугольника SrcRect1. Результаты операции передаются в параметр DstRect.

```
Procedure GDI_PAINT(DC:hDC);
var DstRect, SrcRect1,SrcRect2:TRect;
begin
SetRect(SrcRect1,10,50,100,100);
SetRect(SrcRect2,50,10,150,150);
```

```
SubtractRect(DstRect, SrcRect1,SrcRect2);
{Результат вычисления (10,50,50,100)}
FillRect(DC,SrcRect1,GetStockObject(GRAY_BRUSH));
FillRect(DC,SrcRect2,GetStockObject(LTGRAY_BRUSH));
FillRect(DC,DstRect,GetStockObject(DKGRAY_BRUSH));
end;
```

Результат вычитания закрашен темно-серой кистью (рис. 7.4).

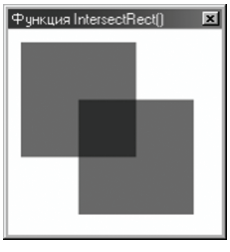


Рис. 7.2. Пересечение областей

Рис. 7.3. Объединение областей

Рис. 7.4. Вычитание областей

Заливка прямоугольной области

Существует достаточно много алгоритмов закрашивания замкнутых областей. Наиболее простой из них заключается в рисовании друг под другом горизонтальных линий от левой до правой границы закрашиваемой фигуры. С точки зрения практики при заливке областей кистью наибольшей популярностью пользуется функция `FillRect()`. Она уже не раз встречалась в наших примерах:

Win GDI

```
Function FillRect(DC : hDC;
                  const Rect : TRect; Brush : HBRUSH):Boolean;
```

Кроме дескриптора контекста устройства `DC` в нее требуется направить координаты прямоугольной области `Rect` и дескриптор кисти `Brush`. Главное достоинство `FillRect()` – высокая скорость работы. Быстрее `FillRect()` работает только функция `PatBlt()`, но с ней мы познакомимся немножко позднее.

Предусмотрена функция инвертирования цвета прямоугольной области:

Win GDI

```
Function InvertRect(DC : hDC; const Rect : TRect):Boolean;
```

Функция `InvertRect()` к подбору цветов подходит с нигилистических позиций: ее кредо – операция `NOT`. Встретив на своем пути нулевой бит, функция превратит его в единицу. Не будет пощады и для единицы, она без капли сожаления трансформируется в ноль. С точки зрения цвета это выглядит следующим образом: если перед вызовом функции область была окрашена в черное, то мы получим белую заливку, вместо белого цвета область зальется траурным черным, красный цвет заменится на голубой и т. д. Вы можете проверить это утверждение, повторив следующий фрагмент кода:

```
Procedure GDI_PAINT(DC:hDC;ClientRect:TRect);
Var Brush:HBrush;
    Rect:TRect;
begin
 Brush:=CreateHatchBrush(HS_DIAGCROSS,RGB(0,0,0));
 FillRect(DC, ClientRect,BRUSH);
 Rect:=PAINTSTRUCT.rcPaint;
 Rect.Left:=Rect.Right div 2;
 InvertRect(DC,Rect);
 SelectObject(DC,GetStockObject(NULL_BRUSH));
 DeleteObject(Brush);
end;
```

Сначала мы заливаем всю площадь клиентской области окна кистью с диагональной штриховкой черного цвета. Затем инвертируем правую часть окна (рис. 7.5).

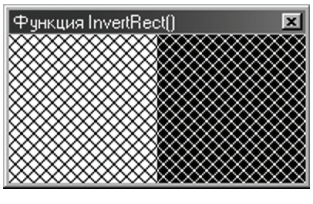


Рис. 7.5. Инвертирование области

Необычны возможности функции `FrameRect()`:

Win GDI

```
Function FrameRect(DC : hDC;
                   const Rect : TRect; Brush : HBRUSH):Boolean;
```

Она рисует рамку области, применяя для этого не перо, а кисть. Толщина рамки равна одной логической единице. Внутренняя область прямоугольника в операции не задействуется.

В недрах GDI прячется функция, позволяющая закрасить поверхность графического устройства, идентифицируемого дескриптором `DC`, тем же рисунком, что используется в качестве фона на рабочем столе компьютера.

Win GDI

```
function PaintDesktop(DC: hDC): BOOL;
```

Интеллектуальная заливка области

Что делать, если при закраске поверхности кистью нельзя выйти за пределы области нестандартной (читай непрямоугольной) формы? В этом случае функция `FillRect()` и ее ближайшие сподвижники окажутся не самыми лучшими помощниками. Тут им на замену выходит функция с зачатками интеллекта:

Win GDI

```
function ExtFloodFill(DC: hDC; X, Y: Integer;
                      Color: COLORREF; FillType: UINT): Boolean;
```

Здесь (X,Y) – координаты точки, из которой начнется заливка области; Color – цвет границы или области, за которую нам нельзя выходить. Ключевой параметр функции – FillType, именно он и определяет алгоритм заливки. Параметр может принимать всего два значения: FLOODFILLBORDER или FLOODFILLSURFACE. Передав в FillType константу FLOODFILLBORDER, мы укажем системе, что закраске подлежит область, ограниченная сплошной линией с цветом Color. Если мы применим константу FLOODFILLSURFACE, то текущая кисть затронет только область с цветом Color.

Ниже предложена процедура, демонстрирующая возможности функции по частичной закраске клиентской области окна. На вход процедуры передаем дескриптор контекста устройства, координаты точки (X,Y), с которой начнем закраску области, и цвет бордюра-ограничителя.

```
Procedure FillArea(DC:HDC; X,Y:Integer; BorderColor:COLORREF);
var Brush:HBrush;
begin
Brush:=CreateHatchBrush(HS_BDIAGONAL,$000000);
SelectObject(DC,Brush);
ExtFloodFill(DC,X,Y,BorderColor,FLOODFILLBORDER);
SelectObject(DC,GetStockObject(NULL_BRUSH));
DeleteObject(Brush);
end;
```

В результате выполнения кода мы получим результат, представленный на рис. 7.6. В клиентской области окна изображены несколько простейших геометрических фигур, нарисованных пером одного и того же цвета. Пользователю достаточно щелкнуть левой кнопкой мыши по той части рисунка, которую требуется закрасить, а все остальное сделает наша процедура...

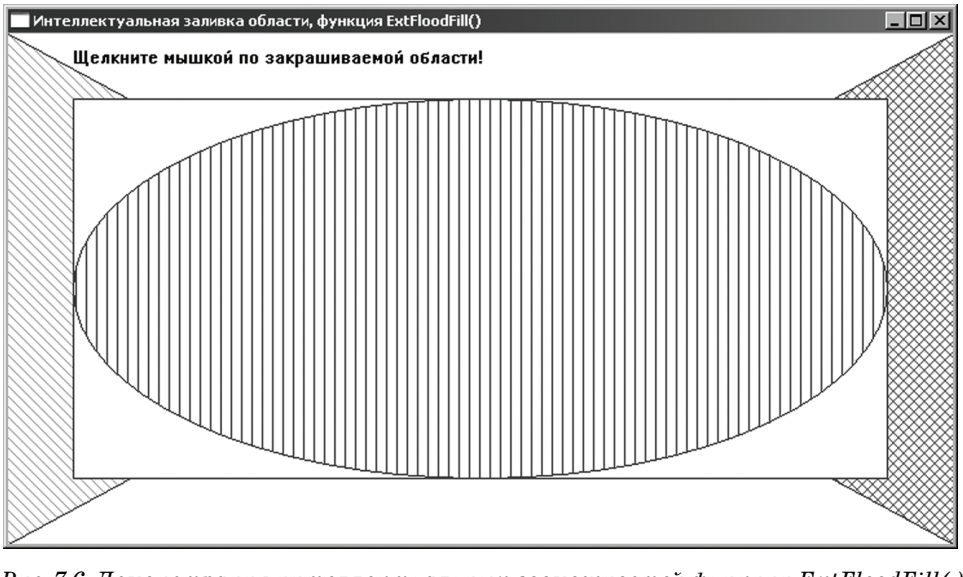


Рис. 7.6. Демонстрация интеллектуальных возможностей функции ExtFloodFill( )

Простейшие геометрические фигуры

На рисовании замкнутых областей специализируется достаточно много функций. В перечень их возможностей входит вывод таких простейших геометрических фигур, как прямоугольник, многоугольники произвольной формы, эллипс, а также сектора и сегменты эллипса. При выводе простейших геометрических фигур на поверхности устройства контекст использует не только перо (для черчения контура фигуры), но и кисть – для заливки замкнутой области. Если контур не нужен, то выберите в контекст перо-пустышку (`NULL_PEN`), почти такое же решение подходит и к заливке – если нет необходимости раскрашивать объект, то подключите к контексту кисть `NULL_BRUSH`.

Прямоугольники и их производные

Для вывода прямоугольника на поверхности графического устройства требуется определиться с координатами его левого верхнего (`Left`, `Top`) и правого нижнего углов (`Right`, `Bottom`) и обратиться к функции:

```
Function Rectangle(DC:hDC;
                   Left, Top, Right, Bottom : Integer):Boolean;
```

Утонченным натурам понравится функция, позволяющая воспроизвести прямоугольник со скругленными углами, которая дополнена еще парой параметров.

```
Function RoundRect(DC:hDC; Left, Top, Right, Bottom : Integer;
                   Width, Height : Integer):Boolean;
```

Здесь `Width` и `Height` определяют ширину и высоту эллипса, контур которого подменит углы прямоугольника. Скруглить углы можно столь сильно, что в итоге прямоугольник превратится в эллипс. И наоборот, передав в последние два параметра нулевые значения, мы получим прямо противоположенный эффект – обычный прямоугольник.

К счастью, нам не потребуется циркуль и твердая рука, чтобы заставить контекст изобразить эллипс. Windows GDI удовлетворится координатами прямоугольной области, в которую надо вписать эллипс, и все остальное за нас сделает функция:

```
Function Ellipse(DC:hDC; Left,Top,Right,Bottom : integer):Boolean;
```

На рис. 7.7 представлен графический смысл функций `Rectangle()`, `RoundRect()` и `Ellipse()`.

Сколько вам известно способов рисования круга? Если вы дружите с геометрией, то по крайней мере два. Во-первых, зная величину радиуса и координаты центра окружности, мы можем выразить координаты точек контура из несложной формулы: $R^2=(X-X_0)^2+(Y-Y_0)^2$. Во-вторых, допустимо воспользоваться синусом и косинусом, тогда X=R*Cos(Angle) и Y=R*Sin(Angle). Начиная с Windows NT 4.0, при рисовании окружности или ее сегмента (рис. 7.8) мы можем не вдаваться в математические тонкости, а вместо этого положиться на функцию `AngleArc()`:

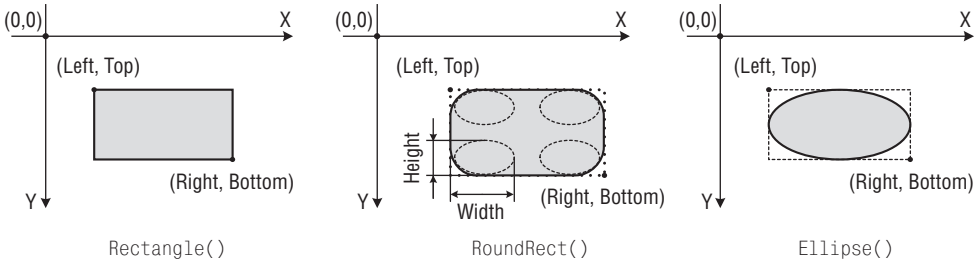


Рис. 7.7. Определение координат прямоугольника, прямоугольника со скругленными углами и эллипса

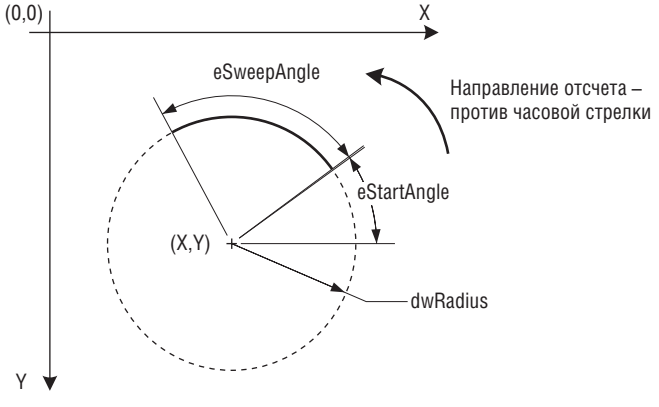


Рис. 7.8. Вывод сегмента окружности

Win GDI

```
function AngleArc(DC: hDC; X,Y: integer;
                  dwRadius:  Longword;
                  eStartAngle, eSweepAngle : real):boolean;
```

Здесь `X` и `Y` – координаты центра окружности; `dwRadius` – радиус окружности в логических единицах; `eStartAngle` – угол (в градусах) относительно оси `X`, определяющий начальную точку сектора; `eSweepAngle` – угол раскрытия сектора. Отсчет градусов ведется против часовой стрелки. Функция рисует только контур фигуры, применяя для этого текущее перо. На выбранную в контекст кисть функция не реагирует, поэтому заливка области не производится.

Чем сложнее задача, тем больше параметров надо подготовить. Например, для определения сектора круга или эллипса (рис. 7.9) кроме классического для всех функций дескриптора контекста устройства потребуется еще 8 аргументов:

Win GDI

```
Function Pie(DC:hDC; Left, Top ,Right ,Bottom,
             X1,Y1,X2,Y2 : integer):Boolean;
```

При исполнении своих обязанностей функция `Pie()` также опирается на координаты прямоугольной области, в которую будет вписан эллипс. Для определения сектора (рис. 7.9) мысленно проведите из центра эллипса два луча и выберите на каждом из них по точке (`X1`,`Y1`) и (`X2`,`Y2`). Позднее эти лучи превратятся в стороны сектора. В точках пересечения лучей с воображаемым периметром эллипса они соединятся дугой.

```
SetArcDirection(DC,AD_CLOCKWISE); {по часовой стрелке}
Pie(dc,10,10,110,110, 110,50,50,10,); {координаты сектора}
```

По умолчанию дуга направлена против часовой стрелки, в системах Windows NT это направление разрешено изменить, для этого потребуется вызов функции `SetArcDirection()`.

Для изображения хорды потребуется функция:

Win GDI

```
Function Chord(DC:hDC; Left, Top, Right, Bottom,
                X1,Y1,X2,Y2 : integer) : Boolean;
```

Перечень и назначение параметров этой функции аналогичны аргументам функции, рисующей сектор. Единственное отличие в результате – в хорду включается область между частью периметра эллипса и соединяющей две точки периметра секущей (рис. 7.10).

При выводе сегмента и хорды функции `Pie()` и `Chord()` используют текущее перо и кисть.

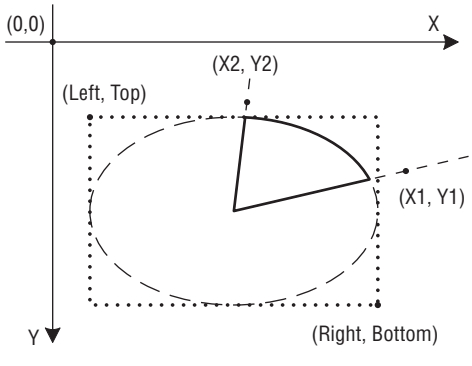


Рис. 7.9. Сектор эллипса

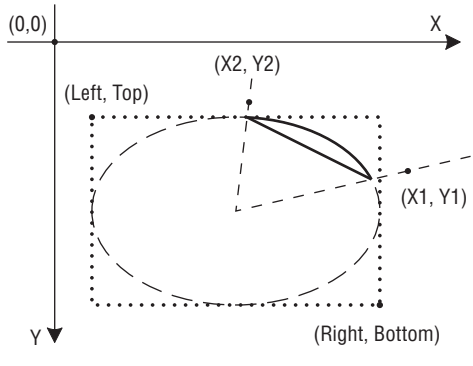


Рис. 7.10. Хорда

Многоугольники

Для построения более солидной фигуры, чем банальный прямоугольник, неплохую услугу способна оказать функция:

Win GDI

```
Function Polygon(DC:hDC; var Points; Count : Integer);
```

Функция нарисует многоугольник, каждая из вершин которого описана в массиве `Points`. Последний параметр метода ограничит число вершин много-

угольника. Фигура всегда получается замкнутой – функция соединит последнюю точку с первой. Характеристики соединения определяются настройками текущего пера.

Для рисования группы многоугольников за один вызов предназначена функция:

Win GDI

```
Function PolyPolygon(DC:hDC; var Points; var PolyCounts; Count :
Integer);
```

Перечень всех вершин многоугольников передается в массив точек `Points`. Третий параметр функции также представляет собой массив, но на этот раз целых чисел. В нем указывается количество точек, принадлежащих каждому многоугольнику. Например, если с помощью `PolyPolygon()` рисуются треугольник и квадрат, то в первом элементе массива `PolyCounts` должно оказаться значение 3, а во втором – 4. Последний параметр функции определяет, сколько всего многоугольников требуется нарисовать. Все фигуры рисуются текущим пером и кистью. Как видите, идея рисования многоугольников весьма схожа с идеей вывода ломаных линий.

```
Procedure GDI_PAINT(DC:hDC);
var
  p:Array[0..9] of TPoint;
  PolyCounts : Array[0..1] of integer;
begin
SelectObject(DC,GetStockObject(GRAY_BRUSH));
p[0].X:=10; p[0].Y:=10; p[1].X:=10; p[1].Y:=100; p[2].X:=100; p[2].Y:=100;
PolyCounts[0]:=3;
p[3].X:=60;  p[3].Y:=20;
p[4].X:=165; p[4].Y:=20;
p[5].X:=135; p[5].Y:=75;
p[6].X:=35;  p[6].Y:=75;
PolyCounts[1]:=4;

SetPolyFillMode(DC,WINDING);
PolyPolygon(DC,p,PolyCounts,2);
end;
```

Результаты работы кода представлены на рис. 7.11.

В рамках примера нам встретилась новая функция `SetPolyFillMode()`, с ее помощью программист определяет режим заполнения областей со сложной формой.

Рис. 7.11. Демонстрация PolyPolyline( )

Режим заполнения сложной области

При выводе сложных многоугольников, в особенности многоугольников с самопересечением сторон, у GDI могут возникнуть некоторые сложности с определением принадлежности точки к внутренней области фигуры. В результате при заливке фигуры кистью какие-то ее части могут остаться незакрашенными. Когда-нибудь вы столкнетесь с подобной проблемой, поэтому уже сейчас стоит познакомиться с функцией SetPolyFillMode(), определяющей режим заполнения многоугольников:

Win GDI
```
Function SetPolyFillMode(DC : hDC; FillMode : Integer) : Integer;
```

От выбранного режима зависит, будет данная область региона закрашена или нет. Подобную задачу контекст решает одним из двух способов: в соответствии с режимом ALTERNATE или WINDING (табл. 7.1).

Таблица 7.1. Режимы закраски точки функции SetPolyFillMode( )

Значение	Описание
ALTERNATE	Режим по умолчанию. Решение о том, что точка принадлежит области, принимается следующим образом. Из контролируемой точки в положительном направлении оси X проводится воображаемый луч. Если с контуром региона луч пересечется нечетное число раз, то система полагает, что точка находится внутри области.
WINDING	Из контролируемой точки в положительном направлении оси X проводится воображаемый луч. При пересечении луча с контуром области, направленным по часовой стрелке, счетчик пересечений увеличивается, а при пересечении с контуром, направленным против часовой стрелки, – уменьшается. Если результирующее значение отлично от нуля, то принимается решение, что точка принадлежит области.

О том, какой режим заливки в данный момент установлен в контексте, позволяет судить функция:

Win GDI
```
Function GetPolyFillMode(DC : hDC) : Integer;
```

Функция исследует соответствующий атрибут контекста и возвращает соответствующую константу из табл. 7.1.

Имитация элементов управления

В состав Windows GDI входит набор функций, нацеленных на упрощение жизни программиста, разрабатывающего внешний вид сложных элементов пользовательского интерфейса.

При создании своих собственных версий элементов управления или при имитации их внешнего вида пригодится функция прорисовки границ прямоугольной области.

Win GDI
```
Function DrawEdge(DC : hDC; var Rct : TRect;
                  Edge : cardinal; grfFlags :cardinal) : boolean;
```

В качестве первого параметра используется дескриптор контекста устройства, на поверхности которого мы нарисуем границы. В переменную Rct передаем координаты прямоугольной области. Третий параметр определяет вид бордюра (табл. 7.2).

Таблица 7.2. Константы вида границ функции DrawEdge( )

Константа	Описание
BDR_RAISEDINNER	Выпуклая внутренняя граница
BDR_SUNKENINNER	Впалая внутренняя граница
BDR_RAISEDOUTER	Выпуклая внешняя граница
BDR_SUNKENOUTER	Впалая внешняя граница
EDGE_BUMP	Комбинация BDR_RAISEDOUTER и BDR_SUNKENINNER
EDGE_ETCHED	Комбинация BDR_SUNKENOUTER и BDR_RAISEDINNER
EDGE_RAISED	Комбинация BDR_RAISEDOUTER и BDR_RAISEDINNER
EDGE_SUNKEN	Комбинация BDR_SUNKENOUTER и BDR_SUNKENINNER

Наконец, почти 20 флагов (не считая их комбинаций) могут быть направлены в последний, четвертый параметр функции DrawEdge(). Флаги определяют, какие именно границы подлежат выводу на экран. Имена флагов раскрывают их назначение, поэтому я их просто перечислю: BF_ADJUST, BF_BOTTOM, BF_BOTTOMLEFT, BF_BOTTOMRIGHT, BF_DIAGONAL, BF_DIAGONAL_ENDBOTTOMLEFT, BF_DIAGONAL_ENDBOTTOMRIGHT, BF_DIAGONAL_ENDTOPLEFT, BF_DIAGONAL_ENDTOPRIGHT, BF_FLAT, BF_LEFT, BF_MIDDLE, BF_MONO, BF_RECT, BF_RIGHT, BF_SOFT, BF_TOP, BF_TOPLEFT и BF_TOPRIGHT. Осталось продемонстрировать возможности функции.

```
Procedure  GDI_PAINT(DC:hDC);
var rct:TRect;
begin
 SetRect(rct,5,5,45,45); DrawEdge(DC,rct,EDGE_RAISED,BF_RECT or BF_SOFT);
 OffsetRect(rct,45,0);   DrawEdge(DC,rct,EDGE_ETCHED,BF_RECT);
 OffsetRect(rct,45,0);   DrawEdge(DC,rct,EDGE_SUNKEN,BF_RECT);
 OffsetRect(rct,45,0);   DrawEdge(DC,rct,EDGE_BUMP,BF_RECT);

 SetRect(rct,5,50,45,90); DrawEdge(DC,rct,EDGE_RAISED,BF_BOTTOMRIGHT);
 OffsetRect(rct,45,0);    DrawEdge(DC,rct,EDGE_RAISED,BF_DIAGONAL_ENDBOTTOMLEFT);
 OffsetRect(rct,45,0);    DrawEdge(DC,rct,EDGE_ETCHED,BF_DIAGONAL_ENDTOPRIGHT);
 OffsetRect(rct,45,0);    DrawEdge(DC,rct,EDGE_SUNKEN,BF_BOTTOMLEFT);
EndPaint(Window,PAINTSTRUCT);
end;
```

Результаты кропотливого труда по рисованию границ предложены на рис. 7.12.

Весьма интересными возможностями порадует программиста функция:

Win GDI

```
Function DrawFrameControl(DC hDC; Rect : TRect; uType: Cardinal;
                          uState : Cardinal) : boolean;
```

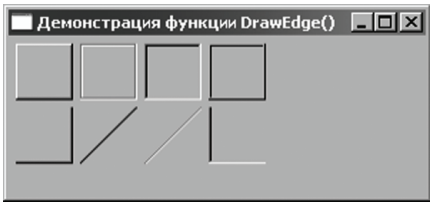


Рис. 7.12. Варианты начертания границ элемента управления

Наша новая знакомая может найти применение в ситуациях, когда нам требуется нарисовать элемент управления (подчеркиваю – именно нарисовать, а не создать). Для этого программист должен определиться с контекстом устройства `DC` и координатами прямоугольной области, в которые будет вписан «эрзац-элемент». Последние два параметра определяют тип элемента `uType` (табл. 7.3) и особенности его отображения `uState` (табл. 7.4).

Таблица 7.3. Возможные значения параметра uType

Значение	Описание
`DFC_BUTTON`	Обычная кнопка
`DFC_CAPTION`	Кнопки заголовка окна
`DFC_MENU`	Полоса меню
`DFC_POPUPMENU`	Элемент всплывающего меню
`DFC_SCROLL`	Полоса прокрутки

Перечень передаваемых в последний параметр значений метода зависит от того, какой именно элемент управления мы планируем отобразить. Список констант весьма обширен, поэтому мы ограничимся рассмотрением одного частного случая – прорисовки кнопки (`uType=DFC_Button`). С остальными вариантами познакомьтесь самостоятельно в SDK.

Таблица 7.4. Особенности вывода изображения кнопки (uType=DFC_BUTTON)

Значение uState	Описание
`DFCS_BUTTON3STATE`	Кнопка с тремя состояниями (флажок)
`DFCS_BUTTONCHECK`	Флажок
`DFCS_BUTTONPUSH`	Утопленная кнопка
`DFCS_BUTTONRADIO`	Переключатель
`DFCS_BUTTONRADIOIMAGE`	Изображение для переключателя
`DFCS_BUTTONRADIOMASK`	Маска для переключателя

С вопросом практического применения функции-имитатора никаких сложностей возникнуть не должно. Ниже представлен фрагмент кода, в результате которого окно программы покрывается изображениями кнопок, переключателей, флажков и системных иконок (рис. 7.13).

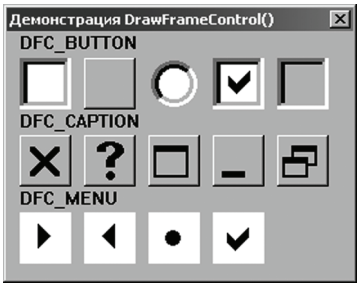


Рис. 7.13. Имитация элементов управления

```
Procedure  GDI_PAINT(DC:hDC);
var Rct:TRect;
begin
TextOut(DC,10,0,pCHAR('DFC_BUTTON'),10);
SetRect(Rct,10,20,50,60); DrawFrameControl(DC,Rct,DFC_BUTTON,DFCS_BUTTON3STATE);
OffsetRect(Rct,50,0);  DrawFrameControl(DC,Rct,DFC_BUTTON,DFCS_BUTTONPUSH);
OffsetRect(Rct,50,0);  DrawFrameControl(DC,Rct,DFC_BUTTON,DFCS_BUTTONRADIO);
OffsetRect(Rct,50,0);  DrawFrameControl(DC,Rct,DFC_BUTTON,DFCS_BUTTONCHECK or
                                                          DFCS_CHECKED);
OffsetRect(Rct,50,0); DrawFrameControl(DC,Rct,DFC_BUTTON,DFCS_INACTIVE);

TextOut(DC,10,61,pCHAR('DFC_CAPTION'),11);
SetRect(Rct,10,80,50,120);
DrawFrameControl(DC,Rct,DFC_CAPTION,DFCS_CAPTIONCLOSE);
OffsetRect(Rct,50,0); DrawFrameControl(DC,Rct,DFC_CAPTION,DFCS_CAPTIONHELP);
OffsetRect(Rct,50,0);  DrawFrameControl(DC,Rct,DFC_CAPTION,DFCS_CAPTIONMAX);
OffsetRect(Rct,50,0);  DrawFrameControl(DC,Rct,DFC_CAPTION,DFCS_CAPTIONMIN);
OffsetRect(Rct,50,0);  DrawFrameControl(DC,Rct,DFC_CAPTION,DFCS_CAPTIONRESTORE);

TextOut(DC,10,121,pCHAR('DFC_MENU'),8);
SetRect(Rct,10,140,50,180); DrawFrameControl(DC,Rct,DFC_MENU,DFCS_MENUARROW);
OffsetRect(Rct,50,0); DrawFrameControl(DC,Rct,DFC_MENU,DFCS_MENUARROWRIGHT);
OffsetRect(Rct,50,0); DrawFrameControl(DC,Rct,DFC_MENU,DFCS_MENUBULLET);
OffsetRect(Rct,50,0);  DrawFrameControl(DC,Rct,DFC_MENU,DFCS_MENUCHECK);
EndPaint(Window,PAINTSTRUCT);
end;
```

Имитация фокуса ввода

Очередная функция из коллекции Windows API по нашей просьбе «поддела-ет» фокус ввода.

Win GDI

```
Function DrawFocusRect(DC : hDC; Rct : TRect):boolean;
```

Для этого ей достаточно знать дескриптор контекста устройства `DC` и координаты прямоугольника `Rct`, в пределах которого имитируется фокус ввода. На этом особенности `DrawFocusRect()` не заканчиваются. Для снятия фокуса необходим повторный вызов `DrawFocusRect()`.

Вывод заголовка окна

Если вам вдруг потребуется нарисовать заголовок окна, совсем не стоит изобретать велосипед, гораздо проще воспользоваться функцией:

Win GDI

```
function DrawCaption(Wnd: HWND; DC: hDC; const Rct: TRect;
                     Flags: UINT): BOOL;
```

Функция потребует от программиста заполнить четыре параметра. Во-первых, функции необходим дескриптор окна `Wnd`, содержащего текст исходного заголовка и иконку окна. Во-вторых, функции надо передать дескриптор поверхности `DC`, на которой мы планируем нарисовать заголовок. В-третьих, потребуются координаты прямоугольной области `Rct` будущего заголовка. В-четвертых, перечень флагов `Flags`, определяющих особенности вывода заголовка (табл. 7.5). В случае успешного выполнения функция возвратит `true`.

Таблица 7.5. Флаги функции DrawCaption( )

Флаг	Описание
DC_ACTIVE	При выводе изображения функция воспользуется цветом активного заголовка.
DC_BUTTONS	Нарисовать стандартные кнопки заголовка окна (только для Windows XP и Vista).
DC_GRADIENT	Градиентный вывод заголовка. Не поддерживается в Windows 95.
DC_ICON	При выводе заголовка будет нарисована иконка окна.
DC_INBUTTON	Заголовок выводится цветом поверхности кнопки.
DC_SMALLCAP	Функция нарисует уменьшенный вариант заголовка.
DC_TEXT	Выводить текст.

Ниже представлен фрагмент листинга, демонстрирующий порядок вызова функции `DrawCaption()`; результат работы программы представлен на рис. 7.14.

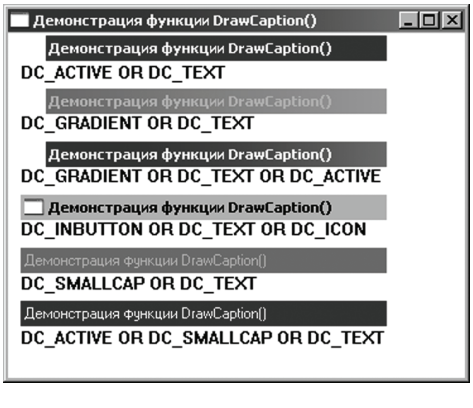


Рис. 7.14. Вывод заголовков окна с помощью функции DrawCaption( )

```
Procedure  GDI_PAINT(DC:hDC; Wnd:HWND);
var rct:TRect;
BEGIN
   SetRect(Rct, 10, 2, 300, 22);
   DrawCaption(Wnd, dc, rct, DC_ACTIVE OR DC_TEXT);
   TextOut(DC, 10, 23, pCHAR('DC_ACTIVE OR DC_TEXT'), 20);
   //…
   EndPaint(Window, PAINTSTRUCT);
END;
```

Пассивное состояние элемента управления

Для информирования пользователя о недоступности того или иного элемента пользовательского интерфейса целесообразно воспользоваться услугами функции:

```
function DrawState(DC: hDC; Brush: HBRUSH; CBFunc: TFNDrawStateProc;
                   lData: LPARAM; wData: WPARAM;
                   X, Y, Width, Height: Integer; Flags: UINT): BOOL;
```

Здесь DC – дескриптор контекста графического устройства. При включенном флаге DSS_MONO необходимо в параметр Brush направить дескриптор используемой для вывода изображения кисти, во всех остальных случаях функция удовлетворится неопределенным указателем или нулевым значением. Третий по счету параметр CBFunc представляет собой указатель на необязательную функцию обратного вызова (при этом следует установить флаг DST_COMPLEX). В параметры lData и wData передаем дополнительные данные о рисунке или тексте. Параметры X, Y, Width и Height определяют местоположение левого верхнего угла и габариты результирующего изображения в единицах измерения устройства. Завершающий параметр содержит один обязательный (табл. 7.6) и дополнительный флаг (табл. 7.7).

Таблица 7.6. Обязательные флаги функции DrawState( )

Флаг	Описание
DST_BITMAP	В роли изображения выступает битовый образ, дескриптор которого следует передать через параметр lData. При этом битовый образ не должен быть выбран в контекст графического устройства.
DST_COMPLEX	Изображение формируется в рамках функции обратного вызова, определенной параметром CBFunc.
DST_ICON	В качестве изображения выступает иконка. Ее дескриптор передается в параметр lData.
DST_PREFIXTEXT	Функция обслуживает текстовую строку, содержащую символ амперсанда. Символ, следующий за символом «&», подчеркивается. Указатель на строку направляется в параметр lData, в параметр wData следует передать длину строки. Если строка заканчивается нулем, то в wData следует передавать нулевое значение.
DST_TEXT	Функция обслуживает текстовую строку. Параметры lData и wData используются, как при работе с флагом DST_PREFIXTEXT.

Таблица 7.7. Дополнительные флаги функции DrawState( )

Флаг	Описание
DSS_DISABLED	Соответствует пассивному режиму работы, рисуется рельефный контур изображения.
DSS_HIDEPREFIX	Флаг заставляет функцию игнорировать символ «&». Следующий за амперсандом символ не подчеркивается.
DSS_MONO	Выводит изображение, используя кисть с дескриптором Brush.
DSS_NORMAL	Обычный режим. Выводит не модифицированное изображение.
DSS_PREFIXONLY	Рисует только символ подчеркивания, обозначенный амперсандом &. Остальной текст игнорируется. Используется совместно с флагом DST_PREFIXTEXT.
DSS_RIGHT	Команда на выравнивание текста по правому краю.
DSS_UNION	Полутоновое размытое изображение.

При работе с флагом DST_COMPLEX в третий параметр функции передается ссылка на функцию обратного вызова, в рамках которой программист может внести свои собственные усовершенствования в процесс вывода изображения. Синтаксическая конструкция функции обратного вызова выглядит следующим образом:

```
Function DrawStateProc(DC : hDC; lData: LPARAM; wData : WPARAM;
                       X,Y : Integer):boolean; stdcall;
```

Назначения параметров DrawStateProc() аналогичны параметрам головной функции.

Ниже представлен листинг, демонстрирующий наиболее популярные режимы работы функции.

```
var
DIB:HBitmap;
//…
Procedure  GDI_PAINT(DC:hDC);
Var Brush:HBrush;
    s:string;
begin
//работа с битовым образом
  //1) нормальный режим
  DrawState(DC,0,nil,DIB,0,1,1,32,32, DST_BITMAP OR DSS_NORMAL);
  //2) эффект пассивного состояния
  DrawState(DC,0,nil,DIB,0,1,40,32,32, DST_BITMAP OR DSS_DISABLED);
  //3) наложение кисти
  Brush:=CreateHatchBrush(HS_DIAGCROSS,$000000);
  DrawState(DC,Brush,nil,DIB,0,1,80,32,32, DST_BITMAP OR DSS_MONO);
  DeleteObject(Brush);
  //4) полутоновое размытое изображение
  DrawState(DC,0,nil,DIB,0,1,120,32,32, DST_BITMAP OR DSS_UNION);
//работа с текстом
```

```
    //5) нормальный режим
    s:='File';
    DrawState(DC,0,nil,INTEGER(pChar(s)),4,1,170,0,0, DST_TEXT OR DSS_NORMAL);
    //6) эффект пассивного состояния
    s:='File'+#0;
    DrawState(DC,0,nil,INTEGER(pChar(s)),0,1,200,0,0, DST_TEXT OR DSS_DISABLED);
    //7) учет амперсанда и пассивное состояние
    s:='&File'+#0;
    DrawState(DC,0,nil,INTEGER(pChar(s)),0,1,230,0,0, DST_PREFIXTEXT OR
DSS_DISABLED);
END;
```

Результат работы представлен на рис. 7.15.

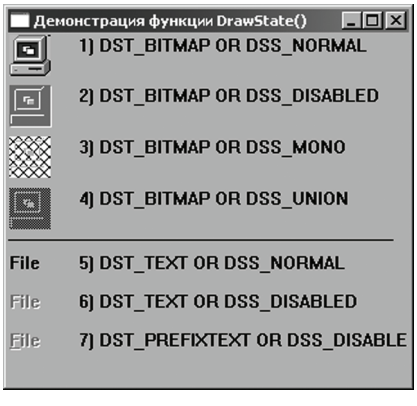


Рис. 7.15. Демонстрация функции DrawState( )

РЕЗЮМЕ

Любой, даже весьма сложный рисунок, можно построить за счет использования простейших функций, рисующих графические примитивы. В арсенале GDI таких предостаточно, начиная с функций черчения прямых линий и заканчивая кубическими сплайнами Безье. Базисом для построения более сложных геометрических фигур выступает обычный прямоугольник – в него вписываются эллипс и его фрагменты. При выводе на поверхность контекста простейших геометрических фигур задействуется не только перо, но и кисть. Перо обеспечивает прорисовку контура фигуры, а кисть – заливку ее внутренней области.

8

Траектории

Имеющиеся в распоряжении GDI функции научены рисовать только простейшие геометрические фигуры (линии, сплайны, прямоугольники и их производные), поэтому они неспособны удовлетворить запросы программ промышленного черчения, программ моделирования и, конечно же, самой «полезной» формы программного обеспечения – игрушек, для которых как воздух необходим механизм, позволяющий создавать фигуры нестандартной формы. Такой недостаток послужил толчком для разработки принципиально новых объектов GDI, умеющих описывать фигуры сложной формы. Одним из решений наболевшего вопроса стал объект *траектория* (*path*)[1]. Траектория не похожа ни на один из рассмотренных ранее объектов GDI, это математическое описание объединенных в единую структуру одной или несколько фигур, которые, в свою очередь, строятся за счет вызовов функций вывода графических примитивов и текста.

Хотя траектория представляет собой такой же равноценный объект GDI, как перо, регион, битовый образ или кисть, но, в отличие от всех перечисленных объектов, траектория явным образом не идентифицируется дескриптором. Траектория вообще неспособна существовать самостоятельно, она неразрывно связывается с контекстом устройства и хранится в нем в виде отдельного атрибута.

Описав траекторию, мы сможем обвести ее текущим пером и залить кистью. В GDI не существует документированной функции, позволяющей ухватить траекторию за «хвост» для того, чтобы, например, передать траекторию от одного контекста к другому или просто отредактировать. Этот весьма существенный недостаток ограничивает область применения траектории в наших программных продуктах, правда, он немного компенсируется возможностью конвертирования траектории в коллекцию линий и кривых Безье или в регион.

---

[1] В ряде изданий вместо термина «траектория» используют термин «путь» или «контур».

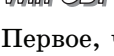
Уникальность траектории в том, что в одно и то же время контекст может содержать всего один объект траектории. Указанный объект описывается в координатах устройства, и, как следствие, вывод траектории на экран осуществляется практически молниеносно.

Траектория представляет собой совокупность линий, кривых и простейших геометрических фигур, которые рисуются за счет вызовов стандартных функций GDI. Для того чтобы построенная нами траектория предстала перед взором зачарованного пользователя, программисту необходимо пройти через два этапа:

1. Создание траектории.
   а) Инициализация траектории.
   б) Описание траектории.
   в) Завершение описания.
2. Вывод траектории.

Создание траектории

Дабы уведомить контекст о том, что мы приступили к сложному творческому процессу создания траектории, в самом начале работы вызываем функцию:



```
Function BeginPath(DC : HDC) : Boolean;
```

Первое, что сделает функция, – проверит, связана ли с контекстом другая траектория, и если это так, то устаревшая траектория будет сброшена. Затем контексту дается указание на то, что все следующие за `BeginPath()` обращения к функциям GDI не подлежат выводу на поверхности графического устройства, описываемого дескриптором контекста `DC`. У них более существенная задача – они формируют образ будущей траектории. В Windows предусмотрен строгий перечень функций, допустимых для описания траектории. Все входящие в VIP-список персоны можно разделить на три категории:

- **Функции, позволяющие строить линии, дуги и кривые:** `LineTo()`, `MoveToEx()`, `Arc()`, `ArcTo()`, `Polyline()`, `PolyPolyline()`, `PolylineTo()`, `PolyBezier()`, `PolyBezierTo()` и `PolyDraw()`.
- **Функции простейших геометрических фигур:** `Rectangle()`, `RoundRect()`, `Ellipse()`, `Chord()`, `Pie()`, `Polygon()` и `PolyPolygon()`.
- **Текстовые функции:** `ExtTextOut()` и `TextOut()`.

При выводе траектории, содержащей вызовы текстовых функций, убедитесь, что в контекст выбран контурный шрифт (TrueType или OpenType). На установленные по умолчанию растровые шрифты траектория не реагирует, поэтому текстовая информация не сможет быть отображена.

Завершив построение траектории, нам необходимо уведомить об этом контекст графического устройства. Сигналом того, что «конструктор» собран, станет обращение к функции:

Win GDI
```
Function EndPath(DC : HDC) : Boolean;
```

Функция EndPath() завершает формирование траектории и дает команду GDI выбрать траекторию в атрибут контекста устройства.

Методика хранения информации о траектории очень напоминает порядок описания кривых Безье средствами функции PolyDraw()*, рассмотренной в главе 6 этой книги. Для этого траектория также использует два массива: массив координат точек и вспомогательный массив описания типов точек (см. табл. 6.1).*

Вывод траектории

После того как атрибут контекста устройства, отвечающий за хранение траектории, будет нами заполнен «до краев», можно приступать ко второму этапу – выводу траектории. Для отображения траектории на поверхности графического устройства можно воспользоваться услугами функции:

Win GDI
```
Function StrokePath(DC : HDC) : Boolean;
```

Функция чертит все входящие в состав траектории линии и кривые, применяя текущее перо. Предлагаю рассмотреть небольшой пример построения траектории и вывода ее на поверхности окна. Допустим, что мы строим программный продукт, позволяющий моделировать принципиальные электрические схемы. Одна из частных задач подобного ПО заключается в создании некоего банка шаблонов радиодеталей. В демонстрационных целях мы ограничимся всего одним элементом – обычным биполярным транзистором (рис. 8.1).

```
Procedure DC_PAINT(DC:HDC);
var   Pen,OldPen:HPen; LogBrush:TLogBrush;
      P: array[0..12]of TPoint;
      Vert : array [0..1] of CARDINAL;
begin
  Vert[0]:=7;
  p[0].X:=0;   p[0].y:=125;  p[1].X:=90;  p[1].y:=125;
  p[2].X:=90;  p[2].y:=60;   p[3].X:=90;  p[3].y:=190;
  p[4].X:=90;  p[4].y:=90;   p[5].X:=190; p[5].y:=60;
  p[6].X:=190; p[6].y:=0;
```

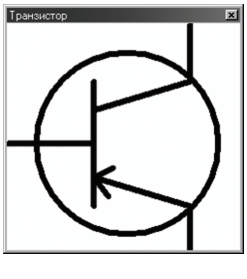


Рис. 8.1. Изображение p-n-p транзистора

```
  Vert[1]:=6;
  p[7].X:=190; p[7].y:=250;  p[8].X:=190;  p[8].y:=190;
  p[9].X:=90;  p[9].y:=160;  p[10].X:=110; p[10].y:=150;
  p[11].X:=90; p[11].y:=160; p[12].X:=105; p[12].y:=180;

BeginPath(DC);
  Ellipse(dc,30,30,220,220);
  PolyPolyLine(DC,p,Vert,2);
EndPath(DC);

LogBrush.lbColor:=$0; LogBrush.lbStyle:=BS_SOLID;
Pen:=ExtCreatePen(PS_GEOMETRIC OR PS_ENDCAP_ROUND OR PS_SOLID,7,LogBrush, 0, nil);
OldPen:=SelectObject(DC, Pen);
StrokePath(DC);

SelectObject(DC,OldPen);
DeleteObject(Pen);
end;
```

Изображение биполярного p-n-p транзистора (см. рис. 8.1) построено за счет сведения в единую траекторию эллипса и нескольких ломаных линий. С помощью эллипса чертится ограничивающий круг, а ломаные линии выводят на экран базу, коллектор и эмиттер транзистора. Обращения к функциям `Ellipse()` и `PolyPolyLine()` заключены внутрь конструкции `BeginPath()` … `EndPath()`. Вывод полученной траектории на поверхность контекста осуществляет функция `StrokePath()`.

Одно из существенных достоинств траектории заключается в том, что она никогда не прорисовывает один и тот же пиксел дважды.

При черчении траектории функция `StrokePath()` задействует только перо и не обращает никакого внимания на текущую кисть. Если такая дискриминация вам не по душе, то обязательно обратите внимание на функцию:

Win GDI

```
Function StrokeAndFillPath(DC:HDC) : Boolean;
```

Даже своим названием функция сигнализирует о том, что она предназначена не только для обводки контура, но и для заливки траектории. Возможности `StrokeAndFillPath()` наглядно продемонстрированы в следующем фрагменте кода:

```
Procedure DC_Paint(DC:HDC);
var  P:Array[0..4] of TPoint;
begin
BeginPath(DC);

 p[0].X:=50; p[0].y:=0;  p[1].X:=100; p[1].y:=80;
 p[2].X:=0;  p[2].y:=80; p[3].X:=50;  p[3].y:=0;
 Polyline(DC,p,4);
 Rectangle(DC,50,50,130,130);
 Ellipse(DC,90,90,190,190);

EndPath(DC);
```

```
SelectObject(DC,GetStockObject(NULL_PEN));
SelectObject(DC,GetStockObject(GRAY_BRUSH));
StrokeAndFillPath(DC);
End;
```

В единую траекторию объединены три геометрические фигуры: треугольник, квадрат и круг (рис. 8.2). Их внутренняя поверхность залита серой кистью. А перо мы принудительно отключили, передав контексту недееспособную пустышку NULL_PEN (это наша маленькая месть функции StrokePath(), игнорирующей кисть контекста). Впрочем, этого же результата можно было бы достичь посредством обращения к функции:

Win GDI
```
Function FillPath(DC:HDC) : Boolean;
```

Функция только закрашивает траекторию текущей кистью, не реагируя на выбранное в контекст перо.

После того как траектория будет нарисована на поверхности графического устройства, захваченные ею ресурсы освобождаются. Поэтому повторный вызов любого из методов черчения или заливки траектории не имеет смысла.

Траектория может содержать несколько фигур. Сигналом контексту, что построение фигуры завершено, станет вызов функции:

Win GDI
```
Function CloseFigure(DC : HDC) : Boolean;
```

В результате в последний элемент массива описания типов точек помещается флаг PT_CLOSEFIGURE (см. табл. 6.1). Он сигнализирует о том, что последняя точка фигуры должная быть соединена линией с начальной точкой – фигура замыкается.

Если по каким-либо причинам необходимость в траектории отпадает, то ее сброс обеспечит функция:

Win GDI
```
Function AbortPath(DC : HDC) : Boolean;
```

Говоря о «сбросе», я в первую очередь имею очистку отвечающего за хранение траектории атрибута контекста.

Рис. 8.2. Применение кисти при выводе траектории

Преобразование траектории в набор отрезков

В предыдущих главах уже упоминалось, что кубический сплайн имени господина Безье – великолепный инструмент, позволяющий нарисовать на поверхности графического устройства кривые самых замысловатых форм. Однако далеко не всегда можно положиться на подобную кривую. Причин тому много. Одна из них кроется в отсутствии простого математического аппарата, позволяющего вычислить длину сплайна, а такая задача часто встречается, например, при прокладке маршрута на топографических картах или при построении чертежей в программах автоматизированного проектирования. То же самое можно сказать о вычислении площади замкнутой кривой – задача значительно упрощается при трансформации такой фигуры в многоугольник.

За конвертирование криволинейных участков в прямые отвечает функция:

```
Function FlattenPath(DC : HDC) : Boolean;
```

Функция может применяться только после того, как наша программа сформирует траекторию и вызовет функцию `EndPath()`. В результате выполнения функции текущий объект траектории модифицируется (превращается в набор отрезков), но траектория по-прежнему остается траекторией – она хранится в атрибуте контекста и ее можно вывести на экран с помощью рассмотренных на предыдущих страницах методов `StrokePath()`, `FillPath()` и `StrokeAndFillPath()`. Если же нам требуется получить информацию о логических координатах, полученных в результате преобразования отрезков, то придется просить о помощи метод `GetPath()`.

Преобразование кривой линии в набор прямых осуществляется в соответствии со следующим рекурсивным алгоритмом: кривая делится пополам до тех пор, пока она не превратится в набор обычных прямых отрезков, с достаточной точностью аппроксимирующих изгибы кривой.

В качестве ближайшего соратника `FlattenPath()` рекомендую функцию:

```
Function WidenPath(DC:HDC) : Boolean;
```

Основная цель функции аналогична – конвертирование траектории в набор отрезков. Функция должна работать с замкнутой траекторией и в результате генерирует также замкнутую фигуру. Еще одна особенность `WidenPath()` в том, что при формировании замкнутой фигуры учитывается текущее перо, и например, если оно не единичной толщины, то размер результирующей области станет больше, чем размер траектории. Единственное условие, предъявляемое функцией к выбранному в контекст перу, – оно должно быть геометрическим.

Так же как и `FlattenPath()`*, функция* `WidenPath()` *преобразует кривые линии траектории в набор прямых отрезков. Но самая существенная особенность нашей новой знакомой в другом – функция трансформирует траекторию в замкнутую фигуру.*

Анализ траектории

Контекст устроен таким образом, что не способен хранить данные более чем об одной траектории. На момент разработки GDI этот факт ни в коем случае не считали недостатком. При конструировании контекста устройства программисты Microsoft во главу угла ставили не глубину его карманов, которые можно наполнить всевозможными объектами, а скорость, с которой он станет выводить картинку на поверхность. Поэтому (видимо, исходя из практического опыта, – всякий знает, что набитые карманы несколько мешают при беге) создатели Windows GDI ограничили контекст возможностью обслуживания единственной траектории. Облегчив работу своему любимому GDI, разработчики Windows несколько усложнили процесс борьбы за хлеб насущный нашему брату – ленивому программисту. Траектория – действительно великолепная штука, но при программировании даже не очень заумного графического модуля одного экземпляра траектории зачастую не хватает. Встает вопрос о сохранении одной траектории в ОЗУ или в физическом файле в тот момент, когда контексту передаются данные об ее коллеге. Мы и рады что-нибудь сохранить, но что? Неужели надо перекидывать в файл все массивы точек и вызовы функций Windows GDI, заключенные в рамки `BeginPath()`...`EndPath()`? Полученный результат скорее напомнит метафайл, что само по себе неплохо, но нам нужна лишь траектория. Для нас это не единственный повод затаить обиду на способ описания траектории в контексте. Второй случай, показывающий острую необходимость в информации о контрольных точках траектории, связан с осуществлением преобразований на плоскости, например преобразований перспективы. Полагаю, вы согласитесь, что тяжело удвоить неизвестный размер или повернуть на несколько градусов линию, хранящую за семью замками информацию о своем местоположении. Все это несколько напоминает классический царский указ из сказки: «Иди туда, не знаю куда, принеси то, не знаю что...», но в этом случае родство с народным фольклором мало утешает.

Как всегда, официальная документация Microsoft скромно умалчивает, каким именно образом осуществляется описание точек траектории. Можно лишь догадываться, что это какая-то сложная структура данных, прячущаяся от нас в адресном пространстве процесса. Единственное, что известно точно, так это то, что контекст хранит траекторию не в логических координатах, а в координатах устройства. Однако, осознавая, что такое положение вещей вряд ли устроит творцов программного обеспечения, корпорация самоотверженно документировала функцию `GetPath()`, способную дать некоторую подсказку о внутреннем мире траектории.

Win GDI

```
Function GetPath(DC:HDC; var Points, Types; Size : Integer) : Integer;
```

Функция позволяет провести подробное исследование траектории. Результаты анализа направляются в параметры `Points` и `Types`. Массив `Points` получит список всех описывающих траекторию точек, представленных в виде логических координат. В свою очередь, массив `Types` будет содержать дополнительную характеристику каждой из точек. Это не что иное, как константы типов

точек, применяемых в функции PolyDraw(). Они представлены в табл. 6.1 главы 6 «Линии и кривые», поэтому сейчас их повторно описывать не станем. В последнем параметре функции программист определяет, сколько именно точек он рассчитывает проанализировать. Таким образом, выпытав у траектории ее характеристики, мы сможем обрабатывать их так, как нам заблагорассудится, и затем выводить на поверхности контекста, но теперь за счет функции PolyDraw() (при этом помня, что в Win95/98/Me она не поддерживается).

При анализе траектории мы столкнемся с определенной трудностью: она заключается в том, что нам заранее неизвестно количество точек в траектории. Соответственно, мы не знаем, какие именно потребуются размеры массивов для сбора информации. Предусмотрено два варианта выхода из положения. Наиболее прост первый вариант, он заключается в предварительном создании двух массивов с заведомо большими размерами. Но более рациональным решением станет второй вариант действий, он заключается в двукратном обращении к GetPath(). За счет первого вызова функции мы должны уточнить число точек в траектории, второй вызов решит основную задачу – возвратит описание траектории. Для уточнения количества точек в траектории в последний параметр функции GetPath() следует передать ноль.

Ниже представлен фрагмент кода, позволяющий исследовать состав траектории, представляющей собой круг, вписанный в квадрат. В данном случае мы задействуем динамические массивы: P – для сбора координат точек и T – для сбора данных о типах точек. Результаты анализа передаются в список (рис. 8.3).

```
Procedure GetPathInfo;
var A:Integer;
    s:string;
    P:Array of TPoint;
    T:Array of Byte;
```

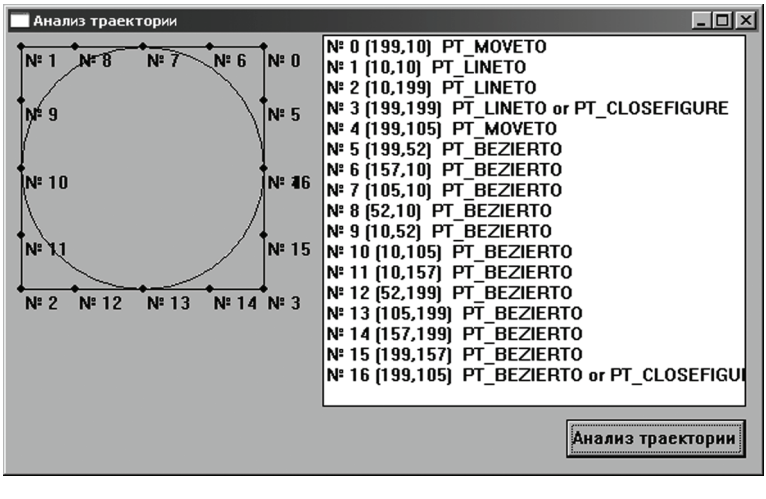


Рис. 8.3. Исследование траектории

```
    Size:Integer;
begin
DC:=GetDC(Wnd);
  BeginPath(DC);
  Rectangle(DC,10,10,200,200);
  Ellipse(DC, 10,10,200,200);
  EndPath(DC);

size:=GetPath(DC,P,T,0); //узнаем число точек

SetLength(P,Size); {распределяем память под динамические массивы}
SetLength(T,Size);
GetPath(DC,P[0],T[0],Size); //собираем данные

  for a:=0 to Size-1 do
     begin
      s:=Format('№ %d (%d,%d)',[a,P[a].X,P[a].y]);
      CASE  T[a] OF
        PT_MOVETO                  : s:=s+' '+'PT_MOVETO';
        PT_LINETO                  : s:=s+' '+'PT_LINETO';
        PT_BEZIERTO                : s:=s+' '+'PT_BEZIERTO';
        PT_LINETO+PT_CLOSEFIGURE   : s:=s+' '+'PT_LINETO OR PT_CLOSEFIGURE';
        PT_BEZIERTO+PT_CLOSEFIGURE : s:=s+' '+'PT_BEZIERTO OR PT_CLOSEFIGURE';
       END;
       SendMessage(ListBox1,LB_ADDSTRING,0,INTEGER(pChar(s)));
     end;

PolyDraw(DC,P[0],T[0],Size);
DeleteDC(DC);
end;
```

В отличие от GDI, в современной версии графической библиотеки GDI+ реализована возможность одновременного создания нескольких траекторий. Изменения ни в коем случае не затронули контекст графического устройства (для хранения траектории у него по-прежнему имеется один-единственный атрибут). Но в GDI+ предусмотрена альтернативная возможность описания траектории, для этого предназначен класс `TGPGraphicsPath`*. Для каждой новой траектории необходимо всего лишь создавать очередной экземпляр класса и передавать в него характеристики траектории. Когда понадобится нарисовать траекторию на графическом устройстве, достаточно выбрать требуемый объект* `TGPGraphicsPath` *и направить данные из него в холст GDI+. Весьма простое решение, но к нему разработчики графической библиотеки Windows шли более десяти лет…*

Преобразование траектории в регион

Для преобразования траектории в регион предназначена функция Windows GDI:

```
Function PathToRegion(DC : HDC) : HRGN;
```

Перед обращением к функции надо убедиться, что контекст содержит траекторию и она закрыта с помощью `EndPath()`. Функция замкнет все незамкнутые фигуры траектории и конвертирует их в регион. Подчеркиваю, что в отличие от траектории регион обладает дескриптором, это означает, что указанный объект разрешено хранить в памяти и обращаться к нему по мере необходимости. С траекторией такой фокус не получится, так как она одноразовый объект.

В полученный в результате преобразования регион не войдут правая и нижняя стороны исходной траектории.

Вопросом создания из траектории региона отсечения ведает функция `SelectClipPath()`*, с ней мы познакомимся немного позднее, в главе 10 «Отсечение и регионы контекста устройства».*

РЕЗЮМЕ

Траектория представляет собой мощный инструмент, предназначенный для создания фигур нестандартной формы. Практическим примером использования траекторий может стать построение массива шаблонов графических объектов, которые позднее смогут применяться в различного рода программах автоматизированного проектирования. Единственный недостаток траектории GDI в том, что в одно и то же время может существовать всего один экземпляр этого объекта. Это объясняется тем, что траектория хранится в отдельном специальном атрибуте контекста и если логика программы предполагает создание новой траектории, то, увы, приходится уничтожать старую...

9

Регионы

Наряду с кистями и перьями регион является старожилом графической составляющей современной операционной системы – он появился на свет вместе с первой версией Windows в 1985 году. Регион – это объект GDI, описывающий произвольное множество точек двумерного пространства. В состав региона может входить весь перечень точек множества либо ни одной из них. В зависимости от способа определения региона он в состоянии приобрести прямоугольную, эллиптическую или любую другую форму, ограничений всего два – множество допустимых точек пространства и уровень вашего воображения.

Спектр задач, при решении которых нам не обойтись без содействия региона, весьма обширен. В простейшем случае это обычный вывод на графическом устройстве представленных в формате региона фигур сложных форм. Кроме того, с помощью региона можно задать нестандартную форму для окна, определить нуждающуюся в перерисовке область, проверить факт вхождения точки или прямоугольной области в состав региона и обеспечить работу одного из наиболее важных направлений во всей деятельности пользовательского интерфейса многооконной операционной системы Windows – технологии отсечения, которой полностью посвящена следующая глава этой книги.

У региона, как и большинства объектов GDI, имеется уникальный идентификатор. В случае региона это дескриптор `HRGN`, указанный дескриптор возвращается всеми функциями, создающими и комбинирующими регионы.

Создание региона

Для создания региона нам не требуется помощь контекста устройства, программисту достаточно только определить, какие точки координатного пространства должны ему принадлежать. Поэтому один и тот же объект региона допускается использовать одновременно при работе с различными контекстами.

Простейший из регионов имеет прямоугольную форму. Для его описания стоит подружиться с функцией:

```
Function CreateRectRgn(Left, Top, Right, Bottom : Integer) : HRGN;
```

Функция потребует от разработчика определиться с логическими координатами левого верхнего и нижнего правого углов региона и при удачном стечении обстоятельств (если вы не перемудрите с координатами) возвратит дескриптор региона. Если же попытка создать регион окажется безрезультатной, то в ответ на все наши старания мы получим неопределенный указатель `nil`. Это замечание в полной мере относится ко всем функциям, предназначенным для создания экземпляра региона.

Согласно документации SDK координаты региона должны находиться в пределах диапазона значений $0..2^{27}$ по осям X и Y. Другими словами, несмотря на то, что при назначении границ региона используются логические единицы измерения, его максимальные пределы ограничены координатами устройства.

Лучшая подружка функции `CreateRectRgn()` — функция:

```
Function CreateRectRgnIndirect(const Rct : TRect) : HRGN;
```

К ней обращаются в ситуации, когда координаты прямоугольной области будущего региона представлены в виде структуры `TRect`.

Прямоугольный регион со скругленными углами формируется при посредничестве функции:

```
Function CreateRoundRectRgn(Left,Top,Right,Bottom : Integer;
                            Width,Height : Integer):HRGN;
```

Назначение первых четырех аргументов такое же, как и у квартета параметров `CreateRectRgn()`. Аргументы `Width` и `Height` определяют ширину и высоту скругляющего эллипса, все точно так, как при работе с функцией `RoundRect()`. После всего сказанного функцию, создающую регион в форме эллипса, даже не стоит отдельно комментировать.

```
Function CreateEllipticRgn(Left,Top,Right,Bottom : Integer) : HRGN;
```

Программисту достаточно определиться с координатами органичивающего прямоугольника, а GDI позаботится, чтобы в него уместился эллептический регион. У этой функции также есть напарница, или конкурент, называйте как хотите.

```
Function CreateEllipticRgnIndirect(const Rct : TRect) : HRGN;
```

Функция создает точно такой же эллиптический регион, как и `CreateEllipticRgn()`, с той лишь разницей, что вместо четырех параметров она удовлетворится единственным аргументом `TRect`.

При определении координат региона с помощью функций CreateRoundRectRgn(), CreateEllipticRgn() *или* CreateEllipticRgnIndirect() *не забывайте, что во множество точек региона не войдут его нижняя и правая границы.*

Если вам наскучили прямоугольные регионы, то пора переходить к изучению более сложных функций. Многоугольный регион произвольной формы позволит описать функция:

```
Function CreatePolygonRgn(const Points; Count : Integer;
                          FillMode : Integer) : HRGN;
```

Координаты вершин многоугольника направляются в массив точек Points, количество вершин региона передается в параметр Count. Последний аргумент функции определяет особенности заливки фигуры, он может принимать одно из двух значений: ALTERNATE или WINDING. Режимы оказывают влияние на методы определения принадлежности точки к множеству точек региона, о них мы говорили в седьмой главе этой книги (см. табл. 7.1).

В наше время регионом в виде банального многоугольника, пусть даже произвольной формы, никого не удивишь. То ли дело регион, описанный несколькими сложными фигурами. Подобный «салат» в два счета приготовит функция:

```
Function CreatePolyPolygonRgn(const Points; const Vert;
                              count : Integer; FillMode : Integer) : HRGN;
```

Порядок определения параметров функции схож с идеей работы функции PolyPolyline(), изученной нами в главе 6 «Линии и кривые». Координаты вершин всех многоугольников передаются в массив Points. Сколько именно точек отведено для описания вершин той или иной фигуры, определяется в массиве Vert. Параметр Count содержит общее число многоугольников. Завершающий параметр определяет режим заполнения региона (см. табл. 7.1).

Ниже предложен фрагмент листинга, демонстрирующий влияние режима определения принадлежности точки на заполнение региона.

```
Procedure GDI_PAINT(DC:HDC);
var RGN:HRGN;
    P:array[0..9] of TPoint;  V:array[0..1] of integer;
    i:integer;
begin
V[0]:=5;
 p[0]:=Point(20,20);  p[1]:=Point(120,20);  p[2]:=Point(120,120);
 p[3]:=Point(20,120); p[4]:=Point(20,20);

V[1]:=5;
 p[5]:=Point(70,10);  p[6]:=Point(130,70);  p[7]:=Point(70,130);
 p[8]:=Point(10,70);  p[9]:=Point(70,10);

RGN:=CreatePolyPolygonRgn(p,v,2,ALTERNATE );
 FillRgn(DC,RGN,GetStockObject(GRAY_BRUSH));
 TextOut(DC,p[6].X,p[6].Y,PChar('ALTERNATE'),9);
 DeleteObject(RGN);

for i:=0 to 9 do p[i].X:=p[i].X+200; //смещаем регион по оси X
```

```
    RGN:=CreatePolyPolygonRgn(p,v,2,WINDING);
    FillRgn(DC,RGN,GetStockObject(GRAY_BRUSH));
    TextOut(DC,p[6].X,p[6].Y,PChar('WINDING'),7);
    DeleteObject(RGN);
end;
```

Результат выполнения кода представлен на рис. 9.1. Режим ALTERNATE позволил оставить нетронутой внутреннюю часть восьмиугольного региона. В свою очередь многоугольник, нарисованный в режиме WINDING, на все 100% заполнен серой кистью.

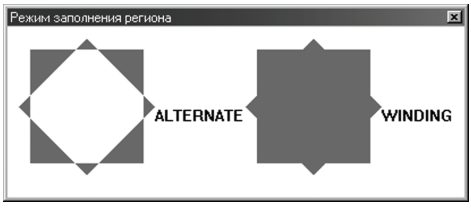


Рис. 9.1. Зависимость порядка заполнения региона от состояния параметра FillMode

Вдоволь поиграв с регионом, не забудьте отпустить на волю захваченные им ресурсы. Как и большинство объектов GDI, регион уничтожается посредством вызова функции DeleteObject().

Мы рассмотрели практически все способы создания региона, за исключением одного – формирования объекта региона на базе структуры TRgnData. Для этого предназначена функция ExtCreateRegion(). Ее мы изучим несколько позднее, в подразделе, посвященном получению информации о регионе.

В главе 8 этой книги упоминается еще один способ создания региона. Это преобразование траектории в регион с помощью функции PathToRegion(). *Кстати, это единственная работающая с регионами функция, которой необходим дескриптор контекста графического устройства.*

Вывод региона

При выводе региона на поверхности графического устройства Windows использует только кисть и никоим образом не задействует перо. Вследствие такого подхода GDI к изобразительному искусству корректнее говорить не рисование, а заливка региона. Для этой цели предусмотрено две функции. Первая из них:

Win GDI

```
Function PaintRgn(DC : HDC; Rgn : HRGN) : Boolean;
```

Она закрасит регион текущей кистью контекста графического устройства. Вторая функция для заливки региона готова воспользоваться любым экземпляром кисти:

Win GDI

```
Function FillRgn(DC : HDC; Rgn : HRGN; Brush : HBRUSH) : Boolean;
```

Для этого в перечень ее параметров добавлен аргумент Brush.

Еще одна функция, связанная с процессом отображения региона, позволяет инвертировать цвета на поверхности устройства, описываемого дескриптором DC.

Win GDI

```
Function InvertRgn(DC : HDC; Rgn : HRGN) : Boolean;
```

Если пределом наших мечтаний станет не заливка, а черчение контура региона, то нет повода для отчаяния – задуманное вполне реально, но и в этом случае нам не обойтись без посредничества кисти.

Win GDI

```
Function FrameRgn(DC : HDC; Rgn : HRGN; Brush : HBRUSH;
                  Width, Height : Integer) : Boolean;
```

Как отмечалось ранее, регион недолюбливает перо. Поэтому функция примечательна тем, что вместо пера для вывода контура региона использует кисть. Решение не лишено оригинальности, создатели функции разрешили нам проводить индивидуальную настройку ширины (width) и высоты (height) шаблона кисти. Благодаря такому высокому доверию к нашему брату со стороны разработчиков функции вертикальные и горизонтальные линии контура региона допустимо сделать различной толщины, например, так, как показано в следующем листинге:

```
Procedure GDI_PAINT(DC:HDC);
var  RGN:HRGN;
begin
RGN:=CreateRectRgn(10,10,200,100);
FrameRgn(DC, RGN, GetStockObject(BLACK_BRUSH),2,8);
DeleteObject(RGN);
RGN:=CreateEllipticRgn(10,110,200,250);
FrameRgn(DC, RGN, GetStockObject(BLACK_BRUSH),2,8);
DeleteObject(RGN);
end;
```

Результат работы процедуры представлен на рис. 9.2. Горизонтальные контуры границ регионов нарисованы кистью толщиной 8 логических единиц, вертикальные – 2.

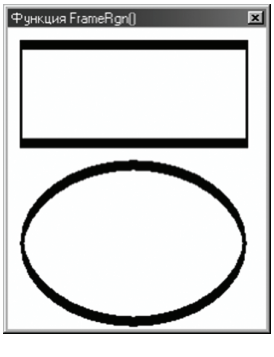


Рис. 9.2. Демонстрация функции FrameRgn( )

По сравнению с большинством других операций рисования создание регионов и вывод их на поверхности устройства сопряжены со значительными затратами как памяти, так и процессорного времени. В особой мере это замечание относится к функции `FrameRgn()`. *Поэтому в меру возможности старайтесь сократить в своей программе обращение к функциям такого рода и заменять их функциями рисования прямоугольников, эллипсов и траекторий.*

Операции с регионами

Весьма часто перед разработчиком программного обеспечения встает задача по проверке принадлежности точки определенной области, например, элементу управления или его части. Еще больших проблем может добавить ситуация, когда форма области отличается от квадратной; такая ситуация возникает практически во всех игровых приложениях. Данную задачу легко решить, если исследуемую область описывает регион, тогда проверку вхождения точки в регион `Rgn` проведет функция:

```
Function PtInRegion(Rgn : HRGN; X,Y : Integer) : Boolean;
```

Если регион является счастливым обладателем точки с координатами (`X`,`Y`), то функция возвратит `true`. Еще раз напомню, что точка, находящаяся на нижней или правой границе области, не входит в регион. Анонсированная функция часто находит применение при анализе местоположения указателя мыши. Для этого оконную процедуру необходимо дополнить секцией обработки сообщения `WM_MOUSEMOVE`, приходящего в адрес окна в момент движения указателя.

```
Var //…
  xPos,yPos:integer;
  Rgn:HRGN;

function WindowProc(Window: HWND; Msg: UINT;
                    wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
  case Msg of
       WM_CREATE:   Rgn:=CreateEllipticRgn(40,40,200,200);
       WM_MOUSEMOVE: begin
                       xPos := LOWORD(lParam); //координата X
                       yPos := HIWORD(lParam); //координата Y

                       DC:=GetDC(Window);
                        IF PtInRegion(Rgn,xPos,yPos)=true then
                           TextOut(DC,0,0,pChar('TRUE '),5) else
                           TextOut(DC,0,0,pChar('FALSE'),5);
                      ReleaseDC(DC,Window);
                     end;
//…
end;
```

Схожая задача, связанная с проверкой вхождения в состав региона не просто точки, а произвольной прямоугольной области, элементарно решается с помощью функции:

```
Function RectInRegion(Rgn : HRGN; const Rect : TRect) : Boolean;
```

Функция возвратит `true` в случае, если хотя бы частичка прямоугольника `Rect` окажется в «территориальных водах» региона.

Идентичность двух регионов проверяется посредством вызова функции:

```
Function EqualRgn(Rgn1, Rgn2 : HRGN) : Boolean;
```

Если множества точек регионов полностью совпадают, то функция возвратит значение `true`. Отличие регионов хотя бы по одной точке послужит причиной возврата значения `false`.

Регион может быть смещен по осям абсцисс и ординат. Процессом смещения заведует функция:

```
Function OffsetRgn(Rgn : HRGN; XOffset, YOffset : Integer) : Integer;
```

Направление и величина смещения региона `Rgn` задаются с помощью параметров `XOffset` и `YOffset`.

Объединение регионов

Обладая дескрипторами двух регионов, можно получить их объединение. Результат объединения также станет регионом. По сути, это операции с двумя множествами точек. В роли специалиста по теории множеств выступает функция:

```
Function CombineRgn(RgnDest, RgnSource1, RgnSource2 : HRGN;
                    CombineMode: Integer) : Integer;
```

Исходные регионы – `RgnSource1` и `RgnSource2`. Результат объединения направляется в параметр `RgnDest`. Какую именно из операций следует проводить, определяет параметр `CombineMode`. Он представлен в табл. 9.1.

Таблица 9.1. Варианты значений параметра CombineMode

Константа	Значение	Результат операции
`RGN_AND`	1	Пересечение двух регионов.
`RGN_OR`	2	В регион войдут точки, принадлежащие хотя бы одному из регионов.
`RGN_XOR`	3	Множество точек региона формируется из точек, принадлежащих обоим регионам, за исключением их общих точек. Такая операция называется симметричной разностью.
`RGN_DIFF`	4	Из региона `RgnSource1` вычитается `RgnSource2`.
`RGN_COPY`	5	В результате мы получим копию региона `RgnSource1`.

О результате операции объединения двух регионов можно судить по возвращаемому функцией значению. Оно описано в табл. 9.2.

Таблица 9.2. Результат выполнения функций CombineRgn( ) и GetRgnBox( )

Константа	Описание
NULLREGION	Регион пуст.
SIMPLEREGION	Регион представляет собой обычный прямоугольник.
COMPLEXREGION	В результате получено несколько прямоугольников.
ERROR	Ошибка выполнения.

Перед обращением к `CombineRgn()` *необходимо создать регион-получатель. Для этого достаточно сформировать пустой регион, например, вызвав функцию* `CreateRectRgn()` *и передав во все ее параметры нулевые значения.*

Преобразование региона в прямоугольный регион

Вне зависимости от причудливости формы региона мы обладаем возможностью выяснить координаты ограничивающего его прямоугольника.

```
Function GetRgnBox(Rgn : HRGN; var Rect : TRect) : Integer;
```

Значения координат возвращаются при посредничестве последнего параметра `Rect`.

Обладая дескриптором региона, мы сможем преобразовать его в прямоугольник, в который этот регион вписан. Для этого надо воспользоваться функцией:

```
Function SetRectRgn(Rgn : HRGN; Left, Top, Right, Bottom : Integer) :
HRGN;
```

Форма и геометрические размеры исходного региона значения не имеют.

Получение информации о регионе

Заранее предвижу удивление читателя: «Если я создал регион, то зачем мне какие-то исследовательские инструменты для выведывания его характеристик? Я ведь их и так знаю!» И да, и нет... Действительно, нет никакого смысла заниматься исследованием статичного региона, который был создан единожды, местоположение и границы которого тверже гранита. Но ведь это всего лишь частный случай из бурной жизни регионов. Вполне вероятно и кардинально противоположное поведение региона. Он может скакать по экрану быстрее зайца, ведь все определяется стоящими перед ним задачами. За термином «регион» способен скрываться автомобиль, несущийся по извилистым дорогам игры, или флегматичная медуза из видеоролика. Все зависит от воображения...

Для организации хранения региона в памяти специалисты Microsoft изрядно потрудились. В первую очередь, инженеры корпорации столкнулись с проблемами экономии памяти (попробуйте рационально описать множество точек, входящих в регион нетривиальной формы, скажем облака в штанах...)

и обеспечения приемлемой скорости работы. После того как программисты свою задачу выполнили, разработанную ими структуру корпорация сразу «засекретила» (по сложившейся в этой компании славной традиции отказалась ее документировать), а в наше распоряжение (на страницах SDK) выдала некий эрзац – симбиоз записей _RGNDATA и _RGNDATAHEADER.

```
type
PRgnData = ^TRgnData;
  _ TRgnData = record
    rdh: TRgnDataHeader; //заголовок региона
    Buffer: array[0..0] of CHAR;
    Reserved: array[0..2] of CHAR;
  end;
```

В свою очередь, заголовок региона TRgnDataHeader представляет собой запись следующего вида:

```
type
  PRgnDataHeader = ^TRgnDataHeader;
  _ TRgnDataHeader = packed record
    dwSize: DWORD;    //размер заголовка
    iType: DWORD;     //всегда RDH_RECTANGLES, трапеции не используются
    nCount: DWORD;    //количество прямоугольных областей в регионе
    nRgnSize: DWORD;  //размер буфера с информацией о регионе
    rcBound: TRect;   //ограничивающий регион прямоугольник
  end;
```

Перечисленные записи вряд ли отражают реалии внутреннего представления региона в графическом механизме Windows. Это всего лишь описание региона для нужд программистов, но что поделаешь, и на том спасибо.

Для заполнения указанных структур данными необходимо завести дружбу с функцией:

```
Function GetRegionData(Rgn : HRGN; BufSize : Cardinal;
                       RgnData : pRgnData) : Cardinal;
```

Она преобразует тайную внутреннюю структуру региона в формат представленных записей. Здесь Rgn – дескриптор исследуемого региона; BufSize – размер буфера, требуемый для размещения данных; RgnData – указатель на запись TRgnData, в которую помещается с таким трудом добытая информация.

Для получения информации о регионе требуется сделать двукратный поклон функции-зазнайке GetRegionData(). Первый вызов позволяет выяснить размер буфера, который потребуется для сбора данных. Собственно данные возвратит второй вызов функции.

```
var RgnData : TRgnData;

Procedure GDI_PAINT(DC:HDC);
var RGN,RGN_Ext:HRGN;
    s:string;
    BuffSize:Cardinal;
begin
```

```
RGN:=CreateEllipticRgn(5,5,105,55);
 BuffSize:=GetRegionData(RGN,0,nil);
   if BuffSize>0 then
      begin
        GetRegionData(RGN,BuffSize,@RgnData);
        s:=Format('nCount =%d',[RgnData.rdh.nCount]);
        TextOut(DC,RgnData.rdh.rcBound.Right+5,1,PChar(s),Length(s));
        s:=Format('nRgnSize =%d',[RgnData.rdh.nRgnSize]);
        TextOut(DC,RgnData.rdh.rcBound.Right+5,20,PChar(s),Length(s));
        s:=Format('dwSize =%d',[RgnData.rdh.dwSize]);
        TextOut(DC,RgnData.rdh.rcBound.Right+5,40,PChar(s),Length(s));
        s:=Format('rcBound (%d,%d,%d,%d)',
                  [RgnData.rdh.rcBound.Left,RgnData.rdh.rcBound.Top,
                   RgnData.rdh.rcBound.Right,RgnData.rdh.rcBound.Bottom]);
        TextOut(DC,RgnData.rdh.rcBound.Right+5,60,PChar(s),Length(s));
      end;

      RGN_Ext:=ExtCreateRegion(nil,GetRegionData(RGN,0,nil),RgnData);
      FillRgn(DC,RGN_Ext,GetStockObject(GRAY_BRUSH));
      DeleteObject(RGN_Ext);
      DeleteObject(RGN);
end;
```

Разочарованы? И это все, на что способна графическая подсистема Windows при информировании нас об объекте региона? И действительно, разве велика ценность информации о том (рис. 9.3), что рассматриваемый в примере эллиптический регион описывается 39 прямоугольниками и ограничен областью (5,5,104,54)? Пилюлю немного подсластит тот факт, что на основе полученных данных достаточно просто создать новый экземпляр региона, для этого предназначена функция `ExtCreateRegion()`.

Win GDI

```
Function ExtCreateRegion(XForm : PXForm; BufSize : Cardinal;
                         const RgnData : _RgnData) : HRGN;
```

Эта функция нам уже встречалась в предыдущем листинге. Основное достоинство `ExtCreateRegion()` в том, что она способна проводить с регионом двумерное аффинное преобразование. Для этого она обладает параметром `XForm` – указателем на матрицу `TXForm`, задающую правила мировых преобразований. Более подробно матрица представлена в главе 12 «Мировые координаты», а пока отмечу лишь то, что в простейшем случае задействуются лишь поля `eM11` и `eM22`, – в них передаются единицы. Такие значения задают тождественное (без каких либо изменений) отображение региона в регион и ничего более.

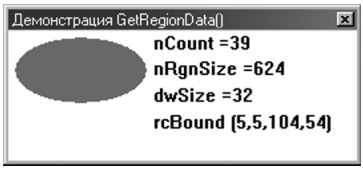


Рис. 9.3. Анализ эллиптического региона

Впрочем, этого же эффекта легко добиться, передав в первый параметр неопределенный указатель `nil`.

РЕЗЮМЕ

Регион представляет собой множество точек в двумерном координатном пространстве. Данное множество может быть пустым или представлять собой сложную фигуру любой формы. Большинство функций создания экземпляра региона не зависят от контекста устройства, поэтому один и тот же объект региона может использоваться в разных контекстах. Обычно регионы предназначены для определения формы окон, осуществления простейших операций с множествами. Одну из ключевых возможностей региона – отсечение, мы рассмотрим в очередной главе.

10

Отсечение и регионы контекста устройства

Отсечение (*clipping*) – один из самых важных механизмов GDI, и, как бы это ни показалось странным, он предназначен не для вывода графики на поверхность контекста, а напротив – для исключения части изображения из процесса прорисовки. Подтверждений существенной роли отсечения в жизнедеятельности графической составляющей операционной системы бесчисленное множество. Например, вследствие того, что система Windows является многооконной средой, на переднем плане зачастую находится лишь одно окно, остальные окна перекрываются друг другом. При перерисовке частично закрытых окон не самым разумным решением станет использование ресурсов компьютера для обслуживания спрятанных от взора пользователя участков. Это все равно, что давать спектакль за опущенным занавесом. Поэтому скрытая часть окна отсекается (исключается из процесса перерисовки), а графический вывод продолжается только в видимой области окна. Приведу другой пример. Не станет серьезной ошибкой, если в коде программы вы попытаетесь заставить приложение провести линию из точки (0,0) в точку, расположенную за пределами окна. Но как только линия покинет границы клиентской области, графический механизм Windows прекратит обращать на нее внимание. Линия вышла за системный регион и рисовать больше негде, поэтому GDI перестает ломать копья в отсутствии публики.

С примерами пока достаточно. Теперь поговорим о том, каким образом операционная система принимает решение на использование своего графического инструментария при выводе пикселов, линий, кривых, текста и т. п. Для этого в недрах Windows, не покладая рук, трудится *конвейер отсечения* (*clipping pipeline*). Он занят тем, что перехватывает все попытки процессов что-нибудь где-нибудь нарисовать. Как и любой другой бюрократ, наш новый знакомый в некоторой степени притормаживает процесс графического вывода, ведь конвейер должен оценить место предстоящего вывода и дать на это добро. Несмотря на столь высокую должность в «министерстве» GDI, ему особо не позавидуешь. Работа контейнера – не сахар: каждый графический

вызов преодолевает несколько уровней отсечения и на каждом из них требует пристального внимания.

На рис. 10.1 представлены области окна, непосредственно влияющие на формирование первоначального уровня отсечения.

Во-первых, это образуемая в момент создания окна (вспоминайте функцию `CreateWindow` или ее «продвинутую» коллегу `CreateWindowEx`) *прямоугольная область окна*. Во-вторых, это *регион окна*. Совсем не обязательно, чтобы регион окна обладал прямоугольной формой. За счет обращения к функции `SetWindowRgn()` мы сможем придать окну форму чего угодно, хоть эскимо на палочке (это я вам докажу чуть позднее), правда, с единственным ограничением – размеры региона окна не должны превышать прямоугольную область, отводимую под это окно. В-третьих, при работе с контекстом графического устройства клиентской области окна на отсечение повлияют границы этой *клиентской области*. В-четвертых, существует понятие *обновляемого региона*. Эта область, нуждающаяся в немедленной перерисовке, например, после закрытия выведенного в модальном режиме (поверх других окон) диалогового окна. В этом случае система должна перерисовать не всю поверхность окна, а только ту часть, которая пряталась от нас под диалогом. В-пятых, окно может быть перекрыто другим окном, в этом случае прорисовка осуществляется только в пределах *видимой области*.

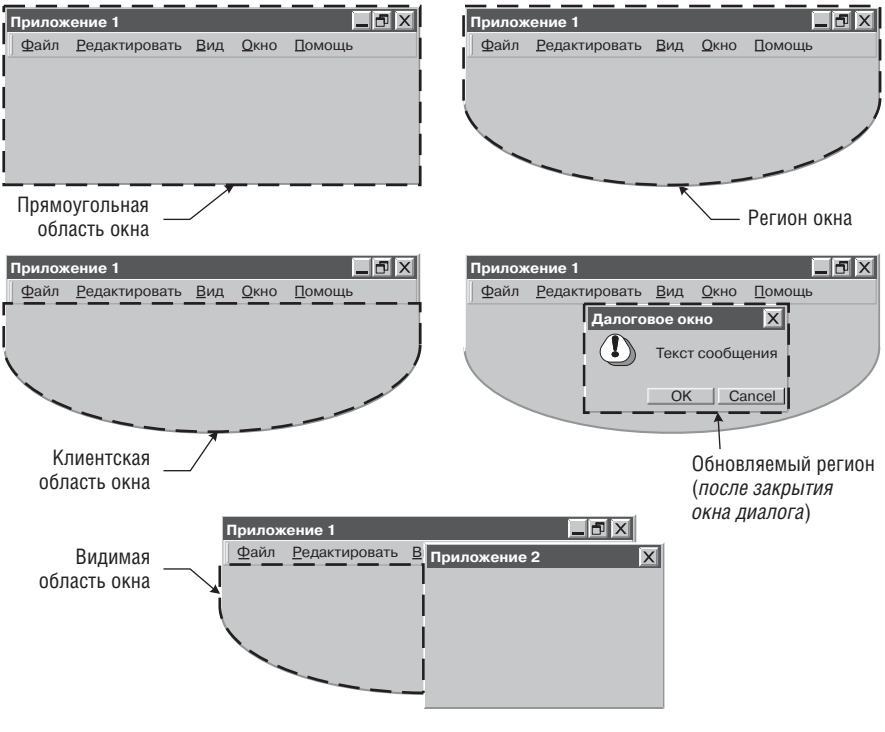


Рис. 10.1. Области, оказывающие влияние на границы системного региона

На основе комбинации пяти перечисленных областей формируется *системный регион* (*system region*). Это своего рода небожитель, ведь на Олимпе (в контексте устройства) ему отведено особое место. Изменять характеристики системного региона из обычных программ невозможно, так как привилегией управлять системным регионом обладает только Windows. Впрочем, помимо своенравного системного региона в контексте устройства нашлось место еще четырем его коллегам, имеющим непосредственное отношение к отсечению: собственно сам регион отсечения, метарегион, регион API и регион Рао [6]. Эти регионы более покладисты и в хорошем настроении подпускают к себе нашего брата – программиста.

Системный регион обслуживается и управляется только операционной системой. Из своих программ мы можем лишь получить его описание.

В качестве основного инструмента, позволяющего настроить границы области, в которых будет осуществляться вывод, выступает *регион отсечения* (*clipping region*). Если полагать, что системный регион описывает первый уровень отсечения, то можно утверждать, что регион отсечения формирует следующий – второй уровень. В момент старта приложения регион отсечения не определен, и зона вывода ограничивается только средствами системного региона. Мы можем создать регион отсечения и определить его границы. В результате этих действий на область вывода накладываются дополнительные ограничения.

Редко, но случается, что для определения области вывода перестает хватать возможностей региона отсечения. В таких случаях на помощь региону отсечения призывают *метарегион*. Метарегион способен временно возложить на себя обязанности региона отсечения, высвободив последний из процесса. Воспользовавшись услугами метарегиона, программист сможет переконфигурировать регион отсечения и вновь включить его в процесс.

Таким образом, мы готовы сформулировать основное правило отсечения в Windows GDI. Область вывода в контексте устройства определяется областью пересечения трех регионов: системного, отсечения и метарегиона (рис. 10.2).

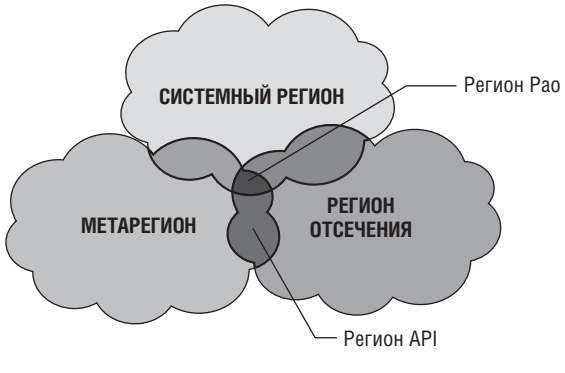


Рис. 10.2. Взаимосвязь регионов контекста устройства

Нам осталось обсудить еще два региона. Регионы API и Pao предназначены для ускорения процесса вывода и находятся под управлением GDI. *Регион API* формируется на пересечении региона отсечения с метарегионом. В свою очередь, *регион Pao* (назван в честь программиста, предложившего хранить его в контексте устройства) представляет собой пересечение системного региона и региона API [6].

В отличие от системного региона, который создается вместе с контекстом устройства, регион отсечения и метарегион изначально не существуют.

Окно нестандартной формы

Прежде чем мы всерьез приступим к исследованию функций GDI, управляющих процессом отсечения, продемонстрируем весьма поучительный пример из жизни механизма отсечения. Мы попробуем изменить очертания стандартного прямоугольного окна, такие «подвиги» в некоторых книгах по программированию называют чуть ли не хакерством, хотя в реальности все дело только в ловкости рук... Говоря точнее, результат достигается за счет упоминавшейся на первой странице главы функции API:

Win API
```
function SetWindowRgn(Wnd: HWND; Rgn: HRGN; Redraw: BOOL): Integer;
```

Эта функция назначит для окна `Wnd` регион отсечения `Rgn`. Последний параметр в состоянии `true` заставит GDI немедленно перерисовать окно, а в состоянии `false` отложит перерисовку до поступления команды от ОС. Порядок работы с функцией представлен в следующей процедуре:

```
Procedure SetCustomRegion(Wnd : HWnd);
var Rgn1,Rgn2,Rgn3:HRGN;
    Rect:TRect;
begin
GetWindowRect(Wnd,Rect);
Rgn1:=CreateRectRgn(0,0,Rect.Right-Rect.Left,(Rect.Bottom - Rect.Top) div 2);
Rgn2:=CreateEllipticRgn(0,0,Rect.Right-Rect.Left,Rect.Bottom - Rect.Top);
Rgn3:=CreateRectRgn(0,0,0,0);
   CombineRgn(Rgn3,Rgn1,Rgn2,RGN_OR);
   SetWindowRgn(Window,Rgn3,true);
DeleteObject(Rgn1);
DeleteObject(Rgn2);
DeleteObject(Rgn3);
end;
```

Мы изменяем форму окна `Window` сразу после его создания (код следует вызывать в момент обработки сообщения `WM_CREATE`). Кроме того, о процедуре целесообразно вспоминать во время изменения размеров окна (`WM_SIZE`) и во время восстановления из свернутого состояния (`WM_SHOWWINDOW`). Новые контуры окна (рис. 10.3) складываются из комбинации двух регионов – прямоугольного `Rgn1` и региона эллиптической формы `Rgn2`.

Рис. 10.3. Нестандартная форма окна

Обращение к регионам контекста устройства

Изучение возможностей GDI, нацеленных на обслуживание регионов, мы начнем с функций, позволяющих обратиться к регионам. Чаще всего нас будет интересовать регион отсечения:

Win GDI
```
Function GetClipRgn(DC : HDC; Rgn : HRGN) : Integer;
```

Функция возвратит копию региона отсечения, принадлежащего контексту `DC`. Особо акцентирую внимание на слове «копия». Попытка изменить геометрические размеры этого региона, воздействуя на регион, описываемый дескриптором `Rgn`, обречена на неудачу. Если в текущем контексте регион отсечения пока не определен, то функция возвратит нулевое значение, если таковой имеется, то мы получим единицу. Отрицательное значение свидетельствует об ошибке выполнения.

Иногда вместо дескриптора региона удобнее работать с координатами ограничивающей прямоугольной области. Хорошую помощь в этом способна оказать функция:

Win GDI
```
Function GetClipBox(DC HDC; var Rect : TRect) : Integer;
```

Она возвратит координаты пересечения системного региона и региона отсечения. Вполне естественно, что указанная функция не сумеет точно описать область с неординарной внешностью, она просто вернет координаты ограничивающего прямоугольника в параметр `Rect`. В этом случае о степени сложности обработанного региона можно судить по результату выполнения функции (см. табл. 9.2 из главы 9).

По причине несколько неудачно подобранных имен констант возможно возникновение путаницы в понятиях «пустой регион» и «неопределенный регион». Термин «пустой регион» применяется к регионам, в состав которых не входит ни одна точка. В случае неопределенного региона картина прямо обратная, он включает все точки системного региона.

Для получения информации о метарегионе требуется помощь функции:

Win GDI
```
Function GetMetaRgn(DC:HDC; Rgn:HRGN) : Integer;
```

После выполнения функции дескриптор копии метарегиона окажется в параметре `Rgn`.

Самые существенные возможности по обращению к регионам предоставляет функция:

```
Function GetRandomRgn(DC:HDC; Rgn:HRGN; Num:Integer) : integer;
```

Перед вызовом функции требуется создать регион, например, с помощью рассмотренной в предыдущей главе функции `CreateRectRgn()`. Последний параметр определяет, копию какого региона мы хотим получить. В случае успеха функция возвратит 1.

В документации SDK утверждается, что `GetRandomRgn()` появилась на свет исключительно ради того, чтобы мы смогли получить доступ к копии системного региона отсечения. С этой целью в последний параметр функции всегда передается константа `SYSRGN`. Такое описание несколько настораживает – какой смысл внедрять в функцию параметр ради того, чтобы он всегда был равен `SYSRGN`? Любого поневоле начнут терзать смутные сомнения в том, что официальная документация Microsoft нас снова не водит за нос, тем более, что константе `SYSRGN` соответствует значение 4. Почему именно 4, а не 0 или, скажем, 33? Но это еще не все! Не знаю, в знак протеста или, наоборот, в знак солидарности с Microsoft, программистами Borland функция `GetRandomRgn()` в состав модуля Windows среды Delphi 2007 (как, впрочем, и всех ее предшественниц) вообще включена не была. Действительно, функцией больше, функцией меньше, какая, собственно, разница?

Впрочем, недоработка Borland устраняется в два счета. Достаточно создать свой собственный модуль, экспортирующий незаслуженно забытую функцию `GetRandomRgn()` из библиотеки gdi32.dll.

```
unit GDI32_ADD;

interface
uses Windows;
Const CLIPRGN=1; METARGN=2; APIRGN=3; SYSRGN=4;

function GetRandomRgn(DC: HDC; Rgn: HRGN; Num : Integer): Integer; stdcall;

implementation
function GetRandomRgn; external gdi32 name 'GetRandomRgn';
end.
```

Заметьте, что в модуле объявлены целые четыре константы (а не одна, как рекомендует SDK). Оказывается, в силу своей природной скромности программисты Microsoft решили умолчать, что кроме обращения к системному региону функция `GetRandomRgn()` способна получить данные региона отсечения (константа `CLIPRGN=1`), метарегиона (`METARGN=2`) и региона API (`APIRGN=3`).

Определение региона отсечения

Наконец мы пришли к самому интересному – рассмотрению функций, предназначенных для определения региона отсечения для графического устройства. Наиболее простое решение предоставляет функция:

Win GDI `Function SelectClipRgn(DC : HDC; Rgn : HRGN):Integer;`

Функция манипулирует двумя параметрами: дескриптором контекста устройства DC и дескриптором уже существующего региона Rgn (заранее созданного другими функциями GDI, например: CreateRectRgn, CreateRectRgnIndirect, CreateRoundRectRgn и т. п.). Вызов функции превратит обычный регион Rgn в регион отсечения, ко всему прочему функция вернет код результата операции (табл. 9.2).

Очередной небольшой пример демонстрирует порядок назначения региона отсечения для окна с дескриптором контекста устройства DC и клиентской областью ClientRect.

```
Procedure GDI_PAINT(DC:HDC;ClientRect:TRect);
var ClipRgn:HRGN;
begin
ClipRgn:=CreateRoundRectRgn(ClientRect.Left+30,  ClientRect.Top+30,
                            ClientRect.Right-30, ClientRect.Bottom-30,
                            40,40);
SelectClipRgn(DC,ClipRgn);
{заливка региона}
FillRect(DC,ClientRect,GetStockObject(LTGRAY_BRUSH));
{чертим диагонали}
MoveToEx(DC,0,0,nil); LineTo(DC,ClientRect.Right,ClientRect.Bottom);
MoveToEx(DC,ClientRect.Right,0,nil); LineTo(DC,0,ClientRect.Bottom);
DeleteObject(ClipRgn);
end;
```

Нами создается регион отсечения в форме прямоугольника со скругленными углами, отступающий на 30 пикселов от границ клиентской области окна. После заливки региона светло-серой кистью мы пытаемся провести две диагональные прямые из угла в угол окна. По рис. 10.4 видно, что вывод линий начинается лишь в тот момент, когда они появляются над областью региона отсечения. Для большей убедительности при описании оконного класса

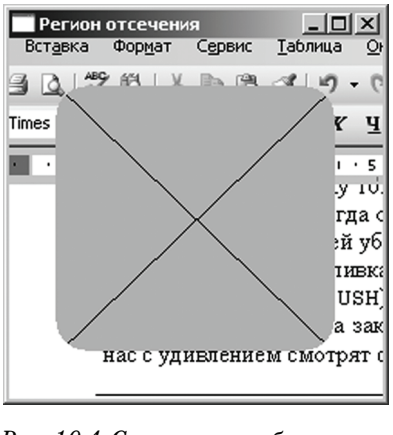


Рис. 10.4. Сокращение области вывода за счет отсечения

я указал, что заливка окна должна производиться кистью-пустышкой (NULL_BRUSH). В итоге границы клиентской области выпали из процесса закраски, и, как следствие, сквозь окно на нас с удивлением смотрят строки этой главы...

Вне зависимости от текущих настроек страничных или мировых координат регионы отсечения всегда задаются в координатах устройства. Эта система координат рассматривается подробно в главе 11 «Системы координат и режимы отображения». А пока нам достаточно знать, что начала координат устройства расположены в левом верхнем углу. Ось X направлена слева направо, ось Y – сверху вниз.

Комбинирование регионов

По сравнению с SelectClipRgn() несколько большими полномочиями в формировании региона отсечения обладает функция:

```
Function ExtSelectClipRgn(DC : HDC; Rgn : HRGN; Mode : Integer) : Integer;
```

Она позволяет комбинировать текущий регион отсечения контекста и регион, описанный параметром Rgn. Особенности объединения регионов определяют значения параметра Mode. В допустимый перечень значений входят константы RGN_AND, RGN_OR, RGN_XOR, RGN_DIFF и RGN_COPY, с которыми мы уже встречались в прошлой главе при рассмотрении функции CombineRgn (см. табл. 9.1.).

В некоторой степени повторением ExtSelectClipRgn() являются функции:

```
Function IntersectClipRect(DC : HDC;
         LeftRect, TopRect, RightRect, BottomRect : integer) : integer;
```

и

```
Function ExcludeClipRect(DC : HDC;
         LeftRect, TopRect, RightRect, BottomRect : integer) : integer;
```

Первая из них создает регион отсечения путем объединения текущего региона и прямоугольной области, описываемой параметрами LeftRect, TopRect, RightRect, BottomRect. Вторая функция исключает из текущего региона отсечения эту прямоугольную область. Таким образом, функция IntersectClipRect() работает в режиме RGN_AND, а ExcludeClipRect() – RGN_DIFF. По завершении работы обе функции возвращают код выполнения, представленный в табл. 9.2.

Траектория в качестве региона отсечения

На роль прародителя региона отсечения вполне может претендовать и траектория:

```
Function SelectClipPath(DC : HDC; Mode  : Integer) : boolean;
```

Перед обращением к функции обязательно убедитесь, что контекст содержит траекторию. Траектория не заменит, а дополнит текущий регион отсечения. Особенности взаимной интеграции траектории и действующего ре-

гиона отсечения определяются во втором параметре функции. Это константы, представленные в табл. 9.1.

Смещение региона

При необходимости текущий регион отсечения может быть перемещен.

Win GDI
```
Function OffsetClipRgn(DC : HDC; XOffset, YOffset : integer) : integer;
```

Направление и расстояние перемещения по осям X и Y соответственно назначаются параметрами `XOffset` и `YOffset`.

Проверка вхождения в регион отсечения

Иногда требуется выяснить, принадлежит ли точка (X,Y) текущему региону отсечения. Для этой цели в состав GDI включена функция:

Win GDI
```
Function PtVisible(DC : HDC; X,Y : integer) : boolean;
```

Она возвратит `true`, если точка входит в состав региона отсечения контекста.

Признаком того, что хотя бы одна точка из множества точек прямоугольной области `Rect` входит в текущий регион отсечения, станет значение `true`, возвращаемое функцией:

Win GDI
```
Function RectVisible(DC : HDC; Const Rect : TRect) : Boolean;
```

Определение метарегиона

В момент рождения контекста устройства метарегиона не существует. Система вполне справедливо полагает, что для организации отсечения нам достаточно услуг системного региона и региона отсечения. Но вполне реально стечение обстоятельств, когда при решении графической задачи повышенной сложности программисту перестанет хватать возможностей перечисленных регионов. Тогда дополнительный уровень отсечения можно ввести с помощью метарегиона.

Метарегион способен ненадолго взять на себя обязанности региона отсечения и тем самым высвободить последний для определения новых границ отсечения.

Так как при старте приложения метарегион отсутствует, то наша работа должна начинаться с его определения. Для этого подойдет функция:

Win GDI
```
Function SetMetaRgn(DC:HDC) : Integer;
```

Функция непритязательна к параметрам и удовлетворится одним лишь дескриптором контекста, в рамках которого и формируется наш регион. Для того чтобы новоиспеченный метарегион не отсек бы чего-нибудь лишнего, он точно повторит границы текущего региона отсечения и сбросит его. Таким образом, в момент появления метарегиона результирующая область отсечения

в контексте не изменится. Все будет оставаться так, как было до тех пор, пока мы вновь не начнем управлять регионом отсечения, например не переместим его в сторону, как показано в следующем листинге:

```
Procedure GDI_PAINT(DC:HDC;Rect:TRect);
var  Brush:HBrush;
     Rgn:HRGN;
     SavedDC:Integer;
begin
SavedDC:=SaveDC(DC);
//исходный регион представляет собой правильный круг
Rgn:=CreateEllipticRgn(10,10,200,200);
//1. Работа с метарегионом
  SelectClipRgn(DC,Rgn); //на основе круга создаем регион отсечения
  SetMetaRgn(DC); //передаем данные в метарегион
  Brush:=CreateHatchBrush(HS_DIAGCROSS,$FF0000);
  FillRect(DC,Rect,Brush); //закрашиваем всю клиентскую область
  DeleteObject(Brush);
  //теперь область метарегиона закрашена наклонными линиями

//2. Вновь работаем с регионом отсечения
  SelectClipRgn(DC,Rgn); //создаем регион на базе круга (10,10,200,200)
  OffsetClipRgn(DC,50,50); //смещаем регион вправо вниз
  Brush:=CreateHatchBrush(HS_CROSS,$00FF00);
  FillRect(DC,Rect,Brush); //проводим заливку клиентской области новой кистью
  DeleteObject(Brush);
  //регион закрашен вертикальными и горизонтальными линиями
RestoreDC(DC,SavedDC);
DeleteObject(Rgn);
end;
```

Исходный метарегион представляет собой правильный круг с координатами (10,10,200,200). Создав метарегион, мы направляем GDI команду залить штриховкой из наклонных линий всю клиентскую часть окна. Однако система смогла закрасить лишь эллиптическую область, входящую в состав метарегиона, это означает, что метарегион взял на себя ответственность по определению границ отсечения. Во второй половине листинга мы вновь создаем регион отсечения (его основой точно так же служит правильный круг с теми же координатами, что и у метарегиона). С этого момента отсечение определяется не только системным регионом и метарегионом, но и границами региона отсечения. Для усложнения эксперимента мы перемещаем регион отсечения вправо и вниз на 50 пикселов. После сдвига графической системе вновь ставится задача закрасить всю клиентскую область окна, но теперь кистью с вертикальной и горизонтальной штриховкой.

Взгляните на снимок окна (рис. 10.5) – в итоге система смогла закрасить новой кистью только область, находящуюся на пересечении метарегиона и региона отсечения. Все получилось точно так, как и было обещано на первых страницах главы (см. рис. 10.2), когда утверждалось, что реальная область отсечения определяется тремя регионами: системным, отсечения и метарегионом.

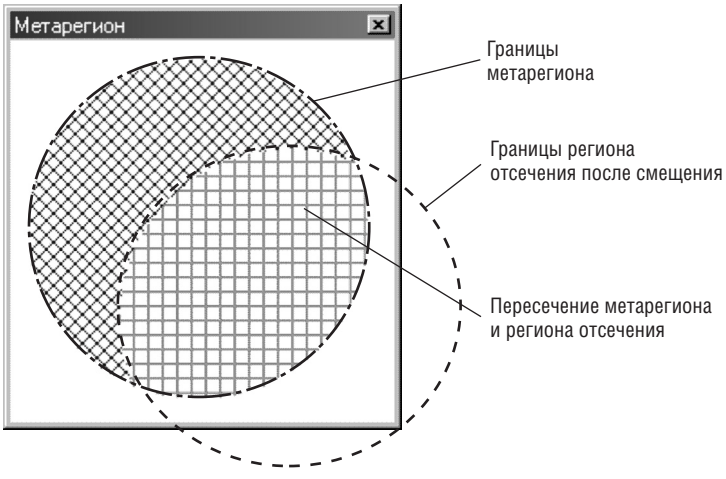


Рис. 10.5. Работа с метарегионом

К недостаткам метарегиона стоит отнести невозможность освободить контекст устройства от услуг ранее созданного метарегиона. Поэтому операции с метарегионом рекомендуется осуществлять в обрамлении функций `SaveDC()` – `RestoreDC()`*, позволяющих возвращать контекст к предыдущему (ранее записанному) состоянию.*

Перерисовка региона

Для принудительного однократного обновления региона требуется обратиться к функции:

Win GDI

```
Function InvalidateRgn(Wnd : HWND; Rgn : HRGN;  Erase : Boolean) :
Boolean;
```

Благодаря вызову `InvalidateRgn()` регион `Rgn` добавляется к текущему перерисовываемому региону окна. Обязательно указание дескриптора окна `Wnd`, потому что обновление будет происходить в рамках обработки сообщения `WM_PAINT`, отправленного к этому окну. Если в последний параметр передать значение `true`, то обновляемой области окна система отправит сообщение `WM_ERASEBKGND`, это стимулирует предварительное стирание области `Rgn`.

С точностью наоборот действует функция:

Win GDI

```
Function ValidateRgn(Wnd : HWND; Rgn : HRGN) : Boolean;
```

Она позволяет убрать регион `Rgn` из обновляемой области окна. Обе функции в случае успешного выполнения возвращают `true`.

```
WM_COMMAND :
Begin
```

```
    Rgn:=CreateRectRgn(50,50,100,100);
    DC:=GetDC(Window);
    InvalidateRgn(Window,Rgn,False);
    FillRgn( DC, Rgn, GetStockObject(GRAY_BRUSH));
         ValidateRgn(Window,Rgn);
         DeleteObject(Rgn);
         ReleaseDC(Window,DC);
end;
```

В GDI имеется еще пара функций, нацеленных на выполнение схожих задач. Это функции `InvalidateRect()` *и* `ValidateRect()`*. Их единственное отличие в том, что вместо региона они оперируют прямоугольными областями* `TRect`.

РЕЗЮМЕ

Область вывода в контексте устройства определяется областью пересечения трех регионов: системного, отсечения и метарегиона. Процедура ограничения области, доступной для вывода изображения, называется отсечением. Благодаря отсечению система предотвращает излишний расход системных ресурсов на обслуживание графики в невидимых участках окна.

11

Системы координат и режимы отображения

Работая с современными текстовыми и графическими редакторами, мы уже давно не удивляемся их сервисным возможностям. Без особых хлопот, фактически одним щелчком кнопки мыши, мы подбираем приемлемый масштаб рисунка, выводим на экран требуемую часть изображения, увеличиваем понравившуюся фотографию или перемещаем по экрану страницу с текстом. Воспринимая доброжелательный сервис приложений как данность, мы даже не задумываемся о том, какие процессы протекают в недрах операционной системы. Но то, что простительно для рядового пользователя, ни в коем случае не допустимо для разработчика программного обеспечения. Поэтому в этой главе мы поговорим об одной из интереснейших способностей графической подсистемы Windows – возможности управления областью вывода изображения. Это триединая задача, включающая:

1. Определение части рисунка, которую мы рассчитываем отобразить.
2. Выбор места вывода этого изображения.
3. Настройку единицы измерения.

Свой рассказ мы начнем с обсуждения координатного пространства графических устройств Windows.

Современные операционные системы Microsoft Windows XP и Vista для определения областей на поверхности вывода используют четыре системы координат:

1. Мировые координаты.
2. Страничные координаты.
3. Координаты устройства.
4. Физические координаты.

Перечисленные координатные системы трудятся на наше благо и позволяют решать сложнейшие графические задачи. Какие? Исследование начнем с первоначальных понятий о назначении координатных систем.

Мировые координаты – это ключевая система координат для работы графической системы GDI+ в Windows XP и Vista, они предназначены для осуществления двумерных аффинных преобразований (операций вращения растра, сдвига, скручивания и т. п.). Такие преобразования часто используются в современной компьютерной графике. В мировом пространстве может быть адресовано $2^{32}\times2^{32}$ элементов. Эта система координат не поддерживается в устаревших Windows 95/98/Me.

Страничные координаты соответствуют координатам любой области, обладающей контекстом устройства. Например, в проектах Delphi, построенных на фундаменте VCL, контекстом описывается поверхность формы (класс `TForm`) и поверхность области для рисования (класс `TPaintBox`). Это система координат, устанавливаемая по умолчанию для всех контекстов во всех версиях Windows, так что, даже не подозревая этого, все свои графические задачи мы до сих пор решали именно в этой системе. Страничная система координат, так же как и мировая, – 32-разрядная, т. е. состоит из 2^{32} адресуемых элементов по вертикали и 2^{32} элементов по горизонтали.

Координаты устройства определяются техническими характеристиками аппаратного устройства, с которым работает Windows. Здесь в качестве единицы измерения всегда применяется пиксел. Ориентация координат также задана жестко: слева направо для оси X и сверху вниз для оси Y. Начало отсчета координат расположено в верхнем левом верхнем углу. Размерность координатной системы составляет $2^{27}\times2^{27}$ пикселов.

Физические координаты напрямую используются драйвером графического устройства и явным образом недоступны из обычных программ. Физическая система координат представляет собой матрицу пикселов размерностью $2^{27}\times2^{27}$ точек.

Во время программирования средствами функций GDI нам доступны первые три из четырех перечисленных выше координатных пространств. Право обращения к физическим координатам предоставлено только драйверу устройства.

Сразу запомним, что все четыре координатные системы взаимоувязаны. И, если мы что-то нарисуем в мировых координатах, прежде чем изображение появится на графическом устройстве (мониторе или принтере), Windows последовательно преобразует рисунок из мировых координат в страничные, затем из страничных в координаты устройства, и в самом конце (опираясь на поддержку драйвера графического устройства) трансформирует координаты устройства в физические.

Почти каждое из преобразований имеет свое название. Так, преобразование из мировых координат в страничные называется *мировым преобразованием*, в свою очередь преобразование из страничной координатной системы в координаты устройства именуют *страничным преобразованием*.

Еще одно весьма важное замечание – единицей измерения в мировых и страничных системах координат могут выступать не только пикселы, но и миллиметры, сантиметры, дюймы, метры, пункты и даже аршины (если нам так захочется). Поэтому, работая в мировых и страничных системах, расстояния откладывают в *логических единицах измерения*. Практически все методы GDI требуют передачи логических координат, обретя эти значения, операционная система найдет способ трансляции их в координаты устройства.

Довольно часто в литературе страничные координаты называют логическими координатами. Это не вполне корректно, так как логические единицы измерения применяются и в мировых координатах. Чтобы избежать путаницы, мы эту традицию поддерживать не станем. Поэтому, встретив в тексте этой книги термин «логические координаты», вы должны понимать, что речь ведется и о страничной, и о мировой координатных системах.

Взаимные преобразования координат

Каждое окно Windows (будь это рабочий стол, главное или дочернее окно приложения, окно элемента управления) обладает индивидуальной координатной системой. При позиционировании какого-либо объекта (например, окна программы) относительно рабочего стола Windows говорят, что работа осуществляется в *экранных координатах*. При работе с окном применяют еще два термина: *координаты рабочей области окна* и *полные координаты окна*. Полные координаты окна определяют окно программы целиком, включая заголовок, меню, полосы прокрутки и рамку окна. Координаты рабочей области описывают только непосредственно внутреннюю поверхность окна, называемую рабочей (клиентской) областью. Все три системы координат отличаются физическим расположением точки начала отсчета координат (0,0) по умолчанию. Для экранных координат это левая верхняя точка экрана. Для координат рабочей области это левая верхняя точка рабочей области окна. Началом координат для полных координат окна выступает левый верхний угол рамки окна (рис. 11.1).

В наших программах в рамках координатного пространства (например, рабочей области окна) могут использоваться мировые и страничные координаты.

Одновременное сосуществование нескольких координатных пространств породило необходимость быстрого взаимного преобразования значений координат. Если речь идет о трансформации координат точки из рабочей (клиентской) области окна в экранные координаты, то стоит воспользоваться функцией:

Win GDI

```
Function ClientToScreen(Wnd: HWND; var pPoint : TPont):LongBool;
```

Здесь `Wnd` – дескриптор окна, рабочие координаты которого мы планируем конвертировать. Переменная `pPoint` двойного назначения, при обращении к методу в нее направляются координаты интересующей нас точки, после выполнения функции в `pPoint` окажутся преобразованные координаты.

Рис. 11.1. Три пространства координат устройства с точками начала отсчета координат по умолчанию

Для решения обратной задачи – по преобразованию экранных координат в клиентские – понадобится помощь функции из «зазеркалья»:

Win GDI
```
Function ScreenToClient(Wnd: HWND; var pPoint : TPont):LongBool;
```

Назначение параметров те же самые, с той лишь разницей, что теперь перед вызовом функции в `pPoint` передаются экранные координаты, а в ответ мы получаем клиентские.

Представленный ниже пример научит нас не только транслировать координаты, но и продемонстрирует порядок обслуживания сообщения `WM_MOUSEMOVE`, генерируемого во время перемещения указателя мыши над окном. Вместе с этим сообщением в адрес окна направляются и клиентские координаты указателя мыши. Оба числа передаются в оконную процедуру через 4-байтный параметр `lParam`. Причем значение координаты X хранится в младшем слове, а координаты Y – в старшем слове этого параметра.

```
Var p:TPoint; s:string;
//…
function WindowProc (Window: HWND; Msg: UINT;
```

```
                    wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
  CASE Msg of
    WM_MOUSEMOVE:
     begin
     DC:=GetDC(Window);
       P.X:=LOWORD(lParam); //координата X-  младшие 16 разрядов
       P.Y:=HIWORD(lParam); //координата Y - старшие 16 разрядов

      s:=Format('Клиентские координаты (%d,%d)',[P.X,P.Y]);
      TextOut(DC,0,0,pChar(s),Length(s));
      ClientToScreen(Window,P); //узнаем экранные координаты
      s:=Format('Экранные координаты (%d,%d)',[P.X,P.Y]);
      TextOut(DC,0,22,pChar(s),Length(s));

      ReleaseDC(DC,Window);
     end;
// обработка остальных сообщений
```

Иногда требуется произвести пересчет клиентских координат набора точек, принадлежащих одному окну, в клиентские координаты другого окна.

Win GDI

```
Function MapWindowPoints(WndFrom:HWND; WndTo: HWND;
                         var Points; Count: Cardinal): integer;
```

Функция требует указания дескрипторов окон – участников операции пересчета. Здесь `WndFrom` – окно, которому принадлежат пересчитываемые координаты, `WndTo` – окно, относительно которого будет произведен пересчет. В параметр `Points` направляется массив точек `TPoint`, параметр `Count` определяет, сколько точек требуется обработать. Если в параметр `WndFrom` вместо дескриптора окна передать неопределенный указатель, то функция станет полагать, что преобразуемые координаты принадлежат экрану. Получив `nil` во второй параметр, функция `MapWindowPoints()` координаты рабочей области окна преобразует в экранные координаты.

В библиотеке VCL реализованы близнецы-братья функций `ClientToScreen()` *и* `ScreenToClient()`. *Это одноименные методы класса* `TControl`. *Кроме того, в Delphi описан удобный метод* `ClientToParent` *(аналог* `MapWindowPoints`*), позволяющий транслировать клиентские координаты дочернего элемента управления в клиентские координаты родительского (*`parent`*) компонента.*

Общие характеристики режимов отображения

Практически все функции Windows GDI (и как следствие методы класса `TCanvas` из состава VCL) в качестве параметров требуют передачи *логических координат*. Направляя Delphi следующую строку кода:

```
TextOut(DC, 5,5, pChar('Hello, Word!'),12)
```

мы информируем систему вывода о том, что рассчитываем увидеть надпись «Hello, World!» в точке с координатами (5,5). Но в этом случае совсем необязательно речь ведется о пикселах, как мы привыкли до сих пор. Вполне веро-

ятно, что контекст графического устройства в качестве единицы измерения применяет дюймы, миллиметры или что-то другое. Более того, может оказаться так, что точка начала отсчета координат окажется не в привычном для нас левом верхнем углу рабочей области окна, а где-то в другом месте.

Логические координаты созданы с целью максимального упрощения нашей работы. А операционная система без вмешательства извне сумеет преобразовать логические координаты в *физические координаты* принтера или дисплея. Мы уже привыкли к тому, что по умолчанию система координат берет начало в левом верхнем углу рабочей поверхности – в точке с координатами (0,0). Горизонтальная ось называется осью X и она проведена из крайней верхней точки поверхности рисования в крайнюю правую точку. Ось Y берет начало в той же точке (0,0) и идет вертикально вниз. Теперь пора узнать, что этот самый распространенный режим отображения называется текстовым (в терминах Windows ему соответствует идентификатор `MM_TEXT`). Причина названия режима «текстовым» заключается в том, что оси X и Y системы координат режима `MM_TEXT` проведены точно так, как мы читаем текст, – слева направо, сверху вниз. Как видите, в данном случае физические единицы устройства пиксела полностью совпадают с нашими логическими координатами, однако в Windows, помимо текстового режима реализовано еще целых семь режимов отображения (табл. 11.1.)

Таблица 11.1. Режимы отображения в Windows

Классификация	Режим отображения	Логические единицы	Направления осей
Текстовый режим	`MM_TEXT`	Пиксел	X – вправо Y – вниз
Метрические режимы	`MM_LOMETRIC`	0.1 мм	X – вправо Y – вверх
	`MM_HIMETRIC`	0.01 мм	
	`MM_LOENGLISH`	дюйма (примерно 0.254 мм)	
	`MM_HIENGLISH`	0.001 дюйма (примерно 0.0254 мм)	
	`MM_TWIPS`	1/1440 дюйма (примерно 0.0176 мм)	
Пользовательские режимы	`MM_ISOTROPIC`	Настраиваемые	Настраивается
	`MM_ANISOTROPIC`	Настраиваемые	Настраивается

В первую очередь режимы отображения различаются единицами измерения. Если не брать во внимание настраиваемые пользовательские режимы отображения, то самый точный из режимов – `MM_HIMETRIC`, он позволяет осуществлять операции рисования с шагом в одну сотую миллиметра, самый грубый – `MM_LOENGLISH`, его единица измерения – одна сотая дюйма. Второе

различие между режимами заключается в направлении вертикальной координатной оси Y.

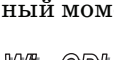
По умолчанию в системе установлен текстовый режим MM_TEXT. *Соответствующая указанному режиму система координат идентична координатным системам устройств отображения. Еще одна особенность текстового режима в том, что его логическая единица измерения полностью совпадает с физической, – это пиксел. Геометрический размер пиксела определяется техническими характеристиками устройства вывода.*

За назначение режима отображения отвечает функция:

```
Function SetMapMode(DC : HDC; MapMode : Integer) : Integer;
```

где DC – дескриптор контекста, MapMode – константа режима отображения. В случае успешного выполнения метод возвращает значение предыдущего режима отображения, в противном случае – 0.

Для определения, какой именно режим отображения используется в данный момент, обращаемся к:



```
Function GetMapMode(DC : HDC) : Integer;
```

Функция требует от нас указать дескриптор контекста DC и возвращает запрашиваемое значение.

Для того чтобы привыкнуть работать в метрических режимах отображения, надо немного потренироваться. На первых порах особенно сложно привыкнуть к поведению вертикальной оси, в предлагаемом примере она может быть направлена не только вниз, но и вверх.

```
Procedure GDI_PAINT(DC:HDC);
  var Rct_R,Rct_G,Rct_B:TRect;
      S:string;   Size: TSize;
Begin
  SetRect(Rct_R,150,-50,350,-200);  //координаты красной области
  SetRect(Rct_G,20,20,100,-100);    //координаты зеленой области
  SetRect(Rct_B,50,50,100,100);     //координаты синей области
//проверяем, какой из переключателей выбран пользователем
     IF SendMessage(rBtn1,BM_GETCHECK,0,0)=BST_CHECKED
     THEN
     SetMapMode(DC,MM_TEXT)
     ELSE
     IF SendMessage(rBtn2,BM_GETCHECK,0,0)=BST_CHECKED
     THEN
     SetMapMode(DC,MM_LOMETRIC)
     ELSE
     IF SendMessage(rBtn3,BM_GETCHECK,0,0)=BST_CHECKED
     THEN
     SetMapMode(DC,MM_LOENGLISH);
//ВЫВОДИМ ТРИ ФИГУРЫ
  SetDCBrushColor(DC,$FF0000); //синий прямоугольник
```

```
    SelectObject(DC,GetStockObject(DC_BRUSH));
    Rectangle(DC,Rct_B.Left,Rct_B.Top,Rct_B.Right,Rct_B.Bottom);

    SetDCBrushColor(DC,$00FF00); //зеленый прямоугольник
    SelectObject(DC,GetStockObject(DC_BRUSH));
    Rectangle(DC,Rct_G.Left,Rct_G.Top,Rct_G.Right,Rct_G.Bottom);

    SetDCBrushColor(DC,$0000FF); //красный прямоугольник
    SelectObject(DC,GetStockObject(DC_BRUSH));
    Rectangle(DC,Rct_R.Left,Rct_R.Top,Rct_R.Right,Rct_R.Bottom);
End;
```

Окно снабжено тремя переключателями: переключатель с дескриптором `rBtn1` включает режим отображения `MM_TEXT`, соответственно, `rBtn2` – режим `MM_LOMETRIC` и `rBtn3` – `MM_LOENGLISH`. В каждом из режимов по единожды заданным координатам структур `Rct_R`, `Rct_G` и `Rct_B` программа пытается нарисовать в клиентской области окна три разноцветных прямоугольника. Но показать все три геометрические фигуры одновременно у программы не получается ни в одном из режимов отображения (см. рис. 11.2 на цв. вклейке, с. 417).

В текстовом режиме за пределами клиентской области оказывается красный прямоугольник и большая часть зеленого. В режиме `MM_LOENGLISH` ситуация обратная – в пределы окна не сумел попасть синий квадрат. Такая «дискриминация» объясняется элементарно. В режиме отображения `MM_LOMETRIC` направление оси Y противоположно направлению этой же оси в текстовом режиме (см. табл. 11.1). Поэтому в штатном режиме `MM_TEXT` возникают проблемы с отображением фигур с отрицательными значениями координаты Y, а во всех метрических режимах за пределами области отображения оказываются прямоугольники с положительными координатами Y. Как же научить программу одновременно отображать все фигуры? Подобную проблему можно решить путем перемещения начала координат, например, в центр окна. Как это сделать, будет продемонстрировано во второй части этой главы.

Настройка страничных координат и координат устройства

Наличие разнообразных режимов отображения добавляет приложению, умеющему работать в страничной системе координат, новые степени свободы. Теперь оно в состоянии определиться с логической единицей измерения, началом и направлением координатных осей. За счет гибкости страничной системы координат пользователь грамотно написанного программного обеспечения приобретает неоспоримые преимущества; теперь ему дозволено указать, какую именно часть изображения и в каком месте области вывода необходимо отобразить. Именно на основе этой идеи построены интерфейсы многих современных текстовых и графических редакторов, программ промышленного черчения, геоинформационных систем и многих других программных пакетов.

Однако для решения перечисленных выше задач совершенно недостаточно умения переключаться между режимами отображения с помощью весьма

полезной, но далеко не всемогущей функции `SetMapMode()`. Ко всему прочему наша программа должна научиться управлять квартетом атрибутов контекста устройства, определяющим:

1. Начало координат окна (*window origin*), обладающего контекстом графического устройства.
2. Начало координат области вывода (*viewport origin*), в которую будут спроецировано изображение из окна.
3. Протяженность окна (*window extents*).
4. Протяженность области вывода (*viewport extents*).

С помощью атрибутов протяженности окна и области вывода программист назначает логическую единицу измерения, причем в анизотропном режиме отображения допускается назначить разные единицы измерения для горизонтальной и вертикальной осей. Кроме того, отношение протяженности вывода к протяженности окна представляет собой коэффициент масштабирования, позволяющий увеличивать (уменьшать) изображение.

Установка перечисленных атрибутов организуется при посредничестве группы функций GDI:

`SetWindowOrgEx()`, `SetViewportOrgEx()`, `SetWindowExtEx()` и `SetViewportExtEx()`.

Сразу запомним, что функции, в наименование которых входит слово «window» (окно), управляют страничной системой координат и в состоянии использовать любые единицы измерения. Функции с названием, содержащим «viewport» (область вывода), обслуживают систему координат устройства и умеют работать только с физическими единицами – пикселами.

Теперь несколько слов о терминах «окно» и «область вывода». *Окно* – это некоторый прямоугольный регион в страничной системе координат (рис. 11.3). На практике это может быть окно программы, написанной на Windows API, клиентская область формы `TForm` стандартного проекта Delphi или поверхность любого другого элемента управления. В отличие от окна, *область вывода* – это прямоугольник, но уже в системе координат устройства; на рисунке она изображена на втором плане. Именно в эту систему координат будет спроецирован образ из страничной системы. Управляя начальными координатами и размерами окна, мы получаем возможность определять отображаемую часть изображения. В свою очередь, настраивая область просмотра, мы назначаем место вывода отображаемой части изображения на поверхности устройства.

Наиболее выдающихся результатов по настройке систем координат можно добиться в пользовательских режимах отображения. Изменяя значения начал координат и протяженностей окна и области вывода, мы сможем смещать, масштабировать изображения и даже выводить их зеркальные отражения. Для того чтобы узнать, как это сделать, уделим немного внимания математике. Итак, при пересчете страничных координат в координаты устройства Windows опирается на пару простейших математических выражений:

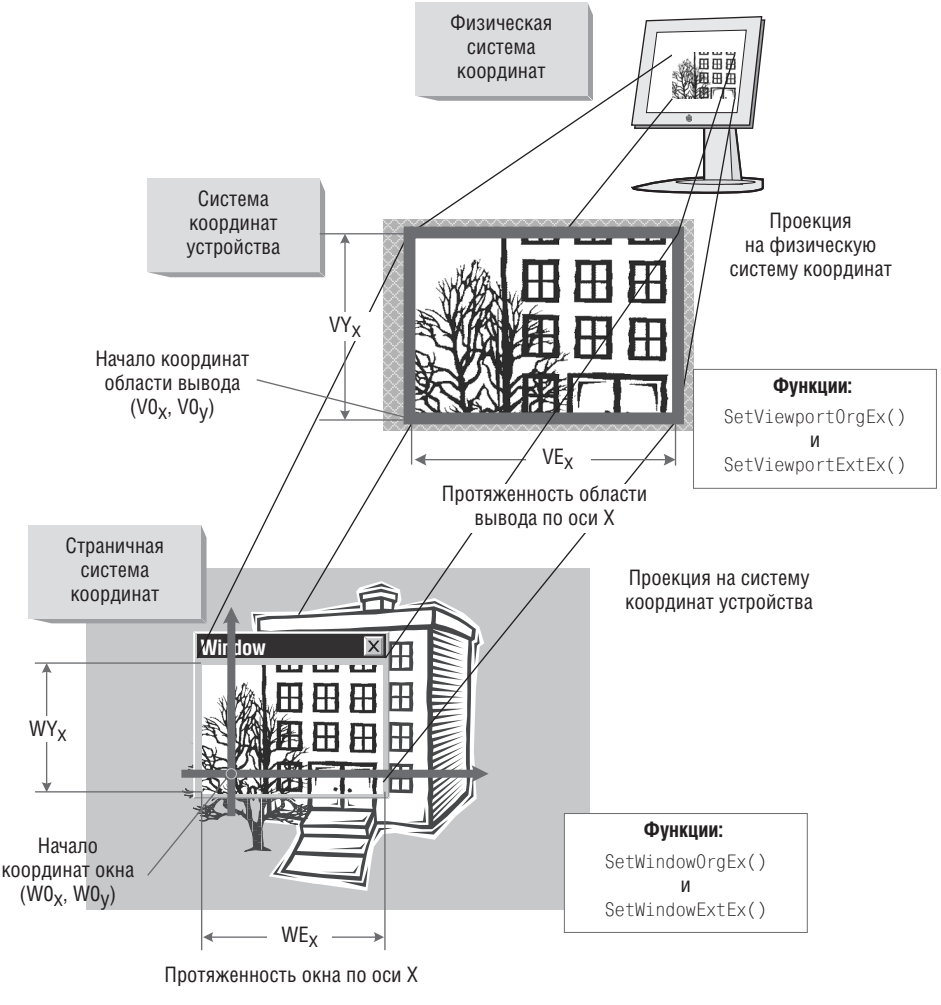


Рис. 11.3. Представление окна и области вывода

Для координаты X:

$$D_x = VO_x + (x - WO_x) \cdot \frac{VE_x}{WE_x} \tag{11.1}$$

Для координаты Y:

$$D_y = VO_y + (y - WO_y) \cdot \frac{VE_y}{WE_y} \tag{11.2}$$

Так как идея построения формул для координат X и Y одна и та же, то прокомментирую только первое выражение. Здесь:

D_x – результат вычисления, физическая координата X аппаратного устройства;

VO_x – начало координат области вывода в координатах устройства;

x – преобразуемая логическая координата x текущего режима отображения;

WO_x – начало координаты оси X окна в логических координатах;

VE_x – протяженность области вывода в координатах устройства;

WE_x – протяженность окна в логических координатах.

*Параметры VE_x и WE_x по отдельности особого внимания не заслуживают, а вот дробь VE_x/WE_x представляет определенный интерес. Отношение протяженности окна к протяженности области вывода – это **коэффициент масштабирования**, используемый при переводе логических координат в координаты устройства.*

Начала координат области вывода и окна

Операционная система позволяет изменять местоположение точки начала отсчета координат как области вывода, так и окна. Для этого предусмотрены две функции Windows GDI. Для установки начала координат области вывода, описываемого контекстом устройства DC, предназначен метод:

```
Function SetViewportOrgEx(DC : HDC; X, Y : Integer;
                          Point : PPoint) : Boolean;
```

Здесь X,Y – координаты нового начала координат области вывода *в единицах измерения устройства*, другими словами, это физические координаты, в качестве единицы измерения которых выступает наш горячо любимый пиксел. Point – указатель на структуру TPoint, в нее сохранится предыдущее значение начала координат. Если эти данные не понадобятся, то синтаксис функции вместо указателя на структуру допускает передачу пустышки nil. В случае успешного выполнения функция порадует нас результатом true.

Следующий пример демонстрирует процесс переноса начала координат в центр проекции окна.

```
WM_PAINT   :
begin
DC:=BeginPaint(Window,PAINTSTRUCT);
SetBKMode(DC,TRANSPARENT);

  {устанавливаем начало координат области вывода в центр окна}
  SetViewportOrgEx(DC,
     PAINTSTRUCT.rcPaint.Right div 2, {начало проекции оси x}
     PAINTSTRUCT.rcPaint.Bottom div 2,{начало проекции оси y}
     nil);

   MoveToEx(DC,-100, 0,nil); LineTo(DC,100,0);//рисуем ось x
   MoveToEx(DC,0,-100,nil);  LineTo(DC,0,100);//рисуем ось y
   TextOut(DC,1,1,PChar('0,0'),3);//подпись начала координат

EndPaint(Window,PAINTSTRUCT);
end;
```

Рис. 11.4. Перенос начала координат в центр проекции окна

В результате выполнения кода мы добились того, что точка (0,0) страничной системы координат стала отображаться в центр системы координат устройства. Координаты новой точки выясняются при помощи элементарных операций деления ширины и высоты клиентской области на два. После переноса начала координат в новое место мы рисуем две линии, точно соответствующие новым координатным осям X и Y. Затем чуть ниже точки начала координат выводим подпись «0,0» (рис. 11.4).

Практически аналогичный результат может быть получен с помощью функции `SetWindowOrgEx()`. Хотя она определяет начала координат вывода окна (а не проекции окна, как ее коллега по цеху `SetViewportOrgEx`), визуальная сторона работы функции та же самая.

Win GDI
```
Function SetWindowOrgEx(DC : HDC; X, Y : Integer; Point : PPoint) :
Boolean;
```

Еще одно отличие функции `SetWindowOrgEx()` от `SetViewportOrgEx()` в том, что здесь аргументы (`x`,`y`) задаются *в логических координатах*. Таким образом, метод позволяет манипулировать удобными для пользователя единицами измерения, например миллиметрами и дюймами. Чтобы добиться того же эффекта, как и в предыдущем примере, потребуется заменить обращение к функции `SetViewportOrgEx()` на следующие строки кода:

```
SetWindowOrgEx(DC,
               -PAINTSTRUCT.rcPaint.Right div 2, {начало оси x}
               -PAINTSTRUCT.rcPaint.Bottom div 2,{начало оси y}
               nil);
```

Заметьте, что в последнем случае в параметры X и Y передаются отрицательные значения.

Настала очередь рассказать о функциях, информирующих программиста о положении начала области координат.

Win GDI
```
Function GetViewportOrgEx(DC : HDC; pPoint : PPoint) : Boolean;
```

и

Win GDI
```
Function GetWindowOrgEx(DC : HDC; pPoint : PPoint) : Boolean;
```

Оба метода у контекста DC собирают информацию о координатах начала области вывода и помещают данные в структуру типа TPoint, на которую ссылается указатель pPoint. Первый метод возвращает значение в координатах устройства, второй – в логических координатах.

Метрические режимы отображения

По сути, метрические режимы изображения – это заранее подготовленные шаблоны, устанавливающие фиксированные протяженности окна и области вывода. Несколько забегая вперед, отмечу, что в метрических режимах отвечающие за протяженность окна и области вывода функции SetWindowExtEx() и SetViewportExtEx() теряют свою профпригодность и, как следствие, не смогут оказать нам помощь, например, в масштабировании результирующего рисунка. На долю программиста остается лишь возможность управления точкой начала координат (функции SetWindowOrgEx и SetViewportOrgEx). У большинства метрических режимов направление оси абсцисс X – слева направо, ось ординат Y направлена снизу вверх.

Специализированный режим MM_TWIPS в первую очередь предназначен для программ компьютерной верстки. В типографском деле за единицу измерений принят пункт, в режиме MM_TWIPS логическая единица соответствует 1/20 пункта, что примерно составляет 1/1440 дюйма (0,0176 мм). Разрешение режима – 1440 dpi.

Режимы MM_LOMETRIC и MM_HIMETRIC предназначены для построения программ, предпочитающих работать с привычными жителям России миллиметрами. Логическая единица грубого режима MM_LOMETRIC соответствует 0,1 мм. Прецизионный режим MM_HIMETRIC различает величину 0,01 мм. Разрешение режима MM_LOMETRIC – 254 dpi, MM_HIMETRIC – 2540 dpi.

Режимы MM_LOENGLISH и MM_HIENGLISH специфичны для программного обеспечения Великобритании и США. В этих странах за основную единицу измерения принят английский дюйм. За единицу измерения режима низкого разрешения MM_LOENGLISH принята величина, равная 0,01 дюйма, соответственно, разрешение режима равно 100 dpi. Режим MM_HIENGLISH на порядок точнее, его логическая единица – 0,001дюйма, разрешение режима – 1000 dpi.

Рассмотрим пример, демонстрирующий порядок рисования сантиметровой и дюймовой линеек. Он состоит из двух фрагментов: процедуры вывода линейки PaintRuler() и привычной для нас процедуры прорисовки окна GDI_PAINT():

```
{процедура вывода горизонтальной линейки}
procedure PaintRuler(DC: hDC);
var X : Integer; S : String;
begin
MoveToEx(DC,0,0,nil);  LineTo(DC,1000,0); //горизонтальная линия линейки
X:=0;
REPEAT
  MoveToEx(DC,X,0,nil); LineTo(DC,X,15);  //риски линейки
  s:=IntToStr(X div 100);
```

```
    TextOut(DC,X,-10,PChar(s),Length(s)); //подпись значения линейки
    INC(X,100); //приращение X на 100 логических единиц
  UNTIL  X>=1000;
  end;
```

Процедура, рисующая линейку, вызывается из тела процедуры `GDI_PAINT()`:

```
Procedure GDI_PAINT(DC:HDC);
begin
SetMapMode(DC,MM_LOMETRIC); //шаг режима 0.1 мм
SetWindowOrgEx (DC, -100, 100, nil); //перенос точки начала координат
PaintRuler(DC);
SetMapMode(DC,MM_LOENGLISH); //шаг режима 0.1 дюйма
SetWindowOrgEx (DC, -40, 100, nil); // перенос точки начала координат
PaintRuler(DC);
end;
```

В рамках процедуры перерисовки окна мы дважды меняем режим отображения. Первоначально выбирается режим `MM_LOMETRIC`, позволяющий работать с логической единицей равной 0,1 миллиметра. Установив начало координат в точке (–100,100), мы обращаемся к `PaintRuler()`. Указанная процедура выводит на поверхности устройства с дескриптором контекста `DC` шкалу с десятком рисок. Шаг между рисками равен 100 логическим единицам. В случае, когда режим отображения `MM_LOMETRIC`, шаг соответствует одному сантиметру, а когда ведется речь о режиме `MM_LOENGLISH`, то шаг составляет один дюйм. Результаты нашего инженерного творчества представлены на рис. 11.5.

Рис. 11.5. Окно с изображением сантиметровой и дюймовой линеек

Пользовательские режимы отображения

Для реализации нестандартных режимов отображения в Windows предусмотрено два пользовательских режима отображения: `MM_ANISOTROPIC` и `MM_ISOTROPIC`. Ключевая особенность пользовательских режимов в том, что они способны изменять направления осей абсцисс и ординат. Еще одна положительная черта режимов в том, что, в отличие от стандартных метрических режимов, пользовательские режимы отображения предоставляют программисту право самостоятельно *назначать логическую единицу измерения*. Тем самым мы получаем дополнительную степень свободы, благодаря которой сможем создавать программное обеспечение, например нацеленное на промышленное черчение или на построение топографических систем с единицами измерения, на порядки отличающимися от фундаментальных дюймов

и миллиметров. Теперь пара слов об отличиях между двумя пользовательскими режимами отображения. Разница между ними отражена в их названиях. Так, имя константы MM_ISOTROPIC произошло от слова *isotropic* – изотропный, одинаковый, MM_ANISOTROPIC – от *anisotropic* – анизотропный, неодинаковый. Изотропный режим отображения разрешает программисту самостоятельно назначать логическую единицу измерения, но только при условии, что логические единицы измерений осей X и Y будут эквивалентны друг другу. Напротив, анизотропный режим допускает, чтобы логические единицы измерений горизонтальной и вертикальной осей были различными, например в качестве единиц измерения для оси X программист может назначить миллиметры, а для Y – километры.

*Из всех режимов отображения только пользовательские (*MM_ISOTROPIC *и* MM_ANISOTROPIC*) предоставляют программисту возможность определять коэффициент масштабирования.*

Изотропный режим MM_ISOTROPIC

Сразу после перевода режима отображения в состояние MM_ISOTROPIC разработчик программного обеспечения приобретает право управлять коэффициентом масштабирования. Для этой цели задействуются возможности функций SetWindowExtEx() и SetViewportExtEx(), совместные действия которых позволят устанавливать требуемые значения протяженности логического окна и рабочей области. В страничной системе координат за размеры окна отвечает функция:

```
Function SetWindowExtEx(DC: HDC; nXExtent, nYExtent : Integer;
                        SIZE : pSize ) : boolean;
```

здесь DC – дескриптор контекста; nXExtent и nYExtent протяженности окна по осям X и Y, соответственно, в логических единицах длины; последний параметр является ссылкой на структуру TSize, предназначенную для хранения предыдущих значений протяженности. Использование последнего параметра необязательно, и синтаксис метода позволяет передавать вместо него неопределенный указатель nil. В случае успешного выполнения метод возвратит значение true.

Размерностью рабочей области в координатной системе устройства ведает функция:

```
Function SetViewportExtEx(DC: HDC; nXExtent, nYExtent : Integer;
                          SIZE : pSize ) : boolean;
```

Аргументы аналогичны параметрам функции SetWindowExtEx() за единственным исключением – аргументы nXExtent и nYExtent определяют протяженности области вывода в единицах длины устройства, т. е. в пикселах.

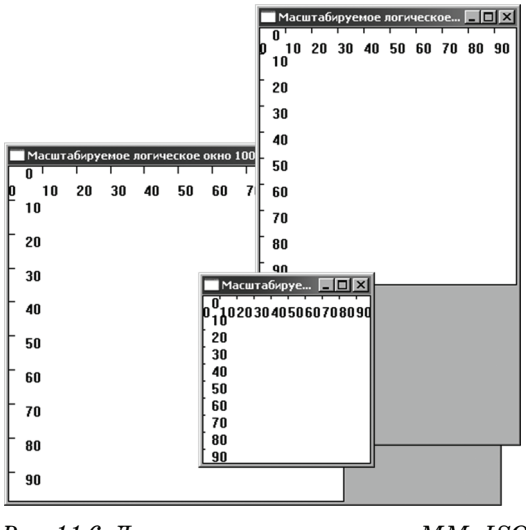
Функции SetWindowExtEx() *и* SetViewportExtEx() *могут применяться только в пользовательских режимах отображения. Причем установка логических размеров окна должна осуществляться ранее, чем настройка размеров рабочей области.*

Представьте себе, что мы формируем свою собственную систему координат, традиционно начинающуюся в левом верхнем углу (0,0) и протяженностью 100 на 100 единиц, т. е. правая нижняя точка имеет координаты (99,99). Взгляните на следующий листинг:

```
Procedure GDI_PAINT(DC:HDC; ClientRect:TRect);
Begin
SetMapMode(DC, MM_ISOTROPIC);    //назначаем изотропный режим отображения
SetWindowExtEx(DC, 99, 99, nil); //указываем протяженность логического окна
{определяем ширину и высоту рабочей области}
SetViewportExtEx(DC, ClientRect.Right, - ClientRect.Bottom, nil);
Rectangle(DC,0,0,99,-99);   //рисуем квадрат со стороной 100 логических единиц
End;
```

Операционная система Windows автоматически осуществляет контроль за тем, чтобы размерность логического окна соответствовала характеристикам рабочей области. Например, в нашем случае в качестве логического окна определен квадрат 100×100 единиц, при изменении размеров рабочей области операционная система «впишет» логическое окно в отведенное ему пространство (рис. 11.6).

Операционная система не допустит вывода изображения за пределами логического окна, и если вы определили протяженность окна в 100 единиц, то попытка нарисовать что-то в точке с логическими координатами, превышающими это значение, не имеет никакого смысла.



В режиме MM_ISOTROPIC *операционная система всегда контролирует и при необходимости исправляет заданные программистом протяженности логического окна и области вывода. Корректировка осуществляется на основе коэффициента сжатия устройства так, чтобы логические единицы измерения по обеим осям имели правильные размеры.*

Рис. 11.6. Логическое окно в режиме MM_ISOTROPIC

Обязательно упомянем пару функций, информирующих программиста о текущих настройках пользовательского режима:

```
function GetViewportExtEx(DC: HDC; var Size: TSize): BOOL;
function GetWindowExtEx(DC: HDC; var Size: TSize): BOOL;
```

Уже по названиям можно догадаться, что `GetViewportExtEx()` возвращает данные о протяженности области просмотра, а `GetWindowExtEx()` – о протяженности окна. Результаты работы функций передаются в параметр `Size`.

Масштабирование изображения

Еще раз на минутку вернитесь к концу подраздела «Настройка координат» и взгляните на выражения (11.1) и (11.2) пересчета страничных координат в координаты устройства. Мы тогда уже говорили о существенном влиянии дроби VE/WE на порядок вывода результирующего изображения. Отношение протяженности области вывода к протяженности логического окна определяет коэффициент масштабирования результирующего изображения. Можете вернуться к предыдущему примеру и внести изменения в аргументы функции `SetViewportExtEx()`.

```
SetViewportExtEx(DC, ClientRect.Right div 2,
                   - ClientRect.Bottom div 2, nil);
```

В этих строках я в два раза уменьшил значения параметров, определяющие протяженность рабочей области, что в свою очередь визуально в два раза уменьшит размер логического окна.

Вопросы организации масштабирования в GDI перед программистом ставятся нередко, например при выводе рисунков. В следующем примере мы рассмотрим способ масштабирования векторного рисунка (метафайла Windows). Для сокращения объема нашей работы создадим классическое (опирающееся на библиотеку VCL) приложение Delphi.

Если вы не хотите работать на VCL, то из библиотеки на сайте издательства скачайте аналогичный пример, написанный на Windows API.

В секции частных объявлений упомянем переменную MF, она предназначена для работы с метафайлом в стиле Delphi (см. главу 24 «Растровые и векторные изображения в VCL»). Кроме того, в секции `private` опишем переменные, специализирующиеся на хранении значений протяженности окна и области вывода.

```
private
  MF:TMetaFile;
  WindowExt_X, WindowExt_Y, WindowOrg_X, WindowOrg_Y : integer;
```

На поверхности формы разместим один-единственный компонент – `TComboBox`. В момент создания формы загружаем демонстрационный метафайл, для этого я подготовил достаточно большую картинку размерами 800×800 пикселов и переименовал ее в «home.emf». Моя картинка представляет собой квадрат, но это скорее исключение из правил. На практике размеры сторон могут быть не одинаковы. Поэтому далее в коде программы выясняется, какая из сторон

больше, и ее размер берется для установки протяженности окна (переменные `WindowExt_X`**,** `WindowExt_Y`**). В последних строках события** `OnCreate()` **заполняются строки комбинированного списка значениями масштаба.**

```
procedure TForm1.FormCreate(Sender: TObject);
begin
MF:=TMetaFile.Create; //создаем экземпляр класса метафайла
MF.LoadFromFile('..\GRAPHICS\home.emf'); //загружаем изображение

if MF.Width>MF.Height then  //настройка переменных протяженности окна
    WindowExt_X:=MF.Width else WindowExt_X:=MF.Height;
WindowExt_Y:=WindowExt_X;   //у изотропного режима высота равна ширине

with ComboBox1 do   //компонент TComboBox с перечнем масштабов
    begin
    Clear;
    Items.Add('25%');
    Items.Add('50%');
    Items.Add('100%');
    Items.Add('200%');
    ItemIndex:=2;      //по умолчанию масштаб 100%
    OnChange(sender);  //инициируем пересчет
    end;
end;
```

Любое изменение масштаба приводит к пересчету значений протяженности области просмотра.

```
procedure TForm1.cbScaleChange(Sender: TObject);
begin
{событие OnChange() комбинированного списка - изменился коэффициент
масштабирования}
case cbScale.ItemIndex of  {настройка переменных протяженности области}
  0 : WindowOrg_X:=WindowExt_X div 4; {масштаб 25%}
  1 : WindowOrg_X:=WindowExt_X div 2; {масштаб 50%}
  2 : WindowOrg_X:=WindowExt_X;       {масштаб 100%}
  3 : WindowOrg_X:=WindowExt_X * 2;   {масштаб 200%}
end;
WindowOrg_Y:=WindowOrg_X;
Form1.Repaint;  //перерисовываем форму
end;
```

Сердце программы – событие обновления формы. В нем устанавливается изотропный режим отображения и настраиваются протяженности окна и области просмотра.

```
procedure TForm1.FormPaint(Sender: TObject);
var DC:hDC;
    s:string;
begin
DC:=Form1.Canvas.Handle;

SetMapMode(DC, MM_ISOTROPIC);
SetViewportOrgEx(DC, 0, 0, nil); {начало координат окна в левом верхнем углу}
```

```
SetWindowOrgEx(DC, 0, 0, nil); {начало координат области вывода}

SetWindowExtEx(DC, WindowExt_X, WindowExt_Y, nil);  {протяженность окна}
SetViewportExtEx(DC, WindowOrg_X, WindowOrg_Y, nil);{протяженность области вывода}

form1.Canvas.Draw(0,0,mf);
end;
```

Строки с установкой точки начала координат необязательны и оставлены только для того, чтобы напомнить читателю о возможности управления началом координат. В конце работы программы надо не забыть освободить ресурсы, занятые под метафайл.

```
procedure TForm1.FormDestroy(Sender: TObject);
begin
MF.Destroy;
end;
```

Внешний вид работающего приложения предложен на рис. 11.7.

Специально для организации масштабирования в пользовательских режимах отображения в состав Windows GDI включена функция: `ScaleWindowExtEx()`.

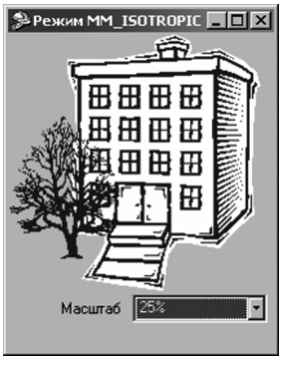


Рис. 11.7. Масштабирование

Отражение изображения

Кроме изменения масштаба изображения пользовательские режимы отображения позволяют добиться эффекта отражения. Например, для элементарной постановки рисунка с ног на голову в событии `OnPaint()` предыдущего примера достаточно изменить всего два значения:

1. Функцией `SetViewportOrgEx()` перенести начало координат окна в левый нижний угол:

```
SetViewportOrgEx(DC, 0, Form1.ClientHeight, nil);
```

2. С помощью `SetViewportExtEx()` направить ось Y снизу вверх:

```
SetViewportExtEx(DC, WindowOrg_X, -WindowOrg_Y, nil);
```

Результат работы предложен на рис. 11.8.

Рис. 11.8. Отражение рисунка

Смещение изображения

Изменяя значения начал координат окна и области вывода, мы сможем сместить изображение. Для этого потребуется в рамках события `OnPiant()` формы отредактировать следующие строки кода:

```
procedure TForm1.FormPaint(Sender: TObject);
//…
SetViewportOrgEx(DC, 0, 0, nil);
SetWindowOrgEx(DC, WindowOrg_X div 3, WindowOrg_Y div 3, nil);
```

В строках процедуры я вернул начало координат окна в верхнюю левую точку клиентской области формы и переместил область вывода влево вверх ровно на треть рисунка. В результате мы получили возможность заглянуть в одно из окон нарисованного дома (рис. 11.9).

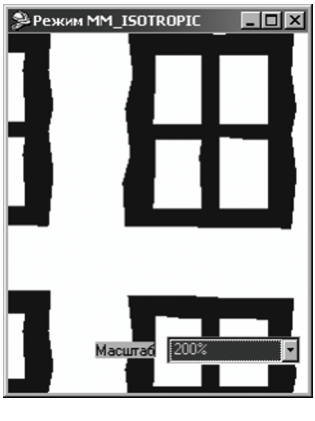


Рис. 11.9. Смещение и масштабирование рисунка

Анизотропный режим MM_ANIISOTROPIC

В изотропном режиме отображения Windows всегда преобразует протяженности области вывода и логического окна таким образом, чтобы они имели одинаковое выражение в физических единицах измерения. В анизотропном режиме Windows такого преобразования не выполняет, другими словами, в режиме MM_ANIISOTROPIC операционная система не создает пропорционального коэффициента сжатия. Изменение протяженности области вывода приводит к сжатию или растягиванию соответствующей стороны логического окна, что искажает изображение.

```
Procedure GDI_PAINT(DC:HDC; ClientRect:TRect);
begin
 SetMapMode(DC, MM_ANISOTROPIC);
 {под область вывода отводим всю клиентскую область ClientRect}
 SetViewportExtEx(DC,
                  ClientRect.Right,
                  -ClientRect.Bottom,
                  nil);
 {определяем логическое окно протяженностью 11 единиц}
 SetWindowExtEx(DC, 10, 10, nil);

 {переносим начала координат в центр клиентской области}
 SetWindowOrgEx(DC,-5,5,nil);
 //рисуем эллипс
 Ellipse(DC,-5,-5,5,5);
end;
```

На рис. 11.10 приведены снимки форм с изображенным на них в режиме MM_ANISOTROPIC эллипсом. Нетрудно заметить, что эллипс приобретает очертания правильного круга только в случае равенства сторон области вывода. Во всех остальных случаях размер логического окна подстраивается под размер клиентской области.

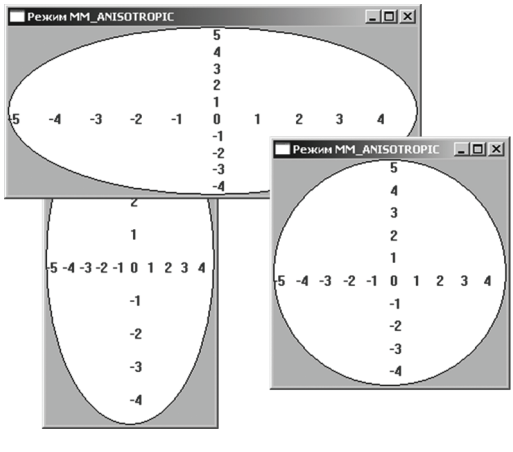


Рис. 11.10. Изображение эллипса в анизотропном режиме отображения

Из всех существующих режимов отображения анизотропный наиболее универсальный, так как не ограничивает настройку размеров окна и области просмотра.

Вне зависимости от направления оси X в пользовательских режимах отображения текстовые строки будут выводиться слева направо.

Альтернативный способ масштабирования в пользовательских режимах отображения

В изотропном и анизотропном режимах отображения предусмотрен еще один способ управления размером результирующего изображения. Настройку коэффициента масштабирования производят за счет четырех множителей и делителей – параметров функции `ScaleWindowExtEx()`:

Win GDI

```
Function ScaleWindowExtEx(DC:HDC; Xnum, Xdenom, Ynum, Ydenom : Integer;
                          Size:pSize):LongBool;
```

Коэффициент умножения назначается раздельно для осей абсцисс и ординат и определяется парой элементарных математических выражений:

$$XNew = \frac{XOld+Xnum}{Xdenom} \quad (11.3)$$

и

$$YNew = \frac{YOld+Ynum}{Ydenom} \quad (11.4)$$

Последний параметр функции представляет собой указатель на структуру, хранящую предыдущие значения множителей. Если эти данные не нужны, то в этот параметр разрешено передавать неопределенный указатель.

РЕЗЮМЕ

В распоряжении Windows 95/98/Me имеются три взаимоувязанных системы координат: страничные, координаты устройства и физические координаты. Само по себе существование нескольких координатных сеток не представляло бы особого интереса, если бы не одна замечательная особенность – наши программы могут управлять режимами отображения (управлять началами координат, настраивать логические единицы измерения). В современные версии Windows введена еще одна система координат, называемая мировыми координатами. Более подробно о ней мы поговорим в очередной главе.

12

Мировые координаты и аффинные преобразования

Как уже говорилось в предыдущей главе, в современных версиях Windows (начиная с Windows NT/2000) появилась четвертая по счету координатная система – *мировые координаты*. Новая система не только позволила с заданной степенью точности описывать истинное положение объектов в пространстве, но и добавила к (и без того отменным) картам GDI еще один козырь. Переведенный в мировые координаты контекст графического устройства приобрел уникальную возможность осуществлять двумерные преобразования на плоскости – проводить однозначное (при необходимости обратимое) отображение одного множества точек в другое. Такая операция основана на математическом аппарате, именуемом *двумерными аффинными преобразованиями*. С практической точки зрения разработчика программного обеспечения аффинное преобразование определяет порядок перехода от прямоугольной декартовой системы координат к назначаемой программистом системе (при необходимости даже косоугольной). Сегодня без этих операций не в состоянии обойтись ни одно современное графическое приложение. Обширный круг решаемых задач включает в себя как банальный перенос изображения с одного места на другое без внесения в него каких-либо искажений, так и сложные процедуры растяжения (сжатия), деформации и вращения.

Перевод контекста в мировые координаты

В первых версиях Windows понятия мировых координат не существовало, видимо, поэтому даже в Windows Vista контекст устройства по умолчанию устанавливается в страничную систему координат. Это утверждение подтвердит функция:

```
Function GetGraphicsMode(DC:HDC):integer;
```

информирующая нас о текущей системе координат контекста устройства с дескриптором DC. В случае успешного выполнения функция вернет одну из двух констант: GM_COMPATIBLE или GM_ADVANCED. Константа GM_COMPATIBLE свидетельствует о том, что контекст установлен в совместимый графический режим (речь ведется о совместимости со старым Win16 GDI API в более не поддерживаемых Microsoft операционных системах Windows 95/98/Me). Соответственно, в совместимом режиме и не может идти речи о мировой системе координат. Значение GM_ADVANCED сигнализирует о том, что система координат контекста настроена на работу в мировых координатах.

Для смены системы координат предназначена функция:

```
Function SetGraphicsMode(DC:HDC; Mode : integer):integer;
```

В первый параметр передается дескриптор контекста графического устройства, во второй – одна из двух вышепоименованных констант.

Собственно процессом преобразования мировых координат в страничные для контекста устройства DC управляет одна-единственная функция GDI:

```
Function SetWorldTransform(DC : HDC; XForm: TXForm) : Boolean;
```

Результат работы функции определяется правилами двумерных аффинных преобразований, передаваемых программистом в параметр XForm.

Аффинные преобразования на плоскости

Прежде чем мы приступим к программированию, позволю себе совсем чуть-чуть развлечь вас математикой. При работе в мировых координатах нам позволяется переносить точку с координатами (x, y) в точку с координатами (x', y') по законам *двумерного аффинного преобразования* [6,9]. Слово «двумерного» указывает на то, что преобразования осуществляются на плоскости. В природе существует и понятие трехмерных аффинных преобразований, но ни GDI, ни GDI+ таких операций не поддерживает. В общем виде алгоритм преобразования можно представить в виде пары несложных формул:

$$x' = x \times eM11 + y \times eM21 + eDx; \qquad (12.1)$$

$$y' = x \times eM12 + y \times eM22 + eDy. \qquad (12.2)$$

Здесь: (x, y) – исходная точка; $eM11$, $eM21$, $eM12$, $eM22$, eDx и eDy – коэффициенты, управляющие преобразованием. Процесс аффинного преобразования математики обычно записывают в матричном виде:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} eM11 & eM21 & eDx \\ eM12 & eM22 & eDy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}. \qquad (12.3)$$

За счет констант *eM*11, *eM*21, ..., *eDy* формируется *матрица преобразования*. Умножение матрицы преобразования на матрицу, содержащую координаты точки (x, y), и возвращает новую координату (x', y'). Самое простое назначение – у элементов матрицы *eDx* и *eDy*. Достаточно одного взгляда на выражения (12.1) и (12.2), чтобы понять, что они лишь определяют смещение результата преобразования по горизонтали и вертикали в текущих логических единицах. Для того чтобы не потерять *eDx* и *eDy*, матрицы (12.3) дополнены третьей строкой, состоящей из нулей и единиц. Перемножение матриц выполняется по законам линейной алгебры. Например, для обычного сдвига координат матрица преобразований должна выглядеть следующим образом:

$$\begin{bmatrix} 1 & 0 & eDx \\ 0 & 1 & eDy \\ 0 & 0 & 1 \end{bmatrix}. \tag{12.4}$$

Подставляя матрицу (12.4) в выражения (12.1) и (12.2), получаем:

$$x' = x + eDx; \tag{12.5}$$

$$y' = y + eDy. \tag{12.6}$$

Это элементарный случай преобразований, представляющий собой обычный сдвиг координатной системы на значения *eDx* и *eDy* (рис. 12.1).

В Windows GDI матрица аффинных преобразований хранится в виде структуры `TXForm`:

```
type
  TXForm = packed record
    eM11: Single;
    eM12: Single;
    eM21: Single;
    eM22: Single;
    eDx : Single;
    eDy : Single;
end;
```

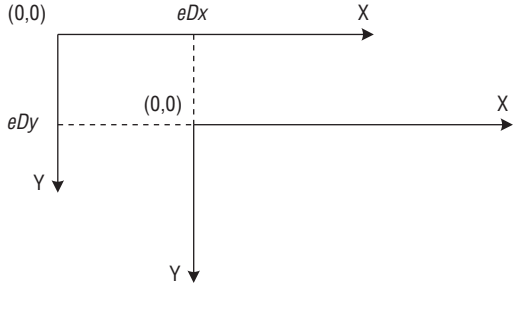


Рис. 12.1. Сдвиг координат

Каждое поле структуры входит в математические выражения, определяющие алгоритм преобразования точек. Назначение двух последних полей структуры (*eDx* и *eDy*) мы уже знаем, они позволят сдвигать координаты. Роль полей *eM*11, *eM*21, *eM*21, *eM*22 гораздо значительнее. Для разъяснения предлагаю обратиться к табл. 12.1.

Таблица 12.1. Назначение полей eM11, eM12, eM21 и eM22 структуры TXForm

Вид операции	**Поля структуры**			
	***eM*11**	***eM*12**	***eM*21**	***eM*22**
Тождественное отображение	1	0	0	1
	$x' = x$ $y' = y$			
Отражение	Отражение по горизонтальной оси -1	0	0	Отражение по вертикальной оси -1
	$x' = -x$ $y' = -y$			
Поворот	Косинус угла $cos(\theta)$	Синус угла $sin(\theta)$	Отрицательный синус $-sin(\theta)$	Косинус угла $cos(\theta)$
	$x' = cos(\theta) \times x - sin(\theta) \times y$ $y' = sin(\theta) \times x + cos(\theta) \times y$			
Масштабирование	Коэффициент масштабирования по горизонтали	0	0	Коэффициент масштабирования по вертикали
	$x' = x \times eM11$ $y' = y \times eM22$			
Поперечная деформация	1	Константа сдвига по вертикали	Константа сдвига по горизонтали	1
	$x' = x + y \times eM21$ $y' = x \times eM12 + y$			

Таким образом, если, например, нам требуется изменить масштаб изображения, то следует передать соответствующие коэффициенты масштабирования в поля `eM11` и `eM22` записи `TXForm`, при этом не забыв записать в остальные поля нулевое значение. Например, следующая матрица преобразования:

$$\begin{bmatrix} 1/2 & 0 & 0 \\ 0 & 1/2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

уменьшит масштаб в два раза по горизонтальной и вертикальной осям. Коэффициенты eM11 и eM22 могут быть и отрицательными, например, значение –1 позволит получить зеркальное отображение.

Допускается комбинировать несколько операций. В следующем листинге предложен пример двумерного аффинного преобразования. Мы деформируем прямоугольник по горизонтальной оси с одновременным масштабированием (увеличением вдвое) по вертикали и горизонтали.

```
Procedure GDI_PAINT(DC:HDC;ClientRect:TRect);
var XForm:TXForm;
    X1,Y1,X2,Y2:integer;
    Pen:HPen;
begin
X1:=10; Y1:=10; X2:=60; Y2:=30;
SetGraphicsMode(dc,GM_COMPATIBLE);//контекст в совместимом режиме

Pen:=CreatePen(PS_DOT,1, $0000FF);
SelectObject(DC,Pen);
Rectangle(DC,X1,Y1,X2,Y2); // рисуем исходный прямоугольник
{заполняем матрицу - назначаем правило аффинных преобразований}
XForm.eM11:=2;    XForm.eM12:=0;
XForm.eM21:=1.5;  XForm.eM22:=2;
XForm.eDx:=0;     XForm.eDy:=0;

SetGraphicsMode(DC,GM_ADVANCED);//перевод контекста в мировую систему координат
SetWorldTransform(DC,XForm);    //аффинные преобразования

SelectObject(DC,GetStockObject(DC_PEN)); //создание нового пера
SetDCPenColor(DC,$FF0000);               //смена цвета пера
Rectangle(DC,X1,Y1,X2,Y2);               //рисуем деформированный прямоугольник
DeleteObject(Pen);
end;
```

Результаты нашего кропотливого труда представлены на рис. 12.2. В итоге изначальный прямоугольник (нарисованный красным пером) не только удвоился в размерах, но и превратился в параллелограмм. Увеличение размеров произошло за счет передачи значения 2 в поля eM11 и eM12 структуры TXForm. Деформация прямоугольника по горизонтали осуществлена благодаря полю eM21, в которое мы направили значение 1.5.

Для поворота изображения нам следует лишь пересчитать значения синуса и косинуса для угла поворота (см. табл. 12.1). Попробуем на практике. Для большего удобства создадим стандартный проект Delphi и воспользуемся компонентом класса TUpDown (он позволит пользователю изменять значение угла). Установите ограничение максимального значения градусов (свойство Max) в 359.

Листинг события OnPaint() формы представлен ниже. Львиная доля кода посвящена не сколько повороту, сколько переносу начала координат в центр формы и прорисовке координатных осей X и Y. Благодаря такой предусмотрительности проект приобретет требуемую наглядность.

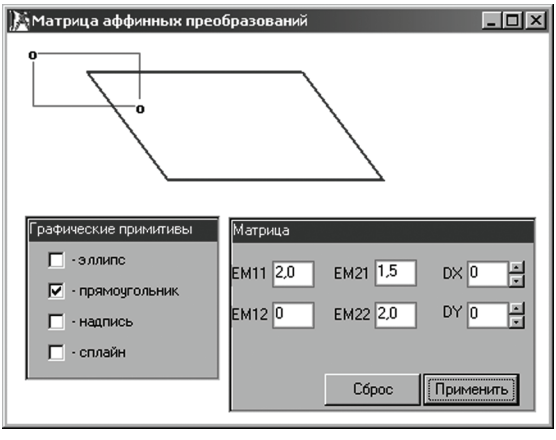


Рис. 12.2. Пример аффинного преобразования

```
procedure TForm1.FormPaint(Sender: TObject);
var dc:hDC;
    XForm:TXForm;
    c,s:real;
begin
DC:=Form1.Canvas.Handle;
SetGraphicsMode(dc,GM_COMPATIBLE);
{начало координат - в центре клиентской области}
SetViewportOrgEx(dc, Form1.ClientWidth div 2, Form1.ClientHeight div 2, nil);
SetMapMode(DC,MM_LOMETRIC);
{рисуем оси абсцисс и ординат}
MoveToEx(DC,-1000, 0,nil); LineTo(DC,1000,0);
MoveToEx(DC,0,-1000,nil);  LineTo(DC,0,1000);
{пересчет градусов в косинус и синус угла}
 c:=COS(DegToRad(UpDown1.Position)); s:=SIN(DegToRad(UpDown1.Position));
{назначаем правило аффинного преобразования}
XForm.eM11:=c;  //косинус угла
XForm.eM12:=s;  //синус угла
XForm.eM21:=-s; //отрицательный синус угла
XForm.eM22:=c;  //косинус

SetGraphicsMode(DC,GM_ADVANCED);        //перевод контекста в мировые координаты
SetWorldTransform(DC,XForm);            //аффинные преобразования
SelectObject(DC,GetStockObject(DC_PEN));//выбор пера
SetDCPenColor(DC,clBlue);               //смена цвета пера
MoveToEx(DC,0,0,nil); LineTo(dc,500,0); //вывод отрезка
end;
```

Обратите внимание, что поворот осуществляется против часовой стрелки. В заключение не забудьте инициировать перерисовку формы (метод `Repaint`) в момент щелчка по компоненту `UpDown1`, иначе вы не увидите никаких результатов аффинных преобразований.

```
procedure TForm1.UpDown1Click(Sender: TObject; Button: TUDBtnType);
begin
Form1.Repaint;
end;
```

Работа листинга продемонстрирована на рис. 12.3.

Возьму на себя смелость напомнить, что функции `Cos()` *и* `Sin()` *в качестве параметра требуют передачи значения угла в радианах, а не в градусах. Для преобразования градусов в радианы (функция* `DegToRad`*) к проекту требуется подключить модуль математики* `Math` *или вспомнить нехитрую формулу из учебника по геометрии:*

$$Радианы = Градусы \times \pi / 180.$$

Контекст устройства никогда не забудет, какое именно мировое преобразование осуществлялось в последний раз. Для того чтобы удостовериться в этом, обратитесь к функции:

Win GDI

```
Function GetWorldTransform(DC : HDC; XForm: TXForm) : Boolean;
```

Она возвратит структуру `XForm` с заполненными значениями предыдущего преобразования.

Если по какой-либо причине мы не можем получить требуемого результата за счет единственного аффинного преобразования, допускается создать их комбинацию. Специально для этого реализована функция:

Win GDI

```
Function CombineTransform(var p1: TXForm; const p2 : TXForm;
                          const p3 : TXForm):Boolean;
```

Здесь константы `p2` и `p3` – первая и вторая структуры `TXForm`; результат их комбинирования окажется в первом параметре – переменной `p1`.

Еще один способ оказать влияние на назначенное в текущий момент преобразование – воспользоваться услугами функции:

Win GDI

```
Function ModifyWorldTransform(DC : HDC; XForm: TXForm;
                              Mode : cardinal) : Boolean;
```

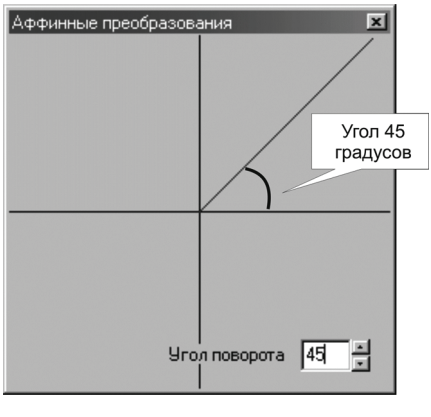


Рис. 12.3. Поворот

Третий параметр функции может принимать одно из трех значений. Тривиальный вариант, когда в параметр Mode передается константа MWT_IDENTITY. В этом случае все остается на своих местах, и функция не оказывает никакого влияния на текущее аффинное преобразование. Все становится гораздо интереснее, когда функция получает одну из двух оставшихся констант: MWT_LEFTMULTIPLY или MWT_RIGHTMULTIPLY. В этом случае, соответственно, осуществляется левое или правое преобразование. Другими словами, текущее преобразование умножается на содержимое параметра XForm справа или слева. Дело в том, что в аффинных преобразованиях (в отличие от обычной арифметики) перестановка множителей местами оказывает существенное влияние на результат.

Пример «Стрелочные часы»

Дабы скучная теория не расходилась с увлекательной практикой, немного потрудимся за клавиатурой. Наша задача – используя возможности аффинных преобразований, создать приложение «Стрелочные часы». Начнем с оконной процедуры программы:

```
const ClockTimer = 1; //идентификатор таймера
Var …
      Hour, Min, Sec, MSec: Word;//часы, минуты, секунды и миллисекунды
      eHour,eMin,eSec :DOUBLE;   //углы поворотов стрелок часов

function WindowProc (Window: HWND; Msg: UINT;
                     wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
  case Msg of
  WM_CREATE     :  SetTimer(Window, ClockTimer, 1000, nil);
  WM_SHOWWINDOW :  SendMessage(Window,WM_TIMER,0,0);
  WM_TIMER      :  begin
                  {пересчет времени в значения углов поворота стрелок}
                    DecodeTime(Time,Hour, Min, Sec, MSec);
                    eSec:=Sec*pi/30;
                    eMin:=Min*pi/30;
                    eHour:=Hour*pi/6+Min/60*pi/6;
                    InvalidateRect(Window,nil,false);
                    end;

       WM_PAINT   : begin
                     DC:=BeginPaint(Window,PAINTSTRUCT);
                     PaintClock(DC,PAINTSTRUCT.rcPaint); //рисуем часы
                     EndPaint(Window,PAINTSTRUCT);
                     end;
       WM_DESTROY : begin
                     KillTimer(Window, ClockTimer); //таймер более не нужен
                     PostQuitMessage (0);
                     end;
  end;
  WindowProc := DefWindowProc (Window, Msg, WParam, LParam);
end;
```

Для организации отсчета времени в программе создается таймер. Для этого вызывается функция Windows API `SetTimer()`. С этого момента приложение с периодичностью один раз в 1000 миллисекунд начинает получать сообщение `WM_TIME`. При обработке сообщения декодируем системное время (функция `DecodeTime`), заполняя заранее подготовленные переменные `Hour`, `Min`, `Sec` и `MSec` (часы, минуты, секунды и миллисекунды, соответственно). На следующем этапе программа конвертирует полученные значения в углы поворота стрелок часов. Например, углы поворота минутной и секундной стрелок (переменные `eMin` и `eSec`) вычисляются следующим образом: текущее значение времени умножается на величину $\pi/30$.

```
_eSec:=Sec*pi/30; //секунды
_eMin:=Min*pi/30; //минуты
```

С переменной угла поворота часовой стрелки немножко сложнее. При определении ее местоположения учитываются не только часы (значение `Hour`), но и минуты (значение `Min`). Переменная `eHour` определяет грубое местоположение стрелки. Например, если системное время 12 часов ровно, то стрелка укажет на 12-часовую отметку циферблата. Переменная `Min` уточняет степень отклонения стрелки от часовой риски.

```
_eHour:=Hour*pi/6+Min/60*pi/6; //отклонение часовой стрелки
```

Если системное время 12 часов 30 минут, то за счет переменной Min к углу поворота стрелки добавится значение $\pi/12$. Таким образом, часовая стрелка получит дополнительное смещение от вертикального положения, указывающее, что скоро наступит час дня.

С этого момента в руках приложения оказываются все требуемые величины. На рис. 12.4 представлен подробный чертеж будущего циферблата нашего проекта.

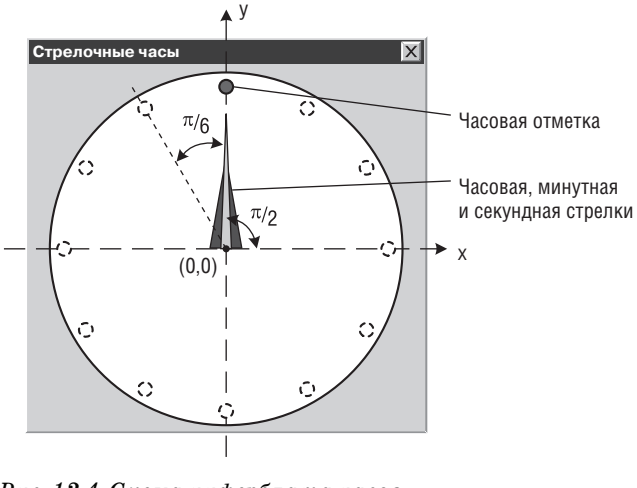


Рис. 12.4. Схема циферблата часов

Дело остается за малым – нарисовать часы средствами GDI. Для этого познакомьтесь с исходным кодом процедуры `PaintClock()`:

```
Procedure PaintClock(DC:HDC;ClientRect:TRect);
var XForm:TXForm;
    Ar:Array[0..2] of TPoint;
    eHourLabel:Real;
    CS,SN:DOUBLE;
begin
SetMapMode(DC, MM_ISOTROPIC); //переход в изотропный режим отображения
{в области вывода переносим начала координат в центр окна}
SetViewportOrgEx(DC, ClientRect.Right DIV 2, ClientRect.Bottom DIV 2, nil);
SetWindowExtEx(DC, 99, 99, nil);  //протяженность окна 100×100
SetViewportExtEx(DC, ClientRect.Right, ClientRect.Bottom, nil);
Ellipse(DC,-45,-45,45,45);    //эллипс - циферблат

SetGraphicsMode(DC,GM_ADVANCED);//переход в мировую систему координат

eHourLabel:=0; //рисуем 12 часовых меток
   while eHourLabel<2*pi do
       begin
       CS:=COS(eHourLabel); SN:=SIN(eHourLabel);
       XForm.eM11:=CS; XForm.eM12:=SN; XForm.eM21:=-SN; XForm.eM22:=CS;
       SetWorldTransform(DC,XForm);
       Ellipse(DC,40,2,44,-2);
       eHourLabel:=eHourLabel+pi/6;
       end;
//Часовая стрелка
  CS:=COS(eHour); SN:=SIN(eHour);
  XForm.eM11:=CS; XForm.eM12:=SN; XForm.eM21:=-SN; XForm.eM22:=CS;
  SetWorldTransform(DC,XForm);
  Ar[0].x:=-4; Ar[0].y:=0; Ar[1].x:=4;Ar[1].y:=0; Ar[2].x:=0;Ar[2].y:=-30;
  Polygon(DC,ar,3);
//Минутная стрелка
  CS:=COS(eMin); SN:=SIN(eMin);
  XForm.eM11:=CS; XForm.eM12:=SN; XForm.eM21:=-SN; XForm.eM22:=CS;
  SetWorldTransform(DC,XForm);
  Ar[0].x:=-3; Ar[0].y:=0; Ar[1].x:=3;Ar[1].y:=0; Ar[2].x:=0;Ar[2].y:=-40;
  Polygon(DC,ar,3);
//Секундная стрелка
  CS:=COS(eSec); SN:=SIN(eSec);
  XForm.eM11:=CS; XForm.eM12:=SN; XForm.eM21:=-SN; XForm.eM22:=CS;

  SetWorldTransform(DC,XForm);
  MoveToEx(DC,0, 5,nil); LineTo(DC,0,-42);
end;
```

В первых строках процедуры переводим страничную систему координат в изотропный режим отображения и устанавливаем начала координат осей в центр окна. В результате ось Y направлена из центра окна на 12-часовую отметку, а ось X – из центра окна на 3-часовую отметку циферблата. Для вывода на экран минутной и часовой стрелок вызывается функция `Polygon()`, в которую пе-

редается массив Ar, состоящий из трех точек. С секундной стрелкой еще проще – это обычная линия, проведенная парой функций MoveToEx() и LineTo(). По умолчанию все стрелки «смотрят» на 12-часовую отметку часов. Для того чтобы они повернулись на требуемый угол, воспользуемся аффинными преобразованиями, благо требуемые углы стрелок уже посчитаны.

Таким образом, наши знания нашли отражение в полезном программном продукте – стрелочных часах (рис. 12.5), благодаря которому (надеюсь) вы не опоздаете на деловую встречу, свидание или, что еще важнее... на обед. К несомненным достоинствам нашего творения стоит отнести компактность программы – она «весит» менее 45 килобайт. О чем обязательно не забудьте похвастаться перед своими друзьями, программирующими на C #...

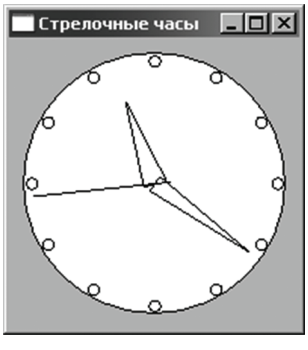


Рис. 12.5. Часы в работе

РЕЗЮМЕ

Обслуживая вывод изображений на поверхность графического устройства, современные версии операционной системы Windows способны работать в четырех системах координат. Координатные системы высокого уровня (мировая и страничная) отданы во власть программиста. Управляя ими с помощью функций Windows GDI, можно изменять начала осей координат окна и области просмотра, настраивать логические единицы измерения координатных осей, масштабировать, поворачивать изображение, осуществлять сложные двумерные аффинные преобразования. Для того чтобы нарисованный в мировых координатах графический объект был выведен на экран или принтер, операционная система производит последовательное преобразование координат рисунка из систем высокого уровня в координаты устройства.

13

Представление цвета в RGB-модели

Работа с цветом в Windows – едва ли не самое сложное, но вместе с тем интересное занятие. Не станем утруждать себя воспоминаниями об аксиомах из эргономики, в которых рассказывается об удобстве восприятия полноцветного изображения человеком. Это понятно и без доказательств. Особенно если вспомнить эпоху появления на свет первых ЭВМ. Мониторы этих машин не могли похвастаться цветным изображением. Они были монохромными – работали в черно-белом режиме. Для вывода картинок... (пожалуй, возьму слово «картинок» в кавычки). Итак, заново. Для вывода «картинок» применялись символы псевдографики. Разрешение и частота кадровой развертки экранов были постоянным поводом для критики. Даже сидеть перед подобным устройством отображения было просто наказанием, что уж говорить о работе...

Постепенно развитие технологии привело к повышению производительности процессоров и, что очень важно, – удешевлению микросхем памяти. Все это позволило конструкторам создать первые цветные видеоустройства. Отсчет начался с 16 цветов древнего видеоадаптера CGA[1]. Появление видеокарт, способных отображать 256 цветов, в свое время было чуть ли не откровением. Сегодня мы работаем в режимах High Color и True Color, поддерживающих соответственно 2^{16} и 2^{32} цветов.

Однако все начиналось даже не с адаптера CGA. Применяемая в Windows цветовая модель была разработана задолго до того, как в 1939 году Джон Атанасов и Клиффорд Бэрри из Университета штата Айова изобрели электронный цифровой компьютер. Еще на рубеже XVIII и XIX веков Томас Юнг ставил эксперименты с тремя лучами света, пропускаемыми через красный, зеленый и синий светофильтры. В результате смешивания окрашенных лучей получались различные цветовые оттенки. В середине XIX века Джеймс Максвелл развил идею Юнга и, регулируя яркость лучей, вывел коэффициенты их смешивания и предложил первый колориметр. Многие современные

[1] Первый цветной графический адаптер (Color Graphics Adapter, CGA), был реализован в 1982 году.

приборы построены на научном фундаменте, созданном этими учеными. Характерный пример – современные электронно-лучевые трубки (ЭЛТ). В ЭЛТ имеется три электронные пушки, бомбардирующие внутреннюю поверхность кинескопа потоком электронов. Каждая из пушек отвечает только за свою цветовую составляющую: красную, зеленую или синюю. Любой из цветов радуги, который вы увидите на экране монитора, получается в результате сложения (в той или иной пропорции) этих трех цветовых составляющих. Если необходимо получить больше красных оттенков, то мощность электронного потока красной пушки усиливается. Требуется чистый желтый цвет – синяя пушка выключается, а красная и зеленая «стреляют», не покладая рук. Для вывода белого цвета все три пушки начинают работать на своей максимальной мощности. Такую цветовую модель называют RGB, по первым буквам английских названий цветов (*Red*, *Green* и *Blue*).

Человеческое зрение также ориентировано на модель RGB. Наш глаз воспринимает цвета в диапазоне длин волн 700-400 нанометров благодаря специальным клеткам – колбочкам. Их три типа: колбочки, предпочитающие красный цвет, зеленый или синий. Кстати, так как колбочек, специализирующихся на зеленом цвете, больше, мы наиболее чувствительны к оттенкам зеленого.

Хранение данных о цвете в памяти

Рассказ о способах хранения данных о цвете построим по принципу от простого к сложному. Проще всего описать монохромную картинку, здесь не до изысканности. Одного бита памяти вполне достаточно для определения белого или черного цвета. Таким образом, один байт способен хранить данные о восьми монохромных пикселах. Очень экономно, но очень некрасиво...

Если вы не скряга и для описания отдельного пиксела способны пожертвовать целый байт, то мы сможем говорить о 256 цветах. Хотя это и значительно лучше монохромной картинки, но для отображения окружающего мира байта все равно окажется недостаточно. В те незапамятные времена (когда видеоадаптеры только научились работать с 8-битным цветом) предпринимались попытки охватить необъятное – построить полноценное цветное изображение на базе 8 бит. Первоначально для этого из всего доступного человеческому глазу цветового спектра выбирались 256 находящихся друг от друга на равном расстоянии фиксированных цветов. Но очень быстро выяснилось, что это не совсем удачная идея и подобная палитра не способна удовлетворить потребности всех программ – отсутствие полутонов резало глаз, вследствие этого изображения получались посредственными. Забраковав первый вариант освоения «просторов» 8-битного пространства, программисты пошли по другому пути и вместо фиксированного набора цветов предложили ввести динамическую цветовую таблицу, содержащую 256 элементов. В зависимости от обрабатываемого изображения, таблицу можно наполнять требуемыми полновесными 24-битными цветами. Например, для рисунка пустыни Сахары понадобится больше желтых тонов, а для описания изображения пасмурного неба – темно-синих оттенков. Из-за схожести с палитрой художника эту таблицу также назвали *палитрой* (*palette*).

От идеи построения унифицированной палитры с фиксированным набором цветов полностью не отказались и сегодня. При работе с 8-битным цветом предусмотрена возможность работы с 256 цветами, равномерно распределенными по частотному спектру. Это достигается благодаря функции `CreateHalftonePalette()`.

Даты рождения 8-битного и 16-битного формата представления цвета разделяет весьма незначительный промежуток времени. С момента появления 16-битного представления цвета (High Color) отпала необходимость в применении палитры. Два байта – это существенное пространство, позволяющее хранить значения всех трех цветовых составляющих. Единственный недостаток числа 16 в том, что оно не делится на три без остатка. Поэтому существует два варианта распределения имеющихся в двух байтах элементарных разрядов между цветами. Первый вариант High Color заключается в том, что старший бит остается неопределенным, а остальные 15 бит в равных долях распределяются между красной, зеленой и синей составляющими (рис. 13.1). Второй вариант High Color более эффективно распоряжается 16-битным пространством. Учитывая, что человеческий глаз более чувствителен к зеленому цвету, программисты решили отдать лишний бит в распоряжение зеленой составляющей (рис. 13.2).

Современное программное обеспечение ориентировано на работу с еще более разнообразными цветамие. Для удовлетворения изысканного вкуса таким приложениям требуется 24- или 32-разрядное представление цвета, называемое True Color. И в том, и другом случае на описание каждой цветовой составляющей отводится по 8 бит. Если цвет 32-разрядный, то оставшийся четвертый байт хранит значение прозрачности. Непрозрачному пикселу соответствует значение 255, если же в четвертом байте хранится нулевое значение, то это признак того, что пиксел абсолютно прозрачен и на экран выводиться не будет. Составляющая прозрачности называется альфа-каналом (см. главу 21 «Прозрачность и градиентная заливка»).

На описании цвета в Windows специализируется структура `COLORREFF`, в Delphi ее аналогом является тип данных `TColor`.

```
type TColor = -$7FFFFFFF-1..$7FFFFFFF;
```

Это не что иное, как тип данных `Cardinal`, позволяющий описать 32-битное неотрицательное значение.

Рис. 13.1. High Color – 15-разрядный формат представления цвета

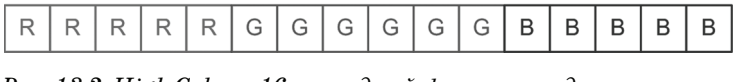


Рис. 13.2. High Color – 16-разрядный формат представления цвета

Определение характеристик цвета контекста устройства

Для того чтобы выяснить текущую глубину цвета, необходимо обратиться к функции GetDeviceCaps() с ключом BITSPIXEL. Если глубина цвета не превышает 8 бит, то (кроме покупки нового видеоадаптера) я могу посоветовать вновь воспользоваться услугами GetDeviceCaps(), но на этот раз с ключом NUMCOLORS. В этом случае функция возвратит число цветов в текущей цветовой палитре. Количество бит, затрачиваемых на представление одного пиксела, поможет узнать ключ COLORRES. Впрочем, об этих характеристиках мы уже подробно говорили в главе 2 «Контекст устройства» (см. табл. 2.1).

```
Procedure GDI_PAINT(DC:HDC);
var NUMCOLORS:integer;
    s:string;
begin
CASE GetDeviceCaps(DC,BITSPIXEL) OF
    24..32 : s:='True Color';
    15..16 : s:='High Color';
    ELSE
      begin
       s:='Color';
       NUMCOLORS:=GetDeviceCaps(DC,NUMCOLORS);
       if NUMCOLORS<>-1 then
          S:=S+' '+IntToStr(NUMCOLORS)+' цветов в палитре';
      end;
   END;
TextOut(DC,0,0,pChar(s),Length(s));
S:=IntToStr(GetDeviceCaps(DC,COLORRES))+' бит/пиксел';
TextOut(DC,0,14,pChar(s),Length(s));
end;
```

Макросы

При назначении цвета можно воспользоваться системными константами, определяющими цвет в соответствии со стилем оформления Windows, назначить цвет вручную или обратиться к услугам макросов GDI. У каждого из перечисленных способов есть свои достоинства и недостатки. Системные константы хороши лишь при определении наиболее распространенных цветов, так как не охватывают и одной десятитысячной части спектра цветов. Если вы на лету способны конвертировать десятичные значения в шестнадцатеричные и наоборот, то способа лучше, чем определение цвета вручную, не найдешь. Например, для создания кисти чистого зеленого цвета достаточно воспользоваться значением $00FF00:

```
Brush:=CreateSolidBrush($00FF00);
```

А что вы скажете об оттенках hD10FBB и h0A90EC? Я – без калькулятора – ничего... Кроме того, при «ручном» подходе вы столкнетесь со сложностями при смешивании значений трех основных цветов, особенно в динамике, когда

значения вычисляются в коде программы. Поэтому нашими лучшими помощниками при определении цвета станут макросы работы с цветом (табл. 13.1).

Таблица 13.1. Макросы для работы с цветом

Макрос	Назначение
`Function RGB(Red,Green,Blue : Byte): TColor;`	Макрос формирует результирующий цвет на основе сложения трех составляющих.
`Function GetRValue(Color : TColor) : Byte;`	Группа макросов, решающих обратную задачу. Они выделяют из смешанного цвета отдельную цветовую составляющую.
`Function GetGValue(Color : TColor) : Byte;`	
`Function GetBValue(Color : TColor) : Byte;`	

У макросов есть еще одна положительная черта – при формировании цвета они учитывают текущие настройки устройства вывода.

Работа с отдельным пикселом

С некоторой степенью допущения можно сказать, что рисунок любой степени сложности представляет собой мозаику из раскрашенных пикселов. Если у читателя нет принципиальных возражений против такого утверждения, то всю свою остальную работу по освоению графических просторов Windows можете построить на основе одной-единственной функции:

```
Function SetPixel(DC: HDC;  X, Y : integer; Color: TColor): TColor;
```

Здесь `DC` – дескриптор контекста, на поверхности которого мы станем рисовать; `X` и `Y` – координаты подопытного пиксела; `Color` – новый цвет. После выполнения функция возвратит значение нового цвета. Функция `SetPixel()` не в состоянии похвастаться особой прытью, но, к счастью, у нее имеется более расторопная родственница:

```
Function SetPixelV(DC: HDC;  X, Y : integer; Color: TColor): Boolean;
```

Скорости добавилось за счет того, что функция возвращает не новый цвет, а обычное логическое значение: `true` – работа завершилась успешно; `false` – сбой при закраске пиксела.

Все профессиональные графические редакторы для выбора цвета предоставляют в распоряжение пользователя удобные и наглядные диалоговые окна. Наличие трех цветовых составляющих предполагает применение объемных моделей, поэтому зачастую создается трехмерный куб, идея построения которого представлена на рис. 13.3. Для удобства на рисунке обозначены значения цветов для вершин куба. Куб повернут к нам своим самым светлым углом, в нем достигают максимума значения всех цветовых составляющих (`$FFFFFF`). Черный угол от нас скрыт, он находится в точке начала координат.

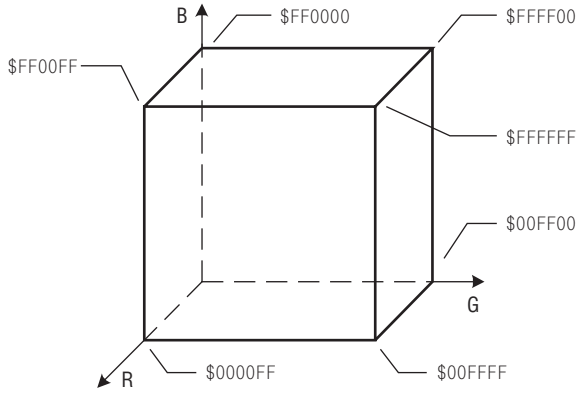


Рис. 13.3. Плоскости цветов куба

Попробуем повторить подвиг разработчиков графических редакторов. Для этого нам потребуются услуги функции `SetPixelIV()` **и макроса** `RGB()`**. Кроме того, мы закрепим знания двумерных аффинных преобразований.**

```
procedure GDI_PAINT(DC:HDC);
var R,G,B,X,Y : Byte;
    XForm:TXForm;
begin
{настраиваем режим отображения, центр и направление осей координат}
 SetMapMode(DC, MM_ISOTROPIC);
 SetWindowExtEx(DC, PAINTSTRUCT.rcPaint.Right, PAINTSTRUCT.rcPaint.Bottom, nil);
 SetViewportExtEx(DC, PAINTSTRUCT.rcPaint.Right, -PAINTSTRUCT.rcPaint.Bottom, nil);
 SetViewportOrgEx(DC, 50,400, nil);  //перенос центра координат
{=== фронтальная грань куба ===}
for X:=0 to 255 do
  for Y:=0 to 255 do
   begin
   G:=X; B:=Y;//значения зеленой (G) и синей (B) составляющих
   SetPixelIV(DC,X,Y,RGB($FF,G,B));//установка цвета пиксела (X,Y)
   end;
{=== правая грань куба ===}
XForm.eM11:=1/3;     XForm.eM12:=1/3; //правила аффинных преобразований
XForm.eM21:=0;       XForm.eM22:=1;   //деформация и сдвиг вправо фигуры
XForm.eDx:=255;      XForm.eDy:=0;
SetGraphicsMode(DC,GM_ADVANCED);//перевод контекста в мировую систему координат
SetWorldTransform(DC,XForm);     //применяем аффинные преобразования

for X:=0 to 255 do
 for Y:=0 to 255 do
  begin
   R:=255-X; B:=Y;//значения цвета красной (R) и синей (B) составляющих
   SetPixelIV(DC,X+1,Y+1,RGB(R,$FF,B));//установка цвета пиксела (X,Y)
  end;
{=== верхняя грань куба ===}
```

```
XForm.eM11:=1;     XForm.eM12:=0;
XForm.eM21:=1/3;   XForm.eM22:=1/3;
XForm.eDx:=0;      XForm.eDy:=256;

SetWorldTransform(dc,XForm);
for X:=0 to 255 do
 for Y:=0 to 255 do
    begin
    R:=255-Y; G:=X;//значения цвета красной (R) и зеленой (G) составляющих
    SetPixelIV(DC,X,Y,RGB(R,G,$FF));//установка цвета пиксела (X,Y)
  end;
end;
```

В первых строках кода мы переводим контекст в изотропный режим отображения, устанавливаем центр координат в точку (50,400) и изменяем направление оси Y – теперь она направлена снизу вверх. Завершив подготовительные мероприятия, переходим к самому главному – строим трехмерную модель. Первоначально на экран выводится передняя грань куба. Значение красной составляющей максимальное ($FF=255), остальные составляющие изменяются в рамках первого цикла программы. Затем настает черед правой грани (максимум зеленой составляющей). Эта грань рисуется не в совместимом режиме, а в мировых координатах (GM_ADVANCED). Собственно, идея вывода изображения этой плоскости точно такая, как и для фасада куба, за исключением того, что нами задаются правила аффинных преобразований – деформация и перенос деформированного квадрата вправо, за все остальное отвечает GDI. Практически аналогичные действия производятся и с верхней частью куба (максимум синей составляющей). К сожалению, при печати невозможно передать всю радугу оттенков (см. рис. 13.4 на цв. вклейке, с. 417), но это не беда, если вы читаете эти строки за компьютером. В прилагаемой к книге библиотеке на сайте издательства вы найдете исходный код и действующее приложение с таким кубом.

У функции SetPixel(), позволяющей окрасить пиксел в любой цвет, есть подружка – функция GetPixel(). У нее задача обратная – узнать текущее значение цвета пиксела.

```
Function GetPixel(DC: HDC;  X, Y : integer): TColor;
```

Пиксел адресуется координатами (X,Y). Если функция по каким-то причинам не смогла выяснить значение цвета или у нее просто плохое настроение, то она возвратит константу CLR_INVALID. Во всех остальных случаях GetPixel() порадует нас значением TColor.

Для того чтобы услуги функции GetPixel() стали максимально полезными, нам придется немного «подрессировать» мышь. Мы попробуем научить ее рассказывать приложению о местонахождении своего указателя. Для этого необходимо обработать сообщение WM_MOUSEMOVE, поступающее в адрес окна во время перемещения указателя мыши над клиентской частью окна. Координаты указателя хранятся вместе в младшем параметре lParam оконной процедуры: координата X в младшем слове, Y – в старшем. Ниже предложен листинг, демонстрирующий способ исследования пикселов окна:

```
VARx, y : Integer;
S: string; clr : TColor;
//…
function WindowProc (Window: HWND; Msg: UINT; wParam: WPARAM;
                     lParam: LPARAM) : LongInt; stdcall;
Begin
  case Msg of
  WM_MOUSEMOVE: //сообщение движения указателя мыши
     begin
     DC:=GetDC(Window);

X:=LOWORD(lParam); //извлечение координат указателя мыши
Y:=HIWORD(lParam);

     clr:=GetPixel(DC,x,y);  //узнаем цвет пиксела (x,y)
     Rct.Left:=0; Rct.Top:=0; Rct.Bottom:=18; Rct.Right:=200;
     FillRect(dc,Rct,GetStockObject(WHITE_BRUSH)); //закраска текстового блока
     s:=Format('R=%d G=%d B=%d',[GetRValue(Clr),GetGValue(Clr),GetBValue(Clr)]);
     TextOut(DC,5,1,pChar(s),Length(s)); //вывод значений цветовых составляющих
     ReleaseDC(DC,Window);
     end;
// WM_PAINT и остальные сообщения
End;
```

Единственный комментарий к оконной процедуре `WindowProc()` касается особенности передачи в нее координат курсора мыши. В момент движения курсора в адрес оконной процедуры направляются сообщения `WM_MOUSEMOVE`, а координаты курсора одновременно передаются в параметр `LParam`. Координата `X` окажется в младшем слове параметра, а координата `Y` — в старшем.

Системные цвета

У каждого из нас есть свои предпочтения. Кому-то нравится черный кофе, кто-то пьет только чай. Мы носим одежду своего стиля, посещаем любимые кафе, слушаем музыку, которая нам нравится. Вполне естественно, что у вас и у меня есть предпочтительные цвета. Взгляните на рабочие столы Windows своих друзей или коллег по работе. Наверняка каждый из них подобрал подходящую его вкусу цветовую схему. Благодаря цветовой схеме в Windows для всех приложений централизованно назначаются цвета поверхностей окон, элементов управления, их рамок и отбрасываемых теней, полос прокрутки, текста, всплывающих сообщений и т. д. Весьма удобно. Каждое приложение может выяснить, какие цвета пользователю больше всего по душе, и накинуть соответствующую «одежку».

Неудивительно, что в Windows API имеются функции, позволяющие программисту обращаться к системным цветам операционной системы. В первую очередь это функция, возвращающая значение цвета в соответствии с названием системной константы.

Win API

```
Function GetSysColor(ColorIndex :Integer) : TColor;
```

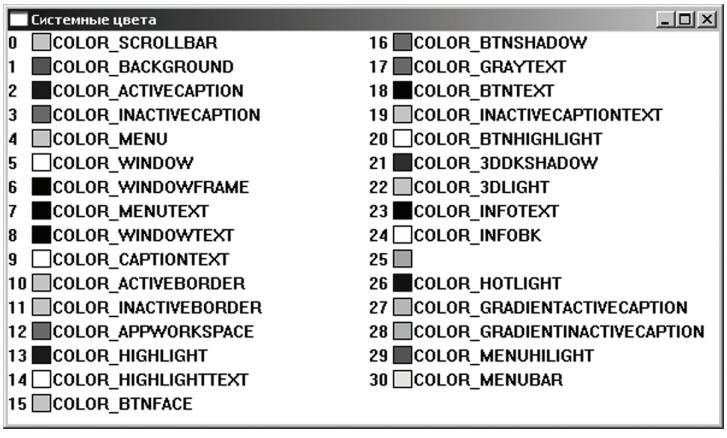


Рис. 13.5. Названия констант системных цветов Windows

Название константы передается в единственный параметр функции – ColorIndex. Таких констант три десятка. Например: COLOR_INACTIVECAPTION возвратит цвет неактивного заголовка окна, COLOR_BTNSHADOW – цвет тени, отбрасываемой кнопкой, COLOR_MENU – цвет пункта меню. Все названия перечислять не стоит, вы их найдете в справочной системе (имена констант начинаются со слова «COLOR_») и на рис. 13.5.

Для изменения системного цвета нам потребуется функция:

Win API
```
function SetSysColors(ElCount: Integer; const Elements; const RgbValues): BOOL;
```

Рассмотрение функции начнем со второго параметра. Это указатель на массив с именами изменяемых констант. В первом элементе мы уведомляем функцию, сколько всего элементов мы собираемся изменить. Последний параметр – указатель на массив, содержащий значения цветов. Пример работы с обеими функциями вы найдете в следующем фрагменте кода:

```
VAR //…
  OldColor : Colorref; Button1 : HWND;
  Elements : array[0..3] of integer; RgbValues : array[0..3] of Colorref;

function WindowProc (Window: HWND; Msg: UINT;
                     wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
BEGIN
 CASE Msg OF
  WM_CREATE :
  begin
   Button1:= CreateWindow('BUTTON', Синий цвет',
              WS_CHILD OR WS_BORDER OR WS_VISIBLE OR BS_PUSHBUTTON OR BS_LEFT,
              10, 10, 174, 30, Window, 0, HInstance, nil);

   OldColor:=GetSysColor(COLOR_DESKTOP); //запомним исходный цвет рабочего стола
   Elements[0]:=COLOR_DESKTOP; RgbValues[0]:=$FF0000;
  end;
```

```
  WM_COMMAND : if LParam=Button1 then SetSysColors(1,Elements,RgbValues);
  WM_DESTROY :
  begin {перед выходом восстановим прежний цвет рабочего стола}
   RgbValues[0]:=OldColor;
   SetSysColors(1,Elements,RgbValues);
   PostQuitMessage (0);
   Exit;
  end;
 // обработка других сообщений
 END;
END;
```

Нажатие кнопки (сообщение `WM_COMMAND`) изменяет цвет рабочего стола. В момент завершения работы приложения (сообщение `WM_DESTROY`) мы восстанавливаем его прежний цвет.

Альтернативные цветовые модели

Модель воспроизведения цвета на основе красного, зеленого и синих цветов весьма распространена, но далеко не единственна. Кроме RGB существует еще несколько часто используемых цветовых моделей.

В первую очередь отметим модели CMY и CMYK. Аббревиатура CMY появилась из первых букв составляющих ее цветов: *бирюзовый* (*Cyan*), *пурпурный* (*Magenta*) и *желтый* (*Yellow*). CMY – прямой антипод модели RGB. Смотрите сами, в шестнадцатеричном представлении красный цвет описывается значением `h0000FF`, а бирюзовый – `hFFFF00`. Зеленый – `h00FF00`, а пурпурный – `hFF00FF`. Наконец синий – `hFF0000`, а его противник желтый – `h00FFFFF`. Модель CMY стоит на вооружении большого числа цветных печатающих устройств. Модель CMYK представляет собой усовершенствованный вариант CMY. Доработка заключается в добавлении к трем основным цветам четвертого – *черного цвета* (*blacK*). Тем самым в типографиях повысили качество печати черного цвета.

Человеческому представлению о цвете лучше всего подходят цветовые пространства HLS и HSB. Модель HLS характеризуется *оттенком* (*Hue*), *интенсивностью* (*Lightness*) и *насыщенностью* (*Saturation*). А в модели HSB вместо интенсивности применяется *яркость* (*Brightness*).

Еще одна модель – LAB. Здесь цвет описывается интенсивностью (L) и отношениями зеленого/красного (A) и синего/желтого цветов.

РЕЗЮМЕ

Основной цветовой моделью Windows является RGB. Ее особенность в том, что результирующий цвет формируется из трех основных цветовых составляющих: красной, зеленой и синей. Наилучшее качество изображения достигается в режиме True Color, использующем 32-битные значения. Наиболее слабыми возможностями обладает 8-битный цвет, в этом случае значения цветов кодируются в специальной таблице, называющейся палитрой.

14

Цветовые палитры

Так и хочется начать эту главу с фразы: «Цветовые палитры были придуманы от безысходности». И это нисколько не преувеличение. В старые добрые времена, когда мегабайт оперативной памяти стоил безумных денег, пользователи были вынуждены обходиться минимальным набором цветов. Поэтому даже 8-битный режим работы видеоадаптера считался чуть ли не царской роскошью. Вместе с тем, уже тогда устройства видеовывода были готовы работать в полноцветном режиме, но видеокарты не могли использовать возможности мониторов в полном объеме. Поэтому разработчики программного обеспечения постоянно сталкивались с проблемой качественного представления цветных изображений в неуютном 8-битном графическом режиме. В прошлой главе уже упоминалось о том, что первоначально рассматривался следующий вариант работы. Во всем доступном человеческому глазу цветовом спектре через равные промежутки отбирались 256 цветов (ведь именно столько позволяет закодировать один байт). Однако очень скоро стало понятно, что сама по себе идея (в столь стесненных обстоятельствах создать универсальный набор цветов, способный удовлетворить потребности всех приложений) хороша ровно настолько, насколько и утопична. Объекты реального мира окрашены совсем по-другому. Вспомните закат на море, утопающее в темно-зеленой воде солнце и теплые всполохи на гребне волн от последних его лучей... Согласитесь, что при выводе подобного пейзажа на экран большая часть красок из выхваченных из спектра 256 цветов нам просто не сможет ничем помочь. В этом случае было бы лучше, чтобы в нашем распоряжении оказалось больше спокойных оттенков красного, темно-зеленого и синего вместо бесполезных в данном случае экспрессивных цветов.

Поэтому на смену идее равномерного распределения цветов пришли таблицы цветов, позволяющие в динамическом режиме подстраиваться под пожелания приложения и изменять свое содержимое. Из-за сходства с палитрой художника эти таблицы получили название цветовой палитры. Каждая ячейка палитры хранит полноценное 24-битное значение цвета, для адресации этого цвета применяется 8-битный индекс ячейки. В 256-разрядном

видеорежиме приложение манипулирует не реальными значениями цветов, а их индексами, что позволяет повысить качество результирующего изображения. При необходимости цветовая палитра может быть реконфигурирована – заполнена требуемыми цветовыми оттенками.

Перевод дисплея в 8-битный режим

Для того чтобы при выводе всего пользовательского интерфейса Windows на экран GDI начал использовать палитру, необходимо установить 8-битную глубину цвета. Проще всего это сделать, воспользовавшись утилитой настройки экрана, однако предложу на ваше рассмотрение более рациональный путь. Он основан на возможностях изученной в главе 3 этой книги функции `ChangeDisplaySettings()`. Напомню, что функция предназначена для изменения графического режима дисплея. Желаемые настройки (глубина цвета, разрешение и частота кадровой развертки) определяются в записи `TDeviceModeA`. Одной из положительных черт `ChangeDisplaySettings()` является возможность временного перехода в новый режим отображения, для этого предназначен флаг `CDS_FULLSCREEN`. После завершения работы приложения экран возвращается в прежний графический режим.

```
var DM:TDeviceModeA;
//…
WM_SHOWWINDOW : {проверка факта поддержки палитры}
begin
     IF ODD(GetDeviceCaps(DC,RASTERCAPS) SHR 8)=False then
     begin
       //RC_PALETTE=$100 - это 9-й бит значения возвращаемого GetDeviceCaps
       DM.dmSize:=SizeOf(TDeviceModeA);
       DM.dmDriverExtra:=0;
       EnumDisplaySettings(nil,0,DM);
       DM.dmBitsPerPel:=8; DM.dmDisplayFrequency:=100;
       DM.dmPelsWidth:=1024; DM.dmPelsHeight:=768;
      if ChangeDisplaySettings(DM,CDS_FULLSCREEN)<>DISP_CHANGE_SUCCESSFUL
       then begin
       MessageBox(Window,PChar('Не удалось перейти в заданный
                          графический режим!'),PChar('Ошибка'),MB_ICONERROR);
       PostQuitMessage (0);
       end;
     end;
end;
```

Большая часть примеров этой главы рассчитана на 256-цветовой режим. Если вас не устраивает программный способ перевода дисплея к 8-битному цвету – обратитесь к свойствам экрана и осуществите переключение вручную.

Работа представленного выше кода начинается с выяснения обстановки, нас интересует – использует ли в данный момент времени палитру контекст. Для этого в момент вывода окна на экран мы обращаемся к функции `GetDeviceCaps()` с просьбой вернуть состояние флага `RASTERCAPS`. Получив флаг,

анализируем его 9-й бит (имя бита RC_PALETTE=h100). Если бит установлен, то ответ положительный – опрашиваемый контекст использует палитру. Если ответ отрицательный, то переводим экран в 256-цветный режим.

Структура палитры

Прежде чем мы приступим к изучению функций GDI, специализирующихся на обслуживании палитры, стоит разобраться со способом представления палитры в памяти. Пока о палитре нам известно лишь то, что это таблица из 256 элементов, каждый из которых хранит 24-битное значение цвета. Для начала совсем неплохо, но все-таки это как раз тот случай, когда подробности не окажутся лишними.

Отдельный элемент палитры Windows представляет собой запись TPaletteEntry, он включает информацию о каждой цветовой составляющей плюс поле флагов, используемое для формирования логической палитры:

```
type TPaletteEntry  = packed record
     peRed  : Byte; //значение красной составляющей цвета
     peGreen: Byte; //значение зеленой составляющей цвета
     peBlue : Byte; //значение синей составляющей цвета
     peFlags: Byte; //поле флагов
end;
```

Поле флагов (табл. 14.1) представляет интерес только в момент формирования палитры.

Таблица 14.1. Флаги структуры TPaletteEntry

Флаг	Значение	Описание
PC_RESERVED	h01	Зарезервировать строку, находящееся в ней значение станет использоваться для анимации.
PC_EXPLICIT	h02	Не изменять системную палитру. Цвета системной палитры используются в качестве индексов логической палитры. Индекс цвета в системной палитре заносится в первые два байта структуры TPaletteEntry.
PC_NOCOLLAPSE	h04	Производить поиск цвета в системной палитре только при отсутствии свободных мест, в противном случае сразу занимать свободную ячейку в логической палитре.

Если поле флагов не заполнено (установлено нулевое значение), то перед включением цвета в логическую палитру GDI пытается найти его в системной палитре. Если такой цвет имеется, то в логической палитре для него задействуется строка с тем же самым индексом. Иначе цвет сразу заносится в первую свободную ячейку логической палитры.

Все записи TPaletteEntry в свою очередь интегрируются в структуру:

```
type TLogPalette  = packed record
    palVersion    : Word;  //версия
```

```
    palNumEntries: Word;  //количество элементов палитры
    palPalEntry  : array[0..0] of TPaletteEntry; //массив элементов
end;
```

Познакомьтесь, это и есть палитра. Сразу выясняется, что присутствие в таблице палитры всех 256 цветовых значений – совсем не обязательное условие. Так как в структуре не установлен жесткий размер массива `palPalEntry`, то ничто не запрещает построить палитру, например, для 64 или 16 цветов. Количество цветов определяется стоящими перед нами задачами. В чернобелой палитре всего 2 цвета, в классической 8-битной – 256, в Windows NT\XP\2000 структура может содержать до 1024 значений.

Макросы для работы с палитрой

При работе с палитрой раз и навсегда следует отказаться от использования макроса RGB. Он не подходит для применения всего по одной прозаичной причине – возвращаемый им 24-битный цвет может отсутствовать в палитре. Безусловно, при работе в 8-битном режиме, получив подобное значение, GDI все равно постарается вывести заказанный цвет, но это будет эрзац, состоящий из комбинации пикселов, окрашенных во входящие в палитру цвета. На практике это выглядит совсем не привлекательно. Ко всему прочему, попытка добиться хоть какой-то схожести приведет к дополнительному расходованию системных ресурсов, что также не приветствуется. Для устранения этого недостатка в составе GDI предусмотрен макрос:

```
Function PALETTERGB(Red,Green,Blue : Boolean) : COLORREF;
```

Макрос возвратит цвет, не просто сложив три цветовые составляющие, а на основе подбора наиболее подходящего цвета из логической палитры устройства.

Кроме `PALETTERGB()` при работе с палитрой часто применяется макрос:

```
Function PALETTEINDEX(PaletteIndex  : Word) : COLORREF;
```

Он позволяет выяснить, какой цвет ассоциирован с индексом `PaletteIndex` логической палитры.

Системная палитра

Системная палитра (*system palette*) принадлежит не какому-то отдельному контексту, а является разделяемым ресурсом всей операционной системы. Системная палитра существует во всех графических режимах Windows, вне зависимости от установленной разрядности представления цвета. Хотя системная палитра принадлежит операционной системе, но к ней имеет право обратиться и даже попытаться внести в нее изменения любой контекст устройства.

Для исследования системной палитры (текущей палитры контекста устройства) предназначена функция:

```
Function GetSystemPaletteEntries(DC : HDC;
                                 StartIndex, Entries : Word;
                                 var PALETTEENTRY) : Word;
```

Здесь DC – дескриптор исследуемого контекста, StartIndex – указывает индекс элемента, с которого мы планируем начать исследовать палитру; Entries – сколько всего элементов палитры мы изучаем; PALETTEENTRY – массив типа TPaletteEntry.

Число элементов в системной палитре позволит узнать функция GetDeviceCaps(). *Для этого к ней следует обратиться с использованием константы* SIZEPALETTE. *Правда, и здесь не обошлось без ограничений, они подробно изложены в приложении ко второй главе настоящей книги. В подавляющем большинстве случаев надо исходить из предположения, что таблица системной палитры содержит 256 строк.*

Способ исследования системной палитры предложен в следующем фрагменте кода:

```
Procedure GDI_PAINT(DC:HDC);
var PE:array[0..255] of TPaletteEntry;
    X,Y:integer; clr:COLORREF;
    EntryNum : byte;
begin
s:=Format('Глубина цвета %d бит',[GetDeviceCaps(DC,BITSPIXEL)]);
TextOut(DC,10,1,PChar(s),Length(s));
GetSystemPaletteEntries(DC,0,255,PE);
x:=10; y:=20;
 for EntryNum:=0 to 255 do
 begin
   clr:=PALETTERGB(PE[EntryNum].peRed,PE[EntryNum].peGreen,PE[EntryNum].peBlue);
       SetDCBrushColor(DC,clr);
       SelectObject(DC,GetStockObject(DC_BRUSH));
       Rectangle(DC,X,Y,X+16,Y+16);
       INC(X,18);
       If (EntryNum+1) mod 16 = 0 then
         begin
         x:=10; INC(Y,18);
         end;
end; end;
```

Экспериментируя с различной глубиной цвета, мы получим примечательные результаты. Из рис. 14.1 (см. цветную вклейку, с. 417) явно видно, что при глубине цвета выше 8 бит исходная системная палитра содержит всего-навсего 20 цветов. Они расположены в начале и в конце таблицы, все остальные 236 элементов палитры черны как уголь. В системной палитре, сформированной при 8-битной глубине цветов, на аналогичных позициях присутствуют те же 20 цветов, но теперь все оставшиеся элементы палитры (или только их часть) могут быть заполнены другими цветами.

Из 20 создаваемых по умолчанию системных цветов 16 повторяют цвета давно устаревшего режима VGA и 4 определяются в соответствии с текущей цветовой схемой Windows.

Системные цвета VGA

Как мы выяснили, в любом цветовом режиме первая и последняя десятка строк системной палитры (адреса $00...$09 и $F6...$FF) отведены для хранения статических цветов и цветов текущей цветовой схемы Windows. Из них 16 цветов статических и 4 (ячейки палитры $08, $09, $F6 и $F7) определяются текущей цветовой схемой. Перечень 16-ти статических цветов VGA представлен в табл. 14.2.

Таблица 14.2. Таблица статических цветов VGA

№	Цвет	Значение			Индекс в палитре	Именованная константа Delphi
		R	G	B		
1	Черный	$00	$00	$00	$00	clBlack
2	Бордовый	$80	$00	$00	$01	clMaroon
3	Темно-зеленый	$00	$80	$00	$02	clGreen
4	Оливковый	$80	$80	$00	$03	clOlive
5	Темно-синий	$00	$00	$80	$04	clNavy
6	Пурпурный	$80	$00	$80	$05	clPurple
7	Голубой	$00	$80	$80	$06	clTeal
8	Светло-серый	$C0	$C0	$C0	$07	clSilver
9	Темно-серый	$80	$80	$80	$F8	clGray
10	Красный	$FF	$00	$00	$F9	clRed
11	Зеленый	$00	$FF	$00	$FA	clLime
12	Желтый	$00	$FF	$FF	$FB	clYellow
13	Синий	$00	$00	$FF	$FC	clBlue
14	Малиновый	$FF	$00	$FF	$FD	clFuchsia
15	Небесно-голубой	$00	$FF	$FF	$FE	clAqua
16	Белый	$FF	$FF	$FF	$FF	clWhite

В Delphi константы статических цветов объявлены в модуле Graphics.

При остром желании мы сможем управлять количеством статических цветов в системной палитре.


```
Function SetSystemPaletteUse(DC : HDC; Usage : UINT) : UINT;
```

Набор возможностей не столь велик. Режим определяется параметром `Usage`, который представлен в табл. 14.3. Если функция выполнилась успешно, то она возвратит константу предыдущего состояния. В противном случае – значение `SYSPAL_ERROR`.

Таблица 14.3. Режим использования системной палитры

Константа	Значение	Описание
SYSPAL_STATIC	h01	Системная палитра содержит статические цвета, они не могут быть удалены другими приложениями при реализации их логических палитр.
SYSPAL_NOSTATIC	h02	Системная палитра содержит два статических цвета – белый и черный.
SYSPAL_NOSTATIC256	h03	Режим доступен только в Windows 2000/XP. Системная палитра не содержит статических цветов.

Для удобства пользователя и программистов в Windows имеется цветовая схема, позволяющая централизованно настраивать рабочую раскраску приложений. Все цвета, предусмотренные цветовой схемой, поименованы. Названий констант более трех десятков, их вы найдете в Windows SDK (имя начинается со слова «COLOR», например, цвет клиентской области окна описывается константой COLOR_WINDOW). Конвертированием имени константы (точнее, ее числового значения) в значение цвета занимается знакомая по прошлой главе функция Windows API GetSysColor().Функция может найти применение во всех операциях, связанных с выбором системных цветов. Например, так может быть создано сплошное перо единичной толщины с цветом, соответствующим активному заголовку окна.

```
Pen:=CreatePen(PS_SOLID,1,GetSysColor(COLOR_ACTIVECAPTION));
```

Раз мы заговорили о цветовой схеме Windows, то обязательно вновь упомянем функцию, позволяющую изменять настройки схемы SetSysColors(). В приведенном ниже примере массив ElementsName хранит наименования цветовых констант (они соответствуют элементам графического интерфейса пользователя), а массив RgbValues – соответствующие им цвета. Первый параметр функции Elements определяет число элементов в массиве.

```
varElementsName: Array[0..1] of cardinal;
  RGBValues: Array[0..1] of colorref;
//…
ElementsName[0]:=COLOR_BTNTEXT; //константа цвета текста
ElementsName[1]:=COLOR_DESKTOP; //константа цвета рабочего стола
RGBValues[0]:=$00AA00;
RGBValues[1]:=$CCCCCC;
SetSysColors(2,ElementsName,RgbValues);
```

В момент изменения системных цветов всем окнам верхнего уровня приходит сообщение WM_SYSCOLORCHANGE.

Логическая палитра

Логическая палитра (*logical palette*) выступает в роли проводника идей системной палитры, но теперь уже на уровне конкретного контекста графического устройства. Это полноценный объект GDI, являющийся атрибутом

контекста устройства и существующий во всех графических режимах (даже в полноцветных). Для обращения к стандартной логической палитре применяют функцию `GetStockObject()` с константой `DEFAULT_PALETTE`.

Логическая палитра присутствует в любом контексте устройства независимо от текущей глубины цвета.

Необходимость существования логической палитры обусловлена невозможностью удовлетворения потребностей всех одновременно функционирующих в системе процессов за счет набора из 256 цветов. Одним приложениям необходимо работать с равномерно распределенными цветами, другим требуются оттенки серого, третьим – оттенки синего и желтого... Однако видеоадаптер физически не в состоянии порадовать каждого просителя – он настроен на 8-битный режим и способен выводить лишь цвета системной палитры. Для разрешения противоречия интересов было принято решение дать активному приложению право адаптировать системную палитру под свои нужды. В момент создания или в момент активизации окна процесс заполняет системную палитру своими цветами, оставляя нетронутыми лишь 20 системных ячеек. Все остальные неактивные приложения вынуждены довольствоваться тем, что есть, перекрашиваясь в имеющиеся в системной палитре цвета.

Для того чтобы контекст устройства смог хранить свои предпочтительные цвета, он снабжен логической палитрой. Она создается в момент рождения контекста на основе системной палитры. При активизации окна контекст окна имеет право заполнить системную палитру своими данными.

Палитра с равномерно распределенными цветами

Исходя из того, что в момент создания логическая палитра «впитывает» цвета системной палитры, можно предположить, что ее изначальная гамма будет весьма куцей – 20 цветов. Простейшим путем расширения набора цветов палитры до 256 значений станет применение функции:

Win GDI

```
Function CreateHalftonePalette(DC : HDC) : HPALETTE;
```

Это как раз тот случай, о котором мы с некоторой иронией рассуждали в начале главы. Видимый спектр равномерно разрезается на 256 минус 20 статических цветов частей и полученными значениями заполняется цветовая таблица. Получается палитра, включающая равномерно распределенные полутона цветов, часто называемая *полутоновой*.

Для того чтобы вновь созданной палитрой смог воспользоваться контекст устройства, необходим вызов еще одной функции:

Win GDI

```
Function SelectPalette(DC : HDC; Palette : HPALETTE;
                       ForceBackground : Boolean) : HPALETTE;
```

Кроме дескрипторов контекста и палитры программист должен указать, насколько кардинально мы собираемся изменить системную палитру. Передав в параметр `ForceBackground` значение `true`, мы создадим *фоновую палитру*. Фоновая палитра необходима для сохранения приемлемого качества изобра-

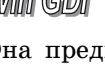
жений для неактивных окон. Точнее говоря, для попытки сохранения качества. Исходя из того, что это вспомогательная задача, в системной палитре заменяются лишь незадействованные ячейки, а все остальное остается как есть. Если незадействованных ячеек достаточно для нужд фоновых окон, то им повезло, а если недостаточно, то придется обходиться тем, что есть. Другой вариант (значение `false`) говорит о том, что мы пошли по пути создания *основной палитры*. Основная палитра содержит цвета активного окна. Это означает, что замене подлежит ровно столько цветов системной палитры (за исключением системных), сколько необходимо для качественного вывода изображения. Если после создания основной палитры в ней не останется свободных ячеек, то использующим альтернативную палитру неактивным окнам придется очень тяжело – для вывода своего изображения они будут вынуждены довольствоваться имеющимися цветами. По завершению своей работы функция возвращает дескриптор предыдущей палитры.

Выбрав палитру в контекст, переходим к последнему шагу – заполняем системную палитру цветами из созданной нами логической палитры.

```
Function RealizePalette(DC : HDC) : Word;
```

Это операция называется *реализацией логической палитры*. В случае успешной реализации функция возвратит число обновленных элементов системной палитры.

Антиподом функции `RealizePalette()`, предназначенной для отображения цветов логической палитры в системную палитру, является функция:



```
Function UnrealizeObject(GDIObj  : HGDIOBJ) : boolean;
```

Она предназначена для возвращения логической палитры в исходное состояние.

Следующий фрагмент кода демонстрирует порядок создания палитры с равномерно распределенными цветами, ее выбор и реализацию в момент активизации окна приложения.

```
var Palette, OldPalette : HPALETTE;
//…
CASE Msg OF
WM_SHOWWINDOW :
    begin
    DC:=GetDC(Window);
    Palette:=CreateHalftonePalette(DC);
    ReleaseDC(DC,Window);
    End;
WM_ACTIVATE :
    begin
    DC:=GetDC(Window);
    OldPalette:=SelectPalette(DC,Palette,False);
    RealizePalette(DC);
    //…
```

```
    end;
end;
```

Число цветов логической палитры и их характеристики нам подскажет функция:

```
Function GetPaletteEntries(Palette : HPALETTE;
                StartIndex, Entries : Word; var PALETTEENTRY): Word;
```

Здесь Palette – дескриптор изучаемой логической палитры; StartIndex указывает индекс элемента, с которого мы планируем начать исследовать палитру; Entries – сколько всего элементов палитры мы изучаем; PALETTEENTRY – массив типа TPaletteEntry. Одним словом, все очень похоже на представленную ранее функцию GetSystemPaletteEntries().

Импортируя функцию GetPaletteEntries() из библиотеки gdi32 в ранние версии Delphi, в Borland немножко перемудрили. Суть в том, что исходная (написанная на Си) функция позволяла узнать число элементов системной палитры. Для этого в ее последний параметр было достаточно передать неопределенный указатель nil. Наличие ключевого слова var в объявлении функции на языке Паскаль несколько усложняет задачу, заставляя нас добавить пару строк кода. Например, допустимо объявить переменную-указатель типа pointer, установить его в никуда (присвоить nil) и передать в параметр PALETTEENTRY. Но помните, что у указателей весьма буйный нрав, посему в ответ на любую некорректность в обращении с ними Delphi с маниакальной настойчивостью станет нас «радовать» ошибкой выполнения № 216. Поэтому для подсчета строк в системной таблице палитры лучше обращаться к GetDeviceCaps().

На рис. 14.2 (см. цветную вклейку, с. 418) представлен снимок окна с полутоновой палитрой.

Пользовательская палитра

В ситуации, когда набор цветов полутоновой палитры перестает удовлетворять потребности приложения, нам придется потрудиться над созданием своей собственной, пользовательской палитры цветов. В этом случае работа начинается с обращения к функции:

```
function CreatePalette(const LogPalette: TLogPalette): HPalette;
```

Единственный параметр функции – константа LogPalette, представляющая собой структуру TLogPalette, специализирующуюся на хранении палитры. Функция возвращает дескриптор пустой пользовательской палитры.

Для определения элементов пользовательской палитры применяют функцию:

```
function SetPaletteEntries(Palette: HPALETTE;
                           StartIndex, NumEntries: Word;
                           var PaletteEntries): Word;
```

Здесь Palette – дескриптор настраиваемой логической палитры; StartIndex – номер элемента палитры, с которого мы начинаем заполнение; NumEntries –

количество обновляемых элементов; PaletteEntries — массив элементов TPaletteEntry.

Изменение размера логической палитры

GDI позволяет управлять размером уже существующей логической палитры. Специалистом по раздвижению (или сжатию) пространств логической палитры считается функция:

```
function ResizePalette(Palette: HPALETTE; Entries: Word): Boolean;
```

Кроме дескриптора палитры функция желает знать, сколько ячеек должно остаться в палитре. Для этого предназначен параметр Entries. Напомню, что хотя классический размер таблицы равен 256 строкам, но размер палитры может колебаться от 2 (черный и белый цвета) до 1024 элементов (в современных версиях Windows).

Сообщения, связанные с изменением палитры

В начале раздела уже упоминалось, что системная палитра умеет подстраиваться под запросы различных приложений. Это утверждение легко проверить, но перед этим стоит познакомиться с системными сообщениями, связанными с модификацией палитры.

Наиболее популярное у программистов сообщение называется WM_PALETTECHANGED. Оно рассылается всем перекрытым окнам и окнам верхнего уровня после того, как активное в данный момент приложение изменит системную палитру. Если ваше приложение зависит от состояния палитры, то будьте добры — отслеживайте это сообщение. Простейший вариант реакции проекта на изменение палитры заключается в полном обновлении поверхности окна.

```
WM_PALETTECHANGED : InvalidateRect(Window, nil, true);
```

Существует и более эффективное решение. Кроме самого сообщения в параметр WPARAM оконной процедуры передается дескриптор окна, инициировавшего изменение палитры. Полное обновление может оказаться излишним, если палитра была изменена нашим окном. В этом случае корректнее просто вызвать функцию UpdateColors(), осуществляющую лишь ускоренное обновление цветов контекста в соответствии с логической палитрой.

```
function WindowProc (Window: HWND; Msg: UINT; wParam: WPARAM; lParam: LPARAM) :
LongInt; stdcall;
begin
 CASE Msg OF
   WM_PALETTECHANGED :
   begin
     if WParam<>Window then InvalidateRect(Window, nil, true)
     else UpdateColors(DC);
   end;
//другие сообщения
```

Кроме рассмотренного выше WM_PALETTECHANGED внимания заслуживает еще пара сообщений. Во-первых, это сообщение WM_PALETTEISCHANGING. С точки зрения очередности оно следует перед WM_PALETTECHANGED. Задача сообщения – информировать все заинтересованные процессы о том, что какое-то из приложений намерено изменить логическую палитру.

Второе сообщение называется WM_QUERYNEWPALETTE. Оно трудится в интересах неактивных окон. Допустим, что пока наше окно находилось в пассивном состоянии, сторонний процесс поменял системную палитру. Если после этого наше окно получит фокус ввода и предпримет попытку обновиться цветами чужой таблицы, мы получим искаженное изображение. Дабы предотвратить такое развитие событий, операционная система уведомляет получающее фокус ввода (пока еще пассивное) окно о том, что системная палитра была модифицирована. В ответ на это сообщение приложение должно вновь выбрать, реализовать свою палитру и перерисовать окно. Вместо полной перерисовки окна можно обновить цвета в окне, вызвав функцию:

```
Function UpdateColors(DC : HDC) : boolean;
```

Функция просмотрит все пикселы и отобразит их цветовые индексы из текущей системной палитры на наиболее схожие по цветовым характеристикам индексы новой системной палитры.

Поиск цвета в палитре

Палитра – это всего лишь эрзац полноценного цветового пространства и никогда в полной мере не сможет удовлетворить потребности современного приложения. Но, хотя область применения 256-цветного режима крайне узка, мы не можем исключать вероятность того, что наше приложение станет в нем работать. Что делать приложению, рассчитанному на 16- или 24-битную глубину цвета в таких стесненных условиях? Ответ прост – подстраиваться. В первую очередь надо научить программу подбирать в палитре наиболее подходящий цвет-заменитель. Для этого в GDI предусмотрены две функции:

```
function GetNearestColor(DC: HDC; Clr: COLORREF): COLORREF;
```

и

```
function GetNearestPaletteIndex(Palette: HPALETTE; Clr: COLORREF): UINT;
```

Они ищут ближайший подходящий цвет (параметр Clr) в палитре. Функция GetNearestColor() анализирует палитру, связанную с контекстом устройства DC, а функция GetNearestPaletteIndex() работает непосредственно с палитрой Palette. Первая из функций возвращает значения цвета в формате COLORREF, вторая – индекс цвета в палитре.

РЕЗЮМЕ

Современные версии Windows предназначены для работы в полноцветных режимах High Color и True Color, поэтому требующий использования палитры 8-ми битный графический режим – скорее исключение из правил. Казалось бы, умение работать с палитрой становится анахронизмом, но это не так. Палитры до сих пор активно применяются при работе с цветными битовыми образами, при хранении растровых изображений (в частности, таких как BMP, GIF и TIFF), при программировании WEB-приложений.

15

Аппаратно-зависимые растры

Все существующие на данный момент многочисленные разновидности графических форматов изображений разделяются на два фундаментальных класса: *растровую* и *векторную* графику. Суть различия заключается в особенностях представления изображения в памяти. В первом случае это битовый массив, в ячейках которого содержатся данные о цветовых характеристиках каждого пиксела изображения. Во втором случае рисунок представляется в виде коллекции функций, способных рисовать графические примитивы. При последовательном вызове этих функций строится целостная картинка. Каждый из графических форматов имеет положительные и отрицательные черты, исходя из которых и определяется его область применения. Так, основное назначение растровой графики – хранение фотографических изображений, а векторная графика незаменима при построении чертежей и схем.

Этой главой открывается материал, посвященный рассмотрению вопросов хранения и обработки растровых изображений. Видов растровых объектов GDI всего два: *аппаратно-зависимый* (*Device-Dependent Bitmap*, DDB) и *аппаратно-независимый битовый образ* (*Device-Independent Bitmap*, DIB).

Аппаратно-зависимым называется растр, предназначенный для воспроизведения на конкретном устройстве со статическими графическими характеристиками. Тесная связь с графическим устройством сказывается как на достоинствах, так и на недостатках растров DDB. К преимуществам аппаратно-зависимого растра стоит отнести относительную простоту (по сравнению с рассматриваемом в следующей главе DIB) представления образа в памяти. Технически это всего лишь массив пикселов, считав который сверху вниз, мы получим изображение (лишь бы образ соответствовал суровым требованиям устройства вывода). Герой следующей главы этой книги – DIB-растр – приобрел независимость от аппаратной части за счет существенного усложнения собственной структуры. Для того чтобы любое устройство смогло вывести рисунок DIB, последний кроме собственно изображения содержит сложный заголовок и другую справочную информацию, рассказывающие

устройству, каким именно образом следует обращаться с битовым образом. Еще одно достоинство DDB – возможность применения функций GDI для раскраски его пикселов и проведения более сложных растровых операций. Такими качествами образ DIB похвастаться не может. Если рассуждать о недостатках DDB, то они раскрыты в его названии. Вся жизнедеятельность DDB находится в тесной связи с создавшим его приложением. Вполне вероятно, что при смене глубины цветов одно и то же приложение окажется не в состоянии работать с созданным им же объектом DDB. Ко всему прочему существуют ограничения на размер DDB растра. Так, в Windows 95/98 (в которых GDI 16-разрядный) максимальный размер аппаратно-зависимого растра составляет 16 Мбайт. Это много только на первый взгляд. Например, 32-битная картинка размером 1280×1024 пикселов (классическая развертка 19-дюймового монитора) задействует почти треть этого пространства – около 5 Мбайт. В более продвинутых версиях Windows (начиная с Windows NT 4.0) верхний предел увеличен до 48 Мбайт. При этом появляется другая проблема – в Windows NT/XP/2003 память для DDB выделяется в адресном пространстве ядра, а это ведет к перерасходу системных ресурсов.

Ключевое достоинство DDB заключается в том, что Windows поддерживает прямую запись в пикселы аппаратно-зависимого растра посредством команд GDI.

DDB – это полноценный объект GDI, обладающий своей структурой и набором обслуживающих его функций. Доступ к объекту DDB производится посредством дескриптора:

```
HBitmap = type LongWord;
```

Аппаратно-зависимые растры просто неоценимы при выводе изображения в режиме двойной буферизации. Это тот случай, когда результирующее изображение предварительно формируется в оперативной памяти и только затем переносится на экран. Благодаря такой последовательности действий полностью исключается раздражающий пользователя эффект мерцания.

Представление монохромного DDB в памяти

Как уже отмечалось ранее, аппаратно-зависимые растры не стандартизированы. Уточним – за малым исключением. Созданный с помощью `CreateBitmap()` простейший черно-белый образ DDB понятен всем устройствам вывода. Все благодаря тому, что формат представления монохромного битового образа в памяти не отличается особым изыском. Белому пикселу соответствует двоичная единичка, черному – двоичный ноль.

Говоря о битовых образах, часто применяют термин «скан-линия». Представленный на рис. 15.1 черно-белый образ содержит восемь таких линий. Первая строка описывается значением 11100111 (hE7 в шестнадцатеричном представлении), вторая – 11100111 и так далее. Так как размер скан-линии должен быть кратен 16, для описания битовых образов требуется применять сдвоенные байты (тип данных `Word`) или двойное слово (`DWord`).

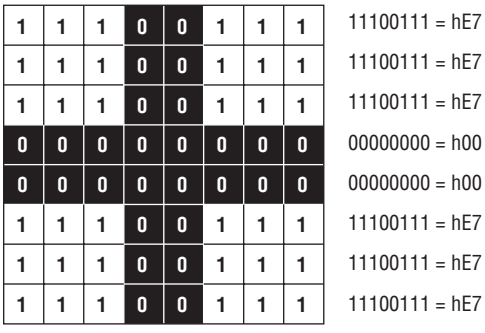


Рис. 15.1. Представление монохромного DDB в памяти

Анализируя рис. 15.1, можно сделать еще один вывод: изображения DDB строятся в системе координат, соответствующей режиму отображения `MM_TEXT`. Другими словами, чтение аппаратно-зависимого битового образа должно производиться слева направо, сверху вниз.

Растр DDB всегда считывает скан-линии сверху вниз, в отличие от него растр DIB привык читать данные снизу вверх.

При описании любого не монохромного аппаратно-зависимого битового образа свои коррективы начинает вносить цветность, здесь формат представления данных зависит от глубины цвета и количества цветовых плоскостей. Привычные нам режимы SVGA характеризуются всего одной плоскостью, но так было не всегда. Например, в старом добром 16-цветном VGA использовалась другая схема. В этом случае задействовался только один бит на пиксел, но зато имелось целых четыре плоскости: красная, зеленая, синяя и плоскость интенсивности.

Создание DDB

Процесс создания аппаратно-зависимого образа относительно прост. В GDI реализовано три функции, специализирующиеся на формировании экземпляра DDB: `CreateBitmap()`, `CreateBitmapIndirect()` и `CreateCompatibleBitmap()`. Кроме этого, функция `CreateDIBitmap()` умеет формировать аппаратно-зависимый растр на основе DIB. Изучение перечисленных функций начнем с самой общей функции:

```
Function CreateBitmap(Width, Height, Planes, BitCount : Integer;
                      pBits : Pointer) : HBitmap;
```

Функция требует указания ширины и высоты будущей картинки (`Width` и `Height`), числа цветовых плоскостей `Planes` и глубины цвета `BitCount`. Самый последний параметр `pBits` – указатель на массив раскрашенных пикселов (в простейшем случае это может быть статический массив соответствующей размерности). Если указанный параметр пуст, то создается неинициализированный растровый рисунок. В случае неудачи функция отрапортует нам

значением ERROR_INVALID_BITMAP, а если все в порядке, мы получим дескриптор нового DDB-растра. Ниже представлен пример создания монохромного инициализированного DDB, повторяющего битовый образ с рис. 15.1.

```
const Bits : Array[0..7] of Word =($E7,$E7,$E7,$0,$0,$E7,$E7,$E7);
var   DDB:HBitmap;
begin
DDB:=CreateBitmap(8,8,1,1,@Bits);
//другие операции, например вывод на экран
DeleteObject(DDB);
end;
```

Аналогом CreateBitmap() является функция:

```
function CreateBitmapIndirect(const BMP: TBitmap): HBITMAP;
```

Разница между коллегами по аппаратно-зависимому формату заключена в способе передачи данных для описания характеристик и инициализации битового образа. В случае CreateBitmapIndirect() требуется лишь один параметр. Это структура, содержащая основные данные о будущем рисунке.

```
type TBitmap = packed record
    bmType: Longint;       //тип битового образа, всегда равен 0
    bmWidth: Longint;      //ширина
    bmHeight: Longint;     //высота
    bmWidthBytes: Longint; //ширина скан-линии в байтах, значение кратно 2
    bmPlanes: Word;        //количество цветовых плоскостей
    bmBitsPixel: Word;     //глубина цвета
    bmBits: Pointer;       //указатель на массив с данными о цветах пикселов
end;
```

Пожалуйста, не спутайте описанную в модуле Windows структуру TBitmap *с одноименным классом из модуля Graphics! В этой главе мы работаем со структурой, а с классом мы познакомимся в главе 24 « Растровые и векторные изображения в VCL».*

Нетрудно заметить, что перечень полей структуры TBitmap полностью дублирует параметры функции CreateBitmap(). Два «лишних» поля (bmType и bmWidthBytes) предназначены скорее для того, чтобы немножко взбодрить программиста. Тип битового образа bmType никогда не используется, и, хотя в SDK говорится, что поле всегда должно быть равным нулю, в Delphi допустимо передавать в него любое число. Если вы работаете со стандартной для Windows глубиной цвета (8, 16, 24 или 32 бит/пиксел), то расчет ширины скан-строки в байтах (поле bmWidthBytes) достоин задачника по арифметики для 1-го класса:

```
bmWidthBytes:=bmWidth*bmBitsPixel div 8;
```

Таким образом, с точки зрения программирования CreateBitmap() и CreateBitmapIndirect() представляют разные стороны одной медали.

Попрактикуемся вновь, на этот раз рассмотрим пример инициализации цветного DDB размером 16×16 пикселов и вывод его в окне.

```
Procedure GDI_PAINT(DC:HDC);
var Bits:Array[0..255] of DWORD;
    memDC:hDC;
    Bitmap:TBitmap;
    DDB:HBitmap;
    X:Integer;
begin
 memDC:=CreateCompatibleDC(DC);

 for x:=0 to 255 do Bits[x]:=$FF0000; //закрашиваем в красный цвет весь массив

 with Bitmap do
  begin
  bmType:=0; bmPlanes:=1;
  bmWidth:=16; bmHeight:=16; bmBitsPixel:=32;
  Bitmap.bmWidthBytes:=bmWidth*bmBitsPixel div 8;
  Bitmap.bmBits:=@Bits;
  end;

  DDB:=CreateBitmapIndirect(Bitmap);

  SelectObject(memDC,DDB);

  BitBlt(DC,100,50,100,100,memDC,0,0,SRCCOPY);
  DeleteDC(memDC);
  DeleteObject(DDB);
end;
```

В модуле Graphics.pas среды Delphi объявлен одноименный класс `TBitmap`*, инкапсулирующий растровый образ. Чтобы избежать конфликта имен при обращении к структуре* `TBitmap`*, в проектах VCL перед названием имени класса указывайте название модуля:*

```
var Bitmap:Windows.TBitmap;
```

Нам потребовалось приложить немало усилий для осуществления элементарной операции – вывода в окне красного квадратика. Построенный нами DDB образ описывается единственной плоскостью и 32-битной глубиной цвета. Самое обидное в том, что программа сохранит работоспособность только при условии, что текущие настройки устройства графического вывода будут полностью соответствовать параметрам DDB. В противном случае контекст окна проигнорирует все наши «художества». Для вывода растра в окне мы были вынуждены пойти на хитрость. Первоначально мы создали совместимый контекст в памяти компьютера, затем выбрали в него образ DDB и только потом с помощью растровой операции `BitBlt()` скопировали изображение из совместимого контекста в контекст окна. Заметьте – это вовсе не шаманский трюк, а вполне классический для GDI способ вывода изображений битовых образов, позволяющий избежать эффекта мерцания.

Хотя набор возможностей `CreateBitmap()` *и* `CreateBitmapIndirect()` *вполне позволяет создавать цветные битовые образы, официальная документация Microsoft SDK рекомендует применять эти функции только для работы с черно-белыми растрами, иначе вам придется тратить значительные силы на преодоление проблемы*

несовместимости DDB и устройства вывода. Для цветных образов целесообразно использовать `CreateCompatibleBitmap()`.

Структура `TBitmap` окажется одинаково полезной как при работе с DDB, так и с DIB. Например, рекомендую взять на вооружение следующую функцию, возвращающую геометрические размеры растрового объекта, описываемого дескриптором `hBMP:HBitmap`.

```
Function GetBitmapSize(hBMP:HBitmap):TSize;
var hBitmap:^Windows.TBitmap;
begin
 GetMem(hBitmap,SizeOf(Windows.TBitmap)); //выделяем память для структуры
 GetObject(hBMP,SizeOf(TBitmap),hBitmap); //сбор данных
 Result.cx:=hBitmap^.bmWidth;             //ширина образа
 Result.cy:=hBitmap^.bmHeight;            //высота образа
 FreeMem(hBitmap,SizeOf(Windows.TBitmap));//освобождаем память
end;
```

Сердцем нашей функции является обращение к функции `GetObject()`, которая умеет собирать основную информацию о ряде стандартных объектов GDI (в нашем случае битовом образе). Полученную информацию мы передаем в поля записи `TSize`.

На страницах этой книги мы еще неоднократно воспользуемся услугами функции `GetBitmapSize()`.

DDB, совместимый с контекстом устройства

Дабы несколько снизить остроту проблемы обеспечения совместимости аппаратно-зависимого растра и устройства вывода при создании цветного DDB, программисты обычно используют функцию:

```
Function CreateCompatibleBitmap(DC : HDC;
                                Width, Height : integer) : HBitmap;
```

При работе с функцией мы освобождаемся от проблем определения числа цветовых плоскостей и глубины цвета. Все это назначается без вмешательства программиста в соответствии с текущими настройками контекста устройства, описываемого дескриптором `DC`. От нас требуется лишь определиться с размерами (`Width` и `Height`) будущего рисунка DDB.

Ниже предложен пример применения функции `CreateCompatibleBitmap()` при выводе изображения в режиме двойной буферизации.

```
Procedure GDI_PAINT(DC:HDC);
var memDC:hDC;
    DDB:HBitmap;
    Brush:HBRUSH;
begin
memDC:=CreateCompatibleDC(DC); //создаем совместимый контекст в памяти
   DDB:=CreateCompatibleBitmap(memDC,100,100); //создаем образ DDB
   SelectObject(memDC,DDB);    //выбираем DDB в совместимый контекст
```

```
//ТЕПЕРЬ МЫ РИСУЕМ В ПАМЯТИ НА ПОВЕРХНОСТИ ОБРАЗА DDB
   Brush:=CreateHatchBrush(HS_DIAGCROSS,$000000);
   SelectObject(memDC,Brush);
   Ellipse(memDC,0,0,99,99);
//РИСОВАНИЕ В ПАМЯТИ ЗАВЕРШЕНО
   BitBlt(DC,0,0,99,99,memDC,0,0,SRCCOPY); //копируем рисунок из памяти
DeleteDC(memDC);
DeleteObject(DDB);
DeleteObject(Brush);
end;
```

Обратите внимание на то, что, в отличие от рассмотренных выше функций создания DDB, функция `CreateCompatibleBitmap()` формирует неинициализированный DDB. Для того чтобы наполнить рисунок содержимым, мы выбрали битовый образ в совместимый контекст и нарисовали в нем эллипс.

В состав 32-разрядного GDI включена еще одна функция, умеющая создавать неинициализированные битовые образы в формате DDB – `CreateDiscardableBitmap()`. Она предназначена лишь для поддержки обратной совместимости с устаревшими 16-разрядными версиями Windows.

К сожалению, применение аппаратно-независимого образа в проектах с двойной буферизацией имеет одно существенное ограничение. Растровый образ (выбранный в качестве рабочей поверхности в контексте в памяти) крайне критичен к операциям, связанным с мировыми и страничными преобразованиями!

DDB на основе аппаратно-независимого растра

Кроме изученных нами трех функций, позволяющих формировать зависящий от устройства битовый образ, в недрах GDI от нас прячется функция, умеющая конструировать DDB-образ на основе растра DIB. С некоторой степенью допущения можно говорить, что `CreateDIBitmap()` конвертирует DIB в DDB (сначала создается аппаратно-независимый растр, и только затем он преобразуется в DDB).

```
function CreateDIBitmap(DC: HDC;
                        var InfoHeader: TBitmapInfoHeader;
                        INIT: DWORD;
                        InitBits: PChar;
                        var InitInfo: TBitmapInfo;
                        Usage: UINT): HBITMAP;
```

Чтобы программистам не было скучно, функция решила впечатлить нас обширным перечнем параметров. Впрочем, привыкайте – это характерная черта большинства функций, специализирующихся на работе с изображениями. В части, касающейся первого аргумента, все понятно без слов: это дескриптор контекста устройства. Он необходим на завершающем этапе работы, когда уже будет сформирован DIB-растр. Благодаря дескриптору `DC` функция создаст из аппаратно-независимого растра совместимый с контекстом образ DDB. Оставшиеся параметры предназначены для описания будущего образа DIB. Переменная `InfoHeader` представляет собой заголовок опи-

сания растра, включающий более десятка полей. Параметр INIT определяет, нужно ли инициализировать создаваемый образ. Если в него передано нулевое значение, то в результате работы функции сформируется пустой объект. Если же направить в параметр константу CBM_INIT, то в оставшихся трех параметрах надо описать DIB-растр. Для этого в InitBits передается указатель на массив пикселов, в InitInfo – цветовая таблица. Наконец, заключительный параметр Usage указывает, что именно хранится в цветовой таблице:

- индексы палитры (константа DIB_PAL_COLORS);
- реальные цвета (константа DIB_RGB_COLORS).

Пока я сознательно не раскрываю подробности структур TBitmapInfoHeader и TBitmapInfo, к ним мы вернемся в следующей главе этой книги, при обсуждении DIB-растра.

Загрузка растра из ресурса

Растровые образы активно эксплуатируются в системе в качестве пояснительных рисунков и пиктограмм к элементам управления и пунктам меню. Наличие вспомогательного изображения делает приложение более привлекательным, а интерфейс управления программой – интуитивно понятным. Предусмотрено несколько способов подключения к проекту битовых образов. Чаще всего они интегрируются непосредственно в исполнительный файл приложения или во внешний файл ресурса, в роли последнего обычно выступают динамические библиотеки. Несмотря на то, что в ресурсе растры хранятся в формате DIB, доступ к ним предоставляется в формате DDB-растра. Для этого предназначена функция:

```
function LoadBitmap(Instance: HINST; BitmapName: PAnsiChar): HBITMAP;
```

Здесь Instance – экземпляр библиотеки, загруженной с помощью функции LoadLibrary(). Второй параметр BitmapName – название ресурса. Если название неизвестно, то вместо него допускается передать числовой идентификатор, но в этом случае нам потребуются услуги макроса MAKEINTRESOURCE(). В случае успешного выполнения функция возвратит дескриптор аппаратно-зависимого растра.

Имеется соглашение об именовании модулей, содержащих в себе ресурсы пользовательского интерфейса. Обычно названия таких файлов заканчиваются буквами «ui» (сокр. от англ. User Interface).

В составе Windows имеется несколько десятков файлов, содержащих ресурсы битовых образов. Мы проведем исследование библиотеки themeui.dll из состава Windows XP, она содержит API тем оформления Windows, поэтому наверняка в ней найдется что-нибудь интересное. Если место установки вашей операционной системы стандартно, то ресурс обнаружится по адресу:

```
const
  LibName = 'C:\WINDOWS\system32\themeui.dll'; //название библиотеки
```

Для загрузки библиотеки в адресное пространство процесса воспользуемся функцией Windows API LoadLibrary(). Обращение к ней производим в момент создания окна – в обработчике сообщения WM_CREATE. Выгрузка библиотеки осуществляется в рамках обработки сообщения WM_DESTROY, здесь мы вызовем функцию FreeLibrary().

```
var LibInst:Integer; //идентификатор библиотеки

function WindowProc(Window: HWND; Msg: UINT;
                    wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
  case Msg of
 WM_CREATE  : LibInst:=LoadLibrary(pChar(LibName)); //загрузка библиотеки
  WM_DESTROY : begin
               FreeLibrary(LibInst); //выгрузка библиотеки
               PostQuitMessage(0);
               Exit;
               end;
  WM_PAINT   : begin
               DC:=BeginPaint(Window,PAINTSTRUCT);
               SetBKMode(DC,TRANSPARENT);
               GDI_PAINT(DC);               //вывод библиотеки
               EndPaint(Window,PAINTSTRUCT);
               end;
  end;
  WindowProc := DefWindowProc (Window, Msg, WParam, LParam);
end;
```

Вывод входящих в библиотеку битовых образов произведем в процедуре GDI_PAINT(). Заметьте, что в ней мы воспользовались помощью разработанной несколькими страницами ранее функции GetBitmapSize().

```
Procedure GDI_PAINT(DC:HDC);
var   DDB:HBitmap;
      Size:TSize;
      ResNum,Y:word;
      S:String;
      memDC:HDC;
Begin
memDC:=CreateCompatibleDC(DC);
Y:=5;
if LibInst>0 then //если библиотека успешно загружена
   for ResNum:=0 to 199 do
   begin
   DDB:=LoadBitmap(LibInst,MAKEINTRESOURCE(ResNum));
    if DDB>0 then
     begin
     Size:=GetBitmapSize(DDB); //узнаем размерность образа
     SelectObject(memDC,DDB);
     BitBlt(DC,5,Y,Size.cx,Y,memDC,0,0,SRCCOPY);
     s:='№ '+IntToStr(ResNum);
     TextOut(dc,Size.cx+10,Y,PChar(s),Length(s));
```

```
        Y:=Y+Size.cy+2;
        end;
      DeleteObject(DDB);
      end;
  TextOut(DC,0,Y,pChar(LibName),Length(LibName));
  DeleteDC(memDC);
  End;
```

Так как числовые идентификаторы растровых ресурсов нам не были заранее известны, пришлось проанализировать более сотни номеров `ResNum`, пока мы не добрались до намеченной цели. В недрах библиотеки пряталось целых семь битовых образов (идентификаторы 120...126). Эти изображения применяются при настройке свойств экрана, вы их обнаружите на странице «Параметры» соответствующего диалога (рис. 15.2).

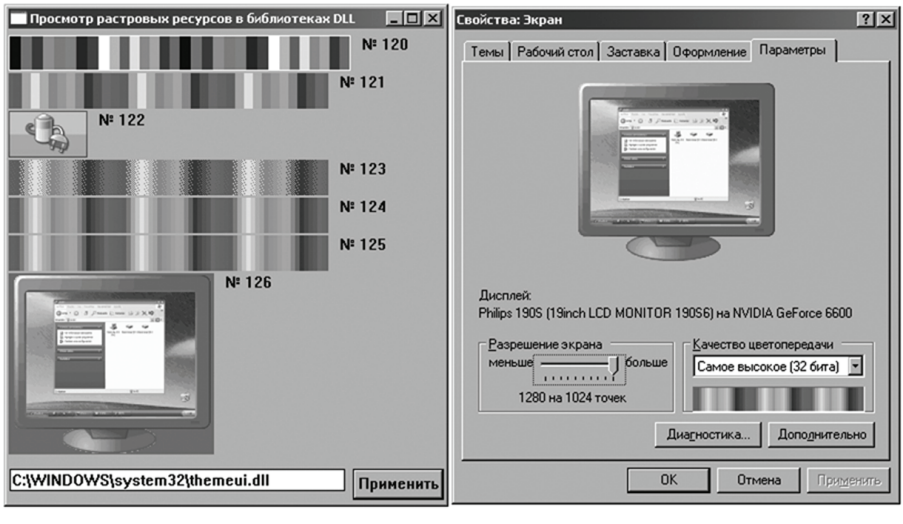


Рис. 15.2. Ресурс библиотеки themeui.dll и страница «Параметры» свойств экрана Windows XP

Среди примеров к главе 35 «Изображения в GDI+» вы найдете приложение, позволяющее экспортировать графический ресурс из библиотеки пользовательского интерфейса в файл BMP.

Универсальная функция отображения битового образа

На протяжении этой главы мы неоднократно сталкивались с задачей вывода растрового образа на поверхности окна, и каждый раз для этого требовалось 5–6 строк кода. Появление повторяющегося кода в листингах программы – явный признак того, что пора задуматься о преобразовании этих строк в процедуру или функцию. Кроме дескриптора обслуживаемого образа заголовок

функции отображения битового образа должен содержать дескриптор контекста (на поверхности которого мы собираемся вывести растр), а также координаты точки вывода.

```
Function DrawBitmap(DC:HDC; hBMP:HBitmap; X,Y:Integer):boolean;
var Size:TSize;
    memDC:hDC;
begin
if GetObjectType(hBMP)=OBJ_BITMAP then
begin
  Size:=GetBitmapSize(hBMP); //выясняем размеры рисунка
  memDC:=CreateCompatibleDC(DC);
  SelectObject(memDC,hBMP);
  Result:=BitBlt(DC,X,Y,Size.cx,Size.cy,memDC,0,0,SRCCOPY);
  DeleteDC(memDC);
end else result:=false;
end;
```

При работе с полученным извне битовым образом функция обязана уметь выяснить основные его характеристики, в первую очередь – геометрический размер рисунка. Для решения этой задачи нам на помощь пришла функция `GetObject()`. При ее посредничестве мы заполняем структуру `Bitmap` и узнаем габариты картинки.

Представленная процедура демонстрирует очень популярный среди программистов прием формирования изображения в памяти с последующим выводом на поверхность реального графического устройства. Такой поход называется ***буферизацией вывода****, он позволяет избежать неприятного побочного эффекта мерцания рисунка, вызванного задержками в работе графического адаптера и монитора.*

С задачей вывода растра на поверхность контекста устройства способны справиться еще несколько функций. Наиболее значимая из них – `StretchBlt()`*, у нее имеется еще одна положительная черта – она позволяет масштабировать изображения.*

РЕЗЮМЕ

Аппаратно-зависимые растры играют существенную роль в графическом механизме Windows. Их основное преимущество в том, что DDB-растр может быть не только источником данных, но и приемником операций графического вывода. Основной недостаток DDB – жесткая привязка к графическим параметрам устройства, малейшее изменение глубины цвета или числа цветовых плоскостей делает образ несовместимым с графическим устройством.

16

Аппаратно-независимые растры

Наряду с аппаратно-зависимыми DDB-растрами в Windows активно применяются их коллеги – *аппаратно-независимые битовые образы* (*Device-Independent Bitmaps*, DIB). Для того чтобы приобрести столь желанную независимость от устройства вывода, DIB-растр платит высокую цену, он вынужден всегда носить с собой дополнительный груз – служебные данные, в первую очередь – содержащие описание цветовых характеристик изображения. Ознакомившись с технической информацией, устройство самостоятельно подберет подходящий способ отображения DIB-растра. Пойдя на такие жертвы, DIB-растр заслужил право существовать не только в памяти компьютера, но и сохраняться в виде файла или ресурса.

Сразу упомянем серьезную проблему DIB. Она заключается в том, что GDI не способен напрямую обращаться к пикселам аппаратно-независимого образа.

Формат DIB-растра и файла BMP

Наш предыдущий знакомый (аппаратно-зависимый растровый объект) лишен каких-либо архитектурных излишеств, он содержит только самое главное – массив с данными о цветах пикселов. Поэтому на фоне DDB-растра конструкция битового образа DIB выглядит более чем солидно. Помимо собственно изображения, DIB включает больше десятка информационных полей, разъясняющих устройству, как требуется этот рисунок отображать. Здесь данные о размере изображения, его цветовой организации, алгоритме сжатия, разрешении и о многом другом.

В структуру DIB-образа входят две обязательные и две опциональные части:

- Заголовок описания растра.
- Битовая маска (только для несжатых изображений с глубиной цвета 16 и 32 бита).
- Цветовая таблица (только при глубине цвета 8 бит и меньше).
- Массив данных.

К настоящему времени различными разработчиками программного обеспечения реализовано несколько десятков способов хранения аппаратно-независимого растра в виде файла. Ряд из них стандартизирован или фактически стал стандартом (GIF, JPEG, TIFF), ряд постоянно совершенствуется (PNG). Еще с момента становления Windows ее родным растровым форматом стал файл BMP (*BitMaP*). На сегодняшний день различают несколько версий этого формата. Наиболее старый (и, соответственно, наиболее простой) файл BMP был спроектирован для операционной системы фирмы IBM – OS/2. К разработке этой ОС имели непосредственное отношение многие программисты Microsoft, так что Windows переняла формат BMP на вполне законных основаниях. Не берусь предполагать, куда делся BMP версии 2, но следующие по счету варианты форматов файлов BMP имеют нумерацию 3, 4 и 5. Наиболее распространенный формат файла BMP считается третьей версией, он разрабатывался вместе с 16-разрядной Windows 3.0. Очередная четвертая модификация файла BMP осуществлена в Windows 95 и Windows NT 4.0, пятая появилась благодаря Windows 98.

Для того чтобы DIB-образ смог превратиться в файл BMP, к нему добавляют еще один заголовок (рис. 16.1). Заголовок файла одинаков для всех вариа-

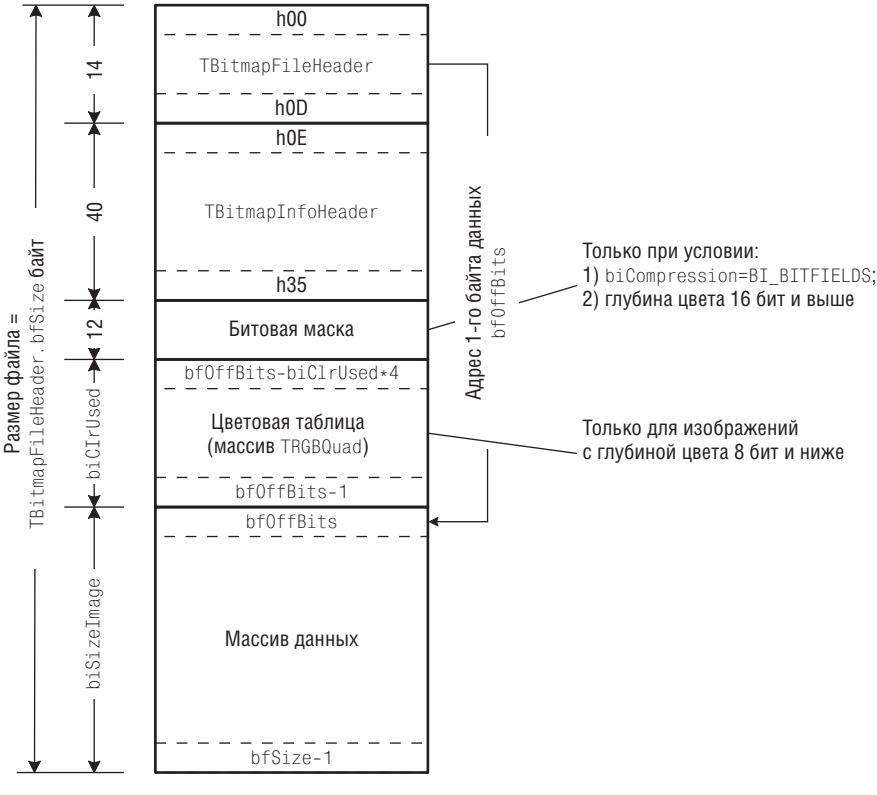


Рис. 16.1. Структура BMP-файла третьей версии

ций файлов на тему BMP. Это структура из пяти полей общей размерностью 14 байт:

```
type TBitmapFileHeader= packed record
  bfType: Word;       //тип файла, сигнатура «BM»
  bfSize: DWord;      //размер файла в байтах
  bfReserved1: Word;  //не используется
  bfReserved2: Word;  //не используется
  bfOffBits: DWord;   //порядковый номер первого байта массива данных
end;
```

Первые два байта записи всегда содержат визитную карточку всех файлов BMP. Это ASCII коды латинских символов «B» (h42) и «M» (h4D). Во втором по счету поле `bfSize` находятся данные о размере файла, 3-е и 4-е поля не используются. Последнее поле записи хранит адрес первого байта массива данных растрового рисунка.

Все различия между форматами DIB раскрываются в заголовке описания растра. Заголовок растра DIB-образа для OS/2 определяется записью `TBitmapCoreHeader`. Аппаратно-независимый растр третьей версии – `TBitmapInfoHeader`, четвертой – `TBitmapV4Header`, пятой – `TBitmapV5Header`. Каждая из последующих модификаций заголовка включает все поля из предыдущей версии, плюс вносит дополнительную информацию. В этой главе мы рассмотрим наиболее распространенную структуру растра третьей версии `TBitmapInfoHeader`:

```
type TBitmapInfoHeader = packed record
    biSize: DWORD;            //размер структуры
    biWidth: Longint;         //ширина рисунка в пикселах
    biHeight: Longint;        //высота рисунка в пикселах
    biPlanes: Word;           //число цветовых плоскостей, всегда равно 1
    biBitCount: Word;         //глубина цвета в битах на пиксел
    biCompression: DWORD;     //способ сжатия изображения
    biSizeImage: DWORD;       //размер массива данных, иногда может быть равен 0
    biXPelsPerMeter: Longint; //разрешение по оси X
    biYPelsPerMeter: Longint; //разрешение по оси Y
    biClrUsed: DWORD;         //размер цветовой таблицы
    biClrImportant: DWORD;    //число цветов в таблице
end;
```

Цветовое разрешение DIB (поле `biBitCount`) может принимать строго определенное значение: 0, 1, 4, 8, 16, 24 или 32 бит/пиксел. Нулевое значение глубины цвета – косвенный признак того, что мы работаем не с обычным растром, а, например, с изображением в формате JPEG или PNG. При разрешении цвета 16 бит/пиксел и выше необходимость в цветовой таблице отпадает.

Поле `biCompression`, определяющее метод сжатия растрового изображения, описано в табл. 16.1.

Поля `biClrUsed` и `biClrImportant` описывают цветовую таблицу DIB-образа. Без такой таблицы не могут обойтись растровые образы с глубиной цвета 8 бит и ниже. Если параметр `biClrUsed` установлен в ноль, то это прямой признак того, что эта таблица не используется. Строка цветовой таблицы (DIB-образов 3, 4 и 5 версии) представляет собой 32-битную структуру:

```
type TRGBQuad = packed record
    rgbBlue:     Byte;//синий цвет
    rgbGreen:    Byte;//зеленый цвет
    rgbRed:      Byte;//красный цвет
    rgbReserved: Byte;//резерв
  end;
```

Таблица 16.1. Компрессия изображения, возможные значения поля biCompression

Константа	Описание
BI_RGB	Несжатый битовый образ.
BI_RLE8	Построчное сжатие (*run-length encoded*, RLE), применяемое для образов с глубиной цвета 8 бит. Формат эффективен только если в изображении имеются строки одинакового цвета.
BI_RLE4	Сжатие RLE, применяемое для 4-битных образов.
BI_BITFIELDS	Несжатый полноцветный образ (16..32 бит/пиксел). В этом случае в состав DIB включается 3 значения (тип DWORD) цветовой маски для красной, зеленой и синей составляющих. В 16-битном формате с помощью масок определяется, сколько бит отводится для основных цветов: 5-5-5 (15-битный цвет) или 5-6-5 (полный 16-битный цвет с усиленной зеленой составляющей). При 32-битной глубине цвета синяя, зеленая и красная маски принимают значения: h000000FF, h0000FF00, h00FF0000.
BI_JPEG	Изображение формата JPEG.
BI_PNG	Изображение формата PNG.

Элемент цветовой таблицы файла BMP формата OS/2 определен записью TRGBTriple *размерностью в 24 бита.*

Так как лучше один раз увидеть, чем сто раз услышать, проведем наглядный эксперимент – рассмотрим дамп небольшого растрового файла в формате bmp.

На рис. 16.2 (см. цветную вклейку, с. 418) представлен шестнадцатиразрядный дамп растрового файла fly.bmp из библиотеки к настоящему изданию на сайте издательства. Это маленькая картинка размерностью 630 байт, соответствующая файлу bmp третьей версии. Геометрические размеры рисунка 32×32 пиксела, он построен всего из 16 цветов, поэтому содержит палитру.

В дампе легко обнаружить все признаки файла bmp. В первую очередь это 14 байт заголовка файла TBitmapFileHeader, начинающегося со значений 42, 4D – коды латинских символов «B» и «M». Сразу за заголовком файла следует 40 байт описания растра TBitmapInfoHeader. Так как рисунок не полноцветный, то между структурой TBitmapInfoHeader и собственно изображением включена цветовая таблица – 16 записей TRGBQuad. Таким образом, на служебные данные отводится 118 байт из 630, на рисунок осталось ровно пол килобайта – 512 байт. В оставшихся 512 байтах нашлось место для описания 32×32 =1024 пикселов изображения – 2 пиксела в одном информационном

байте. Несмотря на такую оптимизацию при визуальном рассмотрении дампа данных можно заметить контуры рисунка – перевернутой бабочки. Почему перевернутой? Потому что в отличие от DDB образы в формате DIB должны считываться снизу вверх.

Отображение DIB-растра

Для обеспечения вывода аппаратно-независимого растра в GDI реализована функция StretchDIBits(). Она позволяет полностью или частично копировать битовый образ DIB на поверхность устройства, описываемого контекстом DC.

```
function StretchDIBits(DC: HDC;
                       DestX, DestY, DestWidth, DestHeight,
                   SrcX, SrcY, SrcWidth, SrcHeight: Integer;
                   Bits: Pointer;
                   var BitsInfo: TBitmapInfo;
                   Usage: UINT; ROP: DWORD): Integer;
```

Координаты левого верхнего угла и размеры области вывода определяются четверкой параметров DestX, DestY, DestWidth, DestHeight. Для того чтобы функция могла работать в различных режимах отображения, размеры прямоугольника-получателя передаются в логических единицах. Очередной квартет аргументов (SrcX, SrcY, SrcWidth, SrcHeight) определяет аналогичные характеристики DIB-растра. На этот раз в качестве единицы измерения используется пиксел. Затем следует указатель на массив пикселов – параметр Bits. Параметр BitsInfo не прочь получить описание битового образа.

```
type TBitmapInfo = packed record
     bmiHeader: TBitmapInfoHeader;       //заголовок растра
     bmiColors: array[0..0] of TRGBQuad; //цветовая таблица
  end;
```

При обращении к структуре TBitmapInfo в своих программах не забывайте, что по умолчанию под цветовую таблицу (поле bmiColors) выделяется всего 4 байта памяти. Такого объема хватит только для одного элемента таблицы. Посему распределяйте память самостоятельно. В качестве эффективного подручного средства рекомендую функцию GetMem().

```
var BitmapInfo:^TBitmapInfo;
begin
GetMem(BitmapInfo, SizeOf(TBitmapInfoHeader) + 12 + 256 * SizeOf(TRGBQuad));
//…
FreeMem(BitmapInfo,SizeOf(BitmapInfo^));
end;
```

Продолжим речь о StretchDIBits(). Параметр Usage определяет порядок работы с таблицей цветов DIB-растра, он может принимать одно из двух значений: DIB_PAL_COLORS или DIB_RGB_COLORS. Первая константа указывает функции, что массив пикселов содержит 16-битные указатели на логическую палитру, второй вариант – цветовая таблица образа хранит реальные RGB цвета. Последний параметр определяет растровую операцию, в соответствии

с которой пикселы рисунка станут комбинироваться с пикселами поверхности устройства отображения. Это тема достойна отдельной главы (см. главу 17 «Растровые операции»), а пока, не вдаваясь в подробности, в этот параметр мы станем передавать константу SRCCOPY. В этом случае пикселы приемного прямоугольника станут замещаться соответствующими пикселами битового образа. Еще одна особенность функции заключается в возможности масштабирования рисунка и создания его зеркального отображения. Для достижения этого требуется поэкспериментировать со значениями и знаком параметров, отвечающих за размеры области вывода и растра.

Архаичные машины с 4–8 Мбайт ОЗУ при выводе на экран растрового рисунка больших размеров испытывали существенные затруднения. Проблемы с объемом доступной памяти, умноженные на ограничения устаревшего 16-разрядного GDI (Windows 3.X/95/98), приводили к тому, что файл BMP даже полностью не загружался в память. Поэтому специалистами Microsoft была предложена функция, позволяющая передавать DIB-образ не весь сразу, а по частям:

Win GDI

```
function SetDIBitsToDevice(DC: HDC; DestX, DestY: Integer; Width,
                    Height: DWORD; SrcX, SrcY: Integer;
                    StartScan, NumScans: UINT;
                    Bits: Pointer;
                    var BitsInfo: TBitmapInfo; Usage: UINT): Integer;
```

Приемное устройство описывается тремя параметрами: дескриптором DC и координатами стартовой точки (DestX, DestY). Заметьте, что здесь отсутствуют параметры, задающие ширину и высоту приемной области, их значения будут соответствовать размерам исходного прямоугольника. Следующая четверка аргументов уже посвящена DIB-образу: Width, Height – ширина и высота, (SrcX, SrcY) – координата левого нижнего пиксела, с которого начнется вывод. Еще раз повторюсь – начало координат для пиксельных данных находится не в левом верхнем, а в левом нижнем углу. Обещанная изюминка функции кроется в параметрах StartScan, NumScans – это номер сканлинии, с которой начнется вывод, и количество скан-линий рисунка, отправляемых на вывод за один вызов функции. Далее следуют точно такие же параметры, как и у функции StretchDIBits(), за единственным ограничением – отсутствует код растровой операции.

Масштабирование

Изменение геометрических размеров – больное место всех растровых рисунков. Решение этого вопроса сопряжено с внесением искажений в результирующее изображение. Проще всего решается задача увеличения размера битового образа. Нам достаточно отобразить один исходный пиксел в несколько пикселов получателей. При уменьшении рисунка задачи GDI значительно усложняются. В работу вступает алгоритм, управляющий процессом отображения группы пикселов в один. В Windows NT/2000/XP таких алгоритмов четыре. Каким именно алгоритмом станет руководствоваться Windows при изменении размеров изображения, зависит от функции:

Win GDI
```
function SetStretchBltMode(DC: HDC; StretchMode: Integer): Integer;
```

Как вы заметили, функция не выводит изображение на поверхность контекста – это прерогатива функции StretchBlt(). Она лишь определяет алгоритм масштабирования (табл. 16.2) и возвращает константу предыдущего режима масштабирования.

Таблица 16.2. Алгоритмы масштабирования при уменьшении размеров рисунка

Константа	Описание
BLACKONWHITE, STRETCH_ANDSCANS	Между цветами удаляемого и сохраняемого пиксела выполняется операция логического «И». Алгоритм применяется только для монохромных растров.
COLORONCOLOR, STRETCH_DELETESCANS	Быстрый алгоритм для цветных рисунков. В этом режиме удаляются «лишние» пикселы без обработки находящихся в них данных.
HALFTONE, STRETCH_HALFTONE	Наиболее эффективный алгоритм с точки зрения качества рисунка. Результирующий цвет пиксела определяется среднеарифметическим значением от цвета нескольких пикселов исходного изображения. Опция не поддерживается в Windows 95/98/Me.
WHITEONBLACK, STRETCH_ORSCANS	Между цветами удаляемого и сохраняемого пиксела выполняется операция логического «ИЛИ».

Какой именно режим масштабирования в настоящий момент применяет контекст, расскажет функция:

Win GDI
```
function GetStretchBltMode(DC: HDC): Integer;
```

Она возвратит одну из констант, представленных в табл. 16.2.

В модуле GraphUtil *среды Delphi объявлена функция* ScaleImage(), *позволяющая физически масштабировать битовый образ, представленный в виде экземпляра класса* TBitmap.

Пример загрузки образа DIB из файла BMP

Настал этап демонстрации особенностей работы с растровым изображением, сохраненным в файл формата BMP. Ради сокращения объема листинга и для повышения наглядности кода оговорим некоторые ограничения. Разрабатываемая процедура предназначена только для работы с файлами BMP наиболее распространенной – 3-й версии, для остальных форматов мы оставим «заглушки», которые позднее вы сможете заполнить самостоятельно. Кроме того, наш файл не должен быть сжат (biCompression= BI_RGB). Даже с такими упрощениями листинг получился достаточно внушительным:

```
Procedure ShowBMP(DC:HDC; FileName:pAnsiChar);
var F:THandle;
    BitmapFileHeader:TBitmapFileHeader; BitmapInfoHeader:TBitmapInfoHeader;
    BitmapInfo : ^TBitmapInfo; HeaderSize,X,Y,I :DWORD;
```

```
    Bits : array of DWORD;
begin
BitmapInfo:=nil;
F:=CreateFile(FileName, {открываем файл FileName}
              GENERIC_READ,FILE_SHARE_READ,nil,
              OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0);
TRY
  {читаем заголовок файла}
  ReadFile(F,BitmapFileHeader,SizeOf(TBitmapFileHeader),X,nil);
  IF BitmapFileHeader.bfType=$4D42 then {проверка на наличие сигнатуры «BM»}
  begin
  ReadFile(F,HeaderSize,SizeOf(DWORD),X, nil);{выясняем размер заголовка растра}
  //возвращаемся к началу заголовка растра
  SetFilePointer(F,SizeOf(TBitmapFileHeader),nil,FILE_BEGIN);
  CASE HeaderSize OF //благодаря размеру заголовка мы выясняем версию BMP файла
   SizeOf(TBitmapCoreHeader): //заглушка - версия для OS/2;

   SizeOf(TBitmapInfoHeader):
begin //3-я версия
     ReadFile(F,BitmapInfoHeader,SizeOf(TBitmapInfoHeader),X,nil);
     //смещаемся к началу массива данных
     SetFilePointer(F,BitmapFileHeader.bfOffBits,nil,FILE_BEGIN);
     //распределяем память для массива данных
     if BIH.biSizeImage>0 then
       SetLength(Bits, BIH.biSizeImage DIV SizeOf(WORD))
     Else //иногда бывает и так biSizeImage=0!!!
       SetLength(Bits, (BIH.biWidth*BIH.biHeight*BIH.biBitCount)DIV SizeOf(WORD));
     //считываем массив данных
     for i:=0 to HIGH(Bits)-1 do ReadFile(F,Bits[i], SizeOf(DWORD),X, nil);
     {распределяем память для BitmapInfo, предполагая наличие цветовой таблицы}
     GetMem(BitmapInfo, SizeOf(TBitmapInfoHeader) + 12 + 256 * SizeOf(TRGBQuad));
           BitmapInfo^.bmiHeader:=BitmapInfoHeader;

if BitmapInfoHeader.biClrUsed>0 then //если применяется цветовая таблица
begin
  {адрес первого байта таблицы: bfOffBits-biClrUsed*4}
  Y:=BitmapFileHeader.bfOffBits-BitmapInfoHeader.biClrUsed*4;
  SetFilePointer(F,Y,nil,FILE_BEGIN); //смещаемся к началу цветовой таблицы
  for I:=Y to BitmapFileHeader.bfOffBits-1 do BFH.bfOffBits-1 //читаем данные
     ReadFile(F,BitmapInfo^.bmiColors[I-Y], SizeOf(TRGBQuad), X, nil);
end;
//выводим DIB на поверхности графического устройства
StretchDIBits(DC,0,0,BitmapInfo^.bmiHeader.biWidth,
              BitmapInfo^.bmiHeader.biHeight,
              0,0,BitmapInfo^.bmiHeader.biWidth,
              BitmapInfo^.bmiHeader.biHeight,
              Bits, BitmapInfo^, DIB_RGB_COLORS, SRCCOPY);
end;
SizeOf(TBitmapV4Header): //4-я версия файла;
SizeOf(TBitmapV5Header): //5-я версия файла;
end; //конец селектора CASE
```

```
end else Abort; //это не файл формата BMP
FINALLY
if Assigned(BitmapInfo) then FreeMem(BitmapInfo,SizeOf(BitmapInfo^));
Finalize(Bits);
CloseHandle(F);
END;
end;
```

Пример в большей степени учебный, и ему пока не стоит искать практическое применение. Однако в листинге раскрыты основные аспекты по работе с файлом BMP. Посредством обращения к функции `CreateFile()` мы получаем доступ к файлу с именем, указанным в параметре `FileName`. Считав файловый заголовок в переменную-запись `BitmapFileHeader`, проверяем данные в поле `bfType`. Наличие сигнатуры «BM» свидетельствует о том, что открытый нами файл действительно BMP. Следующим нашим шагом станет выяснение версии файла. Проще всего это сделать, проанализировав поле `biSize` растрового заголовка. Вне зависимости от версии BMP размер заголовка хранится в следующих после заголовка файла 4 байтах. Считав байты h0E...h11 в переменную `HeaderSize`, в рамках селектора `Case` выясняем версию файла. Вся остальная работа сводится к последовательному чтению данных из файла и заполнению соответствующих структур и массивов. В заключение обращаемся к функции `StretchDIBits()` и выводим изображение на поверхности устройства.

Рассматривайте эту процедуру не более, чем как учебный пример. Чуть позднее (при обсуждении DIB-секции) мы разработаем более полезную функцию, позволяющую загружать DIB-образ из файла. Эту функцию вы сможете применять в своих проектах.

Перенос пикселов между DIB и DDB

Сам факт существования двух способов представления битовых образов предполагает наличие функций, позволяющих конвертировать растры друг в друга. Функция `GetDIBits()` получает массив пикселов из DDB-образа и представляет его в формате, приемлемом для DIB. Ее напарница – функция `SetDIBits()` – наоборот, переносит массив пикселов DIB-образа в DDB, используя представление цвета аппаратно-независимого растра. Обе функции обладают идентичным набором параметров:

```
function GetDIBits(DC: HDC; Bitmap: HBitmap;
                   StartScan, NumScans: UINT;
                  pBits: Pointer;
                  var BitInfo: TBitmapInfo; Usage: UINT): Integer;
```

или

```
function SetDIBits(DC: HDC; Bitmap: HBITMAP;
                   StartScan, NumScans: UINT;
                 pBits: Pointer;
                 var BitsInfo: TBitmapInfo; Usage: UINT): Integer;
```

Обширный список аргументов условно можно разделить на две части. Первые два аргумента описывают DDB, а оставшиеся пять – DIB-образ. В обеих функциях DC – это дескриптор контекста устройства, его графические характеристики станут использоваться в качестве эталонных при копировании пикселов между образами. Параметр Bitmap – дескриптор DDB-растра. Пара StartScan и NumScans – номер скан-линии, с которой начнется перенос данных, и число переносимых линий (благодаря этим параметрам возможно постепенное копирование данных между растрами). Если у функции SetDIBits() параметр pBits содержит указатель на исходный массив данных, то у функции GetDIBits() в параметре pBits должен находиться указатель на буфер, принимающий массив конвертируемых пикселов. В ситуации, когда размер буфера заранее неизвестен, функцию придется вызвать дважды. При первом обращении в pBits направляется неопределенный указатель nil, в этом случае GetDIBits() возвратит количество байт, необходимых для корректной инициализации буфера. Распределив требуемый объем памяти, повторно обращаемся к GetDIBits(), теперь приемный буфер заполнится данными. Описание формата DIB-растра направляется в параметр BitsInfo. Последний параметр может принимать значение DIB_PAL_COLORS (цветовая таблица содержит индексы цветов в палитре) или DIB_RGB_COLORS (цветовая таблица содержит реальные цвета).

В предыдущей главе мы упомянули еще одну функцию, осуществляющую преобразование DIB в DDB. Функция CreateDIBitmap() *создает новый аппаратно-зависимый растр на основе описания DIB-образа.*

DIB-секция

Проводя сравнительный анализ DDB и DIB образов, мы отметили положительные и отрицательные черты обоих подходов к хранению изображений. Аппаратно-зависимый растр позволяет создавать инициированные образы и даже допускает прямое обращение к пикселам изображения, но DDB привязан к устройству и теряет жизнеспособность при малейшем изменении графических параметров этого устройства. Аппаратно-независимый растр свободен от недостатка DDB, заложенная в нем дополнительная информация позволяет формировать изображения, способные работать на любых графических устройствах вывода. Но, к сожалению, при создании классического DIB мы не сможем инициировать его образ из приложения. Ко всему прочему DIB не разрешает рисовать на своей графической поверхности с помощью функций Windows GDI.

Попытка программистов объединить ключевые достоинства аппаратно-зависимых и аппаратно-независимых образов привела к появлению на свет очередного растрового объекта GDI – *DIB-секции*. DIB-секция является полноценным объектом GDI, доступ к которому осуществляется посредством дескриптора HBITMAP. Что послужило первопричиной выбора такого несколько необычного названия, скрыто от нас в коридорах Microsoft. Сегодня уже поздно дискутировать на эту тему, посему нам лишь остается пользоваться устоявшейся терминологией.

На основе всего сказанного попробуем сформулировать определение: DIB-секция – это DIB-растр, позволяющий приложению напрямую работать со своими данными.

При работе с DIB-секцией могут использоваться все функции Windows API, предназначенные для обслуживания образов DDB, поэтому своих собственных функций у секции всего три.

Для создания экземпляра DIB-секции предназначена функция:

```
function CreateDIBSection(DC: HDC; const BMI: TBitmapInfo;
                          Usage: UINT; var pBits: Pointer;
                hDIBSection: THandle; Offset: DWORD): HBITMAP;
```

Здесь DC – дескриптор контекста; BMI – структура содержащая атрибуты секции. Параметр Usage может принимать одно из двух значений: DIB_RGB_COLORS или DIB_PAL_COLORS. Первый вариант указывает на то, что массив пикселов хранит цвета в формате RGB. Второй вариант определяет то, что секция опирается на цветовую таблицу, в которой хранятся 16-разрядные индексы цветов из логической палитры устройства (идентифицируемого дескриптором DC). Выходной параметр pBits содержит адрес переменной, в которую функция передаст адрес пустого массива пикселов создаваемой DIB-секции. Для инициализации DIB-секции может использоваться файл, отображаемый в память, в этом файле должен находиться массив пикселов. Если мы направились по этому пути, то в параметр hDIBSection требуется передать дескриптор созданного функцией CreateFileMapping объекта файлового отображения, а порядковый номер первого байта массива пикселов необходимо указать в параметре Offset. Значение Offset должно быть кратным двойному слову DWORD. Если необходимости в применении механизма файлового отображения нет, в параметр hDIBSection направляется нулевое значение. При удачном выполнении функция возвращает дескриптор DIB-секции и массив пикселов (через параметр pBits).

*При описании заголовка DIB-секции нельзя использовать константы сжатия (*BI_RLE4, BI_RLE8, BI_JPEG *и* BI_PNG*).*

Если секция создается на основе объекта файлового отображения, то в параметр hDIBSection передается указатель на структуру TDIBSection:

```
type  TDIBSection = packed record
   dsBm: TBitmap;              //основные характеристики образа (см. гл. 14)
   dsBmih: TBitmapInfoHeader; //заголовок битового образа
   dsBitfields: array[0..2] of DWORD; {3 цветовые маски,
                                       применяемые при глубине свыше 8 бит}
   dshSection: THandle; //дескриптор объекта файлового отображения из функции
   dsOffset: DWORD; {адрес первого байта массива пикселов, расположенного
в отображаемом файле}
end;
```

Если при создании секции объект файлового отображения не применяется, нам остается определиться с размерами образа и особенностями цветовой

реализации – заполнить структуру TBitmapInfo. В этом случае массив пикселов размещается в адресном пространстве процесса, посредником между этим массивом и нами выступает указатель pBits.

Ниже представлен фрагмент листинга, демонстрирующий процесс создания DIB-секции. В коде формируется растровый образ с глубиной цвета 8 бит и размерами 50×50 пикселов. Таблица цветов секции инициируется оттенками серого цвета.

```
Procedure CreateDIBS(DC:HDC);
var pBMI:^TBitmapInfo;
    memDC:HDC;
    DIB_S:HBITMAP;
    pPixels:Pointer;
    i:byte;
begin
//инициализация заголовка
GetMem(pBMI,SizeOf(TBitmapInfoHeader) + 12 + 256 * SizeOf(TRGBQuad));
with pBMI^.bmiHeader do
 begin
 biSize:=SizeOf(TBitmapInfoHeader);
 biWidth:=50;
 biHeight:=50;
 biPlanes:=1;
 biBitCount:=8; //используем таблицу цветов
 biCompression:=BI_RGB;
 biSizeImage:=biWidth*biHeight*biBitCount div 8;
 biClrUsed:=0;
 end;
{заполняем таблицу оттенками серого}
  for i:=0 to 255 do
    with pBMI^.bmiColors[i] do
    begin
    rgbBlue:=i; rgbGreen:=i; rgbRed:=i;
    end;
{создаем совместимый контекст в памяти}
memDC:=CreateCompatibleDC(DC);
DIB_S:= CreateDIBSection(memDC,pBMI^,DIB_RGB_COLORS,pPixels,0,0);

SelectObject(memDC,DIB_S);
SelectObject(memDC,GetStockObject(WHITE_PEN));
Ellipse(memDC,10,10,40,40);
BitBlt(DC,0,0,50,50,memDC,0,0,SRCCOPY);

FreeMem(pBMI);
DeleteObject(DIB_S);
DeleteDC(memDC);
end;
```

После создания DIB-секции в наше распоряжение попадают два ценных «предмета»: дескриптор секции DIB_S и указатель на массив пикселов pPixels. Первый пригодится для обращения к секции с помощью функций Windows

GDI, второй позволит работать с рисунком напрямую. По умолчанию все пикселы пустого образа инициируются черным цветом. Мы, в свою очередь, рисуем эллипс.

Цветовая таблица DIB-секции

Для работы с изображениями с цветовым разрешением 8 бит/пиксел и ниже в состав DIB-секции включена таблица, содержащая перечень используемых в изображении цветов. Для обслуживания указанной таблицы в Windows GDI подготовлена пара функций:

```
function GetDIBColorTable(DC: HDC; StartIndex, Entries: UINT;
                          var RGBQuadStructs): UINT;
```

и

```
function SetDIBColorTable(DC: HDC; StartIndex, Entries: UINT;
                          var RGBQuadStructs): UINT;
```

Функция `GetDIBColorTable()` исследует `Entries` элементов из цветовой таблицы DIB-секции. Анализ начинается с элемента с порядковым номером `StartIndex`. Результат возвращается при посредничестве параметра `RGBQuadStructs` в виде указателя на массив типа `TRGBQuad`. Перед обращением к функции необходимо убедиться, что битовый образ выбран в совместимый контекст устройства. Если работа функции завершилась успешно, то она проинформирует нас о количестве переданных нам элементов цветовой таблицы. Вторая функция `SetDIBColorTable()` решает обратную задачу – она изменяет цветовую таблицу образа. Назначение ее параметров аналогично функции `GetDIBColorTable()` за единственным исключением – здесь параметр `RGBQuadSTructs` предназначен для получения данных.

Пример загрузки образа DIB из файла BMP с помощью DIB-секции

Наконец, настала очередь написать серьезную функцию, позволяющую загружать растровый рисунок из файла. Функция построена по образу и подобию представленной несколькими страницами ранее функции `ShowBMP`, но на этот раз мы воспользовались DIB-секцией. Благодаря этому наша функция возвращает дескриптор загруженного растрового изображения, что позволит работать с битовым образом с помощью многих функций Windows GDI.

```
Function LoadBMP(DC:HDC; const FileName : pANSIChar):HBitmap;
var F:THandle; memDC:hDC; DIB_S:HBitmap;
    BFH:TBitmapFileHeader;  BIH:TBitmapInfoHeader; pBitmapInfo : ^TBitmapInfo;
    Bits : array of WORD;   pBits:Pointer;
    X,Y,HeaderSize,I:DWORD;
begin
Result:=0;
if FileAge(FileName)<0 then //проверка факта существования файла
    RAISE EExternal.Create('Файл '+StrPas(FileName)+' не обнаружен!');
```

```
F:=CreateFile(FileName,
              GENERIC_READ,FILE_SHARE_READ,nil,
              OPEN_EXISTING,FILE_ATTRIBUTE_NORMAL,0);
TRY
 ReadFile(F,BFH,SizeOf(TBitmapFileHeader),X,nil); //читаем заголовок файла
 //Проверка на наличие сигнатуры «MB»
 IF BFH.bfType<>$4D42 then RAISE EExternal.Create('Неверный формат файла!');
//выясняем размер заголовка растра, по нему мы подберем нужную структуру
 ReadFile(F,HeaderSize,SizeOf(DWORD),X,nil);
 //возвращаемся к началу заголовка
 SetFilePointer(F,SizeOf(TBitmapFileHeader),NIL,FILE_BEGIN);

 IF HeaderSize=SizeOf(TBitmapCoreHeader) then
    ReadFile(F,BIH,SizeOf(TBitmapCoreHeader),X,nil) //формат OS/2
    else ReadFile(F,BIH,SizeOf(TBitmapInfoHeader),X,nil); {3-я и любая другая
версия}
//смещаемся к началу битового массива
   SetFilePointer(F,BFH.bfOffBits,nil,FILE_BEGIN);
//устанавливаем размер массива
    if BIH.biSizeImage>0 then
    SetLength(Bits, BIH.biSizeImage DIV SizeOf(WORD))
    Else //иногда поле biSizeImage=0
    SetLength(Bits,
             (BIH.biWidth*BIH.biHeight*BIH.biBitCount DIV 8)DIV SizeOf(WORD));
   //заполняем массив пикселов
   for i:=0 to HIGH(Bits)-1 do ReadFile(F,Bits[i],SizeOf(WORD),X, nil);

   //распределение памяти под структуру TBitmapInfo
   if BIH.biClrUsed>0 then
       GetMem(pBitmapInfo,
              SizeOf(TBitmapInfoHeader) + 12 + 256 * SizeOf(TRGBQuad))
       else GetMem(pBitmapInfo, SizeOf(TBitmapInfoHeader));

    pBitmapInfo^.bmiHeader:=BIH;

    if BIH.biClrUsed>0 then //если есть цветовая таблица
    begin
       //первый байт цветовой таблицы bfOffBits-biClrUsed*4
       Y:=BFH.bfOffBits-BIH.biClrUsed*4;
       SetFilePointer(F,Y,NIL,FILE_BEGIN);
       //заполняем цветовую таблицу
       for i:=Y to BFH.bfOffBits-1 do
         {последний байт цветовой таблицы BFH.bfOffBits-1}
          ReadFile(F,pBitmapInfo^.bmiColors[i-Y],sizeof(TRGBQuad),X, nil);
    end; //работа с таблицей цветов завершена

    memDC:=CreateCompatibleDC(DC); //создаем совместимый контекст устройства
    //создаем DIB-секцию с параметрами, повторяющими DIB-образ из исходного файла
    DIB_S:=CreateDIBSection(memDC,pBitmapInfo^,DIB_RGB_COLORS,pBits,0,0);
    SelectObject(memDC,DIB_S);
    {из-за того, что секция создается с неинициализированным массивом
     пикселов pBits, переносим в этот массив данные, полученные из файла}
    StretchDIBits(memDC, 0, 0, pBitmapInfo^.bmiHeader.biWidth,
```

```
                pBitmapInfo^.bmiHeader.biHeight, 0, 0,
                pBitmapInfo^.bmiHeader.biWidth,
                pBitmapInfo^.bmiHeader.biHeight,
                Bits, pBitmapInfo^, DIB_RGB_COLORS, SRCCOPY);
    Result:=DIB_S;
    DeleteDC(memDC)
FINALLY
CloseHandle(F);
END;
end;
```

РЕЗЮМЕ

Аппаратно-независимый битовый образ (*Device-Independent Bitmaps*, DIB) помимо массива пикселов, хранящего описание рисунка, как минимум вынужден содержать структуру с информацией о параметрах изображения – заголовок описания растра. Именно благодаря этому дополнению DIB приобретает ценную возможность отображаться на любом графическом устройстве вывода, вне зависимости от его текущих настроек. Поэтому образы DIB послужили фундаментом для создания различных форматов файлов для хранения растровой графики (в качестве наиболее популярного представителя семейства стоит привести файл BMP).

DIB-секция – это следующий шаг в процессе совершенствования битовых образов. В секции устранен ключевой недостаток обычного DIB – невозможность прямого редактирования рисунка, вместе с этим секция сохранила независимость от графического устройства. Таким образом, DIB-секция объединила лучшие положительные черты аппаратно-зависимого и аппаратно-независимого представления битового образа.

17

Растровые операции

На первый взгляд, рисование в Windows не что иное, как банальное закрашивание пикселов. Казалось бы – единственная задача GDI заключается в отыскании требуемого пиксела и заполнении его подходящим цветом. А если все это было проделано в кратчайшие сроки и с наименьшей нагрузкой на процессор, то пользователь просто обязан оказаться на седьмом небе от счастья. Что здесь такого? Вспомните, то же самое делали и мы, рисуя чертиков в тетрадках в клеточку на уроках вместо того, чтобы с благоговением внимать учителю литературы... Вместе с тем, создавая графическое ядро Windows, неугомонные программисты Microsoft не ограничились единственной функцией `SetPixel()` и реализовали великое множество разнообразных высокохудожественных (и не очень) методов. Они чертят линии и дуги, выводят текст, закрашивают области и делают много чего еще полезного и непонятного. Что общего у всех этих методов? А все то же – они находят нужные пикселы и присваивают им соответствующие цвета. Мы вернулись к тому, с чего начинали? И да, и нет...

Представьте себе все доступные пикселы экрана (точнее, кадрового буфера видеокарты) в виде двумерного массива D размерностью x на y. Процесс прорисовки пиксела заключается в указании его координат и передаче по этому адресу нового значения цвета P, скажем: $D[x, y]=P$. Все очень просто и как нельзя лучше подойдет для вывода где-то на экране точки, и это, пожалуй, все... Программистов очень скоро перестало устраивать подобное упрощенное решение. И вот в чем дело. Когда краска с кисти художника ложится на холст, она взаимодействует с поверхностью холста или с другими красками, изменяя результирующий цвет. Когда мы пишем в тетради, цвет чернил складывается с белизной бумаги. Когда на вас опрокидывается бокал с соком, цвет пятна представляет собой производную от цветов фруктов и материала брюк. Почему бы не реализовать такую интересную возможность и в GDI? Не пугайтесь, речь идет не о симбиозе прохладительных напитков и дорогой одежды, а о взаимодействии двух составляющих – цвета закрашиваемого пиксела и цвета пера.

Раз мы намереваемся учитывать характеристики холста, то предложенная выше упрощенная формула $D[x, y]=P$ преобразуется в более серьезную: $D[x, y]=f(D[x, y], P)$. Здесь f – функция, описывающая взаимодействие старого $D[x, y]$ и нового P цветов пиксела. Подобная функция двух параметров в Windows называется *бинарной растровой операцией* (*binary Raster OPeration*), сокращенно *ROP2*. Существуют и более сложные растровые операции, именуемые *тернарными* и *кватернарными*. В первом случае результирующий цвет пиксела является результатом выполнения функции трех параметров: $D[x, y]=f(D[x, y], P, S)$. Тернарные операции имеют место при объединении цветов пиксела-источника S и пиксела-получателя $D[x, y]$ с учетом цвета текущего пера (кисти) P. Другими словами– это случай наложения одного изображения (например, вашей фотографии) на другое изображение (фотография пляжа олимпийского Сочи) с учетом цвета холста. Кватернарные операции предназначены для того, чтобы окончательно повергнуть в трепет программистов, разобравшихся с более простыми растровыми операциями. В дополнение к трем параметрам (используемым в тернарных операциях) кватернарная операция получает 4-й аргумент – монохромный растр маски.

Участники растровых операций

Управляя процессом вывода растровой картинки на экран, GDI производит поразрядные булевы операции между вовлеченными в процесс объектами. В случае ROP2 в роли участников растровой операции выступают два действующих лица:

1. Пиксел-получатель D (*destination*).
2. Пиксел пера P (*pen*).

При проведении тернарных операций к перечисленным выше объектам добавляется третий товарищ:

3. Пиксел-источник S (*source*).

Зачастую в тернарных операциях, называя второго участника операции, вместо термина «перо» используют термин «кисть» или «шаблон», но смысл операции от этого не изменяется.

Кватернарные операции дополнены еще одним параметром:

4. Растр маски M (*mask*).

С только что анонсированными действующими лицами осуществляются следующие логические операции:

- Конъюнкция AND (логическое «И»).
- Дизъюнкция OR (логическое «ИЛИ»).
- Инверсия NOT (логическое «НЕТ»).
- XOR (логическое исключение «ИЛИ»).

Таким образом, в случае бинарных растровых операций мы имеем право утверждать о существовании $2^{2\times2}=16$ вариантов операций. Если речь ведется о тернарных операциях, то в нашем распоряжении окажется $2^{2\times2\times2}=256$ вер-

сий логических операций с растром. При использовании кватернарных операций предусмотрено $2^{2\times2\times2}\times2=512$ модификаций.

Бинарные растровые операции, ROP2

Бинарные растровые операции относятся к простейшим. Они определяют способ объединения пера с пикселом-получателем. В GDI поименовано 16 разновидностей бинарных растровых операций (табл. 17.1).

Таблица 17.1. Коды бинарных растровых операций

Код ROP2	Операция	Описание
R2_BLACK	D=0	Пиксел-получатель *D* устанавливается в черный цвет.
R2_COPYPEN	D=P	Пиксел *D* принимает цвет пера *P*.
R2_MASKNOTPEN	D=D AND (NOT P)	Конъюнкция цветов пиксела *D* и инвертированного пера *P*.
R2_MASKPEN	D=D AND P	Конъюнкция пера *P* и приемника *D*.
R2_MASKPENNOT	D=(NOT D) AND P	Конъюнкция цветов пера *P* и инвертированного пиксела *D*.
R2_MERGENOTPEN	D=D OR (NOT P)	Дизъюнкция пиксела *D* и инвертированного пера *P*.
R2_MERGEPEN	D=D OR P	Пиксел *D* представляет собой дизъюнкцию цветов пера *P* и пиксела *D*.
R2_MERGEPENNOT	D=(NOT D) OR P	Дизъюнкция цветов инвертированного пиксела *D* и пера *P*.
R2_NOP	D=D	Цвет пиксела *D* не изменяется.
R2_NOT	D=NOT D	Инверсия цвета пиксела *D*.
R2_NOTCOPYPEN	D=NOT P	Цвет пиксела *D* инвертируется по отношению к цвету пера *P*.
R2_NOTMASKPEN	D=NOT (D AND P)	Инверсия операции R2_MASKPEN.
R2_NOTMERGEPEN	D=NOT (D OR P)	Инверсия операции R2_MERGEPEN.
R2_NOTXORPEN	D=NOT (D XOR P)	Инверсия операции R2_XORPEN.
R2_WHITE	D=1	Пиксел *D* устанавливается в белый цвет.
R2_XORPEN	D=D XOR P	Пиксел *D* и перо *P* объединяются по правилу исключения «ИЛИ».

С процессом управления бинарными растровыми операциями успешно справляется пара функций GDI:

Win GDI
```
Function GetROP2(DC : HDC):integer;
```

и

Win GDI
```
Function SetROP2(DC : HDC; DrawMode : integer):integer;
```

Первая из них информирует программиста о том, какой именно режим ROP2 в текущий момент установлен у графического устройства с дескриптором контекста DC. По умолчанию это режим R2_COPYPEN, соответствующий обычному переносу цвета пера в пиксел-приемник. Для смены растровой операции предназначена функция SetROP2(). Для этого она вооружена параметром DrawMode, в который и направляется подходящий код ROP2. В случае успешного выполнения функция возвращает код предыдущего состояния ROP2.

Хотите получить яркий пример роли программных продуктов ROP2 в формировании графического интерфейса? Нет проблем. Разработчики графических редакторов, систем автоматизированного проектирования, игрушек и других программных продуктов часто применяют режим R2_NOT или R2_NOTXORPEN для вывода на экран временных линий, например для отображения перекрестья (допустим, прицела снайперской винтовки). Максимально упрощенная идея подобного кода представлена в следующем листинге. Ядро листинга составляет обработка сообщения WM_MOVE, возникающая в момент перемещения указателя мыши в клиентской области окна.

```
var //…
  xPos, yPos, xMem, yMem : word;
  OldDrawMode:Integer;
//…
Procedure PaintCross(DC:HDC; X, Y : word); //перекрестье
begin
MoveToEx(DC, X-50, Y, nil); LineTo(DC, X+50, Y);
MoveToEx(DC, X, Y-50, nil); LineTo(DC, X, Y+50);
end;

function WindowProc (Window: HWND; Msg: UINT;
                     wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
CASE Msg OF
WM_MOUSEMOVE :
begin
DC:=GetDC(Window);
 xPos:=LOWORD(lParam);             //координата X указателя мыши
 yPos:=HIWORD(lParam);             //координата Y указателя мыши
 OldDrawMode:=SetROP2(DC,R2_NOT); //выбираем режим инвертирования
 PaintCross(DC,Xmem,Ymem);         //стирание старых линий
 PaintCross(DC,xPos,yPos);         //вывод новых линий
 SetROP2(DC, OldDrawMode);         //восстанавливаем предыдущий режим ROP2
 Xmem:=xPos; Ymem:=yPos;           //сохраняем координаты указателя
ReleaseDC(DC,Window);
end;
//обработка остальных сообщений
```

В проекте объявлены две глобальные переменных xMem и yMem. Они специализируются на хранении старых координат курсора. Для прорисовки и последующего быстрого стирания линии достаточно дважды вызвать вложенную вовнутрь обработчика события процедуру рисования перекрестья PaintCross() с указанием старых и новых координат курсора.

Если мы опираемся на цветовое пространство RGB, то бинарные растровые операции применяются не для логического цвета в целом, а для каждой отдельной составляющей цвета в модели RGB. Например, в этом случае цвет RGB (0,200,255) после инвертирования превратится в RGB (255,55,0). Если же мы работаем с цветовой палитрой, то растровые операции применяются к цветовым индексам. В этом случае результат операции зависит от порядка следования цветов в палитре.

Тернарные растровые операции

Для проведения тернарной операции GDI задействует троих участников: пиксел-получатель, кисть и пиксел-источник. В литературе, описывая растровые операции, кисть зачастую называют шаблоном, поэтому не пугайтесь, когда встретите этот термин. В качестве специалистов по тернарным операциям рекомендую функции GDI Windows: `PatBlt()`, `BitBlt()` и `StretchBlt()`. Рассмотрим их по мере возрастания сложности решаемых задач.

Функция PatBlt()

Название функции `PatBlt()` получилось из английского словосочетания *pattern block transfer*, что дословно означает «перенос блока шаблона». Идея в следующем – функция закрашивает прямоугольную область кистью-шаблоном. Но это не просто «заливка» области. Ко всему прочему `PatBlt()` выполняет логическую операцию между пикселами прямоугольника и шаблона.

```
Function PatBlt(DC:hDC; X, Y, Width, Height : integer;
                ROP : cardinal) : Boolean;
```

Здесь `DC` – контекст устройства, `X` и `Y` – координаты левого верхнего угла прямоугольной области, `Width` и `Height` – логическая ширина и логическая высота прямоугольной области, последний параметр метода `ROP` – код операции. Функция имеет право использовать любой (из описанных в табл. 17.1) код, но наиболее часто применяют операции, представленные в табл. 17.2.

Таблица 17.2. Код растровой операции для функции PatBlt( )

Значение	Описание
`PATCOPY`	Функция копирует кисть-шаблон в область-получатель.
`PATINVERT`	Комбинирует цвета кисти-шаблона и прямоугольной области, используя логическую операцию XOR.
`DSTINVERT`	Инвертирует прямоугольную область.
`BLACKNESS`	Заливает прямоугольную область цветом, имеющим нулевой индекс в физической палитре (по умолчанию это черный цвет).
`WHITENESS`	Заливает прямоугольную область цветом, имеющим индекс 1 в физической палитре (по умолчанию это белый цвет).

Представленный ниже листинг демонстрирует работу `PatBlt()`. Вызов функции с ROP-кодом `WHITENESS` закрасит принимающую область белым цветом, с кодом `BLACKNESS` – черным (рис. 17.1).

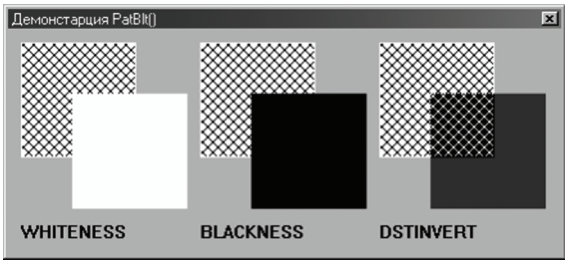


Рис. 17.1. Демонстрация возможностей PatBlt( )

```
Procedure GDI_PAINT(DC:HDC);
var Brush:hBrush;
begin
SelectObject(DC,GetStockObject(DC_PEN));
SetDCPenColor(DC,$FFFFFF);
Brush:=CreateHatchBrush(HS_DIAGCROSS, $000000);
SelectObject(DC,Brush);

Rectangle(DC, 10,10,100,100);
PatBlt (DC, 50, 50, 90, 90, WHITENESS);

Rectangle(DC, 150,10,240,100);
PatBlt (DC, 190, 50, 90, 90, BLACKNESS);

Rectangle(DC, 290,10,380,100);
PatBlt (DC, 330, 50, 90, 90, DSTINVERT);

DeleteObject(Brush);
end;
```

Самое интересное происходит при работе с цветом. В этом случае Windows проводит логические операции для каждой цветовой составляющей по отдельности. Например: ROP-код `DSTINVERT` инвертирует результирующую область: черный цвет в белый, белый в черный, светло-серый в темно-серый, красный в голубой и т. д.

Вновь вернемся к особенностям определения координат прямоугольной области в функции `PatBlt()`. Как уже отмечалось ранее, ее параметры (`X`, `Y`) задают верхнюю левую вершину прямоугольника, вторая пара (`Width` и `Height`) отвечает за его логическую ширину и логическую высоту. Надо помнить, что такое распределение обязанностей параметров справедливо только для режима отображения `MM_TEXT`. С переходом в метрические режимы (`MM_LOMETRIC`, `MM_HIMETRIC`, `MM_LOENGLISH`, `MM_HIENGLISH` или `MM_TWIPS`) функция начинает «безобразничать». В подобных случаях координаты верхнего левого угла начинают определяться знаками параметров `Width` и `Height`. В зависимости от режима отображения и знаков параметров `xWidth` и `yHeight` левым верхним углом прямоугольной области может стать не классическая точка (`X`, `Y`), а точка (`X`, `Y + yHeight`), или (`X + xWidth`, `Y`), или (`X + xWidth`, `Y + yHeight`). Это утверждение справедливо для всех функций `Blt…` Такое, на первый взгляд необычное, поведение методов объясняется тем, что в различных режимах ото-

бражения координата Y может возрастать при движении вверх (а не вниз, как в MM_TEXT), соответственно, координата X может увеличиваться при движении вправо (а не влево, как в MM_TEXT).

Функция BitBlt()

Функция BitBlt() представляет собой усовершенствованную версию PatBlt(). Неординарный результат достигается за счет вовлечения в тернарную операцию дополнительного действующего лица – контекста области-источника.

```
Function BitBlt(DestDC : HDC; X, Y, Width, Height : Integer;
                SrcDC : HDC; XSrc, YSrc : Integer;
                ROP : Cardinal) : boolean;
```

Функциональная нагрузка первой пятерки параметров ничем не отличается от задач аналогичных параметров рассмотренной ранее функции PatBlt(), они определяют область-получатель. Шестой по счету параметр SrcDC описывает контекст устройства источника. Следующая за ним пара аргументов XSrc и YSrc хранит начала координат прямоугольника-источника. Ширину и высоту этой области отдельно описывать не требуется, функция предполагает, что она такая же, как Width и Height. Наконец, в параметр ROP направляется код тернарной операции (табл. 17.3).

Таблица 17.3. Часто используемые растровые операции для функции BitBlt( )

Константа	Операция	Описание
BLACKNESS	D=0	Закрашивает область-получатель кистью. Цвет кисти соответствует цвету физической палитры с нулевым индексом. Обычно это черный цвет.
DSTINVERT	D=(**NOT** D)	Инвертирует область-получатель
MERGECOPY	D=P **AND** S	Объединяет цвета всех участников операции (исходный прямоугольник и кисть-шаблон), используя логическую операцию «И».
MERGEPAINT	D=(**NOT** D) **OR** S	Объединяет цвета инвертированной области-получателя и области-источника на основе операции «ИЛИ».
NOTSRCCOPY	D=**NOT** S	Копирует инвертированный прямоугольник-источник в область-получатель.
NOTSR-CERASE	D=**NOT** (D **OR** S)	Операция объединяет цвета области-получателя и области-источника на основе операции «ИЛИ» и инвертирует результат.
PATCOPY	D=P	Заполняет область-приемник текущей кистью.
PATINVERT	D=**NOT** (D **OR** P)	Операция не использует область-источник. Результат представляет собой инвертированную комбинацию цветов кисти-шаблона и прямоугольника-получателя.
PATPAINT	D=(**NOT** S) **OR** P **OR** D	В операции принимают участие все три ингредиента, объединенные логикой «ИЛИ». Перед началом объединения цвета области-источника инвертируются.

Таблица 17.3 (продолжение)

Константа	Операция	Описание
SRCAND	D=D **AND** S	Комбинация областей получателя и источника, объединенных логической операцией «И».
SRCCOPY	D=S	Копирует исходный прямоугольник в область получатель без изменений.
SRCERASE	D=(**NOT** D) **AND** S	Объединение инвертированного прямоугольника-приемника и прямоугольника-источника на основе операции «И».
SRCINVERT	D=D **XOR** S	Комбинация приемной и исходной областей на основе логики XOR.
SRCPAINT	D=D **AND** S	Объединение приемной и исходной областей логикой «И».
WHITENESS	D=1	Заливает область кистью с индексом 1 в физической палитре. Обычно это белый цвет.

Предсказать результат выполнения тернарной операции особенно сложно при объединении цветных изображений. Подтверждением тому может стать экранный снимок на рис. 17.2 (см. цветную вклейку, с. 419) с десятью вариантами объединения картинки с серым фоном окна.

В качестве примера работы с тернарными операциями предложу небольшую программу, позволяющую разложить любую цветную растровую картинку на три цветовые составляющие. Подобную услугу предоставляют все серьезные графические редакторы, так что и нам стоит испытать свои силы.

На этот раз для упрощения работы наше приложение построим не на основе Windows API, а на базе VCL. Проект содержит две формы – главную `frmMain` и дочернюю `frmChild`. Процесс создания главной формы сопровождается загрузкой растрового образа в память. В этот же момент времени создается совместимый контекст, в который выбирается наш рисунок. Для того чтобы контекст стал доступен для всех дочерних форм приложения, его объявление вынесено в секцию `public`.

```
…
public
    memDC : hDC; //дескриптор совместимого контекста
end;

var  frmMain: TfrmMain;

implementation
uses child; //модуль с описанием дочерних форм
{$R *.dfm}

procedure TfrmMain.FormCreate(Sender: TObject);
var bmp : TBitmap;
begin
bmp:=TBitmap.Create;
bmp.LoadFromFile('..\Graphics\RGBW.bmp');
```

```
memDC:=CreateCompatibleDC(GetDC(frmMain.Handle));
SelectObject(memDC, bmp.Handle);
bmp.free;
end;
```

В библиотеке к книге имеется рисунок RGBW.bmp, на нем изображены четыре круга красного, зеленого, синего и белого цветов на черном фоне. Именно это «произведение искусства» в нашем эксперименте с тернарными растровыми операциями станет играть роль подопытного кролика. В момент вывода главной формы на экран создаем 4 экземпляра дочерних форм. Они отличаются друг от друга только текстом в заголовках и числовым значением в свойстве `tag`.

```
procedure TfrmMain.FormShow(Sender: TObject);
begin
with TfrmChild.Create(frmMain) do //полноцветная картинка
 begin
 Caption:='RGB'; Tag:=0;
 end;
with TfrmChild.Create(frmMain) do //форма для вывода только красного канала
 begin
 Caption:='Red'; Tag:=1;
 end;
with TfrmChild.Create(frmMain) do //форма для вывода только зеленого канала
 begin
 Caption:='Green'; Tag:=2;
 end;
with TfrmChild.Create(frmMain) do //форма для вывода только синего канала
 begin
 Caption:='Blue'; Tag:=3;
 end;
end;
```

В момент закрытия главной формы освобождаем дескриптор в памяти `memDC`.

```
procedure TfrmMain.FormClose(Sender: TObject; var Action: TCloseAction);
begin
DeleteDC(memDC);
end;
```

На этом работу с главной формой проекта мы закончили, настал черед дочерней. Здесь нам понадобится только событие `OnPaint()`, генерируемое всякий раз, когда дочерняя форма нуждается в перерисовке.

```
procedure TfrmChild.FormPaint(Sender: TObject);
var DC:HDC;
    frm:TfrmChild;
    br:HBrush;
begin
frm:=self as TForm;
DC:=GetDC(frm.Handle);
if frm.Tag=0 then  {выводим рисунок без изменений}
```

```
BitBlt (DC, 0, 0, Frm.ClientWidth, Frm.ClientHeight,
        frmMain.memDC, 0, 0, SRCCOPY)
else
begin
   case frm.Tag of
    1 : br:=CreateSolidBrush(clRed);   //красный канал
    2 : br:=CreateSolidBrush(clGreen); //зеленый канал
    3 : br:=CreateSolidBrush(clBlue);  //синий канал
   end;
   SelectObject(DC,br);
   FillRect(DC,frm.ClientRect,br);
   BitBlt (DC, 0, 0, Frm.ClientWidth, Frm.ClientHeight,
           frmMain.memDC, 0, 0, MERGECOPY);
   SelectObject(DC,GetStockObject(NULL_BRUSH));
   DeleteObject(br);
end;
ReleaseDC(frm.Handle,DC);
end;
```

В коде мы анализируем значение в свойстве `tag` каждой из дочерних форм. Если оно нулевое, это признак того, что изображение надо выводить без изменений, в этом случае метод `BitBlt()` вызывается в режиме `SRCCOPY`. В случае, когда в свойстве `tag` дочерней формы обнаруживается значение `1` (мы планируем вывести только красную составляющую), создается красная кисть-шаблон и функция `BitBlt()` вызывается с ключом `MERGECOPY`. Аналогичные действия осуществляются для зеленой (`tag=2`) и синей составляющих (`tag=3`) цвета.

Результат наших трудов отражен на рис. 17.3. В верхнем дочернем окне (с заголовком RGB) отображен исходный битовый образ. Три нижних окна представляют только свои цветовые составляющие – красную, зеленую и синюю.

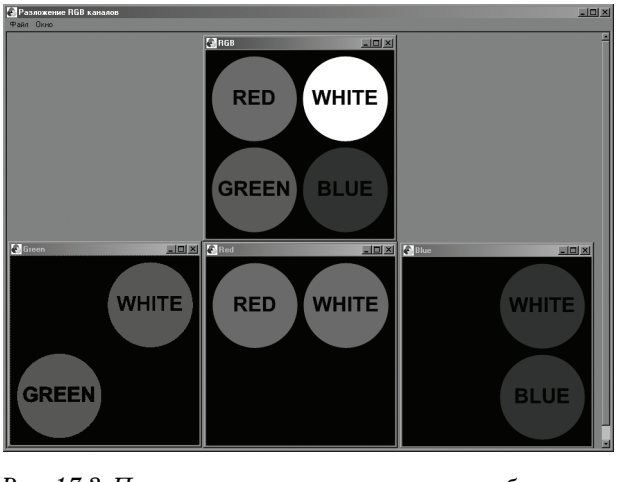


Рис. 17.3. Пример разложения цветного изображения на каналы RGB

Функция StretchBlt()

Как нет в мире совершенства, так и не существует идеальных программных решений. К сожалению, и функция BitBlt() не представляет собой исключения из этого правила. Один из наиболее существенных недостатков нашей новой знакомой проявляется при решении задач, связанных с масштабированием результирующего изображения. Причина кроется в том, что размерности прямоугольной области-получателя и области-источника у BitBlt() одинаковы, ведь они задаются одними и теми же параметрами Width и Height. Услышав такое утверждение, въедливый читатель сразу напомнит мне, что логические размеры протяженности областей одинаковы только в рамках текстового режима отображения (MM_TEXT). Соответственно, переведя исходный или приемный контекст устройства в метрический режим отображения, мы приобретем дополнительные рычаги влияния на пропорции рисунка. Это действительно так, но оставим эти забавы на долю экспериментаторов, а сами познакомимся с наиболее сильным тернарным методом Windows GDI – функцией StretchBlt():

```
Function StretchBlt(DCDest : HDC; X, Y, Width, Height : integer;
                    DCSrc : HDC;
                    XSrc, YSrc, WidthSrc, HeightSrc : integer;
                    ROP : cardinal) : boolean;
```

Функция StretchBlt() имеет различные параметры ширины и высоты для исходной (WidthSrc и HeightSrc) и приемной (Width и Height) прямоугольных областей. Управляя указанными параметрами, мы сможем не просто растягивать или сжимать изображение, но и даже выворачивать его «наизнанку» – вращать вокруг горизонтальной и вертикальной осей.

```
Function FlipVerticalBitmap(DC:HDC; BMP:HBitmap; X,Y:integer):boolean;
var Bitmap:^Windows.TBitmap;
    memDC:hDC;
begin
Result:=false;
if GetObjectType(BMP)=OBJ_BITMAP then
begin
  GetMem(Bitmap,SizeOf(Windows.TBitmap));
  GetObject(BMP,SizeOf(TBitmap),Bitmap);
  memDC:=CreateCompatibleDC(DC);
  SelectObject(memDC,BMP);

  Result:=
  StretchBlt(DC, X+Bitmap^.bmWidth, Y, -Bitmap^.bmWidth, Bitmap^.bmHeight,
             memDC, 0, 0, Bitmap^.bmWidth, Bitmap^.bmHeight, SRCCOPY);

  FreeMem(Bitmap,SizeOf(Windows.TBitmap));
  DeleteDC(memDC);
end;
end;
```

Вполне естественно, что за такие удобства приходится расплачиваться ресурсами процессора и памяти машины. Функция StretchBlt() работает значи-

тельно медленнее своих товарок, это весьма заметно при обслуживании больших растровых блоков, особенно если идет речь не о кратном масштабировании (в разы), а об увеличении или уменьшении размеров, скажем, в полтора раза. Поэтому относитесь к StretchBlt() с должным уважением, так как чтобы порадовать нашего брата программиста, ей приходится изрядно потрудиться, поочередно комбинируя в памяти пикселы двух (или более) столбцов и строк.

Кватернарные операции, функция MaskBlt

В современных версиях Windows на вершину пирамиды растровых операций вознесена функция MaskBlt(). По сути, функция комбинирует две тернарные растровые операции, порядок применения которых определяется монохромной битовой маской.

```
Function MaskBlt(DCDest : HDC; X, Y, Width, Height : integer;
                   DCSrc : HDC; XSrc, YSrc: integer;
                 Mask : HBitmap; XMask, YMask : integer;
                 ROP : cardinal) : boolean;
```

Стартовая восьмерка параметров функции MaskBlt() точно такая же, как и у BitBlt(). Первые пять параметров определяют прямоугольную область на поверхности-получателе операции. За ними следуют еще три аргумента, на этот раз они описывают прямоугольник на поверхности-источнике. Монохромная маска создается в формате аппаратно-зависимого образа DDB, ее дескриптор передается через параметр Mask. В параметрах XMask, YMask указывают горизонтальный и вертикальный отступы маски. Код кватернарной растровой операции складывается из двух кодов тернарных операций. Если пиксел маски равен 1, то применяется *основной* код. Второй код называется *фоновым*, он соответствует нулевому биту маски. Для формирования суммарного кода применяют макрос MAKEROP4:

```
Function MAKEROP4(Fore, Back : DWORD) : DWORD;
```

Макрос объединяет основной и фоновый 24-разрядные тернарные коды в 32-разрядный кватернарный. Параметр Fore описывает основной, а Back – фоновый код.

Прозрачность

Теоретически прозрачное изображение может быть получено с помощью рассмотренной в главе, посвященной растровым операциям, функции MaskBlt(). Однако в Microsoft решили порадовать программистов и разработали группу специализированных функций, предназначенных для реализации эффекта прозрачности. Наиболее популярная из них функция (поддерживаемая почти всеми версиями Windows, за исключением Windows 95 и Windows NT 4.0) называется TransparentBlt().

Win GDI

```
function TransparentBlt(DestDC: HDC;
                        XOriginDest, YOriginDest, WidthDest, HeightDest: Integer;
                        SrcDC: HDC;
                        XOriginSrc, YOriginSrc, WidthSrc, HeightSrc: Integer;
                        Transparent: LongWord): BOOL;
```

Функция осуществляет перенос битового образа из прямоугольника-источника, идентифицируемого дескриптором контекста SrcDC, в прямоугольник-получатель DestDC. Каждый прямоугольник описывается логическими координатами верхнего левого угла (XOriginDest, YOriginDest – получатель, XOriginSrc, YOriginSrc – источник), шириной и высотой (WidthDest, HeightDest – получатель, WidthSrc, HeightSrc – источник) в логических единицах измерения. Последний параметр функции Transparent определяет значение цвета, которое требуется сделать прозрачным. Эффект прозрачности достигается за счет того, что функция копирует в область получателя только пикселы, не совпадающие с цветовым ключом.

Функция TransparentBlt() *встречается во всех 32-разрядных операционных системах Windows, за исключением Windows 95 и Windows NT4.0.*

Ниже предложен исходный код процедуры, демонстрирующий порядок использования функции TransparentBlt(). Основу кода составляет разработанная нами в главе 15 «Аппаратно-зависимые растры» универсальная функция вывода растрового образа, но с двумя существенными отличиями. Во-первых, новая функция выясняет цвет левого верхнего пиксела отображаемой картинки, это значение будет принято в качестве цветового ключа. Во-вторых, вывод изображения осуществляется не за счет функции BitBlt(), а с помощью нашей новой знакомой TransparentBlt().

```
Function DrawTransparentBitmap(DC:HDC; BMP:HBitmap; X,Y:Integer):boolean;
var Bitmap:^Windows.TBitmap;
    memDC:hDC;
    TransparentColor : COLORREF;
begin
Result:=false;
if GetObjectType(BMP)=OBJ_BITMAP then
begin
  GetMem(Bitmap,SizeOf(Windows.TBitmap));
  GetObject(BMP,SizeOf(TBitmap),Bitmap);
  memDC:=CreateCompatibleDC(DC);
  SelectObject(memDC,BMP);
  TransparentColor:=GetPixel(memDC,0,0); //выясняем цвет левого верхнего пиксела
  Result:= TransparentBlt(DC,X,Y, Bitmap^.bmWidth, Bitmap^.bmHeight,
            memDC,0,0,Bitmap^.bmWidth,Bitmap^.bmHeight,TransparentColor);
  FreeMem(Bitmap,SizeOf(Windows.TBitmap));
  DeleteDC(memDC);
end;
end;
```

Методика получения прозрачной картинки за счет изъятия из результирующего изображения обозначенного определенным цветом пиксела не всегда

в состоянии удовлетворить потребности приложения. Вполне вероятно, что пользователь намерен сделать прозрачными не только пикселы определенного цвета, но и другие части растрового изображения. Поэтому в Windows предусмотрен ряд альтернативных способов создания эффекта прозрачности, в частности, с помощью функции:

```
function PlgBlt(DestDC: HDC; const PointsArray;
                SrcDC: HDC; XSrc, YSrc, Width, Height: Integer;
                Mask: HBITMAP; xMask, yMask: Integer): BOOL;
```

Функция управляет прозрачностью не с помощью цветового ключа, а посредством применения задаваемой произвольным образом маски. К дополнительным возможностям надо отнести способность осуществления аффинного преобразования прямоугольного растра в параллелограмм. В растровой операции принимают участие три объекта:

1. Контекст-приемник, идентифицируемый дескриптором `DestDC`.
2. Контекст-источник с дескриптором `ScrDC`.
3. Монохромная битовая маска `Mask`.

Белые пикселы маски (значение 1) соответствуют пикселам, копируемым из источника в приемник, черные пикселы (значение 0) не затрагивают приемник, они и обеспечивают эффект прозрачности.

Два заключительных параметра функции `xMask` и `yMask` определяют смещение маски. Координаты и геометрические размеры прямоугольника-источника описываются в логических единицах измерения в параметрах `XSrc`, `YSrc`, `Width`, `Height`.

С координатами приемной области все обстоит гораздо интереснее. Они передаются через массив `PointsArray` в виде трех точек:

1. Первый элемент массива определяет левую верхнюю точку области, в которую будет отображена левая верхняя точка прямоугольной области из контекста-источника.
2. Второй элемент содержит верхнюю правую точку отображения.
3. Третий элемент – левую нижнюю точку.

Необходимость в определении четвертой точки отсутствует, это объясняется тем, что аффинное преобразование однозначно определяется за счет отображения трех точек одного пространства в три точки другого пространства.

Продемонстрируем возможности функции на практике. В нашем примере мы воспользуемся изображением размерностью 32×32 пиксела и получим его дескриптор `hDIB`. Кроме того, нам потребуется дескриптор аппаратно-зависимого битового образа маски `hMask`.

```
var hDIB,hMask:HBITMAP;
```

Для создания маски опишем битовый массив `Bits`. Это восемь чередующихся белых и черных горизонтальных полос толщиной 4 пиксела каждая.

```
const Bits : Array[0..31] of DWord =
      ($FFFFFFFF,$FFFFFFFF,$FFFFFFFF,$FFFFFFFF,$0,$0,$0,$0,
       $FFFFFFFF,$FFFFFFFF,$FFFFFFFF,$FFFFFFFF,$0,$0,$0,$0,
       $FFFFFFFF,$FFFFFFFF,$FFFFFFFF,$FFFFFFFF,$0,$0,$0,$0,
       $FFFFFFFF,$FFFFFFFF,$FFFFFFFF,$FFFFFFFF,$0,$0,$0,$0);
```

На основе массива `Bits` с помощью функции `CreateBitmap()` мы создаем черно-белый битовый образ маски с размерами 32×32 пиксела. Кроме того, для определения области-получателя нам потребуется массив `PointsArray` с тремя вершинами параллелограмма.

```
Procedure GDI_PAINT(DC:HDC);
var PointsArray:Array[0..2] of TPoint;
    Bitmap:^Windows.TBitmap;
Begin
   hMask:=CreateBitmap(32,32,1,1,@Bits); //создание маски

   memDC:=CreateCompatibleDC(DC); //совместимый контекст
   SelectObject(memDC,hDIB);
   {заполняем точки вершин параллелограмма}
   PointsArray[0].X:=150;  PointsArray[0].Y:=10;
   PointsArray[1].X:=250;  PointsArray[1].Y:=10;
   PointsArray[2].X:=200;  PointsArray[2].Y:=110;

   PlgBlt(DC,PointsArray,MemDC,0,0,32,32,hMask,0,0); //вывод образа
   DeleteObject(hMask); DeleteDC(memDC);
end;
```

На рис. 17.4 представлен экранный снимок окна получившейся программы, правда, в этом случае использовался несколько усложненный исходный код. В нем для наглядности отдельно выведены изображения: маски, растрового образа цветка, прозрачного образа без искажения размеров. Последняя картинка соответствует нашему примеру – отображенному в параллелограмм рисунку с прозрачными полосами.

Функция `PlgBlt()` *не поддерживается в операционных системах Windows 95/98/Me.*

Рис. 17.4. Демонстрация возможностей PltBlt( )

РЕЗЮМЕ

При определении цвета пиксела изображения графическая система Windows вправе использовать несколько разновидностей растровых операций. Самая элементарная из них – бинарная. Результирующий цвет пиксела, к которому применена такая растровая операция, формируется на основе логического объединения двух элементов: пиксела-получателя и пера. Порядок объединения определяется функцией `SetROP2()`. Вторая по степени сложности растровая операция именуется тернарной. Такое название обусловлено тем, что в подобной операции задействованы три участника: пиксел-получатель, кисть (шаблон) и пиксел-источник. Реализацией тернарных операций на практике занимаются функции `PatBlt()`, `BitBlt()` и `StretchBlt()`. Однако возможности этого трио меркнут перед их наиболее продвинутой коллегой `MaskBlt()`. Функция `MaskBlt` способна осуществить кватернарную растровую операцию, итоги которой зависят от квартета ингредиентов: пикселов источника и получателя, шаблона и монохромной маски.

18

Расширенный формат метафайла, EMF

В тех случаях, когда рисунок может быть представлен в виде набора графических примитивов (линий, кривых, эллипсов, прямоугольников и т. п.), растровая графика уступает место векторным изображениям. На основе векторной графики построены современные программы промышленного черчения (например, Auto Cad) и графические пакеты, к числу которых относятся такие гранды, как Corel Draw и Microsoft Visio.

А чем так плохи битовые образы? Неужели они такие неудачники, что не справятся с описанием чертежа карданного вала? Ведь всем известно, что растровые рисунки обладают рядом неоспоримых достоинств, в числе которых высокая точность передачи графического образа, хорошая скорость вывода, поддержка растровых операций и сравнительная простота хранения данных. Все это действительно так, однако идеальных вещей на свете не существует... Достоинства растровой графики меркнут, когда на первый план выступают вопросы масштабирования и точного измерения расстояний, кроме того, при хранении изображения DIB и DDB предъявляют повышенные требования к объему памяти (в особенности, если они представлены в 32-битном цвете). Но ключевой недостаток битового образа даже не в этом. Растровый рисунок весьма неудобен для редактирования. Инженер-конструктор нуждается в таком способе описания изображения, воспользовавшись которым он сможет создавать чертеж из набора шаблонов с возможностью легкой реконфигурации изображения (переноса, поворота объектов, изменения их размеров, настройки единиц измерения и т. п.). А это как раз то, чем не в состоянии похвастаться массив раскрашенных пикселов. Поэтому там, где не справляется растровый рисунок, Windows старается воспользоваться услугами векторной графики.

Векторная графика в современной Windows зиждется на понятии расширенного *метафайла* (*metafile*) – коллекции структур, способной представлять рисунок в формате, независящем от устройства. Метафайл хранит не массив окрашенных пикселов, а множество обращений к функциям GDI. В момент вывода изображения на поверхности устройства метафайл последовательно

вызывает все описанные в нем функции, поэтому часто используют выражение «воспроизведение метафайла».

С точки зрения Windows GDI метафайл – это такой же полноценный объект, как перо, кисть, шрифт или недавно изученные нами растры DDB, DIB. По сложившейся традиции доступ к метафайлу осуществляется посредством обращения к его дескриптору `HMETAFILE` или `HENHMETAFILE`. Наличие двух типов дескрипторов метафайла объясняется тем, что в GDI существует два вида подобных объектов: *метафайл* (*Windows Metafile, WMF*) и *метафайл с расширенными возможностями* (*Enhanced Metafile, EMF*). Метафайл WMF появился на свет в 1985 году вместе с Windows 1.0 и к сегодняшнему дню морально устарел. Ключевое различие между форматами WMF и EMF в том, что новая версия метафайла не зависит от устройства и способна использовать современные команды GDI. Поэтому в этой главе мы сосредоточим свое внимание на современном формате векторной графики – метафайле с расширенными возможностями EMF.

В GDI+ поддерживается обновленный формат метафайла с расширенными возможностями, называемый EMF+. Указанный метафайл воспринимает записи в формате команд GDI+.

Структура метафайла

Итак, мы уже выяснили, что метафайл представляет собой набор записей, каждая из которых содержит описание вызова какого-то графического примитива GDI. Но каким образом осуществляется хранение этой информации? Ведь функции и макросы GDI обладают различными названиями, перечнем параметров и порядком вызова! Ответ на этот вопрос считается краеугольным камнем векторной графики Windows, поэтому изучение метафайла мы начнем с уяснения идеи хранения данных в EMF.

Каждое обращение к функции GDI в метафайле представляется в виде отдельной записи. Запись расширенного метафайла (далее мы ее станем называть *записью EMR*, *enhanced metafile record*) включает базовую и дополнительную часть. Базовая часть у всех записей `EMR` одинакова и состоит из двух четырехбайтных полей (рис. 18.1). Структура и размеры дополнительной части зависят от вызываемой функции GDI.

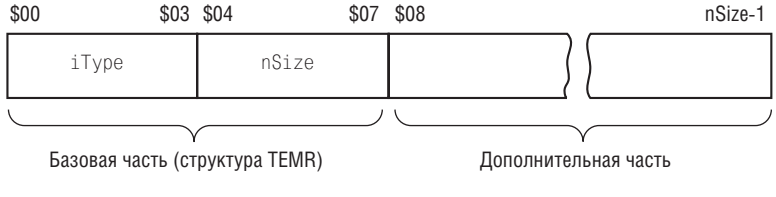


Рис. 18.1. Обобщенное представление записи EMR

В Delphi базовая часть записи расширенного метафайла описывается структурой:

```
type TEMR = packed record
    iType: DWORD; //тип записи
    nSize: DWORD; //размер всей записи (базовой и дополнительной частей)
end;
```

В поле iType структуры хранится константа типа записи EMR. Название константы начинается со слова «EMR_». В файле windows.pas Delphi 2007 объявлено 120 таких констант (начиная от EMR_MIN=1 и заканчивая EMR_MAX=120). Покопавшись в заголовочном файле wingdi.h из состава Microsoft Visual Studio 2005, можно обнаружить еще парочку: EMR_COLORMATCHTOTARGETW (значение 121) и EMR_CREATECOLORSPACEW (значение 122).

Зачастую с выходом в свет очередной операционной системы количество типов записей EMF увеличивается. Например, в Windows NT 4.0 перечень констант заканчивался значением 104, а 120 их стало в Windows 2000.

Мы не станем перечислять все существующие на сегодня константы (если потребуется, то вы их найдете в SDK), но запомните, что обычно их имена связаны с названиями соответствующих функций GDI. Допустим, что в исследуемой нами записи осуществляется вызов функции Windows GDI, рисующей эллипс Ellipse(). Это означает, что в поле iType окажется константа EMR_ELLIPSE. Получив такую информацию, GDI проявит сообразительность и вспомнит, что для вывода эллипса необходимо знать координаты левого верхнего и правого нижнего углов ограничивающего прямоугольника, для чего потребуется четыре значения типа Integer. Такие расчеты и станут основанием для распределения дополнительной части записи EMR. Для большего удобства в GDI объявлено несколько десятков структур, упрощающих заполнение второй части записи EMR. В нашем примере константе EMR_ELLIPSE соответствует структура:

```
type tagEMRELLIPSE = packed record
    emr: TEMR;      {базовая часть}
    rclBox: TRect;  {дополнительная часть,
                     в данном случае ограничивающий прямоугольник}
  end;
```

Обратите внимание на то, что структура tagEMRELLIPSE включает поле emr, а это не что иное, как базовая часть записи TEMF и поле, хранящее координаты ограничивающего прямоугольника, – как раз то, что необходимо для работы функции Ellipse().

В самом общем виде базовая и дополнительная части записи EMR могут быть представлены универсальной структурой:

```
type TEnhMetaRecord = packed record
    iType: DWORD;    //тип записи
    nSize: DWORD;    //размер
    dParm: array[0..0] of DWORD; //дополнительные данные
end;
```

Это и есть запись метафайла. Универсальность структуры достигается за счет безразмерного поля параметров dParam, представляющего собой динамический массив двойных слов. В этот массив могут быть переданы аргументы вызываемой функции, например координаты ограничивающего прямоугольника, по которым следует прочертить эллипс.

Запись EMR *не что иное, как надстройка над базовой структурой* TEMR. *Помимо константы со значением типа записи и размера структуры в ней хранятся дополнительные данные, например параметры функции.*

Из-за того, что функции GDI содержат различное количество разнотипных параметров, структуры записей EMR отличаются друг от друга. Поэтому второе поле структуры TEMR (поле nSize) хранит суммарный размер базовой и дополнительных частей записей EMR, это позволяет операционной системе быстрее «соображать» при переходе от одной записи к другой. Ко всему сказанному добавлю, что запись EMR выравнивается по границе двойного слова.

Трейлер и палитра метафайла

Объект метафайла содержит не только коллекцию записей EMR. Его финальные байты отведены для заключительной записи метафайла – трейлера, типизированного записью TEMREOF. Завершающая структура информирует об окончании метафайла и предоставляет информацию об элементах палитры метафайла.

```
type TEMREOF = packed record
 emr: TEMR;              //базовая структура
 nPalEntries: DWORD;     //число элементов в палитре
 offPalEntries: DWORD;   //цветовая палитра
 nSizeLast: DWORD;       //размер последней записи EMF
end;
```

Нам нет смысла заботиться о создании палитры, это функциональные обязанности GDI. В момент записи расширенного метафайла графическое ядро самостоятельно анализирует применяемые в рисунке цвета и «складирует» их в палитре. Если в метафайле используется цветовая палитра, то доступ к ней реализуется посредством обращения к функции:

Win GDI

```
function GetEnhMetaFilePaletteEntries(hEMF: HENHMETAFILE;
                                      CountEntries: UINT; pPE: Pointer): UINT;
```

На вход функции подается дескриптор метафайла и количество элементов в палитре. Указатель на палитру возвращается через выходной параметр pPE, это знакомый нам по главе 14 «Цветовые палитры» массив записей TPaletteEntry. Кроме того, функция проинформирует нас об общем числе элементов в палитре цветов. Добравшись до палитры метафайла, мы сможем создать логическую палитру и реализовать ее в нашем приложении с помощью функций CreatePalette(), SelectPalette() и RealizePalette().

Для того чтобы вывод на экран метафайла, построенного на основе палитры, не приводил к смене текущих цветов для всех приложений, логическая палит-

ра EMF выбирается только в качестве фоновой. Напомню, что фоновая палитра не имеет права на удаление используемых цветов из системной палитры, ей разрешено лишь использовать незадействованные ячейки таблицы цветов.

Формат заголовка файла EMF

Для того чтобы расширенный метафайл смог превратиться в файл, к нему подключается увесистый заголовок, типизированный записью TEnhMetaHeader. Рассмотрим структуру заголовка наиболее распространенной четвертой версии расширенного метафайла. Почти в двух десятках полей содержатся данные, необходимые для того, чтобы GDI сумел без искажений воспроизвести расширенный метафайл на поверхности любого графического устройства.

```
type TEnhMetaHeader = packed record
 iType: DWORD;          //всегда 1 - константа EMR_HEADER
 nSize: DWORD;          //размер структуры в байтах
 rclBounds: TRect;      //координаты ограничивающего прямоугольника
                        //в единицах устройства
 rclFrame: TRect;       //координаты ограничивающего прямоугольника,
                        //ед.измерения 0.01 мм.
 dSignature: DWORD;     //константа ENHMETA_SIGNATURE (сигнатура «EMF»)
 nVersion: DWORD;       //номер версии метафайла
 nBytes: DWORD;         //размер метафайла в байтах
 nRecords: DWORD;       //число записей EMF в составе метафайла
 nHandles: Word;        //количество указателей в таблице
                        //дескрипторов метафайла THandleTable
 sReserved: Word;       //резерв, должно быть равным 0
 nDescription: DWORD;   //число символов Unicode в текстовом описании метафайла
 offDescription: DWORD; //относительное смещение строки от начала заголовка
 nPalEntries: DWORD;    //количество элементов в палитре метафайла
 szlDevice: TSize;      //размеры поверхности устройства в пикселах
 szlMillimeters: TSize; //размеры поверхности устройства в миллиметрах
 cbPixelFormat: DWORD;  //размер структуры TPixelFormatDescriptor
 offPixelFormat: DWORD; //смещение TPixelFormatDescriptor
 bOpenGL: DWORD;        //признак наличия команд OpenGL
end;
```

Следующая по счету версия заголовка файла EMF имеет индекс 5.0. Это полная копия TEnhMetaHeader плюс одно 8-байтное дополнительное поле szMicrometers : TSize. Поле описано в самом конце структуры и хранит размер поверхности эталонного устройства в микрометрах.

С момента появления пятой версии заголовка прошло немало времени (выход в свет Windows 2000), но специалисты Borland до сих пор не спешат переписывать модуль windows.pas. Среда Delphi 2007 по-прежнему ориентируется на EMF 4.0. Такая нерасторопность программистов, скорее всего, объясняется тем, что szMicrometers *по сути дублирует данные из поля* szlMillimeters*, так что необходимость в нововведении не столь очевидна. Но есть и обратная сторона медали, про которую стоит помнить при работе с файлами EMF. Заголовок метафайла версии 4.0 составляет 100 байт, а заголовок 5.0 – 108 байт. Необходимо учитывать это различие при чтении данных из файла.*

Для чтения заголовка из файла в GDI создана специальная функция:

```
function GetEnhMetaFileHeader(hEMF: HENHMETAFILE; BufSize: UINT;
                              pEMH: PEnhMetaHeader): UINT;
```

В составе функции три параметра: `hEMF` – дескриптор метафайла, получаемый с помощью функции `GetEnhMetaFile()`; `BufSize` – размер буфера, в который помещается полученная информация; `pEMH` – указатель на структуру `TEnhMetaHeader`. Согласно рекомендациям SDK функция должна вызываться дважды. В момент первого обращения в третий параметр `GetEnhMetaFileHeader()` передается неопределенный указатель `nil`. В ответ функция возвратит размер требуемого приемного буфера. Полученное значение понадобится для инициализации буфера при повторном обращении к функции.

В следующих строках кода продемонстрирован порядок «выуживания» из заголовка метафайла координат ограничивающего прямоугольника.

```
Function GetEMFBounds (hEMF: HENHMETAFILE):TRect;
var BufSize: UINT;
    pEMH:PEnhMetaHeader;
begin
  BufSize:=GetEnhMetaFileHeader(hEMF, BufSize, nil); //размер структуры
  GetMem(pEMH,BufSize);                              //распределяем память
  GetEnhMetaFileHeader(hEMF, BufSize, pEMH);         //считываем данные
  Result:=pEMH^.rclBounds;                           //заполняем результат
  FreeMem(pEMH);                                     //освобождаем память
end;
```

С подобной задачей мы столкнемся в ситуации, когда нам будет необходимо вывести рисунок на экран с соблюдением его пропорций.

Загрузка метафайла из файла

Для обслуживания хранящегося на диске компьютера файла EMF потребуется помощь трех функций, каждая из которых нацелена на решение отдельной задачи: загрузки, воспроизведения и удаления метафайла из памяти.

Для получения дескриптора метафайла обращаемся к функции:

```
function GetEnhMetaFile(FileName: PChar): HENHMETAFILE;
```

Функция считывает файл с именем `FileName`, на его основе создает в памяти объект расширенного метафайла и возвращает программисту его дескриптор. Согласитесь, что в сравнении с растровыми образами DIB, для чтения которых нам пришлось самостоятельно разработать сложную функцию, здесь все значительно проще. Обладая дескриптором метафайла, мы сможем его воспроизвести на поверхности графического устройства, для этого вызываем функцию:

```
function PlayEnhMetaFile(DC: HDC; hEMF: HENHMETAFILE; const RCT: TRect):
BOOL;
```

Здесь `DC` – дескриптор контекста устройства, на котором мы планируем воспроизвести содержимое метафайла; `hEMF` – дескриптор метафайла; `RCT` – пря-

моугольная область (в логических единицах измерения), в границах которой планируется воспроизвести метафайл.

Ниже предложен листинг, демонстрирующий порядок применения перечисленных функций.

```
Procedure GDI_PAINT(DC:HDC);
var hEMF:HENHMETAFILE;
    Rct:TRect;
begin
 hEMF:=GetEnhMetaFile('..\Graphics\balloon.emf');
 SetRect(Rct,0,0,200,300);
 PlayEnhMetaFile(DC,hEMF,Rct);
 DeleteEnhMetaFile(hEMF);
end;
```

В представленном фрагменте кода встречается еще одна новая функция:


```
function DeleteEnhMetaFile(hEMF: HENHMETAFILE): BOOL;
```

Она необходима для освобождения ресурса метафайла с дескриптором `hEMF`. Функция ни в коем случае не уничтожает метафайл, сохраненный на диске, но если дескриптор `hEMF` ссылается на метафайловый объект, расположенный в памяти компьютера, то от него не останется и следа.

Создание расширенного метафайла

Мы уже умеем главное – загружать метафайл из файла и воспроизводить его на поверхности графического устройства. В 99% случаев этих знаний вполне достаточно для снабжения наших проектов симпатичными масштабируемыми рисунками, но их не хватит для полноценного понимания возможностей Windows по работе с векторной графикой. Поэтому нам стоит разобраться с процессом программного создания экземпляра расширенного метафайла.

Генерация расширенного метафайла начинается с обращения к функции:

```
function CreateEnhMetaFile(DC: HDC; FileName: PChar;
                           Rect: PRect; Desc: PChar): HDC;
```

Это еще не создание объекта EMF, это подготовительная операция, формирующая *метафайловый контекст*, который станет использоваться для записи метафайла. Если вы ненадолго вернетесь к главе 2 (посвященной контексту графического устройства), то обязательно вспомните, что это пятый по счету тип контекста в GDI (наряду с контекстом дисплея, контекстом печатающего устройства, информационным контекстом и контекстом в памяти). Для того чтобы функция `CreateEnhMetaFile()` возвратила дескриптор метафайлового контекста, она должна получить четыре параметра:

1. `DC` – дескриптор контекста, его настройки берутся за образец;
2. `FileName` – имя файла, в который сохранится метафайл. Параметр необязателен для заполнения, и, если оставить его неопределенным, то метафайл формируется в памяти;

3. Rect – прямоугольные размеры метафайла, в качестве единицы измерения используются сотые доли миллиметра (0,01 мм). Для Microsoft это весьма серьезное самопожертвование (в США не очень жалуют метрическую систему, предпочитая работать с дюймами). Прямоугольные размеры метафайла допускается не определять, в этом случае GDI проанализирует координаты всех функций, вызываемых в метафайле, и самостоятельно вычислит ограничивающий прямоугольник. Как понимаете, его размеры будут минимальны – лишь бы вместить рисунок.
4. Desc – текстовая строка с описанием метафайла. Порядок формирования строки с комментарием требует отдельного пояснения. Предполагается, что описание состоит из двух частей: названия приложения, создавшего EMF, и названия рисунка. Текстовые блоки должны разделяться нулевым символом «#0», кроме того, все описание следует завершать двумя нулевыми символами.

Как только в ваших руках окажется дескриптор метафайлового контекста, включайте фантазию и рисуйте (используя функции GDI) все, что хочется. Графическая подсистема Windows сохранит все обращения к функциям в коллекции записей EMF будущего объекта-метафайла. Завершив художественные изыскания, обязательно отпустите метафайловый контекст на волю, он нам больше не понадобится. Роль «освободителя» контекста возложена на функцию:

```
function CloseEnhMetaFile(DC: HDC): HENHMETAFILE;
```

Единственный параметр – дескриптор метафайлового контекста (который мы получили стараниями функции CreateEnhMetaFile). Функция не просто освобождает выделенный ресурс, она возвратит дескриптор созданного метафайлового объекта. С этого момента расширенный метафайл можно воспроизводить, копировать, считывать заголовок, исследовать записи EMF, короче говоря, делать все, что душе захочется (естественно, в рамках возможностей GDI).

Для закрепления теории рассмотрим небольшой отрывок кода. Обращение к функции CreateMETAFILE() приводит к созданию метафайла new.emf.

```
Function CreateMETAFILE:HENHMETAFILE;
var emfDC:HDC;
begin
emfDC:=CreateEnhMetaFile(GetDC(0),'new.emf',nil,'Test'+#0+#0);
  SelectObject(emfDC, GetStockObject(DC_PEN));
  Rectangle(emfDC,0,0,50,50);
  SetDCPenColor(emfDC,$FF0000);
  Ellipse(emfDC,10,10,60,60);
  TextOut(emfDC,0,0,pChar('Метафайл'),8);
Result:=CloseEnhMetaFile(emfDC);
end;
```

Процесс создания метафайла немножко напоминает механизм описания траектории. Вспоминайте, траектория формируется внутри конструкции

BeginPath() и EndPath(). Некоторая аналогия наблюдается и здесь – все вызовы функций GDI заключены в блок между обращениями к функции CreateEnhMetaFile() и CloseEnhMetaFile(). Полученный дескриптор метафайла может быть передан функции PlayEnhMetaFile() с целью отображения сформированного в коде программы векторного рисунка на поверхности контекста графического устройства.

Размещение метафайла в памяти

Скорость доступа к ячейкам памяти микросхем ОЗУ в сотни раз превосходит операции чтения данных с самых ультрасовременных жестких дисков. Поэтому вся серьезная обработка данных, в том числе и векторной графики, осуществляется в оперативной памяти. Удобный способ размещения метафайла в памяти предоставляет функция:

```
function GetEnhMetaFileBits(hEMF: HENHMETAFILE;
                            BuffSize: UINT; pBuff: PByte): UINT;
```

Функция снабжена тремя параметрами: дескриптором расширенного метафайла hEMF; параметром BuffSize, определяющим размер буфера, в который будет направлена коллекция записей, и указателем на этот буфер pBuff. В момент первого обращения к функции в завершающий параметр передается неопределенный указатель nil, в этом случае GetEnhMetaFileBits() возвратит размер коллекции в байтах. Полученное значение помещается в параметр BuffSize при втором вызове функции. При успешном завершении работы функция возвратит указатель на буфер через свой третий аргумент pBuff. Кроме того, функция проинформирует нас об общем размере метафайла в байтах.

Для обратной операции изъятия данных из памяти предназначена функция:

```
function SetEnhMetaFileBits(BuffSize: UINT; pBuff: PChar): HENHMETAFILE;
```

Для того чтобы функция смогла вернуть жизнеспособный дескриптор размещенного в памяти метафайла, ей требуется получить два аргумента: указатель на буфер Buff и его размер BuffSize. Если запрошенные данные окажутся некорректными, то вместо дескриптора функция позабавит незадачливого программиста неопределенным указателем.

Копирование метафайла

Обладая дескриптором расширенного метафайла, мы с легкостью сможем скопировать EMF в файл либо в память компьютера. Для этих целей разработана функция:

```
function CopyEnhMetaFile(hEMF: HENHMETAFILE;
                         FileName: PChar): HENHMETAFILE;
```

Второй параметр функции указывает на строку с названием файла, если вместо имени передать nil, копия метафайла формируется в памяти. Функция возвратит дескриптор вновь созданного объекта.

Построчное воспроизведение метафайла

Для получения возможности вносить в векторный рисунок исправления необходимо научиться расшифровывать коллекцию записей EMF. Для этого реализована функция, способная построчно просмотреть все записи из коллекции EMF:

```
function EnumEnhMetaFile(DC: HDC; hEMF: HENHMETAFILE;
                         EnhMetaFunc: TFNEnhMFEnumProc; Data: Pointer;
                         const Rect: TRect): BOOL;
```

Для обеспечения работы `EnumEnhMetaFile()` в ее адрес необходимо передать два дескриптора: дескриптор контекста устройства `DC` и дескриптор метафайла `hEMF`. Третий по счету параметр ссылается на функцию обратного вызова `EnhMetaFileProc()`, прототип которой представлен ниже по тексту. Затем следует указатель на дополнительные данные `Data`, его допускается оставлять неопределенным. Вереницу параметров завершает аргумент `Rect`, определяющий координаты ограничивающего прямоугольника, в пределах которого станет воспроизводиться метафайл. Функция последовательно перебирает все записи EMF до тех пор, пока не будет достигнута последняя запись, или пока функция обратного вызова не возвратит ноль. В случае успешного выполнения `EnumEnhMetaFile()` возвращает значение `true`.

В паре с `EnumEnhMetaFile()` всегда трудится функция обратного вызова, ее назначение – обслуживание отдельной записи метафайла. Прототип функции выглядит следующим образом:

```
Function EnhMetaFileProc (DC : HDC;  pHTable : pHandleTable;
                          const pEMFR : pEnhMetaRecord;  CountObj : Integer;
                          Data : LPARAM ) : integer; stdcall;
```

Помимо дескриптора контекста, функция обратного вызова интересуется указателем на таблицу `pHTable`, хранящую список дескрипторов объектов GDI (палитра, кисти, перья и шрифты), участвующих в создании рисунка метафайла. В `CountObj` передается размер этой таблицы. Параметр `pEMFR` содержит указатель на текущую запись EMF. Последний параметр `Data` позволяет получить дополнительные данные от головной функции `EnumEnhMetaFile()`. Заполнение всех перечисленных параметров не наша забота, они обслуживаются автоматически силами опорной функции `EnumEnhMetaFile()`.

В процессе описания метафайла применяются различные перья, кисти и палитры. Для того чтобы корректно воспроизвести рисунок Windows, приходится «выуживать» описание объектов GDI из коллекции записей EMF. Поэтому перед началом воспроизведения векторного рисунка GDI создает таблицу, в которую собираются дескрипторы всех задействованных в метафайле перьев, кистей, шрифтов и палитр. В Delphi эта таблица описана структурой `THandleTable`.

Совместно с функцией обратного вызова `EnhMetaFunc()` часто используется функция:

Win GDI

```
function PlayEnhMetaFileRecord(DC: HDC;
                               var HTable: THandleTable;
                               const EMFR: TEnhMetaRecord;
                               CountObj: UINT): BOOL;
```

Она позволяет выводить метафайл в контекст устройства DC построчно (в соответствии с порядком следования записей EMF). Не станем повторяться, перечисляя параметры функции, они почти идентичны аргументам функции косвенного вызова. Единственная разница в том, что вместо указателя на таблицу заголовков и указателя на запись EMF передаются их явные значения.

Вставляя паузы между обращениями к функции построчного воспроизведения PlayEnhMetaFileRecord *(например, с помощью функции* Sleep*), можно добиться эффекта замедленного вывода метафайла.*

Разработаем пример, позволяющий увидеть работу всех трех функций на практике. Начнем с оконной процедуры. В ее рамках в момент получения сообщения WM_CREATE мы создаем кнопку и список, обрабатываем щелчок по кнопке и обслуживаем сообщение прорисовки окна.

```
function WindowProc (Window: HWND; Msg: UINT;
                     wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
CASE Msg OF
   WM_CREATE : {создание кнопки и списка}
   begin
    Btn:= CreateWindow( 'BUTTON', 'Загрузить',
          WS_CHILD OR WS_BORDER OR WS_VISIBLE OR BS_PUSHBUTTON OR BS_LEFT,
          10, 10, 84, 30, Window, 0, HInstance, nil);

    LISTBOX:= CreateWindow('LISTBOX', '',
          WS_CHILD OR WS_BORDER OR WS_VISIBLE OR BS_LEFT
          OR (LBS_STANDARD - LBS_SORT),
          50, 17, 100, 30, Window, 0, HInstance, nil);
   end;

   WM_COMMAND  :
   {Щелчок по кнопке Button - загрузка файла EMF}
     if lParam=Button then
        begin
         if hEMF>0 then DeleteEnhMetaFile(hEMF);
         hEMF:=GetEnhMetaFile('balloon1.emf');
         //очистка строк списка
         while SendMessage(LISTBOX,LB_GETCOUNT,0,0)>0 do
               SendMessage(LISTBOX,LB_DELETESTRING,0,0);
           EMFBounds:=GetEMFBounds(hEMF); //размеры метафайла
           InvalidateRect(wnd,@EMFBounds,true);
        end;

   WM_PAINT   :
   BEGIN
     DC:=BeginPaint(Window,PAINTSTRUCT);
```

```
      if hEMF>0 then EnumEnhMetaFile(DC,hEMF,@EnhMetaFileProc, nil, EMFBounds);
      EndPaint(Window,PAINTSTRUCT);
    END;
//… обработка других сообщений
```

Нам осталось описать функцию обратного вызова. В ее коде мы сделаем две вещи: проведем анализ расширенного метафайла на предмет используемых в нем графических примитивов и выведем рисунок на поверхность графического устройства.

```
Function EnhMetaFileProc(DC : HDC;  pHTable : pHandleTable;
                         const pEMFR : pEnhMetaRecord; CountObj : Integer;
                         Data : LPARAM ) : integer; stdcall;
var s :string;
BEGIN
case pEMFR^.iType of
{конвертируем значение константы pEMFR^.iType в ее название 'EMR_...'}
  1   : s:='EMR_HEADER';
  2   : s:='EMR_POLYBEZIER';
  3   : s:='EMR_POLYGON';
  4   : s:='EMR_POLYLINE';
  //…другие константы
  119 : s:='EMR_SETLINKEDUFIS';
  120 : s:='EMR_SETTEXTJUSTIFICATION';
  else Result:='Unknown value';
end;
{передаем текст в список LISTBOX}
SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(S+#0)));
//вывод записи EMF на поверхности графического устройства
PlayEnhMetaFileRecord(DC,pHTable^,pEMFR^,CountObj);
END;
```

На рис. 18.2 представлены результаты нашего титанического труда, мы разложили изображение воздушного шара на составные части.

Современному компьютеру не составит особого труда за одно мгновение вывести на экран векторный рисунок, состоящий из сотен графических примитивов; для того чтобы подчеркнуть эффект построчного вывода, можете воспользоваться функцией `Sleep()`, позволяющей приостанавливать выполнение кода.

В отличие от битового образа, время вывода которого определяется глубиной цвета и геометрическими размерами рисунка, время отображения метафайла определяется количеством содержащихся в нем команд.

Функция обратного вызова способна и на более существенные подвиги, чем просто воспроизведение рисунка. Она позволяет управлять процессом прорисовки изображения, например отменить вывод каких-то графических примитивов или настроить координаты вывода. Но основное применение технологии построчного вызова метафайла связано с работой профессиональных графических редакторов, таким образом они производят декомпо-

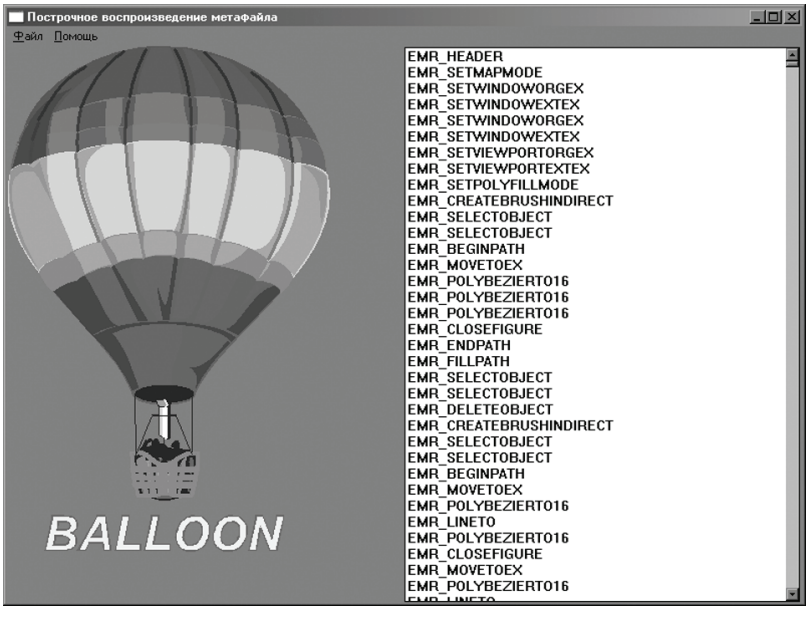


Рис. 18.2. Состав записей метафайла

зицию векторного рисунка на составные объекты, за которые пользователь уже сможет «ухватиться» указателем мыши и переместить на новое место, удалить, изменить форму и т. п., одним словом, отредактировать.

Передаваемая в адрес функции обратного вызова запись EMF не позволит редактировать свои поля, так как она доступна только для чтения.

Комментарий к метафайлу

Изучая заголовок метафайла `TEnhMetaHeader`, а затем ведя речь о процессе создания расширенного метафайла, мы упоминали, что файл EMF может быть снабжен пояснительным текстом. По задумке разработчиков строка комментария состоит из двух частей. Первая часть описания отражает название программы, в которой создавалось изображение, а вторая – имя рисунка. Но в составе объекта метафайла обе части комментария хранятся в виде единой текстовой строки, поэтому для расчленения комментария к метафайлу на две части приходится применять символ разделитель «#0». Кроме того, вся строка комментария должна завершаться парой точно таких же нулевых символов. Если все правила соблюдены, вы без труда прочитаете комментарий к метафайлу с помощью функции:

```
function GetEnhMetaFileDescription(hEMF: HENHMETAFILE;
                                   BuffSize: UINT; Buff: PChar): UINT;
```

Кроме дескриптора метафайла hEMF на вход функции необходимо подать размер приемного буфера BuffSize. А результат мы получим через указатель на текстовую строку Buff.

Для вставки комментария в метафайл используется функция:

```
function GDIComment(DC: HDC; BuffSize: UINT; Buff: PChar): BOOL;
```

Здесь DC – дескриптор метафайла, BuffSize – размер буфера с новым комментарием и Buff – указатель на массив с описанием.

Метафайл может быть снабжен комментарием в формате Unicode, в таком случае для чтения описания файла необходимо использовать функцию GetEnhMetaFileDescriptionW().

РЕЗЮМЕ

В современных версиях Windows основным способом представления векторной графики считается расширенный метафайл EMF. Это идеальное средство для описания изображений, подготовленных в виде графических примитивов (линий, кривых, прямоугольников и т. п.). Физически метафайл представляет собой специализированное хранилище вызовов функций GDI. Последовательное обращение к этим функциям воспроизводит рисунок на поверхности графического устройства. Метафайл может быть легко сохранен в памяти компьютера или в виде файла. Еще одна немаловажная положительная черта расширенного метафайла в том, что он независим от аппаратной части.

19

Шрифты

Все специалисты по разработке программного обеспечения в один голос утверждают, что в графическом программировании работа со шрифтами – это одна из наиболее сложных областей. Термин «шрифт» появился примерно в середине XV века вместе с печатным станком Гуттенберга, использующим для воспроизведения текста типографские литеры. Наборщик собирал свинцовые литеры в матрицу, оттиск которой создавал газетную полосу или страницу книги. В печатном деле свинцовые матрицы широко применялись до второй половины XX столетия, ровно до тех пор, пока не были разработаны первые программные продукты электронной верстки, послужившие толчком к революционным изменениям в деле книгоиздания.

Несмотря на то, что от появления на свет первых шрифтов до наших дней прошло более половины тысячелетия, с тех времен понятие «шрифт» не претерпело существенных изменений. Как с точки зрения первопечатника, так и по мнению современного программиста шрифт – это совокупность изображений символов (глифов, от англ. *glyph*) одного размера (кегля), обладающих однотипным графическим стилем (гарнитурой). Но зато как сильно изменилось материальное воплощение шрифта – от грубых свинцовых шаблонов до изящных математических описаний!

На настоящий момент существует несколько способов программного представления изображений шрифта:

- Растровый.
- Векторный.
- Контурный (TrueType и OpenType).

С точки зрения истории программирования самыми первыми на свет появились *растровые шрифты*. Глиф символа такого шрифта представляется в виде обычного битового образа. Как следствие, растровые шрифты практически не масштабируются, при увеличении символ превращается в угловатое чудовище (рис. 19.1), способное в два счета отпугнуть от компьютера слабо-

Рис. 19.1. Растровый шрифт MS Serif

нервного пользователя. А с задачами уменьшения растрового шрифта GDI вообще не связывается – текст теряет разборчивость. Из-за капризов с масштабированием в ранних версиях Windows приходилось содержать несколько ресурсов растрового шрифта одного и того же начертания, каждый ресурс предназначался для работы только со своим разрешением и кеглем. Вместе с тем, несмотря на изрядную древность, растровые шрифты до сих пор применяются в супермодных версиях Windows. Причина кроется в высокой скорости вывода битового образа на экран. Поэтому подписи на кнопках, названия пунктов меню в подавляющем большинстве современных приложений обычно рисуются растровым шрифтом.

Векторный шрифт устраняет основной недостаток растрового шрифта – проблему с масштабированием. Хороший результат достигается за счет принципиально другого способа воссоздания внешнего вида символа. В векторном шрифте глиф представляет собой не битовый образ, а математическое описание изображения, выводимого с помощью пера в виде последовательности отрезков (за счет этого векторные шрифты находят широкое применение в плоттерах). Для изменения размера достаточно управлять коэффициентами умножения. С увеличением размера символов система пропорционально увеличивает толщину пера, таким образом достигается приемлемое качество изображения (рис. 19.2). Однако векторные шрифты также не стали образцом совершенства, так как при значительном увеличении символа такого шрифта становится заметна прямолинейность его начертания.

Своим рождением *контурные шрифты TrueType* обязаны компании Apple, разрабатывавшей в 80-е годы XX века операционные системы для микропроцессоров Motorola. Корпорация Microsoft «взялась» за этот вид шрифта достаточно поздно – в период работы над Windows 3-й версии (которая вышла в свет в 1992 г.). Шрифты TrueType развили идею, заложенную в их векторных собратьях, – изображения символов описываются математически, но с той разницей, что теперь формулы обслуживают не составляющие изображение символа линии, а его контур. Более того, при описании глифа применяются не только прямые, но и кривые линии. Поэтому при масштабировании шрифт TrueType сохраняет плавность начертания. Еще одно пре-

имущество контурных шрифтов заключается в возможности вывода символа под любым углом относительно оси X; такой замечательной способностью ни растровые, ни векторные коллеги шрифта TrueType не располагают.

Разработанные в стенах корпораций Adobe и Microsoft шрифты OpenType представляют собой расширение TrueType, их ключевая положительная черта – поддержка принтеров PostScript. Кроме того, технология OpenType позволяет объединить несколько шрифтов OpenType в один файл, называемый коллекцией шрифтов TrueType.

Набор символов

В документации SDK к Windows выделяется 5 основных групп наборов символов. В первую очередь это *набор символов Windows* – эквивалент набора ANSI. Каждый символ этого набора идентифицируется значением размерностью в 1 байт, таким образом, набор Windows может содержать не более 2^8=256 символов. Вторая группа называется *набором Unicode*. Его принципиальное отличие от набора ANSI заключается в количестве байт, отводимых для хранения кода одного символа. В настоящее время Unicode в распоряжение одного символа предоставляет два байта памяти, что позволяет

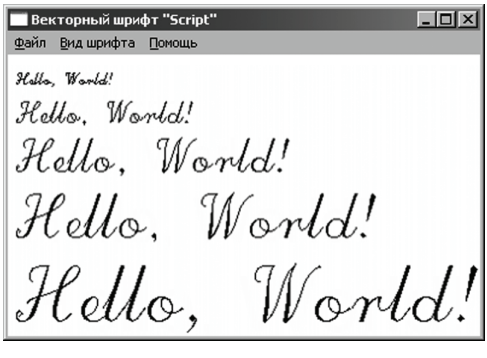


Рис. 19.2. Векторный шрифт Script

Рис. 19.3. Контурный шрифт Times New Roman

описать 2^{16}=65536 символов. Для поддержки окон MS-DOS в Windows сохранен старый *набор символов OEM* (*original equipment manufacturer*). Помимо шрифтов, содержащих изображения цифр и букв различных алфавитов, в Windows эксплуатируются шрифты, поддерживающие *специализированный набор символов* (например, шрифты Symbol, WebDings и WingDings). К *пятой группе* относятся наборы символов сторонних разработчиков, предназначенных для обслуживания специфичных устройств вывода (например, набор *Extended Binary Coded Decimal Interchange Code*, EBCDIC). Для использования такого набора в Windows необходим драйвер, способный осуществлять взаимные преобразования между набором символов операционной системы и набором символов устройства.

Набор символов OEM (ASCII) и набор символов Windows (ANSI) совпадают только до 127 (0x7F) значения, далее следуют символы национальных алфавитов.

Современный компьютерный шрифт обязан уметь работать с *кодовыми страницами* (*code pages*), в которых каждому символу соответствует одно- или двухбайтное значение. Кодовая страница набора символов Windows, специализирующаяся на представлении символов кириллицы, идентифицируется номером 1251. Первая половина таблицы целиком и полностью соответствует 7-битной кодировке ANSI, в которой значения с 0 по 31 отведены для управляющих кодов, далее следуют математические знаки, арабские цифры, прописные и строчные латинские буквы. Вторая половина таблицы (символы с номерами 128–255) отведена под буквы кириллицы и дополнительные символы (рис. 19.4).

Несмотря на то, что 8-битных кодировок вполне достаточно для обслуживания подавляющего большинства национальных алфавитов, в качестве точки опоры современные версии Windows выбрали 16-разрядную кодировку Unicode. За счет удвоения объема памяти, отводимого для хранения одного символа, кодировка Unicode способна описать 65 536 символов, а этого вполне достаточно для обслуживания всех алфавитов земного шара, включая тысячи китайских, корейских и японских иероглифов.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	□
80	Ђ	Ѓ	‚	ѓ	„	…	†	‡	€	‰	Љ	‹	Њ	Ќ	Ћ	Џ
90	ђ	'	'	"	"	•	–	—	~	™	љ	›	њ	ќ	ћ	џ
A0		Ў	ў	Ј	¤	Ґ	¦	§	Ё	©	Є	«	¬	-	®	Ї
B0	°	±	І	і	ґ	µ	¶	·	ё	№	є	»	ј	S	s	ї
C0	А	Б	В	Г	Д	Е	Ж	З	И	Й	К	Л	М	Н	О	П
D0	Р	С	Т	У	Ф	Х	Ц	Ч	Ш	Щ	Ъ	Ы	Ь	Э	Ю	Я
E0	а	б	в	г	д	е	ж	з	и	й	к	л	м	н	о	п
F0	р	с	т	у	ф	х	ц	ч	ш	щ	ъ	ы	ь	э	ю	я

Рис. 19.4. Состав кодовой страницы 1251

Ключевые метрики логического шрифта

При выводе текста на поверхности графического устройства строка размещается вдоль виртуальной *базовой линии* (рис. 19.5). Указанная линия выступает не только в роли опоры для подавляющего большинства символов европейских алфавитов, но и дает начало отсчета для всех вертикальных метрик шрифта. Область ниже базовой линии именуется *подстрочным интервалом*. Здесь размещаются «хвосты» ряда символов (например, букв русского алфавита «р» и «у»). Выше базовой линии находится *надстрочный интервал*. Редкий символ достигает верхней границы надстрочного интервала. Оставшееся свободным пространство называется *внутренним полем*, такой запас сделан ради размещения над глифами диакритических знаков. Над надстрочным интервалом располагается еще одна область – *внешнее поле*, оно предназначено для отделения текстовых строк друг от друга.

В GDI ключевой вертикальной метрикой служит высота символа шрифта, она состоит из подстрочного и надстрочного интервалов. Размер текста принято измерять в пунктах[1], с некоторой степенью округления принято считать, что один дюйм состоит из 72 пунктов.

*Ведя речь об основных метриках шрифта, не стоит забывать, что единицы измерения Windows GDI несколько отличаются от единиц измерений, применяемых в издательском деле. Например, широко используемое понятие «**кегль**» никоим образом не является синонимом анонсированного выше термина «высота шрифта», так как кегль не включает внутреннее поле.*

Горизонтальные метрики шрифта складываются из трех составляющих: собственно ширины символа (*метрика B*) и небольших пустых полей слева (*метрика A*) и справа (*метрика C*) от крайних точек глифа. Сумма трех слагаемых называется *полной шириной* символа. Обычно величина пустых полей положительна, но иногда ширина этих полей может принимать и отрицательные значения, таким образом достигается эффект уплотнения символов в строке.

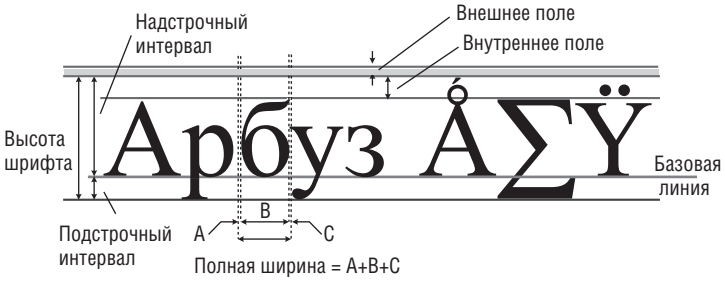


Рис. 19.5. Основные метрики шрифта

[1] Единица измерения «пункт» введена Пьером Симоном Фурнье (*Pierre Simon Fournier*).

Семейства шрифтов

Семейство – это группа объединенных одним названием (типографский работник сказал бы «одной гарнитуры») шрифтов со сходными характеристиками. Так, в семейство инсталлированных на моем компьютере шрифтов гарнитуры «Arial» входят: Arial, **Arial Bold**, ***Arial Bold Italic*** и *Arial Italic*. Заметьте, что имена физических шрифтов формируются из названия гарнитуры и начертания шрифта.

Если приложение позволяет пользователю самостоятельно выбирать тот или иной шрифт, то при перечислении доступных шрифтов программисты стараются строить не полный список имеющихся в системе физических шрифтов, а ограничиться только названиями гарнитур семейств. Такое решение более практично, так как перечень доступных семейств шрифтов в 3–4 раза короче их полного списка. Для изменения начертания шрифта пользовательский интерфейс большинства программных продуктов содержит дополнительные элементы управления, позволяющие назначать нормальное, курсивное, полужирное, подчеркнутое или другие начертания. При выборе пользователем курсивного начертания шрифта из семейства Arial система подставляет Arial Italic, для полужирного начертания – Arial Bold и т. д. При отсутствии в семействе шрифта с условным названием «Arial Strike», предназначенного для вывода зачеркнутого текста, GDI не потеряет присутствия духа и постарается синтезировать такой шрифт, предварительно поколдовав с глифами наиболее подходящего по начертанию физического шрифта из этого же семейства.

Обычно символ шрифта состоит из одного глифа, но бывает и так, что символу соответствует комбинация из нескольких изображений. Такая ситуация наиболее характерна для иероглифов.

Наличие разных семейств шрифтов предполагает существование классификации шрифтов по принадлежности к семейству. Такая классификация предложена в табл. 19.1. Забегая немного вперед, отметим, что представленные в таблице флаги определяют семейство логического шрифта в рамках поля `lfPitchAndFamily` структуры `TLogFont`.

Таблица 19.1. Классификация семейства шрифтов

Семейство	Флаг семейства	Описание	Пример
Decorative	`FF_DECORATIVE`	Оформительский шрифт. Содержит картинки символов.	Old English
Dontcare	`FF_DONTCARE`	Шрифт с произвольными характеристиками.	
Modern	`FF_MODERN`	Шрифт с одинаковой шириной символов.	Courier New
Roman	`FF_ROMAN`	Пропорциональный шрифт, обладающий переменной шириной символов и засечками.	Times New Roman

Семейство	Флаг семейства	Описание	Пример
Script	FF_SCRIPT	Векторный шрифт, имитирующий рукописный текст.	Script
Swiss	FF_SWISS	Пропорциональный шрифт без засечек, с переменной шириной символов.	Arial

Описание логического шрифта, структура TLogFont

В работе со шрифтами можно добиться просто впечатляющих результатов, лишь немного покопавшись в недрах Windows API. Особого исследования достойна структура TLogFont, хранящая параметры логического шрифта.

```
type TLogFont = packed record
 lfHeight: Longint;      //высота символа в логических единицах
 lfWidth: Longint;       //ширина символа в логических единицах
 lfEscapement: Longint;  //угол поворота строки в десятых долях градуса
 lfOrientation: Longint; //определяет угол поворота символа, в графическом
                         //режиме GM_COMPATIBLE не играет роли
 lfWeight: Longint;      //определяет толщину линий шрифта от 0 до 900
 lfItalic: Byte;         //ненулевое значение задает курсивное начертание
 lfUnderline: Byte;      //ненулевое значение – подчеркивание
 lfStrikeOut: Byte;      //ненулевое значение – зачеркивание
 lfCharSet: Byte;        //назначает набор символов шрифта
 lfOutPrecision: Byte;   //определяет, в какой степени точность вывода должна
                         //соответствовать заданным параметрам
 lfClipPrecision: Byte;  //точность отсечения символов, выступающих
                         //за пределы региона
 lfQuality: Byte;        //качество вывода шрифта
 lfPitchAndFamily: Byte; //определяет ширину и семейство шрифта (см. табл. 19.1)
 lfFaceName: array[0..LF_FACESIZE - 1] of AnsiChar; //название гарнитуры
end;
```

В рамках настоящей главы мы лишь познакомились с составом структуры, ограничившись краткими комментариями к каждому из полей. За более существенными подробностями настоятельно рекомендую обратиться к Windows SDK.

Сбор информации об установленных шрифтах

При построении списка доступных шрифтов наибольшей популярностью пользуется функция EnumFontFamiliesEx(). Функция не просто перечисляет все имеющиеся в системе шрифты, она позволяет анализировать семейства шрифтов. Видимо, чтобы мы не сильно расслаблялись, это достоинство в Microsoft с лихвой компенсировали сложностью работы – необходимостью обращения к функции обратного вызова FontEnumProc(). Но обо всем по порядку. Синтаксис основной функции таков:

```
function EnumFontFamiliesEx(DC: HDC;
                            var LogFont: TLogFont;
                          FontEnumProc : TFNFontEnumProc;
                          Param: LPARAM; Flags: DWORD): Boolean;
```

В первый параметр передается дескриптор исследуемого контекста устройства. Параметр LogFont ссылается на переменную, хранящую структуру TLogFont. В данном случае функцию интересуют лишь два поля из этой структуры. В первую очередь это поле lfCharset. Оно определяет, какой именно набор символов нам необходим. Наиболее универсальное значение – DEFAULT_CHARSET, соответствующее полному списку шрифтов. В поле lfFaceName функция возвратит название гарнитуры. Третий по счету параметр и есть тот подводный камень, о который может разбиться программист, который не дочитает эту страницу до конца. Параметр FontEnumProc предназначен для обращения к функции обратного вызова. Она вызывается для каждого семейства шрифтов. Четвертый параметр Param позволяет нам передавать в функцию обратного вызова дополнительные данные. Флаг функции Flags всегда устанавливается в ноль.

Прототип функции обратного вызова выглядит следующим образом:

```
function EnumFontsProc(var LogFont: TLogFont; var TextMetric: TTextMetric;
                       FontType: Integer; Data: Pointer): Integer; stdcall;
```

Чтобы избежать дополнительной путаницы, я сознательно не поставил пиктограмму «Windows GDI». Дело в том, что EnumFontsProc() всегда находится в тени EnumFontFamiliesEx() и никогда не вызывается самостоятельно. За наполнение аргументов функции содержанием отвечает GDI. Здесь первый параметр – структура типа TLogFont. Физические атрибуты шрифта найдут свое отражение в параметре TextMetric. Параметр FontType позволяет судить о типе шрифта, здесь, как говорится, без комментариев, так как имена констант говорят сами за себя: DEVICE_FONTTYPE, RASTER_FONTTYPE или TRUETYPE_FONTTYPE. Заключительный параметр Data при необходимости позволит функции обратного вызова получать дополнительные данные от головной функции EnumFontFamiliesEx().

Порядок применения функций сбора шрифтов в список ListBox продемонстрирован в следующем фрагменте кода:

```
var //…
  ListBox : HWnd;
  LogFont: TLogFont;

{функция обратного вызова}
function EnumFontsProc(var LogFont: TLogFont; var TextMetric: TTextMetric;
                       FontType: Integer; Data: Pointer): Integer; stdcall;
var FontName: string;
begin
  FontName := LogFont.lfFaceName;
  if (SendMessage(ListBox,LB_FINDSTRING,0,LONGINT(PChar(FontName)))=LB_ERR)
       and (FontType=TRUETYPE_FONTTYPE)
  {если имя шрифта отсутствует в списке ListBox и это шрифт TrueType}
  then //тогда вносим имя шрифта в список
```

```
  SendMessage(ListBox,LB_ADDSTRING,0,LONGINT(PChar(FontName)));
  Result := 1;
end;
{оконная процедура}
function WindowProc (Window: HWND; Msg: UINT;
                     wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
  case Msg of
     WM_CREATE :
       begin
       ListBox:= CreateWindow( 'LISTBOX', '',
                 WS_CHILD OR WS_BORDER OR WS_VISIBLE OR LBS_STANDARD,
                 10, 10, 200, 100, Window, 0, HInstance, nil);

                 LogFont.lfCharset := DEFAULT_CHARSET;
                 EnumFontFamiliesEx(GetDC(0), LogFont, @EnumFontsProc, 0, 0);
       end;
//… обработка остальных сообщений
end;
```

В момент обработки сообщения `WM_CREATE` в нашей программе создается список, он описывается оконным дескриптором `ListBox`. Далее вызывается функция `EnumFontFamiliesEx()`, которая моментально перекладывает заботы о построении перечня доступных шрифтов на функцию обратного вызова `EnumFontProc()`. В рамках функции обратного вызова мы заполняем список именами шрифтов, каждый раз (чтобы исключить повторение имен) опрашивая список на наличие строки с таким именем. В результате список заполняется именами семейств установленных в системе шрифтов (рис. 19.6).

В Windows NT/2000/XP информация об установленных в ОС шрифтах хранится в системном реестре по адресу `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Fonts`.

Зачастую для выбора шрифта пользователем приложения вызывают специальные диалоговые окна. Для этого предназначена функция Windows API – `ChooseFont()`. *Листинг, демонстрирующий порядок работы с этой функцией, вы найдете среди примеров к этой главе. Эту же задачу можно решить и средствами библиотеки VCL, для этого в ее составе имеется компонент* `TFontDilog`.

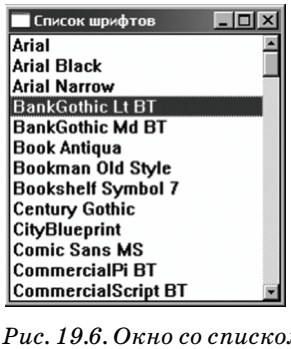


Рис. 19.6. Окно со списком шрифтов

Доступ к системным шрифтам

При описании функции GetStockObject() во второй главе этой книги уже упоминалось о существовании в недрах GDI ряда стандартных объектов. В этом плане шрифты также не являются исключением – в Windows 2000/XP/Vista объявлено 7 констант (см. табл. 2.2), помогающих получить дескриптор соответствующего системного объекта. Например, для выбора в контекст моноширинного системного шрифта остается лишь обратиться к услугам SelectObject().

```
hFont:=GetStockObject(ANSI_FIXED_FONT);
SelectObject(DC,hFont);
//работа с шрифтом
DeleteObject(hFont);
```

С этого момента вывод текстовых строк будет производиться указанным в функции шрифтом.

Логический шрифт

Выполняющиеся в Windows прикладные программы в большинстве своем не способны работать с физическими шрифтами. Но в этом и нет особой необходимости, обслуживание физических шрифтов возложено на могучие плечи соответствующих драйверов, а обычная программа предпочитает общаться с независимым от устройства *логическим шрифтом*.

Создание логического шрифта

Логический шрифт создается на основе реально установленного в системе физического шрифта, при этом GDI сверяется с пожеланиями пользователя, определяющего основные характеристики формируемого объекта. Пользователь указывает название семейства, начертание, размеры, толщину линий и другие отличительные черты будущего логического шрифта, Windows в свою очередь вывернется наизнанку, но синтезирует шрифт с максимально схожими характеристиками. Для создания экземпляра логического шрифта применяется функция:

```
function CreateFont(nHeight, nWidth, nEscapement, nOrientaion,
                    fnWeight: Integer; fdwItalic, fdwUnderline,
                fdwStrikeOut, fdwCharSet, fdwOutputPrecision,
                fdwClipPrecision, fdwQuality, fdwPitchAndFamily: DWORD;
                lpszFace: PChar): HFONT;
```

Вас не должна настораживать бесконечная вереница аргументов. При внимательном рассмотрении перечень параметров оказывается ничем иным, как полями рассмотренной парой страниц ранее структуры TLogFont. Справившись с заданием, функция возвращает дескриптор полноценного объекта GDI – логического шрифта HFONT. Для того чтобы передать последний в контекст, обращайтесь к функции SelectObject(). По окончании работы с логическим шрифтом удалите его с помощью функции DeleteObject().

На мой взгляд, вместо CreateFont() в наших программах значительно удобнее использовать ее коллегу – функцию CreateFontIndirect().

Win GDI
```
function CreateFontIndirect(const LogFont: TLogFont): HFONT;
```

Эта функция решает точно такую же задачу, как и CreateFont(). Она создает экземпляр логического шрифта, но на этот раз вместо десятка аргументов нам приходится оперировать лишь одним. Единственный параметр функции хранит описание создаваемого шрифта в формате записи TLogFont.

Визуальная библиотека компонентов Delphi для создания экземпляра шрифта TFont *предпочитает воспользоваться услугами функции GDI –* CreateFontIndirect().

Благодаря возможностям функций CreateFont() и CreateFontIndirect() мы сможем творить чудеса. Одно из них – вывод повернутого текста, на подобный подвиг не способен ни один из штатных визуальных компонентов VCL до Delphi 7 включительно, а такая возможность вполне может пригодиться, например, при форматировании сложных отчетов.

```
Procedure  GDI_PAINT(DC:HDC);
const s='Hello, World!';
var LogFont: TLogFont;
    Font:HFont;
begin
with LogFont do
   begin
   lfFaceName:=('Courier New');
   lfEscapement := 450; //45 градусов
   lfHeight:=-MulDiv(20, GetDeviceCaps(DC, LOGPIXELSY), 72);
   lfWeight:=FW_BOLD;
   lfStrikeOut:=0;
   lfUnderline:=1;
   lfQuality:=ANTIALIASED_QUALITY;
   end;
Font:=CreateFontIndirect(LogFont);
SelectObject(DC,Font);
TextOut(DC,10,140,pChar(s),Length(s));
end;
```

На рис. 19.7. продемонстрирован результат работы более усложненного листинга. Мы в цикле создаем логический шрифт, позволяющий выводить текст, повернутый к горизонтальной оси под разными углами.

В Delphi 2005, 2006 и 2007 предусмотрена возможность поворота текстовых надписей средствами VCL. Для этого предназначено свойство Orientation *шрифта* TFont.

Изучая листинг, обязательно обратите внимание на «хитрый» способ пересчета размера шрифта (исчисляемого в пунктах) в его физическую высоту в логических единицах.

```
lfHeight:=-MulDiv(20, GetDeviceCaps(DC, LOGPIXELSY), 72); //высота 20 пунктов
```

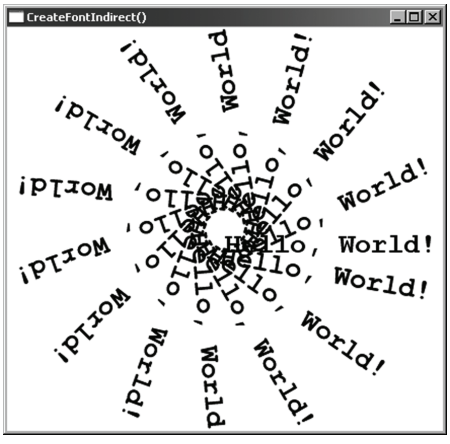


Рис. 19.7. Демонстрация возможностей функции CreateFontInderect( )

Для того чтобы узнать, какое значение следует направить в поле `lfHeight`, чтобы лицезреть шрифт размером в 20 пунктов, нам потребовались некоторые математические расчеты. Функция Windows API `MulDiv()` умножает свой первый аргумент на результаты деления второго аргумента на третий и возвращает ближайшее целое значение. В нашем случае первый аргумент функции – размер шрифта в пунктах, второй – число пикселов в логическом дюйме по высоте (значение получено с помощью `GetDeviceCaps`), третий – количество пунктов в одном дюйме (72). Но помните, что подобная арифметика жизнеспособна только в текстовом режиме отображения (`MM_TEXT`).

С выходом в свет Windows 2000 в состав функций, создающих логические шрифты, влилось очередное пополнение.

Win GDI

```
function CreateFontIndirectEx(const EnumLogFontExA: PEnumLogFontExA):
HFONT;
```

Новая функция работает с развернутой версией структуры `TLogFont`, которая дополнена полями, хранящими имя шрифта, его стиль и описание.

Диалог выбора шрифта

Подавляющее большинство пользователей для выбора шрифта привыкли пользоваться стандартным диалоговым окном, в котором помимо названия семейства шрифтов возможно настроить размер, начертание, цвет и другие параметры будущего экземпляра шрифта. Для вызова указанного диалога требуется помощь функции Windows API:

Win API

```
function ChooseFont(var ChooseFont: TChooseFont): Bool;
```

Функция `ChooseFont` описана в модуле Delphi CommDlg (аналога заголовочного файла commdlg.h из состава Microsoft Visual C++). Для ее работы требуется описать поля почти одноименной структуры `TChooseFont`.

```
type TChooseFont = packed record
    lStructSize: DWORD;    {размер структуры
    hWndOwner: HWnd;       {дескриптор окна-владельца}
    hDC: HDC;              {дескриптор контекста принтера или nil}
    lpLogFont: PLogFontW; {ссылка на структуру TLOGFONT}
    iPointSize: Integer;   {исходный размер шрифта в пунктах, умноженный на 10}
    Flags: DWORD;          {флаги, обычно CF_SCREENFONTS OR CF_EFFECTS}
    rgbColors: COLORREF;   {цвет шрифта}
    lCustData: LPARAM;     {данные, передаваемые в функцию обратного вызова}
    lpfnHook:              {необязательная функция обратного вызова, заполняется
                            при установленном флаге CF_ENABLEHOOK}
              function(Wnd: HWND; Message: UINT;
                       wParam: WPARAM; lParam: LPARAM): UINT stdcall;
    lpTemplateName: PWideChar; {название диалога, флаг CF_ENABLETEMPLATE}
    hInstance: HINST;      {дескриптор модуля с ресурсом диалога, необходим
                            совместно с флагами CF_ENABLETEMPLATEHANDLE
                            или CF_ENABLETEMPLATE }
    lpszStyle: PWideChar; {стиль шрифта}
    nFontType: Word;       {тип шрифта}
    wReserved: Word;       {резерв}
    nSizeMin: Integer;     {минимальный размер шрифта в пунктах, флаг CF_LIMITSIZE}
    nSizeMax: Integer;     {максимальный размер шрифта, флаг CF_LIMITSIZE}
  end;
```

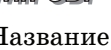
Ниже представлен фрагмент листинга, демонстрирующий порядок создания экземпляра логического шрифта в соответствии с пожеланиями диалогового окна.

```
var LF: TLogFont;
    CF: TChooseFont;
    Font:HFont;
//…
 CF.lStructSize:=SizeOf(TChooseFont);
 CF.hWndOwner:=Window;
 CF.lpLogFont:=@LF;
 CF.Flags := CF_SCREENFONTS OR CF_EFFECTS;

 if ChooseFont(CF)=TRUE then Font:=CreateFontIndirect(LF);
```

Перед обращением к функции `ChooseFont()` мы задействуем минимальный перечень полей структуры `TChooseFont`. В этот список входят поля: `lStructSize` (определение размера данных), `hWndOwner` (дескриптор окна-владельца диалога), указатель на описание логического шрифта `lpLogFont` и направляемые в поле `Flags` флаги `CF_SCREENFONTS` и `CF_EFFECTS`. Вызов функции приводит к отображению диалогового окна выбора шрифта на экране компьютера. В диалоговом окне пользователь (даже не подозревая о своей высокой миссии) заполняет поля записи `TLogFont`, тем самым определяя параметры будущего шрифта. Если работа с окном завершилась нажатием кнопки «OK» (функция возвращает значение `true`), мы создаем новый экземпляр логического шрифта `Font`, который в дальнейшем можно использовать в операциях вывода текстовой информации.

Получение информации о логическом шрифте

В «объятиях» контекста графического устройства всегда находится какой-либо экземпляр логического или системного шрифта. Вспомнить название семейства выбранного в контекст шрифта поможет функция:



```
function GetTextFace(DC: HDC; Count: Integer; Buffer: PChar): Integer;
```

Название семейства помещается буфер, на который ссылается последний параметр функции `Buffer`. В параметр `Count` следует передавать размер приемного буфера. Если функция выполнилась успешно, то она возвратит число символов, помещенных в буфер.

Наиболее полную картину о характеристиках выбранного в контекст логического шрифта предоставит функция:

```
function GetTextMetrics(DC: HDC; var TM: TTextMetric): BOOL;
```

Метрики шрифта передаются в параметр `TM`, представляющий собой структуру `TTextMetric`.

```
type TTextMetric = record
 tmHeight: Longint;           //высота шрифта
 tmAscent: Longint;           //надстрочный интервал
 tmDescent: Longint;          //подстрочный интервал
 tmInternalLeading: Longint; //высота внутреннего поля
 tmExternalLeading: Longint; //высота внешнего поля
 tmAveCharWidth: Longint;     //средняя ширина строчного символа
 tmMaxCharWidth: Longint;     //максимальная ширина символа
 tmWeight: Longint;           //жирность шрифта, в пределах от 0 до 1000
 tmOverhang: Longint;         //величина, на которую увеличивается
                              //ширина символов при эмуляции  шрифта
 tmDigitizedAspectX: Longint; //отношение масштаба устройства по горизонтали
 tmDigitizedAspectY: Longint; //отношение масштаба устройства по вертикали
 tmFirstChar: WideChar;    //код первого символа в наборе
 tmLastChar: WideChar;     //код последнего символа в наборе
 tmDefaultChar: WideChar; //подстановочный символ, применяется когда приложение
                    //пытается вывести символ, код которого отсутствует в наборе
 tmBreakChar: WideChar;   //код символа используемого для переноса слов с одной
                          //строки на другую при выравнивании текста
 tmItalic: Byte;          //курсивное начертание
 tmUnderlined: Byte;      //признак подчеркивания
 tmStruckOut: Byte;       //признак перечеркивания
 tmPitchAndFamily: Byte; //код семейства шрифта
 tmCharSet: Byte;         //набор символов
end;
```

При исследовании контурных шрифтов TrueType и OpenType вместо функции `GetTextMetrics()` *допустимо применять* `GetOutLineTextMetrics()`. *Эта функция возвращает дополнительную информацию о характеристиках логического шрифта.*

Для получения точного значения ширины символов растровых и векторных шрифтов стоит воспользоваться функцией:

```
function GetCharWidth32(DC: HDC;
                        FirstChar, LastChar: UINT; const Widths): BOOL;
```

Функция позволяет узнать ширину символа (либо последовательности символов) при его выводе на поверхности графического устройства, описываемого дескриптором контекста DC. В первом случае в параметры FirstChar и LastChar передается один и тот же код интересующего нас символа. Если же речь идет о выяснении протяженности группы символов, то аргументы FirstChar и LastChar заполняются, соответственно, значениями первого и последнего символа исследуемой последовательности. Значимость последнего варианта применения GetCharWidth32() в некоторой степени условна, так как здесь речь ведется не о любой текстовой строке, а только о последовательности символов, упорядоченных по возрастанию их кода. Таким образом, значение FirstChar всегда должно быть меньше или равно значению, передаваемому в LastChar. Функция возвращает значение в параметр Widths в логических единицах измерения.

Для выяснения горизонтальных метрик шрифтов TrueType вместо GetCharWidth32() возможно обратиться к функции:

```
function GetCharABCWidths(DC: HDC; FirstChar, LastChar: UINT;
                          const ABCStructs): BOOL;
```

Назначение параметров функции аналогично параметрам ее предшественницы, за исключением того, что результат работы GetCharABCWidths() помещается в параметр ABCStructs, описываемый структурой _ABC.

```
Type _ABC = packed record
   abcA: Integer; //левое поле - метрика A
   abcB: UINT;    //ширина глифа - метрика B
   abcC: Integer; //правое поле - метрика C
 end;
```

В результате выполнения функции мы получим информацию обо всех трех горизонтальных характеристиках символа (или последовательности символов) исследуемого шрифта.

При разработке обычных приложений необходимость получения исчерпывающих характеристик о ширине одного-единственного символа относительна. На «бытовом» уровне несравненно более ценна функция, способная разведать геометрические размеры реальной текстовой строки.

```
function GetTextExtentPoint32(DC: HDC; Str: PChar; Count: Integer;
                              var Size: TSize): BOOL;
```

Здесь второй параметр представляет собой ссылку на буфер со строкой Str, в Count передается число символов в строке. Результаты работы функции возвращаются через последний параметр функции. Это структура TSize, состоящая из двух полей типа Longint, в нашем случае предназначенных для хранения ширины и высоты строки в логических единицах измерения.

Инсталляция и удаление шрифта

Для работы обычного приложения, скажем, для базы данных, как правило, достаточно одного-двух семейств шрифтов. Обычно в таких случаях разработчик программного обеспечения в качестве рабочего шрифта использует стандартный (изначально имеющийся в Windows) шрифт. Но возможно, что вы станете творцом программы, использующей специфичный шрифт, которого нет в составе предустановленных шрифтов. Более того, вполне вероятно, что вы захотите оставаться индивидуальным владельцем своего шрифта и не станете его устанавливать обычным порядком. Специально для таких случаев в состав Windows введен ряд функций, позволяющих инсталлировать и удалять шрифты во время работы приложения.

Для того чтобы приложение смогло воспользоваться новым шрифтом, необходимо выполнить его установку. Простейшее решение этой задачи предоставляет функция:

```
function AddFontResource(FileName: PChar): Integer;
```

Функция добавит ресурс `FileName` в Windows. В роли ресурса могут выступать содержащие шрифты стандартные файлы с расширениями: .fon, .fnt, .ttf, .ttc, .fot, .otf, .mmm, .pfb и .pfm. Если шрифт устанавливается из нескольких файлов ресурса, то необходимо перечислить все имена файлов, разделяя их вертикальной чертой. В случае успешного выполнения шрифт заносится в системную таблицу шрифтов и функция возвращает количество успешно добавленных шрифтов. После этого вызвавший функцию `AddFontResource()` процесс сможет работать с новым шрифтом, но этот шрифт по-прежнему будет недоступен другим процессам системы.

Для того чтобы шрифт стал доступен всем приложениям, он должен быть зарегистрирован в системном реестре компьютера.

Правила хорошего тона программирования для Windows требуют, чтобы после установки (удаления) шрифта всем окнам верхнего уровня было разослано сообщение `WM_FONTCHANGE`*, оповещающее их об изменении списка шрифтов.*

С вопросом удаления из системы шрифта, ранее установленного функцией `AddFontResource()`, обычно хорошо справляется функция:

```
function RemoveFontResource(FileName: PChar): Integer;
```

События установки и удаления шрифта должны сопровождаться рассылкой сообщения об изменении в списке шрифтов. Для этого (для оповещения всех окон верхнего уровня) в первый параметр функции `SendMessage()` передается константа `HWND_BROADCAST`, а во второй – `WM_FONTCHANGE`.

```
WM_CREATE :
begin
AddFontResource(PChar('FontName.ttf'));
SendMessage(HWND_BROADCAST, WM_FONTCHANGE, 0, 0);
```

```
end;

WM_DESTROY :
begin
   RemoveFontResource(PChar('FontName.ttf')););
   SendMessage(HWND_BROADCAST, WM_FONTCHANGE, 0, 0);
 end;
```

Представленный пример демонстрирует порядок временного использования вымышленного шрифта, находящегося в файле «FontName.ttf», т. к. по завершению работы программы ресурс изымается. Чтобы установить шрифт в систему на постоянной основе, необходимо осуществить четыре последовательных действия:

1. Скопировать файл шрифта в папку «Windows\Fonts».
2. Добавить ключ в системный реестр: «Software\Microsoft\Windows\CurrentVersion \Fonts».
3. Вызвать функцию AddFontRecource().
4. Отправить широковещательное сообщение об установленном шрифте.

Несколько большими возможностями по управлению процессом инсталляции шрифта обладает функция:

Win GDI

```
function AddFontResourceEx(FileName: PAnsiChar; FR: DWORD;
                           DesignVector: PDesignVector): Integer;
```

Задача первого параметра не изменилась, это информирование системы об имени устанавливаемого ресурса. Второй параметр FR способен принимать одно из двух значений: FR_PRIVATE или FR_NOT_ENUM. Константа FR_PRIVATE определяет, что инсталлируемый шрифт может быть доступен только вызвавшему функцию AddFontResourceEx() процессу. После завершения работы процесса система автоматически деинсталлирует этот шрифт. Передав в функцию константу FR_NOT_ENUM, мы укажем на то, что устанавливаемый шрифт будет скрыт от функции EnumFontFamiliesEx(). Таким образом, ни один из процессов системы, использующий функции перечисления для сбора информации об установленных шрифтах, не сможет увидеть инсталлированный нами шрифт. Третий параметр не используется. При работе с AddFontResourceEx() в рассылке сообщения WM_FONTCHANGE необходимости нет.

Если шрифт устанавливался с помощью AddFontResourceEx(), то для удаления его из системной таблицы шрифтов требуется помощь функции:

Win GDI

```
function RemoveFontResourceEx(FileName: PAnsiChar; FR: DWORD;
                              DesignVector: PDesignVector): BOOL;
```

Назначение параметров функции аналогично аргументам AddFontResourceEx(). Для удаления шрифта, установленного с помощью AddFontResourceEx(), во второй параметр функции следует передавать ту же самую константу, с которой он инсталлировался. Если функция выполнится корректно, то она возвратит ненулевое значение.

Кроме установки шрифтов из файла операционная система способна инсталлировать шрифты из образа в памяти. Это наилучший способ скрывать шрифты от других приложений. Процесс инсталляции начинается с вызова функции:

```
function AddFontMemResourceEx(pFont: Pointer; Count: DWORD;
                              DesignVector: PDesignVector;
                              p4: LPDWORD): THandle;
```

Первая пара параметров определяет адрес и размер блока памяти, содержащего ресурс. Третий параметр зарезервирован и должен быть установлен в 0. Функция возвращает дескриптор установленных шрифтов и их количество (через последний параметр).

И последняя в этой главе функция:

```
function RemoveFontMemResourceEx(p: THandle): BOOL;
```

предназначена для удаления из системы шрифта, установленного из образа в памяти.

РЕЗЮМЕ

Шрифт является основным инструментом, с помощью которого наши программы приобретают возможность представления текстовой информации. Поэтому вопросу работы со шрифтами в Windows GDI посвящен едва ли не самый большой перечень структур, функций и макросов. В этой главе мы рассмотрели основные разновидности шрифтов (растровые, векторные и контурные), познакомились с понятием набора символов, изучили основные метрики шрифтов, разобрали, каким образом осуществляется установка нового шрифта в систему. Таким образом, мы создали хороший плацдарм для работы над следующей главой книги, посвященной операциям с текстом.

20

Вывод текста

Предыдущая глава, посвященная шрифтам, была просто подготовкой к решающему бою, который мы намерены дать GDI сейчас. Цена победы очень высока – управление выводом текстовых данных. И я нисколько не преувеличиваю. Вывод текстовой информации – наиболее распространенная задача, решаемая в подавляющем большинстве программных продуктов, будь это банальный текстовый редактор на уровне блокнота Windows или совершенная программа компьютерной верстки класса Adobe PageMaker. Современные среды программирования предоставляют ряд эффективных инструментов, значительно упрощающих работу с текстом. Мы к ним так привыкли, что при построении приложения уже не в состоянии обойтись без компонентов меток, строк ввода, многострочных редакторов и сеток. Казалось, что все перечисленные элементы управления обладают необходимыми навыками по работе с текстом. Но попробуйте несколько усложнить задачу, выйдите за рамки элементарного приложения и попросите перечисленные элементы вывести текст с эффектом объема или «переломить» текстовую строку пополам. Получилось? То-то и оно, здесь не обойтись без изучаемых в этой главе функций Windows GDI по работе с текстом.

Простейшие приемы вывода текста

Самая элементарная из функций, позволяющих отобразить текстовую строку на поверхности контекста графического устройства, называется `TextOut()`. Все, что требуется от нас, – определиться, что и где выводить.

```
function TextOut(DC: HDC; X, Y: Integer; Str: PChar; Count: Integer):
BOOL;
```

Координаты левой верхней точки строки определяются параметрами `X` и `Y`, указатель на текстовую строку – `Str`, число символов, подлежащих выводу, – `Count`. К услугам `TextOut()` мы уже неоднократно обращались на протяжении всей книги, так что следующий пример вряд ли нуждается в комментариях.

```
WM_PAINT   :
begin
 DC:=BeginPaint(Window,PAINTSTRUCT);
 S:='Hello, World!';
 SetBKColor(DC,$COCOCO); //устанавливаем цвет фона
 TextOut(DC,10,10,pANSIChar(S),Length(S));
 EndPaint(Window,PAINTSTRUCT);
end;
```

Несмотря на заявленную простоту, функция справится с большей частью возникающих перед программистом задач. Поэтому `TextOut()` можно считать именно тем ключевым кирпичиком, из которого строится большинство текстовых приложений. Всякому известно, что из одних и тех же кирпичей можно возвести и дворец, и хижину. Весь вопрос в умении и в затрачиваемом на разработку программы времени. Согласитесь, что небоскреб предпочтительнее возводить из более крупных материалов, чем кирпич, например из стеновых панелей. Именно поэтому Microsoft не остановилась на `TextOut()` и сразу разработала ее расширенную версию.

```
function ExtTextOut(DC: HDC; X, Y: Integer; Options: Longint;
                    Rect: pRect; Str: pChar;
                    Count: Longint; DX: pInteger): BOOL;
```

По сравнению с `TextOut()`, доработанная функция дополнена тремя новыми параметрами. Первый из них – `Options` (табл. 20.1), он определяет некоторые особенности поведения функции, в частности, каким образом она использует область вывода текста, передаваемую через параметр `Rect`. Аргумент `DX` необязателен. Он представляет собой указатель на массив, элементы которого позволяют нам вручную определять позицию каждого глифа текстовой строки. Этот параметр включается в работу при установленном флаге `ETO_PDY`.

Таблица 20.1. Опции функции ExtTextOut( )

Флаг	Описание
`ETO_CLIPPED`	Выступающая часть текста обрезается по границе прямоугольной области `Rect`.
`ETO_GLYPH_INDEX`	Вместо кодов символов в `Str` содержатся 16-разрядные индексы их глифов.
`ETO_IGNORELANGUAGE`	Отключить языковую обработку средствами Windows. Флаг может применяться начиная с Windows NT 4.0.
`ETO_NUMERICSLATIN`	Числа выводятся в общепринятом в Европе формате.
`ETO_NUMERICSLOCAL`	Числа выводятся в соответствии с локальными установками компьютера.
`ETO_OPAQUE`	Прямоугольник `Rect` заливается цветом фона.
`ETO_PDY`	Только для Windows 2000/XP/Vista. Каждый элемент массива (на который ссылается указатель `DX`) содержит пару значений. Первое из них – горизонтальное расстояние, второе – расстояние по вертикали.

Флаг	Описание
ETO_RTLREADING	Флаг применяется только в Windows, локализованной для работы с языками, в которых принято написание справа налево (например, арабским).

Следующая (в порядке возрастания сложности) функция предназначена для вывода многострочных текстовых блоков. По сути – это перекроенная на новый лад функция ExtTextOut(), так как для вывода каждой из строк функция PolyTextOut() вызывает ExtTextOut().

```
function PolyTextOut(DC: HDC; const PolyTextArray; Strings: Integer):
BOOL;
```

Во второй параметр функции направляется указатель на массив структур TPolyText. Число записей в массиве PolyTextArray определяется количеством строк, подлежащих прорисовке на поверхности, описываемой дескриптором контекста DC. Последний параметр определяет, сколько именно строк подлежит выводу. Теперь переключим внимание на определение структуры TPolyText.

```
type TPolyText = packed record
  x: Integer;           //горизонтальная координата строки
  y: Integer;           //вертикальная координата строки
  n: UINT;              //число символов в строке
  PAnsiChar: PAnsiChar; //указатель на строку
  uiFlags: UINT;        //комбинация флагов ETO_OPAQUE (цвет фона)
                        //и ETO_CLIPPED (отсечение)
  rcl: TRect;//ограничивающий область вывода прямоугольник
  pDX: pINT; //указатель на массив значений, определяющий позицию каждого символа
end;
```

Весьма интересен последний параметр структуры. Благодаря нему мы сможем разрядить или наоборот уплотнить символы в строке. Это особенно хорошо заметно по результатам работы следующего примера. В нем выводится две строки «ABCDE» с разной степенью разрежения символов.

```
Procedure GDI_PAINT(DC:HDC);
var PolyTextArray: Array [0..2] OF TPolyText;
    DX: Array [0..4] of integer;
const S='ABCDE';
begin
SelectObject(DC,GetStockObject(ANSI_VAR_FONT));

PolyTextArray[0].x:=30; PolyTextArray[0].y:=10;
PolyTextArray[0].PAnsiChar:=PAnsiChar(S); PolyTextArray[0].n:=Length(S);
PolyTextArray[0].uiFlags:=ETO_OPAQUE;
SetRect(PolyTextArray[0].rcl,30,10,200,40); //область вывода
PolyTextArray[0].pdx:=nil;  //для 1 строки массив пуст

PolyTextArray[1]:=PolyTextArray[0];
PolyTextArray[1].y:=40;
DX[0]:=30; DX[1]:=30; DX[2]:=30; DX[3]:=30; DX [4]:=30;
SetRect(PolyTextArray[1].rcl,30,40,200,70); //область вывода
```

```
  PolyTextArray[1].pdx:=@DX; //для 2 строки применяется разрежение
  PolyTextOut(DC,PolyTextArray,2);
end;
```

Мы выводим две строки с одним и тем же текстом. В первой строке мы не изменяем межсимвольный интервал, оставляя поле `pDx` структуры `TPolyText` без внимания. Во второй строке разряжаем все символы на 30 пикселов. Работу более усложненной версии программы демонстрирует снимок окна, предложенный на рис. 20.1, в ней межсимвольный интервал настраивается перемещением меток. Исходный код усложненной программы, в которой расстояния между символами текстовой строки настраиваются с помощью подвижных меток, вы обнаружите в библиотеке к книге на сайте издательства.

Рис. 20.1. Управление межсимвольным интервалом

Контекст устройства обладает атрибутом, определяющим межсимвольное расстояние на глобальном уровне. Обслуживание этого атрибута осуществляется с помощью функций `SetTextCharacterExtra()` *и* `GetTextCharacterExtra()`.

Управление выводом текста средствами контекста

Контекст устройства не только вооружен арсеналом функций, выводящих текст, но еще обладает рядом атрибутов, определяющих цвета текста и фона, а также режим заполнения фона. Кроме того, у контекста имеются рычаги, управляющие выравниванием выводимого текста. Установка в атрибуты контекста соответствующих значений оказывает влияние на работу всех работающих с текстом функций GDI.

Цвет и фон текста

Настройка атрибута контекста, отвечающего за цвет заливки контуров текстовых символов, осуществляется с помощью пары функций:

Win GDI

```
function GetTextColor(DC: HDC): COLORREF;
function SetTextColor(DC: HDC; Color : COLORREF): COLORREF;
```

Приставка «Get» в названии говорит о том, что это информационная функция, с помощью которой нам позволено лишь уточнить, какой цвет текста в настоящее время использует контекст. Изменить цвет сможет вторая функция. Для этого кроме дескриптора контекста `DC` у нее предусмотрен параметр `Color`. Свидетельством того, что функция `SetTextColor()` выполнила свою работу корректно, станет возвращение ею предыдущего цвета текста.

Управление цветом фона также осуществляется при посредничестве двух функций:

```
function GetBkColor(DC: HDC): COLORREF;
function SetBkColor(DC: HDC; Color : COLORREF): COLORREF;
```

Функция «Get...» узнает, какой именно цвет выбран в ответственный за фон контекста атрибут, в свою очередь, функция «Set...» изменяет состояние этого атрибута.

Особого внимания заслуживает еще одна группа функций GDI:

```
function GetBkMode(DC: HDC) : Integer;
function SetBkMode(DC: HDC;   BkMode : Integer) : Integer;
```

Участницы этого дуэта способны творить чудеса – заставляя контекст формировать прозрачный фон. Первая функция информационная, она возвратит одну из двух констант: OPAQUE или TRANSPARENT. Вторая функция позволяет сменить режим заливки фона, передав перечисленные константы в параметр BkMode. Случай константы OPAQUE элементарен – фоновая часть закрашивается цветом, установленным функцией SetBkColor(). Но ситуация принципиально изменится, как только мы переведем контекст в режим TRANSPARENT. С этого момента контуры символов станут «ложиться» на поверхность контекста устройства, не затрагивая фоновые пикселы, – мы получим эффект прозрачности.

Настройка межсимвольных интервалов

В типографском деле изменение интервала между символами называют *кернингом*, который необходим для улучшения внешнего вида текста и повышения читабельности строк. Windows GDI предлагает несколько способов управления кернингом, наиболее простой из них опирается на функцию:

```
function SetTextCharacterExtra(DC: HDC; CharExtra: Integer): Integer;
```

Ключевой аргумент функции – параметр CharExtra, он и назначает межсимвольное расстояние в логических единицах. Передаваемые в параметр значения могут быть как положительными (что соответствует разряжению текста), так и отрицательными (приводит к уплотнению текстовой строки). Нетрудно догадаться, что у SetTextCharacterExtra() имеется напарница – функция, предоставляющая информацию об установленном в данный момент межсимвольном расстоянии:

```
Function GetTextCharacterExtra(DC:HDC):Integer;
```

По умолчанию у только что созданного контекста межсимвольное расстояние равно нулю.

Разработчиками контурных шрифтов создается таблица кернинга, в которой указываются точные межсимвольные расстояния между различными парами глифов. Это делается исключительно для повышения читабельности текстовых данных. Ваша программа сможет узнать эти данные для текущего шрифта с помощью функции GetKerningPairs().

Выравнивание текста

Для выяснения состояния атрибута контекста устройства, отвечающего за способ выравнивания текста и изменения параметров выравнивания, предназначены функции:

```
function GetTextAlign(DC: HDC): UINT;
function SetTextAlign(DC: HDC; Flags: UINT): UINT;
```

Различают четыре группы флагов, оказывающих влияние на выравнивание текста. Для определения порядка расположения текста внутри ограничивающего прямоугольника разрешено использовать по одному флагу из каждой группы (табл. 20.2).

Таблица 20.2. Флаги выравнивания текста

№	Флаг		Описание
1	TA_BASELINE	Выравнивание по вертикали	Выравнивание по базовой линии текста
	TA_BOTTOM		Выравнивание по нижнему краю ограничивающего прямоугольника
	TA_TOP		Выравнивание по верхнему краю ограничивающего прямоугольника
2	TA_CENTER	Выравнивание по горизонтали	Выравнивание по центру ограничивающего прямоугольника
	TA_LEFT		Выравнивание по левому краю ограничивающего прямоугольника
	TA_RIGHT		Выравнивание по правому краю ограничивающего прямоугольника
3	TA_RTLREADING	Для арабских языков и иврита	Текст выводится справа налево
4	TA_NOUPDATECP	Определение положения левого верхнего угла ограничивающего прямоугольника	За точку отсчета принимается левый верхний угол прямоугольника. При выводе текста текущая позиция не обновляется
	TA_UPDATECP		Координаты левого верхнего угла игнорируются. Вывод начинается из текущей позиции контекста, после чего позиция обновляется

Несколько сложнее обстоят дела при выравнивании текстовой строки по ширине прямоугольной области. Вы наверняка заметили, что в набор обязанностей функции SetTextAlign() такая задача не входит, поэтому нам стоит познакомиться с другой функцией:

```
function SetTextJustification(DC: HDC;
                              BreakExtra, BreakCount: Integer): Integer;
```

Функция SetTextJustification() способна управлять шириной интервала между словами. Обычно в роли разделителя слов выступает пробел, но случаются и исключения. Протяженность всего интервала, который мы собираемся перераспределить, необходимо направить в параметр BreakExtra, а в пара-

метр BreakCount требуется передать общее число разделителей слов в строке. Ниже представлена процедура, обеспечивающая выравнивание текстовой строки S по ширине width.

```
Procedure TextOutByWidth(DC:HDC; S :String; X,Y:Integer; Width : Integer);
var Size:TSize;
    BreakExtra, BreakCount, i : integer;
begin
BreakCount:=0;  BreakExtra:=0;
SetTextJustification(DC,BreakExtra,BreakCount);
GetTextExtentPoint32(DC,pAnsiChar(S),Length(S),Size);//узнаем ширину строки
{подсчитываем число пробелов}
For i:=0 to Length(S)-1 do
    if S[i]=#32 then Inc(BreakCount);

if BreakCount>0 then BreakExtra:=(Width-Size.cx);
if BreakExtra>0 then SetTextJustification(DC,BreakExtra,BreakCount);

TextOut(DC,X,Y, pChar(s), Length(s));
end;
```

Заметьте, что нам потребовались услуги упоминавшейся в предыдущей главе функции GetTextExtentPoint32(). Она предоставила весьма ценную информацию о ширине строки S. На рис. 20.2 показан снимок экранной формы, демонстрирующей работу нашей функции. Первая строка выведена в обычном режиме (выровнена по левому краю), вторая – с помощью разработанной нами процедуры TextOutByWidth(). Вторая строка реагирует на изменения горизонтального размера окна, она старается занять всю доступную область вывода. Таким образом осуществляется выравнивание по ширине.

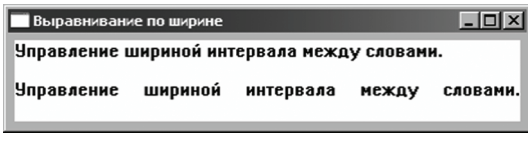


Рис. 20.2. Демонстрация функции SetTextJustification( )

Приемы форматирования текста

В начале главы мы уже говорили, что для вывода простейшего текста на поверхности устройства зачастую достаточно возможностей TextOut(). Вместе с тем в GDI предусмотрен ряд функций, нацеленных на решение специфичных задач.

Вывод табулированного текста

Для вывода текста в табличном формате применяется функция TabbedTextOut(). Ее основное назначение заключается в выводе табулированного текста. Шрифт, цвет выводимого текста и фона определяются настройками контекста.

Win GDI

```
function TabbedTextOut(DC: HDC; X, Y: Integer; Str: PChar;
                       Count: Integer;
                       TabPositions: Integer;
                   var TabStopPositions;
                       TabOrigin: Integer): Longint;
```

Назначение первой пятерки аргументов функции полностью совпадает с задачами одноименных параметров из функции TextOut(). Специфика начинается с шестого параметра. Аргумент TabPositions содержит информацию о количестве задействованных позиций табуляции в массиве TabStopPositions. Позиции табуляции описываются в логических единицах и представляют собой расстояния от левой границы условной невидимой таблицы до места вывода очередной порции текста. Признаком корректного заполнения массива служит то, что его значения упорядочены по возрастанию. Заключительный параметр TabOrigin содержит необязательное смещение, прибавляемое ко всем позициям табуляции. Функция возвращает двойное слово, старшие 16 бит которого содержат высоту, а младшие 16 – ширину результирующей строки. Для того чтобы функция поняла, какие именно блоки текста следует выводить с новой позиции, в требуемые места строки передается ASC-код табуляции – #9.

Рассмотрим пример обращения с функцией TabbedTextOut(). Предположим, у нас имеется строковый массив S, содержащий информацию о сотрудниках предприятия. В каждой строке массива хранятся разделенные символом табуляции данные о фамилии, имени, отчестве, годе рождения и занимаемой должности.

```
var //…
y  :Integer;
S  : Array [0..9] of string;
TSP: Array [0..4] of integer =(0,90,175,300,380);//позиции табуляции

function WindowProc(Window: HWND; Msg: UINT;
                    wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
CASE Msg OF
WM_CREATE :
begin
       S[0]:='Фамилия' +#9+'Имя'+#9+'Отчество'+#9+'Год рожд.'+#9+'Должность';
       S[1]:='Павлова' +#9+'Елена'+#9+'Сергеевна'+#9+'1950 г.р.'+#9+'Директор';
//…
end;
WM_PAINT  :
begin
  DC:=BeginPaint(Window,PAINTSTRUCT);
  FillRect(DC,PAINTSTRUCT.rcPaint,GetStockObject(LTGRAY_BRUSH));
  SetBKMode(DC,TRANSPARENT);

  for y:=0 to 9 do TabbedTextOut(DC,10, y*20,pChar(s[y]),Length(s[y]),5,TSP,0);

  MoveToEx(dc,10,19,nil); LineTo(dc,580,19);
EndPaint(Window,PAINTSTRUCT);
end;
```

```
  //обработка других сообщений
  end;
```

Результаты работы функции представлены на рис. 20.3. Текст выведен в табличном режиме. Положение первой колонки определяется значением параметра X функции TabbedTextOut(), остальные колонки получают смещение в соответствии со значениями из массива табуляции TSP.

Рис. 20.3. Вывод табулированного текста

Массив позиций табуляции может состоять из единственного элемента. В этом случае функция TabbedTextOut() *постарается размещать текстовые колонки на равном удалении друг от друга. Если ширина текстового фрагмента настолько велика, что не может быть выведена с очередной позиции табуляции, то функция воспользуется следующим значением из массива.*

Форматирование абзаца

Для форматирования текста, выводимого в пределах заданной прямоугольной области, предназначена функция:

```
function DrawText(DC: HDC; Str: PChar; Count: Integer;
                  var Rect: TRect; Format: UINT): Integer;
```

Функция DrawText() обладает весьма неординарными возможностями, она способна форматировать однострочный или многострочный текст, на который ссылается указатель на строку Str. Число отображаемых символов определяется аргументом Count. Если строка завершается нулевым значением (#0), то в параметр Count направляется 1. Границы прямоугольника вывода определяются структурой Rect. Правила форматирования назначаются в последнем параметре функции, это комбинация флагов, описанных в табл. 20.3.

Можно выделить несколько групп флагов. В первую очередь – это флаги выравнивания текста по вертикали и горизонтали. Затем следуют флаги однострочного/многострочного форматирования. Очередная группа флагов определяет отношение функции к символу амперсанда (&). По умолчанию ампер-

санд требует, чтобы следующий за ним символ был подчеркнут. Четвертая группа флагов позволяет заменять слишком длинный текст многоточиями. Заключительная группа флагов наделяет функцию полномочиями исследователя.

При успешном завершении функция возвратит высоту текста в логических единицах измерения; исключение составляют случаи, когда используется флаг DT_VCENTER или DT_BOTTOM. В таких ситуациях функция проинформирует нас о расстоянии от верхнего среза ограничивающего прямоугольника до нижней границы текста. Если работа функции завершилась с ошибкой, то она возвратит нулевое значение.

Таблица 20.3. Правила форматирования текста функцией DrawText( )

Значение флага		Описание
DT_TOP	Выравнивание	Выравнивание по верхнему срезу прямоугольника.
DT_VCENTER		Центровка текста по вертикали. Используется только с флагом DT_SINGLELINE.
DT_BOTTOM		Выравнивание текста по нижнему срезу ограничивающего прямоугольника. Используется только с флагом DT_SINGLELINE.
DT_LEFT		Выравнивание влево.
DT_CENTER		Центровка текста по горизонтали.
DT_RIGHT		Выравнивание вправо.
DT_SINGLELINE	Форматирование	Функция игнорирует команду на перевод строки (код #13). Вывод текста всегда осуществляется в одну строку.
DT_WORDBREAK		Позволяет функции выводить текст в многострочном режиме. Переход к новой строке осуществляется в момент достижения границы прямоугольника в месте разрыва между словами.
DT_EDITCONTROL		При выводе текстовых данных функция опирается на правила, заложенные в многострочных текстовых редакторах. В первую очередь это сказывается на способе подсчета средней ширины символа. Кроме того, установленный флаг запрещает функции показ последней строки, если она полностью не помещается в ограничивающий прямоугольник.
DT_TABSTOP		Определение отступа табуляции (по умолчанию 8 символов). Отступ назначается в 8–15 битах параметра Format. Не может применяться совместно с флагами: DT_CALCRECT, DT_EXTERNALLEADING, DT_INTERNAL, DT_NOCLIP и DT_NOPREFIX.
DT_EXPANDTABS		Функция работает с текстом, содержащим символы табуляции (#9). По умолчанию отступ табуляции соответствует ширине 8 символов. Этот флаг не может использоваться совместно с флагами: DT_WORD_ELLIPSIS, DT_PATH_ELLIPSIS и DT_END_ ELLIPSIS.
DT_NOCLIP		Прекращает обрезку текста, вышедшего за пределы ограничивающего прямоугольника.
DT_RTLREADING		Вывод текста справа налево. Применяется при работе с арабскими языками.

Значение флага		Описание
DT_NOPREFIX	Амперсанд	По умолчанию при встрече в строке символа & функция подчеркивает следующий за амперсандом символ. Включение флага вынуждает функцию отказаться от анализа строки на наличие символа амперсанда.
DT_HIDEPREFIX		При выводе текста функция игнорирует символ амперсанда (&).
DT_PREFIXONLY		Вместо вывода текста пометит подчеркиванием «_» место, в котором должен был быть выведен символ, следующий за амперсандом &.
DT_END_ELLIPSIS	Многоточие	Заменяет окончание текстовой строки многоточием, если строка не умещается в отведенные ей границы.
DT_WORD_ELLIPSIS		Все не помещающиеся в пределы ограничивающего прямоугольника слова заменяются символом многоточия.
DT_PATH_ELLIPSIS		Заменяет многоточием середину строки с целью вместить начало и окончание текста в ограничивающий прямоугольник. Если строка содержит обратный слеш (символ «\»), многоточие по возможности добавляется после последнего слеша. Нельзя применять совместно с флагом DT_MODIFYSTRING.
DT_MODIFYSTRING	Исследование	Флаг указывает функции на то, что выводимый текст может быть модифицирован. Совместно с DT_MODIFYSTRING используются флаг DT_END_ELLIPSIS или DT_PATH_ELLIPSIS.
DT_CALCRECT		Исследовательский флаг. Дает команду вычислить ширину и высоту прямоугольника, требуемого для вывода однострочного или многострочного текста. Результат будет возвращен через параметр Rect.
DT_EXTERNALLEADING		При расчете высоты шрифта кроме подстрочного и надстрочного интервала учитывает и внешнее поле (см. рис. 19.5), что обычно не делается.
DT_INTERNAL		Флаг предназначен для расчета метрик шрифта.

В таблицу сознательно не включен еще один флаг – DT_NOFULLWIDTHCHARBREAK. Он упомянут в Windows SDK, но практически не применяется в европейских приложениях, так как предназначен для работы с иероглифами.

На рис. 20.4 представлено несколько экранных снимков, демонстрирующих поведение функции DrawText() с разными флагами форматирования. Текст выводится в пределах белого ограничивающего прямоугольника.

Некоторые дополнительные возможности по абзацному форматированию предоставляет функция:

```
function DrawTextEx(DC: HDC; Str: PChar; Count: Integer;
                    var Rect: TRect; Format : UINT;
                    DTParams: PDrawTextParams): Integer;
```

Отличие расширенной версии функции от рассмотренной немного ранее DrawText() заключается в последнем параметре DTParams. Это указатель на

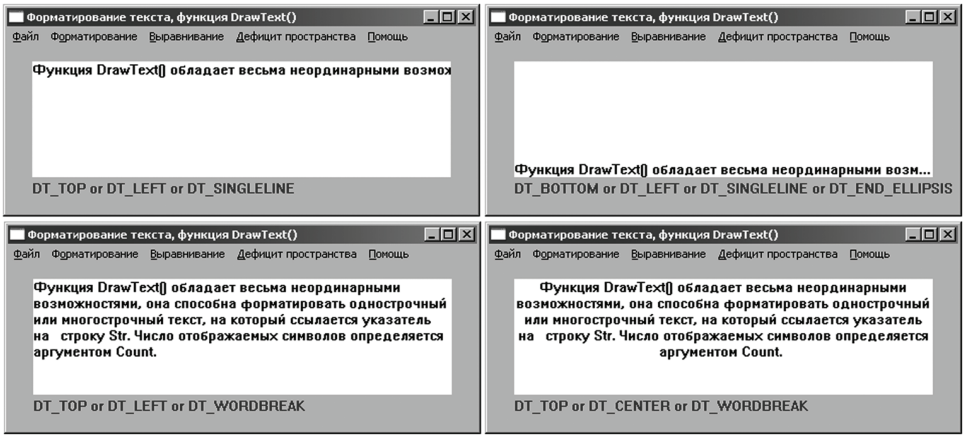


Рис. 20.4. Примеры работы функции DrawText( )

структуру TDrawTextParams, в ней описываются дополнительные опции форматирования.

```
type
   PDrawTextParams = ^TDrawTextParams;
   tagDRAWTEXTPARAMS = packed record
   cbSize: UINT;         //размер структуры в байтах
   iTabLength: Integer;  //величина отступов табуляции в символах
   iLeftMargin: Integer; //левый отступ в символах
   iRightMargin: Integer;//правый отступ в символах
   uiLengthDrawn: UINT;  //возвратит длину строки
end;
```

Дополнительные эффекты при выводе текста

Кроме управления особенностями начертания с использованием возможностей курсивного, полужирного и т. п. начертания символов GDI способен на более серьезные подвиги. Например, интересных эффектов можно добиться за счет многократного вывода текстовой строки с небольшим смещением. Таким образом можно сымитировать тень. Для этого перед выводом основной надписи рисуем темно-серый текст со сдвигом на 3–4 пиксела вправо и вниз относительно опорной точки.

```
Procedure TextOut_Shadow(DC:HDC;X,Y:Integer;S:String); {Эффект тени}
begin
SetTextColor(DC, GetSysColor(COLOR_BTNSHADOW)); //цвет тени
TextOut(DC,X,Y,pAnsiChar(S),Length(s));
SetTextColor(DC, GetSysColor(COLOR_BTNTEXT)); //цвет текста
TextOut(DC,X-3,Y-3,pAnsiChar(S),Length(s));
end;
```

Рельефное начертание получить несколько сложнее – потребуется трехкратное обращение к функции TextOut(). Выпуклая внешность обеспечивается за

счет последовательного вывода белой надписи (смещение (-1,-1)), черной надписи (смещение (1,1)) и, в заключение, – основной строки.

```
Procedure TextOut_Convex(DC:HDC;X,Y:Integer;S:String); {Выпуклый текст}
begin
SetTextColor(DC, GetSysColor(COLOR_WINDOW));
TextOut(DC,X-3,Y-3,pAnsiChar(S),Length(s));
SetTextColor(DC, GetSysColor(COLOR_BTNTEXT));
TextOut(DC,X,Y,pAnsiChar(S),Length(s));
SetTextColor(DC, GetSysColor(COLOR_BTNSHADOW));
TextOut(DC,X-2,Y-2,pAnsiChar(S),Length(s));
end;
```

Для «вдавливания» строки в поверхность контекста нет необходимости пользоваться прессом, мы просто меняем местами белую и черную строку, получается весьма элегантно, что подтверждает рис. 20.5.

```
Procedure TextOut_Pressing(DC:HDC;X,Y:Integer;S:String); {Вдавленный текст}
begin
SetTextColor(DC, GetSysColor(COLOR_BTNTEXT));
TextOut(DC,X,Y,pAnsiChar(S),Length(s));
SetTextColor(DC, GetSysColor(COLOR_WINDOW));
TextOut(DC,X+2,Y+2,pAnsiChar(S),Length(s));
SetTextColor(DC, GetSysColor(COLOR_BTNSHADOW));
TextOut(DC,X+1,Y+1,pAnsiChar(S),Length(s));
end;
```

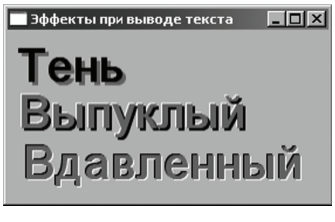


Рис. 20.5. Эффекты при выводе текста

Нестандартная заливка текстовой строки

Все нацеленные на обслуживание шрифтов и текста функции предполагают, что закраска фона текста должна быть однородной. Казалось бы, единственное, что позволено программисту, так это поменять цвет текста и фона с помощью функций SetTextColor() и SetBkColor(). Но это далеко не предел возможностей GDI. Дабы немножко «пошалить», нам достаточно вспомнить об удивительных возможностях функции PatBlt(), способной осуществлять простейшие растровые операции между пикселами поверхности контекста и пикселами выбранной в контекст кисти.

```
Procedure GDI_PAINT(DC:HDC);
var  Brash:HBrush;
     Size:TSize;
const s='Delphi';
```

```
begin
//…
Brash:=CreateHatchBrush(HS_DIAGCROSS,$00FF00); //создаем узорную кисть
SelectObject(DC, Brash);                       //выбираем кисть в контекст
SetBkMode(DC,TRANSPARENT); //перевод контекста в прозрачный режим вывода текста
GetTextExtentPoint32(DC,pChar(S),Length(S),Size); //узнаем размеры строки

PatBlt(DC,10,10,Size.cx,Size.cy,PATINVERT); {1 - первая растровая операция}
TextOut(DC,10,10,pChar(s), Length(s));      {2 - вывод текста}
PatBlt(DC,10,10,Size.cx,Size.cy,PATINVERT); {3 - обратная растровая операция}
```

В представленном фрагменте листинга показан способ заливки текста кистью. Для этого мы создаем узорную кисть и выбираем ее в контекст устройства. Затем переводим контекст в прозрачный режим вывода текста и приступаем к самому интересному:

1. Вызываем `PatBlt()` в режиме `PATINVERT`. В результате растровой операции объединяется фон окна и кисть и эта комбинация инвертируется.
2. В пределах инвертированной области выводим текстовую строку.
3. В заключение вновь вызываем `PatBlt()` в режиме `PATINVERT`. Это действие приводит к восстановлению цвета фона, а текст закрашивается текущей кистью (рис. 20.6).

С текстовыми строками можно проделывать великое множество любопытных фокусов. Например, как добиться контурного начертания? Нет ничего проще. Вспомните о существовании траекторий и функции `StrokePath()`, чертящей траекторию текущим пером. Все остальное зависит от вашего воображения, ведь разновидностей перьев существует великое множество (рис. 20.7).

```
var  LogPen:TLogPen;
const s='Траектория';
//…
LogPen.lopnWidth.X:=1; LogPen.lopnColor:=$FF0000; LogPen.lopnStyle:=PS_DOT;
SelectObject(DC, CreatePenIndirect(LogPen));
BeginPath(DC);  //начало траектории
TextOut(DC,10,70,pChar(s),Length(s));
EndPath(DC);    //завершение траектории
StrokePath(DC); //вывод траектории
```

Раз мы заговорили о траекториях, то вспомните, что при выводе траектории на экран можно задействовать и кисть, но для этого потребуется поддержка со стороны функции `StrokeAndFillPath()`. Самая нижняя надпись на рис. 20.7 нарисована пустым пером и залита узорной кистью с диагональной штриховкой.

Рис. 20.6. Заливка текста кистью

Рис. 20.7. Вывод текста с помощью траекторий

Стоит отметить, что при определении траектории разрешено использовать только текстовые функции `TextOut()` *и* `ExtTextOut()`.

Вывод неактивного текста

В GDI предусмотрена функция, специализирующаяся на рисовании (если следовать дословному переводу) серых строк, т. е. текстовых строк, предназначенных для размещения на пассивных элементах управления:

```
Function GrayString(DC : HDC; Brush : HBRUSH;
                    GrayStringProc: TFNGrayStringProc;
                    lpData : LPARAM;
                  Count, X, Y, Width, Height : Integer) :BOOL;
```

Включая `GrayString()` в свои проекты, надо знать, что желаемого результата можно добиться двумя способами. Простейший из них заключается в заливке текста серой кистью, дескриптор которой передается в параметр `Brush`. Если вместо дескриптора направить в параметр неопределенный указатель, то для «покраски» текста GDI выберет кисть, соответствующую текущей цветовой схеме Windows. Все остальное GDI сделает самостоятельно, нам остается лишь воспользоваться функцией `TextOut()`. Второй подход сложнее, но он позволяет получить более совершенный результат. В этом случае особенности вывода строки определяются программистом в теле функции обратного вызова `GrayStringProc()`. Ссылка на текстовую строку передается в `lpData`. Если строка заканчивается нулем, то в параметр `Count` достаточно передать 0 (функция сама вычислит длину строки), иначе в `Count` требуется указать число отображаемых символов. Параметры `X` и `Y` указывают координаты левой верхней точки выводимой строки. Ширина и высота области вывода назначаются параметрами `Width` и `Height`. Если вы не планируете ограничивать размер строки, то передавайте сюда нулевые значения. Увидев нули, графическое ядро само вычислит требуемые значения. Но этот номер не пройдет с функцией обратного вызова, в этом случае функции необходимо явным образом указать размеры области вывода. Все координаты передаются в единицах измерения устройства.

Заголовок функции обратного вызова должен выглядеть следующим образом:

```
Function GrayStringProc(DC:HDC; lpData : LPARAM; Count:integer) : BOOL; stdcall;
```

Помимо дескриптора контекста устройства функция обратного вызова содержит указатель на текстовую строку lpData и число символов в этой строке Count. Функция обратного вызова может пригодиться при рисовании серого текста нестандартного начертания:

```
Function GrayStringProc(DC:HDC; lpData : LPARAM; Count:integer) : BOOL;stdcall;
var Font:HFont; LogFont:TLogFont; OldDC:HDC;
begin
OldDC:=SaveDC(DC);
SetBKMode(DC,TRANSPARENT);
 GetObject(Font, SizeOf(LogFont), Addr(LogFont));
 with LogFont do
 begin
   lfFaceName:=('Arial');
   lfEscapement := 1800; //переворот строки на 180 градусов
   lfHeight:=-MulDiv(16, GetDeviceCaps(DC, LOGPIXELSY), 72); //высота 16 пунктов
   lfWeight:=FW_BOLD;    //утолщенное начертание
   lfUnderline:=1;       //подчеркивание
 end;
Font:=CreateFontIndirect(LogFont);
SelectObject(DC,Font);
TextOut(DC,0,0,PChar(lpData), Count); //!!!смотри замечание к листингу!!!
RestoreDC(DC,OldDC);
DeleteObject(Font);
end;
```

Обратите внимание на особенность определения места вывода серой строки при использовании функции обратного вызова. Координата левой верхней точки строки определяется суммой значений параметров X, Y головной функции GrayString() и соответствующих параметров функции (например, TextOut()), вызываемой из тела функции обратного вызова.

Обращение к основной функции не должно вызвать у вас каких-либо затруднений.

```
GrayString(DC,0,@GrayStringProc,Integer('Hello, World!'),13,0,10,150,100);
```

В GDI имеется еще одна функция, помогающая нам представлять пассивные элементы управления, она называется DrawState().

РЕЗЮМЕ

Вывод текстовой информации – едва ли не самая часто встречающаяся задача при разработке программного обеспечения. Несмотря на то, что в запасниках GDI имеется всего семь функций (TextOut, ExtTextOut, PolyTextOut, TabbedTextOut, DrawText, DrawTextEx и GrayString), непосредственно отвечающих за работу с текстом, о способах вывода текста можно говорить до бесконечности. Ведь даже наиболее простая функция из этого перечня TextOut() в руках у профессионального программиста способна творить чудеса, что подтверждается материалом настоящей главы.

21

Альфа-наложение и градиентная заливка

Заключительная глава о Windows GDI посвящена рассмотрению двух функций, без которых не может обойтись не одно новомодное приложение, намеренное вскружить голову пользователю (в особенности – работающему с последним детищем Microsoft – операционной системой Windows Vista). Сегодня речь пойдет об альфа-наложении и градиентной заливке. Не могу утверждать, что без указанного сервиса не в состоянии обойтись приложения баз данных, бухгалтерские программы и математические пакеты по расчету траекторий баллистических ракет, но на практике случаются совсем неожиданные вещи. Необъяснимая тяга к прекрасному объединяет подавляющее большинство заказчиков программного обеспечения. И если ваша во всех отношениях безупречная программа не в состоянии похвастаться еще и изысканным, пусть даже пожирающим 99% процессорного времени графическим интерфейсом, то ее место на рынке молниеносно захватят непрестанно «подвисающие», но переливающиеся всеми цветами радуги программы конкурентов. После столь краткого, но, надеюсь, воодушевляющего вступления немедленно приступим к делу.

В Windows GDI за работу с альфа-наложением отвечает функция `AlphaBlend()`. Вторая наша новая знакомая называется `GradientFill()`, она способна осуществить градиентную заливку замкнутой области с поддержкой прозрачности. Еще раз отмечу повышенную ресурсоемкость последних двух функций; шутки шутками, но их работа связана со значительными математическими расчетами, осуществляемыми как головным процессором, так и чипом видеоадаптера.

Альфа-наложение

Ранее, в главе 13, рассуждая о способе описания 32-битного цвета, мы отметили факт существования специального байта, предназначенного для хранения коэффициента прозрачности. В этом случае кроме обработки красной, зеленой и синей составляющей, система обязана анализировать состояние

альфа-канала и осуществлять растровую операцию с учетом его настоятельных пожеланий. Абсолютно непрозрачному пикселу соответствует значение $FF (255 в десятичном представлении), а нулевое значение указывает на полную прозрачность. Несмотря на то, что формат цвета True Color не был новостью и для Windows 95, но функции, позволяющие реализовать эффект прозрачности, в распоряжении программистов появились только начиная с Windows 98.

По сути, альфа-наложение представляет собой очередную разновидность растровой операции, но теперь при формировании результирующего пиксела учету подлежит дополнительный альфа-коэффициент.

Win GDI

```
function AlphaBlend(dcDest: HDC;
            XOriginDest, YOriginDest, WidthDest, HeightDest: Integer;
            dcSrc: HDC;
            XOriginSrc, YOriginSrc, WidthSrc, HeightSrc: Integer;
            BlendFunction: TBlendFunction): BOOL;
```

Практически все параметры функции нам уже знакомы по рассмотренной в главе 17 функции TransparentBlt(). Первая пятерка аргументов описывает область-получатель, вторая пятерка – область-источник. Изюминка функции в последнем параметре, в него направляется структура TBlendFunction, при посредничестве которой программист определяет порядок взаимодействия между пикселами контекста-источника и контекста-получателя.

```
type TBlendFunction = packed record
    BlendOp: BYTE;             {всегда AC_SRC_OVER}
    BlendFlags: BYTE;          {зарезервировано - всегда 0}
    SourceConstantAlpha: BYTE; {альфа-коэффициент от 0 до 255}
    AlphaFormat: BYTE; {формат наложения: 0 - постоянный альфа-коэффициент
                        для всех пикселов;
                        AC_SRC_ALPHA - альфа-коэффициент отдельных пикселов}
end;
```

Если формат альфа-наложения (параметр AlphaFormat) равен нулю, то SourceConstantAlpha определяет степень прозрачности для всех пикселов растра. Как уже говорилось, диапазон ограничен значениями 0...255, где 0 соответствует абсолютной прозрачности, а 255 – полной непрозрачности. Если поле AlphaFormat структуры TBlendFunction принимает значение AC_SCR_ALPHA, то функция приобретает возможность оценивать альфа-коэффициент для каждого пиксела в отдельности. Стоит ли напоминать, что такое возможно только при работе при 32-разрядной глубине цвета, когда альфа-коэффициент каждого пиксела поверхности-источника хранится в его 4-м байте (вместе с тремя байтами красного, зеленого и синего цветов)?

Постоянный альфа-коэффициент

Рассмотрим пример, демонстрирующий работу функции AlphaBlend() с постоянным альфа-коэффициентом. Предполагается, что мы обладаем дескриптором битового образа – hDIB, полученного с помощью разработанной нами функции LoadBMP (Глава 16 «Аппаратно-независимые растры»). Растровая

картинка выбирается в контекст-источник и затем многократно выводится на поверхности контекста-получателя (в области, определенной параметром Rect) с различными (возрастающими с шагом в пять единиц) значениями коэффициента прозрачности.

```
Var hDIB:HBITMAP; //дескриптор битового образа
//…
Procedure GDI_PAINT(DC:HDC;Rect:TRect);
var BlendFunction:TBlendFunction;
    A,Y,X:word;
    memDC:HDC;
begin
BlendFunction.BlendOp:=AC_SRC_OVER;
BlendFunction.BlendFlags:=0;
BlendFunction.SourceConstantAlpha:=0;
BlendFunction.AlphaFormat:=0;

memDC:=CreateCompatibleDC(DC);//создаем контекст в памяти
SelectObject(memDC,hDIB); //забираем образ hDIB в совместимый контекст

X:=0; Y:=0; A:=0;
 while A<=255 do {цикл управления альфа-коэффициентом}
 begin
{переносим образ на экран}
 AlphaBlend(DC,X,Y,32,32,memDC,0,0,32,32,BlendFunction);
 INC(X,33); {выбираем очередное место для вывода рисунка}
 IF X>Rect.Right-33 then
  begin
  INC(Y,50);
  X:=0;
  end;
  INC(A,5); //приращение коэффициента прозрачности
  BlendFunction.SourceConstantAlpha:=A; //устанавливаем альфа-коэффициент
 end;

DeleteDC(memDC);
end;
```

Результаты работы листинга представлены на рис. 21.1 – наш рисунок постепенно «проявляется» на белом фоне окна. Под рисунками проставлены значения альфа-коэффициента.

Эффект постепенного вывода изображения довольно часто применяется в различных играх, но это лишь частный случай работы с функцией AlphaBlend(). Очень часто встречаются задачи, связанные с получением заранее заданного цвета, что требует проведения математических расчетов. В Microsoft SDK приведены следующие «трехэтажные» формулы, характеризующие процесс альфа-наложения в режиме работы AlphaFormat=0:

Красный `Dst.Red = Src.Red * (SCA/255) + Dst.Red * (1 -(SCA/255));`

Зеленый `Dst.Green = Src.Green * (SCA/255) + Dst.Green * (1 -(SCA/255));`

Синий `Dst.Blue = Src.Blue * (SCA/255) + Dst.Blue * (1 -(SCA/255)).`

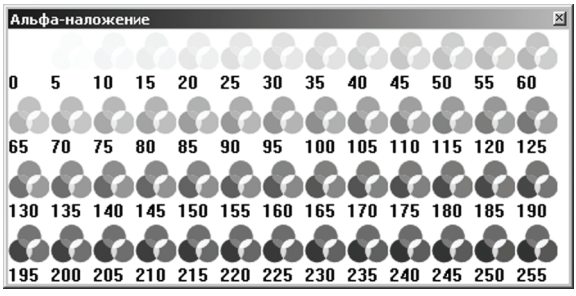


Рис. 21.1. Вывод битового образа с различными коэффициентами прозрачности

Здесь `Dst` определяет контекст-получатель, `Scr` – контекст-источник, `SCA` – сокращение от названия поля `SourceConstantAlpha`.

Если контекст-получатель содержит альфа-канал, то во внимание принимается еще одна формула:

```
Dst.Alpha = Src.Alpha * (SCA/255.0) + Dst.Alpha * (1.0 - (SCA/255.0)).
```

Для проверки приведенных математических выкладок предлагаю реализовать небольшой пример. Предварительно окно заливается белой кистью и в его верхней части рисуются три квадрата чистых цветов (красный, зеленый и синий) со сторонами по 50 пикселов (в дальнейшем эти квадраты будут поочередно использованы в качестве источников операции альфа-наложения). Затем устанавливается постоянный альфа-коэффициент `SourceConstantAlpha`, равный 100 единицам. В заключение с помощью функции `AlphaBlend()` последовательно выводится три квадрата – красный, зеленый и синий. Координаты вывода выбраны с таким расчетом, чтобы квадраты частично перекрывались.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var memDC:HDC; DDB:HBitmap;
    BlendFunction:TBlendFunction;
    Brush:HBRUSH;
begin
memDC:=CreateCompatibleDC(DC);
DDB:=CreateCompatibleBitmap(DC,149,49);
SelectObject(memDC,DDB);
{красный квадрат}
  SetRect(Rct,0,0,49,49);
  Brush:=CreateSolidBrush($0000FF);
  FillRect(memDC,Rct,Brush);
  DeleteObject(Brush);
{зеленый квадрат}
  OffsetRect(Rct,50,0);
  Brush:=CreateSolidBrush($00FF00);
  FillRect(memDC,Rct,Brush);
  DeleteObject(Brush);
{синий квадрат}
```

```
  OffsetRect(Rct,50,0);
  Brush:=CreateSolidBrush($FF0000);
  FillRect(memDC,Rct,Brush);
DeleteObject(Brush);

BitBlt(DC,0,0,159,49,memDC,0,0,SRCCOPY);

BlendFunction.BlendOp:=AC_SRC_OVER;
BlendFunction.BlendFlags:=0;
BlendFunction.SourceConstantAlpha:=100;
BlendFunction.AlphaFormat:=0;

AlphaBlend(DC,50,100,50,50,memDC,0,  0,49,49,BlendFunction);
AlphaBlend(DC,75,125,50,50,memDC,50,0,49,49,BlendFunction);
AlphaBlend(DC,65,85,  50,50,memDC,99,0,49,49,BlendFunction);

DeleteDC(memDC);
DeleteObject(DDB);
end;
```

Результат работы программы представлен на рис. 21.2 (цв. вклейка, с. 419).

Здесь нас больше всего интересуют значения результирующих цветов. Проверьте формулы Microsoft для постоянного коэффициента `SCA=100`. Возьмем простейший случай – наложение красного квадрата на белую поверхность окна. Наши исходные данные таковы:

Поверхность-источник: `Src.Red=255`, `Src.Green=0`, `Src.Blue=0`.

Поверхность-получатель: `Dst.Red=255`, `Dst.Green=255`, `Dst.Blue=255`.

```
Dst.Red   = 255 * (100/255) + 255 * (1 - (100/255))= 255
Dst.Green =   0 * (100/255) + 255 * (1 - (100/255))= 155
Dst.Blue  =   0 * (100/255) + 255 * (1 - (100/255))= 155
```

Затем задача усложняется – поверх красного квадрата рисуется зеленый. Исходные данные изменились:

Поверхность-источник: `Src.Red=0`, `Src.Green=255`, `Src.Blue=0`.

Поверхность-получатель: `Dst.Red=255`, `Dst.Green=155`, `Dst.Blue=155`.

```
Dst.Red   =   0 * (100/255) + 255 * (1 - (100/255))= 155
Dst.Green = 255 * (100/255) + 155 * (1 - (100/255))= 194
Dst.Blue  =   0 * (100/255) + 155 * (1 - (100/255))=  94
```

Результаты всех математических операций вы найдете на рис. 21.2, посему не стану больше утомлять вас математикой. Если есть желание, то проверьте наши расчеты на практике, дополнив проект обработчиком сообщения `WM_MOUSEMOVE`, в рамках которого можно анализировать цвета пикселов поверхности под указателем мыши:

```
Var clr:TColor; X,Y:Integer;
//…
WM_MOUSEMOVE:
begin
  DC:=GetDC(Window);
```

```
  X:=LOWORD(lParam); {Координаты указателя мышки}
  Y:=HIWORD(lParam);

  clr:=GetPixel(DC,X,Y);
  s:=Format('R=%d G=%d B=%d',[GetRValue(Clr),GetGValue(Clr),GetBValue(Clr)]);
  TextOut(DC,175,15,pChar(s),Length(s));
  ReleaseDC(DC,Window);
end;
```

Альфа-коэффициент отдельных пикселов

Обсудим второй режим альфа-наложения – AlphaFormat=AC_SRC_ALPHA. Этот режим может использоваться только при 32-разрядной глубине цвета, потому что теперь данные альфа-канала берутся из пикселов источника. В соответствии с Microsoft SDK расчет результирующих значений цветов и альфа-канала получаются из следующей группы формул:

Сначала формируется растр-источник:

```
Src.Red   = Src.Red   * SCA/255;
Src.Green = Src.Green * SCA/255;
Src.Blue  = Src.Blue  * SCA/255;
Src.Alpha = Src.Alpha * SCA/255.
```

Затем модифицированные пикселы источника накладываются на область получателя:

```
Dst.Red   = Src.Red   + (1 - Src.Alpha) * Dst.Red;
Dst.Green = Src.Green + (1 - Src.Alpha) * Dst.Green;
Dst.Blue  = Src.Blue  + (1 - Src.Alpha) * Dst.Blue;
Dst.Alpha = Src.Alpha + (1 - Src.Alpha) * Dst.Alpha.
```

Как видно из формул, здесь не забыт и постоянный коэффициент SourceConstantAlpha (сокр. SCA) структуры TBlendFunction, он вносит одинаковый коэффициент ослабления для всех четырех составляющих поверхности получателя.

Как правило, внесение данных альфа-канала в растр-источник применяется только при программировании современных игр. Сцены таких игр разрабатываются в специальных редакторах. Посему для демонстрации второго режима работы функции AlphaBlend() мы воспользуемся упрощенным решением – дополним предыдущий проект несколькими строками кода. За счет троекратного вызова функции BitBlt() мы рисуем три копии группы прозрачных квадратов из прошлого примера. Затем переключаем режим альфа-наложения в состояние AC_SRC_ALPHA и с помощью функции AlphaBlend() накладываем на первую группу квадратов квадрат чистого красного цвета, на вторую – зеленого цвета, на третью – синего цвета.

```
Procedure  GDIPLUS_PAINT(DC:HDC);
//…
begin
//…
{три копии группы квадратов}
   BitBlt(DC,10,200,100,100,DC,50,85,SRCCOPY);
```

```
    BitBlt(DC,110,200,100,100,DC,50,85,SRCCOPY);
    BitBlt(DC,210,200,100,100,DC,50,85,SRCCOPY);
{переключение режима альфа-наложения}
BlendFunction.AlphaFormat:=AC_SRC_ALPHA;
//обязательно сравните режимы AC_SRC_ALPHA и 0!!!
//BlendFunction.AlphaFormat:=0;
{альфа-наложение цвета на группу квадратов}
    AlphaBlend(DC,10,200,100,100,DC,0,0,49,49,BlendFunction);  //красный цвет
    AlphaBlend(DC,110,200,100,100,DC,50,0,49,49,BlendFunction); //зеленый цвет
    AlphaBlend(DC,210,200,100,100,DC,100,0,49,49,BlendFunction); //синий цвет
end;
```

Результаты работы примера демонстрирует рис. 21.3 (см. цветную вклейку, с. 419). Переведя функцию в режим AC_SRC_ALPHA, мы добились интересного эффекта. При наложении на область-получатель квадрата красного цвета мы добавляем красную составляющую в цвета всех пикселов области-получателя, при наложении зеленого квадрата – зеленую составляющую, синего – синюю. При таком решении не страдает белый фон окна, но искажаются остальные цвета.

Градиентная заливка области

Весьма популярная в современных приложениях функция произведет градиентную заливку поверхности, идентифицируемую дескриптором контекста DC:

Win GDI

```
function GradientFill(DC: HDC; Vertex: PTriVertex;
                      NumVertex: ULONG; Mesh: Pointer;
                NumMesh, Mode: ULONG): BOOL;
```

Функция позволить заполнить прямоугольные или треугольные области различными оттенками цветов. Формат области и направление заливки определяются заключительным параметром Mode, он может принимать одно из трех вариантов значений. Константа GRADIENT_FILL_RECT_H указывает, что заливке подвергается прямоугольная область. В этом случае оттенки цвета станут изменяться по горизонтали, слева направо. Константа GRADIENT_FILL_RECT_V также заставит функцию обслуживать прямоугольную область, но на этот раз градиент заливки направится по вертикали сверху вниз. Наконец, третий вариант осуществит закраску треугольной области, для этого в параметр Mode передается константа GRADIENT_FILL_TRIANGLE.

Вершины закрашиваемой области определяется во втором параметре Vertex. Это указатель на массив структур TTriVertex.

```
type TTriVertex = packed record
    x: Longint;     //координата x вершины
    y: Longint;     //координата y вершины
    Red: COLOR16;   //16-разрядная красная составляющая
    Green: COLOR16; //16-разрядная зеленая составляющая
    Blue: COLOR16;  //16-разрядная синяя составляющая
    Alpha: COLOR16; //16-разрядное значение альфа-канала
```

```
    end;
```

В элементе массива (описываемого записью TTriVertex) хранятся координаты вершин закрашиваемой области и 16-разрядные значения трех цветовых составляющих и прозрачности. Если мы обслуживаем одну-единственную прямоугольную область, то массив Vertex должен содержать два элемента, определяющих левую верхнюю и правую нижнюю точки прямоугольника. При работе с одним треугольником для идентификации местоположения его вершин нам потребуется три элемента.

В структуре TTriVertex *вместо привычных для нас 8-битных составляющих цвета применяются 16-разрядные значения. Поэтому, если вам требуется передать в поля структуры 8-битные значения, – применяйте тогда операцию сдвига влево на 8 позиций.*

До Delphi 7 включительно второй параметр функции GradientFill() *почему-то был объявлен не как указатель на массив, а как переменная* var Vertex: TTriVertex. *Указанный недостаток устранен в Delphi 2005.*

Функция не ограничивает число одновременно обслуживаемых прямоугольных или треугольных областей – массив в состоянии описывать несколько однотипных фигур. Но при этом нам необходимо явно определить число закрашиваемых фигур (параметр NumMesh), число вершин (параметр NumVertex) и указать соответствие между вершинами и фигурами, с этой целью в параметр Mesh передается указатель на массив структур TGradientTriangle или TGradientRect.

```
type TGradientTriangle= packed record
    Vertex1: ULONG;
    Vertex2: ULONG;
    Vertex3: ULONG;
  end;
```

```
type TGradientRect = packed record
    UpperLeft: ULONG;
    LowerRight: ULONG;
  end;
```

Структура TGradientTriangle предназначена для хранения индексов вершин треугольников. Структура TGradientRect хранит индексы левого верхнего и правого нижнего углов прямоугольника.

Ввиду относительной сложности в работе с функцией предложу вашему вниманию два примера. Следующие строки кода демонстрируют порядок применения функции при организации простейшей градиентной заливки прямоугольной области Rct, цвета назначаются параметрами clrStart и clrEnd.

```
Procedure GDI_PAINT(DC:HDC; Rct:TRect; clrStart,clrEnd : COLORREF);
var Vertexes: array[0..1] of TTriVertex;
    GradientRect:TGradientRect;
begin
Vertexes[0].x := Rct.Left;
Vertexes[0].y := Rct.Top;
Vertexes[0].Red  := GetRValue(clrStart) shl 8;
Vertexes[0].Green:= GetGValue(clrStart) shl 8;
Vertexes[0].Blue := GetBValue(clrStart) shl 8;
```

```
Vertexes[0].Alpha := 0;

Vertexes[1].x := Rct.Right;
Vertexes[1].y := Rct.Bottom;
Vertexes[1].Red  := GetRValue(clrEnd) shl 8;
Vertexes[1].Green:= GetGValue(clrEnd) shl 8;
Vertexes[1].Blue := GetBValue(clrEnd) shl 8;
Vertexes[1].Alpha:= 0;

GradientRect.UpperLeft := 0;
GradientRect.LowerRight := 1;
{!!! код работоспособен только начиная с Delphi 2005 !!!}
GradientFill(DC, @Vertexes[0], 2, @GradientRect, 1, GRADIENT_FILL_RECT_V);
end;
```

Второй пример покажет, как надо работать с двумя треугольниками. Из них мы «склеим» ромб с градиентной заливкой.

```
Procedure GDI_PAINT(DC:HDC; clrStart,clrEnd:COLORREF);
Var Vertexes: array[0..3] of TTriVertex;
    GradientTriangle: array[0..1] of TGradientTriangle;
begin
    Vertexes[0].x := 0; Vertexes[0].y := 140;
    Vertexes[0].Red   := GetRValue(clrStart) SHL 8;
    Vertexes[0].Green := GetGValue(clrStart) SHL 8;
    Vertexes[0].Blue  := GetBValue(clrStart) SHL 8;

    Vertexes[1].x := 140; Vertexes[1].y := 0;
    Vertexes[1].Red   := GetRValue(clrEnd) SHL 8;
    Vertexes[1].Green := GetGValue(clrEnd) SHL 8;
    Vertexes[1].Blue  := GetBValue(clrEnd) SHL 8;

    Vertexes[2].x := 140; Vertexes[2].y := 280;
    Vertexes[2].Red   := GetRValue(clrEnd) SHL 8;
    Vertexes[2].Green := GetGValue(clrEnd) SHL 8;
    Vertexes[2].Blue  := GetBValue(clrEnd) SHL 8;

    Vertexes[3].x := 280; Vertexes[3].y := 140;
    Vertexes[3].Red   := GetRValue(clrStart) SHL 8;
    Vertexes[3].Green := GetGValue(clrStart) SHL 8;
    Vertexes[3].Blue  := GetBValue(clrStart) SHL 8;

    GradientTriangle[0].Vertex1:=0; //индексы вершин 1-го треугольника
    GradientTriangle[0].Vertex2:=1;
    GradientTriangle[0].Vertex3:=2;

    GradientTriangle[1].Vertex1:=1; //индексы вершин 2-го треугольника
    GradientTriangle[1].Vertex2:=3;
    GradientTriangle[1].Vertex3:=2;

GradientFill(DC, @Vertexes, 4, @GradientTriangle, 2, GRADIENT_FILL_TRIANGLE);
end;
```

Так как треугольники обладают двумя общими вершинами (точки `Vertexes[1]` **и** `Vertexes[2]`**), для описания двух фигур нам оказалось достаточно**

четырех элементов массива `Vertexes`. Результат нашей работы представлен на рис. 21.4 (см. цветную вклейку, с. 420).

Если порядок работы с функцией `GradientFill()` *вам показался излишне сложным, то можете отдать свои предпочтения функции* `GradientFillCanvas()` *из модуля* `GraphUtil`. *Указанная функция адаптирована для работы с компонентами Delphi, обладающими свойством* `Canvas`. *К сожалению,* `GradientFillCanvas()` *способна работать только с прямоугольниками и не умеет заливать треугольные области.*

РЕЗЮМЕ

В состав Windows GDI включены вспомогательные функции, позволяющие реализовать эффекты градиентной заливки (`GradientFill`) и альфа-наложения (`AlphaBlend`). Не применяйте указанные функции без особой на то необходимости, так как за красивые визуальные эффекты приходится расплачиваться существенной дополнительной нагрузкой на процессор.

II

Графика в VCL

22

Визуальная библиотека компонентов Delphi

Несмотря на то, что пик развития объектно-ориентированного программирования (ООП) приходится на сегодняшнее время, многие понятия объектно-ориентированного подхода зародились еще в середине 60-х годов прошлого века. Отправной точкой ООП считается разработанный в Норвегии язык Simula. Язык создавался для моделирования процессов реального мира, поэтому модули языка строились не на процедурах, а на моделируемых физических объектах.

Парадоксально, но факт – появившийся на свет на 30 лет позднее прикладной интерфейс программирования Windows API не вполне соответствует многим ключевым критериям современного объектно-ориентированного программирования. Любой программист, серьезно работавший с Windows на уровне вызовов функций API, подтвердит мое утверждение. Судите сами, на протяжении первых 20 глав книги, говоря об «объектах GDI», мы ни разу не воспользовались терминами: «свойство», «метод» или «событие»! Очень удивительно для современного разработчика программного обеспечения, взращенного в райских кущах доброжелательных объектно-ориентированных сред, разработанных в Borland. О «наследовании» и «полиморфизме» также не пришлось сказать ни слова. Впрочем, я даже избегал термина «инкапсуляция», хотя этого ингредиента ООП в Windows API хоть отбавляй. Windows с такой исключительной самоотверженностью скрывает от нас особенности реализации, что это иногда больше идет во вред, чем на пользу.

С некоторой степенью допущения можно сказать, что работа Windows основана на формировании объектов на основе известных структур (например, окон) и структур, о составе полей которых можно только догадываться (хороший тому пример – главный герой этой книги – контекст графического устройства). Основной способ взаимодействия объектов между собой и операционной системой заключается в отправке и получении сообщений. Обращение к объектам осуществляется посредством передачи их уникальных

идентификаторов (дескрипторов) в качестве параметров функций API и никак иначе. Можете разбиться в лепешку, но пока вам в руки не попадет дескриптор, ни одно из заклинаний не достигнет цели. Windows стоит в такой непрошибаемой обороне, что ее стоит ставить в пример нашим футбольным вратарям.

Каким чудесным образом объекты GDI из неприступной операционной системы Windows были «перенесены» в объектно-ориентированный язык Delphi? Для того чтобы ответить на этот вопрос, нам необходимо уделить хотя бы немного внимания рассмотрению основных идей, заложенных в ООП. Только после этого мы сможем приступить к изучению особенностей разработки графического интерфейса на основе *визуальной библиотеки компонентов* (*Visual Components Library, VCL*) Borland Delphi.

Концепция ООП и опорные классы VCL

Уже само название концепции «объектно-ориентированное программирование» указывает на то, что ключевой фигурой в ООП выступает объект. В окружающем нас мире объект – это то, что можно пощупать руками, это книга, кнопка на клавиатуре, сама клавиатура, настольная лампа, спящая под ней моя кошка и проезжающий за окном автомобиль. Каждый из объектов в первую очередь обладает некоторыми характеристиками, рассматривая и сравнивая которые, мы можем судить о сходстве или различиях между объектами. Так, несмотря на всю похожесть друг на друга кнопок клавиатуры, мы отличаем их по сопоставленному каждой из них символу алфавита. Кроме того, есть характеристики, взглянув на которые, мы можем судить о текущем состоянии объекта: лампа может быть включена или выключена, кнопка нажата или отпущена, автомобиль – стоять или мчаться. Все присущие объекту характеристики в терминах ООП называют полями, по сути, это принадлежащие объекту переменные определенного типа, в которые записываются значения, отражающие состояние объекта. Для управления объектом предназначены функции, которые в объектно-ориентированных технологиях обычно называются *методами*. Настольной лампой управляют два ключевых метода: включить и выключить. У автомобиля этих методов значительно больше: он заводится, набирает скорость, изменяет направление движения, замедляет движение и все это время пожирает бензин. В первую очередь методы воздействуют на поля объекта, так, практически все методы автомобиля сосредоточены вокруг поля, описывающего его скорость. Если методы объекта возвращают какое-нибудь значение, то они реализуются в виде функции, если нет – метод представляется процедурой.

Одна из ключевых черт объекта, созданного программным способом, – уникальность его идентификации. Это только у нас во дворе одновременно бегают несколько барбосов, дружно откликающихся на дежурную кличку «Шарик»; в программировании такое решение нежизнеспособно. Поэтому сразу после создания объекта ему автоматически присваивается индивидуальный код. Система гарантирует, что этот код не будет повторен ни у какого другого объекта.

Объект не может быть соткан из воздуха, в объектно-ориентированных языках в качестве предтечи объекта выступает *класс*. Класс – это не что иное, как чертеж проектируемого объекта, но не нарисованный карандашом, а описанный на алгоритмическом языке. В чертеж включается перечень полей будущего объекта и управляющие им методы.

Признанный гуру в области объектно-ориентированного проектирования Гради Буч (Grady Booch) в своих работах выделяет четыре обязательных элемента, без которых идея объектно-ориентированного подхода оказывается недееспособной. К обязательным ингредиентам концепции ООП относятся: абстрагирование, инкапсуляция, модульность и иерархичность (наследование).

Абстрагирование

Все наши приложения описывают реальные физические объекты и/или процессы, в которых участвуют эти объекты. Возьмем даже программы-игрушки, допустим, стратегии. В них моделируется поведение полководца, гоняющего свою электронную армию по экрану компьютера с единственной целью – побить максимум врагов. В хороших стратегиях учитываются боевые качества юнитов, толщина брони, дальнобойность и калибр артиллерии, характеристики местности и многие другие факторы, оказывающие прямое или косвенное влияние на результат боя. Чем больше мелочей смог описать разработчик, тем правдоподобнее впечатление. Но, тем не менее, это не реальность, а всего лишь ее абстрактное представление, удобное для игроков. Приведу другой пример, рассмотрим такой объект, как автомобиль. Это весьма сложная конструкция, включающая тысячи деталей, работающих (или нет) по сложным математическим, физическим, химическим и другим законам. Но с точки зрения автоинспектора любой автомобиль можно абстрагировать как механическое транспортное средство с уникальными номерами кузова и двигателя. Чтобы закончить абстракцию, добавим к перечисленному марку, цвет и номерной знак над бампером. И все! Инспектору нет никакого дела до плотности электролита в вашем аккумуляторе...

Абстракция выделяет ключевые характеристики некоторого объекта, отличающие его от всех других видов объектов, и отбрасывает незначащие детали. Таким образом, абстракция определяет концептуальные границы объекта с точки зрения программиста или пользователя.

На протяжении всех предшествующих глав мы уже работали с одной очень удобной абстракцией – контекстом графического устройства. Нас интересовали лишь способы доступа к пикселам, и мы даже не задумывались о нелегкой судьбе жидких кристаллов, наполняющих матрицу дисплея...

Инкапсуляция

Следующим за абстрагированием столпом, на котором держится концепция ООП, считается инкапсуляция. Основное назначение инкапсуляции – скрытие от пользователя особенностей реализации класса. Проводя параллель с окружающим нас миром, скажу, что точно так же от нас спрятаны внут-

ренности телевизора. Снаружи – только необходимые органы управления, используя которые, мы можем найти интересующий нас канал, изменить громкость, яркость. При этом телезрителя мало интересуют частота гетеродина, кадровая и строчная синхронизации или уровень напряжения в блоке дежурного питания, а для многих названные мною термины сродни тарабарщине. Если что-то сломалось, то мы вызываем мастера. Он специалист по инкапсуляции и бесстрашно снимает заднюю стенку телевизора...

Уже интуитивно понятно, что благодаря инкапсуляции повышается надежность функционирования программного объекта, так как в отличие от четырех-пяти винтов в задней стенке телевизора неопытный программист не сможет выкрутить винты из программно реализованного объекта. Но помимо повышения общей надежности объекта инкапсуляция обеспечивает программную логику его работы, а это еще более важное звено в идее объектно-ориентированного программирования. Классическим примером использования инкапсуляции являются способ предоставления доступа к полям объекта через его свойства и методы.

Модульность

Модульность – это способность системы строиться на основе совокупности взаимодействующих самостоятельных элементов. Подавляющее большинство окружающих нас вещей состоит из таких элементов. Например, системный блок компьютера содержит модули процессора, памяти, видеокарты, которые в свою очередь также состоят из модулей (диодов, транзисторов, резисторов и т. д.).

Поддержание модульности в программировании сродни искусству. В классическом структурном программировании нужны определенные способности, чтобы грамотно разнести используемые совместно подпрограммы по модулям, в объектно-ориентированном программировании требуется умело разделить классы и объекты. Деление на модули значительно упрощает процесс разработки программного обеспечения, мы собираем программу из заранее подготовленных кубиков. Наивысший пилотаж – умение программиста разработать программу, в которой можно заменять один «кубик» на другой без перекомпиляции исходного кода.

Иерархичность (наследование)

Следующим ингредиентом объектно-ориентированного программирования считается наследование. Это совсем простое понятие, его смысл в том, что при описании нового класса допустимо брать за основу другой, родительский класс. Таким образом достигается преемственность поколений – дочерний класс наследует все методы и свойства своего предка. Ограничений на длину цепочки наследования нет, в Delphi объявлены классы, имеющие в родословной больше десятка предков. Таким образом, самый последний в иерархии наследования класс, если так можно выразиться, – самый опытный: он вобрал все лучшее от всех своих «дедушек» и «прадедушек».

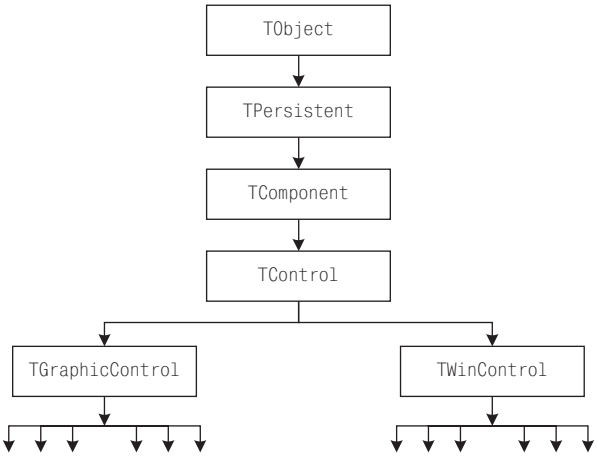


Рис. 22.1. Базовые классы VCL

Вне зависимости от функционального назначения объекта в роли его самого «древнего» предка выступает основа основ иерархии библиотеки визуальных компонентов Delphi – класс `TObject`. Кроме того, `TObject` – единственный класс, не имеющий предка, он создан с чистого листа. На рис. 22.1 представлены базовые классы VCL Delphi, именно они служат основой для создания всех визуальных компонентов языка программирования. По ходу дальнейших глав книги нас в большей степени будут интересовать потомки класса `TGraphicControl` – графические элементы управления.

В результате объединения усилий инкапсуляции, модульности и иерархичности появляется едва ли не самый сильный из козырей объектно-ориентированного программирования – *полиморфизм*. Дословный перевод этого термина обозначает обладание многими формами. Идея полиморфизма заключается в праве экземпляра некоторого класса представлять все классы из его иерархической цепочки. Более того, благодаря полиморфизму мы получаем право скрывать или переопределять поведение унаследованных методов. Поэтому различные по содержанию процедуры и функции всевозможных объектов могут использовать одно и то же имя. А вызов метода будет приводить к выполнению кода, соответствующего конкретному экземпляру объекта.

Основа основ – класс TObject

Класс `TObject` возглавляет иерархию классов в `Delphi`. Другими словами, он родоначальник всех классов. `TObject` – *абстрактный класс*, это означает, что класс физически не способен стать полноценным объектом, а представляет лишь шаблон для последующих поколений классов. `TObject` не имеет никаких свойств. В его арсенале есть только методы, которые позволяют создавать новые объекты на основе базового класса и производить с ними простейшие манипуляции. К задачам, решаемым классом `TObject`, в первую очередь стоит отнести:

- Создание, поддержку и уничтожение объекта. Распределение, инициализация и освобождение памяти, необходимой для этого объекта.
- Осуществление надстройки над информацией о *типе времени выполнения* (*Run Time Type Information*, *RTTI*).
- Поддержку взаимодействия объекта с внешней средой.

Несмотря на то, что непосредственно класс `TObject` никогда не сможет стать физическим объектом, в этом классе подготовлены ключевые методы, решающие вопросы жизни и смерти объекта. Недолгий век любого объекта начинается с вызова его *конструктора*. В `TObject` объявлен самый первый конструктор, который впоследствии будет унаследован всеми компонентами библиотеки визуальных компонентов:

```
constructor Create;
```

Конструктор предназначен для создания нового экземпляра класса, инициализации памяти и обработчика исключительных ситуаций.

В противовес дарующему объекту жизнь конструктору, существует его антипод – специальный метод, именуемый деструктором:

```
destructor Destroy;
```

Деструктор обеспечивает уничтожение экземпляра класса, он без сожаления освобождает ресурсы, используемые этим экземпляром.

Логическим развитием деструктора считается метод:

```
procedure Free;
```

Вместо прямого вызова деструктора `Destroy()` *для уничтожения объекта рекомендуется воспользоваться процедурой* `Free()`.

Процедура выполняет те же самые функции разрушения объекта, но в дополнение к этому перед вызовом деструктора проверяет сам факт существования уничтожаемого объекта, что позволяет избежать ошибок при освобождении ресурсов, используемых объектом.

При завершении работы приложения оно старается самостоятельно уничтожить все принадлежащие ему объекты. Однако существует правило, гласящее, что все объекты, созданные самостоятельно программистом, также должны быть уничтожены программистом.

Класс TPersistent

Абстрактный класс `TPersistent` – это второй по популярности класс VCL. Его основная задача – научить создаваемые из него объекты сохранять свое описание и загружать его из потока (файла или специально зарезервированной области памяти). Благодаря этому все элементы управления способны взаимодействовать с файлом `*.dfm`, а программист получает возможность сохранять все изменения, сделанные в проекте. Как и в случае с `TObject`, основу рассматриваемого класса составляют методы. К наиболее заслуживающим внимание методам `TPersistent` стоит отнести методы присваивания данных:

```
procedure Assign(Source: TPersistent);
procedure AssignTo(Dest: TPersistent);
```

Метод `Assign()` позволяет элементу управления присваивать себе данные, связанные с другим объектом. Например, таким образом объект `Image2`, специализирующийся на хранении и отображении графической информации, получит доступ к картинке, содержащейся в элементе управления `Image1`.

```
Image2.Picture.Assign(Image1.Picture);
```

Вполне естественно, что особенности реализации метода `Assign()` изменяются от класса к классу, поэтому у потомков `TPersistent` процедура `Assign()` вполне может быть переписана, или, как часто говорят, – перекрыта. Метод `AssignTo()` – близкий приятель метода `Assign()`. С помощью него элемент управления сможет разделить свои данные с другим объектом. Например, специализирующийся на отображении графических файлов компонент `Image1` (класс `TImage`) может поделиться загруженным в него изображением с коллегой по цеху – компонентом `Image2`.

```
Image1.Picture.AssignTo(Image2.Picture);
```

Надо знать, что если это возможно, методы `Assign()` и `AssignTo()` не станут создавать дубликат каких-то данных, а постараются получить ссылку на одну и ту же область памяти.

Простейшие графические объекты VCL

Изучение графической составляющей визуальной библиотеки компонентов Delphi мы начнем с простейших «кирпичиков» – классов, инкапсулирующих элементарные объекты GDI, такие как перо, кисть и шрифт. Это наши старые знакомые, в книге им уделено немало места – по целой главе на брата. Теперь рассмотрим особенности реализации аналогичных объектов в Delphi. Мы, как специалисты в объектно-ориентированной технологии, изучение классов начнем с анализа иерархической цепочки наследования (рис. 22.2).

Перо (`TPen`), кисть (`TBrush`) и шрифт (`TFont`) построены на основе абстрактного базового класса `TGraphicsObject`. Их общая задача – избавить программиста

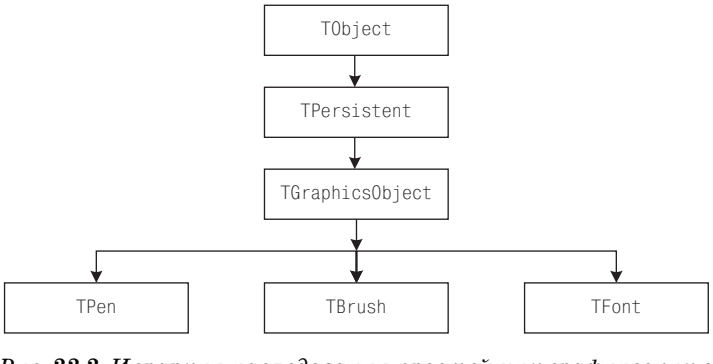


Рис. 22.2. Иерархия наследования простейших графических объектов GDI

Delphi от утомительной обязанности создавать экземпляры объектов GDI, выбирать их в контекст устройства, а затем (по мере ненадобности) и удалять. О том, что фундаментом этих классов являются функции Windows API контекста графического устройства, непосвященному программисту напоминает лишь свойство `Handle`, хранящее дескриптор объекта. Описывая в редакторе будущее перо, кисть или шрифт, мы и не подозреваем, какая работа кипит за кулисами Delphi. А между тем, в процессе создания экземпляра пера, кисти и шрифта задействованы не только функции Windows и классы с рис. 22.1. По команде на формирование нового объекта в работу включается невидимый глазу менеджер ресурсов (сформированный на основе класса `TResourceManager`). В свою очередь, он пользуется услугами объекта синхронизации – критической секции, которая просто необходима для грамотного разделения ресурса (например, экземпляра логического пера) между несколькими потоками. При желании дополнительную информацию о критической секции можно получить, воспользовавшись объявленным на уровне класса `TGraphicsObject` свойством:

```
property OwnerCriticalSection: PRTLCriticalSection;
```

Самое главное, что можно выяснить с помощью указанного свойства, – заблокирован ли экземпляр пера, кисти или шрифта, другими словами, разрешено или запрещено обращаться к объекту из других потоков.

Кроме того, в классе `TGraphicsObject` объявлено событие, вызываемое в момент любых изменений в атрибутах объектов (например, ширины пера, стиля кисти, высоты шрифта):

```
property OnChange: TNotifyEvent;
```

Любое изменение свойств пера, кисти или шрифта приводит к разрушению старого и созданию нового объекта GDI.

Косметическое перо – класс TPen

Класс `TPen` (перо) применяется для определения характеристик линий, рисуемых на поверхности графического устройства. Среда Delphi «ленится» и из всего многообразия способов формирования пера, предоставляемых GDI, пользуется лишь одним – обращением к функции `CreatePenIndirect()`. Таким образом, средствами VCL можно создать только простейшее косметическое перо, а геометрическое – увы, классу `TPen` не по силам! Безусловно, в 9 приложениях из 10 возможностей косметического пера вполне достаточно, но не стоит забывать, что геометрическое перо обладает более широким спектром возможностей. В частности, оно позволяет управлять узором, типом наконечника, типом соединения (стыком) и штриховкой (см. главу 5), кроме того, проведенные геометрическим пером линии чувствительны к двумерным аффинным преобразованиям.

Класс `TPen` *способен инкапсулировать только экземпляр косметического пера. Для создания более совершенного геометрического пера необходимо обратиться за помощью к функции GDI* `ExtCreatePen()`.

После вызова метода Create() по умолчанию создается сплошное черное перо толщиной в 1 пиксел. Единственная ниточка, связывающая экземпляр класса TPen с соответствующим объектом GDI, доступна благодаря свойству:

```
property Handle: HPen;
```

Это не что иное, как дескриптор пера.

Если атрибуты пера по умолчанию не соответствуют пожеланиям разработчика, то к его услугам предоставлены три свойства: цвет, толщина и стиль. Цвет пера определяется в свойстве:

```
property Color: TColor;
```

Толщина линии в логических единицах:

```
property Width: Integer;
```

Стиль пера выбирается из перечня стилей:

```
type TPenStyle = (psSolid, psDash, psDot, psDashDot, psDashDotDot,
                  psClear, psInsideFrame);
```

и назначается в свойстве:

```
property Style: TPenStyle;
```

Если вместо констант TPenStyle вам более по душе константы из состава API (см. табл. 5.1), то пользуйтесь ими так, как предложено в следующих строках кода:

```
var Pen:TPen;
begin
Pen:=TPen.Create;
Pen.Style:=TPenStyle(ps_DashDotDot);
//…
Pen.Free;
end;
```

На рис. 22.3 представлены семь базовых стилей косметического пера, реализуемого классом TPen в Delphi. Использование перечисленных стилей (за исключением psSolid и psInsideFrame) имеет смысл только при толщине пера не более 1 пиксела, в противном случае вы получите не штриховую, а обычную сплошную линию. Особых комментариев достойны стили psClear и psInsideFrame. Стиль psClear соответствует пустому перу – рисование отключается. Стиль psInsideFrame обычно используется при выводе простейших геометрических фигур пером шириной более 1 пиксела. Дело в том, что при использовании утолщенного пера (Style <> psInsideFrame) границы фигуры расширяются относительно заданных координат. Если вы нарисуете прямоугольник с координатами левого верхнего угла (10,10) и правого нижнего (100,100) пером psSolid толщиной 10 пикселов, действительный верхний угол окажется в точке (1,1), а нижний – в (109,109). Границы фигуры раздвинутся ровно на ширину пера. Установка текущего пера в состояние psInsideFrame запретит расширяться границам фигуры – линия будет утолщена вовнутрь.

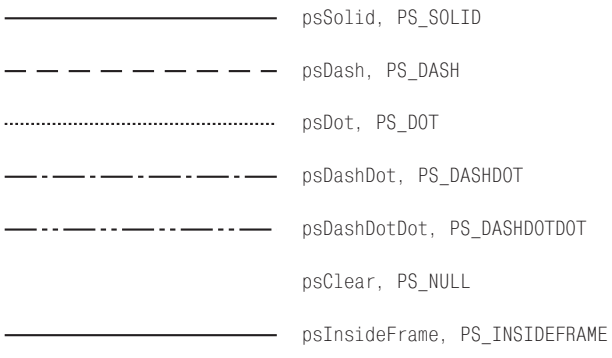


Рис. 22.3. Стили пера TPen

В классе TPen специалисты Borland не просто повторили косметическое перо GDI, но и внесли в него одно весьма полезное усовершенствование – режим прорисовки линии на холсте:

```
property Mode: TPenMode;
type TPenMode = (pmBlack, pmWhite, pmNop, pmNot, pmCopy, pmNotCopy,
                 pmMergePenNot, pmMaskPenNot, pmMergeNotPen, pmMaskNotPen,
                 pmMerge, pmNotMerge, pmMask, pmNotMask, pmXor, pmNotXor);
```

Представление цветной линии в таком случае определяется не только цветом пера, но и цветом области холста (по которой проходит линия) и характером сложения этих цветов. В Delphi предложено 16 стандартных режимов взаимодействия линии и холста. Это не что иное, как знакомые нам ROP2 коды (см. табл. 17.1), определяющие характер бинарной растровой операции между пикселами пера и пикселами поверхности.

Кисть – класс TBrush

Класс TBrush предназначен для инкапсуляции кисти GDI. Для создания экземпляра кисти Delphi пользуется услугами наиболее универсальной (предназначенной для формирования кисти) функции Windows API– CreateBrushIndirect(). Традиционный для объектов GDI дескриптор находится в свойстве:

```
property Handle: HBRUSH;
```

Стили штриховки определяются типом данных TBrushStyle (рис. 22.4):

```
type TBrushStyle = (bsSolid, bsClear, bsHorizontal,
bsVertical, bsFDiagonal,
bsBDiagonal, bsCross,
bsDiagCross);
```

Нетрудно заметить, что названия штриховок в Delphi созвучны именам констант Windows GDI (см. табл. 4.3). Для назначения стиля обращаемся к свойству:

```
property Style : TBrushStyle;
```

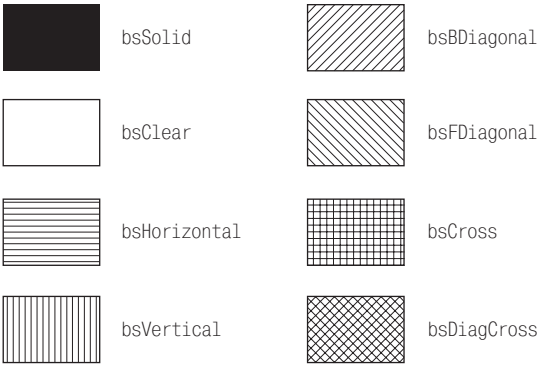


Рис. 22.4. Стандартные стили штриховки кисти

За цвет заливки отвечает свойство:

```
property Color: TColor;
```

На внешний вид кисти оказывает влияние цвет заполнения фона. В следующем примере мы закрашиваем два ряда прямоугольных областей, изменяя не только цвет и стиль кисти, но и цвет заполнения с помощью функции Windows API `SetBkColor()`. Верхний ряд выводится белой кистью с черным фоном, нижний – наоборот, черной кистью с белым фоном.

```
procedure TForm1.FormPaint(Sender: TObject);
var Rct:TRect;
    i:byte;
begin
Form1.Color:=clWhite; //клиентская часть формы окрашена в белый цвет
With Form1.Canvas do
begin
Rct:=RECT(5,15,55,65);
for i:=0 to 7 do{верхний ряд квадратов}
  begin
   Brush.Style:=TBrushStyle(i);  {изменяем стиль кисти}
   Brush.Color:=clWhite;         {цвет кисти белый}
   SetBkColor(Form1.Canvas.Handle,clBlack); {устанавливаем черный фон}
   FillRect(Rct);
   OffsetRect(Rct,55,0);
  end;
Rct:=RECT(5,70,55,120);
for i:=0 to 7 do{нижний ряд квадратов}
  begin
   Brush.Style:=TBrushStyle(i);  {изменяем стиль кисти}
   Brush.Color:=clBlack;         {цвет кисти черный}
   SetBkColor(Form1.Canvas.Handle,Form1.Color); {устанавливаем белый фон}
   FillRect(Rct);
   OffsetRect(Rct,55,0);
  end;
end;
```

Результат работы программы представлен на рис. 22.5.

Вы имеете право создавать и собственные кисти на основе битовых образов. Для этого необходимо загрузить битовый шаблон в свойство:

```
property Bitmap: TBitmap;
```

Имейте в виду, что по умолчанию Delphi не инициализирует свойство `Bitmap` кисти, поэтому перед загрузкой растрового рисунка необходимо создать экземпляр объекта `TBitmap` и присоединить его к кисти так, как представлено в следующем фрагменте кода.

```
with Form1.Canvas do
begin
Brush.Bitmap:=TBitmap.Create;           //инициализация свойства
Brush.Bitmap.LoadFromFile('mybmp.bmp'); //загрузка шаблона кисти
FillRect(Form1.ClientRect);             //закраска поверхности
Brush.Bitmap.Free;                      //очистка ресурса
end;
```

Кисть позволяет копировать атрибуты, установленные в другой кисти. Для этого в ней переписан унаследованный еще от `TPersistent` метод:

```
procedure Assign(Source: TPersistent);
```

Шрифт – класс TFont

Класс `TFont` описывает характеристики шрифта, используемые в Windows. Все элементы управления из палитры компонентов Delphi, способные выводить текст, обладают свойством `Font`. Для создания экземпляра класса `TFont` обращается за помощью к функции GDI `CreateFontIndirect()`. Созданный объект GDI идентифицируется дескриптором, который мы обнаружим в свойстве:

```
property Handle: HFont;
```

Самое главное свойство шрифта содержится в свойстве:

```
property Name: TFontName;
```

Здесь определяется имя шрифта, а точнее, имя семейства шрифтов. Начиная с Delphi версии 2005 по умолчанию каждый вновь создаваемый шрифт принадлежит семейству Tahoma, до этого (в Delphi 1.0...Delphi 7.0) в качестве шрифта по умолчанию назначался MS Sans Serif.

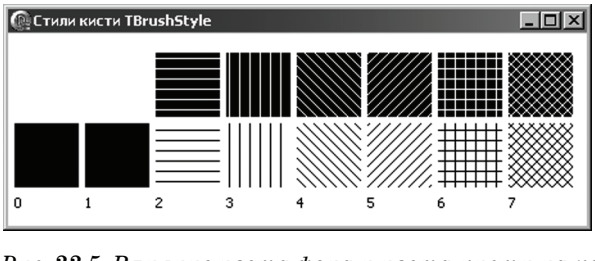


Рис. 22.5. Влияние цвета фона и цвета кисти на характер заливки области

В последних версиях Delphi у шрифта появилось новое свойство, позволяющее поворачивать надписи:

```
property Orientation: Integer;
```

Свойство определяет угол поворота текстовой строки в градусах относительно оси X.

Описание остальных свойств класса шрифтов предложено в табл. 22.1

Таблица 22.1. Основные характеристики TFont

Свойство	Описание
`property Charset: TFontCharset;`	Номер набора символов шрифта. По умолчанию соответствует `DEFAULT_CHARSET`. Русская кириллица – `RUSSIAN_CHARSET`.
`property Color: TColor;`	Цвет шрифта.
`property FontAdapter: IChangeNotifier;`	Автоматически информирует объекты ActiveX о шрифте.
`property Height: Integer;`	Высота шрифта в пикселах.
`property Size: Integer;`	Высота шрифта в пунктах Windows.
`type TFontPitch = (fpDefault, fpVariable, fpFixed);` `property Pitch: TFontPitch;`	Способ установки ширины символов. `fpDefault` –по умолчанию; `fpFixed` – у всех ширина одинакова; `fpVariable` – переменная ширина.
`property PixelsPerInch: Integer;`	Число точек на дюйм, используется Windows для установления соответствия между изображениями шрифта на экране и принтере.
`property Style: TFontStyles;` `TFontStyle = (fsBold, fsItalic, fsUnderline, fsStrikeOut);`	Способ начертания шрифта: жирный, курсив, подчеркнутый и перечеркнутый.

Размер шрифта можно изменять при помощи свойств `Height` *или* `Size`. *Свойства связаны формулой:*

```
Font.Size = -Font.Height * 72 / Font.PixelsPerInch
```

Простейший способ изменения настроек шрифта в компоненте-метке `Label1` представлен в следующем листинге:

```
with Label1.Font do
  begin
  Name:='Arial';   //семейство Arial
  Size:=8;         //размер 8 пунктов
  Color:=clRed;    //цвет красный
  Style:=Font.Style+[fsBold]; //стиль жирный
  end;
```

Приведенный ниже листинг демонстрирует способы сбора информации о шрифтах системы в комбинированный список `ComboBox1` и определяет способ начертания каждого шрифта внутри элемента управления.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
ComboBox1.Items:=Screen.Fonts; //сбор шрифтов системы
ComboBox1.ItemIndex:=0;        //выбираем первый элемент в списке
ComboBox1.style:=csOwnerDrawFixed;
end;
```

В рамках события прорисовки строк комбинированного списка `ComboBox1DrawItem()` мы изменяем порядок вывода данных.

```
procedure TForm1.ComboBox1DrawItem(Control: TWinControl; Index: Integer;
                                   Rect: TRect; State: TOwnerDrawState);
var sRect :TRect;
const S ='ABCDE';
begin
with Combobox1.Canvas do
begin
 FillRect(Rect);
 sRect:=Rect; sRect.Right:=TextWidth(S);
 Font.Name:=ComboBox1.Items.Strings[index];
 TextRect(sRect, sRect.Left, sRect.Top, s);
 Font.Name:='System';
 TextOut(60, Rect.Top, ComboBox1.Items.Strings[index]);
end;
end;
```

Для того чтобы показать пользователю, каким образом будет выглядеть надпись, если она будет выведена выбранным шрифтом, в каждой из строк отображается последовательность из пяти символов «ABCDE». Затем, правее демонстрационной строки, мы выводим название семейства шрифта (рис. 22.6).

Для выбора шрифта можно воспользоваться стандартным диалогом `TFontDialog`*, компонент находится на странице Dialogs палитры компонентов Delphi.*

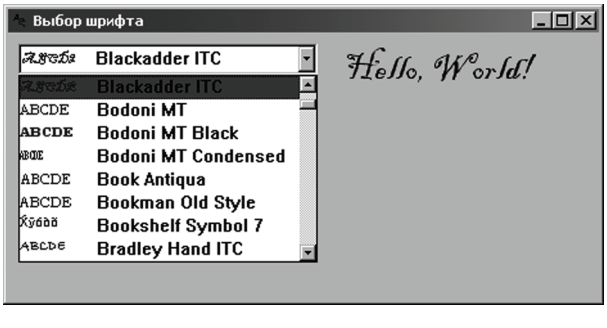


Рис. 22.6. Сбор названий шрифтов в комбинированный список

Восстановление состояния объекта, классы TRecall

При построении графического интерфейса современного приложения не обойтись одним-единственным пером, кистью и шрифтом. Практически всегда логика работы проекта предполагает совместное использование нескольких однотипных объектов, отличающихся друг от друга незначительными деталями: толщиной пера, цветом кисти, стилем начертания шрифта. Технически нет никакой сложности в создании необходимого числа различных экземпляров пера, кисти и шрифта и в использовании их по мере вывода той или иной части изображения, но в этом решении кроется несколько неприятных недостатков. Во-первых, каждый вновь создаваемый объект GDI захватывает часть системного ресурса, поедая драгоценную память. Во-вторых, в операционной системе Windows существует ограничение на количество дескрипторов GDI, которые могут быть доступны процессу. В работе [6] утверждается, что предельное число объектов GDI, которыми вправе обладать 32-разрядный процесс, ограничено примерно 12 тысячами экземпляров. На первый взгляд, 12 000 – это огромное число, но подвох в том, что всего в Windows 2000 может существовать примерно 16 тысяч объектов GDI [6], а ведь в одно и то же время в системе функционируют и другие процессы – им нельзя мешать. В-третьих, огромное число объявленных в коде программы переменных не лучшим образом сказывается на читабельности листинга.

Существует несколько способов борьбы с указанной проблемой. Простейший из них – использование одного единственного экземпляра объекта каждого типа с изменением его характеристик по мере необходимости[1]. Таким образом мы никогда не доберемся до ограничения по числу дескрипторов и одновременно уменьшим остроту проблемы свободной памяти. Правда, бесконечная череда созданий и уничтожений экземпляров перьев, кистей и шрифтов несколько снизит быстродействие программы, но ведь всегда приходится чем-то жертвовать. В пользу этого решения говорит еще одно обстоятельство – в Delphi реализован вспомогательный механизм, позволяющий несколько сократить темпы роста популяции простейших объектов GDI. Идея основана на быстром сохранении характеристик простейшего объекта GDI с возможностью их восстановления. Процессом руководят потомки класса `TRecall` (рис. 22.7).

Нас больше всего интересуют четыре метода наследников `TRecall`. Конструктор `Create()`, создающий экземпляр класса, способный быстро сохранить характеристики пера, кисти или шрифта. Деструктор `Destroy()`, после вызова которого характеристики обслуживаемого объекта возвращаются в исходное состояние. Для запоминания нового исходного состояния объекта вызывается процедура:

```
procedure Store;  //запоминает новое исходное состояние
```

[1] Не забывайте, что изменение любого свойства `TPen`, `TBrush` и `TFont` на деле приводит к созданию нового объекта GDI.

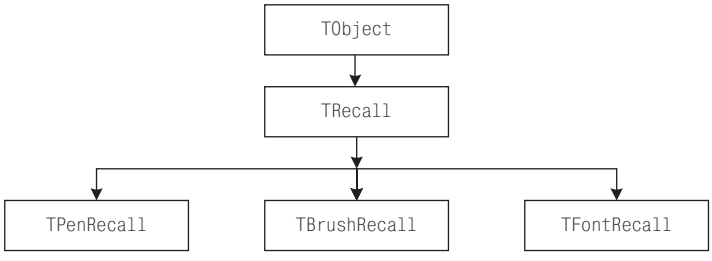


Рис. 22.7. Иерархия наследования класса TRecall

Если вас вполне устраивает текущее состояние пера, кисти или шрифта, но логикой программы предусмотрено уничтожение объекта TRecall, то для предотвращения возврата запомненного объекта в исходное состояние перед обращением к деструктору вызовите процедуру:

```
procedure Forget; //запрет на возврат объекта в исходное состояние
```

Если вы вдруг забыли, какой именно объект мы сопровождаем, то за напоминанием обращаетесь к свойству:

```
property Reference: TPersistent;
```

Здесь находится ссылка на объект, параметры которого мы сохранили.

Класс TRecall служит родоначальником целой плеяды классов, предназначенных для восстановления исходного состояния разнотипных объектов. В частности, для обслуживания пера, кисти и шрифта на базе TRecall создано три специализированных дочерних класса: TPenRecall, TBrushRecall и TFontRecall. В зависимости от особенностей сопровождаемого объекта GDI каждый из классов несколько видоизменил унаследованный им от предка конструктор:

```
constructor Create(APen: TPen);    //конструктор TPenRecall
constructor Create(ABrush: TPen);  //конструктор TBrushRecall
constructor Create(AFont: TFont);  //конструктор TFontRecall
```

В момент создания вспомогательного объекта в единственный параметр его конструктора передается ссылка на обслуживаемый им объект.

Продемонстрируем на небольшом примере порядок использования потомков класса TRecall:

```
procedure TForm1.FormPaint(Sender: TObject);
var Rct:TRect;
    BrRecall:TBrushRecall;
begin
Rct:=RECT(5,5,50,50);
  With Form1.Canvas do
  begin
  Brush.Color:=clWhite;      //белая кисть
  Brush.Style:=bsDiagCross;  //диагональная штриховка
  BrRecall:=TBrushRecall.Create(Brush); //сохраняем кисть
```

```
    FillRect(Rct); TextOut(Rct.Right+5,Rct.Top,'- исходное состояние кисти');
    OffsetRect(Rct,0,60);

    Brush.Style:=bsCross; //изменяем штриховку кисти
    FillRect(Rct); TextOut(Rct.Right+5,Rct.Top,'- изменен стиль кисти');

    OffsetRect(Rct,0,60);
    BrRecall.Destroy; {уничтожаем экземпляр TBrushRecall, что приводит
                       к восстановлению прежних характеристик кисти}
    FillRect(Rct); TextOut(Rct.Right+5,Rct.Top,'- кисть возвращена в исходное');
  end;
end;
```

В рамках события, связанного с перерисовкой формы, мы закрашиваем три прямоугольные области. Первая область заливается кистью с диагональной штриховкой, затем эта кисть сохраняется с помощью объекта `BrRecall: TBrushRecall`. Перед закраской второй области мы изменяем штриховку кисти. Затем перед заливкой третьего квадрата мы уничтожаем объект сопровождения `BrRecall`, что автоматически приводит к восстановлению исходного состояния кисти.

Помимо простейших объектов, таких как перо, кисть и шрифт, разработчику программного обеспечения стоит знать о существовании двух автоматически создаваемых объектов – потомках классов `TScreen` и `TMonitor`.

Глобальный объект «экран» – класс TScreen

Задача класса `TScreen` заключается в отражении состояния экрана. Экземпляр класса создается автоматически при запуске программы и содержит информацию о характеристиках экрана, списке доступных шрифтов и курсоров, составе форм программы, активной форме приложения и элементе управления, находящимся в фокусе ввода. Доступ к объекту производится благодаря глобальной переменной `Screen`. Практически все свойства экрана доступны только в режиме для чтения.

Информация об экране

Высота, ширина и координаты левого верхнего угла рабочего стола хранятся в свойствах:

```
property DesktopHeight: Integer;
property DesktopWidth: Integer;
property DesktopLeft: Integer;
property DesktopTop: Integer;
```

Высота экрана в пикселах.

```
property Height: Integer;
```

Ширина экрана в пикселах.

```
property Width: Integer;
```

Количество пикселов в одном дюйме по вертикали.

```
property PixelsPerInch: Integer;
```

Если компьютер сконфигурирован для одновременной работы с несколькими мониторами, то для этого случая предложены два свойства:

```
property Monitors[Index: Integer]: TMonitor;
property MonitorCount: Integer;
```

Свойство `Monitors` предлагает доступ к каждому из мониторов согласно их индексу. Общее количество мониторов содержится в свойстве `MonitorCount`.

Управление курсором приложения

Класс `TScreen` содержит информацию обо всех доступных курсорах системы.

```
property Cursors[Index: Integer]: HCursor;
```

Для изменения текущего курсора стоит обратиться к свойству:

```
property Cursor: TCursor;
```

Как правило, изменение вида курсора должно информировать пользователя о событиях, происходящих в системе. Например, о занятости процессора красноречиво свидетельствуют песочные часы. Предположим, что одна из функций программы выполняет достаточно высокий объем вычислений и поэтому «притормаживает» систему. Для того чтобы пользователь узнал об этом, просто измените вид указателя мышки на песочные часы, пока не отработала ресурсоемкая функция.

```
var crs : TCursor;
begin
crs:=Screen.Cursor;          //запомним курсор
Screen.Cursor:=crHourgLass; //песочные часы
MySuperHardFunction;         //действия программы
Screen.Cursor:=crs;          //возврат к предыдущему курсору
end;
```

Информация о шрифтах системы

Список всех доступных шрифтов системы можно получить, обратившись к свойству:

```
property Fonts: TStrings;
```

Так как шрифты хранятся в виде набора строк, то не представляет особого труда передать набор любому компоненту, обладающему свойством `Items` или `Lines`.

```
ComboBox1.Items := Screen.Fonts;
```

Для обновления списка шрифтов лучше всего подойдет метод:

```
procedure ResetFonts;
```

Для изменения шрифта, применяемого при выводе оперативной подсказки, меню или имен файлов, в стандартных диалоговых окнах открытия/сохранения файла предназначены свойства:

```
property HintFont: TFont;
property MenuFont: TFont;
property IconFont: TFont;
```

Информация о формах проекта

Класс TScreen держит на контроле все отображаемые на экране компьютера формы. Для обращения к конкретной форме используйте свойство:

```
property Forms[Index: Integer]: TForm;
```

Для этого надо указать индекс формы.

Общее количество форм содержится в свойстве:

```
property FormCount: Integer;
```

Для доступа к активной в данный момент форме и ее активному элементу управления посмотрите на содержимое свойств:

```
property ActiveForm: TForm;
property ActiveControl: TWinControl;
```

Если приложение содержит модули данных, то обратиться к ним возможно через свойство:

```
property DataModules[Index: Integer]: TDataModule;
```

Их количество:

```
property DataModuleCount: Integer;
```

Вполне возможно, что проект вместо классических форм (экземпляров класса TForm) работает с потомками класса TCustomForm. В такой ситуации для доступа к формам стороннего разработчика понадобятся свойства:

```
property ActiveCustomForm: TCustomForm;
property CustomForms[Index: Integer]: TCustomForm;
property CustomFormCount: Integer;
```

Глобальный объект «монитор» – класс TMonitor

Класс TMonitor описан в модуле Forms. Экземпляр класса упрощает организацию взаимодействия приложения и монитора (мониторов, если их несколько). Объект создается автоматически вместе со стартом приложения, для обращения к нему необходимо воспользоваться свойством Monitors глобального объекта Screen.

Все методы монитора унаследованы от класса TObject, поэтому с ними мы знакомиться не станем и сразу перейдем к свойствам (табл. 22.2).

Таблица 22.2. Свойства класса TMonitor

Свойство	Описание
`property Handle: HMONITOR;`	Дескриптор монитора.
`property MonitorNum: Integer;`	Номер монитора в списке `Monitors` глобального объекта `Screen`.
`property Primary: Boolean;`	Состояние `true` сигнализирует о том, что монитор первичный.
`property BoundsRect: TRect;`	Размеры всего монитора. Координаты верхнего левого угла соответствуют координате (0,0) первичного монитора.
`property WorkareaRect: TRect;`	Размеры рабочей области монитора, в нее не входят панель задач и другие дополнительные панели.
`property Left: Integer;`	Логическая позиция левой границы монитора.
`property Top: Integer;`	Логическая позиция верхней границы монитора.
`property Width: Integer;`	Ширина монитора в пикселах
`property Height: Integer;`	Высота монитора в пикселах.

РЕЗЮМЕ

Концепция объектно-ориентированного программирования зиждется на трех базисных понятиях: наследовании, инкапсуляции и полиморфизме. Благодаря наследованию мы получаем право передавать родовые черты от класса-предка к классу-потомку – обеспечивать преемственность поколений. Инкапсуляция необходима для сокрытия от пользователя особенностей реализации класса, таким образом, разработчик класса повышает его отказоустойчивость, защищая его от некорректных внешних действий. Полиморфизм предоставляет классу-потомку право вызывать, скрывать или при необходимости переопределять поведение унаследованных методов.

Класс – это описание объекта на алгоритмическом языке. В классе объявляются поля и методы будущего объекта. Напрямую доступ к полям объекта запрещен, вместо этого программист работает со свойствами. Свойство инкапсулирует поле. Жизненный путь объекта начинается с момента вызова его конструктора и завершается обращением к деструктору.

Визуальная библиотека объектов Delphi инкапсулирует такие объекты GDI, как перо, кисть и шрифт в классах `TPen`, `TBrush` и `TFont`. Такой подход значительно упрощает процесс разработки стандартных приложений для Windows. Вместе с этим перечисленные классы не полностью повторяют возможности соответствующих им объектов GDI. Например, класс `TPen` не способен создавать геометрическое перо, а набор атрибутов шрифта `TFont` несколько меньше перечня полей структуры `TLogFont`. Из трех изученных классов к объектам GDI ближе всего по духу оказалась кисть `TBrush`, это объясняется тем, что экземпляр кисти создается за счет наиболее универсальной функции GDI – `CreateBrushIndirect()`.

23

Холст VCL – класс TCanvas

Класс TCanvas (дословный перевод с английского – холст) описан в модуле graphics. Холст воедино связывает контекст графического устройства, атрибуты контекста и сокращенный (к сожалению) набор функций GDI. Таким образом, TCanvas представляет собой объектно-ориентированное воплощение так горячо любимого нами контекста графического устройства Windows. В своей иерархии наследования класс недалеко ушел от корней библиотеки VCL, по счету он всего лишь третий от TObject (рис. 23.1). Несмотря на такую близость к праотцам TCanvas – уже не абстрактный класс и в состоянии создавать полноценные объекты. Эта полезная способность позволяет разработчикам компонентов внедрять холст в экземпляры своих собственных объектов, таким образом наделяя их почти всем спектром возможностей GDI.

Холст присутствует в большинстве визуальных компонентов Delphi, доступ к нему осуществляется посредством опубликованного у них свойства Canvas. *Кроме того, в VCL предусмотрен специальный объект, готовый исполнять обязанности холста, – это компонент* TPaintBox.

О том, что под «личиной» холста скрывается реальное графическое устройство, напоминает дескриптор контекста, возвращаемый свойством:

```
property Handle: HDC;
```

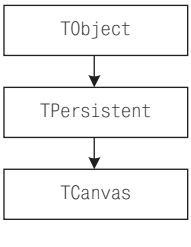


Рис. 23.1. Место TCanvas в иерархии наследования VCL

Дескриптор пригодится для обращения к тем функциям Windows GDI, которым в описании класса TCanvas по тем или иным причинам не нашлось места. Таких немало! Это функции по работе с траекториями и регионами, функции, определяющие режимы отображения Windows, мировые преобразования, ряд функций, связанных с отсечением, и многие другие функции API, рассмотренные нами в первой части этой книги. Вместе с тем в границах класса TCanvas собран золотой минимум возможностей, позволяющих без особого труда создавать проекты с графическим наполнением малой степени сложности.

В холсте инкапсулированы уже знакомые нам кисть, перо и шрифт, взаимодействие с ними осуществляется через одноименные свойства:

```
property Brush: TBrush; //кисть
property Pen: TPen;     //перо
property Font: TFont;   //шрифт
```

Доступ к наиболее часто используемым логическим объектам посредством обращения к свойству весьма удобен. Программисту Delphi нет нужды держать в голове если не сложный, то, по крайней мере, рутинный алгоритм, применяемый при работе с кистью, пером и шрифтом в рамках обычного GDI. Работая на «голом» Windows API, надо последовательно пройти через несколько этапов: создать графический объект, запомнить его дескриптор, выбрать объект в контекст устройства, а на заключительной стадии не забыть освободить отработанный ресурс.

Хотя по поводу урезанных возможностей TCanvas программистами Delphi пролито немало слез, но горю это не помогает, и ни Borland, ни их последователь CodeGear по-прежнему не проявляют особого энтузиазма в совершенствовании холста. Едва ли не единственным положительным сдвигом стало появление возможности поворота текстовых строк в Delphi 2005, но это в большей степени заслуга класса TFont, чем холста. Не получая ответа на мольбы, нашему брату программисту приходится идти на хитрость, объединяя достоинства объектно-ориентированного холста и возможности Windows API. Классический пример «скрещивания» TCanvas с GDI предложен в следующих строках кода:

```
const S='Hello, World!';
begin
  With TCanvas.Create do
  begin
  Handle := GetDC(GetWindow(GetDesktopWindow, GW_OWNER));
  Font.color := clRed; Font.Size :=50;
  Font.name := 'Times new roman';
  SetBKMode(Handle,TRANSPARENT);
  TextOut(1,1, s);
  end;
end;
```

Смысл примера заключается в том, что создав экземпляр класса TCanvas, мы подключаем его к контексту рабочего стола и далее приобретаем право рисовать на его поверхности средствами TCanvas, практически не обращаясь на-

прямую за помощью к Windows API. Теперь вместо заполнения структуры TLogFont для создания экземпляра шрифта мы воспользовались домашней заготовкой Borland – свойством Font холста Delphi.

Холст отличается богатейшим набором методов для рисования, но самый простой из них прячется в свойстве:

```
property Pixels[X, Y: Integer]: TColor;
```

Благодаря Pixels мы получим доступ к любой точке холста по ее координатам X и Y так, как будто это обычный элемент массива. Свойство позволяет не только выяснить, какой цвет установлен в любой точке холста, но и назначить нужной точке требуемый цвет. Простейший пример использования свойства представлен ниже:

```
procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,Y: Integer);
begin
If (ssLeft IN SHIFT) THEN Form1.Canvas.Pixels[X,Y]:=clRed;
end;
```

Мы закрашиваем в красный цвет пикселы на поверхности формы в рамках события, обслуживающего перемещение указателя мыши с удерживаемой в нажатом состоянии левой кнопкой мышки.

Текущая позиция пера хранится в свойстве:

```
property PenPos : TPoint;
```

Способ наложения изображения на холст определяется свойством:

```
property CopyMode: TCopyMode default cmSrcCopy;
type TCopyMode = Longint;
```

Класс TCopyMode представляет собой константы кодов бинарных растровых операций, которые вы найдете, вернувшись к табл. 17.1.

Класс TCanvas *может похвастаться двумя незначительно усовершенствованными потомками:* TControlCanvas *и* TMetafileCanvas. *Первый из них инкапсулируется в ряде элементов управления, допускающих операции прорисовки на своей поверхности. К таким элементам относятся потомки* TCustomForm, TGraphicControl, TCustomCombo *и т. д. В свою очередь,* TMetafileCanvas *специализируется на работе с метафайлами – векторными изображениями.*

Холст практически не обладает рычагами, позволяющими управлять механизмом отсечения. Единственное достойное внимания свойство:

```
property ClipRect : TRect; //только для чтения
```

содержит прямоугольные координаты доступной для графического вывода области.

Линии и кривые

По большому счету, для того чтобы нарисовать изображение любой степени сложности, достаточно присвоить нужным пикселам необходимый цвет

(свойство холста Pixels) и в результате получить желаемый шедевр. Но это чрезмерно трудоемкий процесс, и им не стоит злоупотреблять. Тем более, в классе TCanvas инкапсулирован достаточно обширный набор методов рисования линий и простейших геометрических фигур.

Для того чтобы вывести простейшую прямую, понадобится два метода холста:

```
procedure MoveTo(X, Y: Integer);
procedure LineTo(X, Y: Integer);
```

Процедура MoveTo() позиционирует перо на начальную точку отрезка. Метод LineTo() рисует отрезок из текущей точки до конечной (причем конечная точка не закрашивается) и оставляет перо в этой точке. В представленном ниже примере благодаря перечисленным ранее методам поверхность формы покрывается вертикальными линиями с шагом в 5 пикселов.

```
var X:integer;
begin
X:= 0;
While X<Form1.ClientWidth do
   begin
   Form1.Canvas.MoveTo(X,0);
   Form1.Canvas.LineTo(X,Form1.ClientHeight);
   X:=X+5;
   end;
end;
```

Помимо прямых отрезков арсенал методов холста обеспечивает построение ломаных линий:

```
procedure Polyline(Points: array of TPoint);
```

Для изображения ломаной необходимо заполнить массив координат логических точек, через которые пройдет линия. Начальной и конечной точке ломаной линии соответствуют первый и последний элементы массива.

```
Form1.Canvas.Polyline([Point(10,10),Point(20,10),Point(40,50),Point(40,0)]);
```

Ближайший родственник ломаной линии – многоугольник.

```
procedure Polygon(Points: array of TPoint);
```

Отличие рассмотренного метода от процедуры Polyline() заключается в том, что последняя объявленная в массиве точка соединяется линией с первой и область внутри многоугольника закрашивается текущей кистью.

```
procedure Polygon(Points: array of TPoint);
var Arr : array [0..99] of TPoint;
begin
  Form1.Canvas.Brush.Style:=bsHorizontal;
  Form1.Canvas.Brush.Color:=clRed;
  Arr[0]:=Point(10,10); Arr[1]:=Point(20,10);
  //. . .
  Arr[99]:=Point(100,100);
  Form1.Canvas.Polygon(Arr);
end;
```

Если вы планируете вывести на экран не весь массив точек, а только его часть, то воспользуйтесь функцией `Slice()`*. Второй параметр функции ограничивает верхнюю границу массива.*

```
Form1.Canvas.Polygon(Slice(Arr, 10));
```

Кривые Безье

Об истории появления кривых Безье мы уже говорили в главе 6. Напомню, что для описания простейшего двухмерного сплайна Безье достаточно четырех точек. Две точки несут привычную нагрузку – задают стартовые и конечные координаты кривой. Изюминка заключается в роли оставшихся точек – они оттягивают кривую от прямой, соединяющей две крайние точки (см. рис. 6.5). От расположения отклоняющих точек зависит, где и каким образом кривая сменит направление. Для построения кривой Безье (кубического сплайна) воспользуйтесь методом:

```
procedure PolyBezier(const Points: array of TPoint);
```

Для задания первого сплайна необходимо четыре точки, каждому последующему сплайну достаточно трех новых точек, поскольку отправная точка новой кривой есть конечная точка предыдущей. Координатами конечных точек первой кривой служат первый и четвертый элементы массива `Points`. В качестве точек смещения выступают второй и третий элементы массива.

```
var pArr : array [1..4] of TPoint;
begin
//1 и 4 элементы массива - конечные точки кривой
pArr[1]:=Point(10,100); pArr[4]:=Point(200,100);
// 2 и 3 - отклоняющие точки
pArr[2]:=Point(100,0);  pArr[3]:=Point(400,200);
Form1.Canvas.PolyBezier(pArr);
end;
```

В арсенале холста имеется еще один метод, специализирующийся на построении сплайнов:

```
procedure PolyBezierTo(const Points: array of TPoint);
```

Единственное отличие от процедуры `PolyBezier()` заключается в том, что в качестве первой точки кривой Безье используется текущая позиция пера.

Разработчики GDI+ реализовали новый вид сплайна, его внешний вид также описывается четырьмя точками. Но в сплайне GDI+ усовершенствован способ определения формы кривой – теперь вместо того, чтобы оттягивать кривую от центральной оси, отклоняющие точки явным образом определяют ее траекторию.

Простейшие геометрические фигуры

Для рисования прямоугольника проще всего воспользоваться методом `Rectangle()`. Процедура является перегружаемой и в качестве параметров прини-

мает координаты верхнего левого (Left, Top) и нижнего правого (Right, Bottom) углов прямоугольника или координаты, заданные в формате TRect.

```
procedure Rectangle(Left, Top ,Right ,Bottom: Integer); overload;
procedure Rectangle(const Rect: TRect); overload;
```

Прямоугольник с более эстетичным контуром получится при использовании метода для рисования прямоугольников со скругленными углами:

```
procedure RoundRect(Left, Top ,Right ,Bottom, Width, Height: Integer);
```

Первые четыре параметра соответствуют координатам вершин функции Rectangle(). Значения Width и Height определяют степень сглаживания углов, задавая ширину и высоту скругляющего эллипса (см. рис. 7.7).

Вы уже заметили, что большинство методов, рисующих простейшие фигуры, тесно связаны с прямоугольником. Дело в том, что геометрические фигуры рисуются внутри ограничивающей прямоугольной области. Например, эллипс вписывается в рамки прямоугольника с координатами, аналогично процедуре Rectangle().

```
procedure Ellipse(Left, Top ,Right ,Bottom: Integer); overload;
procedure Ellipse(const Rect: TRect); overload;
```

Следующие три родственных метода посвящены прорисовке части эллипса:

```
procedure Arc  (Left, Top, Right, Bottom, X3, Y3, X4, Y4 : Integer);
procedure Chord(Left, Top, Right, Bottom, X3, Y3, X4, Y4 : Integer);
procedure Pie  (Left, Top, Right, Bottom, X3, Y3, X4, Y4 : Longint);
```

Процедура Arc() чертит сегмент, Chord() – хорду, Pie() – сектор эллипса. Как вы уже догадались, все изображения вписываются в эллипс (а точнее, в прямоугольник, содержащий эллипс) с координатами вершин (Left, Top) и (Right, Bottom). Логические координаты (X3, Y3) и (X4, Y4) описывают воображаемые лучи, исходящие из центра эллипса. Места пересечения лучей с контуром эллипса соединяются дугой. Начальной точкой дуги считается пересечение эллипса с лучом (X3, Y3), конечной – (X4, Y4).

Все перечисленные функции базируются на одноименных функциях Windows GDI. Их графическую интерпретацию вы найдете, возвратившись к рис. 7.7–7.9 этой книги.

Методы Rectangle(), Ellipse(), RoundRect(), Chord() *и* Pie() *предназначены не только для рисования простейших геометрических фигур, но и для закрашивания ограниченной этими фигурами области текущей кистью. По умолчанию к холсту подключается сплошная белая кисть.*

На этом методы для работы с прямоугольной областью не заканчиваются. Контур толщиной в 1 пиксел для области Rect нарисует процедура:

```
procedure FrameRect(const Rect: TRect);
```

Оконтуривание осуществляется не пером, а текущей кистью.

Эффект наличия фокуса ввода на прямоугольнике Rect обеспечивает метод:

```
procedure DrawFocusRect(const Rect: TRect);
```

При повторном вызове процедуры фокус ввода снимается, изображение приводится к первоначальному виду.

Обычное копирование изображения, находящегося в пределах прямоугольника Source из холста Canvas, произведет процедура:

```
procedure CopyRect(Dest: TRect; Canvas: TCanvas; Source: TRect);
```

Результат копирования помещается в прямоугольник Dest холста, вызвавшего процедуру. Изменяя размерность прямоугольника-получателя, можно получить эффект увеличения или уменьшения изображения.

Скопировать прямоугольник Source из битовой карты Bitmap в прямоугольник Dest поможет процедура:

```
procedure BrushCopy(const Dest: TRect; Bitmap: TBitmap;
                    const Source: TRect; Color: TColor);
```

Особенность метода в том, что при копировании цвет Color замещается текущим цветом кисти. Этот нюанс позволяет нарисовать прозрачную картинку. Для этого достаточно заменить фоновый цвет копируемого изображения на фоновый цвет холста.

```
var Bitmap: TBitmap; Rect1, Rect2: TRect; sFileName: string;
begin
  sFileName:= . . .;
  Rect1 := Rect(0,0,100,100); Rect2 := Rect(110,0,310, 200);
  Bitmap := TBitmap.Create;
  Bitmap.LoadFromFile(sFileName);
  Form1.Canvas.Brush.Color:=Form1.color;
  Form1.Canvas.BrushCopy(Rect1, Bitmap, Rect1, clWhite);
  Form1.Canvas.CopyRect(Rect2,Bitmap.Canvas,Rect1);
  Bitmap.Free;
end;
```

В листинге продемонстрирован пример использования функций BrushCopy() и CopyRect(). Метод BrushCopy() размещает на холсте растровую картинку из файла sFileName. Причем фоновый цвет битового образа Bitmap замещается фоновым цветом кисти холста. Метод CopyRect() копирует изображение в прямоугольник вдвое большего размера, тем самым достигается эффект увеличения.

Способ наложения изображения на холст определяется свойством CopyMode. *По умолчанию оно установлено в режим копирования «один в один» –* cmSrcCopy.

Заливка области

Заливку области Rect текущей кистью осуществляет метод:

```
procedure FillRect(const Rect: TRect);
```

Заливка текущей кистью любой замкнутой области осуществляется методом:

```
procedure FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle);
```

```
type TFillStyle = (fsSurface, fsBorder);
```

Заливка начинается из точки (X,Y). Режим заливки определяется значением параметра FillStyle. Если параметр принимает значение fsSurface, то заливка осуществляется до тех пор, пока вокруг точки (X,Y) не останется точек с цветом Color. В обратном режиме fsBorder заливка прекратится при выходе на границу с цветом Color. Приведенный ниже листинг демонстрирует способ закрашивания части окружности.

```
With Form1.Canvas do
begin
   Ellipse(10, 10, 100, 100);
   MoveTo(10, 10);
   LineTo(100, 100);
   Brush.Color:=clRed;
   FloodFill(30, 55, clBlack, fsBorder);
end;
```

В Delphi 2005 предложена удобная надстройка над функцией Windows GDI GradientFill(), позволяющей осуществлять градиентную заливку прямоугольной области:

```
procedure GradientFillCanvas(const ACanvas: TCanvas;
                             const AStartColor, AEndColor: TColor;
                             const ARect: TRect;
                             const Direction: TGradientDirection);
```

Функция требует указать холст ACanvas, на котором будет проводиться операция, стартовое AStartColor и конечное AEndColor значения цвета заливки, координаты обслуживаемой области ARect. Последний параметр функции Direction определяет направление заливки gdHorizontal (горизонтальное) либо gdVertical (вертикальное).

Следующая строка кода демонстрирует способ градиентной заливки клиентской области главной формы приложения:

```
GradientFillCanvas(Application.MainForm.Canvas,
                   RGB(0, 255, 255), RGB(255, 255, 255),
                   Application.MainForm.ClientRect, gdVertical);
```

По сравнению с GradientFill() наша функция Delphi значительно упрощена. Она не умеет закрашивать треугольные области и не в состоянии работать с несколькими фигурами одновременно.

Вывод текста

К разряду наиболее часто используемых методов для вывода текста относится процедура:

```
procedure TextOut(X, Y: Integer; const Text: string);
```

Параметрами X и Y определяется начальная позиция строки в логических координатах. В эту точку помещается левый верхний угол первого символа выводимой строки.

Несколько большими возможностями обладает метод:

```
procedure TextRect(Rect: TRect; X, Y: Integer; const Text: string);
```

В процедуру введен дополнительный параметр, задающий границы выводимого текста. Все, что не поместилось в границах прямоугольника Rect, безжалостно отсекается.

```
var Rct: TRect;
begin
Rct.Left:=10; Rct.Top:=10; Rct.Right:=58; Rct.Bottom:=25;
Form1.Canvas.TextRect(Rct,10,10,'Функция TextRect');
end;
```

Зачастую совместно с TextRect() используют информационные функции, возвращающие высоту и ширину строки в пикселах для текущего шрифта.

```
function TextHeight(const Text: string): Integer;
function TextWidth(const Text: string): Integer;
```

Вместо двух предыдущих методов можно использовать функцию:

```
function TextExtent(const Text: string): TSize;
```

С особенностями вывода текста на графическом устройстве тесно связано свойство холста:

```
property TextFlags: LongInt;
```

В свойстве инкапсулированы параметры функции Windows API ExtTextOut(). Возможные значения флагов приведены в табл. 20.1.

Вывод изображений

В наборе методов холста предусмотрен способ для вывода графических изображений, хранящихся в экземпляре класса TGraphic. Это могут быть растровые образы в формате BMP и JPEG, ресурс иконки ICO и векторные рисунки WMF и EMF. Методов всего два, в первую очередь, это процедура:

```
procedure Draw(X,Y: Integer; Graphic: TGraphic);
```

Метод отобразит картинку, хранящуюся в объекте Graphic. Координата левого верхнего угла области отображения определяется параметрами X и Y.

Более существенными возможностями обладает метод:

```
procedure StretchDraw(const Rect: TRect; Graphic: TGraphic);
```

Метод StretchDraw() рисует объект TGraphic в границах прямоугольной области Rect. Если размер исходного графического объекта не совпадает с размерами прямоугольника, то рисунок масштабируется – вписывается в заданную нами прямоугольную область.

В Delphi классом TGraphic *могут быть представлены различные растровые образы и метафайлы. Подробнее о возможностях* TGraphic *мы поговорим в следующей главе.*

Работа с холстом в многопоточном режиме

Иногда операции перерисовки одной и той же области инициируются несколькими программными потоками одновременно. Сложность подобной ситуации в том, что два (или более) потока в один и тот же промежуток времени могут попытаться провести с холстом абсолютно несовместимые операции, например один из потоков очищает область рисования, в то время как другой пытается выводить в ней текст. При этом затрачиваются значительные системные ресурсы, а пользователь в лучшем случае увидит мерцание. Задача программиста – научить потоки разделять холст между собой так, чтобы между ними не возникало конфликтных ситуаций.

Для выяснения, сколько раз был заблокирован холст, обращаемся к свойству:

```
property LockCount: Integer; //только для чтения
```

Если свойство возвратит нулевое значение, то это признак, что холст не захвачен ни одним из потоков.

Захват холста в монопольное пользование одним потоком обеспечивается двумя способами: процедурой:

```
procedure Lock;
```

или методом:

```
function TryLock: Boolean;
```

Отличие между методами в том, что метод `TryLock()` проинформирует программиста, смог он (значение `true`) или не смог (`false`) завладеть ресурсом. Сразу после захвата ресурса потоком получает приращение внутренний счетчик холста `LockCount`.

С этого момента времени поток получает право на монопольное использование холста. Завершив операции с холстом, поток должен его освободить. Для этого вызывается метод:

```
procedure Unlock;
```

В момент освобождения значение счетчика `LockCount` уменьшается на единицу.

РЕЗЮМЕ

Класс `TCanvas` инкапсулирует ключевые атрибуты и функции Windows GDI, обслуживающие контекст графического устройства. Спектр возможностей холста `TCanvas` вполне достаточен для проектирования графического интерфейса приложений малой степени сложности, но как только перед программистом ставятся нетривиальные задачи, ему необходимо обращаться за помощью к функциям Windows GDI. Ведь холст VCL практически не умеет управлять отсечением, не приспособлен для работы с траекториями и регионами, упрощенно относится к выводу текста, не владеет навыками работы с мировыми координатами. К сожалению, это далеко не полный перечень недостатков холста...

24

Растровые и векторные изображения в VCL

Нам уже известно, что основной формой представления растровых изображений в Windows GDI считаются аппаратно-зависимые (DDB) и аппаратно-независимые битовые образы (DIB). Вне зависимости от формата растрового рисунка, его размеров, глубины цвета работа с битовым образом средствами Windows API сопряжена с существенными сложностями. Подтверждение этому – разработанная нами в главе 16 этой книги функция `LoadBMP()`. Только взгляните на исходный код этого «монстра». Нам понадобилось несколько десятков строк кода для того, чтобы совершить, казалось бы, элементарное действие – получить дескриптор растрового рисунка из файла BMP!

Что-что, а реализация в VCL таких объектов GDI, как растровые и векторные изображения, просто выше всяких похвал. Программисты Borland предложили весьма элегантное решение задачи обслуживания растровых образов. Вне зависимости от характеристик изображения оно ведется экземпляром универсального класса `TBitmap`. Кроме класса `TBitmap` в этой главе мы познакомимся с классом `TIcon`, предназначенным для обслуживания ресурса иконки Windows, изучим класс `TMetafile` (он специализируется на работе с метафайлами), поговорим о формате JPEG и рассмотрим универсальное хранилище изображений – класс `TPicture`. Но традиционно изучение перечисленных объектов мы начнем с их предка – абстрактного класса `TGraphic` (рис. 24.1).

Класс TGraphic

В библиотеке компонентов Delphi класс `TGraphic` – едва ли не самый крупный специалист по поддержке основных графических форматов Windows. Он выступает в роли основополагающего абстрактного класса для таких графических объектов, как растровые изображения BMP, иконки, метафайлы и частная форма растрового изображения – сжатая растровая картинка JPEG. Вышеперечисленные объекты наследуют от `TGraphic` основные свойства и методы, например, обеспечивающие загрузку и сохранение изображений.

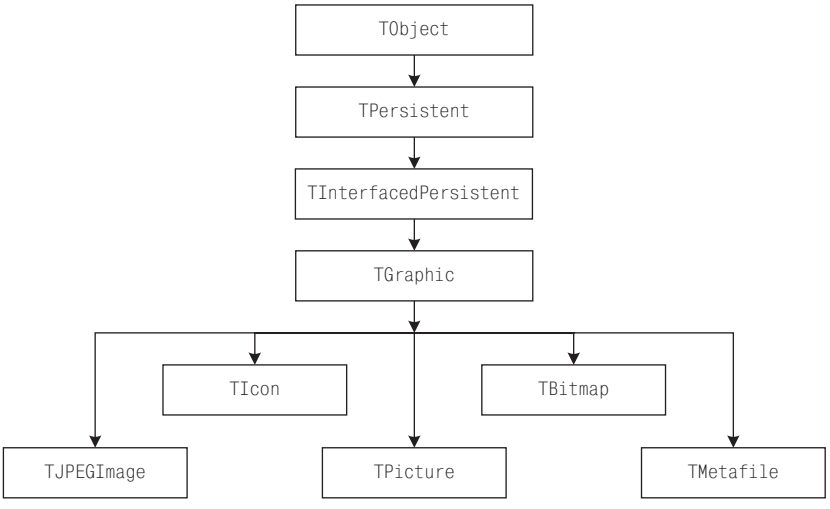


Рис. 24.1. Иерархия растровых и векторных изображений в VCL

Класс TGraphic поддерживает операции по работе с потоком, файлом и буфером обмена Windows. Для этого он оснащен группой методов загрузки (их названия начинаются со слова Load) и сохранения (Save). Для работы с потоком в памяти предназначены процедуры:

```
procedure LoadFromStream(Stream: TStream);
procedure SaveToStream(Stream: TStream);
```

С файлом:

```
procedure LoadFromFile(const FileName: string);
procedure SaveToFile(const FileName: string);
```

С буфером обмена Windows:

```
procedure LoadFromClipboardFormat(AFormat: Word; AData: THandle;
                                  APalette: HPALETTE);
procedure SaveToClipboardFormat(var AFormat: Word; var AData: THandle;
                                var APalette: HPALETTE);
```

В качестве параметров при работе с буфером обмена выступают: AFormat – зарегистрированный графический формат буфера обмена Windows; дескрипторы: AData – данных; APalette – цветовой палитры.

Загрузка графических объектов больших размеров занимает много времени. В классе TGraphic предусмотрена возможность информирования пользователя о процессе загрузки данных. Для этого реализовано событие:

```
property OnProgress: TProgressEvent;
TProgressEvent = procedure (Sender: TObject; Stage: TProgressStage;
                            PercentDone: Byte; RedrawNow: Boolean;
                            const Rct: TRect;
```

```
                                const Msg: string) of object;
type TProgressStage = (psStarting, psRunning, psEnding);
```

Здесь Sender – элемент-источник сообщения; Stage – сигнализирует о стадии процесса (psStarting – старт, psRunning – процесс в работе, psEnding – завершение); PercentDone – приблизительный процент выполненной работы; RedrawNow – состояние прорисовки изображения; Rct – прямоугольник, содержащий координаты области изображения, нуждающегося в прорисовке; Msg – может содержать текстовое сообщение, описывающее текущую операцию.

```
procedure TForm1.Image1Progress(Sender: TObject; Stage: TProgressStage;
          PercentDone: Byte; RedrawNow: Boolean; const R: TRect;
          const Msg: String);
begin
case Stage of
     psStarting: frmProgress.Show;
     psRunning : frmProgress.ProgressBar1.Position:=PercentDone;
     psEnding  : frmProgress.Close;
end;
end;
```

В листинге приведен пример использования события OnProgress() совместно с компонентом-шкалой TProgressBar. Состоянием параметра Stage определяется поведение формы frmProgress (показ/закрытие).

Не все графические объекты способны использовать обработчик события OnProgress(). *Такой возможностью обладает формат JPEG.*

Свойство:

```
property Empty: Boolean;
```

принимает значение true, если графический объект не содержит данных.

Свойство, показывающее, изменялся ли данный графический объект:

```
property Modified: Boolean;
```

Дескриптор цветовой палитры графического изображения находится в свойстве:

```
property Palette: HPALETTE;
```

Если изображение не использует палитру, то значение свойства равно 0.

При изменении цветовой палитры изменяется состояние свойства:

```
property PaletteModified: Boolean;
```

За прозрачность графического объекта отвечает свойство:

```
property Transparent: Boolean;
```

Размеры (в пикселах) графического объекта по вертикали и горизонтали:

```
property Height: Integer;
property Width: Integer;
```

В некоторых случаях для изменения геометрических размеров можно использовать процедуру:

```
procedure SetSize(AWidth: Integer; AHeight: Integer);
```

Здесь AWidth – ширина; AHeight – высота изображения.

Для полиморфного присваивания графических объектов воспользуйтесь методом:

```
procedure Assign(Source: TPersistent);
```

Иконка – класс TIcon

Класс инкапсулирует иконку. Иконка представляет один из наиболее «древних» способов представления рисунков в Windows. Объект-иконка загружается из файла с расширением «ico» или другого ресурса. Набор свойств и методов класса невелик. Доступ к дескриптору ресурса осуществляется через свойство:

```
property Handle: HIcon;
```

Высоту и ширину пиктограммы позволят уточнить свойства:

```
property Height: Integer;
property Width: Integer;
```

Они доступны только для чтения.

Иконка всегда прозрачна, свойство:

```
property Transparent: Boolean;
```

всегда установлено в true.

Метод:

```
function ReleaseHandle: HIcon;
```

уничтожает связанный с иконкой объект GDI и устанавливает ссылку на него в NIL.

Класс TIcon, как наследник TGraphic, поддерживает операции по работе с потоком и файлом, но не воспринимает буфер обмена Windows.

В арсенале Windows API предусмотрена возможность извлечения иконки, ассоциированной с файлом. Для этого стоит обратить внимание на функцию:

```
function ExtractAssociatedIcon(HINSTANCE:hInst; Path : PChar;
                               lpiIcon : Word):HIcon;
```

Она ищет иконку в файле, указанном в параметре Path. Если в теле файла иконка не обнаружена, то поиск продолжается в приложении, обслуживающем данный тип файла. В случае успеха функция возвращает дескриптор иконки. Ниже приведен листинг с примером использования функции ExtractAssociatedIcon(). Если вы хотите использовать пример на практике, то не забудьте подключить к проекту модуль ShellAPI.

```
uses ShellAPI;

function TForm1.GetIcon(FileName: String): TIcon;
var i:word;
begin
if FileAge(FileName)>-1 then
begin
   Result:=TIcon.create;
   Result.Handle:=ExtractAssociatedIcon(application.handle, PChar(FileName), i);
end
ELSE Result:=Nil;
end;
```

В классе TIcon не предусмотрено методов, позволяющих создавать изображение иконки во время выполнения программы. Скорее всего, это объясняется тем, что в своих проектах мы пользуемся набором заранее подготовленных стандартных пиктограмм. Поэтому в состав TIcon разработчики решили не включать метод Windows API:

```
function CreateIconIndirect(var IconInfo: TIconInfo): HICON;
```

Единственный параметр функции – структура, которая может использоваться для описания иконки или курсора:

```
type  TIconInfo = packed record
    fIcon: BOOL;{true - для создания иконки,
                 false - для создания курсора}
    xHotspot: DWORD;//для иконки игнорируется
    yHotspot: DWORD; //для иконки игнорируется
    hbmMask : HBITMAP;//дескриптор маски иконки
    hbmColor: HBITMAP;//дескриптор цветовой таблицы
  end;
```

Перед обращением к функции CreateIconIndirect() структура заполняется данными. Кроме того, следует убедиться, что битовые образы hbmMask и hbmColor не были выбраны в контекст, иначе функция не сможет создать ресурс иконки.

Растровое изображение – класс TBitmap

Класс TBitmap нацелен на обслуживание растровой графики. На практике битовые образы используются для хранения высококачественных изображений (фотографии и кадры видео). Еще одно преимущество растровых изображений – в высокой скорости вывода рисунка на экран. За хорошее качество приходится платить существенным объемом используемой памяти. Потребности в памяти возрастают в прямой пропорции с геометрическими размерами изображения и глубиной цвета. Другой недостаток растровых образов заключается в сложности качественного масштабирования картинки. Любое изменение размера битового образа требует либо дублирования строк и столбцов, либо их удаления, что неблагоприятным образом сказывается на качестве итоговой картинки.

Еще со времен первых версий Windows в ней поддерживается два формата хранения растровых изображений – аппаратно-зависимый битовый образ (DDB) и аппаратно-независимый битовый образ (DIB). Индикатором формата образа служит свойство:

```
property HandleType: TBitmapHandleType;
type TBitmapHandleType = (bmDIB, bmDDB);
```

Расцвет формата DDB давно прошел, в старозаветные времена он применялся в первых версиях Windows. Недостаток аппаратно-зависимой битовой карты заключается в сложности вывода изображения на графических устройствах с иной цветовой организацией. Сегодня DDB обычно используется для обеспечения механизма двойной буферизации – это тот случай, когда для устранения мерцания изображение перед выводом на экран формируется в памяти компьютера. Формат DIB сохраняет свою жизнеспособность на любом графическом устройстве, так как он содержит дополнительную служебную информацию о своих цветовых характеристиках. Чаще всего файлы в формате DIB имеют расширение .bmp, реже – .dib и .rle.

Черно-белый битовый образ для хранения информации об одном пикселе требует всего один бит. Для выяснения, не является ли наш объект монохромным, обратитесь к свойству:

```
property Monochrome: Boolean;
```

На рис. 15.1 главы, посвященной образам DDB, изображен монохромный растровый рисунок размерностью 8×8 пикселов. Эту картинку можно представить в виде рядов битов (скан-линий). Указатель на необходимую сканлинию предоставит свойство:

```
property ScanLine[Row: Integer]: Pointer;
```

В скан-линии черному пикселу соответствует нулевое значение, белому пикселу – единица. В памяти строки битовой карты DIB обычно упорядочиваются снизу вверх.

Хранение цветного изображения несколько сложнее и требует дополнительных бит. Число цветов, представляемых в битовом образе, соответствует 2 в степени «число бит на пиксел»: 16 цветов – 4 бита на пиксел, 256 – 8, High Color – 15 или 16, True Color – 24 или 32 бита на пиксел.

Глубина цветов битового образа определяется состоянием свойства:

```
property PixelFormat: TPixelFormat;
type TPixelFormat = (pfDevice, pf1bit, pf4bit, pf8bit, pf15bit,
                     pf16bit, pf24bit, pf32bit, pfCustom);
```

Формат `pfCustom` предназначен для реализации программистом собственных графических объектов – потомков класса `TBitmap`.

Как видно из рис. 24.1, класс `TBitmap` построен на основе класса `TGraphic` и унаследовал все объявленные в предке свойства и методы. `TBitmap` инкапсулирует битовый образ Windows, включая палитру. Поэтому объект `TBitmap` обладает двумя дескрипторами (образа и цветовой палитры):

```
property Handle: HBitmap;
property Palette: HPalette;
```

Дескриптор битового образа используется для доступа к функциям GDI. Дескриптор палитры предназначен для чтения или изменения цветовой палитры.

Два очень похожих метода отключают от TBitmap объекты GDI:

```
function ReleaseHandle: HBitmap;   //отключение от растрового образа
function ReleasePalette: HPalette; //отключение от палитры
```

Функции используются для передачи дескриптора какому-то другому объекту. После этого экземпляр TBitmap обнуляет дескриптор. Возможность разделения дескриптора между объектами объясняется наличием механизма кэширования. Но как только вы вновь обратитесь к дескриптору битовой карты, разделение одной картинки между несколькими объектами заканчивается.

Процедура:

```
procedure Dormant;
```

создает в памяти растровый рисунок в формате DIB, уничтожая разделяемые дескрипторы предыдущей битовой карты.

Процедура:

```
procedure FreeImage;
```

освобождает занятую битовым образом память.

Помимо унаследованных от класса TGraphic возможностей загрузки изображения из файла, потока и буфера обмена класс TBitmap поддерживает загрузку из ресурса.

```
procedure LoadFromResourceID(Instance: THandle; ResID: Integer);
procedure LoadFromResourceName(Instance: THandle; const ResName: string);
```

Параметр Instance хранит идентификатор ресурса. Выбор битового образа из ресурса осуществляется при помощи параметра ResID (идентификатор ресурса) или ResName (имя элемента ресурса).

Находясь в 256-цветовом режиме, можно отказаться от реализации цветовой палитры битовым образом (заполнении системной палитры цветами из палитры рисунка). В этом случае значительно ускоряется прорисовка изображения, но искажаются цвета результирующей картинки:

```
property IgnorePalette: Boolean;
```

Растровый рисунок вполне можно сделать прозрачным. Для этого нужно указать цвет фона битовой карты:

```
property TransparentColor: TColor;
```

и установить в состояние tmFixed свойство:

```
property TransparentMode: TTransparentMode;
type TTransparentMode = (tmAuto, tmFixed);
```

Если свойство Transparent установлено в автоматический режим (tmAuto), то за фоновый цвет битовой карты будет приниматься цвет ее левого верхнего пиксела.

Для использования имеющегося растрового образа в качестве маски для других битовых образов предусмотрено свойство:

```
property MaskHandle: HBitmap;
```

В нем вы обнаружите дескриптор маски.

Для создания маски из обычного образа применяют метод:

```
procedure Mask(TransparentColor: TColor);
```

Метод конвертирует битовую карту в монохромную картинку, заменив TransparentColor в белый цвет, а все остальные цвета – в черный.

Отключение образа-маски производится методом:

```
function ReleaseMaskHandle(): HBITMAP;
```

Продемонстрируем возможности класса TBitmap на практике. Для этого разработаем программу, позволяющую собрать все битовые образы из определенного пользователем каталога и отобразить их в уменьшенном виде. Программирование начнем с декларации в секции частных объявлений проекта списка:

```
private
    FileList:TStringList;
```

Список FileList предназначен для хранения имен файлов *.bmp.

Позаботьтесь о своевременном создании и разрушении списка. Для этого нам потребуются события OnCreate() и OnDestroy() главной формы проекта.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
FileList:=TStringList.Create; //создание экземпляра списка
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
FileList.Free; //уничтожение экземпляра списка
end;
```

Следующим этапом нашей работы станет описание процесса сбора файлов растровых изображений. Для этого разместите в нижней части формы строку ввода Edit1 (она предназначена для ввода пути к папке с рисунками) и кнопку Button1, выдающую команду на начало сбора. Событие OnClick() по кнопке представлено ниже:

```
procedure TForm1.Button1Click(Sender: TObject);
var SR:TSearchRec;
begin
FileList.Clear; //очистка содержимого списка
  If FindFirst(Edit1.Text+'\*.bmp', faAnyFile ,SR)=0 then {цикл сбора файлов}
```

```
  Repeat
    FileList.Add(SR.Name); //заносим в список название файла
  Until FindNext(SR)<>0; //поиск продолжается, пока в папке имеются файлы
FindClose(SR); //поиск завершен
Form1.Repaint; //команда на перерисовку формы
end;
```

Наконец, мы дошли до кульминационного момента – описания события прорисовки формы `OnPaint()`**.**

```
procedure TForm1.FormPaint(Sender: TObject);
var BMP:TBitmap; i:integer; Rct:TRect;
begin
if FileList.Count>0 then {если в списке имеются файлы}
begin
   Rct:=RECT(1,1,51,51); {координаты области вывода первой пиктограммы}
   BMP:=TBitmap.Create;  {создание экземпляра класса TBitmap}
   i:=0;
  WHILE (i<FileList.Count-1) and (Rct.Bottom<Form1.ClientHeight) do
  Begin {цикл продолжается, пока в списке есть файлы
         и не достигнута нижняя граница клиентской области формы}
  if Rct.Right>Form1.ClientWidth then //перенос области вывода на новую строку
     begin
     Rct.Left:=1; Rct.Top:=Rct.Bottom+1;
     Rct.Right:=51; Rct.Bottom:=Rct.Bottom+51;
     end;
    BMP.LoadFromFile(Edit1.text+'\'+FileList.Strings[i]); //загрузка файла
    form1.Canvas.StretchDraw(Rct, BMP); //вывод уменьшенной пиктограммы
    OffsetRect(Rct,52,0);  //сдвиг области вывода по горизонтали
    INC(i); //приращение счетчика файлов
  end;
BMP.Free; //уничтожение экземпляра TBitmap
end;
end;
```

Проект завершен (рис. 24.2). Если вы повторили его безошибочно, то нажимайте на кнопку «F9».

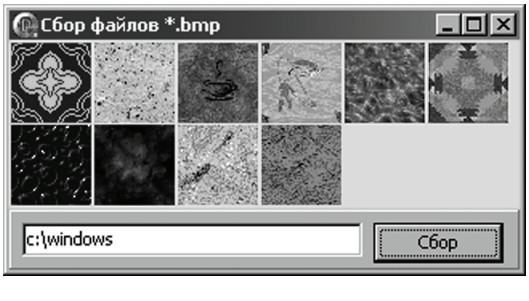


Рис. 24.2. Проект сбора и отображения изображений в формате BMP

Физическое масштабирование битового образа

На страницах этой книги мы уже обсуждали сложность решения задач, связанных с качественным масштабированием растровых изображений. Одно из готовых решений предлагает Delphi, это процедура:

```
procedure ScaleImage(const SourceBitmap, ResizedBitmap: TBitmap;
                     const ScaleAmount: Double);
```

В процедуру направляются три параметра: исходное изображение `SourceBitmap`, коэффициент масштабирования `ScaleAmount`, результат операции помещается в `ResizedBitmap`.

```
var SourceBitmap, ResizedBitmap: TBitmap;
begin
 SourceBitmap:=TBitmap.Create;
 SourceBitmap.LoadFromFile('demo.bmp');
 ResizedBitmap:=TBitmap.Create;
 GraphUtil.ScaleImage(SourceBitmap, ResizedBitmap, 2.5);
 Form1.Canvas.Draw(10,10,SourceBitmap);
 Form1.Canvas.Draw(10,10+SourceBitmap.Height,ResizedBitmap);
 SourceBitmap.Free;
 ResizedBitmap.Free;
end;
```

Метафайл – класс TMetafile

В отличие от представителя растровой графики `TBitmap` класс `TMetafile` (метафайл) предназначен для обслуживания векторных форматов изображений. Это файлы формата WMF и EMF, содержащие не массив пикселов, а набор вызовов графических функций GDI. Благодаря такой особенности метафайла пропадает необходимость хранить данные, описывающие каждый пиксел. Функции, рисующие линии, кривые, геометрические фигуры, с прорисовкой пикселов справятся самостоятельно. Обычно рисунки, построенные на основе метафайлов, называют векторной графикой и широко используют в программах инженерной графики.

Преимущества векторной графики над растровой не требуют особых комментариев: масштабирование без потери качества и малые размеры требуемой памяти. Единственное, в чем метафайл не особо силен, – построение изображений фотографического качества и скорость вывода изображения. Теоретически метафайл способен справиться и с описанием растрового изображения, но вряд ли найдутся желающие конвертировать битовые образы в набор функций GDI.

В Windows широко применяются два формата метафайлов: устаревший WMF и более современная версия – расширенный метафайл EMF. По умолчанию класс `TMetafile` нацелен на работу с расширенным метафайлом. Для обеспечения совместимости с устаревшим форматом WMF опубликовано свойство:

```
property Enhanced: Boolean; //состояние true - EMF, false - WMF
```

Главное усовершенствование формата EMF заключается в обширном информационном заголовке файла и поддержке им новых функций и структур данных.

Низкоуровневый доступ к функциям GDI реализуется через дескрипторы метафайла и его палитры:

```
property Handle: HMetafile;
property Palette: HPalette;
```

Целый набор свойств связан с геометрическими размерами метафайла. Экранные размеры в пикселах:

```
property Height: Integer;
property Width: Integer;
```

Размеры в сотых долях миллиметра (0,01 мм):

```
property MMHeight: Integer;
property MMWidth: Integer;
```

Число точек на дюйм в координатной системе метафайла:

```
property Inch: Word;
```

Информацию о создателе и краткое описание метафайла можно получить из свойств (только для чтения):

```
property CreatedBy: string;
property Description: string;
```

Объект TMetafile предназначен для вывода изображения на графическом устройстве. А для того чтобы нарисовать метафайл, необходимо использовать возможности класса TMetafileCanvas.

Класс TMetafileCanvas

Холст метафайла – это потомок уже известного нам класса TCanvas. TMetafileCanvas в сотрудничестве с TMetafile позволит нам создавать новые рисунки векторной графики. Перечень свойств и методов практически на 100% соответствует возможностям родительского класса TCanvas, но за некоторым исключением.

У TMetafileCanvas переопределен конструктор:

```
constructor Create(AMetafile: TMetafile; ReferenceDevice: HDC);
```

Конструктор создает экземпляр холста метафайла и выделяет ему дескриптор. Параметр AMetafile соответствует объекту TMetafile, ReferenceDevice – дескриптор графического устройства.

```
procedure NewMetafile;
var mtf:TMetaFile;
begin
  mtf:=TMetaFile.Create;
  with TMetaFileCanvas.Create(mtf,0) do
  begin
   Brush.Color := clRed;
```

```
    Ellipse(0,0,100,100);
    Brush.Color := clBlue;
    Rectangle(25,25,50,50);
    Brush.Color:=clWhite;
    TextOut(15,100,'Метафайл');
    Destroy;
   end;
    Form1.Canvas.Draw(0,0,mtf);
    mtf.SaveToFile('C:\Metafile.emf');
  mtf.free;
end;
```

Предложенная в листинге процедура `NewMetafile` создает векторный рисунок из трех объектов: красного эллипса, синего квадрата и текстовой надписи. Затем сохраняет метафайл в формате EMF.

Еще одно дополнение класса `TMetafileCanvas` – конструктор, отвечающий за внесение в метафайл справочной текстовой информации (вспомните свойства класса `TMetafile` – `CreatedBy` и `Description`):

```
constructor CreateWithComment(AMetafile: TMetafile; ReferenceDevice: HDC;
                              const CreatedBy, Description: String);
```

Класс TJPEGImage

Класс `TJPEGImage` инкапсулирует возможности популярного в цифровой фотографии формата сжатой растровой картинки, называемого JPEG (Joint Photographic Expert Group). Формат JPEG был специально разработан для хранения 24-битных изображений. Исключительное достоинство JPEG заключается в том, что ему практически нет равных (за исключением формата PNG) в искусстве сжатия фотографических изображений. Стандарт предусматривает 4 режима сжатия: иерархический, прогрессивный, последовательный и сжатие без потерь. Простейший из них последовательный, сложнейший – иерархический.

К особенностям класса `TJPEGImage` следует отнести:

- отсутствие у него холста (хотя объект `TJPEGImage` может нарисовать себя на холсте других объектов);
- класс не обеспечивает доступа к отдельному пикселу своего изображения;
- благодаря объекту `TJPEGData` допускает совместное использование своего дескриптора.

Экземпляр класса `TJPEGImage` *инкапсулирует еще один объект – экземпляр класса* `TJPEGData`*. Объект* `TJPEGData` *создается автоматически и заведует всей внутренней стороной кухни хранения и обслуживания растрового образа в формате JPEG.*

Объект `TJPEGImage` обязан не только сжимать растровые изображения, но и отображать распакованные данные в формате DIB. Самый простой вариант взаимодействия между сжатой и обычной растровой картинкой осуществляется при помощи метода:

```
procedure Assign(Source: TPersistent);
```

Для поддержки проектом Delphi файлов в формате JPEG подключите к нему одноименный модуль.

В таком случае производится автоматическое конвертирование форматов: из DIB в JPEG или наоборот.

Для преобразования рисунка в формате JPEG в DIB предназначен метод:

```
procedure DIBNeeded;
```

Обычно процедура используется перед вызовом метода `Draw()` холста. Такой подход ускоряет прорисовку картинки.

Обратное преобразование (из DIB в JPEG) осуществляется методом:

```
procedure JPEGNeeded;
```

В результате создается новый экземпляр класса `TJPEGData`, содержащий конвертированные в JPEG данные.

Для сжатия обычного растрового изображения в формат JPEG прежде всего надо установить степень упаковки. Для этого в классе `TJPEGImage` опубликовано свойство:

```
type TJPEGQualityRange = 1..100;
property CompressionQuality: TJPEGQualityRange;
```

Чем больше значение `CompressionQuality`, тем меньше потерь качества изображения, но больше размер результирующего файла. С уменьшением значения `CompressionQuality` степень сжатия возрастает, но ухудшается рисунок. По умолчанию значение качества сжатия равно 90.

Метод, осуществляющий сжатие растровой картинки, называется

```
procedure Compress;
```

Эта процедура автоматически вызывается при сохранении объекта JPEG в файл и поток.

Используемые в формате JPEG методы сжатия относятся к технологиям сжатия с потерями. Поэтому формат JPEG не стоит использовать в качестве промежуточного способа хранения изображения, в особенности, когда приходится многократно редактировать и сохранять рисунок.

Два взаимосвязанных свойства определяют способ вывода изображения на экран – по мере распаковки (такой метод называют прогрессивной разверткой) или только по окончанию распаковки. Вывод по мере распаковки реализуется за счет того, что изображение JPEG разбивается на множество сканов. Начальные сканы являются грубым представлением картинки, но каждый последующий скан улучшает качество изображения.

```
property ProgressiveEncoding: Boolean;
property ProgressiveDisplay: Boolean;
```

Прогрессивная развертка картинки (ProgressiveDisplay = true) возможна только при условии, что файл сохранен в режиме прогрессивного сжатия (ProgressiveEncoding = true).

В том случае, если вы используете прогрессивную развертку, допустимо сглаживание изображения:

```
property Smoothing: Boolean;
```

Качество изображения зависит не только от степени сжатия растровой картинки. Ряд свойств класса определяет работу TJPEGImage при декомпрессии и показе изображения.

Критерий скорость распаковки/качество изображения задается в свойстве:

```
property Performance: TJPEGPerformance;
type TJPEGPerformance = (jpBestQuality, jpBestSpeed);
```

Выбрав значение jpBestQuality, мы затратим несколько больше времени на распаковку файла, но получим более качественную картинку. Выбрав значение jpBestSpeed, получим обратный результат. На качество сохраняемого изображения Performance не влияет.

Критерий скорость вывода изображения/глубина цветов определяется свойством:

```
property PixelFormat: TJPEGPixelFormat;
type TJPEGPixelFormat = (jf24Bit, jf8Bit);
```

При выборе значения jf24Bit вы получите полнокровное 24-битное изображение, но на его вывод затратите значительно больше времени, чем на 8-битное.

Наряду с уже известным нам методом TCanvas StretchDraw(), позволяющим масштабировать графические объекты (потомки класса TGraphic), класс TJPEGImage поддерживает свой, более эффективный (быстрый) способ масштабирования:

```
property Scale: TJPEGScale;
type TJPEGScale = (jsFullSize, jsHalf, jsQuarter, jsEighth);
```

В соответствии со значениями TJPEGScale вы получите изображение в соотношении 1:1, 1:2, 1:4 и 1:8 к размерам исходной картинки.

Хранилище изображения – класс TPicture

Класс TPicture служит классическим примером инкапсуляции и полиморфизма. Экземпляр класса представляет собой универсальное хранилище изображения, способное содержать любой из рассмотренных в этой главе графических объектов: TIcon, TBitmap, TMetafile и TJPEGImage. Более того, TPicture превосходно справляется с решением вопросов по вызову методов, характерных для конкретного потомка TGraphic, тем самым упрощая работу программиста.

Обращение к графическому ресурсу реализуется при помощи свойства:

```
property Graphic: TGraphic;
```

В дополнение к этому доступ (в зависимости от типа объекта) осуществляется через свойства:

```
property Icon: TIcon;
property Bitmap: TBitmap;
property Metafile: TMetafile;
```

Методы, связанные с загрузкой/сохранением графического объекта, перед выполнением анализируют тип графического объекта, а затем вызывают соответствующие этим объектам методы.

```
procedure LoadFromFile(const FileName: string);
procedure SaveToFile(const FileName: string);
procedure LoadFromClipboardFormat(aFormat: Word; aData: THandle;
                                  APalette: HPALETTE);
procedure SaveToClipboardFormat(var aFormat: Word; var aData: THandle;
                                var APalette: HPALETTE);
```

Здесь `aFormat` – константа, описывающая формат данных, находящихся в буфере обмена. Например, именованная константа `CF_BITMAP` укажет, что мы планируем получить (передать) обычное растровое изображение. Параметр `aData` – собственно указатель на данные, `aPalette` – дескриптор палитры.

При разработке новых графических классов на базе `TGraphic` методы класса

```
type TGraphicClass = class of TGraphic;
class procedure RegisterFileFormatRes(const AExtension: String;
                ADescriptionResID: Integer; AGraphicClass: TGraphicClass);
class procedure RegisterFileFormat(const AExtension, ADescription: string;
                                   AGraphicClass: TGraphicClass);
class procedure RegisterClipboardFormat(AFormat: Word;
                                        AGraphicClass: TGraphicClass);
```

позволят зарегистрировать новые форматы ресурса, файла и буфера обмена.

Обратная операция осуществляется при помощи метода:

```
class procedure UnregisterGraphicClass(AClass: TGraphicClass);
```

Процедура отменяет все действия по регистрации нового графического класса, осуществленные при помощи методов `RegisterFileFormat()`, `RegisterFileFormatRes()` и `RegisterClipboardFormat()`.

РЕЗЮМЕ

В состав VCL среды Delphi включены классы, нацеленные на обслуживание наиболее популярных в среде Windows форматов растровой и векторной графики. Так, для работы с битовыми образами BMP предназначен класс `TBitmap`, с ресурсом иконки лучше всех справится `TIcon`, работу со сжатыми растровыми изображениями JPEG обеспечит класс `TJPEGImage`, на векторных форматах WMF и EMF специализируется `TMetaFile`. В библиотеке компонентов имеется универсальный класс `TPicture`, способный обрабатывать все перечисленные выше графические форматы.

25

Коллекционируем изображения

Для того чтобы интерфейс приложения приобрел привлекательность и стал интуитивно понятен, разработчики программного обеспечения большинству элементов управления сопоставляют изображение-подсказку. Подготовленный пользователь уже знает, что если на кнопке нарисована дискета, то этот элемент управления отвечает за сохранение данных в файл. Увидев пиктограмму с рисунком ножниц, мы догадываемся, что элемент управления отвечает за операцию «вырезать в буфер обмена». Картинка с изображением принтера сигнализирует о готовности программы распечатать документ. В современных программных продуктах один и тот же результат достигается абсолютно разной последовательностью действий. Например, для копирования текстового фрагмента в буфер обмена пользователь может воспользоваться кнопкой с панели управления приложением, обратиться к элементам главного или контекстного меню. Для наибольшей наглядности все эти элементы помечаются одинаковой пиктограммой. С этой целью, начиная с первых версий Delphi, ключевые компоненты оснащались полями-контейнерами, способными хранить растровые изображения. Доступ к загруженным в компонент рисункам производился с помощью специальных свойств. Так кнопки `TBitBtn` и `TSpeedButton` оборудованы свойством:

```
property Glyph: TBitmap;
```

Элемент меню `TMenuItem` снабжен свойством:

```
property Bitmap: TBitmap;
```

Все просто и понятно. Находим подходящую картинку и подключаем ее к компоненту, в результате на его поверхности «проявляется» пиктограмма. И все бы хорошо, но, к сожалению, в этом случае простое решение не стало наиболее удачным. Его недостаток на поверхности. Допустим, что одну и ту же операцию (например, копирование в буфер) осуществляют три различных элемента управления. Начинающий программист, не задумываясь о ресурсах приложения, смело загружал три копии одной и той же картинки в каждый из компонентов. Теперь подсчитайте. Один растровый файл разме-

рами 16×16 пикселов с 8-битной глубиной цвета занимает примерно 1,33 килобайта. Три его копии – около 4 килобайт. И эти килобайты (66% из которых лишние) мы насильно включаем в исполняемый файл. Скажете: «4 килобайта – это пустяк». Может быть. А что будет, если наше приложение содержит сотню-другую пиктограмм?

Проанализировав этот недостаток, программисты предложили весьма элегантный выход из ситуации. Идея в следующем: создать универсальный контейнер, специализирующийся на хранении растровых картинок с возможностью предоставления своего содержимого другим элементам управления проекта.

Контейнер изображений, класс TImageList

Герой этой главы – невизуальный элемент управления, предназначенный для хранения графических образов и обслуживания пиктограмм приложения – компонент `TImageList`. Класс описан в модулях `Controls` и `ImgList`, компонент проживает на линейке Win32 палитры компонентов Delphi. Иерархия наследования класса представлена на рис. 25.1. На роль рисунков могут претендовать только одинаковые по габаритам растровые файлы bmp и иконки ico. Контейнер изображений обладает удобным интерфейсом, позволяющим предоставлять доступ к загруженным в него рисункам другим элементам управления.

Контейнер предъявляет ряд требований к обслуживаемым изображениям. В первую очередь, это предельные геометрические размеры рисунков. По умолчанию это картинка 16×16 пикселов. Размеры устанавливаются парой свойств:

```
property Height: Integer; //высота
property Width: Integer;  //ширина
```

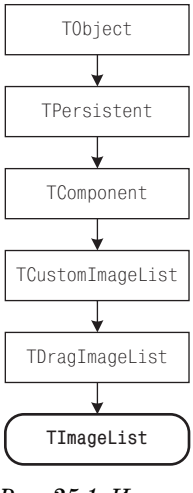


Рис. 25.1. Иерархия наследования TImageList

Если размеры рисунка превосходят значения, указанные в свойствах `Height` и `Width`, компонент предложит на выбор два варианта развития событий: разбиение изображения на несколько частей или сжатие его до приемлемых границ. Строгость в подходе к габаритам пиктограмм объясняется особенностями обслуживания переданных в `TImageList` изображений. Технически загруженные в элемент управления рисунки хранятся не по отдельности, а в виде единого блока (рис. 25.2). Другими словами, все заносимые в контейнер картинки склеиваются (на манер лоскутного одеяла) в единый растровый образ. Когда программист обращается к пиктограмме по ее индексу, компонент (зная, что все ресурсы обладают одинаковыми размерами) быстро находит в матрице область требуемого рисунка и возвращает соответствующие ему пикселы.

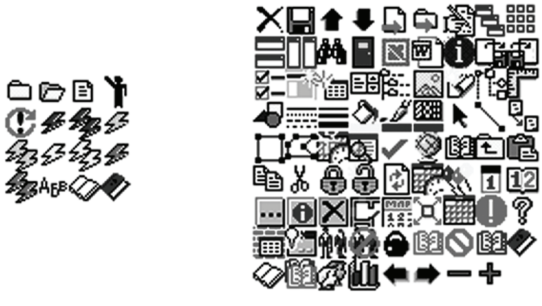


Рис. 25.2. Хранение пиктограмм в TImageList

Кроме собственно цветных изображений контейнер может содержать и аналогичное число масок (монохромных битовых образов). Благодаря маске мы получаем возможность вывода на экран пиктограммы с эффектом прозрачности – маска складывается с основным рисунком, причем черные пикселы маски заменяются соответствующим цветом поверхности.

Для определения типа списка стоит обратиться к свойству:

```
property ImageType: TImageType; //по умолчанию itImage
type TImageType = (itImage, itMask);
```

Значение `itImage` означает, что список изображений не содержит маски; `itMask` – список изображений хранит маски.

Загрузка растровых образов в контейнер

Средой Delphi предусмотрено два способа загрузки рисунков в контейнер `TImageList`. Во время разработки программы нам на помощь приходит интегрированный редактор компонента. Если же возникла необходимость управления содержимым компонента во время выполнения программы, то нам не обойтись без поддержки методов элемента управления.

Управление списком изображений в период разработки программы

Простейшим способом заполнения списка изображений пиктограммами считается использование встроенного редактора. Для вызова последнего достаточно обратиться к контекстному меню компонента либо дважды щелкнуть по нему кнопкой мыши. Интерфейс редактора интуитивно понятен (рис. 25.3).

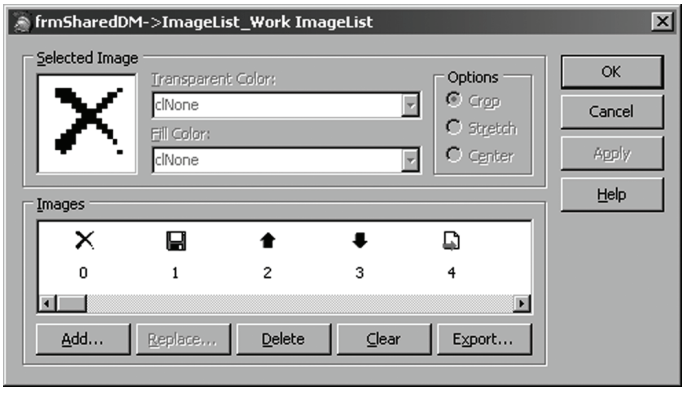


Рис. 25.3. Встроенный редактор компонента TImageList

В нижней части окна расположены кнопки, позволяющие добавлять (Add...), заменять (Replace), удалять (Delete) пиктограммы. Кроме того, нам позволено полностью очистить контейнер (Clear) и экспортировать его содержимое в bmp-файл. Если вас не устраивает порядок следования рисунков, то (удерживая левую кнопку мыши) перетащите пиктограммы на новые места. При необходимости в выпадающих списках Transparent Color и Fill Color задайте цвета прозрачности и заливки. Если размеры загружаемого рисунка расходятся со значениями, установленными в свойствах `Height` и `Width` компонента, то обязательно определитесь с опциями (Options). Режим «Center» выведет рисунок точно по центру, «Stretch» растянет изображение и «Crop» обрежет лишнее.

Управление списком изображений во время выполнения программы

Добавление к списку иконки осуществляет функция:

```
function AddIcon(Image: TIcon): Integer;
```

Для одновременного добавления в список очередной пиктограммы `Image` и ее маски (шаблона) `Mask` подойдет метод:

```
function Add(Image, Mask: TBitmap): Integer;
```

Как уже отмечалось ранее, маска позволяет определять, какие из пикселов основного изображения станут прозрачными. Если маска не требуется, то передайте во второй параметр неопределенный указатель `nil`. Вместо контуров маски допускается определить цвет прозрачности. Для этого понадобится другая функция:

```
function AddMasked(Image: TBitmap; MaskColor: TColor): Integer;
```

Все три функции добавляют изображение в конец списка и в качестве результата возвращают индекс пиктограммы в списке.

Если логика программы требует вставки пиктограммы в определенное место списка, то целесообразно воспользоваться одной из процедур, представленных в табл. 25.1.

Таблица 25.1. Процедуры вставки пиктограммы в список TImageList

Метод	Описание
`procedure InsertIcon(Index: Integer; Image: TIcon);`	Вставляет иконку `Image` в позицию `Index`.
`procedure Insert(Index: Integer; Image, Mask: TBitmap);`	Вставляет изображение `Image` и маску `Mask` в позицию `Index`.
`procedure InsertMasked(Index: Integer; Image: TBitmap; MaskColor: TColor);`	Заменяет изображение `Image` в позицию `Index` и генерирует для него маску на основе цвета `MaskColor`.

За включение или отключение маски отвечает свойство `Masked`*. Если это свойство установлено в* `True`*, результирующее изображение будет представлять комбинацию рисунка и маски.*

Список изображений можно забрать из другого компонента класса `TCustomImageList`. С такой задачей справится процедура:

```
procedure AddImages(Value : TCustomImageList);
```

По умолчанию компонент `TImageList` резервирует в памяти место, достаточное для размещения четырех пиктограмм. После заполнения этой области (перед добавлением пятой картинки) элемент запрашивает у менеджера памяти пространство для очередных четырех рисунков. Программист имеет возможность влиять на алгоритм выделения памяти для хранения растровых образов – изменив шаг приращения в свойстве:

```
property AllocBy: Integer; //по умолчанию 4
```

Такое вмешательство может оказаться полезным в случаях, когда логика разрабатываемого программного продукта предполагает многократные операции по загрузке/выгрузке значительных порций растровых образов. В подобной ситуации целесообразно несколько увеличить шаг `AllocBy`, подобное решение ускорит процесс записи данных в контейнер. Безусловно, размер шага должен быть меньше прогнозируемого числа хранимых пиктограмм, ведь нет смысла захватывать память под пару сотен картинок, если в среднем контейнер будет содержать не более пятидесяти.

Если позиция пиктограммы в списке вас не устраивает, то для ее перемещения на более достойное место потребуется процедура:

```
procedure Move(CurIndex, NewIndex : Integer);
```

Метод передвинет образ из позиции `CurIndex` в позицию `NewIndex`.

Для замены одного рисунка другим используются процедуры, представленные в табл. 25.2.

Таблица 25.2. Процедуры замены пиктограмм в списке TImageList

Метод	Описание
`procedure ReplaceIcon(Index: Integer; Image: TIcon);`	Заменяет иконку с индексом `Index` на новую иконку `Image`.
`procedure Replace(Index: Integer; Image, Mask: TBitmap);`	Заменяет пиктограмму с индексом `Index` изображением `Image` и маской `Mask`.
`procedure ReplaceMasked(Index: Integer; NewImage: TBitmap; MaskColor: TColor);`	Заменяет изображение с номером `Index` новым растровым образом `NewImage` и генерирует для него маску на основе цвета `MaskColor`.

Загрузка ресурса

Список изображений умеет загружать данные из ресурса. Типом данных `TResType` предусмотрено существование трех вариантов таких данных: растровый рисунок (`rtBitmap`), курсор (`rtCursor`) и иконка (`rtIcon`). К наиболее универсальным методам обслуживания ресурса относится пара перезагружаемых функций:

```
function GetInstRes(Instance: THandle; ResType: TResType;
            const Name: string; Width: Integer; LoadFlags: TLoadResources;
            MaskColor: TColor): Boolean; overload;

function GetInstRes(Instance: THandle; ResType: TResType;
            ResID: DWORD; Width: Integer; LoadFlags: TLoadResources;
            MaskColor: TColor): Boolean; overload;
```

Здесь `Instance` – дескриптор контейнера; `ResType` – тип ресурса; `MaskColor` – цвет маски. Разница между методами – в способе указания ресурса. В первом случае нам потребуется его имя (параметр `Name`), во втором – числовой идентификатор (параметр `ResID`). С помощью комбинации флагов `LoadFlags` определяются особенности загрузки ресурса. Например, флаг `lrMonoChrome` указывает на то, что нам требуется черно-белое изображение, флаг `lrFromFile` говорит, что данные загружаются из внешнего файла.

Практически все остальные методы доступа к внешнему ресурсу выступают в роли надстроек над функцией `GetInstRes()`. Если ресурс интегрирован в тот же файл, которому принадлежит компонент `ImageList`, или к нашему процессу статически подключена внешняя библиотека с интересующим нас ресурсом, то для загрузки данных следует применять функции:

```
function GetResource(ResType: TResType; const Name: string;
                     Width: Integer;
                     LoadFlags: TLoadResources;
                     MaskColor: TColor): Boolean;
function ResourceLoad(ResType: TResType; const Name: string;
                      MaskColor: TColor): Boolean;
```

Если мы планируем поместить в список изображений данные из файла с именем Name, то вместо GetInstRes() проще вызвать ее упрощенный аналог:

```
function FileLoad(ResType: TResType; const Name: string;
                  MaskColor: TColor): Boolean;
```

При наличии дескриптора внешнего контейнера, в котором хранится требуемый ресурс (например, динамической библиотеки или пакета), с ролью загрузчика превосходно справится функция:

```
function ResInstLoad(Instance: THandle; ResType: TResType;
                     const Name: string; MaskColor: TColor): Boolean;
```

В случае успешного завершения все рассмотренные функции возвратят true.

В момент любого изменения содержимого списка генерируется событие:

```
property OnChange: TNotifyEvent;
```

Об общем количестве пиктограмм в контейнере нас проинформирует свойство:

```
property Count: Integer; //только для чтения
```

Удаление пиктограмм

Полная очистка коллекции изображений осуществляется процедурой:

```
procedure Clear;
```

Для удаления одного рисунка из списка необходимо воспользоваться методом:

```
procedure Delete(Index: Integer);
```

Единственный параметр процедуры – индекс удаляемого рисунка.

Особенности отображения пиктограмм

К вопросу отображения рисунка на поверхности элементов управления компонент TImageList подошел со всей ответственностью. Вне зависимости от выбранной пользователем цветовой схемы рабочего стола компонент способен прорисовать свою пиктограмму на любом элементе управления с обеспечением прозрачности и эффекта фокуса ввода. В зависимости от сложности задачи результат достигается или за счет обычного копирования пиктограммы в нужную область, или посредством проведения тернарной растровой операции между тремя объектами: поверхностью элемента управления, пиктограммой и кистью заливки (маской)[1]. Порядок вывода пиктограммы на поверхности элемента управления находится в прямой зависимости от состояния свойства.

```
property DrawingStyle: TDrawingStyle;
```

Предусмотрено 4 режима работы компонента (табл. 25.3). По умолчанию компонент настроен на нормальный вывод пиктограмм (стиль dsNormal).

---

[1] Операция обеспечивается за счет обращения к функции Windows GDI BitBlt().

Таблица 25.3. Возможные значения TDrawingStyle

Значение	Описание	Примечание
dsFocus	Эффект фокуса ввода. К растру рисунка подмешивается 25% цвета указанного в свойстве BlendColor.	Эффект проявится только для набора изображений, содержащих маску.
dsSelected	Эффект выбранного объекта. К растру подмешивается 50% цвета, указанного в свойстве BlendColor.	
dsNormal	Обычный режим. Вывод изображения осуществляется с учетом цвета BkColor. Если значение BkColor не установлено (clNone), то для обеспечения эффекта прозрачности пиктограмма опирается на маску.	
dsTransparent	Эффект прозрачности. Достигается без учета указанного в BkColor цвета.	

Над обеспечением видимости эффекта фокуса ввода на пиктограмме трудится еще одно свойство:

```
property BlendColor: TColor;
```

Это свойство учитывается только в тех случаях, когда свойство DrawingStyle установлено в dsFocus или dsSelected. Если элемент управления выбран пользователем, то при проведении растровой операции этот цвет подмешивается в растровую операцию.

Список изображений TImageList способен не только хранить растровые образы, но и выводить их на поверхности элементов управления, обладающих своим холстом. Для этого предназначены процедуры:

```
procedure Draw(Canvas: TCanvas; X, Y, Index: Integer;
               Enabled: Boolean=True); overload;
procedure Draw(Canvas: TCanvas; X, Y, Index: Integer; ADrawingStyle: TDrawingStyle;
               AImageType: TImageType; Enabled: Boolean=True); overload;
```

Здесь Canvas – канва элемента управления; X, Y – координаты начала вывода рисунка; Index – номер рисунка; в состоянии true параметра Enabled процедура выводит цветное изображение, в состоянии false – его монохромную версию; от состояния AImageType зависит, будет использоваться маска или нет.

Еще одна пара перегружаемых процедур умеет выводить на поверхности другого элемента изображение, перекрытое оверлейным образом.

```
procedure DrawOverlay(Canvas: TCanvas; X, Y: Integer; ImageIndex: Integer;
                      Overlay: TOverlay; Enabled: Boolean = True); overload;
procedure DrawOverlay(Canvas: TCanvas; X, Y: Integer; ImageIndex: Integer;
                      Overlay: TOverlay; ADrawingStyle: TDrawingStyle;
                      AImageType: TImageType; Enabled: Boolean = True); overload;
```

Перед вызовом процедуры `DrawOverlay()` *не забывайте назначить оверлейный образ. Это осуществляется за счет обращения к методу* `Overlay()`.

По сравнению с предыдущими процедурами здесь добавился новый параметр `Overlay`. Это индекс оверлейной маски, он может принимать значение в диапазоне от 0 до 3.

Индекс пиктограммы, которая станет использоваться в роли оверлейной маски, определяется функцией:

```
function Overlay(ImageIndex: Integer; Overlay: TOverlay): Boolean;
```

Оверлейный образ может выводиться поверх основного с обеспечением прозрачности с помощью процедуры `DrawOverlay()`.

Прозрачность

Для того чтобы система определилась, какие именно пикселы пиктограммы надо сделать прозрачными, программист должен либо явным образом указать значение замещаемого цвета, либо назначить маску-шаблон, при бинарном сложении которой с основным рисунком достигается эффект прозрачности.

Если мы пойдем по первому пути, то прозрачный цвет определяется для всех находящихся в контейнере пиктограмм в свойстве:

```
property BkColor: TColor;
```

Второй подход позволит определить прозрачные пикселы для каждой пиктограммы отдельно. В этом случае нужно задействовать любой из уже известных нам методов `AddMasked()`, `InsertMasked()`, `Add()` или `Insert()`. Второе решение обладает наибольшей гибкостью, так как позволяет делать прозрачными любые пикселы изображения, а не только пикселы одного определенного цвета.

Для включения режима прозрачности установите в `true` свойство:

```
property Masked: Boolean;
```

Вывод на экран прозрачного рисунка осуществляется в два этапа. На первом этапе производится логическое умножение (операция AND) пикселов рисунка и его маски. Затем полученное изображение переносится на поверхность элемента управления, причем пикселы изображения объединяются с пикселами поверхности на основе исключения ИЛИ (операция XOR).

Экспорт пиктограмм из контейнера

Наиболее близкая по духу к функциям GDI пара методов позволяет получить дескрипторы хранящихся в контейнере изображений:

```
function GetImageBitmap: HBITMAP;
function GetMaskBitmap: HBITMAP;
```

Первый метод возвращает дескриптор блока основных изображений, второй – дескриптор маски.

Если требуется извлечь из контейнера отдельную пиктограмму в виде родных для программиста Delphi растрового образа TBitmap или иконки TIcon, то обращаемся к одной из следующих процедур:

```
procedure GetBitmap(Index: Integer; Image: TBitmap);
procedure GetIcon(Index: Integer; Image: TIcon); overload;
procedure GetIcon(Index: Integer; Image: TIcon; ADrawingStyle: TDrawingStyle,
                  AImageType: TImageType); overload;
```

В параметре Index указываем индекс пиктограммы, параметр Image возвратит изображение в формате TBitmap (процедура GetBitmap) или в формате TIcon (процедура GetIcon). Более полная версия перегружаемого метода GetIcon() потребует двух дополнительных параметров, оказывающих влияние на внешний вид возвращаемого рисунка. В первую очередь, это параметр ADrawingStyle, определяющий стиль вывода пиктограммы на экран (dsFocus, dsSelected, dsNormal или dsTransparent), и параметр AImageType, конкретизирующий, что именно нас интересует, – картинка itImage или ее маска itMask.

Если нас одновременно интересуют и образ, и его маска, то убить двух зайцев одним выстрелом способен метод:

```
procedure GetImages(Index: Integer; Image, Mask: TBitmap);
```

Взаимодействие с элементами управления

К основным потребителям услуг списка изображений относятся:

- Кнопки TToolButton (элементы управления панели TToolBar).
- Пункты меню TMenuItem (элементы главного TMainMenu и всплывающего меню TPopupMenu).
- Команды TAction (из состава менеджера TActionManager и списка команд TActionList).
- Иерархическое дерево – компонент TTreeView.
- Список с расширенными возможностями – компонент TListView.
- Закладки и блокноты TTabControl и TPageControl.

В табл. 25.4 представлено соответствие между основными элементами пользовательского интерфейса и свойствами, обеспечивающими подключение к элементу набора изображений TImageList.

Установив связь с компонентом TImageList, обращаемся к хранящейся в списке изображений пиктограмме. Для этого требуется указать ее индекс.

```
property ImageIndex: TImageIndex;
```

или

```
property StateIndex: Integer;
```

В результате этих действий на поверхности элемента управления будет нарисована соответствующая пиктограмма.

Таблица 25.4. Соответствие свойств типа TCustomImageList и компонентов

Компонент	Свойства типа TCustomImageList			
	Images	StateImages	LargeImages	SmallImages
TToolBar	+	–	–	–
TCoolBar	+	–	–	–
TMenuItem	+	–	–	–
TAction	+	–	–	–
TPageControl	+	–	–	–
TTabControl	+	–	–	–
TTreeView	+	+	–	–
TListView	–	+	+	+

РЕЗЮМЕ

Список изображений TImageList предназначен для хранения коллекции одинаковых по размерам растровых рисунков формата bmp и ico. Основное направление компонента – централизованное предоставление загруженных в него пиктограмм в распоряжение других элементов управления и меню.

26

Графические элементы управления VCL

Весь спектр визуальных элементов управления Delphi разделяется на два больших множества – оконные и графические элементы управления. Описание оконных элементов, к сожалению, не входит в рамки этой книги. Отметим лишь то, что все они построены на основе абстрактного класса `TWinControl`. Кстати, термин «оконный» появился благодаря приставке «Win» в названии класса. У всех наследников `TWinControl` выделяются три ключевых особенности:

1. Наличие дескриптора окна – связующего звена с функциями API Windows.
2. Способность получать фокус ввода.
3. Возможность выступать в качестве контейнера для других элементов управления.

На этом разговор об оконных элементах завершаем и переходим к их антиподам – графическим компонентам. Свое исследование графических элементов управления мы начнем с их базового класса – `TGraphicControl`. На основе `TGraphicControl` построен ряд компонентов, отвечающих за вывод графических изображений. У всех компонентов опубликовано свойство `Canvas`, обеспечивающее доступ к основным графическим функциям GDI.

Класс TGraphicControl

Абстрактный класс `TGraphicControl` создан на основе `TControl` (рис. 26.1). Потомки класса `TGraphicControl` не могут получить фокус ввода, не способны служить контейнерами для других элементов и совершенно не желают обрабатывать события, поступающие от клавиатуры. У них даже нет дескриптора! Одним словом, это полные антагонисты оконных элементов управления. Вместе с тем графические элементы управления в совершенстве решают задачи, связанные с выводом и обработкой графики, и при этом потребляют значительно меньше системных ресурсов, чем их соседи по палитре компонентов Delphi.

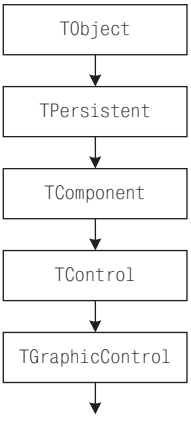


Рис. 26.1. Иерархия TGraphicControl

Свою работу мы начнем с ознакомления с ключевыми свойствами, методами и обработчиками событий, которые были унаследованы классом `TGraphicControl` от его предков.

Идентификация объекта

Во время программирования средствами Windows API мы привыкли, что в Windows идентификатором объекта обычно выступает его дескриптор – уникальное целое число. В объектно-ориентированных средах программирования такой подход зачастую бывает неудобен, тем более что язык программирования способен предложить более очеловеченный способ обращения к объекту по его имени. Для этого в классе `TComponent` был объявлено свойство:

```
property Name: TComponentName;
```

Не знаю, стоит ли напоминать, что в пределах проекта Delphi имя объекта обязательно должно быть уникальным, иначе вы будете обруганы средой программирования.

Иногда в качестве дополнительного идентификатора программисты используют свойство:

```
property Tag: Integer;
```

Это свойство имеется у всех наследников класса `TComponent` и предназначено для хранения дополнительной числовой информации.

Создание и уничтожение объекта

Жизненный цикл любого объекта в Delphi начинается с вызова его конструктора `Create()` и завершается обращением к деструктору `Destroy()`. Заголовки перечисленных методов упомянуты еще на уровне патриарха всей библиотеки VCL – класса `TObject`.

Если компонент был размещен на форме программистом еще в период визуального проектирования, то необходимости в явном обращении к конструктору и деструктору нет. Это действие выполняется автоматически в момент создания и уничтожения формы. Но действительно гибкий интерфейс приложения для Windows может быть сформирован только в «ручном» режиме, когда все этапы построения внешнего вида проекта явным образом описываются в коде программы. Пользователя больше порадует программный продукт, способный адаптироваться к текущей ситуации, чем пассивное монументальное творение.

После создания элемента управления необходимо назначить ему родительский компонент, иначе Delphi не сможет вывести его на экран. Родительский компонент, а точнее, контейнер, на поверхности которого размещается элемент управления, определяется свойством:

```
property Parent: TWinControl;
```

В качестве контейнера способен выступать практически любой потомок класса `TWinControl`, обычно это форма или панель. Родительский компонент хранит список всех принадлежащих ему компонентов и отвечает за уничтожение своих подчиненных.

Вопросу динамического создания группы элементов посвящен следующий пример. Это заготовка, которая может быть использована при создании входящей в поставку Windows игры «Сапер». В момент вывода формы на экран вся клиентская область окна заполняется кнопками (экземплярами класса `TSpeedButton`), нажимая на которые игрок осуществляет разминирование.

```
procedure TForm1.FormShow(Sender: TObject);
const BtnSize=15; //высота и ширина кнопки
var btn:TSpeedButton;
    x,y,z:Integer;
begin
x:=0;y:=0;z:=0;
while TRUE do
begin
  btn:=TSpeedButton.Create(Form1); //создание объекта
  btn.Parent:=Form1; //родительский компонент - форма Form1
  btn.Width:=BtnSize; btn.Height:=BtnSize; //назначение размеров
  btn.GroupIndex:=z; INC(z);//индекс группировки
  btn.Left:=x; btn.Top:=y; //местоположение кнопки
  X:=X+BtnSize+1; //смещение кнопки по оси X
  if Form1.ClientWidth<X+BtnSize then //
     begin
     x:=0;
     y:=y+btnSize+1; //смещение кнопки по оси Y
     if Form1.ClientHeight<y+btnSize then Break; //вся форма заполнена
     end;
  end;
end;
```

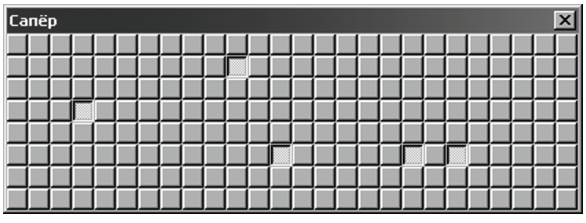


Рис. 26.2. Динамическое создание элементов управления

Выход из цикла `while..do` осуществляется после заполнения формы кнопками `TSpeedButton`. Результат представлен на рис. 26.2. Согласитесь, что применение конструктора `Create()` в коде программы значительно удобнее, чем кропотливое размещение кнопок на форме во время визуального проектирования.

Расположение и размеры элемента управления

В среде Delphi координаты элемента управления задаются относительно клиентской области окна. Направления осей абсцисс и ординат соответствуют наиболее распространенному в Windows режиму отображения – `MM_TEXT`. Другими словами, начало координат находится в левой верхней точке клиентской области окна, ось X направлена слева направо, ось Y – сверху вниз. По умолчанию в роли единицы измерения выступает пиксел.

Геометрические размеры любого визуального элемента управления определяются парой свойств:

```
property Height: Integer; //высота
property Width: Integer;  //ширина
```

Расстояние по горизонтали от левого края формы до верхнего левого угла компонента устанавливается в свойстве:

```
property Left: Integer;
```

Расстояние от верхнего среза клиентской части формы до левого верхнего угла компонента:

```
property Top: Integer;
```

Роль и место перечисленных выше свойств вы найдете на рис. 26.3. В роли демонстрационного элемента управления выбран компонент-изображение `TImage`.

Между тем число свойств, связанных с определением местоположения и размеров элемента управления, не ограничивается этой четверкой. Некоторые программисты предпочитают работать с одним-единственным свойством, хранящим границы прямоугольной области элемента управления:

```
property BoundsRect: TRect;
```

Помимо параметров внешних границ элемента управления необходимо владеть размерами его внутренней клиентской части.

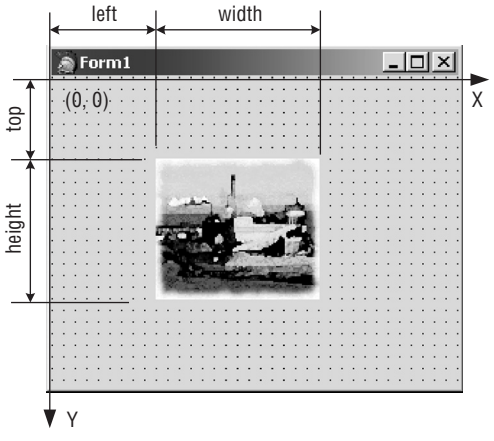


Рис. 26.3. Геометрические размеры элемента управления

```
property ClientHeight: Integer;
property ClientWidth: Integer;
property ClientRect: TRect;
```

Экранные координаты левой верхней точки клиентской области компонента хранятся в свойстве:

```
property ClientOrigin: TPoint;
```

Еще раз подчеркну, что здесь речь идет об экранных координатах, а не о координатах компонента относительно родительского элемента управления, например формы.

Программист вправе предусмотреть систему ограничений на размеры элемента управления. Для этих целей предназначено свойство:

```
property Constraints: TSizeConstraints;
```

Здесь можно установить ограничения на вертикальный и горизонтальный размеры элемента управления: `MaxHeight`, `MaxWidth`, `MinHeight` и `MinWidth`.

Выравнивание элемента управления

В классе `TControl` опубликовано свойство:

```
property Align: TAlign;
```

отвечающее за выравнивание элемента управления, располагаемого в клиентской области контейнера, например формы. По умолчанию свойство отключено (`Align=alNone`). В табл. 26.1 указаны возможные значения, которые может принимать свойство `Align`.

Достоинство свойства `Align` заключается в том, что при изменении размеров контейнера-владельца элемент управления растягивается (сжимается) вместе с ним.

Таблица 26.1. Возможные значения TAlign

Значение	Результат
alNone	Выравнивание отключено.
alTop	Объект займет всю верхнюю часть контейнера. Попытки изменить свойства Width и Top не принесут результата.
alBottom	Элемент управления переместится в нижнюю часть контейнера. Свойство Width недоступно.
alLeft	Объект займет все свободное пространство в левой части контейнера. Свойства Top, Left и Height недоступны.
alRight	Компонент займет правую часть контейнера. Свойство Height недоступно.
alClient	Компонент займет всю свободную клиентскую часть контейнера. Свойства Top, Left, Height и Width недоступны.

Предусмотрена возможность «привязки» элемента управления не только к левому верхнему углу клиентской части контейнера. Свойство Anchors определяет способ позиционирования компонента относительно четырех граней рамки его контейнера.

```
property Anchors: TAnchors;
type TAnchors = set of TAnchorKind;
type TAnchorKind = (akTop, akLeft, akRight, akBottom);
```

Например: установим Anchors = [akRight, akBottom], тогда при изменении размеров контейнера местоположение элемента управления будет неизменным относительно правой (akRight) и нижней (akBottom) рамок.

Видимость и активность элемента управления

Каждый элемент управления обладает свойством:

```
property Visible: Boolean;
```

позволяющим включить (true) или отключить (false) его отображение. Такую же операцию осуществляют две процедуры:

```
procedure Show; // аналог visible := true;
procedure Hide; // аналог visible := false;
```

Для того чтобы элемент управления был видимым, необходимым условием является видимость контейнера-владельца, на котором размещен этот элемент.

В случае, если два элемента управления расположены один над другим (перекрывают друг друга), достать элемент на поверхность или спрятать его помогают, соответственно, методы:

```
procedure BringToFront; //переместить вперед
procedure SendToBack;   //переместить назад
```

Методы BringToFront() *и* SendToBack() *доступны и во время проектирования. Для их вызова щелкните правой кнопкой мыши по компоненту и выберите соответствующий пункт всплывающего меню.*

Ответственность за активность элемента несет свойство:

```
property Enabled: Boolean;
```

При установке Enabled в false элемент управления перестает реагировать на клавиатуру, мышь и на мольбы начинающего пользователя.

Обработка событий

Потомки TGraphicControl способны реагировать на события, связанные с мышью (табл. 26.2–26.4).

Таблица 26.2. События, связанные с перемещением мыши и щелчками кнопками мыши

Событие	Описание
`property OnClick: TNotifyEvent;`	Одинарный щелчок по элементу управления.
`property OnContextPopup: TContextPopupEvent;` `type TContextPopupEvent = procedure(Sender: TObject; MousePos: TPoint; var Handled: Boolean) of object;`	Событие, связанное с контекстным меню элемента управления. Вызывается по щелчку правой кнопкой мыши над поверхностью компонента. Здесь: Sender – компонент источник события; MousePos – координаты курсора мыши; Handled – управляемый программистом признак завершения обработки события.
`property OnDblClick: TNotifyEvent;`	Двойной щелчок.
`property OnMouseActivate: TMouseActivateEvent;` `type TMouseActivateEvent = procedure(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer; HitTest: Integer; var MouseActivate: TMouseActivate) of object;`	Активизация мышью. Событие вызывается при нажатии кнопки мыши в момент нахождения курсора над компонентом, причем родительский элемент управления (владелец графического компонента) должен быть неактивен. Здесь: Sender – компонент источник события; Button – код кнопки мыши; Shift – состояние служебных клавиш клавиатуры; (X,Y) – координаты указателя мыши; HitTest – информирует, над какой областью компонента была нажата кнопка; MouseActivate – управляемый программистом результат события.
`property OnMouseEnter: TNotifyEvent;`	Вызывается в момент появления указателя мыши над поверхностью элемента управления.
`property OnMouseLeave: TNotifyEvent;`	Указатель мыши выходит за пределы поверхности элемента управления.
`property OnMouseDown: TMouseEvent;` `type TMouseEvent = procedure(Sender: TObject; Button: TMouseButton; Shift: TShiftState; X, Y: Integer) of object;`	Генерируется при нажатии кнопки мыши. Здесь: Sender – компонент источник события; Button – код кнопки мыши; Shift – состояние служебных клавиш клавиатуры.
`property OnMouseUp: TMouseEvent;`	Генерируется при отпускании кнопки мыши. Параметры аналогичны событию OnMouseDown().

С реализацией популярного интерфейса перетаскивания (*drag-and-drop*) связаны четыре события (табл. 26.3). Эти события генерируются в строгой последовательности у компонентов-участников операции.

Таблица 26.3. События перетаскивания (drag-and-drop)

Событие	Описание

`property OnStartDrag: TStartDragEvent;`
`type TStartDragEvent =`
`procedure (Sender: TObject; var DragObject: TDragObject) of object;`
	Первое событие в череде событий drag-and-drop. Свидетельствует о том, что над элементом-источником нажата кнопка мыши и начато движение с удержанием этой кнопки в нажатом состоянии.

`property OnDragOver: TDragOverEvent;`
`type TDragOverEvent =`
`procedure(Sender, Source: TObject;`
`X, Y: Integer;`
`State: TDragState;`
`var Accept: Boolean) of object;`
	Второе по очередности событие. Генерируется у элемента-приемника в момент появления указателя мыши над его поверхностью. Здесь: `Sender` – компонент-приемник; `Source` – компонент-источник; (`X`,`Y`) – координаты указателя мыши; `Accept` – управляемый программистом результат, значение `true` разрешает операцию перетаскивания, `false` – запрещает.

`property OnDragDrop: TDragDropEvent;`
`type TDragDropEvent =`
`procedure(Sender, Source: TObject; X, Y: Integer) of object;`
	Третье событие. Вызывается у элемента-приемника в момент отпускания пользователем кнопки мыши. Назначение параметров аналогично аргументам функции `OnDragOver()`.

`property OnEndDrag: TEndDragEvent;`
`type TEndDragEvent =`
`procedure(Sender, Target: TObject;`
`X, Y: Integer) of object;`
	Четвертое и заключительное событие в операции. Возникает у элемента-источника. Обычно используется для деинициализации ранее задействованных ресурсов. Здесь:
	`Sender` – компонент-источник; `Target` – компонент-получатель; (`X`,`Y`) – координаты указателя мыши.

Реализация интерфейса drag-and-drop на практике достаточно проста – построим пример, демонстрирующий технологию перетаскивания растрового изображения из компонента в компонент. Разместите на форме произвольное количество компонентов `TImage` (это прямой наследник `TGraphicControl`, специализирующийся на хранении и отображении рисунков). Минимальный код должен содержать описание пары событий: `OnDragOver()` и `OnDragDrop()`. В первом событии мы удостоверяемся в том, что компонент-источник является экземпляром класса `TImage` и содержит графический ресурс.

```
procedure TForm1.Image1DragOver(Sender, Source: TObject; X, Y: Integer;
                                State: TDragState; var Accept: Boolean);
begin
Accept:=(Sender Is TImage) and
        ((Source As TImage).Picture.Graphic.Empty=false);
end;
```

Второе событие нацелено на передачу изображения из компонента-источника Source в компонент-получатель Sender. После передачи ресурса мы освобождаем ресурс у источника данных.

```
procedure TForm1.Image1DragDrop(Sender, Source: TObject; X, Y: Integer);
begin
{Передаем образ получателю}
(Sender As TImage).Picture:=(Source As TImage).Picture;
{Освобождаем ресурс у источника}
(Source As TImage).Picture.Bitmap.Dormant;
(Source As TImage).Picture.Bitmap.FreeImage;
(Source As TImage).Picture.Bitmap.ReleaseHandle;
{обновляем форму}
Form1.Repaint;
end;
```

Заметьте, что оба события принадлежат одному-единственному компоненту Image1. Чтобы интерфейс drag-and-drop стал доступен всем остальным элементам управления, нам необходимо разделить эти события между всеми имеющимися компонентами класса TImage. Для этого нам потребуется написать несколько вспомогательных строк в событии, возникающем при выводе формы на экран.

```
procedure TForm1.FormShow(Sender: TObject);
var i:integer;
begin
Image1.Picture.LoadFromFile('..\graphics\comp.bmp');//загрузка файла с рисунком
for i:=0 to form1.ControlCount-1 do {перебор всех элементов формы}
  if Form1.Controls[i] is TImage then {если это экземпляр TImage}
      With Form1.Controls[i] as TImage do
      begin
       DragKind:=dkDrag; //интерфейс работы drag-and-drop
       DragMode:=dmAutomatic; //режим автоматический
       if Name<>'Image1' then {разделяем обработчики событий}
          begin
            OnDragDrop:=Image1.OnDragDrop;
            OnDragOver:=Image1.OnDragOver;
          end;
      end;
end;
```

Еще один поддерживаемый потомками класса TGraphicControl интерфейс называется буксировкой (*drag-and-dock*). Обозначенный механизм предназначен для переноса с места на место элемента управления. Впрочем, в случае графических элементов управления корректнее говорить не о полной, а о частичной поддержке буксировки. Дело в том, что буксировка должна производиться между компонентами-доками. В роли дока способны выступать только оконные элементы управления (потомки класса TWinControl) – они способны размещать на своей поверхности другие элементы управления. Графические элементы не могут выступать в роли контейнера для других компонентов, они лишь могут быть взяты на буксир. Из-за ограниченных функциональных

возможностей графические компоненты при буксировке реагируют лишь на два из четырех событий интерфейса drag-and-drop (табл. 26.4).

Таблица 26.4. События буксировки (drag-and-dock)

Событие	Описание
`property OnStartDock: TStartDockEvent;`	Начало буксировки.
`TStartDockEvent = procedure(Sender: TObject; var DragObject: TDragDockObject) of object;`	Здесь: `Sender` – буксируемый элемент; `DragObject` – необязательный объект, создаваемый программистом, содержащий дополнительные данные о буксируемом элементе управления.
`property OnEndDock: TEndDragEvent;`	Завершение буксировки.
`type TEndDragEvent = procedure(Sender, Target: TObject; X, Y: Integer) of object;`	Здесь: `Sender` – буксируемый элемент; `Target` – объект-док; `(X,Y)` – координаты указателя мыши.

Свойства и методы TGraphicControl

Основой графического класса служит единственное свойство:

```
property Canvas: TCanvas;
```

Это наш старый знакомый холст, он обеспечивает упрощенный доступ к функциям графического интерфейса устройства – GDI Windows. Класс `TCanvas` был подробно рассмотрен в предыдущих главах.

Единственный новый метод, объявленный в `TGraphicControl`, отвечает за прорисовку графического элемента управления.

```
procedure Paint; virtual;
```

Прорисовку инициирует родительский контейнер, на котором располагается потомок класса `TGraphicControl`. *Для этого дочернему элементу посылается сообщение операционной системы* `WM_PAINT`.

На рис. 26.4. представлена плеяда наследников класса `TGraphicControl` – графические элементы управления VCL. Практически все компоненты являются прямыми потомками базового класса, единственное исключение представляет метка (компонент `TLabel`), она нуждается в промежуточном классе `TCustomLabel`.

Изображение, компонент TImage

Компонент предназначен для хранения и отображения растрового или векторного изображения. В нем инкапсулирован объект `TPicture`, способный содержать иконку, битовую карту, метафайл или другой графический объект, заранее определенный программистом. Доступ к свойствам и методам `TPicture` осуществляется при помощи свойства:

```
property Picture: TPicture;
```

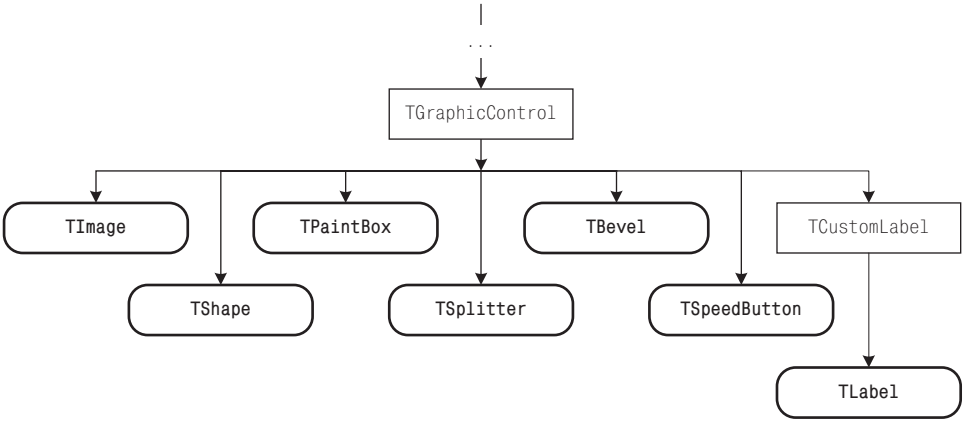


Рис. 26.4. Графические элементы управления

Например, загрузка файла производится следующим образом:

```
Image1.Picture.LoadFromFile('c:\bitmap.bmp');
```

Ряд свойств `TImage` нацелен на настройку способа вывода изображения на поверхности компонента. Центровку изображения обеспечивает свойство:

```
property Center: Boolean;
```

Установив в `true` свойство:

```
property Stretch: Boolean;
```

вы растянете изображение на всю рабочую поверхность компонента. Исключение составляют иконки (`TIcon`) – они на такой подвиг неспособны.

Свойство, отвечающее за прозрачность, определяет, скрывает ли компонент находящиеся под ним объекты:

```
property Transparent: Boolean;
```

При работе с большими по размеру и сжатыми изображениями (например, рисунками в формате JPEG) целесообразно устанавливать в `true` свойство:

```
property IncrementalDisplay: Boolean;
```

При `IncrementalDisplay = true` вывод изображения осуществляется прямо в процессе загрузки файла, в противном случае – сразу по окончанию загрузки.

Фигура, компонент TShape

Компонент `TShape` предназначен для построения простейших геометрических фигур. Тип фигуры (прямоугольник, квадрат, прямоугольник или квадрат с закругленными углами, эллипс, круг) задается свойством:

```
property Shape: TShapeType;
```

```
type TShapeType = (stRectangle, stSquare, stRoundRect,
                   stRoundSquare, stEllipse, stCircle);
```

Тип пера и заливка фигуры определяется соответственно свойствами:

```
property Pen: TPen;
property Brush: TBrush;
```

Область для рисования, компонент TPaintBox

Компонент TPaintBox представляет собой прямоугольную область, ограничивающую пространство, доступное для рисования. Компонент обладает своим собственным холстом (свойством Canvas).

Ключевым методом элемента управления является процедура:

```
procedure Paint;
```

Обработчик события OnPaint() реагирует на сообщение WM_PAINT и инициализирует процесс перерисовки холста компонента.

```
procedure TForm1.PaintBox1Paint(Sender: TObject);
begin
PaintBox1.Canvas.Brush.Color:=clBlack;
PaintBox1.Canvas.Rectangle(10,10,50,50);
end;
```

Все вызовы графических функций обычно сосредотачиваются внутри обработчика события OnPaint(). Такой подход гарантирует полную сохранность изображения при перекрытии рабочей поверхности PaintBox другими формами, при сворачивании формы и в других случаях, когда не обойтись без автоматической перерисовки.

Разделитель, компонент TSplitter

К решению графических задач разделитель имеет весьма отдаленное отношение. Вместе с тем это законный потомок TGraphicControl (см. рис. 26.3), посему на равных с другими компонентами вынесен на наше рассмотрение.

Экземпляр класса TSplitter специализируется на разделении двух объектов (например, панелей TPanel) с одновременным предоставлением услуги по изменению их размеров. Для того чтобы компонент TSplitter смог проявить свои нетривиальные способности, разделяемые компоненты должны быть выровнены с помощью свойства Align. После этого разделитель располагается между обслуживаемыми объектами и по большому счету на этом программирование заканчивается, пользователь перемещает разделительную линию с помощью мышки, а все остальное TSplitter сделает сам...

Впрочем, небольшая инспекция возможностей компонента не повредит. По сравнению со своим предком разделитель богаче на четыре свойства. В первую очередь, это свойство, определяющее минимальный размер разделяемого объекта, по достижению которого TSplitter перестанет уменьшать своего визави:

```
property MinSize: NaturalNumber; //по умолчанию 30 пикселов
```

Достигнув минимального размера, уменьшаемый объект может полностью исчезнуть с экрана. Это произойдет, если свойство:

```
property AutoSnap: Boolean;      //по умолчанию true
```

установлено в true.

Эффект от перемещения разделителя тоже может быть изменен. Для этого предназначено свойство:

```
property ResizeStyle: TResizeStyle;//по умолчанию rsPattern
type TResizeStyle = (rsNone, rsLine, rsUpdate, rsPattern);
```

По умолчанию компонент предлагает нам режим rsPattern. Особенность этого режима в том, что во время перемещения разделителя указателем мыши система реально не изменяет размеры обслуживаемых объектов до тех пор, пока пользователь не отпустит кнопку мыши в новом месте. Точно такой же результат достигается в состоянии rsLine, небольшое различие между двумя режимами заключается в особенности визуализации процесса. В первом случае разделительная полоса рисуется полупрозрачной, а во втором – обычной линией. Режим rsUpdate более нетерпелив, он заставит разделяемые компоненты моментально реагировать на перемещение мыши и изменять свои размеры в динамике (не дожидаясь, пока пользователь соизволит отпустить левую кнопку). Режим rsNone в комментариях не нуждается, в этом состоянии TSplitter теряет работоспособность.

Процесс перемещения разделителя сопровождается парой событий. Первое из них просто информирует программиста, что пользователь передвигает разделительную полосу.

```
property OnMoved: TNotifyEvent;
```

Второе событие более полезно.

```
property OnCanResize: TSplitterCanResizeEvent;
TSplitterCanResizeEvent = procedure(Sender: TObject; var NewSize: Integer;
                                    var Accept: Boolean) of object;
```

Здесь, контролируя предлагаемый системой новый размер NewSize, мы можем запретить (Accept:=false) или разрешить (Accept:=true) изменение габаритов объекта.

Рельефная панель, компонент TBevel

Компонент в большей степени полезен дизайнеру, чем программисту. Благодаря TBevel формы проекта приобретают более эстетичный вид и, в общем, на этом его возможности заканчиваются.

Внешний вид компонента задается свойством:

```
property Shape: TBevelShape;
type TBevelShape = (bsBox, bsFrame, bsTopLine, bsBottomLine,
                    bsLeftLine, bsRightLine, bsSpacer);
```

Бордюр компонента может быть утопленным или приподнятым. Это достигается за счет свойства:

```
property Style: TBevelStyle;
type TBevelStyle = (bsLowered, bsRaised);
```

Быстрая кнопка, компонент TSpeedButton

Из всего имеющегося у Delphi богатого списка кнопок элемент TSpeedButton (если следовать дословному переводу – быстрая кнопка) – единственный не оконный элемент управления. Отсутствие в иерархическом списке наследования класса TWinControl лишило кнопку возможности реагировать на события клавиатуры и получать фокус ввода, но вместе с тем значительно снизило потребляемые компонентом системные ресурсы.

Как правило, наша знакомая применяется в составе панелей инструментов. Но бывают и исключения, помните заготовку для игры «Сапер»? Кнопка TSpeedButton умеет отображать на своей поверхности пиктограмму. Визуальная привлекательность кнопки обеспечивается свойством:

```
property Glyph: TBitmap;
```

Здесь хранится отображаемая компонентом картинка. В дополнение к этому быстрая кнопка способна отказаться от объемного внешнего вида и стать абсолютно плоской, но в момент появления над ней курсора «вынырнуть» на поверхность. Если вам по душе такое поведение, то установите в true свойство:

```
property Flat: Boolean;
```

Воспользовавшись свойством:

```
property Transparent: Boolean;
```

кнопку можно сделать прозрачной (Transparent = true).

Несколько быстрых кнопок обычно объединяются в группу. Для этого свойству:

```
property GroupIndex: Integer;
```

всех кнопок присваивается одинаковое, отличное от нуля значение. Объединение двух и более элементов управления SpeedButton приводит к весьма полезному результату: все кнопки группы приобретают зависимую фиксацию, очень похожую на поведение группы переключателей (компонента TRadioGroup). Теперь щелчок по любой из быстрых кнопок переводит последнюю в утопленное состояние и одновременно вернет в исходное состояние предыдущую нажатую кнопку.

О состоянии кнопки в группе (нажата или нет) можно судить по содержимому свойства:

```
property Down: Boolean;
```

В утопленном состоянии Down = true. Если кнопка отличается особой индивидуальностью и не входит ни в какую из групп (GroupIndex = 0), то свойство Down

теряет работоспособность. Для того чтобы хотя бы одна из кнопок группы всегда находилась в утопленном состоянии, установите в false свойство:

```
property AllowAllUp: Boolean;
```

Метка, компонент TLabel

Класс TLabel описан в модуле стандартных элементов управления StdCtrls. Метка – один из наиболее часто используемых графических компонентов, этот элемент управления предназначен для вывода на экран поясняющего текста. Ключевое свойство компонента, построенного на основе класса TLabel, – его заголовок:

```
property Caption : string;
```

Это отображаемая меткой текстовая строка. Она недоступна для редактирования пользователем и может изменяться только из кода программы.

По своей сути компонент TLabel *является объектно-ориентированным воплощением одной из самых совершенных функций GDI по выводу текста –* DrawText *(см. главу 20 «Вывод текста»).*

За автоподстройку геометрических размеров компонента при отображении текста отвечает пара свойств:

```
property AutoSize: Boolean; //по умолчанию true
property WordWrap: Boolean; //по умолчанию false
```

По умолчанию (AutoSize=true и WordWrap=false) текст в метке располагается в одну строку и метка растягивается по горизонтали так, чтобы вместить весь текст из свойства Caption. Размер компонента по вертикали зависит только от высоты установленного шрифта. Если оба свойства переведены в true, то компонент зафиксирует свой горизонтальный размер, но позволит изменять размер по вертикали для вывода текста в многострочном режиме.

Если режим автоматической подстройки размеров метки отключен (AutoSize=false), программисту будет позволено самостоятельно управлять выравниванием надписи в границах компонента. За выравнивание по горизонтали несет ответственность свойство:

```
property Alignment: TAlignment;
type TAlignment = (taLeftJustify, //по левому краю
                   taRightJustify, //по центру
                   taCenter); //по правому краю
```

Вертикальное выравнивание определяется состоянием свойства:

```
property Layout: TTextLayout;
type TTextLayout = (tlTop,//по верхнему краю
                    tlCenter, //по центру
                    tlBottom);//по нижнему краю
```

Для включения прозрачности установите в true свойство:

```
property Transparent: Boolean; //по умолчанию true
```

Несмотря на то, что метка ни в коем случае не способна получать фокус ввода, она умеет проявить сообразительность и присоединиться к любому оконному элементу управления (потомку класса `TWindowControl`) с помощью свойства:

```
property FocusControl: TWinControl;
```

С момента присоединения к метке оконный элемент управления сможет получать фокус ввода по нажатии пользователем связанной с меткой клавиши-акселератора. Еще одним условием передачи фокуса ввода должно быть включение в состояние `true` свойства:

```
property ShowAccelChar: Boolean; //по умолчанию true
```

и включение в состав отображаемой меткой надписи (свойства `Caption`) амперсанда «&». С этого момента метка начинает подчеркивать следующий за амперсандом символ.

РЕЗЮМЕ

В этой главе мы рассмотрели имеющиеся в распоряжении Delphi графические элементы управления. Их объединительная черта – общий предок `TGraphicControl`. Подобное родство определяет основные семейные черты изученных компонентов. Они не имеют оконного дескриптора, не способны получать фокус ввода и не могут служить в качестве площадки для размещения других компонентов. Перечисленные недостатки компенсируются малой потребностью в системных ресурсах и привлекательным внешним видом.

27

Организация работы с принтером

Может, это кому-то и покажется странным, но с изобретением электронных способов обмена данными оборот бумажных документов не только не сократился, но, напротив, вышел на абсолютно другой качественный уровень. Ее Величество Бюрократия и в XXI веке по-прежнему предпочитает лицезреть бумагу, скрепленную парой-другой печатей, а не какие-то невесомые компьютерные документы, пусть даже подтверждаемые десятком электронных подписей. Поэтому сегодня немыслимо существование ни одного офиса без десятка матричных, струйных и лазерных принтеров, активно размазывающих тонны чернил на листах белоснежной бумаги.

Для обычного пользователя внешняя сторона печати документов выглядит весьма прозаично – романтики здесь не больше, чем в ежедневной поездке на метро. Щелкнув по кнопке печати документа, можно расслабиться за чашечкой кофе, изредка подкидывая в принтер очередную порцию бумаги. А в это время за кулисами Windows кипят нешуточные страсти. Процессом печати заправляет *спулер печати* (*print spooler*) и ряд его агентов:

1. Провайдер печати.
2. Процессор печати (*print processor*).
3. Графическое ядро печати (*graphics engine*).
4. Мониторы портов (*port monitors*).

Спулер – это системный процесс (исполняемый файл и некоторый набор динамически загружаемых библиотек), управляющий приемом, обработкой, диспетчеризацией и *распределением заданий* (*print jobs*) на печать. Спулер включает в задание не только распечатываемые данные, но и некоторый исходный код, определяющий порядок обработки выводимой на бумагу информации. Как только спулер сформирует задание, ему на помощь приходит провайдер печати. Провайдер ответственен за отправку задания на принтер (точнее говоря, на компьютер с физически подключенным к нему принтером). Для этих целей в операционной системе имеется целых четыре провайдера:

локальный, сетевой, провайдер Novell и провайдер HTTP. Ко всему прочему провайдер контролирует очередь печати и умеет запускать и останавливать задания в этой очереди. Выполнив свой функционал, провайдер передает бразды правления процессору печати. Процессор изучает поступающие в задании данные и, пользуясь услугами графического ядра печати, преобразует задание в вызовы функций *DDI* (*device driver interface*). Таким образом, процессор и графическое ядро печати конвертируют высокоуровневые данные в команды, понятные драйверу печатающего устройства. Затем низкоуровневые данные направляются на принтер. Это уже зона ответственности мониторов портов. Монитор определяет, каким образом принтер подключен к компьютеру и управляет процессом ввода/вывода.

Обычно задание на печать формируется в формате расширенного метафайла EMF. Такие данные могут быть распечатаны практически на любом печатающем устройстве. Но кроме формата EMF операционная система Windows умеет работать с низкоуровневыми данными (RAW), текстом (TEXT) и данными PostScript (PSCRIPT1).

Работа с принтером средствами Windows

Работа с принтером средствами API не вызовет особой трудности у программиста, обладающего знаниями GDI Windows. Ведь принтер – это не что иное, как устройство вывода, и на нем можно рисовать почти точно так же, как на экране монитора, с той лишь разницей, что принтер не умеет стирать с листа ранее отпечатанные данные и накладывает ряд существенных ограничений на операции с ROP-кодами.

Любой процесс графического вывода в Windows начинается с получения контекста устройства. Эта операция осуществляется посредством обращения к уже знакомой нам функции `CreateDC()`. При работе с принтером нам понадобится только ее третий параметр, через который требуется передать имя установленного в системе устройства.

```
DC:= CreateDC(Nil, PChar('Canon CP660PS'),Nil, Nil);
```

Предусмотрено несколько способов получения сведений о том, какие принтеры установлены в системе. Если при разработке своей программы вы планируете ограничиться только услугами Windows API, то рекомендую обратиться к функции `EnumPrinters()`. *Этот метод заполняет массив специализированных записей информацией о доступных принтерах, их драйверах и портах. Мы не станем останавливаться на обсуждении* `EnumPrinters()`, *так как Delphi предлагает более простое решение этой задачи, оно реализовано средствами класса* `TPrinter` *и будет рассмотрено несколько позже.*

Став счастливым обладателем контекста принтера, программист отдает указание спулеру печати на формирование нового задания. Для этого вызывается функция:

Win GDI

```
function StartDoc(DC : HDC; const DocInfo : TDocInfo) : Integer;
```

Здесь DC – контекст устройства, DocInfo – запись, содержащая характеристики задания.

```
Type TDocInfo = packed record
    cbSize: Integer;         //размер структуры
    lpszDocName: PWideChar;  //название задания
    lpszOutput: PWideChar;   //имя выходного файла
    lpszDatatype: PWideChar; //поле для хранения типа данных («emf», «raw»),
                             //обычно передается nil
    fwType: DWORD;           //дополнительная информация
  end;
```

При работе со структурой TDocInfo программисту достаточно определить размеры структуры и название задания.

Команду на завершение задания печати отдаст функция:

Win GDI

```
function EndDoc(DC : HDC) : Integer; //конец задания
```

Единственный параметр метода – контекст печатающего устройства.

Задание печати как минимум состоит из одной страницы. Программисту надо информировать драйвер принтера о начале и окончании данных страницы. Для этого предназначена пара функций GDI:

Win GDI

```
function StartPage(DC : HDC) : Integer;//старт новой страницы
function EndPage (DC : HDC) : Integer; //окончание страницы
```

Пример использования всех рассмотренных выше методов представлен в следующем листинге:

```
procedure PrintText(DeviceName, Text: string);
var   DC:HDC;
      DocInfo: TDocInfo;
begin
DC:=CreateDC(Nil, PChar(DeviceName),Nil, Nil); //получение контекста принтера
{инициализация структуры DocInfo, содержащей информацию о задании}
  FillChar(DocInfo, SizeOf(DocInfo), 0);
  with DocInfo do
  begin
    cbSize := SizeOf(DocInfo);
    lpszDocName := PChar('Printer example');
  end;
StartDoc(DC, DocInfo);//старт задания
StartPage(DC); //начало страницы
TextOut(DC,100,100,PChar(Text),Length(Text)); //печать текстовой строки
EndPage(DC); //конец страницы
EndDoc(DC);  //завершаем задание
DeleteDC(DC); //по окончании работы не забываем освободить контекст устройства
end;
```

Придуманной нами процедуре PrintText() для нормального самочувствия требуется два параметра: DeviceName – название принтера (на котором планируется печатать данные) и Text – текстовая строка (которую мы намерены увидеть на бумаге).

Для получения дескриптора контекста принтера вместо вызова функции `CreateDC()` *допустимо воспользоваться свойством* `Handle` *класса* `TPrinter`.

Технические характеристики принтера

В составе Windows API имеется универсальная функция, способная проинформировать нас о технических характеристиках устройств вывода информации. С ней мы уже знакомы из материала второй главы, ее имя `GetDeviceCaps()`. Функция окажет неоценимую помощь при исследовании возможностей установленного в системе принтера.

В набор функций Windows API входит информационная функция `GetPrinter()`. *Получив дескриптор принтера, она в состоянии вернуть различные структуры (*`PRINTER_INFO_1` ... `PRINTER_INFO_9`*), доверху заполненные информацией о печатающем устройстве.*

Описание принтера в Delphi, класс TPrinter

Среда Delphi предлагает один из наиболее простых и одновременно удобных способов работы с печатающим устройством. Для этого к нашему проекту необходимо подключить модуль `Printers`. В результате столь несложной операции программист получает право обращаться к принтеру. Для этого предназначена функция:

```
function Printer: TPrinter;
```

Метод возвратит объект класса `TPrinter`, инкапсулирующий в своем составе практически весь интерфейс печати из Windows GDI.

Если вы создадите свою версию потомка класса `TPrinter`, то Delphi позволит заменить глобальный экземпляр принтера. Для этого надо обратиться к входящей в состав модуля `Printers` функции:

```
function SetPrinter(NewPrinter: TPrinter): TPrinter;
```

Выбор принтера

В рамках операционной системы может быть установлено неограниченное число принтеров. Это локальный принтер, сетевые печатающие устройства и различного рода эмуляторы принтеров, например весьма популярный Adobe Acrobat Distiller, «перегоняющий» любые документы в формат документа Adobe Acrobat, или Microsoft Office Document Image Writer Driver, позволяющий формировать снимки документов в формате растрового изображения TIFF либо MDI[1] (Microsoft Document Imaging). Весь перечень доступных устройств собирается в списке:

```
property Printers: TStrings;
```

[1] Не следует путать аббревиатуру MDI формата файла *.mdi (*Microsoft Document Imaging*) с интерфейсом приложения MDI (*Multi Document Interface*).

Индекс текущего принтера окажется в свойстве:

```
property PrinterIndex: Integer;
```

Меняя порядковый номер, мы назначаем принтер, на который отправится задание. Для обращения к используемому по умолчанию принтеру присвойте свойству PrinterIndex значение –1. Ниже представлен небольшой пример инициализации перечня доступных принтеров во время создания главной формы приложения.

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
ComboBox1.Items.Assign(Printer.Printers);
ComboBox1.ItemIndex:=Printer.PrinterIndex;
end;
```

Взаимодействие с драйвером принтера

Для получения справочной информации о настройках драйвера принтера предназначена процедура:

```
procedure GetPrinter(ADevice, ADriver, APort: PChar; var ADeviceMode: THandle);
```

В первый параметр метода возвращается название принтера, например: «EPSON Stylus 1000 ESC/P 2». В параметр ADriver передается имя драйвера, обслуживающего данный принтер. Третий параметр остался лишь для поддержки обратной совместимости с 16-разрядной версией Windows (в ту пору программистов очень интересовал номер порта принтера) и в настоящее время не используется. В последнюю переменную ADeviceMode помещается указатель на область памяти, в которую загружается структура DEVMODE. Она состоит из двух частей: общей (одинаковой для всех принтеров) и частной (уникальной для разных типов печатающих устройств). Для взаимодействия с DEVMODE в Delphi объявлена запись TDeviceModeA (ANSI версия).

```
type TDeviceModeA = packed record
    dmDeviceName: array[0..CCHDEVICENAME - 1] of AnsiChar; //имя устройства
    dmSpecVersion  :  Word;//минимальная версия Windows,
                           //в которой работает TDeviceModeA
    dmDriverVersion: Word; //версия драйвера
    dmSize         : Word; //размер общей части структуры
    dmDriverExtra  : Word; //размер уникальной части структуры
    dmFields       : DWORD;//набор флагов, инициализирующих поля записи
    dmOrientation  : SHORT;//ориентация страницы DMORIENT_PORTRAIT
                           //или DMORIENT_LANDSCAPE
    dmPaperSize    : SHORT;//размер страницы (например, DMPAPER_A4)
                           //или 0 (пользовательский)
    dmPaperLength  : SHORT;//пользовательская высота страницы
                           //когда dmPaperSize=0
    dmPaperWidth   : SHORT;//пользовательская ширина страницы (dmPaperSize=0)
    dmScale        : SHORT;//масштаб печати
    dmCopies       : SHORT;//количество копий
    dmDefaultSource: SHORT;//источник бумаги
    dmPrintQuality : SHORT;//качество печати (DMRES_HIGH,DMRES_MEDIUM,
```

```
                          //DMRES_LOW, DMRES_DRAFT)
    dmColor        : SHORT;//цветная DMCOLOR_COLOR или черно-белая
                          //DMCOLOR_MONOCHROME печать
    dmDuplex       : SHORT;//для двухсторонней печати
    dmYResolution  : SHORT;//разрешение (точек в дюйме) по вертикали
    dmTTOption     : SHORT;//способ печати TrueType шрифтов
    dmCollate      : SHORT;//режим «разобрать по копиям» при печати
                          //нескольких экземпляров документа
    dmFormName: array[0..CCHFORMNAME - 1] of AnsiChar; //имя формы
    dmLogPixels    : Word; //пикселов в одном логическом дюйме
    dmBitsPerPel   : DWORD;//указывает количество битов цвета,
                          //например, 16 бит - 65536 цветов
    …
  end;
```

В настоящем материале предложена сокращенная редакция структуры DEVMODE. *Ее подробное описание можно получить из SDK.*

```
var ADevice : Array[0..CCHDEVICENAME-1] of Char;
    ADriver : Array[0.. MAX_PATH -1] of Char;
    APort : Array[0..31] of Char;
    ADMode: THandle;
    pDMode: PDeviceMode;
begin
Printer.GetPrinter(ADevice, ADriver, APort, ADMode);
if ADMode>0 then
   begin
     pDMode:=GlobalLock(ADMode);
     pDMode^.dmPaperSize:=DMPAPER_A4;    //формат страницы A4
     pDMode^.dmCopies:=2;                //вывести 2 копии
     pDMode^.dmPrintQuality:=DMRES_HIGH; //наилучшее качество печати
      //...
     GlobalUnLock(ADMode);
   end;
end;
```

Обратите внимание, что перед началом работы с объектом (структурой TDeviceModeA**) мы блокируем эту область в памяти с помощью** GlobalLock()**. Названная функция Windows API забирает объект в единоличное пользование и возвращает указатель на первый байт структуры. Завершив работу с объектом, разблокируем его. Для этого потребуется вызов функции** GlobalUnlock()**.**

При разработке программного обеспечения не забывайте о существовании специальных элементов управления, предоставляющих удобный пользовательский интерфейс настройки драйвера принтера. Это компоненты TPrinterSetupDialog *и* TPageSetupDialog *со страницы палитры стандартных диалогов Dialogs.*

Для назначения текущего принтера с одновременной перенастройкой его драйвера рекомендуется использовать функцию:

Win GDI

```
procedure SetPrinter(ADevice, ADriver, APort: PChar; ADeviceMode: THandle);
```

Аргументы метода аналогичны описанным выше параметрам `GetPrinter()`.

Не путайте принадлежащий классу `TPrinter` *метод* `SetPrinter()` *с одноименной процедурой, описанной в модуле* `Printers`. *Они решают разные задачи: метод изменяет настройки драйвера печатающего устройства, а процедура позволяет заменить экземпляр объекта* `TPrinter` *на другой экземпляр* `TPrinter`.

Управление страницей документа

Вспомним про еще один ингредиент, без которого печать невозможна, – лист бумаги. Средствами визуальной библиотеки компонентов можно легко получить данные о ключевых характеристиках бумаги. В первую очередь, это высота и ширина листа:

```
property PageHeight: Integer;
property PageWidth: Integer;
```

Размерности передаются в пикселах.

Ориентация листа бумаги при печати определяется в свойстве:

```
property Orientation: TPrinterOrientation;
TPrinterOrientation = (poPortrait, poLandscape);
```

Предполагается всего два варианта ориентации – портретная или альбомная.

Информация о некоторых возможностях принтера по управлению заданием содержатся в свойстве:

```
property Capabilities: TPrinterCapabilities;
```

Описание `TPrinterCapabilities` предложено в табл. 27.1.

Таблица 27.1. Возможные значения TPrinterCapability

Значение	Описание
`pcCopies`	Пользователю разрешено определять число распечатываемых копий документа.
`pcOrientation`	Принтер позволяет изменять ориентацию бумаги.
`pcCollation`	При печати нескольких экземпляров одного документа указывает, что экземпляры разбираются по копиям.

Формирование и отправка задания на печать

Вся идея работы с объектом принтера заключается в обращении к поверхности бумаги как к холсту, на котором можно рисовать. Именно для этой цели глобальная переменная `Printer` инкапсулирует хорошо знакомый (и горячо любимый нами) фирменный объект VCL – холст:

```
property Canvas: TCanvas;
```

Далее остается проявить умение и сноровку и отправить на печать данные. В простейшем случае будет достаточно написать менее 10 строк исходного кода.

```
implementation
 uses Printers;

procedure TForm1.Button1Click(Sender: TObject);
begin
 with Printer do
 begin
    BeginDoc;      //старт документа
    Canvas.TextOut(10,10,'Тест печати');
    EndDoc;        //завершение документа
 end;
end;
```

Печать инициируется посредством обращения к процедуре:

```
procedure BeginDoc;
```

Это команда спулеру о начале нового задания. Если все прошло успешно, то не забудьте попрощаться, известив принтер о завершении задания:

```
procedure EndDoc;
```

Индикатором того, что принтер приступил к выполнению задания, станет переход в `true` свойства:

```
property Printing: Boolean; //только для чтения
```

Если существует необходимость разбиения документа на несколько страниц, вспоминаем о существовании процедуры:

```
procedure NewPage;
```

Обращение к этому методу вынуждает принтер завершить работу с текущим листом бумаги и перейти к следующему. Номер распечатываемой в данный момент страницы хранится в свойстве:

```
property PageNumber: Integer;
```

Для того чтобы сформированное нами задание на печать более профессионально смотрелось в окне менеджера печати, перед обращением к процедуре `BeginDoc()` задайте заголовок заданию. Имя задания передается в свойство:

```
property Title: string;
```

Отмена задания

В процессе печати документа спулер периодически проверяет состояние свойства:

```
property Aborted: Boolean; //только для чтения
```

И если названное свойство возвратит значение `true`, то это станет командой на отмену текущего задания. Свойство доступно только для чтения, поэтому при необходимости отменить уже отправленное на принтер задание обратитесь к методу:

```
procedure Abort;
```

Процедура самостоятельно позаботится о переводе Aborted в состояние true. В результате нераспечатанное задание сбрасывается, и принтер останавливается либо переходит к обслуживанию следующей задачи.

Печать многострочного текста

В девяти случаях из десяти принтер применяется для распечатки текста, в простейшем случае это может быть многострочный неформатированный текст, например хранящийся в компоненте TMemo. В подобной ситуации для вывода строк на бумагу стоит воспользоваться услугами холста, а точнее, его метода TextOut(). Напомню, что первые два параметра метода определяют место вывода строки, это, соответственно, горизонтальный и вертикальный отступы от верхнего левого угла страницы. Если с горизонтальным отступом все понятно, при печати неформатированного, выровненного по левому краю текста он каждый раз принимает одно и то же значение, то с вертикальным отступом – немного сложнее. Программист обязан предусмотреть порядок назначения вертикального отступа печати для каждой очередной строки. Поэтому в нашем примере перед отправкой задания мы с помощью метода TextHeight() выясняем высоту строки и сохраняем это значение в локальной переменной LineHeight. Позднее это значение будет использоваться как множитель при пересчете места печати очередной строки из свойства Lines компонента Memo1.

```
Y:=Y+LineHeight*2;
```

Выяснив величину отступа, в качестве второго параметра передаем ее в функцию TextOut(), а затем все как в следующем листинге:

```
var LineNum :Word;
    LineHeight,Y,X : integer;
begin
Printer.Canvas.Font:=Memo1.Font;             //синхронизация шрифтов
LineHeight:=Printer.Canvas.TextHeight('A'); //выясняем высоту строки
Printer.Title:='Название задания';           //назначаем имя задания

X:=GetDeviceCaps(Printer.Handle,PHYSICALOFFSETX); //отступ от левого края листа
Y:=GetDeviceCaps(Printer.Handle,PHYSICALOFFSETY); //отступ от верхнего края листа

Printer.BeginDoc; //старт задания

for LineNum:=0 To Memo1.Lines.Count-1 do {перебор всех строк Memo-поля}
begin
    If Y>Printer.PageHeight-20 then
    begin
    {если страница закончилась, то начинаем очередную}
    Printer.NewPage;
    Y:=GetDeviceCaps(Printer.Handle,PHYSICALOFFSETY);
    end else Y:=Y+LineHeight*2; //вертикальный отступ для очередной строки

    Printer.Canvas.TextOut(X,Y,Memo1.Lines.Strings[LineNum]); //печать строки
end;
```

```
Printer.EndDoc; //завершаем задание
end;
```

Обратите внимание на то, каким образом в программе устанавливаются отступы от верхнего и левого краев листа бумаги. Для этого мы обратились к функции GetDeviceCaps().

При печати текстовых данных, использующих шрифты разной гарнитуры, следует проверять, имеются ли в распоряжении печатающего устройства требуемые шрифты; их перечень содержится в свойстве:

```
property Fonts: TStrings;
```

Свойство возвратит список строк с именами семейств шрифтов.

Печать изображений

Наилучшим помощником при выводе на лист бумаги изображения считается метод класса TCanvas – StretchDraw(). Эта процедура подробно рассмотрена в главе, посвященной классу TCanvas, поэтому сейчас просто напомню о том, что StretchDraw() позволяет не только рисовать на холсте, но и масштабировать рисунок. Все остальное – дело техники. Уделим пару минут примеру. Перед нами процедура, позволяющая распечатать рисунок, загруженный из файла FileName. Процедура PrintPicture() загрузит данные в объект BMP, запомнит соотношение сторон картинки (для этого предназначена переменная koff) и назначит координаты прямоугольной области Rct, в пределах которой распечатается изображение.

```
procedure PrintPicture(FileName: string);
var BMP:TBitmap; koff:double; Rct:TRect;
begin
 BMP:=TBitmap.Create;
 BMP.LoadFromFile(FileName);
 koff:=BMP.Height/BMP.Width; //коэффициент для сохранения пропорций сторон

 Printer.Title:=FileName;
 Printer.Orientation:=poPortrait;
 Printer.BeginDoc; //старт задания

{определяем прямоугольную область печати}
 Rct.TopLeft:=Point(GetDeviceCaps(Printer.Handle, PHYSICALOFFSETX),
                    GetDeviceCaps(Printer.Handle, PHYSICALOFFSETY));
 Rct.BottomRight:=Point(BMP.Width, BMP.Height);

 Printer.Canvas.StretchDraw(Rct ,BMP); {вывод рисунка}
Printer.EndDoc;
BMP.free;
end;
```

Наш пример обладает одним существенным недостатком – при выводе изображения на печать не учитывается разрешающая способность принтера. Из-за этого на бумаге картинка распечатается в существенно уменьшенном виде по сравнению с ее представлением на экране компьютера.

Необходимость учета разрешения принтера при отправке задания на печать объясняется двумя серьезными причинами:

1. *Разрешающая способность самого захудалого принтера значительно превосходит возможности самого лучшего монитора. Поэтому занимающая все пространство экрана картинка размерами 800×600 пикселов при выводе на лист бумаги оставит много свободного места.*
2. *Еще одним поводом для коррекции размеров отправляемого на печать задания может стать поддержка рядом принтеров «нестандартного» разрешения. Довольно часто встречаются модели с неидентичными разрешениями по осям X и Y (360×180, 600×300, 1200×600 и т. п. точек на дюйм). При выводе рисунка на такой принтер без масштабирования мы (вполне закономерно) получим искаженное (вытянутое) изображение.*

Окно предварительного просмотра

Возможность просмотра потенциальных результатов печати на экране компьютера – бесспорный плюс любого программного продукта. Предоставив подобную сервисную возможность, программист избавит пользователя от необходимости перевода дорогостоящей бумаги и картриджа принтера. Да что там бумага и чернила... Они ничто в сравнении с сохраненными нервами и хорошим настроением пользователя.

Создание своего собственного окна предварительного просмотра если и не высший пилотаж, то, по крайней мере, бесспорное свидетельство способности программиста самостоятельно отрываться от земли. Ведь надо умудриться обмануть компьютер, убедив его, что до смешного малая поверхность экранной формы на самом деле – целый лист бумаги.

Практически ни один принтер не способен печатать на всей поверхности листа. Наличие «мертвых» зон по краям страницы объясняется особенностями конструкции печатающего устройства и в первую очередь зависит от его механической части. В технические подробности кинематики принтеров мы вдаваться не станем, но сделаем вывод, что перед формированием задания на печать целесообразно выяснить геометрические размеры страницы и размеры недоступных для печати областей листа бумаги, дабы часть распечатываемой информации не угодила в запретные зоны. Для решения этой задачи в качестве помощника привлечем нашу старую знакомую – функцию `GetDeviceCaps()`. Метод не только укажет ширину (`PHISICALWIDTH`) и высоту (`PHISICALHEIGHT`) страницы, но и уточнит физические возможности печатающего устройства по выводу данных. В первую очередь, это отступы от левого (`PHISICALOFFSETX`) и верхнего (`PHISICALOFFSETY`) краев листа. Размеры правой и нижней «мертвых» зон легко вычисляются, если мы знаем физическую ширину (`HORZRES`) и высоту (`VERTRES`) доступной для печати области (рис. 27.1).

Рассмотрим процедуру, имитирующую на экране компьютера лист бумаги:

```
{!!! Размеры: PaintBox1.ClientWidth= PaintBox1.ClientHeight !!!}
procedure TForm1.PaintBox1Paint(Sender: TObject);
var pWidth,pHeight,pLeftOffset,pTopOffset,pHorz,pVert,LogicSize:integer;
```

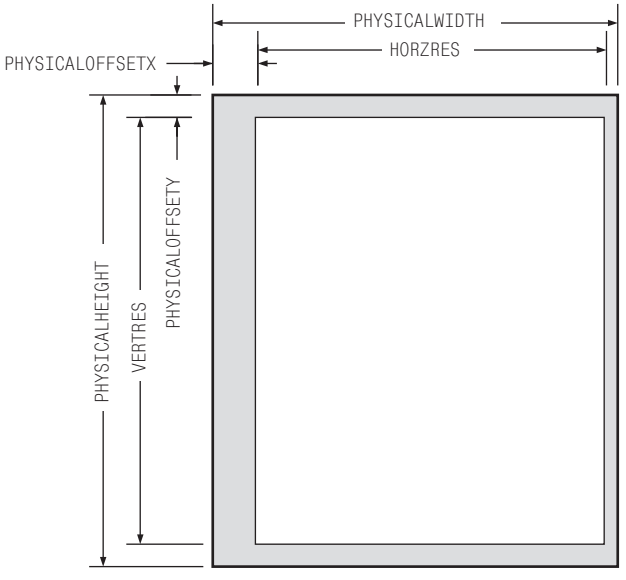


Рис. 27.1. Выяснение размеров бумажного листа с помощью функции GetDeviceCaps( )

```
    LOGPIXELSX, LOGPIXELSY : Integer;
    ShadowRect,PaperRect,WorkRect:TRect;
begin
pWidth:=GetDeviceCaps(Printer.Handle, PHYSICALWIDTH);   //ширина листа бумаги
pHeight:=GetDeviceCaps(Printer.Handle, PHYSICALHEIGHT); //высота листа бумаги
pLeftOffset:=GetDeviceCaps(Printer.Handle, PHYSICALOFFSETX); //отступ слева
pTopOffset:=GetDeviceCaps(Printer.Handle, PHYSICALOFFSETY);  //отступ сверху
pHorz:=GetDeviceCaps(Printer.Handle, HORZRES); //ширина рабочей области
pVert:=GetDeviceCaps(Printer.Handle, VERTRES); //высота рабочей области
with PaintBox1.Canvas do
begin
  Brush.Color:=clAqua;
  FillRect(PaintBox1.ClientRect);

  SetMapMode(Handle, MM_ISOTROPIC);{назначаем изотропный режим отображения}
  {назначим протяженность логического окна на 100 единиц
   больше, чем наибольшая сторона листа бумаги}
    if pWidth>pHeight then LogicSize:=pWidth+100 else LogicSize:=pHeight+100;
    SetWindowExtEx(Handle, LogicSize, LogicSize, nil);
    SetViewportExtEx(Handle, PaintBox1.ClientWidth, PaintBox1.ClientHeight, nil);
    {точка начала отсчета - левый верхний угол изображения листа бумаги}
    SetWindowOrgEx(Handle, -(LogicSize-pWidth) div 2,
    -(LogicSize-pHeight) div 2, nil);

    PaperRect:=RECT(0,0,pWidth,pHeight);{прямоугольная область страницы}
    ShadowRect:=RECT(40,40,40+pWidth,40+pHeight);{прямоугольная область тени}

    WorkRect.Top:=PaperRect.Top+pTopOffset;{границы рабочей области принтера}
    WorkRect.Left:=PaperRect.Left+pLeftOffset;
```

```
    WorkRect.Bottom:=PaperRect.Top+pTopOffset+pVert;
    WorkRect.Right:=PaperRect.Left+pLeftOffset+pHorz;

    Brush.Color:=clGray;  //рисуем тень
    FillRect(ShadowRect);

    Brush.Color:=clWhite; //рисуем лист
    FillRect(PaperRect);

    Pen.Color:=clRed;   //рисуем границы рабочей области
    Pen.Style:=psDash;
    Rectangle(WorkRect);

//код печати


…
end;
end;
```

В момент возникновения события `OnPaint()` с помощью `GetDeviceCaps()` мы узнаем геометрические размеры листа. В роли холста в состоянии выступать поверхность любого компонента, обладающего свойством `Canvas`. В нашем случае это область для рисования `TPaintBox`. Особо отмечу, что для нормальной работы представленного выше кода необходимо соблюдение равенства сторон холста (`PaintBox1.Width=PaintBox1.Height`) в противном случае мы получим искаженную картинку. Поэтому в коде программы стоит предусмотреть место для строк, выравнивающих ширину и высоту компонента. Результаты нашей работы представлены на рис. 27.2.

При решении подобных задач мы не сможем обойтись без поддержки функций Windows API, отвечающих за настройку режима отображения. Сердце

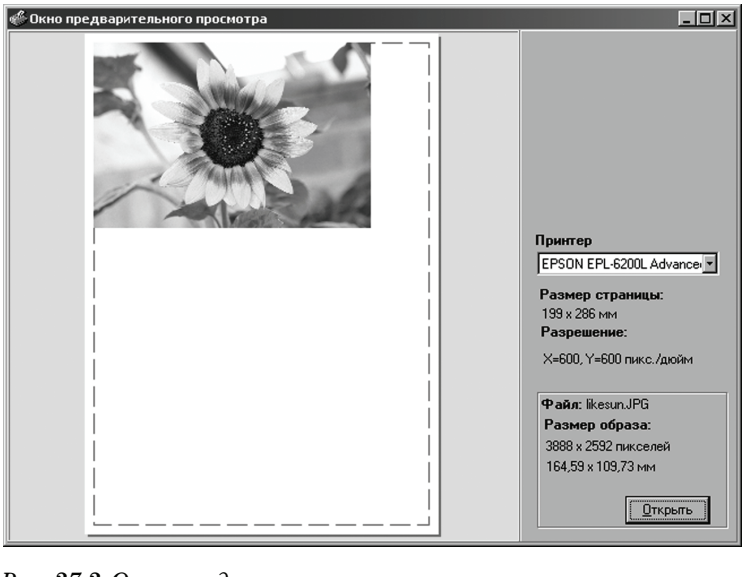


Рис. 27.2. Окно предварительного просмотра

процедуры – перевод холста в пользовательский (изотропный) режим отображения. Для этого вызывается функция `SetMapMode()` с ключом `MM_ISOTROPIC`. С этого момента мы приобретаем эксклюзивное право назначать свою собственную единицу измерения. Несколько позднее, благодаря изотропному режиму отображения и функциям GDI `SetWindowExtEx()` и `SetViewportExtEx()` мы программно изменяем значения протяженности холста. С этого момента программа начинает полагать, что небольшая поверхность компонента `PaintBox` вмещает в себя целый лист бумаги, пусть даже огромного – A0 – размера. Для еще большего удобства с помощью `SetWindowOrgEx()` мы переносим точку отсчета координат в левый верхний угол листа бумаги. Все остальное – дело техники: определяем координаты прямоугольной области, имитирующей лист бумаги (структура `PaperRect`), области тени (`ShadowRect`) и рабочей области (`WorkRect`).

В этой главе уже упоминалось, что печатающее устройство обладает значительно большей разрешающей способностью, чем экран монитора. Допустим, что монитор способен выводить 96 точек в одном дюйме по вертикали. Это разрешение вполне подходит для визуального восприятия информации с экрана. Но при попытке вывода на принтер с разрешением 600 точек на дюйм картинки (занимающей на экране моего монитора площадь 2×2 дюйма) мы получим результат, который едва ли разглядишь под увеличительным стеклом. Почему? Ответ на этот вопрос дадут элементарные расчеты.

Высота изображения = 2 дюйма×96 точек/дюйм = 192 пиксела

192 пиксела/600 точек/дюйм = 0,32 дюйма, что примерно равно 8 мм

Таким образом, 2 «огромных» экранных дюйма на принтере с разрешением 600 dpi превратились в скромные 8 миллиметров. На досуге пересчитайте эту же задачку, но для перспективного принтера с разрешающей способностью 2400 dpi. После этого вы окончательно убедитесь в том, что в программном коде необходимо предусмотреть возможность масштабирования.

Преобразование цветного изображения для печати на монохромном принтере

В теории ответственность за преобразование цветного изображения в черно-белое лежит на драйвере принтера и, вроде бы, нашему брату не стоит вмешиваться в эту кухню. Но на практике случаются и исключения. Первое из них – ошибки в работе драйвера, которые приводят к некорректному выводу цветной графики в черно-белом виде. Второе – реализация «вручную» окна предварительного просмотра результатов печати. Поэтому стоит изучить один весьма полезный прием оказания шефской помощи монохромному принтеру.

Перевод принятой из эфира цветной картинки к черно-белому виду производится путем несложной арифметической операции:

$$Gray = R \times 0{,}3 + G \times 0{,}59 + B \times 0{,}11$$

Здесь R, G и B – соответствующие значения красной, зеленой и синей составляющих цвета. Они умножаются на весовые коэффициенты, причем наи-

меньший вес окажется у синего цвета, а наибольший – у зеленого. Перенос этого решения на язык Delphi элементарен, и его суть отражена в следующих строках листинга:

```
Function RGB_TO_BW(RGBColor:TColor):TColor;
var R,G,B : byte;
Begin
R:= GetRValue(RGBColor); G:= GetGValue(RGBColor); B:= GetBValue(RGBColor);
R:=ROUND(R*0.3+G*0.59 +B*0.11);
Result:=RGB(R,R,R);
End;
```

Привычная для программиста модель сложения цветов RGB не пользуется успехом у принтеров. Печатающие устройства предпочитают модели, исповедующие принцип поглощения (вычитания) цветов. Чаще всего цветные принтеры опираются на модель CMY. Это сочетание ярко-голубого (cyan), пурпурного (magenta) и желтого (yellow) цветов. Перечисленные цвета – инверсные по отношению к красному, зеленому и синему. Нанесение желтой краски на бумагу означает, что поглощается синий цвет. Пурпурная краска поглощает зеленый цвет, голубая – красный.

Принтеры высококачественной цветной печати работают с моделью CMYK – в картридж такого принтера добавлен резервуар с черной краской, что позволяет получать полноценный черный цвет и, что немаловажно, экономить краску в других резервуарах картриджа.

Диалог с принтером

Из тринадцати имеющихся в распоряжении программиста Delphi 2007 компонентов-диалогов три (`TPrintDialog`, `TPrintSetupDialog` и `TPageSetupDialog`) отвечают за организацию взаимодействия с печатающим устройством. Все компоненты сформированы на основе класса `TCommonDialog`.

Настройка печати, диалог TPrinterSetupDialog

Класс `TPrinterSetupDialog` вызывает стандартное окно настройки параметров принтера. Из всех существующих в Delphi компонентов диалог настройки принтера самый неприхотливый в программировании. После выбора компонента в инспекторе объектов мы обнаружим 4 опубликованных свойства (`Name`, `Tag`, `HelpContext` и `Ctl3d`) и 2 события (`OnClose` и `OnShow`). Для активизации диалогового окна предназначена функция:

```
function Execute: Boolean;
```

В результате обращения к указанному методу будет вызвано диалоговое окно, представленное на рис. 27.3.

Такой куцый перечень свойств и методов компонентов объясняется тем, что все установки принтера не возвращаются в наше приложение, а передаются в системный реестр Windows. На этом разговор об одном из самых аскетичных компонентов можно считать законченным.

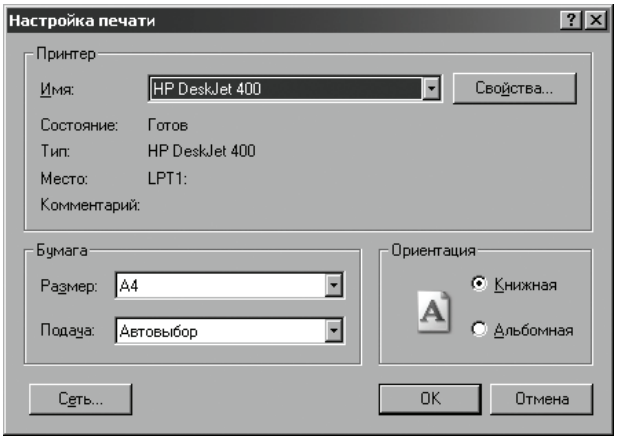


Рис. 27.3. Внешний вид диалогового окна настройки печати

Отправка задания на печать, диалог TPrintDialog

В сравнении с диалогом настройки принтера диалог печати задания отличается существенным спектром параметров. Как и у всех диалогов, любимым методом диалога печати является функция `Execute()`. В результате на экране компьютера возникнет стандартный диалог, позволяющий выбрать принтер, установить диапазон распечатываемых листов и число копий (рис. 27.4).

Особенности поведения диалогового окна определяются опциями, устанавливаемыми в свойстве:

```
property Options: TPrintDialogOptions;
```

Значения флагов свойства приведены в табл. 27.2.

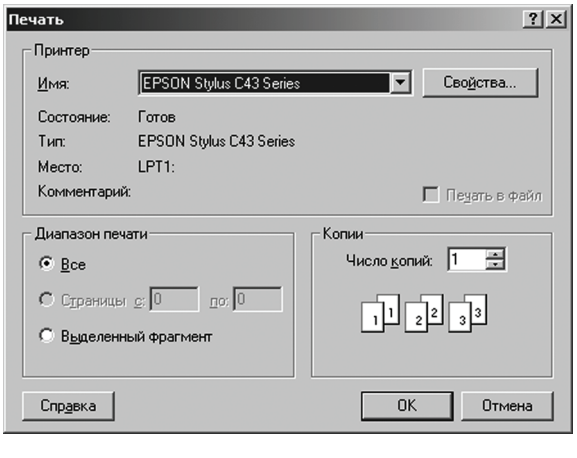


Рис. 27.4. Внешний вид диалогового окна по отправке задания на печать

Таблица 27.2. Флаги TPrintDialogOptions

Значение	Описание
poPrintToFile	Выводит флажок перенаправления задания печати в файл.
poDisablePrintToFile	Запрещает печать в файл. Блокируется флажок «Print To File».
poHelp	Отображает кнопку помощи в окне диалога. Работает не во всех версиях Windows.
poPageNums	Разрешает пользователю выбирать диапазон страниц, отправляемых на печать; в противном случае будет отправлен весь перечень страниц.
poSelection	Разрешает пользователю печатать только выделенный фрагмент текста.
PoWarning	Генерирует сообщение об ошибке при попытке отправить задание на неустановленный принтер.

Большинство свойств диалога тем или иным образом взаимодействуют с опциями окна печати. При использовании флагов poPageNums и poSelection пользователь получает право отправлять на печать несколько страниц или фрагмент задания. Результат выбора пользователя отразится в свойстве:

```
type TPrintRange = (prAllPages, prSelection, prPageNums);
property PrintRange: TPrintRange;
```

При выборе для печати только фрагмента текста свойство PrintRange примет значение prSelection, при выборе нескольких страниц – prPageNums. Если на печать отправляется весь документ, то результат будет соответствовать prAllPages.

Совместно с опцией диалога печати poPageNums трудятся свойства:

```
property FromPage: Integer;
property ToPage: Integer;
```

Значения свойств определяют начальную и конечную страницы диапазона печати. В случае, если оба свойства установлены в 0, на печать направляется весь документ.

При особом желании допустимо ограничить возможности пользователя по определению диапазона направляемых на печать страниц.

```
property MinPage: Integer;
property MaxPage: Integer;
```

По умолчанию значения свойств равны 0, т. е. ограничения отсутствуют. При выходе за границы генерируется сообщение об ошибке.

Если допустимо перенаправление задания печати в файл и пользователь воспользовался такой возможностью, то свойство:

```
property PrintToFile: Boolean;
```

примет значение true.

Количество копий предназначенного для печати документа вы обнаружите в свойстве:

```
property Copies: Integer;
```

Если число соответствует 0 или 1, то будет отпечатан только один экземпляр.

Подавляющее большинство выставляемых в диалоговом окне параметров принтера передается в специальную структуру настройки драйвера принтера – `DEVMODE`. *Аналогами этой структуры в Delphi выступают записи* `TDeviceModeA` *(ANSI версия) и* `TDeviceModeW` *(WideChar версия).*

Параметры страницы, диалог TPageSetupDialog

Диалог настройки параметров страницы – сравнительно недавний гость палитры компонентов Delphi, он появился только в 7-й версии среды программирования. Обращение к стандартному для всех диалогов методу `Execute()` вызывает окно настройки страницы перед печатью (рис. 27.5).

В первую очередь диалог настройки параметров страницы позволяет пользователю определять отступы страницы. Для этого предназначена четверка свойств:

```
property MarginLeft: Integer;{левый отступ}
property MarginRight: Integer;{правый отступ}
property MarginTop: Integer;{верхний отступ}
property MarginBottom: Integer;{нижний отступ}
```

Минимально допустимые значения отступов ограничиваются соответствующими свойствами:

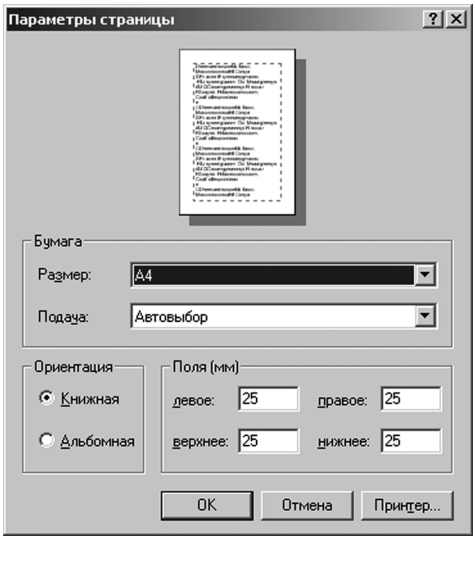


Рис. 27.5. Диалоговое окно настройки параметров страницы

```
property MinMarginLeft: Integer;
property MinMarginRight: Integer;
property MinMarginTop: Integer;
property MinMarginBottom: Integer;
```

Высота и ширина страницы зависят от пары свойств:

```
property PageHeight: Integer;
property PageWidth: Integer;
```

Используемые в диалоге единицы измерения определяются свойством:

```
property Units: TPageMeasureUnits;
type TPageMeasureUnits = (pmDefault, pmMillimeters, pmInches);
```

Здесь `pmMillimeters` — миллиметры, `pmInches` — дюймы и `pmDefault` — единицы, назначенные в локальных установках системы.

Особенности диалогового окна определяет классическое свойство:

```
property Options: TPageSetupDialogOptions;
TPageSetupDialogOptions = set of TPageSetupDialogOption;
```

Опции представлены в табл. 27.3.

Таблица 27.3. Опции диалога TPageSetupDialogOption

Опция	Описание
psoDefaultMinMargins	Устанавливает минимальное значение отступов страницы, которые могут быть назначены пользователем. Значения импортируются, исходя из установок текущего принтера.
psoDisableMargins	Запрещает пользователю настраивать отступы страницы.
psoDisableOrientation	Запрещает изменять ориентацию страницы.
psoDisablePagePainting	Отменяет режим прорисовки диалога по умолчанию.
psoDisablePaper	Запрещает изменять размер бумаги и настраивать особенности ее подачи.
psoDisablePrinter	Отключает кнопку вызова дополнительного диалога настройки принтера.
psoMargins	Устанавливает отступы страницы в 1 дюйм. Передает это значение в свойства MarginLeft, MarginRight, MarginBottom и MarginTop.
psoMinMargins	Устанавливает минимальные отступы из свойств MinMarginLeft, MinMarginRight, MinMarginBottom и MinMarginTop. В противном случае эти отступы определяются параметрами принтера.
psoShowHelp	Показывает кнопку помощи.
psoWarning	Отключает сообщение об ошибке при отсутствии установленного принтера.
psoNoNetworkButton	Скрывает и отключает кнопку сетевых устройств.

Все семь обработчиков события диалога применяются для изменения способа прорисовки окна. Они используются при включенном флаге `psoDisablePagePainting`.

РЕЗЮМЕ

Между нами говоря, огромное количество людей полагает, что компьютер – это просто дорогая печатающая машинка. Поэтому тема использования принтера в проектах Delphi весьма злободневна, а организация печати – очень непростая задача. Основная сложность в том, что это как раз тот случай, когда Delphi не сможет обойтись без помощи процедур и функций Windows API. Впрочем, в простейших случаях попросту неоценимый вклад в проект может внести инкапсулирующий возможности принтера класс `TPrinter`.

III

GDI+

28

Введение в GDI+

На протяжении всех глав, посвященных Windows GDI, неоднократно с сожалением упоминалось, что указанный прикладной интерфейс программирования не вполне соответствует требованиям, предъявляемым к современным объектно-ориентированным средам разработки программного обеспечения. Программисты Microsoft глубоко спрятали в системе все графические объекты, оставив разработчику единственную ниточку – их дескриптор. Что поделаешь! До выхода в свет Windows XP нам приходилось довольствоваться чем есть, мы писали программы, держа в одной руке уникальный описатель «замурованного» объекта, а в другой – россыпь функций Windows API. В GDI никаких других способов доступа к объекту просто не существует. Это весьма неудобно. Ко всему прочему для программирования на «голом» API кроме глубоких знаний операционной системы программисту еще и требуются терпение, внимательность и усидчивость – то, что в коде VCL занимает одну строку, в коде Windows API превращается в полстраницы листинга. Поэтому вместо построения эффективных программ, в которых графическая компонента реализуется непосредственно на базе GDI, начинающие программисты и программисты среднего уровня пользуются услугами доброжелательных графических библиотек VCL. В угоду сохранения преимущества быстрой и комфортной разработки программного продукта они зачастую жертвуют такими значимыми показателями качества приложения, как быстродействие, производительность и объем занимаемой памяти.

Вняв мольбам (по крайней мере, так хочется надеяться) разбалованных объектно-ориентированными средами программистов, к выходу Windows XP корпорация Microsoft несколько пересмотрела свои представления о графической составляющей новых операционных систем. В них на помощь классическому GDI призвана обновленная технология GDI+.

При желании GDI+ можно развернуть и на базе более ранних версий Windows (Windows 98/NT 4.0 SP6/Me/2000), установив библиотеку GdiPlus.dll в эти системы.

С точки зрения разработчика программного обеспечения самое главное достижение GDI+ в том, что он основан на объектно-ориентированной технологии. Отправным классом выступает `GdiplusBase`, на его базисе зиждется все остальное обширное семейство, включающее 4 десятка классов и «классиков». Соответствие новой графической технологии основным требованиям ООП, безусловно, значительно упрощает разработку программного обеспечения, здесь и повышение наглядности, и улучшение процесса отладки, и многое другое. В этом вы убедитесь сами, прочитав оставшиеся главы книги.

Мифы и реальность

Любая сравнительно новая технология на первых порах обрастает комом слухов и домыслов, поэтому прежде чем мы приступим к изучению GDI+, стоит расставить все точки над «i».

Разговор начнем с развенчивания активно культивируемого мифа о том, что GDI+ призван заменить «устаревший» GDI. Хорошо это или плохо, но это не вполне так. Судите сами, GDI+ – это всего-навсего объективно-ориентированное воплощение старого доброго GDI или, говоря простым языком, обычная оболочка. Для подтверждения моих слов достаточно воспользоваться утилитой tdump.exe из каталога BIN среды Delphi. Это небольшая программа позволяет осуществить анализ любой библиотеки или исполняемого файла на предмет экспортируемых и импортируемых им функций. Простейший вариант применения утилиты заключается в сбросе полученных данных в текстовый файл. Для этого скопируйте библиотеку gdiplus.dll и файл tdump.exe в отдельный каталог, с помощью команды Windows «cd» (например: cd c:\) назначьте его текущим и наберите следующую команду в строке:

```
tdump gdiplus.dll gdiplus.txt
```

В результате в текстовом файле gdiplus.txt окажется интересующая нас информация. Откройте сгенерированный файл в любом текстовом редакторе и найдите секцию «Import». Подробно просмотрев ее, вы увидите, что библиотека gdiplus.dll импортирует функции из библиотек kernel32.dll, user32.dll, ole32.dll, advapi32.dll и, конечно, gdi32.dll. Это и есть подтверждение моих слов о том, что GDI+ никак не сможет заменить GDI, ведь он построен на его основе (рис. 28.1)!

Второе распространенное заблуждение относительно GDI+ заключается в ошибочном утверждении, что GDI+ быстрее обычного GDI. К глубокому сожалению, и это не совсем так. Выступая в роли оболочки (пусть даже интеллектуальной) для графических объектов и функций Windows API, объекты GDI+ не в состоянии работать проворнее GDI. Согласитесь, разве можно прибыть на место быстрее автомобиля, в котором едешь? Разве что через лобовое стекло, но это уже клинический случай...

Так какой же смысл в применении в наших проектах механизма GDI+, если он, по сути, не является принципиально новой технологией и не способен обогнать «древний» GDI? Причин включения в свои проекты графики GDI+ предостаточно:

Во-первых. С точки зрения разработчика работа в GDI+ настолько же удобнее GDI, насколько программирование на Delphi удобнее работы на первых версиях Pascal. Значительно сокращается время создания проекта и читабельность кода. Это особенно заметно, когда предполагается создание респектабельного графического интерфейса.

Во-вторых. Опора на GDI+ позволит новичкам и программистам среднего уровня избежать многих ошибок, связанных с работой на «голом» API. Ведь теперь обращения к строптивым функциям инкапсулированы в классы, которые приложат все усилия для предотвращения их некорректного использования. Как следствие, сокращается время отладки проекта.

В-третьих. Обновленная библиотека предоставляет в наше распоряжение уже готовые решения типовых задач по обработке графики. Например, классы по работе с наиболее распространенными форматами изображений. Ко всему прочему, решения подготовлены профессиональными программистами, что всегда лучше, чем старания любителя.

«Классовый» подход к графическому ядру новых операционных систем – далеко не единственное достоинство GDI+. В сравнении с исходным GDI наш новый знакомый приобрел весьма полезные новые способности. Для систематизации разделим их на четыре ветви:

1. Современная двухмерная векторная графика, включающая:
   - Новый вид сплайнов, позволяющих превращать обычную ломаную линию в кривую со сглаженными углами.
   - Усовершенствованные координатные преобразования.
   - Полная и безусловная поддержка мировых преобразований всеми объектами GDI+.

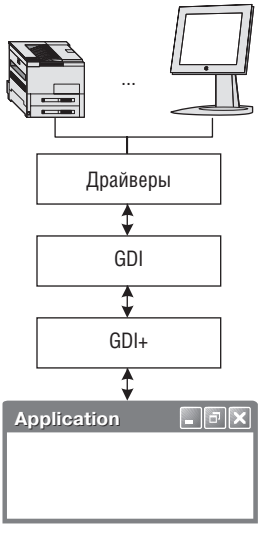


Рис. 28.1. Место GDI+ в механизме графического вывода Windows

- Кисти с поддержкой градиентной заливки.

2. Расширенные возможности по работе с изображениями:
   - Библиотека поддерживает ключевые форматы растровой (BMP, GIF, JPEG, PNG, TIFF, ICON) и векторной графики (WMF, EMF и их логическое продолжение – EMF+).
   - Обработка метаданных цифровой фотографии EXIF[1].
3. Усовершенствованный вывод текста.
4. Встроенная система повышения качества графики, в частности: устранение ступенчатого эффекта на изображениях (*antialiasing*), сглаживание текстовых строк, улучшенные алгоритмы масштабирования растровых образов, широкий спектр возможностей по цветовой коррекции рисунков.

Подготовка к работе

Разобравшись с мифом, перейдем к практической стороне работы в GDI+. У меня для вас две новости. Одна хорошая, а другая не очень. Начну с плохой. К сожалению, приходится констатировать один малоприятный факт – разработчики Delphi проигнорировали появление на свет GDI+. О причинах остается только догадываться... Можно предположить, что в стенах Borland полагают, что возможностей графических классов из состава VCL вполне достаточно для построения приложений малой степени сложности, а при усложнении задачи профессиональный программист сможет напрямую обратиться к Windows API.

Ни одна из вышедших в свет на момент написания этих строк сред программирования Borland Delphi (включая CodeGear RAD Studio с Delphi 2007) не имеет библиотек поддержки GDI+. Поэтому за необходимыми заголовочными файлами надо обратиться по адресу http://www.progdigy.com. Этот сайт содержит бесплатные модули GDI+ для Delphi.

Теперь хорошая новость. Нет необходимости самостоятельно изобретать заголовочные файлы к GDI+. Эта работа уже проведена сторонними разработчиками. Рекомендую воспользоваться сайтом *http://www.progdigy.com*, с которого можно скачать небольшой архив с уже готовыми к использованию (и, что немаловажно, бесплатными) модулями. Архив содержит демонстрационные файлы, пакеты и собственно модули на языке Pascal (каталог pas). Здесь вы обнаружите четыре модуля: GDIPAPI, GDIPOBJ, GDIPUTIL и DirectDraw.

В библиотеке к книге в каталоге SOURCE вы также найдете все необходимые файлы GDI+.

[1] Подключаемая к файлу в формате JPEG информация, содержащая дополнительные данные о фотографии.

Соглашение об именовании классов GDI+ в проектах Delphi

При работе с GDI+ в среде Delphi мы опираемся на программные модули независимого разработчика, не имеющего прямого отношения ни к корпорации Microsoft, ни к корпорации Borland. Но так как целью разработчика было обеспечение возможности работы с расширенной графикой Windows из среды программирования Delphi, то при именовании классов GDI+ создатель библиотеки следовал соглашениям об именах, принятых и в Borland, и в Microsoft. В итоге получился такой симбиоз: название класса складывается из трех составляющих:

1. Символ «T», представляющий классическое начало для названия класса в Delphi.
2. Аббревиатура «GP» (надо полагать, сокращение от слов GDI Plus).
3. Имя класса, принятое в заголовочных файлах GDI+ Microsoft.

Таким образом, если в официальной документации Microsoft SDK класс GDI+, отвечающий за создание сплошной кисти, носит название `SolidBrush`, то в наших проектах он станет именоваться `TGPSolidBrush`, перо `Pen` — `TGPPen`, регион `Region` — `TGPRegion` и т. п.

Основной объект – холст GDI+

Рассуждая об особенностях построения классов GDI+, так или иначе приходится проводить параллели с визуальной библиотекой компонентов, реализованной в средах Delphi и C Builder. В VCL основу графического интерфейса составляет класс `TCanvas`, который на страницах этой книги мы называли холстом. Основная заслуга `TCanvas` перед программистами в том, что благодаря нему разработчик программного обеспечения абстрагируется от неудобных дескрипторов и атрибутов контекста графического устройства и начинает работать с контекстом, как с классическим объектом, обладающим свойствами и методами.

При построении GDI+ программисты Microsoft также спроектировали класс, решающий сходные с `TCanvas` задачи, в SDK он именуется `Graphics`, мы же (следуя ранее оговоренному соглашению об именах классов в Delphi) станем именовать его `TGPGraphics`. Экземпляр класса представляет собой центр мироздания новейшей версии графической системы Windows. Он объединяет в единое целое всю сложную систему объектов GDI+ и управляет процессом взаимодействия между этими объектами и контекстом графического устройства.

К сожалению, привычных нам по компонентам Delphi свойств холст GDI+ не имеет, поэтому вся жизнедеятельность объекта управляется посредством его методов. Причем последних более восьмидесяти (не считая их перегружаемых версий). Кстати, раз наш разговор коснулся возможности перезагрузки, то сразу настрою читателя на то, что в GDI+ директивой `overload` помечен едва ли не каждый метод. Не сильно удивляйтесь, повстречав стайку

одноименных функций, отличающихся друг от друга незначительными нюансами в аргументах. Самое неприятное в этом, что иногда одноименные функции решают разнородные задачи, что вносит некоторую путаницу в процесс их систематизации.

На первый взгляд может показаться, что некоторые методы объектов GDI+ сильно перегружены параметрами. Это объясняется тем, что при построении библиотеки разработчики хотели создать систему программирования без информации о состоянии (stateless). Основная идея подобной системы в том, чтобы каждый из методов стал самодостаточным, другими словами, в его параметрах должны содержаться все необходимые для работы данные.

Класс `TGPGraphics` с разных сторон мы будем изучать на протяжении всех оставшихся глав книги, а сейчас заострим свое внимание на процессе рождения экземпляра класса и познакомимся с его конструкторами.

Создание экземпляра холста GDI+

Жизнь любого объекта начинается с его создания, в этом плане холст GDI+ не является исключением. Он может быть вызван на белый свет четырьмя способами. Наиболее популярный из них опирается на конструктор с одним-единственным параметром – дескриптором контекста устройства:

```
constructor Create(DC: HDC); reintroduce; overload;
```

Рожденный таким образом холст предоставит во власть программиста контекст графического устройства, идентифицируемый дескриптором `DC`. Практически все остальные (посвященные работе с GDI+) примеры в этой книге будут построены на следующем шаблоне:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
begin
GPGraphics:=TGPGraphics.Create(DC);
{...
графические команды
...}
GPGraphics.Free;
end;
```

Заметьте, что основной конструктор холста GDI+ не в состоянии обойтись без классического дескриптора контекста устройства, посему наша процедура также содержит этот параметр. Как правило, в примерах к книге процедура `GDIPLUS_PAINT()` вызывается из секции обработки сообщения `WM_PAINT` оконной процедуры. Таким образом, мы сможем обслуживать клиентскую область окна. Если клиентской области вам покажется недостаточно, то перенесите свое внимание на конструктор:

```
constructor Create(Wnd: HWND; ICM: BOOL{ = FALSE}); reintroduce; overload;
```

Холст ассоциируется с окном, описываемым дескриптором `Wnd`. Установив в `true` опциональный параметр ICM, мы активизируем механизм цветового

регулирования, названный в честь утвердившей его международной организации International Color Consortium.

Несколько заумный конструктор позволяет создать холст для контекста, связанного с определенным устройством:

```
constructor Create(DC: HDC; Device: THandle); reintroduce; overload;
```

С этой целью кроме дескриптора контекста в конструкторе имеется еще один параметр `Device` – указатель на устройство.

Наконец, последний из квартета конструкторов предназначен для совместной работы с изображениями, представленными потомками класса `TGPImage`:

```
constructor Create(Image: TGPImage); reintroduce; overload;
```

Благодаря нему мы сможем рисовать на поверхности векторного или растрового рисунка так, как будто перед нами лист бумаги. С той лишь разницей, что вместо карандашей нам потребуется помощь методов холста GDI+.

Каждому из конструкторов сопоставлена функция-аналог.

```
function FromHDC(DC: HDC): TGPGraphics; overload;
function FromHDC(DC: HDC; Device: THANDLE): TGPGraphics; overload;
function FromHWND(WND: HWND; ICM: BOOL = FALSE): TGPGraphics;
function FromImage(Image: TGPImage): TGPGraphics;
```

Функции вооружены идентичными параметрами и решают ту же самую задачу – создают экземпляр холста GDI+.

Представление цвета в GDI+

Представление цвета в GDI+ немного отличается от цветовой схемы GDI, так как изначально ориентировано на 32-битную модель, сочетающую в себе три классических 8-битных канала цветности RGB и альфа-канал, отвечающий за прозрачность закрашиваемого пиксела. Информация о цвете хранится в двойном слове `ARGB`:

```
Type ARGB = DWORD;
TGPColor = ARGB;
```

В младшем байте находится значение синей составляющей, во втором байте – зеленой, затем следует красная составляющая цвета и шествие завершается старшим байтом состоящим из альфа-канала (рис. 28.2). Нулевое значение альфа-канала соответствует полностью прозрачному режиму, значение 255 ($FF) – абсолютно непрозрачному режиму работы.

Для назначения цвета можно сразу воспользоваться записью в шестнадцатеричном формате, например, значение $FF0000FF соответствует непрозрач-

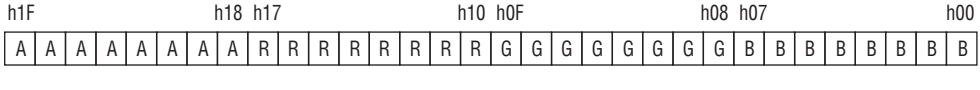


Рис. 28.2. 32-битное представление цвета в GDI+

ному чистому синему цвету, а значение $7F00FF00 – наполовину прозрачному зеленому. Если логика работы программы предполагает динамическое управление цветом, то проще воспользоваться перегружаемой функцией:

```
function MakeColor(R, G, B: Byte): ARGB; overload;
function MakeColor(A, R, G, B: Byte): ARGB; overload;
```

Если предполагается работа с непрозрачным цветом, используйте первый вариант MakeColor(), если же хочется поэкспериментировать с альфа-каналом, то к вашим услугам вторая версия функции. Ниже представлен фрагмент процедуры, демонстрирующий порядок работы с цветом GDI+.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics; //холст
    GPPen : TGPPen;         //перо
    Alpha:byte;             //коэффициент прозрачности
begin
GPGraphics:=TGPGraphics.Create(DC);
GPPen:=TGPPen.Create(0,1); //создаем перо единичной ширины
  for Alpha := 255 downto 0 do
  begin
   GPPen.SetColor(MakeColor(Alpha,0,0,255)); //изменяем прозрачность
   GPGraphics.DrawLine(GPPen,1+Alpha,Alpha,200+Alpha,Alpha); //вывод отрезка
  end;
GPGraphics.Free;
GPPen.Free;
end;
```

В рамках процедуры GDIPLUS_PAINT() нами создается объект TGPGraphics (еще раз напомню, что это аналог холста TCanvas, но для GDI+) и простейшее перо TGPPen. Затем в цикле с помощью метода DrawLine() холста мы с небольшим смещением чертим 256 отрезков. Перед каждым выводом отрезка уменьшаем прозрачность пера. Сразу оговорюсь, что это не самый эффективный способ создания градиентной заливки в GDI+, но для первого проекта такой подход простителен (рис. 28.3).

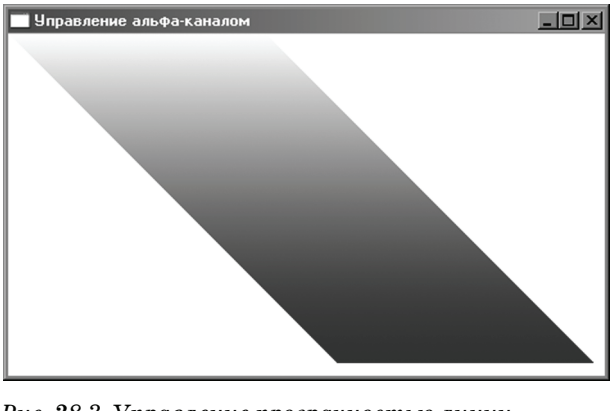


Рис. 28.3. Управление прозрачностью линии

Изъятие информации о соответствующей цветовой составляющей из двойного слова ARGB производится функциями:

```
function GetAlpha(Color: ARGB): BYTE; //значение альфа-канала
function GetRed(Color: ARGB): BYTE;   //значение красной составляющей
function GetGreen(Color: ARGB): BYTE; //значение зеленой составляющей
function GetBlue(Color: ARGB): BYTE;  //значение синей составляющей
```

Взаимные преобразования между классическим способом представления цвета в формате COLORREF и представлением цвета в GDI+ производится парой функций:

```
function ColorRefToARGB(RGB: COLORREF): ARGB;
function ARGBToColorRef(Color: ARGB): COLORREF;
```

При определении цветов совсем не обязательно лишний раз напрягать 7 миллионов колбочек, отвечающих за цветовосприятие наших глаз. Помимо возможности создавать свои собственные цвета мы в праве использовать 141 именованную константу с заранее подготовленными цветами. Константы описаны в модуле GDIPAPI, их названия начинаются с приставки «acl». Здесь вы обнаружите как стандартные названия, например `aclBlack = $FF000000`, так и весьма причудливые, например морской волны `aclSeaGreen = $FF2E8B57` или сбалансированный миндальный `aclBlanchedAlmond = $FFFFEBCD`. Когда читаешь эти названия, то понимаешь, что разработчики графического пользовательского интерфейса UNIX (знаменитая в свое время оболочка X Window) в Массачусетском технологическом институте не зря ели свой хлеб – теперь специально подобранные ими цветовые константы используются практически повсеместно.

Структуры определения координат и размеров

При проектировании GDI+ разработчики отказались от применения привычных нам по GDI и VCL структур определения координат и размеров. Естественно, сама идея координатной точки осталась неизменной, поле X описывает горизонтальную, а поле Y – вертикальную составляющую. Основная причина отказа от классических структур заключается в необходимости хранения данных в вещественных значениях. На смену описывающему координаты точки типу данных `TPoint` призваны `TGPPoint` и `TGPPointF`:

```
type
  PGPPoint = ^TGPPoint;
  TGPPoint = packed record
    X : Integer;
    Y : Integer;
  end;
```

```
type
  PGPPointF = ^TGPPointF;
  TGPPointF = packed record
    X : Single;
    Y : Single;
  end;
```

Структуры практически аналогичны, за тем лишь исключением, что первая используется для хранения целочисленных данных, а вторая описывает точку в вещественном виде.

При заполнении структуры разрешено как прямое обращение к ее полям, так и применение вспомогательных функций:

```
function MakePoint(X, Y: Integer): TGPPoint; overload;
function MakePoint(X, Y: Single): TGPPointF; overload;
```

При определении геометрических размеров вместо TSize следует воспользоваться структурами:

```
type
  PGPSize = ^TGPSize;
  TGPSize = packed record
    Width  : Integer;
    Height : Integer;
  end;
```

```
Type
  PGPSizeF = ^TGPSizeF;
  TGPSizeF = packed record
    Width  : Single;
    Height : Single;
  end;
```

Здесь мы вновь встретим два подхода – хранение целочисленных и действительных размеров.

Существенным изменениям была подвергнута структура TRect. С левой верхней точкой структуры все осталось по-прежнему, но в GDI+ вместо определения нижней правой точки прямоугольной области программист указывает ширину и высоту прямоугольника. Полагаю, что наличие целочисленной и действительной версий структуры читателю уже не покажется удивительным.

```
Type
  PGPRect = ^TGPRect;
  TGPRect = packed record
    X     : Integer;
    Y     : Integer;
    Width : Integer;
    Height: Integer;
  end;
```

```
Type
  PGPRectF = ^TGPRectF;
  TGPRectF = packed record
    X     : Single;
    Y     : Single;
    Width : Single;
    Height: Single;
  end;
```

Для определения координат прямоугольной области в приемлемом для GDI+ виде можно воспользоваться группой перегружаемых функций. Если мы работаем с целыми числами, нам потребуются методы:

```
function MakeRect(X, Y, Width, Height: Integer): TGPRect; overload;
function MakeRect(Location: TGPPoint; Size: TGPSize): TGPRect; overload;
function MakeRect(const Rect: TRect): TGPRect; overload;
```

Для вещественной формы представления:

```
function MakeRect(X, Y, Width, Height: Single): TGPRectF; overload;
function MakeRect(Location: TGPPointF; Size: TGPSizeF): TGPRectF; overload;
```

Отладка проектов GDI+

Для организации контроля над корректностью выполнения той или иной операции над объектами GDI+ многие из них вооружены методом:

```
function GetLastStatus: TStatus;
```

Указанный метод можно обнаружить у кистей, изображений, объектов, обслуживающих шрифты, и у ряда других объектов GDI+. В GDI+ версии 1.1 насчитывается 21 значение статуса операции (табл. 28.1).

Таблица 28.1. Статус операции TStatus

Имя константы	Значение	Описание
`Ok`	0	Вызов метода завершен корректно
`GenericError`	1	Признак общей ошибки
`InvalidParameter`	2	Ошибочный аргумент метода
`OutOfMemory`	3	Нехватка памяти
`ObjectBusy`	4	Объект захвачен другим потоком
`InsufficientBuffer`	5	Переполнение буфера
`NotImplemented`	6	Вызываемый метод не может быть выполнен
`Win32Error`	7	Вызов метода привел к возникновению ошибки Windows
`WrongState`	8	Указывает на то, что в данном состоянии объект не способен выполнить вызываемый метод
`Aborted`	9	Выполнение метода было прервано
`FileNotFound`	10	Файл не обнаружен
`ValueOverflow`	11	Вызов метода привел к переполнению какого-то значения
`AccessDenied`	12	Доступ был отвергнут
`UnknownImageFormat`	13	Неизвестный формат изображения
`FontFamilyNotFound`	14	Не обнаружено семейство запрашиваемого шрифта
`FontStyleNotFound`	15	Семейство шрифта не обладает указанным стилем
`NotTrueTypeFont`	16	Данный шрифт не TrueType
`UnsupportedGdiplusVersion`	17	Несовместимая версия GDI+
`GdiplusNotInitialized`	18	Объекты GDI+ не прошли инициализацию (для этого необходим вызов метода `GdiplusStartup`)
`PropertyNotFound`	19	Запрашиваемое свойство не обнаружено
`PropertyNotSupported`	20	Свойство не поддерживается
`ProfileNotFound`	21	Отсутствует профайл, необходимый для сохранения рисунка в цветовом формате CMYK

Как правило, явного обращения к функции `GetLastStatus()` в наших программах не требуется. Метод вызывается автоматически в рамках методов классов GDI+. Но без него не обойтись в двух случаях: во-первых, при отладке проекта и, во-вторых, при разработке собственных потомков классов GDI+.

РЕЗЮМЕ

Библиотека GDI+ представляет собой объектно-ориентированную оболочку для графических объектов Windows. Это не просто «перевод» функций GDI на рельсы современной ООП технологии. Разработчики обновленной графической библиотеки значительно расширили ее кругозор, дополнив новыми инструментами и уникальными возможностями. Как следствие, существенно упростился процесс разработки графического интерфейса современных приложений и повысилась надежность программ.

Вместе с тем GDI+ ни в коем случае нельзя назвать заменой «устаревшего» GDI. Тому несколько причин. Первую мы уже назвали, GDI+ всего лишь объектно-ориентированное воплощение графического механизма Windows и, как следствие, не в силах функционировать без GDI. Вторая причина в том, что, к сожалению, обновленная графическая библиотека не смогла вобрать в себя весь богатейший набор возможностей классического GDI. Например, у библиотеки GDI+ весьма скромные способности по управлению растровыми операциями, упрощенное представление о настройке страничных координат и координат устройства, ограничения по работе регионами отсечения и т. д.

Основную партию в симфонии GDI+ исполняет холст `TGPGraphics`. По духу он весьма близок к штатному холсту `TCanvas` из состава Delphi. Холст GDI+ так же, как и холст VCL инкапсулирует основной графический объект Windows – контекст графического устройства, но с той лишь разницей, что `TGPGraphics` ориентирован на работу с новой библиотекой gdiplus.dll, а не gdi32.dll.

29

Кисти GDI+

Не знаю, насколько сильно вы разочаруетесь, узнав, что назначение кисти GDI+ ничем не отличается от назначения кисти GDI. И здесь, и там кисть представляет собой основной инструмент для заливки областей. Но если сравнивать возможности нового поколения кистей с их «древними» собратьями, то вы увидите огромную разницу. В первую очередь хочется отметить следующие особенности:

1. Значительно расширен спектр стандартных узоров кистей.
2. Кисти GDI+ обеспечивают градиентную заливку. Это позволяет закрашивать сложные фигуры оттенками с различными законами распределения цвета.
3. Ряд кистей GDI+ поддерживает механизм двумерных аффинных преобразований, что позволяет без труда осуществлять операции поворота, переноса, масштабирования, деформации и отражения заливки.

Если сопоставить особенности реализации старого и нового парка кистей, то в этом случае GDI+ просто окажется вне конкуренции. В обновленной графической системе насчитывается целых шесть классов, специализирующихся на формировании пяти разновидностей кистей. Это, пожалуй, самая богатая плеяда классов (рис. 29.1), ни один другой объект GDI+ более не имеет столько вариантов исполнения.

Основу иерархии составляет абстрактный класс `TGPBrush`. Класс не в состоянии создать жизнеспособный экземпляр кисти, но зато он реализует стандартные функции, создающие и разрушающие объекты, и обладает рядом полезных методов, применяемых во всех его потомках. Конструкторы (их два) и деструктор `TGPBrush` мы обсуждать не станем, так как они кардинальным образом переписаны в дочерних классах, и заострим свое внимание на трех его опубликованных методах.

Основной из них позволит выяснить тип кисти:

```
function GetType: TBrushType;
```

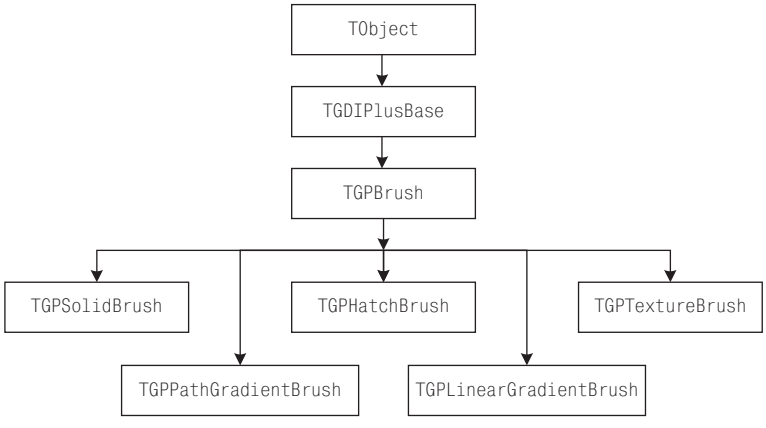


Рис. 29.1. Иерархия кистей в GDI+

Вы наверняка догадались, что тип кисти определяется ее классом.

```
TBrushType = (
 BrushTypeSolidColor,    //сплошная кисть
 BrushTypeHatchFill,     //узорная кисть
 BrushTypeTextureFill,   //текстурная кисть
 BrushTypePathGradient,  //градиентная кисть сложной формы
 BrushTypeLinearGradient //кисть с линейной градиентной заливкой
);
```

Виртуальный метод:

```
function Clone: TGPBrush; virtual;
```

позволит нам создавать дубликаты экземпляра любой кисти.

Последний заслуживающий нашего внимания метод:

```
function GetLastStatus: TStatus;
```

нам уже знаком. Функция проинформирует о том, насколько корректно закончилось выполнение той или иной операции, лучшее, что сможет возвратить функция, – значение 0 (константа `OK`), все остальные значения говорят о том, что произошла ошибка.

Знакомиться с классами кистей мы начнем по мере возрастания их возможностей, от обычной рабочей лошадки – элементарной кисти со сплошной заливкой `TGPSolidBrush` – до рафинированных аристократок – градиентных кистей.

Сплошная кисть TGPSolidBrush

Из всего многообразия кистей GDI+ наиболее непритязательным характером обладает сплошная кисть `TGPSolidBrush`. Конструктор кисти:

```
constructor Create(Color: TGPColor);
```

требует лишь указания цвета новоиспеченного объекта. Однако это ни в коем случае не аналог обычной кисти GDI, формируемой за счет функции Windows API CreateSolidBrush(). Разница хотя бы в том, что кисть TGPSolidBrush (как, впрочем, и все кисти GDI+) работает с 32-битным цветом в формате TGPColor, другими словами, поддерживает операции с альфа-каналом.

Еще два заслуживающих внимания метода связаны с цветовой характеристикой кисти. Для уточнения текущего значения цвета предназначена функция:

```
function GetColor(out Color: TGPColor): TStatus;
```

Для изменения раскраски экземпляра кисти нет необходимости создавать новый объект. Вместо этого достаточно передать новое значение цвета в метод SetColor().

```
function SetColor(Color: TGPColor): TStatus;
```

Представленные ниже строки кода демонстрируют порядок заливки прямоугольной области Rct красной кистью.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPSolidBrush:TGPSolidBrush;
    X:integer;
    Alpha:byte;
    Rct:TGPRect;
begin
GPGraphics:=TGPGraphics.Create(DC);
Alpha:=255;
Rct.X:=5;Rct.Y:=5; Rct.Width:=5; Rct.Height:=50;
GPSolidBrush:=TGPSolidBrush.Create(MakeColor(Alpha,255,0,0));
  While Alpha>0 do
    begin
    GPGraphics.FillRectangle(GPSolidBrush,Rct);
    DEC(Alpha,5);
    GPSolidBrush.SetColor(MakeColor(Alpha,255,0,0));
    Rct.X:=Rct.X+Rct.Width+1;
    end;
GPSolidBrush.Free;
GPGraphics.Free;
end;
```

Для того чтобы продемонстрировать возможность управления прозрачностью кисти, в рамках цикла мы несколько раз смещаем закрашиваемую область вправо с одновременным изменением коэффициента прозрачности.

Узорная кисть TGPHatchBrush

Второе место в славном строю кистей GDI+ занимает узорная кисть TGPHatchBrush. Графический интерфейс GDI+ позволяет создавать кисть с 54 вариантами заранее предопределенной штриховки (что не сравнить с шестью штатными штриховками GDI). Вид узора описывается типом данных THatchStyle,

который вправе принимать любое значение в пределах от 0 до 53. Каждому числовому значению сопоставлена именованная константа: HatchStyleHorizontal=0, HatchStyleVertical=1 и т. д. На перечислении названий констант мы останавливаться не станем, при желании вы их обнаружите в модуле GDIAPI или в SDK.

Для создания экземпляра кисти используйте конструктор:

```
constructor Create(HatchStyle: THatchStyle; ForeColor: TGPColor;
                   BackColor: TGPColor = aclBlack); reintroduce; overload;
```

Метод обладает тремя параметрами, здесь HatchStyle – уже знакомый нам стиль штриховки; ForeColor и BackColor – цвета пикселов узора и фона будущей кисти. По умолчанию отвечающий за цвет фона аргумент необязателен для заполнения, в этом случае фон станет траурно-черным.

Ниже предложен фрагмент кода, демонстрирующий порядок формирования узорной кисти. В рамках цикла for..to..do нами последовательно создается экземпляр кисти GPHatchBrush с очередным вариантом штриховки (определяемым параметром HS). Затем посредством вызова метода FillRectangle() осуществляется заливка кистью квадрата с размерами сторон 50×50 пикселей, после чего кисть уничтожается.

```
Procedure GDIPLUS_PAINT(DC:HDC; ClientRect:TRect);
var GPGraphics:TGPGraphics;
    GPHatchBrush:TGPHatchBrush;
    Rct:TGPRect;
    HS:integer;
    s:string;
begin
GPGraphics:=TGPGraphics.Create(DC);
Rct:=MakeRect(1,1,50,50);
  for HS := 0 to 53 do
   begin
    GPHatchBrush:=TGPHatchBrush.Create(THatchStyle(HS),aclBlack,aclWhite);

    GPGraphics.FillRectangle(GPHatchBrush,Rct);
    s:=IntToStr(HS);
    TextOut(DC,Rct.X,Rct.Y+Rct.Height+1,pChar(s),Length(s));

    if (Rct.x+Rct.Width*2+1<ClientRect.Right) then
        INC(Rct.X,Rct.Width+1)
        else
        Rct:=MakeRect(1,Rct.Y+Rct.Height+15,50,50
     GPHatchBrush.Free;
    end;
GPGraphics.Free;
end;
```

Хотя в обновленной графической библиотеке нет аналога кисти NULL_BRUSH *(*HOLLOW_BRUSH*) из состава GDI, в GDI+ с ролью пустышки справится кисть* HatchStyleTotal *(значение 53). Как бы это ни показалось странным, но эта узорная кисть не обладает узором и поэтому не оставляет следов на экране.*

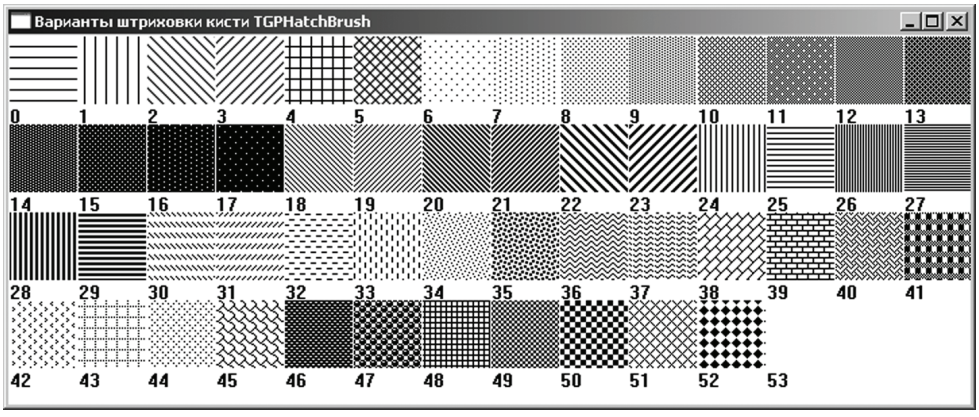


Рис. 29.2. Варианты штриховки кисти GDI+

Изучив рис. 29.2 (представляющий экранный снимок работающей программы), можно сделать вывод, что первые 6 типов узора полностью соответствуют штриховке (см. рис. 4.2) классической кисти GDI. Но остальные четыре с лишним десятка вариантов кисти способны удовлетворить чувство прекрасного даже самого притязательного эстета.

Узорные кисти обладают иммунитетом к аффинным преобразованиям. Поэтому если вы захотите закрасить узорной кистью прямоугольную область и одновременно дадите команду, например, на деформацию области, то преобразования затронут только границы области, но никак не скажутся на внешнем виде узора!

У узорной кисти осталось еще три заслуживающих нашего внимания метода:

```
function GetHatchStyle: THatchStyle; //узор
function GetForegroundColor(out color: TGPColor): TStatus; //цвет фона
function GetBackgroundColor(out color: TGPColor): TStatus; //цвет штриховки
```

Все они информационные и сообщают программисту данные об узоре кисти и цветах штриховки и фона.

Управление базовой точкой узора

У процесса заливки произвольной замкнутой области кистью есть одна примечательная особенность. При закрашивании области узор кисти тиражируется слева направо и сверху вниз. Казалось, что по логике вещей при заливке, скажем, прямоугольника начальная точка узора должна выставляться по верхнему левому углу закрашиваемой геометрической фигуры, но на самом деле это не так. Пусть даже закрашиваемая область находится где-то в середине окна, но при расчете штриховки Windows за опорную точку принимает левую верхнюю точку в клиентской области окна, в котором осуществляется рисование. После этого узор размножается по всей клиентской области, правда, видимым он становится только после попадания в закраши-

ваемый район. Как следствие, рисунок узора зачастую выводится со смещением на 1 или несколько пикселов.

Для борьбы с таким «безобразием» следует привлекать класс TGPGraphics. В качестве дополнительной опции по управлению процессом заливки областей кистью TGPHatchBrush холст GDI+ предлагает способ, позволяющий сместить опорную точку штриховки по осям X и Y. Для этого используется метод:

```
function SetRenderingOrigin(x, y: Integer): TStatus;
```

Наш новый герой выступает достойным продолжателем славных дел функции GDI SetBrushOrgEx (рассмотренной в 4 главе этой книги). Следующие строки кода демонстрируют порядок работы с методом:

```
HatchBrush:=TGPHatchBrush.Create(HatchStyleHorizontalBrick,aclYellow,aclRed);
GPGraphics.FillRectangle(HatchBrush,10,10,100,100);
GPGraphics.SetRenderingOrigin(0, 2);
GPGraphics.FillRectangle(HatchBrush,160,10,100,100);
```

Если читатель вооружится лупой (см. рис. 29.3 на цв. вклейке, с. 420), то заметит, что благодаря смещению узора кисти на две единицы вниз по оси Y кирпичная заливка правой прямоугольной области стала осуществляться, начиная с изображения целого кирпичика, хотя до этого выводилась его смещенная половинка.

За справкой о текущих настройках базовой точки узора следует обращаться к функции:

```
function GetRenderingOrigin(out x, y: Integer): TStatus;
```

У узорной кисти имеется три существенных недостатка: во-первых, пусть большой, но все-таки ограниченный набор шаблонов, во-вторых, в узоре кисти не может быть использовано более двух цветов и, в-третьих, узорная кисть не дружит с аффинными преобразованиями.

Текстурная кисть TGPTextureBrush

С работой, которую не смогли выполнить сплошная и узорная кисть, без труда справится третья по счету кисть – TGPTextureBrush. Текстурная кисть приходит на помощь в той ситуации, когда нам не подходит ни один из заранее предопределенных узоров. Еще одно достоинство текстурной кисти в том, что ее узор способен сочетать практически неограниченное число цветов. Ко всему прочему. кисть TGPTextureBrush не имеет ничего против аффинных преобразований. На столь серьезные подвиги наши предыдущие знакомые, к сожалению, неспособны.

Убедившись, что в нашем проекте следует применять текстурную кисть, выбираем один из семи конструкторов класса TGPTextureBrush, позволяющих собрать требуемую кисть из любого изображения. Простейшему из них:

```
constructor Create(Image: TGPImage; WrapMode:
                   TWrapMode = WrapModeTile); reintroduce; overload;
```

достаточно двух параметров. Это изображение Image и параметр WrapMode, определяющий закон повторения изображений при заливке больших площадей. К параметру WrapMode мы еще вернемся, а пока, т. к. мы еще не рассматривали порядок работы с экземпляром класса TGPImage, предложу небольшой пример, демонстрирующий процесс создания простейшей текстурной кисти на основе файла, хранящего битовый образ.

```
Procedure GDIPLUS_PAINT(DC:HDC;Rect:TRect);
var GPGraphics:TGPGraphics;
    GPImage:TGPImage;
    GPTextureBrush:TGPTextureBrush;
begin
  GPGraphics:=TGPGraphics.Create(DC);
  GPImage:=TGPImage.Create('..\Graphics\fly.bmp');
  GPTextureBrush:=TGPTextureBrush.Create(GPImage);
  GPGraphics.FillRectangle(GPTextureBrush,0,0,Rect.Right,Rect.Bottom);
  GPGraphics.Free;
  GPTextureBrush.Free;
  GPImage.Free;
end;
```

Рисунок кисти формируется на основе находящегося в библиотеке к этой книге файла битового образа 'fly.bmp', создав кисть, мы закрашиваем ею всю клиентскую область окна.

Наиболее любознательному программисту изображение, на основе которого строится кисть, разрешается «потрогать» руками. Для этого потребуется небольшая помощь функции:

```
function GetImage: TGPImage;
```

Она возвратит экземпляр объекта TGPImage. Но у кисти нет метода с условным наименованием «SetImage», позволяющего поменять изображение на лету. Поэтому если текстура вас не устраивает, необходимо создавать новый экземпляр кисти.

Теперь вновь возвратимся к конструктору, точнее говоря, к анонсированному несколько ранее параметру WrapMode, определяющему порядок повторения узора, создавая из него мозаику. Параметр вправе принимать любое значение в диапазоне от 0 до 4. По умолчанию он установлен в состояние WrapModeTile, что соответствует обычному покрытию области повторяющейся картинкой.

```
TWrapMode = (
  WrapModeTile,        // 0 - повторение рисунков
  WrapModeTileFlipX,   // 1 - перевороты по вертикальной оси
  WrapModeTileFlipY,   // 2 - перевороты по горизонтальной оси
  WrapModeTileFlipXY,  // 3 - перевороты по всем осям
  WrapModeClamp        // 4 - заливка отсутствует
);
```

Для управления порядком вывода имеется две функции информирования о текущем режиме и смены режима заливки:

```
function GetWrapMode: TWrapMode;
function SetWrapMode(WrapMode: TWrapMode): TStatus;
```

Особенности поведения кисти в зависимости от состояния параметра WrapMode продемонстрированы на рис. 29.4.

Не сбивайте дыхание, нас дожидаются еще шесть конструкторов текстурной кисти. Вторая вариация конструктора содержит пару функций с дополнительным параметром DstRect:

```
constructor Create(Image: TGPImage; WrapMode: TWrapMode;
                   DstRect: TGPRect); reintroduce; overload;
constructor Create(Image: TGPImage; WrapMode: TWrapMode;
                   DstRect: TGPRectF); reintroduce; overload;
```

Этот параметр позволяет уточнить, какая именно область изображения Image будет использована в качестве текстуры. Того же результата можно добиться несколько иначе с помощью другой группы конструкторов:

```
constructor Create(Image: TGPImage; WrapMode: TWrapMode; dstX,dstY,
                   dstWidth, dstHeight: Single); reintroduce; overload;
constructor Create(Image: TGPImage; WrapMode: TWrapMode; dstX, dstY,
                   dstWidth, dstHeight: Integer); reintroduce; overload;
```

В этом случае размеры исходной области назначаются четырьмя параметрами dstX, dstY, dstWidth, dstHeight.

Широким спектром возможностей обладают конструкторы, позволяющие управлять атрибутами исходного изображения. В результате может быть создана текстурная кисть с уникальными характеристиками:

```
constructor Create(Image: TGPImage; DstRect: TGPRect; ImageAttributes:
                   TGPImageAttributes = nil); reintroduce; overload;
constructor Create(Image: TGPImage; DstRect: TGPRectF; ImageAttributes:
                   TGPImageAttributes = nil); reintroduce; overload;
```

Прямоугольник DstRect определяет, какая часть исходного рисунка Image будет включена в состав кисти. Порядок вывода изображения назначается па-

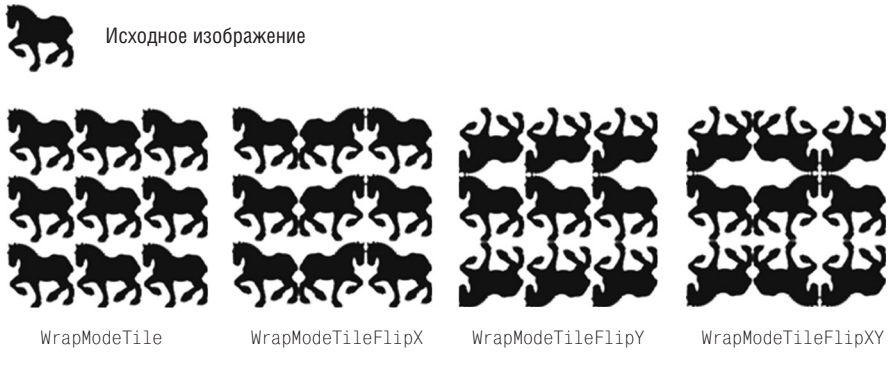


Рис. 29.4. Варианты повторения изображения при работе с текстурной кистью

раметром ImageAttributes, в который должен быть направлен экземпляр класса TGPImageAttributes (это специалист по цветовой фильтрации). Таким образом, можно управлять цветовыми характеристиками кисти, например осуществлять гамма-коррекцию. К сожалению, у последней пары конструкторов TGPTextureBrush отсутствует параметр WrapMode, определяющий закон повторения текстуры при заливке областей больших размеров, чем исходное изображение. Указанный недостаток несколько ограничивает область применения кистей, сформированных этими конструкторами.

Используемая для представления изображения цветовая модель идеализирует графическое устройство. Ее работа основывается на предположении, что между значением цветового компонента и видимым на экране цветом имеется линейная связь. На самом деле устройства отображения не в состоянии реагировать линейно на поступающий сигнал. Благодаря гамма-коррекции мы сможем учесть нелинейные характеристики физического устройства.

В нашем примере на вход процедуры кроме дескриптора контекста передается значение гаммы.

```
Procedure GDIPLUS_PAINT(DC:HDC;Gamma:Single);
var GPGraphics:TGPGraphics;
    GPImage:TGPImage;
    GPTextureBrush:TGPTextureBrush;
    GPRect:TGPRect;
    IAttr:TGPImageAttributes;
    W,H:integer;
begin
  GPGraphics:=TGPGraphics.Create(DC);
  GPImage:=TGPImage.Create('..\Graphics\comp.bmp');
  W:=GPImage.GetWidth; H:=GPImage.GetHeight;
  GPRect:=MakeRect(0,0,W,H);

  IAttr:=TGPImageAttributes.Create;
  IAttr.SetGamma(Gamma,ColorAdjustTypeBitmap);
  GPTextureBrush:=TGPTextureBrush.Create(GPImage,GPRect,IAttr);
  GPGraphics.FillRectangle(GPTextureBrush,GPRect);
  TextOut(DC,W,H div 2,pChar('Кисть после коррекции гаммы'),27);

  GPRect:=MakeRect(0,H,W,H);
  GPGraphics.DrawImage(GPImage,GPRect); //исходное изображение
  TextOut(DC,W,H+H div 2,pChar('Исходный рисунок'),16);

  GPGraphics.Free;
  GPTextureBrush.Free;
  GPImage.Free;
  IAttr.Free;
end;
```

Гамма-коррекция широко применяется в современных графических редакторах. Она позволяет подчеркнуть детали слабоконтрастных изображений без оказания существенного влияния на области теней и бликов. Визуальный эффект от изменения параметров гаммы таков: значения меньше 1 осветля-

ют рисунок, больше 1 – делают изображение более темным (см. рис. 29.5 на цв. вклейке, с. 420).

Более подробно о различных методах цветовой коррекции мы побеседуем в последней главе этой книги, в разделе, посвященном классу `TGPImageAttributes`.

Аффинные преобразования кистей

Технология GDI+ изначально ориентирована на работу в мировых координатах. Именно поэтому текстурная кисть снабжена существенным перечнем методов, предназначенных для организации работы с двумерными аффинными преобразованиями и с матрицей преобразований `TGPMatrix` (табл. 29.1).

Таблица 29.1. Аффинные преобразования кисти

Функция	Описание
`function SetTransform(Matrix: TGPMatrix): TStatus;`	Аффинное преобразование задается матрицей `Matrix`.
`function GetTransform(Matrix: TGPMatrix): TStatus;`	Получение информации о состоянии ячеек матрицы аффинных преобразований `Matrix`.
`function MultiplyTransform(Matrix: TGPMatrix; order: TMatrixOrder = MatrixOrderPrepend): TStatus;`	Дополнение текущих установок аффинных преобразований новыми данными из матрицы `Matrix`.
`function TranslateTransform(dX, dY: Single; order: MatrixOrder = MatrixOrderPrepend): TStatus;`	Сдвиг рисунка кисти. Параметр `dX` – по оси X, `dY` – по оси Y.
`function ScaleTransform(sX, sY: Single; order: TMatrixOrder = MatrixOrderPrepend): TStatus;`	Масштабирование. Коэффициент по оси X определяется параметром `sX`, по оси Y – `sY`.
`function RotateTransform(Angle: Single; order: TMatrixOrder = MatrixOrderPrepend): TStatus;`	Поворот узора по часовой стрелке на угол `Angle`.
`function ResetTransform: TStatus;`	Сброс преобразований.

Аффинные преобразования поддерживаются не только текстурной кистью `TGPTextureBrush`*, но и ее коллегами по цеху: градиентными кистями* `TGPLinearGradientBrush` *и* `TGPPathGradientBrush`*. Поэтому перечисленные выше методы имеются и у этих объектов GDI+.*

Представленные в табл. 29.1 методы позволят нам без существенных затрат сил вносить требуемые геометрические искажения в рисунок заливки. В предложенном ниже примере продемонстрирована работа с частью из рассмотренных выше функций.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPImage:TGPImage;
    GPTextureBrush:TGPTextureBrush;
    GPMatrix: TGPMatrix;
begin
  GPGraphics:=TGPGraphics.Create(DC);
  GPImage:=TGPImage.Create('demo.bmp');
  {исходная кисть}
  GPTextureBrush:=TGPTextureBrush.Create(GPImage);
  GPGraphics.FillRectangle(GPTextureBrush,0,0,100,100);

  {масштабирование}
  GPTextureBrush.ScaleTransform(2,2);
  GPGraphics.FillRectangle(GPTextureBrush,110,0,100,100);

  {поворот}
  GPTextureBrush.ResetTransform;
  GPTextureBrush.RotateTransform(90);
  GPGraphics.FillRectangle(GPTextureBrush,220,0,100,100);

  {поперечная деформация}
  GPMatrix:=TGPMatrix.Create(1,0.3,0.3,1,0,0);
  GPTextureBrush.SetTransform(GPMatrix);
  GPGraphics.FillRectangle(GPTextureBrush,330,0,100,100);

  GPGraphics.Free;
  GPTextureBrush.Free; GPImage.Free; GPMatrix.Free;
end;
```

В ходе выполнения кода мы создаем один-единственный экземпляр текстурной кисти, а затем, с помощью механизма двумерных аффинных преобразований, воздействуем на текстуру: увеличиваем, поворачиваем на 90 градусов и деформируем рисунок (рис. 29.6).

Более подробно о матрице аффинных преобразований `TGPMatrix` *мы поговорим в главе 34 «Координатные системы и преобразования в GDI+».*

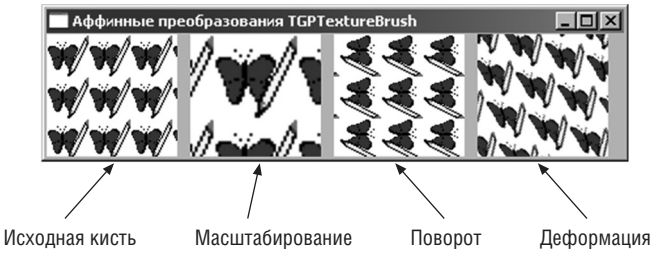


Рис. 29.6. Аффинные преобразования рисунка кисти

Кисть с линейной градиентной заливкой TGPLinearGradientBrush

Кисть `TGPLinearGradientBrush` позволяет производить заливку области плавно изменяющимся цветом. С технической точки зрения кисть с линейной градиентной заливкой описывается двумя параллельными границами – начальной и конечной. С каждой из границ ассоциируется свое цветовое значение Color1 и Color2. Градиент представляет собой вектор (рис. 29.7), выходящий из стартовой границы под углом 90 градусов и направленный к конечной границе. По мере удаления от начала закрашиваемой области и приближения к ее завершению цвет заливки плавно изменяется от Color1 к Color2, причем все точки заливки, параллельные границам, имеют один и тот же цвет.

Для создания экземпляра градиентной кисти предусмотрено три пары конструкторов, различающихся способом определения направления заливки:

1. Фиксированное направление `TLinearGradientMode`.
2. Направление устанавливается стартовой и конечной точками.
3. Направление указывается величиной угла.

Отличие конструкторов внутри пары незначительное. Они разнятся только тем, что при определении области и направления заливки применяются целые или действительные числа. Вместе с тем у всех вариантов конструкторов имеется общая черта – способ определения значений стартового и конечного цветов. Для этого предназначены параметры `Color1` и `Color2`.

Знакомство с вариантами формирования кисти начнем с простейшей пары конструкторов:

```
constructor Create(Rect: TGPRect; Color1, Color2: TGPColor;
                   Mode: TLinearGradientMode); reintroduce; overload;
constructor Create(Rect: TGPRectF; Color1, Color2: TGPColor;
                   Mode: TLinearGradientMode); reintroduce; overload;
```

Первый параметр определяет размеры прямоугольной области, в которой планируется провести градиентную заливку. Направление заливки определяется параметром `Mode`, вариантов немного (всего четыре) – они представлены в табл. 29.2.

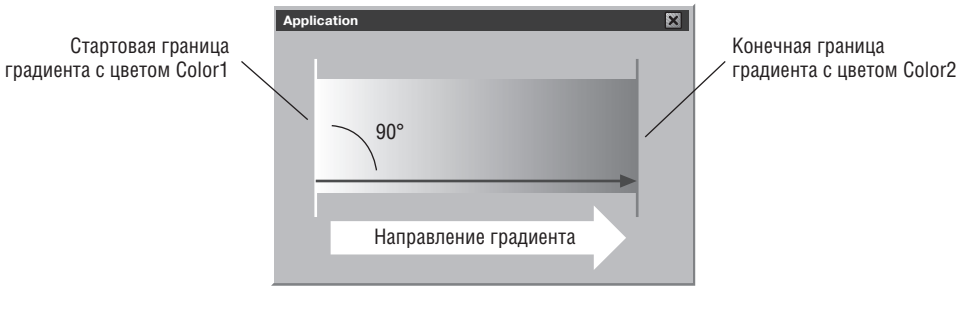


Рис. 29.7. Технология линейной градиентной заливки

На рис. 29.8 (см. цветную вклейку, с. 421) вы увидите графическую иллюстрацию всех вариантов заливки с фиксированным направлением.

Таблица 29.2. Направления заливки TLinearGradientMode

Константа	Значение	Направление заливки
LinearGradientModeHorizontal	0	Цвет изменяется от color1 к color2 по горизонтали слева направо.
LinearGradientModeVertical	1	Цвет изменяется сверху вниз.
LinearGradientModeForwardDiagonal	2	Градиент направлен из левого верхнего угла в правый нижний.
LinearGradientModeBackwardDiagonal	3	Направление изменения цвета из правого верхнего угла в правый нижний.

Сразу отметим, что экземпляр кисти позволяет динамически изменять цвета градиента без пересоздания нового экземпляра кисти. Результат достигается с помощью метода:

```
function SetLinearColors(Color1, Color2: TGPColor): TStatus;
```

в который передаются новые значения цветов. У этой функции имеется коллега, предназначенная для уточнения, какие именно цветовые оттенки в данный момент применяются при построении градиентной заливки:

```
function GetLinearColors(out Color1, Color2: TGPColor): TStatus;
```

С областью изменения градиента несколько сложнее. Ее резрешено изменить, только сформировав новую кисть. Максимум, чего можно добиться от имеющегося экземпляра, – информации о границах прямоугольника.

```
function GetRectangle(out Rect: TGPRectF): TStatus; overload;
function GetRectangle(out Rect: TGPRect): TStatus; overload;
```

При определении прямоугольника заливки Rect *в конструкторе кисти имейте в виду, что* Rect *в первую очередь назначает не область вывода, а область, в пределах которой цвет* Color1 *должен плавно перейти в* Color2*. При заливке области мы имеем право закрасить меньшую или большую часть поверхности, чем задано в полях структуры* Rect*, но в последнем случае мы получим многократно повторяющуюся заливку.*

Наиболее гибким способом определения градиента обладает группа конструкторов, позволяющих указывать направление изменения цвета по двум точкам:

```
constructor Create(const Point1, Point2: TGPPoint;
                   Color1, Color2: TGPColor); reintroduce; overload;
constructor Create(const Point1, Point2: TGPPointF;
                   Color1,Color2: TGPColor); reintroduce; overload;
```

В первой паре конструкторов вектор заливки определяют точки Point1 и Point2. Порядок работы с подобным конструктором представлен в очередном примере.

```
Procedure GDIPLUS_PAINT(DC:HDC;Rect:TRect);
var GPGraphics:TGPGraphics;
    GPLinearGradientBrush:TGPLinearGradientBrush;
    point1,point2: TGPPoint;
begin
   GPGraphics:=TGPGraphics.Create(DC);
   point1:=MakePoint(0,0);
   point2:=MakePoint(Rect.Right,Rect.bottom);
   GPLinearGradientBrush:=TGPLinearGradientBrush.Create(point1,point2,
                                                        aclBlue,aclWhite);
   GPGraphics.FillRectangle(GPLinearGradientBrush,0,0,Rect.Right,Rect.Bottom);
  GPGraphics.Free;
  GPLinearGradientBrush.Free;
end;
```

Мы формируем кисть с плавной заливкой, изменяющейся от синего к белому цвету. Градиент заливки исходит из левого верхнего угла окна и следует в нижний правый угол. Затем посредством обращения к методу FillRectangle() холста GDI+ закрашиваем кистью клиентскую область окна.

Заметьте, что если пространство закрашиваемой области (например, с помощью метода FillRectangle) превышает область, определенную градиентной кисти при ее создании, то GDI+ не станет мудрить, а просто размножит заливку на все доступное пространство так, как продемонстрировано на рис. 29.9.

Обратимся к заключительной паре конструкторов линейной градиентной кисти.

```
constructor Create(Rect: TGPRect; Color1, Color2: TGPColor;
                   Angle: Single; IsAngleScalable: BOOL = FALSE); overload;
constructor Create(Rect: TGPRectF; Color1, Color2: TGPColor; Angle: Single;
                   IsAngleScalable: BOOL = FALSE); overload;
```

Здесь направление заливки определяется значением угла Angle и параметром IsAngleScalable. В режиме работы по умолчанию (IsAngleScalable=FALSE) плавный переход цветов осуществляется вдоль вектора, исходящего из верхнего левого точки прямоугольника Rect под углом Angle к горизонталь-

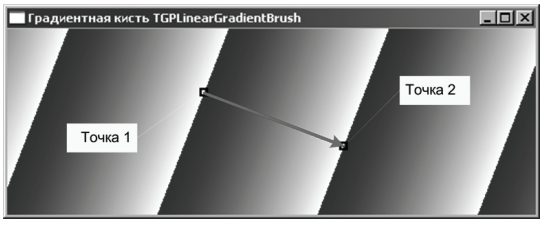


Рис. 29.9. Многократный повтор градиентной заливки

ной оси. При установке последнего параметра конструктора в TRUE на сцену выходит новая формула пересчета. В этом случае угол между горизонтальной осью и вектором заливки рассчитывается следующим образом:

$$\beta = \arctan\left(\frac{rect.widht}{rect.height} \times \tan(angle)\right) \quad (29.1)$$

Заметьте, что указанная формула применима только для угла angle менее 90 градусов.

Помимо конструкторов линейная градиентная кисть обладает существенным набором методов, позволяющих управлять основными характеристиками объекта. Из всей совокупности методов стоит особо выделить две группы:

1. Методы управления аффинными преобразованиями.
2. Методы корректировки цвета заливки.

Первая группа нам уже знакома – названия и задачи методов, определяющих порядок двумерных аффинных преобразований, у градиентной кисти аналогичны методам TGPTextureBrush.

Корректировка цвета градиентной заливки

Рассмотрение методов корректировки цвета заливки начнем с уже знакомой нам гамма-коррекции. Изначально коррекция отключена. Это можно проверить, обратившись к информационной функции:

```
function GetGammaCorrection: BOOL;
```

Включение или отключение режима коррекции производится методом:

```
function SetGammaCorrection(useGammaCorrection: BOOL): TStatus;
```

У созданной по умолчанию кисти по мере удаления от стартовой границы и приближения к конечной границе цвет изменяется равномерно от Color1 к Color2. Технология GDI+ позволяет нам отказаться от плавного перехода между цветами, вместо этого программист сможет ускорить или наоборот – замедлить скорость смены цвета на определенном участке закрашиваемой поверхности. Для этого предназначена функция:

```
function SetBlend(pBlendFactors, pBlendPositions: pSingle;
                  Count: Integer): TStatus;
```

Параметр pBlendPositions представляет собой указатель на массив, содержащий данные о позициях, в которых будет изменен порядок назначения цвета заливки. При определении таких точек надо помнить, что вся протяженность области заливки нормируется к единице, таким образом, если мы собираемся воздействовать на характер заливки ровно в середине области, наш массив должен содержать значения (0, 0,5, 1). Второй параметр, pBlendFactors, также представляет собой указатель на массив вещественных чисел, с точно таким же количеством элементов, как и в массиве позиций. Но в нем хранятся данные о цвете в соответствующих контрольных точках заливки.

Диапазон цветов также нормирован к единице (стартовый цвет Color1 соответствует 0, конечный Color2 равен 1). Допустим, что в BlendFactors хранятся значения (0, 0,8, 1), это означает, что на участке заливки от 0 до 0,5 должно наблюдаться ускоренное возрастание градиента цвета (до 80% от Color2). На втором участке области заливки (от 0,5 до 1), наоборот, скорость смены цвета существенно замедляется, ведь до предельного значения, равного единице, останется добрать всего 20%. В третий параметр функции Count направляется количество элементов в массивах.

Массивы BlendFactors *и* BlendPositions *должны содержать одинаковое число элементов, значения которых должны последовательно возрастать от нуля к единице.*

Все сказанное выше может быть подтверждено следующим примером:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var  GPGraphics:TGPGraphics;
     GPLinearGradientBrush:TGPLinearGradientBrush;
const
   BlendFactors: array[0..3] of Single =(0.0, 0.4, 0.6, 1.0);
   BlendPositions: array[0..3] of Single =(0.0, 0.2, 0.8, 1.0);
begin
GPGraphics:=TGPGraphics.Create(DC);
GPLinearGradientBrush:=TGPLinearGradientBrush.Create(
                      MakePoint(0,0),MakePoint(400,0), aclGreen, aclRed);
{стандартная плавная заливка}
GPGraphics.FillRectangle(GPLinearGradientBrush, 0, 0, 400, 50);

{пользовательские настройки}
GPLinearGradientBrush.SetBlend(@BlendFactors, @BlendPositions, 4);
GPGraphics.FillRectangle(GPLinearGradientBrush, 0, 60, 400, 50);

GPGraphics.Free;
GPLinearGradientBrush.Free;
end;
```

Наши массивы BlendFactors и BlendPositions содержат по четыре элемента. Скорость заливки изменяется в двух точках – 0,2 и 0,8. До точки 0,2 скорость повышена (мы должны успеть изменить цвет на 40%), на участке от 0,2 до 0,8 замедляем изменение цвета (20%) и на участке от 0,8 до 1 вновь наверстываем упущенное (40%). Результаты работы процедуры представлены на рис. 29.10 (см. цветную вклейку, с. 421); для наглядности область заливки разделена на 10 участков от 0 до 1 с шагом 0,1.

При необходимости мы в состоянии провести анализ любой градиентной кисти на предмет неравномерной скорости изменения цвета ее заливки. Для этого необходима помощь двух методов. Первый из них информирует нас о числе участков смены скорости:

```
function GetBlendCount: Integer;
```

Если функция возвращает единицу, это указывает на то, что кисть осуществляет стандартную плавную заливку области. Если же единичное значение

превышено (что свидетельствует о наличии пользовательских настроек), то обращаемся ко второму методу:

```
function GetBlend(pBlendFactors, pBlendPositions: pSingle;
                  Count: Integer): TStatus;
```

Функция позволяет заполнить массивы позиций и цветов текущими значениями.

Возвращаемся к настройкам кисти по умолчанию. Мы уже привыкли, что назначаемый при создании кисти параметр Color2 всегда определяет цвет границы заливки. С таким утверждением готовы поспорить методы:

```
function SetBlendBellShape(Focus: Single; Scale: Single = 1.0): TStatus;
function SetBlendTriangularShape(Focus: Single; Scale: Single = 1.0): TStatus;
```

Они позволяют не только воздействовать на скорость изменения цвета заливки (ускоряя или замедляя его на определенных участках), но и изменить положение конечного цвета, для чего предназначен параметр Focus. Направляемое в параметр значение может колебаться в диапазоне от 0,0 до 1,0. Например, результат, представленный на рис. 29.11 (см. цв. вклейку, с. 421), получен посредством переноса фокуса в середину участка заливки – значение 0,5. Параметр Scale также нормирован к единице, он определяет степень достижения цвета Color2 в фокусной позиции. Установленное по умолчанию значение 1,0 требует, чтобы цвет полностью соответствовал значению Color2, любое меньшее значение можно рассматривать как долю от Color2. Разница между методами – в визуальном эффекте. Фокусная точка, полученная с помощью функции SetBlendTriangularShape, – более контрастная и напоминает смотрящий вверх угол треугольника. Напротив, градиентная заливка, полученная с помощью SetBlendBellShape, в области фокуса заметно мягче.

Нам осталось сломать еще один стереотип – о том, что градиентная заливка в кисти TGPLinearGradientBrush строится только на основе двух цветов. Это далеко не так, и подтверждением моих слов может служить метод:

```
function SetInterpolationColors(pPresetColors: PGPColor;
                                pBlendPositions: PSingle; Count: Integer): TStatus;
```

Без всякого преувеличения SetInterpolationColors() можно считать венцом возможностей класса TGPLinearGradientBrush. Функция позволяет создавать градиентную заливку кисти на основе нескольких цветов. Указатель на массив цветовых значений передается через параметр pPresetColors. Назначение параметра pBlendPositions аналогично одноименному параметру из состава рассмотренного несколько ранее метода SetBlend(). Это указатель на массив позиций. Заключительный параметр хранит число элементов в массивах. Порядок применения функции представлен в следующем примере:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPLinearGradientBrush:TGPLinearGradientBrush;
const
  PresetColors   : Array[0..2] of TGPColor= (aclRed,aclGreen,aclBlue);
  BlendPositions : Array[0..2] of Single=(0,0.5,1);
```

```
begin
GPGraphics:=TGPGraphics.Create(DC);
GPLinearGradientBrush:=TGPLinearGradientBrush.Create(
                        MakePoint(0, 0),MakePoint(400,0),aclWhite,aclBlack);

GPGraphics.FillRectangle(GPLinearGradientBrush, 0, 0, 400, 50);
GPLinearGradientBrush.SetInterpolationColors(@PresetColors,@BlendPositions,3);
GPGraphics.FillRectangle(GPLinearGradientBrush, 0, 60, 400, 50);

GPGraphics.Free;
GPLinearGradientBrush.Free;
end;
```

Для проведения исследования кисти на предмет поддержки многоцветной градиентной заливки нам потребуются услуги двух методов. Первый из них:

```
function GetInterpolationColorCount: Integer;
```

информирует программиста о числе цветов в кисти. Получив эти данные, обращаемся к функции:

```
function GetInterpolationColors(pPresetColors: PGPColor;
                            pBlendPositions: PSingle; Count: Integer): TStatus;
```

В метод необходимо передать указатели на массивы, в которые будут переданы данные о цветах и их позициях.

Градиентная кисть сложной формы TGPPathGradientBrush

Последняя из пяти входящих в состав GDI+ кистей специализируется на градиентной заливке областей неординарной формы и траекторий. С точки зрения потенциальных возможностей (и, как следствие, из-за сложности реализации) указанная кисть не имеет себе равных ни в GDI+, ни тем более в обычном GDI. Изучение класса `TGPPathGradientBrush` традиционно начнем с рассмотрения способов создания его экземпляра. Для этого в распоряжение программиста предоставлено три конструктора. Два из них формируют кисть на основе массива точек:

```
constructor Create(pPoints: PGPPoint; Count: Integer;
      WrapMode: TWrapMode = WrapModeClamp); reintroduce; overload;
constructor Create(pPoints: PGPPointF; Count: Integer;
      WrapMode: TWrapMode = WrapModeClamp); reintroduce; overload;
```

Разница между методами заключается лишь в варианте типизации массивов (`TGPPoint` или `TGPPointF`), а в остальном они идентичны. Кроме указателя на массив необходимо передать в конструктор число входящих в массив точек `Count` и, при необходимости, порядок заполнения `WrapMode`. Ниже предложен пример построения кисти на основе пятиугольника.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPPathGradientBrush:TGPPathGradientBrush;
```

```
const Points : Array[0..4] OF TGPPoint=
    ((x:150;y:0),(x:250;y:60),(x:210;y:200),(x:90;y:200),(x:50;y:60));
begin
 GPGraphics:=TGPGraphics.Create(DC);
 GPPathGradientBrush:=TGPPathGradientBrush.Create(PGPPoint(@Points),
                                                  5,WrapModeClamp);
 GPGraphics.FillRectangle( GPPathGradientBrush, 50, 0, 200, 200);
 GPGraphics.Free;
 GPPathGradientBrush.Free;
end;
```

Обратите внимание, что конструктор не содержит никаких упоминаний о «боевой» раскраске кисти, посему каждый вновь создаваемый экземпляр кисти TGPPathGradientBrush черно-белый. Результат работы кода представлен на рис. 29.12.

Вместо описывающего контуры фигуры массива точек при создании объекта кисти допускается применять траекторию. В этом случае синтаксис конструктора изменяется:

```
constructor Create(Path: TGPGraphicsPath); reintroduce; overload;
```

Если перед нами стоит задача исследовать траекторию уже созданной кисти, то эта операция должна начаться с обращения к методу:

```
function GetPointCount: Integer;
```

информирующему программиста о числе точек в описании сложной фигуры.

Как правило, на вызове конструктора формирование градиентной кисти для заливки нестандартных областей не заканчивается. Кроме описания формы кисти целесообразно определиться с тремя ее атрибутами:

1. Цветом границы.
2. Координатами центра кисти.
3. Цветом центра.

Цвет (или цвета) границы назначаются с помощью метода:

```
function SetSurroundColors(pColors: pARGB; var Count: Integer): TStatus;
```

Рис. 29.12. Градиентная кисть сложной формы

В параметр pColors передается указатель на массив цветов, количество элементов в массиве определяется в параметре Count.

Определение нескольких цветов границы требует некоторой сноровки. Хорошего визуального эффекта можно добиться, следуя следующему правилу – число цветов должно быть равным числу вершин внешней границы кисти, например 3 цвета для треугольника. Не стоит ожидать от GDI+ шедевров при попытке раскрасить во все цвета радуги траекторию неправильной формы либо траекторию, построенную на основе эллипсов, дуг и кривых Безье.

Обратная задача, связанная с выяснением цветов границы, решается последовательным вызовом двух методов. Сначала необходимо узнать общее количество задействованных цветов:

```
function GetSurroundColorCount: Integer;
```

Затем, после распределения памяти под массив Colors : **Array of** TGPColor, призванный хранить перечень цветов, на основе которых строится градиентная заливка, обращаемся ко второму методу:

```
function GetSurroundColors(pColors: pARGB; var Count: Integer): TStatus;
```

Функция, получив указатель на массив цветов границы и число элементов в этом массиве, заполнит его значениями 32-битного цвета.

Координаты центра кисти могут задаваться целыми или вещественными числами, в зависимости от этого и выбирается соответствующий перегружаемый метод:

```
function SetCenterPoint(Point: TGPPointF): TStatus; overload;
function SetCenterPoint(Point: TGPPoint): TStatus; overload;
```

Стоит отметить, что интеллекта кисти вполне достаточно для самостоятельного вычисления координат своего центра, поэтому услуги SetCenterPoint() понадобятся только в ситуации, когда мы собираемся сдвинуть центральную точку куда-нибудь в сторонку.

Если же наша миссия исследовательская, то информацию о местонахождении центральной точки нам предоставят методы:

```
function GetCenterPoint(out Point: TGPPointF): TStatus; overload;
function GetCenterPoint(out Point: TGPPoint): TStatus; overload;
```

Цвет центра сложной фигуры зависит от того, какое значение было передано в метод:

```
function SetCenterColor(Color: TGPColor): TStatus;
```

Синтаксис обратной функции – следующий:

```
function GetCenterColor(out Color: TGPColor): TStatus;
```

После всего вышесказанного очередной пример не должен вызвать каких-либо вопросов.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPGraphicsPath:TGPGraphicsPath;
    GPPathGradientBrush:TGPPathGradientBrush;
    count:integer;
const
  Points: array[0..2] of TGPPointF=((X:5;Y:5), (X:305;Y:5), (X:152.5;Y:255));
  Colors: array[0..2] of TGPColor =(aclRed,aclGreen,aclBlue);
begin
    GPGraphics:=TGPGraphics.Create(DC);

    GPPathGradientBrush:=TGPPathGradientBrush.Create(PGPPointF(@Points),3);
    //1- цвета границы
    count:=3;
    GPPathGradientBrush.SetSurroundColors(@colors, count);
    //2- координаты центра
    GPPathGradientBrush.SetCenterPoint(MakePoint(150,120));
    //3 - цвет центра
    GPPathGradientBrush.SetCenterColor(aclYellow);

    GPGraphics.FillRectangle(GPPathGradientBrush,0,0,400,400);
    GPGraphics.Free;
    GPPathGradientBrush.Free;
end;
```

Нами создана градиентная кисть с многоцветной заливкой в виде треугольника, центр фигуры окрашен в желтый цвет, а вершины – в красный, зеленый и синий цвета. Внешний вид кисти вы можете увидеть на рис. 29.13 (см. цветную вклейку, с. 422).

Форма кисти – отнюдь не догма. Ее разрешено не только исследовать, но и менять «на лету». В этом нам помогут методы:

```
function SetGraphicsPath(Path: TGPGraphicsPath): TStatus;
function GetGraphicsPath(Path: TGPGraphicsPath): TStatus;
```

Кроме того, градиентная кисть `TGPPathGradientBrush` способна:

- изменять скорость смены цветов в градиенте (`SetBlend`);
- осуществлять гамма-коррекцию (`SetGammaCorrection`);
- интерполировать цвета (`SetInterpolationColors`);
- управлять формой фигуры заливки (`SetBlendBellShape` и `SetBlendTriangularShape`).

Эти и ряд других методов уже были рассмотрены при изучении кисти с линейной градиентной заливкой `TGPLinearGradientBrush`.

Одна из отличительных особенностей класса `TGPPathGradientBrush` – возможность масштабирования области фокусной точки заливки:

```
function SetFocusScales(xScale, yScale: Single): TStatus;
```

Управляя коэффициентами `xScale` и `yScale`, мы сможем уменьшать или увеличивать протяженность области, закрашенной таким же цветом, как и у цент-

ральной точки траектории. Как всегда в подобных случаях, у метода `SetFocusScales()` имеется пара:

```
function GetFocusScales(out xScale, yScale: Single): TStatus;
```

Эта функция позволит узнать, какие коэффициенты масштабирования в данный момент использует градиентная кисть.

РЕЗЮМЕ

Кисти GDI+ превосходят своих коллег из классического GDI практически по всем характеристикам. Они вооружены богатым набором предопределенных узоров, поддерживают аффинные преобразования, готовы подвергнуться цветовой коррекции, обладают возможностью градиентной заливки и, что очень важно, достаточно просты в эксплуатации. У них есть всего лишь один недостаток – в сравнении с кистями GDI они более ресурсоемки.

30

Перья GDI+

Современная графическая библиотека не просто сохранила, но и существенно преумножила славные традиции графической подсистемы GDI прежних версий Windows. Преумножила до такой степени, что иногда невольно закрадывается мысль, что при постановке технического задания на разработку GDI+ от программистов потребовали, чтобы вновь создаваемые графические объекты и «объектики» не просто обладали всеми способностями их ближайших родственников из классического GDI, но и могли сделать еще что-нибудь этакое, чтобы у всех захватило дух... Такому предположению во многом соответствует и один из основных объектов расширенной версии GDI – перо.

В первых главах книги упоминалось, что с точки зрения модели объектов графической системы Windows перо GDI – это не что иное, как тонкая кисть. Следуя логике вещей, можно было бы предположить, что и в библиотеке GDI+ обновленное перо (класс `TGPPen`) также станет наследником одного из класса кистей. Но не тут-то было. В иерархии классов GDI+ новое перо, как и подавляющее большинство других небожителей из GDI+, является прямым потомком класса `TGdiplusBase` и среди своих предков не имеет даже отдаленного намека на кисти GDI+. Вот такие, на первый взгляд, чудеса... Но описанное решение ни в коем случае не ошибочно. При проектировании новых объектов GDI+ разработчики из стен Microsoft были вынуждены постоянно оглядываться на графическое ядро Windows, в котором господствует GDI. Поэтому перо GDI+ было просто обязано сохранить преемственность и восприниматься системой как старое доброе перо GDI. И если бы программисты отклонились в сторону хоть на шаг и разорвали связь между перьями старой и новой технологии (например, реализовали перо просто на базе сплошной кисти `TGPSolidBrush`), то современное перо перестали бы понимать все опорные функции GDI. К счастью, это не означает, что новое перо GDI+ не в состоянии воспользоваться возможностями, заложенными в обновленные версии кистей, в этом вы убедитесь, познакомившись с конструкторами класса `TGPPen`.

Создание пера

Возложенная на перо GDI+ функциональная нагрузка практически идентична задачам обычного пера – это черчение линий и обводка контуров фигур. Экземпляр пера может быть создан за счет любого из двух предусмотренных в классе TGPPen конструкторов:

```
constructor Create(Color: TGPColor; Width: Single = 1.0); reintroduce; overload;
constructor Create(Brush: TGPBrush; Width: Single = 1.0); reintroduce; overload;
```

На первый взгляд, ничего нового. Первый конструктор требует от программиста определиться с цветом будущего пера Color и назначить его толщину Width. Второй конструктор отдаленно напоминает функцию ExtCreatePen(), предназначенную для создания геометрического пера GDI, и здесь, и там внешний вид пера задается кистью Brush. За видимостью простоты на самом деле скрывается весьма сложный механизм, ведь в GDI+ появилось целых пять классов кистей (потомков TGPBrush) с возможностями, которые 20 лет назад и не снились разработчикам старого GDI (см. главу 29 «Кисти GDI+»). Каково было в те времена создать перо на основе кисти с линейной градиентной заливкой и нетривиальной фигурой? Правда, нельзя сказать, что GDI+ не оставил места для острот скептиков, которые сразу зададут вопрос о том, как часто в наших проектах требуется столь фантастическое перо? Но это уже возвращение к первому абзацу главы, поэтому не стану «окрылять» читателя примером процедуры, способной в два счета поглотить вычислительный ресурс простенького компьютера при проведении линии пером, выточенным из кисти TGPPathGradientBrush. Взамен предложу более приземленный (читай полезный) код, демонстрирующий порядок применения обеих версий конструкторов.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics; GPPen:TGPPen; Brush:TGPHatchBrush;
begin
    GPGraphics:=TGPGraphics.Create(DC);
{Обычное перо}
    GPPen:=TGPPen.Create(aclBlack,20);
    GPGraphics.DrawLine(GPPen,10,20,210,20);
    GPPen.Free;
{Перо на основе кисти}
    Brush:=TGPHatchBrush.Create(HatchStyleWave,aclBlack,aclWhite);
    GPPen:=TGPPen.Create(Brush,20);
    GPGraphics.DrawLine(GPPen,10,60,210,60);

    GPGraphics.Free; GPPen.Free; Brush.Free;
end;
```

По умолчанию вновь созданному экземпляру пера назначается толщина, соответствующая одной логической единице. Это совсем неплохо, когда за текущую единицу измерения принят физический пиксел. Однако в качестве единицы измерения может быть установлен сантиметр, дюйм или какая-то другая величина. Представьте себе, как будет смотреться на экране компью-

тера линия толщиной в один метр! Поэтому если логика программы предполагает, что независимо от принятой в холсте GDI+ единицы измерения толщина пера должна оставаться минимальной, то при смене единицы измерения необходимо производить пересчет:

```
GPGraphics.SetPageUnit(UnitInch); //единица измерения - дюйм
Pen:=TGPPen.Create(aclBlack,1/GPGraphics.GetDpiY); //толщина линии - пиксел
```

С помощью метода `GetDpiY()` мы уточняем вертикальное разрешение – число пикселов в одном дюйме по вертикали.

В GDI в основу системы классификации перьев было положено понятие косметического и геометрического пера. В GDI+ предполагается несколько иной подход, теперь перья подразделяются по признаку – какой класс кисти использовался для его создания. Отправной точкой классификации выступает тип данных `TPenType`:

```
type TPenType = (
   PenTypeSolidColor     =  ord(BrushTypeSolidColor),
   PenTypeHatchFill      =  ord(BrushTypeHatchFill),
   PenTypeTextureFill    =  ord(BrushTypeTextureFill),
   PenTypePathGradient   =  ord(BrushTypePathGradient),
   PenTypeLinearGradient =  ord(BrushTypeLinearGradient),
   PenTypeUnknown        = -1
  );
```

За информацией о типе пера следует обращаться к методу:

```
function GetPenType: TPenType;
```

Мы уже привыкли, что в GDI+ у каждой функции с названием, начинающимся с приставки «Get», имеется функция-партнер «Set». В данном случае мы сталкиваемся с одним из немногих исключений – у класса `TGPPen` метода с названием «SetPenStyle» не предусмотрено. Для изменения стиля пера необходимо поменять кисть, на основе которой он был создан. Такая операция должна проводиться еще на стадии вызова конструктора пера либо при посредничестве метода:

```
function SetBrush(Brush: TGPBrush): TStatus;
```

В данном случае с именованием методов все в порядке, у пера имеется парная информационная функция:

```
function GetBrush: TGPBrush;
```

Цвет и толщина пера

Наиболее простыми и, пожалуй, наиболее востребованными характеристиками пера считаются толщина проводимой пером линии и ее цвет. Обслуживание указанных атрибутов производится двумя группами функций. Толщина линии определяется методами:

```
function SetWidth(Width: Single): TStatus; //установка толщины
function GetWidth: Single; //получение данных о толщине
```

Цвет линии настраивается методами:

```
function SetColor(Color: TGPColor): TStatus; //установка цвета
function GetColor(out Color: TGPColor): TStatus; //получение данных о цвете
```

Стиль пера

Управляя стилем штриховки пера, программист сможет существенно разнообразить внешний вид рисуемых линий. Так что, если вам набило оскомину банальное сплошное перо, то смело изменяйте его узор на уникальный. Доступ к стилю пера осуществляется при посредничестве метода:

```
function SetDashStyle(DashStyle: TDashStyle): TStatus;
type TDashStyle = (DashStyleSolid,      //0 - сплошная линия
                   DashStyleDash,       //1 - линия из тире
                   DashStyleDot,        //2 - линия из точек
                   DashStyleDashDot,    //3 - чередование тире-точка
                   DashStyleDashDotDot, //4 - чередование тире-точка-точка
                   DashStyleCustom);    //5 - пользовательский стиль
```

Для выяснения, какой именно стиль установлен в данный момент, обращаемся к методу:

```
function GetDashStyle: TDashStyle;
```

Тип данных `TDashStyle` предполагает существование точно таких же стилей, как и у пера GDI (см. табл. 5.1), исключая невидимое перо. Во всем перечислении наиболее интересен пользовательский стиль, благодаря которому мы сможем описать уникальную последовательность точек, тире и пробелов, «увековечив» свой собственный эксклюзивный стиль линий. Конструирование пользовательского стиля рисунка пера представляет собой трехходовую комбинацию, в которой: во-первых, следует определиться с длинами штрихов и пробелов, во-вторых, поместить данные об узоре пера в массив вещественных чисел и, в-третьих, передать указатель на массив с узором в метод:

```
function SetDashPattern(pDashArray: pSingle; Count: Integer): TStatus;
```

Второй параметр функции определяет число элементов в массиве. Порядок работы с методом предложен в следующих строках кода:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPPen:TGPPen;
const
 DashArray: array[0..3] of Single =(5,3,15,3); //пользовательский узор
begin
 GPGraphics:=TGPGraphics.Create(DC);

 GPPen:=TGPPen.Create(aclBlack);
 GPPen.SetDashPattern(pSingle(@DashArray), 4);
 GPGraphics.DrawLine(GPPen,10,10,310,10);

 GPGraphics.Free;
```

```
  GPPen.Free;
end;
```

В результате нами создано перо, способное провести линию, состоящую из последовательно чередующихся отрезков длиной 5 и 15 логических единиц, разделенных пробелами длиною в 3 логические единицы.

Перо TGPPen *свободно от недостатка обычного пера GDI, не умеющего рисовать утолщенные прерывистые линии. В GDI+ стиль пера поддерживается для линий любой толщины.*

Если преследуется цель по исследованию стиля пера, то придется последовательно вызвать два метода:

```
function GetDashPatternCount: Integer;
function GetDashPattern(pDashArray: pSingle; Count: Integer): TStatus;
```

Метод GetDashPatternCount() возвратит число элементов в стиле линии, оно понадобится для распределения памяти в массиве. Второй метод заполнит массив данными об узоре.

```
var
  GPGraphics:TGPGraphics;
  GPPen:TGPPen;
  DashArray : array of Single;
  Count:Integer;
begin
    GPGraphics:=TGPGraphics.Create(DC);
    GPPen:=TGPPen.Create(aclBlack);
    GPPen.SetDashStyle(DashStyleDashDotDot);//стиль тире-точка-точка

    Count:=GPPen.GetDashPatternCount; //число элементов

    GetMem(DashArray,Count*SizeOf(Single));//распределяем память
    GPPen.GetDashPattern(pSingle(DashArray),Count); //получаем данные

    //================ВЫВОДИМ НА ЭКРАН================
     For Y:=0 to Count-1 do
       begin
         S:=Format('DashArray[%d]=%.2f',[y,DashArray[y]]);
         TextOut(DC,10,1+Y*14,pChar(S),Length(S));
       end;
     GPGraphics.DrawLine(GPPen,10,Y*14+20,290,Y*14+20);
    //================================================
    GPGraphics.Free;
    GPPen.Free;
    FreeMem(DashArray);
end;
```

На рис. 30.1 продемонстрированы результаты исследования пера со стандартным стилем DashStyleDashDotDot, в итоге наш массив инициализировался значениями [3,1,1,1,1,1].

Среди констант, определяющих стиль пера в GDI, имеется константа PS_INSIDEFRAME. Она предназначена для создания перьев, применяемых при рисо-

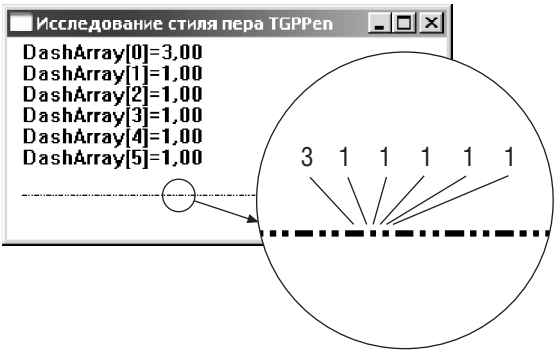


Рис. 30.1. Анализ пера стиля DashStyleDashDotDot

вании геометрических фигур, основанных на прямоугольной области (прямоугольников, эллипсов и т. п.), утолщенным пером. Особенность указанного стиля в том, что при выводе всех графических примитивов, вписанных в прямоугольник, линия не центруется относительно главной оси, вследствие этого выступая за границы примитива, а смещается вовнутрь фигуры. Для того чтобы получить аналогичный результат для линии, проведенной пером GDI+, следует воспользоваться методом:

```
function SetAlignment(PenAlignment: TPenAlignment): TStatus;
```

Параметр `PenAlignment` может принимать одно из двух значений:

```
type TPenAlignment = (PenAlignmentCenter,PenAlignmentInset);
```

Состоянию по умолчанию (когда утолщенная линия центруется относительно главной оси) соответствует режим `PenAlignmentCenter`, смещение линии вовнутрь фигуры обеспечивается режимом `PenAlignmentInset`.

Для выяснения текущего режима выравнивания линии стоит обратиться к методу:

```
function GetAlignment: TPenAlignment;
```

Смещение начальной точки узора

В распоряжении программиста имеется возможность передвигать стартовую точку стиля пера. Например, можно добиться такого эффекта, что узор линии, нарисованной пером `DashStyleDashDot` (тире-точка-точка), станет начинаться не с тире, а с точки. Для этих целей предназначен метод:

```
function SetDashOffset(DashOffset: Single): TStatus;
```

Значение отступа передается в единственный параметр `DashOffset`. Увеличение значения `DashOffset` приводит к сдвигу узора влево (вверх), уменьшение – вправо (вниз). При обращении к только что созданному объекту пера информационная функция:

```
function GetDashOffset: Single;
```

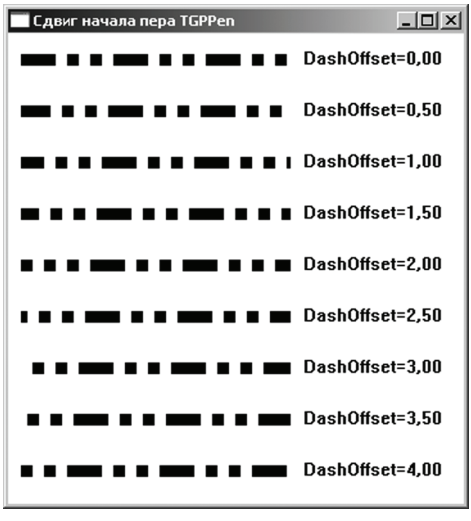


Рис. 30.2. Сдвиг рисунка линии

всегда возвратит нулевое значение. Это означает, что стартовая точка стиля пера не смещена. На рис. 30.2 представлены результаты эксперимента по управлению отправной точкой узора пера со стилем тире-точка-точка. Перед каждым обращением к методу `DrawLine()` отступ получал положительное приращение на 0,5.

Внешний вид разрывов

Спешу рассказать еще одну весьма интересную новость. По сравнению с пером GDI у его обновленного коллеги появилась возможность улучшить начертания разрывов в линии. По умолчанию у вновь созданного пера `TGPPen` они точно такие же, как и у пера GDI, – линия в местах разрыва как будто отрезана ножом. Но это можно поправить, для изменения внешнего вида разрыва необходимо вызвать метод:

```
function SetDashCap(DashCap: TDashCap): TStatus;
type TDashCap = (DashCapFlat  = 0,    //плоский разрыв
                 DashCapRound = 2,    //скругленный разрыв
                 DashCapTriangle= 3); //треугольный разрыв
```

Для того чтобы уточнить текущие настройки разрывов, обратитесь к методу:

```
function GetDashCap: TDashCap;
```

Значению по умолчанию соответствует константа `DashCapFlat`, в этом случае линия в месте разрыва просто отсекается. На рис. 30.3 представлены имеющиеся в нашем распоряжении комбинации стилей и разрывов пера.

Закругленные и треугольные окончания линий выходят за пределы геометрического окончания линии с прямоугольным завершением.

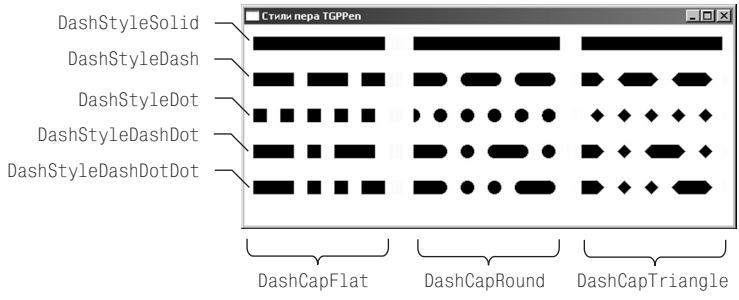


Рис. 30.3. Стили пера TDashStyle и разрывы пера TDashCap

Наконечники пера

По большому счету возможность оснащения пера разнообразными наконечниками – далеко не откровение. В GDI уже имелся такой сервис, но с весьма скудным набором вариантов графических решений, таковых имелось всего три (см. рис. 5.2). В GDI+, кроме появления шести дополнительных наконечников, предусмотрена весьма полезная возможность создания своих собственных версий завершения линий. Для этого придуман специальный вспомогательный класс `TGPCustomLineCap`.

Для определения особенностей начертания начала и окончания линии в классе `TGPPen` имеется две пары функций. Внешний вид стартового отрезка настраивается методами:

```
function SetStartCap(SrtCap: TLineCap): TStatus;
function GetStartCap: TLineCap;
```

Метод `Set...` устанавливает вид наконечника, метод `Get...` информирует программиста, какой наконечник в данный момент используется пером. Окончание линии также может быть изменено, но в этот раз потребуются другие методы:

```
function SetEndCap(EdCap: TLineCap): TStatus;
function GetEndCap: TLineCap;
```

Особенности начертания наконечника определяются типом данных `TLineCap` (табл. 30.1 и рис. 30.4).

Таблица 30.1. Варианты наконечника TLineCap

Константа	Значение	Описание наконечника
`LineCapFlat`	`$00`	Плоский.
`LineCapSquare`	`$01`	Квадратный.
`LineCapRound`	`$02`	Скругленный.
`LineCapTriangle`	`$03`	Треугольный.

Константа	Значение	Описание наконечника
LineCapNoAnchor	$10	Без наконечника.
LineCapSquareAnchor	$11	Квадратный наконечник.
LineCapRoundAnchor	$12	Наконечник в виде окружности.
LineCapDiamondAnchor	$13	Наконечник в виде ромба.
LineCapArrowAnchor	$14	Наконечник-стрелка.
LineCapCustom	$FF	Наконечник, созданный программистом самостоятельно на базе класса TGPCustomLineCap.

Для лиц, предпочитающих получать требуемый результат как можно скорее, предусмотрен метод, позволяющий одновременно сменить оба наконечника пера:

```
function SetLineCap(SartCap, EndCap: TLineCap; DashCap: TDashCap): TStatus;
```

Соответствующие значения передаются в параметры SartCap и EndCap. В качестве бонуса предлагается третий параметр DashCap, он поможет настроить особенности начертания разрывов.

Если для реализации графического интерфейса приложения стандартных наконечников пера оказалось недостаточно, то разрешается создать свои собственные версии рисунков. Для этого предназначен класс TGPCustomLineCap. Создав экземпляр класса и заполнив требуемые атрибуты, присоединяем наконечник к перу. На этих операциях специализируются методы:

```
function SetCustomStartCap(CustomCap: TGPCustomLineCap): TStatus;
function SetCustomEndCap(CustomCap: TGPCustomLineCap): TStatus;
```

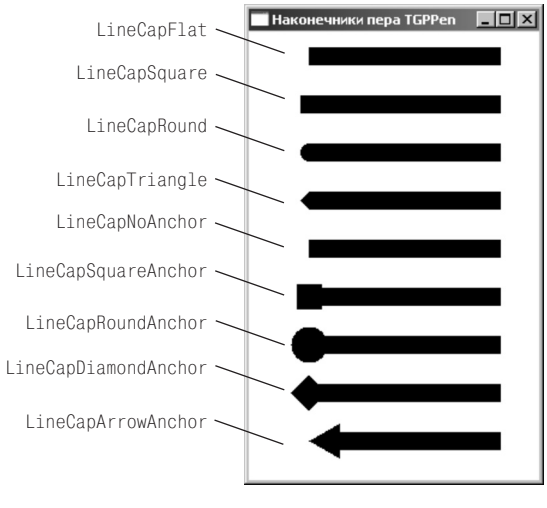


Рис. 30.4. Варианты стандартных наконечников пера

Первый метод прикрепляет наконечник «ручной работы» к стартовому отрезку линии, второй – к финишному. Пара симметричных методов проинформирует нас о текущих наконечниках пера:

```
function GetCustomStartCap(CustomCap: TGPCustomLineCap): TStatus;
function GetCustomEndCap(CustomCap: TGPCustomLineCap): TStatus;
```

Пользовательские наконечники, класс TGPCustomLineCap

Пользовательский наконечник – это самостоятельный объект, построенный на основе класса TGPCustomLineCap. В качестве претендента на трон может выступать только траектория GDI+ (экземпляр класса TGPGraphicsPath), объект траектории направляется в конструктор наконечника:

```
constructor Create(FillPath, StrokePath: TGPGraphicsPath;
                   BaseCap: TLineCap = LineCapFlat;
                   BaseInset: Single = 0); reintroduce; overload;
```

Метод предполагает, что изображение наконечника формируется на базе одной из двух траекторий: FillPath или StrokePath. Разница в том, что при выводе на графическое устройство наконечник, созданный на базе траектории FillPath, заливается кистью, наконечник StrokePath – обводится пером. Еще раз повторюсь, что оба параметра заполнять не обязательно, можно ограничиться только одной из траекторий, направив во второй параметр неопределенный указатель. Параметр BaseCap вносит некоторую сумятицу в процесс формирования пера. Согласно SDK он определяет, какой из стандартных наконечников будет использован в момент создания пера TGPPen. На практике в Delphi экземпляр TGPCustomLineCap отказывается реагировать на любое передаваемое в параметр значение, поэтому оставим BaseCap в покое вместе со значением по умолчанию LineCapFlat (концы линии обрезаны). Последний параметр определяет расстояние отступа базовой линии от наконечника, как правило, это значение устанавливается в ноль, тем более, что позднее отступ можно настроить за счет метода:

```
function SetBaseInset(Inset: Single): TStatus;
```

Парный метод:

```
function GetBaseInset: Single;
```

возвратит информацию о расстоянии между наконечником и базовой линией.

При выводе на графическом устройстве изображения наконечника используются перо (контур наконечника) и кисть (заливка наконечника).

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPPen:TGPPen;
    StrokePath: TGPGraphicsPath;
    GPCustomLineCap:TGPCustomLineCap;
const
  Points : Array[0..4] of TGPPoint=
```

```
            ((X:0;Y:0),(X:-5;Y:-5),(X:5;Y:5),(X:-5;Y:5),(X:5;Y:-5));
  Types  : Array[0..4] of TPathPointType =(0,1, 1, 0, $81);
begin
    GPGraphics:=TGPGraphics.Create(DC);
    GPPen:=TGPPen.Create(aclGreen,4);

    StrokePath:=TGPGraphicsPath.Create(pGPPoint(@Points),pByte(@Types),5);

    GPCustomLineCap:=TGPCustomLineCap.Create(nil,StrokePath,LineCapFlat,0);
    GPPen.SetCustomStartCap(GPCustomLineCap);
//… операции с пером
    GPGraphics.Free;   GPPen.Free;
end;
```

При описании траектории наконечника следует иметь в виду, что центром наконечника станет точка с логическими координатами (0,0).

Из оставшихся в составе класса TGPCustomLineCap возможностей стоит уделить внимание способности увеличить либо уменьшить размер наконечника. Эта операция осуществляется за счет передаваемого в параметр WidthScale коэффициента масштабирования:

```
function SetWidthScale(WidthScale: Single): TStatus;
```

По умолчанию у вновь создаваемого экземпляра TGPCustomLineCap множитель равен 1, это может подтвердить метод:

```
function GetWidthScale: Single;
```

возвращающий коэффициент масштабирования наконечника пера.

Стык линий

Управление способами прорисовки мест соединения сходящихся под острыми углами линий GDI+ практически полностью соответствует порядку работы со стыками у перьев GDI. Формат стыка определяется типом данных TLineJoin, перечень возможных значений которого весьма схож с константами (см. табл. 5.3), направляемыми в параметр PenStyle функции Windows GDI ExtCreatePen().

```
type TLineJoin = (LineJoinMiter,        //заостренное соединение
                  LineJoinBevel,        //соединение с отсечением
                  LineJoinRound,        //закругленное соединение
                  LineJoinMiterClipped);//заостренное соединение с отсечением
```

Одна из указанных констант передается в единственный параметр метода:

```
function SetLineJoin(LineJoin: TLineJoin): TStatus;
```

По сравнению с GDI в перечне TLineJoin появилось всего одно новое значение – константа LineJoinMiterClipped. Стык линий, нарисованных пером с соединением LineJoinMiterClipped, напоминает соединение LineJoinMiter с той лишь разницей, что теперь при превышении углового предела (отношения длины заострения к толщине линии) излишки линии отсекаются.

Как и в GDI, в GDI+ внешний вид соединения определялся не только константой TLineJoin, но и атрибутом протяженности, устанавливаемым с помощью метода:

```
function SetMiterLimit(MiterLimit: Single): TStatus;
```

По умолчанию предел MiterLimit равен 10, уменьшая это значение, мы снижаем протяженность стыка. Например, в следующем фрагменте кода мы устанавливаем предел протяженности равным 4, что приводит к усечению стыка линий (рис. 30.5).

Представленный на рисунке результат получен за счет выполнения следующих строк кода:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPPen:TGPPen;
const
  Points:Array[0..2] of TGPPoint=((X:20;Y:20),(X:320;Y:80),(X:20;Y:140));
begin
    GPGraphics:=TGPGraphics.Create(DC);
    GPPen:=TGPPen.Create(aclLightBlue,20);
    GPPen.SetLineJoin(LineJoinMiterClipped);
    GPPen.SetMiterLimit(4);

    GPGraphics.DrawPolygon(GPPen,pGPPoint(@Points),3);

    GPGraphics.Free;
    GPPen.Free;
end;
```

Для исследования стыков предназначены два метода пера:

```
function GetLineJoin: TLineJoin; //тип стыка
function GetMiterLimit: Single;  //предел стыка
```

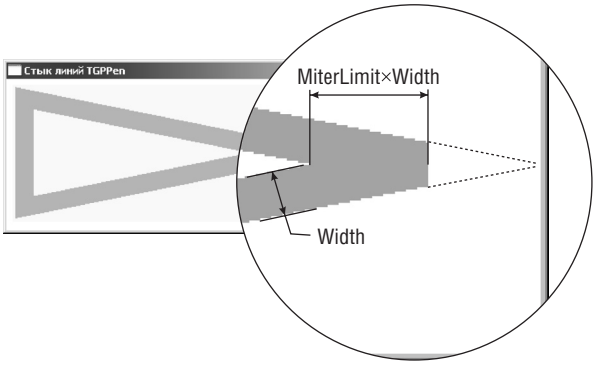


Рис. 30.5. Взаимосвязь углового предела и протяженности стыка

Расслоение пера

Теперь поговорим о ноу-хау пера GDI+, прямо скажем, на такие чудеса старое перо GDI не способно. При рисовании утолщенных линий может пригодиться эффект расслоения пера. Благодаря этому мы получаем чередующийся набор закрашенных и пустых областей, направленных вдоль проведенной пером линии, что создает впечатление, будто бы линия состоит из нескольких параллельно проходящих волокон. Особенности начертания линии назначаются методом:

```
function SetCompoundArray(pCompoundArray: pSingle; Count: Integer): TStatus;
```

В параметр pCompoundArray направляется указатель, содержащий набор элементов, определяющих порядок расслоения. Правила заполнения массива просты. Толщина пера нормируется к единице. Значения элементов определяют расстояние от границы линии до точки расслоения пера. Порядок работы с методом продемонстрирован в следующем примере:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPPen:TGPPen;
const
CompoundArray: Array[0..5] of Single =(0, 0.1, 0.4, 0.8, 0.9, 1);
begin
    GPGraphics:=TGPGraphics.Create(DC);
    GPPen:=TGPPen.Create(aclGreen,90);
    GPPen.SetCompoundArray(pSingle(@CompoundArray), 6);

    GPGraphics.DrawLine(GPPen,15,50,445,180);

    GPGraphics.Free;
    GPPen.Free;
end;
```

На рис. 30.6 представлены результаты работы процедуры.

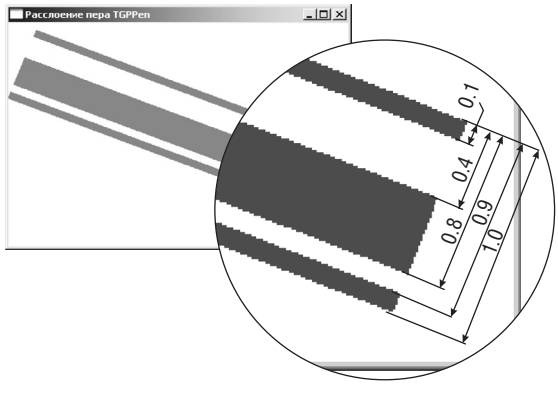


Рис. 30.6. Демонстрация расслоения пера для массива значений (0, 0.1, 0.4, 0.8, 0.9, 1)

Исследование пера на предмет расслоения проводится с помощью последовательного обращения к двум методам. Сначала требуется узнать число элементов в массиве:

```
function GetCompoundArrayCount: Integer;
```

Затем, распределив память под хранение данных, следует обратиться к методу:

```
function GetCompoundArray(pCompoundArray: pSingle; Count: Integer): TStatus;
```

заполняющему массив, на который ссылается указатель `pCompoundArray`.

В заключение разговора о перьях обязательно отметим, что перо GDI+ может быть полноправным участником аффинных преобразований. Для этого объект `TGPPen` оснащен рядом специальных методов: `SetTransform()`, `GetTransform()`, `ResetTransform()`, `MultiplyTransform()`, `RotateTransform()` и `ScaleTransform()`. Перечисленные методы нам уже хорошо знакомы по табл. 29.1 из предыдущей главы. Кроме того, процессу мировых преобразований в GDI+ посвящена большая часть главы 34 этой книги. Поэтому мы не станем ни повторяться, ни забегать вперед и перейдем к выводам по представленному материалу.

РЕЗЮМЕ

В GDI+ перо реализовано в виде единственного класса `TGPPen`. Экземпляр класса обладает всем спектром возможностей классического пера GDI, плюс рядом существенных усовершенствований. К таковым в первую очередь относятся поддержка двумерных аффинных преобразований, расширенный набор наконечников, улучшенная прорисовка прерывистых линий, возможность расслоения пера на волокна и ряд других визуальных эффектов.

31

Траектории GDI+

Очень часто при решении графических задач возникает необходимость создания фигур сложной формы. В классическом GDI для этого применяется коллекция линий, кривых и других фигур, объединенных в единый объект, называемый траекторией[1]. Разработчики Windows XP не оставили эту идею без должного внимания и также взяли траекторию на вооружение и реализовали ее (как всегда, с присущим GDI+ размахом) в рамках класса `TGPGraphicsPath`.

У обновленной траектории имеется одно неоспоримое преимущество над траекторией, появившейся на свет за счет средств обычного GDI. Прежде чем я о нем расскажу, кратко напомню, как организована работа с траекторией в GDI. Входящие в состав траектории GDI примитивы описывались в рамках обращения к функциям `BeginPath()` и `EndPath()` и сохранялись не в виде самостоятельного объекта (как перо, кисть или регион), а записывались в соответствующий атрибут контекста. Если программисту требовалось создать новую траекторию, то ему приходилось распрощаться со старой (ведь у контекста всего один атрибут, специализирующийся на хранении траектории, для второй в нем нет места). Конечно, если старая траектория была столь дорога разработчику ПО, что у него не хватало сил с ней расстаться, то в GDI допускалось конвертировать траекторию в регион или набор кривых Безье, но это весьма утомительное занятие... Разработчики GDI+ не забыли печальных уроков прошлого и сделали единственно верный вывод в этом случае – траекторию следует хранить отдельно от контекста. Для этого они и приспособили специальный объект – экземпляр `TGPGraphicsPath`. Отныне не возбраняется создавать столько траекторий, сколько требуется, ограничение только одно – доступные объемы виртуального адресного пространства нашего компьютера.

Во всем остальном траектория GDI+ весьма схожа с ее предшественником из GDI. Это некоторая совокупность линий и кривых, заданных в логических ко-

---

[1] В ряде изданий вместо термина «траектория» применяется термин «путь» или «контур».

ординатах. Входящие в траекторию примитивы могут объединяться в составные фигуры, таким образом траектория может содержать несколько фигур.

На отображении траекторий специализируются всего два метода холста GDI+:

```
function DrawPath(Pen: TGPPen; Path: TGPGraphicsPath): TStatus;
function FillPath(Brush: TGPBrush; Path: TGPGraphicsPath): TStatus;
```

Первый метод отвечает за черчение контура траектории `Path`. Для этого функции требуется определиться с пером `Pen`. Заливка замкнутой траектории осуществляется вторым методом. Кисть заливки передается через параметр `Brush`.

Траектории GDI весьма подробно описаны в главе 8 этой книги, поэтому сразу перейдем к вопросу реализации траекторий в GDI+.

Траектория, класс TGPGraphicsPath

Создавать траекторию можно несколькими способами. Например, допустимо воспользоваться набором координат точек и передать их в один из представленных ниже конструкторов:

```
constructor Create(pPoints: pGPPointF; pTypes: pBYTE; Count: Integer;
                   FillMode: TFillMode = FillModeAlternate); reintroduce; overload;
constructor Create(pPoints: pGPPoint; pTypes: pBYTE; Count: Integer;
                   FillMode: TFillMode = FillModeAlternate); reintroduce; overload;
```

Разница между методами только в формате представления точек – в виде целых или действительных чисел. Указатель на массив с координатами направляется в первый параметр конструктора `pPoints`. В свою очередь, второй параметр также надеется получить от нас указатель, но на этот раз на массив описаний точек (порядок заполнения которого весьма напоминает порядок работы с рассмотренной в главе 6 функцией GDI `PolyDraw`). Каждый из элементов массива описаний представляет собой комбинацию элементов перечисления `TPathPointType` (табл. 31.1). Параметр `Count` уточняет общее количество элементов в массиве. От состояния четвертого по счету параметра `FillMode` зависит режим заполнения области траектории при ее закраске кистью. Таких режимов всего два:

```
type TFillMode = (FillModeAlternate, FillModeWinding);
```

и они полностью соответствуют режимам закраски региона (см. табл. 9.1), устанавливаемым посредством обращения к функции Windows GDI `SetPolyFillMode()`.

Таблица 31.1. Элементы перечисления TPathPointType

Элемент перечисления	Значение	Описание	
`PathPointTypeStart`	`$00`	Начало фигуры.	Тип точки
`PathPointTypeLine`	`$01`	Начало или окончание линии.	
`PathPointTypeBezier`, `PathPointTypeBezier3`	`$03`	Направляющая либо отклоняющая точка кривой Безье.	
`PathPointTypePathTypeMask`	`$07`	Снимает все флаги в 3–7 битах.	

Элемент перечисления	Значение	Описание	
PathPointTypeDashMode	$10	Не используется.	Атрибуты точки
PathPointTypePathMarker	$20	Маркер траектории. Он разделяет траекторию на секции.	
PathPointTypeCloseSubpath	$80	Флаг завершения фигуры.	

При определении элемента массива описания точек надо знать, что его 0..2 биты определяют тип точки, биты 3..7 содержат флаги (атрибуты) точки. Следующий пример демонстрирует процесс создания траектории:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPPen:TGPPen;
    GraphicsPath : TGPGraphicsPath;
const
Points : Array[0..3] of
         TGPPoint=((X:50;Y:0),(X:100;Y:50),(X:50;Y:100),(X:0;Y:50));
Types  : Array[0..3] of
         TPathPointType =(0, // PathPointTypeStart
                          1, // PathPointTypeLine
                          1, // PathPointTypeLine
                        $81); // PathPointTypeLine OR PathPointTypeCloseSubpath
begin
    GPGraphics:=TGPGraphics.Create(DC);
    GPPen:=TGPPen.Create(aclBlack,1);

    GraphicsPath:=TGPGraphicsPath.Create(pGPPoint(@Points),pByte(@Types),4);
    GPGraphics.DrawPath(GPPen,GraphicsPath);
    GPGraphics.Free; GPPen.Free; GraphicsPath.Free;
end;
```

Формируемая процедурой траектория представляет собой ромб. Она описывается четверкой точек с координатами, хранящимися в массиве Points, и характеристиками точек из массива Types.

Маркер PathPointTypePathMarker *позволяет разделять траекторию на секции, например, по используемым в ней графическим примитивам. Позднее, опираясь на поддержку класса* TGPGraphicsPathIterator*, вы сможете изолировать одну или несколько секций из состава траектории.*

Среди всех подходов к созданию экземпляра траектории описание траектории на основе массива с координатами точек – самый быстрый с точки зрения компьютера, но далеко не самый удобный для программиста. Этот недостаток особо бросается в глаза при попытке построить траекторию с более сложной формой. Поэтому у разработчиков прикладного программного обеспечения гораздо большей популярностью пользуются более простые конструкторы:

```
constructor Create(Path: TGPGraphicsPath); Reintroduce; Overload;
constructor Create(FillMode: TFillMode = FillModeAlternate); Reintroduce; Overload;
```

Первый из методов создает новый экземпляр TGPGraphicsPath на основе уже существующей траектории, по сути это просто клонирование. Второй метод формирует траекторию, не содержащую ни одной точки. Единственное, с чем необходимо определиться перед обращением к этой вариации конструктора, – режим заполнения фигуры FillMode. Позднее к пустой траектории могут быть добавлены любые фигуры.

К любой из траекторий может быть добавлено неограниченное количество фигур, которые в свою очередь состоят из отдельных графических примитивов.

Класс TGPGraphicsPath не возражает против изменения режима заполнения траектории на лету. Для этого предназначен метод:

```
function SetFillMode(Fillmode: TFillMode): TStatus;
```

Привычный напарник метода «Set»:

```
function GetFillMode: TFillMode;
```

возвратит нам данные о текущем режиме.

Формирование фигур траектории

Траектория может состоять из одной или нескольких фигур. Все фигуры могут быть заданы сразу, в момент вызова конструктора класса TGPGraphicsPath, или добавлены к уже существующей траектории. Сигналом о том, что мы начинаем формировать новую фигуру, служит обращение к методу:

```
function StartFigure: TStatus;
```

С физической точки зрения это не что иное, как добавление константы PathPointTypeStart (см. табл. 31.1) в текущий элемент массива описаний точек. После этого производится описание формы траектории.

По логике вещей описание любой траектории должно начинаться с вызова метода StartFigure(), *однако если вы вдруг об этом забудете, не стоит расстраиваться, это будет сделано автоматически.*

При необходимости замкнуть первую и последнюю точку фигуры вызываем метод:

```
function CloseFigure: TStatus;
```

На практике это не что иное, как установка флага PathPointTypeCloseSubpath в соответствующий элемент массива описаний точек. Если по какой-то причине при описании нескольких фигур мы ленились закрывать каждую из фигур по отдельности (вызывать CloseFigure), то можно их «захлопнуть» за один раз. Для этого предназначен метод:

```
function CloseAllFigures: TStatus;
```

Далее представлен фрагмент кода, демонстрирующий порядок вывода двух одинаковых треугольников.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var
  GPGraphics:TGPGraphics;
  GPPen:TGPPen;
  GraphicsPath : TGPGraphicsPath;
begin
    GPGraphics:=TGPGraphics.Create(DC);
    GPPen:=TGPPen.Create(aclNavy,12);

//1 треугольник выводим с помощью DrawLine()
    GPGraphics.DrawLine(GPPen,  15,155,205,155);
    GPGraphics.DrawLine(GPPen,205,155,110,35);
    GPGraphics.DrawLine(GPPen,110,35,15,155);
//2 треугольник рисуем средствами траектории
    GraphicsPath:=TGPGraphicsPath.Create;
     GraphicsPath.StartFigure;
     GraphicsPath.AddLine(235,155,425,155);
     GraphicsPath.AddLine(425,155,330,35);
     GraphicsPath.CloseFigure;
    GPGraphics.DrawPath(GPPen,GraphicsPath);

    GPGraphics.Free;
    GPPen.Free;
    GraphicsPath.Free
end;
```

Первый треугольник рисуется обычными средствами – с помощью трех обращений к методу `DrawLine()`. Второй треугольник формируется в виде отдельной фигуры, поэтому он содержит всего две линии, но благодаря команде `CloseFigure()` были соединены начальная и конечная точка – в итоге два отрезка превратились в замкнутую фигуру. Обратите внимание, что изображение второго треугольника (рис. 31.1) получилось более симпатичным – траектория позаботилась об улучшении вида в местах соединений линий.

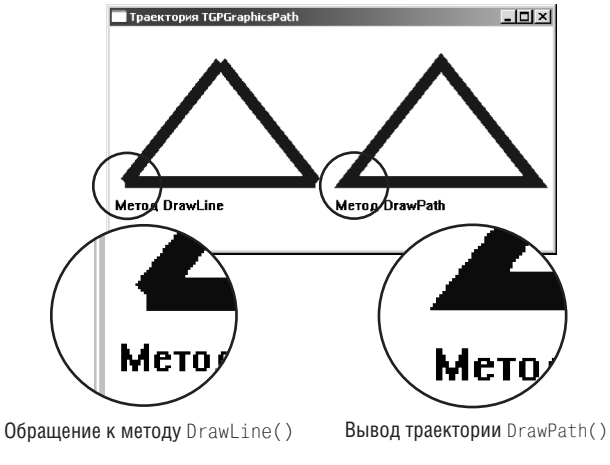


Рис. 31.1. Вывод треугольника обычными средствами и в виде траектории

Траектории обладают рядом полезных особенностей. Во-первых, при выводе сложных фигур средствами траекторий программисту не стоит заботиться об улучшении внешнего вида стыков – за это отвечает графический механизм. Во-вторых, траектория никогда не закрашивает одну и ту же точку дважды. Это особенно важно при работе с печатающим устройством и при проведении растровых операций.

Траектория способна поменять порядок следования элементов в массивах координат и описаний точек. Вывернуться наизнанку поможет метод:

```
function Reverse: TStatus;
```

Для очистки траектории предназначен метод:

```
function Reset: TStatus;
```

Функция сброса не только очистит все элементы массивов, но и вернет режим заполнения в состояние `FillModeAlternate`.

Элементы траектории

Безусловно, с точки зрения разработчика весьма утомительно описывать траекторию так, как было продемонстрировано в самом начале этой главы, – поэлементно заполняя массив координат и их описаний. Поэтому даже в GDI сразу был предусмотрен альтернативный способ построения траектории сложной формы, он основан на обращении к функциям, рисующим графические примитивы (линии, кривые, прямоугольники и т. п.). В GDI+ процесс построения траектории еще более усовершенствован. Для этого в состав класса `TGPGraphicsPath` включен широкий перечень методов (их название начинается с префикса «Add»), позволяющих добавлять к траектории различные графические примитивы.

Для проведения простейшей операции – добавления в состав текущей фигуры линии – предназначены четыре версии перегружаемого метода:

```
function AddLine(x1, y1, x2, y2: Integer): TStatus; overload;
function AddLine(x1, y1, x2, y2: Single): TStatus; overload;
function AddLine(const pt1, pt2: TGPPoint): TStatus; overload;
function AddLine(const pt1, pt2: TGPPointF): TStatus; overload;
```

Услугами одного из перечисленных методов мы не так давно воспользовались. Еще одна пара функций позволяет нарисовать несколько ломаных линий:

```
function AddLines(pPoints: PGPPoint; Count: Integer): TStatus; overload;
function AddLines(pPoints: PGPPointF; Count: Integer): TStatus; overload;
```

В этом случае перед обращением к методу требуется заполнить координатами точек массив и передать на него указатель.

Понадобился прямоугольник? Нет проблем:

```
function AddRectangle(Rect: TGPRect): TStatus; overload;
function AddRectangle(Rect: TGPRectF): TStatus; overload;
```

Главное, не забудьте указать его координаты. Если траектория содержит несколько прямоугольников, то вместо многократного обращения к предыдущему методу целесообразно обратиться к функциям:

```
function AddRectangles(pRects: PGPRect; Count: Integer): TStatus; overload;
function AddRectangles(pRects: PGPRectF; Count: Integer): TStatus; overload;
```

Ближайший «родственник» прямоугольника – эллипс – в два счета рисуется любым понравившимся вам методом:

```
function AddEllipse(X, Y, Width, Height: Integer): TStatus; overload;
function AddEllipse(X, Y, Width, Height: Single): TStatus; overload;
function AddEllipse(Rect: TGPRect): TStatus; overload;
function AddEllipse(Rect: TGPRectF): TStatus; overload;
```

Все входящие в состав траектории кривые всегда конвертируются в сплайны Безье.

Методы, специализирующиеся на черчении многоугольников:

```
function AddPolygon(pPoints: PGPPoint; Count: Integer): TStatus; overload;
function AddPolygon(pPoints: PGPPointF; Count: Integer): TStatus; overload;
```

рассчитывают получить указатель на массив с координатами вершин многоугольника и информацию о числе этих вершин.

Настоящее раздолье ожидает любителей включать в траектории дуги:

```
function AddArc(X, Y, Width, Height,
                StartAngle, SweepAngle: Single): TStatus; overload;
function AddArc(X, Y, Width, Height: Integer;
                StartAngle, SweepAngle: Single): TStatus; overload;
function AddArc(Rect: TGPRect;
                StartAngle, SweepAngle: Single): TStatus; overload;
function AddArc(Rect: TGPRectF;
                StartAngle, SweepAngle: Single): TStatus; overload;
```

Правила рисования таковы. Параметрами `X`, `Y`, `Width`, `Height` или параметром `Rect` задается прямоугольник, в который вписывается эллипс. Параметр `StartAngle` определяет величину угла в градусах от горизонтальной оси до луча, пересечение которого с контуром эллипса задает начало дуги. Параметр `SweepAngle` ожидает получить значение угла между лучами, определяющими начало и конец дуги.

Практически ничем не отличается порядок рисования сектора эллипса:

```
function AddPie(X, Y, Width, Height: Integer;
                StartAngle, SweepAngle: Single): TStatus; overload;
function AddPie(X, Y, Width, Height,
                StartAngle, SweepAngle: Single): TStatus; overload;
function AddPie(Rect: TGPRect;
                StartAngle, SweepAngle: Single): TStatus; overload;
function AddPie(Rect: TGPRectF;
                StartAngle, SweepAngle: Single): TStatus; overload;
```

Впечатляет весьма серьезная поддержка кривых Безье. Для добавления в состав траектории единственного кубического сплайна потребуется один из следующих методов:

```
function AddBezier(X1,Y1, X2, Y2, X3, Y3, X4, Y4: Integer): TStatus; overload;
function AddBezier(X1,Y1, X2, Y2, X3, Y3, X4, Y4: Single): TStatus; overload;
function AddBezier(Pt1, Pt2, Pt3, Pt4: TGPPoint): TStatus; overload;
function AddBezier(Pt1, Pt2, 3t3, 3t4: TGPPointF): TStatus; overload;
```

Здесь: (X1,Y1) или Pt1 – начальная точка сплайна; (X2,Y2) или Pt2 – конечная точка; (X3,Y3) и (X4,Y4) или Pt3 и Pt4 – отклоняющие точки. При работе с двумя или большим числом кубических сплайнов можно воспользоваться услугами следующих методов:

```
function AddBeziers(pPoints: PGPPoint; Count: Integer): TStatus; overload;
function AddBeziers(pPoints: PGPPointF; Count: Integer): TStatus; overload;
```

Для тех, кому непросто управляться с отклоняющими точками, вместо кубических сплайнов Безье можно воспользоваться методами, строящими кривые прямо по точкам:

```
function AddCurve(pPoints: PGPPoint; Count: Integer): TStatus; overload;
function AddCurve(pPoints: PGPPointF; Count: Integer): TStatus; overload;
function AddCurve(pPoints: PGPPoint; Count: Integer;
                           Tension: Single): TStatus; overload;
function AddCurve(pPoints: PGPPointF; Count: Integer;
                           Tension: Single): TStatus; overload;
function AddCurve(pPoints: PGPPoint;
                  Count, Offset, NumberOfSegments: Integer;
                  Tension: Single): TStatus; overload;
function AddCurve(pPoints: PGPPointF;
                  Count, Offset, NumberOfSegments: Integer;
                  Tension: Single): TStatus; overload;
```

Такой способ построения канонических сплайнов – фирменный знак GDI+. Помимо координат точек, по которым будет построена кривая, и их количества некоторые методы требуют дополнительных параметров. Так, параметр Tension определяет степень «натяжения» кривой, это значение, контролирующее длину и изогнутость сплайна. Параметр Offset позволяет изменить номер элемента массива, начиная с которого начнется рисование сплайна. Количество сегментов в кривой определяется параметром NumberOfSegments. О каноническом сплайне мы весьма подробно поговорим в главе 33 «Графические примитивы и заливка областей в GDI+», поэтому переходим к очередным методам траектории.

Для добавления в состав траектории замкнутой кривой обращаемся к любой из представленных ниже функций:

```
function AddClosedCurve(pPoints: PGPPoint; Count: Integer): TStatus; overload;
function AddClosedCurve(pPoints: PGPPointF; Count: Integer): TStatus; overload;
function AddClosedCurve(pPoints: PGPPoint; Count: Integer;
                        Tension: Single): TStatus; overload;
```

```
function AddClosedCurve(pPoints: PGPPointF; Count: Integer;
                        Tension: Single): TStatus; overload;
```

Перечень и назначение параметров методов аналогичны параметрам из методов `AddCurve()`.

Группа методов позволяет включить в состав траектории текстовую строку.

```
function AddString(Str_: WideString; Length: Integer;
                   Family : TGPFontFamily; Style  : Integer;
                   emSize : Single; Origin : TGPPointF;
                   Format : TGPStringFormat): TStatus; overload;
function AddString(Str_: WideString; Length : Integer;
                   Family : TGPFontFamily; Style  : Integer;
                   emSize : Single; LayoutRect: TGPRectF;
                   Format : TGPStringFormat): TStatus; overload;
function AddString(Str_: WideString; Length : Integer;
                   Family : TGPFontFamily; Style  : Integer;
                   emSize : Single; Origin : TGPPoint;
                   Format : TGPStringFormat): TStatus; overload;
function AddString(Str_: WideString; Length : Integer;
                   Family : TGPFontFamily; Style  : Integer;
                   emSize: Single; LayoutRect: TGPRect;
                   Format : TGPStringFormat): TStatus; overload;
```

Здесь `Str` содержит включаемый в состав траектории текст, `Length` – число подлежащих выводу символов, `Family` – семейство шрифтов, которым впоследствии должен быть отображен текст. Параметр `Style` определяет стиль текста (подчеркивание, курсив и т. п.), перечень допустимых значений ограничивается `TFontStyle`. Размер шрифта назначается в аргументе `emSize`. Область вывода ограничивается прямоугольником с координатами `LayoutRect` или определяется верхней левой точкой `Origin`. И, наконец, тонкости форматирования назначаются в параметре `Format`. Многие из перечисленных здесь характеристик шрифта будут подробно рассмотрены в главах 38–39 этой книги, поэтому сейчас на них останавливаться не станем. Но без маленького примера не обойдется. Поставим себе задачу вывести текстовую строку, залитую сплошной голубой кистью и обрамленную контуром красного цвета.

```
Procedure  GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPFontFamily :TGPFontFamily;
    GPSolidBrush:TGPSolidBrush;
    GPPen:TGPPen;
    GPPoint:TGPPointF;
    GraphicsPath : TGPGraphicsPath;
    SF:TGPStringFormat;
const S='Траектория';
begin
GPGraphics:=TGPGraphics.Create(DC);

GPFontFamily:=TGPFontFamily.Create('Courier'); //семейство шрифтов
GPPoint:=MakePoint(0.0,0.0);                   //координата вывода
```

```
GPPen:=TGPPen.Create(aclRed,4); //перо для контура
GPSolidBrush:=TGPSolidBrush.Create(aclAqua); //кисть для заливки
SF:=TGPStringFormat.Create(0);  //простейший способ форматирования

GraphicsPath:=TGPGraphicsPath.Create;
GraphicsPath.AddString(S,-1,GPFontFamily,
                       FontStyleBold OR FontStyleItalic,50,GPPoint,SF);
GPGraphics.DrawPath(GPPen,GraphicsPath);
GPGraphics.FillPath(GPSolidBrush,GraphicsPath);

SF.Free; GPPen.Free; GraphicsPath.Free;
GPSolidBrush.Free;   GPFontFamily.Free; GPGraphics.Free;
end;
```

Результат выполнения процедуры представлен на рис. 31.2. Все получилось так, как мы и планировали, – в клиентской части окна возникла текстовая надпись, закрашенная сплошной кистью и ограниченная утолщенным пером. На такое способен не каждый текстовый редактор!

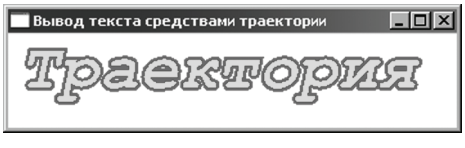


Рис. 31.2. Вывод текста средствами траектории

В заключении данного подраздела упомянем еще одну примечательную особенность класса `TGPGraphicsPath` – траектория может быть дополнена любой другой траекторией:

```
function AddPath(AddingPath: TGPGraphicsPath; Connect: Bool): TStatus;
```

Новая траектория передается через параметр `AddingPath`. Метод может попытаться присоединить первую фигуру добавляемой траектории к последней фигуре нашей траектории. Для этого в параметр `Connect` следует направить значение `true`.

Трансформации и преобразования траектории

Для проведения обычных аффинных преобразований проще всего воспользоваться методом:

```
function Transform(Matrix: TGPMatrix): TStatus;
```

Метод просто изменяет траекторию в соответствии с параметрами матрицы преобразования (параметр `Matrix`). Логическим продолжением метода `Transform()` выступает функция:

```
function Flatten(Matrix: TGPMatrix = nil;
                 Flatness: Single = FlatnessDefault): TStatus;
```

Задача метода точно такая же – трансформация траектории в соответствии с указаниями матрицы преобразования, но это еще не все. Изюминка функ-

ции в том, что все входящие в траекторию сплайны конвертируются в набор прямых отрезков. Качество преобразований находится в прямой зависимости от аргумента Flatness (если следовать дословному переводу с англ. – «гладкость», «ровность»), в который разрешено передавать любые положительные значения. С некоторым допущением можно сказать, что параметр определяет количество ошибок, которые разрешено допустить GDI+ при превращении кривых Безье в набор обычных отрезков. Уменьшение значения Flatness приводит к увеличению числа аппроксимирующих сегментов и, как следствие, приводит к улучшению качества изображения и повышению требований к производительности процессора компьютера. По умолчанию в параметр передается значение 0,25 (константа FlatnessDefault). Наглядным подтверждением моим словам может стать снимок окна программы (рис. 31.3), полученный в результате аффинных преобразований траектории, состоящей из окружности с разными значениями параметра Flatness.

На увеличенных фрагментах части деформированного рисунка заметно существенное ухудшение внешнего вида элементов траектории, трансформированных с параметром Flatness=5.

Контур траектории может быть получен при обращении к методу:

```
function Outline(Matrix: TGPMatrix = nil;
                 Flatness: Single = FlatnessDefault): TStatus;
```

Эта функция (впрочем, как и все изученные в этом подразделе методы) преобразует входящие в состав траектории графические примитивы в коллекцию прямых и, при необходимости, производит над ними аффинные преобразования с заданным качеством.

Эффект расширения контура траектории реализуется с помощью метода:

```
function Widen(Pen: TGPPen; Matrix: TGPMatrix = nil;
               Flatness: Single = FlatnessDefault): TStatus;
```

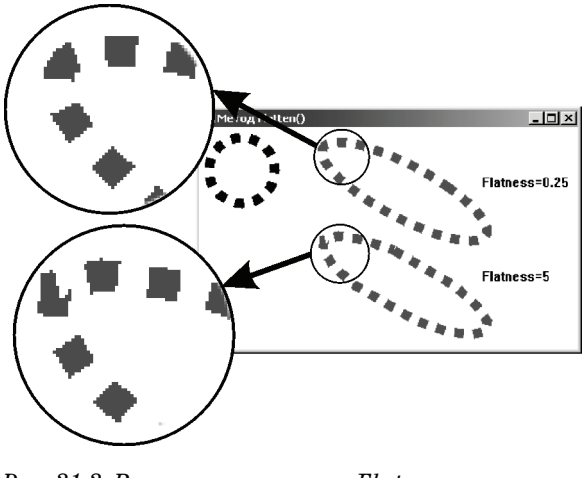


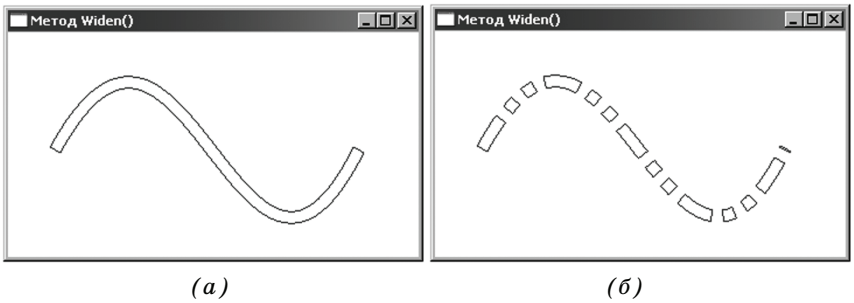
Рис. 31.3. Влияние параметра Flatness на качество результирующего изображения

Метод конвертирует траекторию в набор отрезков. В минимальной нотации результат зависит от стиля и толщины пера, передаваемого в параметр Pen. Рассмотрим следующий листинг:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPPen1,GPPen2:TGPPen;
    GraphicsPath : TGPGraphicsPath;
const Points:Array[0..3] of TGPPoint=
      ((X:40;Y:100),(X:150;Y:-100),(X:200;Y:300),(X:300;Y:100));
begin
    GPGraphics:=TGPGraphics.Create(DC);

    GPPen1:=TGPPen.Create(aclRed,10);
//  GPPen1.SetDashStyle(DashStyleDashDotDot);
    GPPen2:=TGPPen.Create(aclBlue);

    GraphicsPath:=TGPGraphicsPath.Create;
    GraphicsPath.AddBeziers(pGPPoint(@Points),4);
    GraphicsPath.Widen(GPPen1);

    GPGraphics.DrawPath(GPPen2,GraphicsPath);
    GPGraphics.Free; GPPen1.Free; GPPen2.Free; GraphicsPath.Free
end;
```

В коде создаются два экземпляра пера. Перо GPPen1 предназначено для работы в рамках метода Widen(), изначально оно представляет сплошное перо толщиной 10 логических единиц (рис. 31.4а). Пером единичной толщины GPPen2 мы позднее начертим контур траектории. Траектория строится по четырем точкам по правилам формирования кубического сплайна Безье, затем к ней применяется метод Widen(). Если вы снимите комментарий со строки, изменяющей стиль пера SetDashStyle(), то получите траекторию, повторяющую рисунок пера тире-точка-точка (рис. 31.4б).

Нам осталось обсудить два дополнительных параметра метода Widen(). Матрица Matrix позволяет определять порядок аффинных преобразований. Параметр Flatness определяет качество аппроксимации результирующего изображения.



(а) *(б)*

Рис. 31.4. Влияние атрибутов пера на работу метода Widen( )

Деформация и перенос траектории могут быть осуществлены методом:

```
function Warp(pDestPoints: pGPPointF; Count: Integer;
              SrcRect: TGPRectF; Matrix: TGPMatrix = nil;
              WarpMode: TWarpMode = WarpModePerspective;
              Flatness: Single = FlatnessDefault): TStatus;
```

Это как раз тот случай, когда возможно изменить фигуру без проведения аффинных преобразований (хотя метод готов подключить и этот механизм через матрицу Matrix). Деформация осуществляется путем переноса траектории из исходной прямоугольной области (описываемой параметром SrcRect) в четырехугольную область-получатель произвольной формы. Координаты вершин четырехугольника-получателя должны быть заранее сохранены в массиве, указатель на который передается в первый параметр метода pDestPoints. Допускается, чтобы этот массив состоял не из четырех, а из трех элементов, но в последнем случае координаты четвертой точки назначаются автоматически и, как следствие, четырехугольная область-получатель сменит вольный внешний вид на контур строгого параллелограмма. Между элементами массива вершины прямоугольника распределяются следующим образом: нулевой элемент массива pDestPoints содержит левую верхнюю точку; 1-й – правую верхнюю; 2-й – левую нижнюю; 3-й – правую нижнюю. Для того чтобы метод сообразил, каким количеством точек описана приемная область, это число должно быть направлено в параметр Count. Параметр WarpMode вправе принимать одно из двух значений: WarpModePerspective (по умолчанию) или WarpModeBilinear. Параметр Flatness, как и в предыдущем методе, определяет качество аппроксимации новой траектории.

Пример работы с методом представлен в следующих строках:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPPen:TGPPen;
    GraphicsPath : TGPGraphicsPath;
    SrcRect:TGPRectF;
const DestPoints : Array[0..3] OF TGPPointF=
      ((X:300; Y:10),(X:360; Y:10),(X:180;Y:150),(X:480;Y:150));

     PointF : Array[0..9] OF TGPPoint={рисунок пятиконечной звезды}
     ((X:0; Y:60), (X:60;Y:60),(X:80;Y:0),(X:100;Y:60),(X:160;Y:60),
      (X:110; Y:95),(X:130; Y:160),(X:80; Y:120),(X:30;Y:160),(X:50;Y:95));
begin
    GPGraphics:=TGPGraphics.Create(DC);

    GPPen:=TGPPen.Create(aclBlack);
    SrcRect:=MakeRect(0, 0, 160, 160.0); //прямоугольник-источник
    GPGraphics.DrawRectangle(GPPen, srcRect);

    GPPen.SetColor(aclGreen);
    with GPGraphics do {ечетырехугольник-получатель}
    begin
    DrawLine(GPPen, DestPoints[0], DestPoints[1]);
    DrawLine(GPPen, DestPoints[0], DestPoints[2]);
    DrawLine(GPPen, DestPoints[1], DestPoints[3]);
```

```
    DrawLine(GPPen, DestPoints[2], DestPoints[3]);
    end;

    GraphicsPath:=TGPGraphicsPath.Create;
    GraphicsPath.AddLines(pGPPoint(@PointF), 10);
    GraphicsPath.CloseFigure;
    GraphicsPath.AddEllipse(55,60,50,50);

    GPPen.SetColor(aclRed);
    GPGraphics.DrawPath(GPPen, GraphicsPath);//исходная траектория

    GraphicsPath.Warp(pGPPointF(@DestPoints),4,SrcRect);//деформация
    GPPen.SetColor(aclBlue);
    GPGraphics.DrawPath(GPPen,GraphicsPath);//траектория после деформации

    GPGraphics.Free; GPPen.Free; GraphicsPath.Free;
end;
```

Мы осуществляем перенос траектории, представляющей собой пятиконечную звезду, из прямоугольной области, описанной в `SrcRect`, в трапецию `DestPoints`. Результат представлен на рис. 31.5.

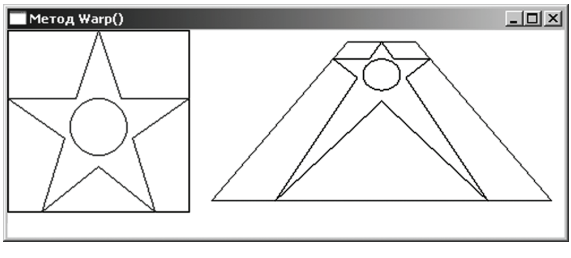


Рис. 31.5. Пример работы метода Wrap( )

Все рассмотренные в подразделе «Трансформации и преобразования траектории» методы, обладающие параметром `Flatness`*, конвертируют входящие в состав траектории графические примитивы в коллекцию прямых. При этом учитываются правила аффинных преобразований (задаваемых матрицей* `Matrix`*) и требуемое качество аппроксимаций (параметр* `Flatness`*).*

Исследование траектории

Экземпляр класса `TGPGraphicsPath` способен поведать нам много интересного о хранимой внутри него траектории. Процесс получения данных всегда начинается с вызова метода:

```
function GetPointCount: Integer;
```

информирующего о числе элементов в траектории. Уяснив размер изучаемой траектории, распределим память для двух динамических массивов (массива координат точек и массива описания точек). Указатель на первый из массивов передается в один из следующих методов:

```
function GetPathPoints(pPoints: PGPPoint; Count: Integer): TStatus; overload;
function GetPathPoints(pPoints: PGPPointF; Count: Integer): TStatus; overload;
```

Для заполнения массива описания типов точек нам потребуется помощь метода:

```
function GetPathTypes(Types: PBYTE; Count: Integer): TStatus;
```

На практике процесс исследования траектории может выглядеть следующим образом:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPPen:TGPPen;
    GraphicsPath : TGPGraphicsPath;
    Points : Array of TGPPoint;
    Types  : Array of TPathPointType;
    Count,i:Integer;
    s:string;
begin
    GPGraphics:=TGPGraphics.Create(DC);
    GPPen:=TGPPen.Create(aclBlack,3);
//создаем подопытную траекторию
    GraphicsPath:=TGPGraphicsPath.Create;
    GraphicsPath.AddRectangle(MakeRect(5,20,150,20));
    GraphicsPath.AddEllipse(120,5,120,120);
//создание траектории завершено
    Count:=GraphicsPath.GetPointCount;
    SetLength(Points,Count);
    SetLength(Types,Count);
    GraphicsPath.GetPathPoints(pGPPoint(Points),Count);
    GraphicsPath.GetPathTypes(pByte(Types),Count);

    GPGraphics.DrawPath(GPPen,GraphicsPath);
    GPPen.Free;

    GPPen:=TGPPen.Create(aclRed,3);
    For i:=0 to count-1 do
       begin
         s:=Format('№%d  (%d,%d) Type=$%x',[i,Points[i].X,Points[i].Y,Types[i]]);
         TextOut(DC,270,i*18,pChar(S),length(s));
         GPGraphics.DrawEllipse(GPPen,Points[i].X-2,Points[i].Y-2,4,4);
         S:=IntToStr(i);
         TextOut(DC,Points[i].X+2,Points[i].Y+2,pChar(s),length(S));
       end;
    GPGraphics.Free; GPPen.Free; GraphicsPath.Free
end;
```

Рассматриваемая траектория содержит два графических примитива: прямоугольник и эллипс (рис. 31.6). Для ее описания потребовалось 17 элементов, в примере для большей наглядности все ключевые точки обведены и пронумерованы.

Обратите внимание на то, что для хранения прямоугольника в массивах нашей траектории задействовано всего 4 элемента (с 0 по 3), они содержат координаты вершин. Оставшиеся 13 элементов отведены под описания эллипса. Такое расточительство объясняется тем, что эллипс (как, впрочем, и все

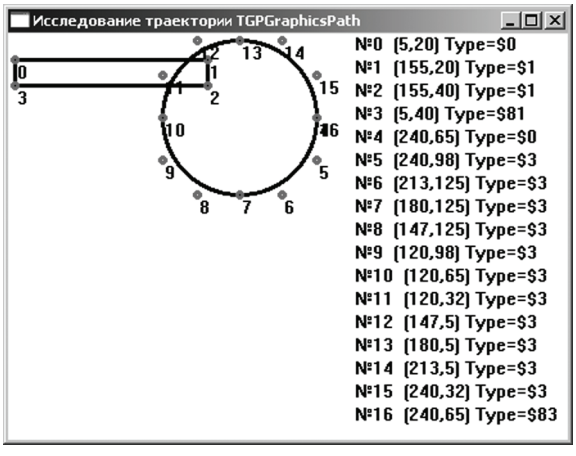


Рис. 31.6. Исследование траектории

кривые, входящие в состав траектории) конвертируется в кубические сплайны Безье. Именно поэтому четвертая точка траектории имеет признак начала фигуры (`PathPointTypeStart=$0`), точки с 5 по 16 помечены как последняя либо отклоняющие точки кривой Безье (`PathPointTypeBezier=$3`), кроме того, последняя точка «размахивает» флагом завершения фигуры.

Полученные в результате исследования траектории данные после незначительной доработки могут быть воспроизведены с помощью функции Windows GDI `PolyDraw()`.

Еще один способ выведывания секретов траектории предоставляет метод:

```
function GetPathData(PathData: TPathData): TStatus;
```

Фактически это не что иное, как объединение методов `GetPointCount()` и `GetPathPoints()`. Результаты возвращаются в параметр `PathData`, представляющий собой экземпляр класса `TPathData`.

```
type TPathData = packed class
  public
    Count  : Integer;    //количество точек в траектории
    Points : pGPPointF;  //указатель на массив координат
    Types  : pBYTE;      //указатель на массив описаний типов точек
    constructor Create;
    destructor Destroy; override;
  end;
```

Для получения координат последней точки траектории вместо штурма траектории средствами только что изученных методов проще обратиться к функции:

```
function GetLastPoint(out LastPoint: TGPPointF): TStatus;
```

Результат окажется в выходном параметре `LastPoint`.

Прямоугольные границы траектории

Для выяснения координат обрамляющей траекторию прямоугольной области стоит обратиться к методам:

```
function GetBounds(out Bounds: TGPRectF;
         Matrix: TGPMatrix = nil; Pen: TGPPen = nil): TStatus; overload;
function GetBounds(out Bounds: TGPRect;
         Matrix: TGPMatrix = nil; Pen: TGPPen = nil): TStatus;  overload;
```

Результат возвращается через параметр `Bounds`. Функции содержат два опциональных параметра. Первый представляет собой матрицу `Matrix`, он необходим в случае, если вы намерены спрогнозировать поведение траектории после осуществления с ней аффинных преобразований. При этом реально траектория не трансформируется, все считается «в уме». Параметр `Pen` повышает точность расчетов. Так как GDI+ разумно полагает, что в состав траектории входят все точки ее внешней границы (которая может быть проведена пером различной толщины), то при возврате координат прямоугольной области методы готовы учесть и атрибуты пера, передаваемые через параметр `Pen`.

Вхождение точки в траекторию

При работе с траекториями, обладающими сложной формой, достаточно часто встречается задача по проверке факта вхождения точки в область траектории. Для этого в состав класса `TGPGraphicsPath` введено четыре метода, отличающихся между собой лишь способом передачи координат точки:

```
function IsVisible(Point: TGPPointF; G: TGPGraphics = nil): BOOL; overload;
function IsVisible(Point: TGPPoint; G : TGPGraphics = nil): BOOL; overload;
function IsVisible(X, Y: Integer; G: TGPGraphics = nil): BOOL; overload;
function IsVisible(X, Y: Single; G: TGPGraphics = nil): BOOL; overload;
```

Если точка входит в состав траектории, то метод возвратит значение `True`. Опциональный параметр `G` необходим только тогда, когда мы намерены узнать, будет ли точка входить в область траектории при осуществлении с холстом какого-то аффинного преобразования. В таком случае требуется задействовать в нашем эксперименте экземпляр холста, провести преобразования и передать ссылку на холст в качестве параметра методу.

Рассмотрим пример использования метода. Обратите внимание на то, что заголовок процедуры содержит не только дескриптор контекста устройства `DC`, но и параметр `FillMode`, определяющий порядок заполнения траектории.

```
Procedure GDIPLUS_PAINT(DC:HDC; FillMode: TFillMode);
var GPGraphics:TGPGraphics;
    GPPen1,GPPen2:TGPPen;
    GPBrush:TGPBrush;
    GraphicsPath : TGPGraphicsPath;
    i:Integer;
const Points:Array[0..4] of TGPPoint=
     ((X:10;Y:20),(X:110;Y:66),(X:150;Y:80),(X:220;Y:60),(X:220;Y:120));
begin
```

```
    GPGraphics:=TGPGraphics.Create(DC);
    GPPen1:=TGPPen.Create(aclRed,3);
    GPPen2:=TGPPen.Create(aclBlue,3);

    GPBrush:=TGPSolidBrush.Create(aclAqua);

    GraphicsPath:=TGPGraphicsPath.Create(FillMode);
    GraphicsPath.AddEllipse(50,50,120,120);
    GraphicsPath.AddEllipse(120,5,120,120);
    GPGraphics.FillPath(GPBrush,GraphicsPath);

    For i:=0 to SizeOf(Points)-1 Do
       if GraphicsPath.IsVisible(Points[i]) then
          GPGraphics.DrawEllipse(GPPen1,Points[i].X-3,Points[i].Y-3,6,6)
          else
          GPGraphics.DrawRectangle(GPPen2,Points[i].X-3,Points[i].Y-3,6,6);

    GPGraphics.Free; GPPen1.Free; GPPen2.Free; GraphicsPath.Free;
end;
```

Нами создана траектория, состоящая из двух пересекающихся окружностей. В цикле мы поочередно анализируем все точки из состава массива `Points` на предмет вхождения в траекторию. Если это так, то точка изображается в виде эллипса, в противном случае рисуется квадрат. На рис. 31.7 представлены два экранных снимка работающей программы, разница между ними заключается в режиме заполнения траектории (левый снимок – `FillModeAlternate`, правый – `FillModeWinding`). Заметьте разницу в оценке факта принадлежности к траектории точки, расположенной точно в области пересечения окружностей. На левом снимке она не вошла в состав траектории, а на правом – вошла.

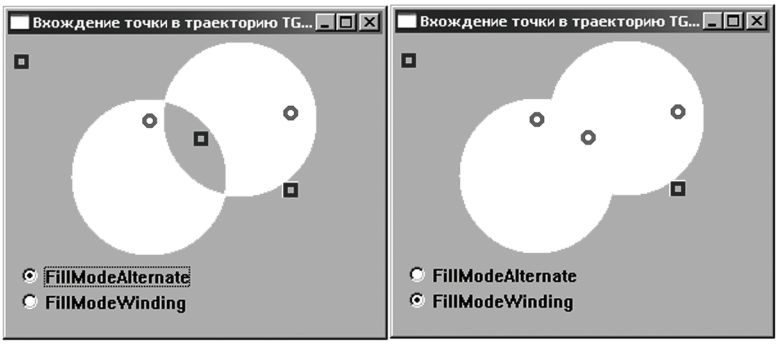


Рис. 31.7. Проверка вхождения точки в траекторию в разных режимах заполнения

Нахождение точки на внешней границе траектории

Еще более сложную задачу призвана решать еще одна четверка методов траектории:

```
function IsOutlineVisible(Point: TGPPointF; Pen: TGPPen;
                          G: TGPGraphics = nil): BOOL; overload;
```

```
function IsOutlineVisible(X, Y: Single; Pen: TGPPen;
                          G: TGPGraphics = nil): BOOL; overload;
function IsOutlineVisible(Point: TGPPoint; Pen: TGPPen;
                          G: TGPGraphics = nil): BOOL; overload;
function IsOutlineVisible(X, Y: Integer; Pen: TGPPen;
                          G: TGPGraphics = nil): BOOL; overload;
```

Методы проверяют факт нахождения точки на внешней границе контура траектории. Функции заслуживают большого уважения еще и потому, что на результат оценки влияет дополнительный ингредиент – перо, передаваемое в параметр `Pen`. В остальном методы похожи на своих коллег `IsVisible()`.

На рис. 31.8 предложена графическая иллюстрация работы метода `IsOutlineVisible()`. Экранный снимок получен за счет незначительной переработки предыдущего примера. Траектория вновь строится из двух окружностей, граница которых чертится пунктирным пером. Обратите внимание на точку, находящуюся в области пересечения двух кругов: хотя она расположилась точно на контуре левой окружности, она, тем не менее, помечена квадратным маркером. Причина в том, что это не внешняя граница траектории.

Рис. 31.8. Проверка нахождения точки на внешней границе траектории

Маркеры траектории

В начале главы при обсуждении массива описаний типа точек мы уже упоминали существование флага `PathPointTypePathMarker` (см. табл. 31.1). Это очередное ноу-хау GDI+ называется маркером траектории. С помощью маркера программист отмечает отдельные фигуры или примитивы траектории, таким образом траектория разделяется на две или большее количество секций.

Траектория `TGPGraphicsPath` не оснащена каким-либо существенным набором возможностей по обслуживанию своих маркеров. Стоит выделить лишь два метода. Первый из них позволяет установить маркер в последний элемент массива описаний точек:

```
function SetMarker: TStatus;
```

Второй метод удаляет все маркеры из траектории:

```
function ClearMarkers: TStatus;
```

На этом зона ответственности объекта TGPGraphicsPath завершается, и наступает черед вспомогательного класса TGraphicsPathIterator.

Последовательный просмотр траектории, класс TGraphicsPathIterator

Класс TGraphicsPathIterator позволяет проводить выборочное обслуживание элементов траектории. Такая услуга может оказаться особенно полезной при работе с траекториями, содержащими несколько сложных фигур. Просматривая подобную траекторию, мы сможем отказаться от вывода каких-то ее компонентов, изменить перо или кисть при прорисовке отдельной фигуры и даже экспортировать выбранную секцию или фигуру в другую траекторию.

Траектория просматривается последовательно от первой до последней точки массива, содержащего описание типов точек. Именно информация, хранящаяся в элементах массива описания, позволит осуществить:

1. Перебор секций траектории по ее маркерам (поиск флага PathPointTypePathMarker).
2. Перебор по входящим в траекторию фигурам (поиск значения PathPointTypeStart).
3. Перебор элементов траектории по всем имеющимся типам контрольных точек и флагов TPathPointType.

Существование объекта TGraphicsPathIterator бессмысленно без самой траектории, поэтому при создании экземпляра класса:

```
constructor Create(Path: TGPGraphicsPath); reintroduce;
```

требуется указать обслуживаемую траекторию.

Перебор секций

Число секций в траектории сумеет узнать метод:

```
function GetSubpathCount: Integer;
```

Перебор секций траектории производится с помощью пары перегружаемых методов:

```
function NextMarker(out StartIndex, EndIndex: Integer): Integer; overload;
function NextMarker(Path: TGPGraphicsPath): Integer; overload;
```

Первая версия метода возвратит индексы первого и последнего элементов очередной секции. Если вы воспользовались этим методом, то для получения данных о точках секции потребуется помощь функции CopyData(). Вторая версия метода NextMarker() всю информацию об исследуемой секции направит в траекторию Path. Ко всему прочему обе функции возвратят количество точек в исследуемой секции.

Ниже предложен листинг, дающий представление о работе с экземпляром класса TGraphicsPathIterator.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics; GPPen:TGPPen;
    GPGraphicsPath, Section : TGPGraphicsPath;
    GPGraphicsPathIterator: TGPGraphicsPathIterator;
    Count:Integer;
begin
    GPGraphics:=TGPGraphics.Create(DC);

    GPGraphicsPath:=TGPGraphicsPath.Create;

    GPGraphicsPath.AddEllipse(MakeRect(10, 10, 60, 30));
    GPGraphicsPath.SetMarker;   {МАРКЕР 1}
    GPGraphicsPath.AddBezier(80,20,100,20,80,0,90,60);
    GPGraphicsPath.AddArc(180, 20, 60, 30, 0.0, 180);
    GPGraphicsPath.SetMarker;   {МАРКЕР 2}
    GPGraphicsPath.AddRectangle(MakeRect(250, 10, 60, 30));
    GPGraphicsPath.SetMarker(); {МАРКЕР 3}
    GPGraphicsPath.AddArc(340, 10, 60, 30, 0.0, 180);
    GPGraphicsPath.CloseFigure();
    GPGraphicsPath.SetMarker(); {МАРКЕР 4}
    GPGraphicsPath.AddEllipse(MakeRect(420, 10, 30, 30));

   GPGraphicsPathIterator:=TGPGraphicsPathIterator.Create(GPGraphicsPath);

   GPPen:=TGPPen.Create(aclBlack,1);
   Section:=TGPGraphicsPath.Create;
   count:=GPGraphicsPathIterator.NextMarker(Section);
   While count<>0 do
   begin
    Randomize;
    GPPen.SetColor(MakeColor(255,RANDOM(255),RANDOM(255),RANDOM(255)));
    GPGraphics.DrawPath(GPPen,Section); //рисуем секцию
    count:=GPGraphicsPathIterator.NextMarker(Section);
   end;

GPGraphics.Free; GPPen.Free; GPGraphicsPath.Free; GPGraphicsPathIterator.Free;
end;
```

Из простейших примитивов формируется траектория и разбивается четырьмя маркерами на пять секций. Далее с помощью метода `NextMarker()` мы последовательно просматриваем все секции и выводим их на поверхность графического устройства пером со случайным цветом.

Зачастую после просмотра всей траектории или ее части возникает необходимость вернуться к ее началу. Обратная перемотка осуществляется процедурой:

```
procedure Rewind;
```

Перебор фигур

К методам `NextMarker()` очень близка группа функций:

```
function NextSubpath(out StartIndex, EndIndex: Integer;
                     out IsClosed: BOOL): Integer; overload;
```

```
function NextSubpath(Path: TGPGraphicsPath;
                     out IsClosed: BOOL): Integer; overload;
```

Но в данном случае методы разделяют траекторию не по маркерам, а по признаку завершения фигуры – флагу PathPointTypeCloseSubpath (см. табл. 31.1). Все параметры функции выходные. Первая версия метода содержит выходные параметры StartIndex и EndIndex, в них будут переданы индексы первой и последней точки обрабатываемой фигуры. Если же мы воспользуемся вторым методом, то в его параметре Path мы обнаружим копию фигуры. По состоянию IsClosed можно судить о том, закрыта или открыта фигура. Обе функции возвращают число точек в фигуре, нулевое значение свидетельствует о том, что работа с траекторией завершена.

Перебор по всем типам точек

Наконец, наиболее универсальный метод позволяет «пролистать» траекторию по всем типам точек:

```
function NextPathType(out PathType: TPathPointType;
                      out StartIndex, EndIndex: Integer): Integer;
```

Метод возвращает стартовый (StartIndex) и конечный (EndIndex) индексы следующей группы точек, имеющих один и тот же тип. Описание типа точки передается в параметр PathType. Например, если мы зададимся целью отслеживать кривые Безье, нам надо контролировать значения PathPointTypeBezier и PathPointTypeBezier3. Впрочем, для проверки наличия в траектории кривых проще обратиться к функции:

```
function HasCurve: BOOL;
```

Функция возвратит значение True, встретив в массиве описаний типов точек первый же сплайн.

Для выяснения, сколько элементов массива задействовано для описания траектории, стоит познакомиться с методом:

```
function GetCount: Integer;
```

Экспорт данных в массивы

Копирование всех данных о траектории в массив координат и в соответствующий ему массив описаний точек осуществляет метод:

```
function Enumerate(pPoints: PGPPointF;
                   pTypes: PBYTE; Count: Integer): Integer;
```

По сути, это аналог принадлежащих классу TGPGraphicsPath методов GetPathPoints() и GetPathTypes().

Метод:

```
function CopyData(pPoints: PGPPointF; pTypes: PBYTE;
                  StartIndex, EndIndex: Integer): Integer;
```

предназначен для копирования подмножества элементов из массива координат и массива описаний точек. Указатели на данные массивы передаются через первый и второй параметры функции. Диапазон копируемых данных назначается с помощью аргументов `StartIndex` и `EndIndex`.

РЕЗЮМЕ

Траектория предназначена для формирования фигур сложной формы. Для описания траекторий в GDI+ введен класс `TGPGraphicsPath`, он обладает точно такими же возможностями по управлению траекториями, как и GDI, кроме того, `TGPGraphicsPath` вооружен существенным спектром вспомогательных методов. Фирменными новинками траектории GDI+ считается поддержка двумерных аффинных преобразований, расширенный набор метод по преобразованию криволинейных участков траекторий в набор прямых отрезков и упрощенный механизм перебора элементов траектории.

32

Регионы GDI+

В Windows регион предназначен для описания определенного множества точек из состава координатного пространства окна, так что к нему логичнее относиться как к некоей загадочной структуре данных, чем как к графическому объекту. Почему загадочной? Потому что Microsoft принципиально не раскрывает, каким образом осуществляется физическое хранение перечня входящих в регион точек, полагая, что разработчику прикладного программного обеспечения это не важно. С этим остается только смириться, хотя ответ на этот вопрос весьма интересен для региона нетривиальной формы.

Ну что ж, продолжим наш разговор о графической подсистеме и регионах. Подтверждением того, что регион не вполне вписывается в стройные ряды остальных объектов GDI, является хотя бы то, что он не связан с контекстом графического устройства. Все функции Windows API, работающие с регионом, не требуют передать им наш любимый параметр `DC:HDC`. Исключение представляет единственная функция, создающая регион из траектории, – `PathToRegion()`. Но наличие дескриптора контекста в перечне ее аргументов объясняется потребностями траектории, а не региона.

В GDI+ прямое отношение к региону имеют два класса – регион `TGPRegion` и холст `TGPGraphics`. Если постараться в двух словах охарактеризовать роль каждого из указанных классов в работе с регионами, то, пожалуй, наиболее кратким определением станет следующее: класс `TGPRegion` представляет собой объектно-ориентированное описание множества точек, входящих в регион, а класс `TGPGraphics` отвечает за работу с самым главным регионом Windows – регионом отсечения. Других принципиальных нововведений в природу регионов GDI+ не внес, но сделал работу с регионами более удобной.

Регион, класс TGPRegion

Как уже упоминалось ранее, объектно-ориентированная версия региона GDI+ немногим отличается от старого региона GDI, перечень возможностей остался практически тем же:

- Хранение данных о совокупности точек координатного пространства.
- Различные способы объединения регионов.
- Проверка принадлежности к региону точки и прямоугольной области.

Самое существенное достижение GDI+ в том, что теперь объект региона инкапсулируется в экземпляре класса TGPRegion и изначально нацелен на работу в мировых координатах, таким образом, новый регион приобрел большую гибкость.

В отличие от региона GDI, хранящего свои данные в координатах устройства, регион GDI+ изначально работает в мировых координатах.

По сложившейся традиции изучение класса TGPRegion начнем с рассмотрения способов его создания. Простейший из конструкторов вдыхает жизнь в регион, включающий все допустимое множество точек. В мировых координатах это ни много ни мало – $2^{32} \times 2^{32}$ элементов.

```
constructor Create; reintroduce; overload;
```

Необходимость в существовании бесконечного региона относительна. Поэтому у программистов большей популярностью пользуются конструкторы, создающие регионы ограниченного размера, например прямоугольные:

```
constructor Create(Rect: TGPRectF); reintroduce; overload;
constructor Create(Rect: TGPRect); reintroduce; overload;
```

Определяя границы прямоугольного региона, помните, что точки, находящиеся на нижней и правой границах прямоугольной области, не войдут в состав точек региона.

Регион на основе траектории создается конструктором:

```
constructor Create(Path: TGPGraphicsPath); reintroduce; overload;
```

Благодаря траектории можно создавать регионы весьма причудливой формы, что мы и сделаем на последних страницах этой книги. Но пока не станем увлекаться и перейдем к очередному конструктору.

Регион допустимо сформировать на основе данных, полученных из другого региона:

```
constructor Create(pRegionData: pBYTE; Size: Integer); reintroduce; overload;
```

Описание региона передается через указатель на массив pRegionData, который в свою очередь может быть заполнен с помощью метода GetData(). Второй параметр метода определяет размер хранящего данные массива.

На первый взгляд, несколько неожиданный реверанс в сторону старого доброго GDI сделан конструктором:

```
constructor Create(hRgn: HRGN); reintroduce; overload;
```

Мы уже достаточно долго работаем с базирующейся на концепции ООП технологией GDI+, но при этом нечасто сталкивались хотя бы с упоминанием о том, что на белом свете существуют дескрипторы графических объектов

(в данном случае – дескриптор региона). Сейчас мы встретились с исключением из правил. Чем объясняется такой, прямо скажем неординарный, поступок программистов Microsoft, до сих пор отчаянно скрывавших роль GDI в жизнедеятельности библиотеки GDI+? Все дело в значительном влиянии регионов на всю систему графического вывода Windows – самим фактом своего существования многооконный интерфейс во многом обязан базирующемся на понятии региона механизме отсечения. В связи с этим значительное число функций Windows API ориентировано на работу с дескриптором региона, а этот факт трудно игнорировать даже зазнайке GDI+. В результате круг замкнулся – мы вновь вернулись к работе с дескриптором.

Стоит предположить, что рассмотренный выше конструктор пользуется услугами метода:

```
function FromHRGN(hRgn: HRGN): TGPRegion;
```

Здесь все то же самое, передаем в параметр дескриптор региона GDI, в ответ получаем объект `TGPRegion`.

В продолжение темы о взаимосвязи региона GDI+ со своими «прародителями» с глубоким удовлетворением отмечу факт существования еще одного метода:

```
function GetHRGN(Graphics: TGPGraphics): HRGN;
```

Указанная функция позволяет вернуть дескриптор региона в принятом в GDI формате. В данном случае мы получим дескриптор региона, выбранного в обозначенном параметром `Graphics` холсте GDI+.

Сравнение регионов

Необходимой преамбулой к истории о сравнении регионов должна стать классификация регионов по описываемому ими множеству точек. Ведь как в GDI, так и в GDI+ сравнение регионов осуществляется по взаимному соответствию между множеством точек, содержащихся в сопоставляемых объектах. Выбрав в качестве базового этот критерий, можно говорить о трех возможных состояниях региона:

1. Регион бесконечен (включает все точки координатного пространства).
2. Регион пуст.
3. Регион содержит некоторое подмножество точек.

Говоря о «бесконечном» регионе, нельзя обойтись без кавычек, мировые координаты отнюдь не бескрайни. Они ограничены пределами двухмерной матрицы со сторонами по 2^{32} единиц. В целом это достаточно много, но далеко не бесконечно. Итак, самый большой регион можно получить двумя способами. Во-первых, воспользовавшись конструктором по умолчанию (именно он открывал материал этой главы). Во-вторых, обратившись к методу:

```
function MakeInfinite: TStatus;
```

Для получения пустого региона также допустимо обратиться к конструктору, например передав в него прямоугольную область с нулевыми размерами сторон.

```
GPRegion :=TGPRegion.Create(MakeRect(0,0,0,0));
```

Второй вариант опирается на применение метода:

```
function MakeEmpty: TStatus;
```

Функция без малейших колебаний в два счета опустошит сколь угодно большой регион.

Наконец, мы подобрались к операции сравнения регионов. В GDI+ все операции сравнения регионов имеют смысл только при учете такого важного атрибута, как холст, поэтому `Graphics` – обязательный параметр рассматриваемых методов. Сравнение регионов, содержащих некоторое подмножество точек, осуществляется за счет метода:

```
function Equals(Region: TGPRegion; Graphics: TGPGraphics): BOOL;
```

Если подмножество точек полностью совпало, метод возвращает `true`.

Что касается бесконечного региона и его антипода – пустого региона, для выяснения кто есть кто проще всего обратиться к методам:

```
function IsInfinite(Graphics: TGPGraphics): BOOL;//бесконечный регион
function IsEmpty(Graphics: TGPGraphics): BOOL;   //пустой регион
```

В случае, если регион действительно содержит все точки координатного пространства, функция `IsInfinite()` возвратит значение `true`. Решение, что регион пуст, на точно таких же условиях уполномочен принимать метод `IsEmpty`.

Комбинирование регионов

В GDI+ различают пять вариантов комбинирования двух регионов – это объединение, пересечение, исключающее ИЛИ, вычитание и дополнение. Графическую интерпретацию всех перечисленных комбинаций вы найдете на рис. 32.1.

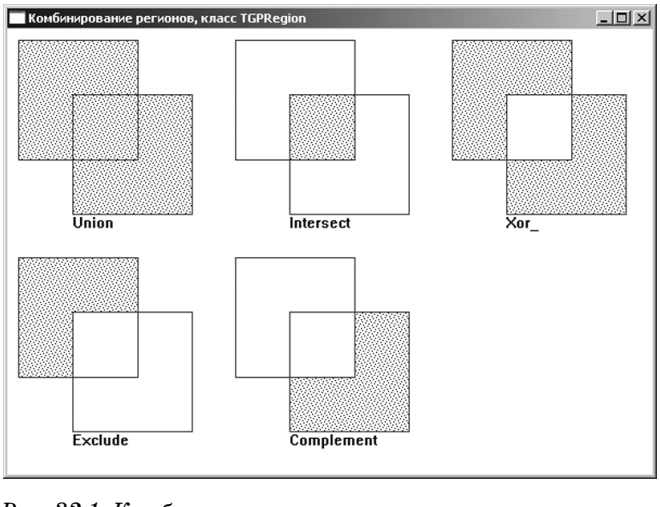


Рис. 32.1. Комбинирование регионов

Наибольшей популярностью пользуется операция объединения двух регионов. Она реализуется при посредничестве квартета перегружаемых функций Union(). Среди этой группы доминирующим является метод:

```
function Union(Region: TGPRegion): TStatus; overload;
```

В результате мы получаем новый регион, в который входят все точки нашего региона и региона, переданного через параметр Region. Оставшаяся тройка методов является производной от первого:

```
function Union(const Rect: TGPRect): TStatus; overload;
function Union(const Rect: TGPRectF): TStatus; overload;
function Union(Path: TGPGraphicsPath): TStatus; overload;
```

К множеству точек нового региона добавляются все точки из прямоугольной области Rect либо из траектории Path.

В результате пересечения регионов в результирующее множество точек включаются лишь точки, одновременно принадлежащие обоим регионам. Операция пересечения множеств реализуется набором функций Intersect(). Головной метод:

```
function Intersect(Region: TGPRegion): TStatus; overload;
```

в качестве параметра рассчитывает увидеть другой экземпляр региона. Остальные методы перечислять не станем, заметим лишь то, что они вновь работают с прямоугольником Rect или траекторией Path.

В группе методов, осуществляющих операцию «исключающее ИЛИ», также имеются четыре функции, вот одна из них:

```
function Xor(Region: TGPRegion): TStatus; overload;
```

Функция сделает так, что в итоговом регионе окажутся только не совместные точки двух регионов.

Вычитанием данных из региона ведает метод:

```
function Exclude(Region: TGPRegion): TStatus; overload;
```

В результате из состава региона, вызвавшего метод Exclude(), удаляются все точки, входящие в регион Region (прямоугольник Rect или траекторию Path). Возможно и обратное действие:

```
function Complement(Region: TGPRegion): TStatus; overload;
```

После обращения к методу в регионе окажутся только ранее не входившие в него точки. Функция Complement() также имеет четыре версии, отличающихся лишь способом определения региона.

Все рассмотренные методы комбинирования регионов опираются на функцию Windows API – CombineRgn(). Об этой функции подробно рассказано в главе 9 «Регионы».

Наших знаний уже вполне достаточно, чтобы вновь поиграть с регионом. Предлагаю создать очередную вредную процедуру, «выгрызающую» прямоугольную дыру в клиентской области окна.

```
Procedure GDIPLUS_PAINT(DC:HDC;Rect:TRect);
var GPGraphics:TGPGraphics;
    GPRegion : TGPRegion;
    GPBrush:TGPHatchBrush;
    GPRect:TGPRect;
begin
   GPGraphics:=TGPGraphics.Create(DC);
   //описываем прямоугольник размерами меньше клиентской области
   GPRect:=MakeRect(Rect.Left+40,Rect.Top+40,Rect.Right-80,Rect.Bottom-80);

   GPRegion :=TGPRegion.Create;  //исходный бесконечный регион
   GPRegion.Xor_(GPRect);  //вычитаем совместные с прямоугольной областью точки

   //вспоминаем функцию Windows API SetWindowRgn()
   //для создания "дырки" в клиентской области окна
   SetWindowRgn(Wnd,GPRegion.GetHRGN(GPGraphics),true);
   //зальем узорной кистью оставшуюся часть поверхности
   GPBrush:=TGPHatchBrush.Create(HatchStylePlaid,aclBlue,aclWhite);
   GPGraphics.FillRegion(GPBrush,GPRegion);
   GPGraphics.Free; GPBrush.Free;GPRegion.Free;
end;
```

Правда, и здесь нам не обойтись без помощи Windows API, поддержку в проделывании дыры нам оказала функция `SetWindowRgn()`, установившая нестандартный режим отсечения. В результате сквозь окно можно не только увидеть, но и «потрогать» курсором иконки наших программ из библиотеки к книге на сайте издательства в окне проводника (рис. 32.2).

В конце главы мы вновь вернемся к механизму отсечения в GDI+, а пока поговорим еще об одной интересной задаче – проверке принадлежности к региону.

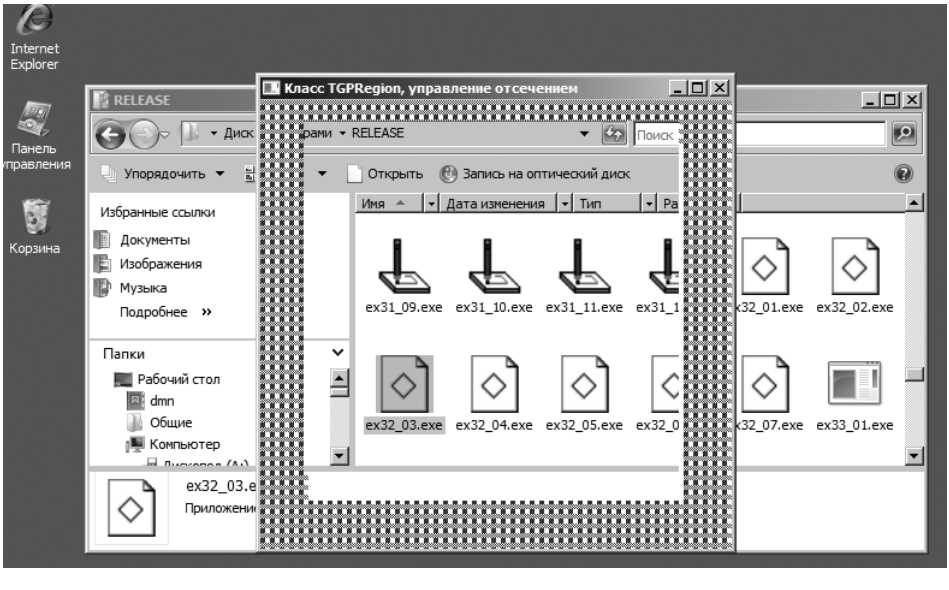


Рис. 32.2. Совместные действия региона GDI+ и функции SetWindowRgn( )

Проверка принадлежности к региону

Вопрос проверки принадлежности к региону точки или прямоугольной области может встретиться во многих графических задачах. Самый наглядный пример – игровой. Допустим, регионом описывается фигура противника в очередной игре-«стрелялке», игрок палит по региону, не жалея кнопок клавиатуры, а на нашу долю остается отвечать на вопрос, попадают ли в мишень электронные залпы.

Начнем с самого простого. Ограничивающий регион прямоугольник может быть получен с помощью любого из представленных ниже методов:

```
function GetBounds(out Rect: TGPRect; G: TGPGraphics): TStatus; overload;
function GetBounds(out Rect: TGPRectF; G: TGPGraphics): TStatus; overload;
```

Получив в руки столь мощное оружие, сразу хочется выяснить размеры прямоугольника, охватывающего бесконечный регион. Можете проверить, можете поверить на слово, но для бесконечного региона функция возвращает следующие значения: Rect=(X=-4194304, Y=-4194304, Width=8388608, Height=8388608). Это всего лишь матрица размером немногим более $2^{23}\times2^{23}$, для бесконечного региона маловато будет…

Факт видимости для пользователя точки из состава региона может подтвердить или опровергнуть перегружаемый метод:

```
function IsVisible(X, Y: Integer; G: TGPGraphics = nil): BOOL; overload;
function IsVisible(X, Y: Single; G: TGPGraphics = nil): BOOL; overload;
function IsVisible(const Point: TGPPoint; G: TGPGraphics = nil): BOOL; overload;
function IsVisible(const Point: TGPPointF; G: TGPGraphics = nil): BOOL; overload;
```

Координаты исследуемой точки передаются в параметры (`X`,`Y`) или `Point`. Для того чтобы метод смог отреагировать на настройки региона отсечения, в его завершающий параметр необходимо направить ссылку на холст GDI+. Если точка входит во множество видимых точек региона – метод возвратит значение `true`. Программисту стоит оказывать особое внимание к точкам, попадающим на границы региона. Следующий пример показывает особенность определения принадлежности точки к простейшему прямоугольному региону:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics; GPRegion : TGPRegion;
    GPPen:TGPPen; GPBrush:TGPSolidBrush; I:Integer; S:String;
const Points:Array[0..6] of TGPPoint=
      ((X:5;Y:5),(X:150;Y:50),(X:10;Y:25),(X:55;Y:10),
       (X:310;Y:60),(X:180;Y:110),(X:320;Y:120));
begin
    GPGraphics:=TGPGraphics.Create(DC);
    GPBrush:=TGPSolidBrush.Create(aclLightGreen);
    GPPen:=TGPPen.Create(aclBlack,3);
    GPRegion :=TGPRegion.Create(MakeRect(10,10,300,100));
    GPGraphics.FillRegion(GPBrush,GPRegion);

   for i:=0 to 6 do
   begin
```

```
        if GPRegion.IsVisible(Points[i],GPGraphics) then
        begin //точка принадлежит региону
         GPPen.SetColor(aclRed);
         GPGraphics.DrawEllipse(GPPen,Points[i].X-2,Points[i].Y-2,4,4);
        end
        else
        begin //точка не принадлежит региону
         GPPen.SetColor(aclBlue);
         GPGraphics.DrawRectangle(GPPen,Points[i].X-2,Points[i].Y-2,4,4);
        end;
       S:=Format('(%d,%d)',[Points[i].X,Points[i].Y]);
       TextOut(DC,Points[i].X,Points[i].Y,PChar(s),Length(s));
     end;
   GPGraphics.Free; GPBrush.Free; GPRegion.Free; GPPen.Free;
end;
```

Для заливки региона кистью следует воспользоваться функцией `FillRegion()`*, это метод холста GDI+.*

В ходе выполнения кода нами создается регион из прямоугольника с координатами `(X=10,Y=10,Width=300,Height=100)` и осуществляется проверка принадлежности точек из массива `Points` к этому региону. Если точка входит в регион, то на ее месте рисуем эллипс, если нет – квадрат. Заметьте, что находящиеся на нижней и правой границах региона точки (180,110) и (310,60) ему не принадлежат (рис. 32.3).

Обсуждение методов, проверяющих вхождение в регион прямоугольника, начнем с небольшого лирического отступления. Стоит отдать должное разработчикам GDI – у них по части выдумки имен функций было все в порядке. В разделе контроля принадлежности региону точки и прямоугольника проверки осуществлялись функциями Windows API с названиями, понятными не только программисту, но и обычному смертному, имеющему некоторые представления об английском языке: `PtInRegion()` и `RectInRegion()`. В GDI+ для проверки принадлежности региону прямоугольника вновь применяется метод с названием `IsVisible()`. Не хватайтесь за таблетки, это не дежавю! GDI+ славится обилием одноименных перегружаемых методов, и подобный

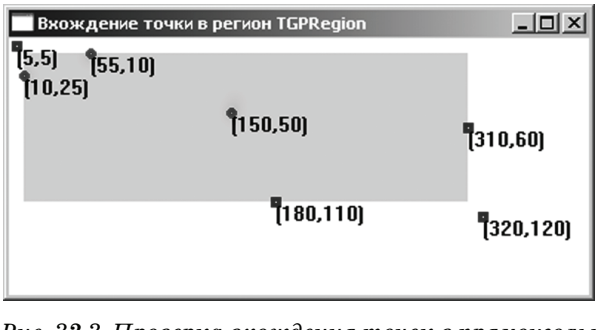


Рис. 32.3. Проверка вхождения точек в прямоугольный регион

подход нужно только приветствовать, когда методы решают идентичную задачу. Но в последнем случае остается только пожимать плечами. Поэтому, когда лет через десять появится какой-нибудь «GDI++», не удивлюсь, если вообще все методы будут называться одинаково... Но вернемся к нашим старым-новым знакомым:

```
function IsVisible(X, Y, Width, Height: Single;
                   G: TGPGraphics = nil): BOOL; overload;
function IsVisible(X, Y, Width, Height: Integer;
                   G: TGPGraphics): BOOL; overload;
function IsVisible(const Rect: TGPRect; G: TGPGraphics = nil): BOOL; overload;
function IsVisible(const Rect: TGPRectF; G: TGPGraphics = nil): BOOL; overload;
```

Координаты контролируемого прямоугольника описываются параметрами `X`, `Y`, `Width`, `Height` либо `Rect`. Все остальное осталось по-прежнему, если хотя бы одна-единственная точка прямоугольника попала в область траектории – метод возвращает `true`.

Преобразования региона

Для обычного перемещения региона в двухмерном координатном пространстве применяются методы:

```
function Translate(dX, dY: Single): TStatus; overload;
function Translate(dX, dY: Integer): TStatus; overload;
```

Параметры `dX` и `dY` задают расстояние, на которое смещаются все точки, входящие в состав региона.

Все более сложные задачи, связанные с деформацией, масштабированием, отображением, поворотом региона, решаются с помощью метода:

```
function Transform(Matrix: TGPMatrix): TStatus;
```

Единственный параметр функции представляет собой матрицу двухмерных аффинных преобразований, работа с которой подробно рассмотрена в главе 34.

Исследование региона

Корпорация Microsoft не документирует, каким именно образом осуществляется хранение данных о множестве точек, входящих в регион. Надо предполагать, что это ни в коем случае не массив, содержащий все точки из множества. Такое решение противоречит здравому смыслу, попробуйте перемножить матрицу с размерами $2^{32} \times 2^{32}$ на 8 байт (размер структур `TGPPoint` и `TGPPointF`) и сразу все поймете.

Дабы предотвратить бунты среди недовольных программистов, активно применяющих регионы в своих программах, разработчики GDI (а затем и GDI+) пошли на компромиссное решение и предложили альтернативный способ представления данных о регионах. В GDI+ таких решений два:

1. Мы в состоянии получить описание региона в виде указателя на массив байт. Что конкретно хранится в этом массиве, сказать трудно, но это со-

держимое позднее может быть использовано совместно с одним из конструкторов региона.

2. Мы имеем возможность аппроксимировать регион набором прямоугольных областей.

Работа в первом направлении как всегда начинается с выяснения размера массива, в который позднее передаются данные о регионе.

```
function GetDataSize: UINT;
```

Выяснив размер, переходим к основной функции:

```
function GetData(pBuf: pBYTE; BufSize: UINT; pSizeFilled: pUINT = nil): TStatus;
```

Здесь `pBuf` – указатель на буфер, в который метод поместит данные; `BufSize` – размер массива, возвращенный методом `GetDataSize()`; `pSizeFilled` – необязательный параметр, позволяющий управлять размером порций данных, передаваемых в буфер.

```
Procedure GPRegionDetermination(GPRegion : TGPRegion);
var BufSize:Word;
    pBuf:pByte;
begin
   BufSize:=GPRegion.GetDataSize; //узнаем размер
   GetMem(pBuf,BufSize);          //распределение памяти
   GPRegion.GetData(pBuf,BufSize);//получаем данные
//… обработка данных …
   FreeMem(pBuf,BufSize);         //очистка памяти
end;
```

Теперь более подробно обсудим второй способ исследования региона. Регион аппроксимируется набором прямоугольных областей. Для этого потребуется помощь двух методов. Первоначально необходимо уяснить, какое количество прямоугольных областей потребуется для описания региона. Для этого предназначен метод:

```
function GetRegionScansCount(Matrix: TGPMatrix): UINT;
```

Наличие матрицы преобразований `Matrix` позволяет сразу применять к набору прямоугольников двумерные аффинные преобразования, например, смещать их по координатным осям, поворачивать, деформировать и т. п. После получения информации о числе областей нам остается распределить память для массива, в который помещаются результаты операции, и вызвать метод:

```
function GetRegionScans(Matrix: TGPMatrix; pRects: pGPRectF;
                        out Count: Integer): TStatus; overload;
function GetRegionScans(Matrix: TGPMatrix; pRects: pGPRect;
                        out Count: Integer): TStatus; overload;
```

Кроме матрицы преобразований `Matrix` и указателя на приемный массив `pRects` в метод требуется передать переменную, в которую вновь будет возвращено число прямоугольных областей.

Ниже предложена процедура, демонстрирующая порядок работы с рассмотренными методами.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;  GPPen:TGPPen;
    GPGraphicPath:TGPGraphicsPath;
    GPRegion : TGPRegion;
    GPBrush:TGPSolidBrush;
    Count, i :Integer;
    Rects:Array of TGPRect;
    GPMatrix:TGPMatrix;
begin
    GPGraphics:=TGPGraphics.Create(DC);
{создаем траекторию из эллипса и прямоугольника}
    GPGraphicPath:=TGPGraphicsPath.Create(FillModeWinding);
    GPGraphicPath.AddEllipse(MakeRect(60,20,100,100));
    GPGraphicPath.AddRectangle(MakeRect(10,50,200,40));
{создаем регион из траектории}
    GPRegion :=TGPRegion.Create(GPGraphicPath);
    GPGraphicPath.Free;
{вывод региона}
    GPBrush:=TGPSolidBrush.Create(aclGray);
    GPGraphics.FillRegion(GPBrush,GPRegion);
    GPBrush.Free;
{матрица аффинных преобразований}
    GPMatrix:=TGPMatrix.Create;
    GPMatrix.Translate(250,0); //смещаем результат вправо
{узнаем число прямоугольников}
    Count:=GPRegion.GetRegionScansCount(GPMatrix);
    SetLength(Rects,Count); //распределяем память
{получаем данные}
    GPRegion.GetRegionScans(GPMatrix,pGPRect(Rects),Count);
{вывод прямоугольников
    GPPen:=TGPPen.Create(aclBlack);
    for i:=0 to Count-1 do GPGraphics.DrawRectangle(GPPen,Rects[i]);
    GPPen.Free; GPGraphics.Free; GPRegion.Free; GPMatrix.Free;
end;
```

Работа примера начинается с создания траектории из перекрывающихся фигур эллипса и прямоугольника, затем из этой траектории формируется регион. Для наглядности полученный регион выводится в левой части окна (рис. 32.4). Создав матрицу аффинных преобразований, мы указываем ей,

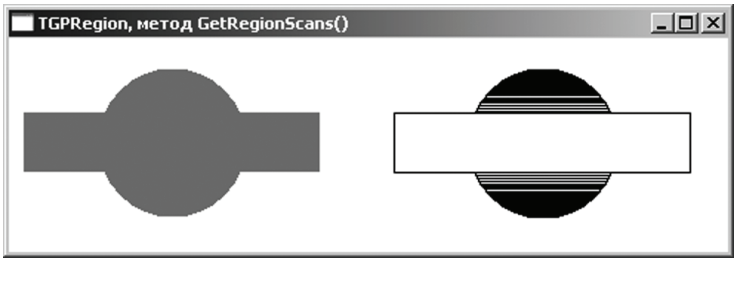


Рис. 32.4. Аппроксимация региона прямоугольниками

что намерены сдвинуть полученные в результате аппроксимации прямоугольники вправо на 250 единиц. Дело осталось за малым – узнаем число прямоугольников, выделяем память для массива Rects и заполняем его с помощью метода GetRegionScans(). В правой части окна (см. рис. 32.4) представлено изображение, полученное из аппроксимирующих регион прямоугольников (в данном случае их потребовалось более 50).

Холст GDI+ и регион отсечения

Технология отсечения играет едва ли не самую главную роль в построении многооконного интерфейса Windows. Благодаря отсечению механизм графического вывода сосредоточит свои усилия в конкретной области графического устройства, не тратя драгоценное время на обслуживание невидимых участков. Однако по сравнению с GDI у GDI+ даже несколько сокращены полномочия в части, касающейся механизма отсечения. Вместо широкого спектра функций Windows API, имеющихся в распоряжении GDI (вспоминайте функции SelectClipRgn, SetMetaRgn, ExtSelectClipRgn), разработчику программного обеспечения выделен метод SetClip (принадлежащий холсту GDI+ класс TGPGraphics) и ряд его производных. Впрочем, в этом нет ничего удивительного, Microsoft все более и более старательно отодвигает программиста от Windows API и не только при работе с графикой (это особенно сильно бросается в глаза в технологии .NET). Хочется надеяться, что причина не в том, что корпорация пытается окутать операционную систему тайной, а в благородном стремлении оградить начинающего программиста от вероятных ошибок.

В классе TGPGraphics предусмотрено шесть версий перегружаемого метода SetClip(), предназначенного для установки региона отсечения. Основной из них:

```
function SetClip(hRgn: HRGN;
         CombineMode: TCombineMode = CombineModeReplace): TStatus; overload;
```

Указанная функция ближе всех к Windows GDI, так как манипулирует понятным для GDI дескриптором hRgn региона отсечения. Предусмотрено несколько вариантов поведения метода, они определяются вторым параметром функции – CombineMode. В зависимости от выбранной константы (табл. 32.1) получается та или иная комбинация из точек установленного на момент обращения к функции региона отсечения и добавляемого региона hRgn.

Таблица 32.1. Варианты комбинирования регионов TCombineMode

Константа	Значение	Результат работы метода
CombineModeReplace	0	Результирующий регион будет полностью повторять переданный в функцию регион.
CombineModeIntersect	1	Новый регион создается только из совместных точек обоих регионов.
CombineModeUnion	2	В новый регион войдут все точки из обоих участвующих в операции регионов.

Таблица 32.1 (продолжение)

Константа	Значение	Результат работы метода
CombineModeXor	3	Результирующий регион будет включать все точки из участвующих в операции регионов, за исключением совместных точек.
CombineModeExclude	4	Итоговый регион получится путем вычитания точек, принадлежащих добавляемому региону, из текущего региона.
CombineModeComplement	5	В новый регион войдут все точки из добавляемого региона за исключением точек, принадлежащих исходному региону.

Здесь нет ничего принципиально нового. Механизм формирования региона отсечения средствами холста GDI+ перекликается с возможностями класса TGPRegion по описанию обычного региона, с которыми мы немного поэкспериментировали в первой половине этой главы (см. рис. 32.1).

Оставшиеся пять вариантов метода SetClip() отличаются лишь способом определения региона в формате GDI+:

```
function SetClip(Region: TGPRegion;
         CombineMode: TCombineMode = CombineModeReplace): TStatus; overload;
function SetClip(Path: TGPGraphicsPath;
         CombineMode: TCombineMode = CombineModeReplace): TStatus; overload;
function SetClip(Rect: TGPRect;
         CombineMode: TCombineMode = CombineModeReplace): TStatus; overload;
function SetClip(Rect: TGPRectF;
         CombineMode: TCombineMode = CombineModeReplace): TStatus; overload;
function SetClip(Graphics: TGPGraphics;
         CombineMode: TCombineMode = CombineModeReplace): TStatus; overload;
```

Как видите, это регион в стиле GDI+ Region, траектория Path, прямоугольная область в формате целых Rect и вещественных чисел RectF и, собственно, все множество точек холста Graphics.

На данный момент наших знаний о регионах и траекториях вполне достаточно, чтобы добиться весьма примечательных результатов. В следующем примере мы создаем регион отсечения из траектории, которая в свою очередь строится из текстовой строки.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    Path:TGPGraphicsPath;
    FontFamily:TGPFontFamily;
    Brush:TGPHatchBrush;
    Pen:TGPPen;
const S='Hello, World!';
begin
  GPGraphics:=TGPGraphics.Create(DC);
  FontFamily:=TGPFontFamily.Create('Forte');
```

```
  Path:=TGPGraphicsPath.Create(); //создаем траекторию
{текст траеткории}
  Path.AddString(S,-1,FontFamily,
        FontStyleRegular OR FontStyleBold,100,MakePoint(10.0,10),nil);
  FontFamily.Free;
{формируем регион отсечения}
  GPGraphics.SetClip(path);
  Path.Free;
{заливаем регион отсечения кистью}
  Brush:=TGPHatchBrush.Create(HatchStyleZigZag,aclRed,aclYellow);
  GPGraphics.FillRectangle(Brush,MakeRect(0,0,10000,10000));
  Brush.Free;
{проводим по холсту линию}
  Pen:=TGPPen.Create(MakeColor(127,0,0,255),20);
  GPGraphics.DrawLine(Pen, 0,25,3000,30);
  Pen.free;
  GPGraphics.Free;
end;
```

На рис. 32.5 представлены результаты наших трудов. Обратите внимание на то, что обращение к методу `DrawLine()` визуально привело не к выводу линии из точки (0,25) в точку (3000,30), а лишь к появлению декоративной диагональной синей полосы на текстовой строке. Это объясняется формой региона отсечения, в которую вошли лишь контуры символов текста, а за пределами региона отсечения все попытки рисования были в корне пресечены операционной системой.

Большинство остальных связанных с регионом отсечения методов класса `TGPGraphics` являются производными от `SetClip()`. Перегружаемых версий как всегда несколько, мы упомянем только их названия. Метод `IntersectClip()` построит регион отсечения на основе пересечения текущего региона и прямоугольника `Rect` или региона `Region` (в зависимости от выбранной версии функции). Метод `ExcludeClip()` исключит из региона отсечения прямоугольную область или регион.

Вдоволь поразвлекавшись с регионом отсечения, не забудем привести его в исходное состояние. Сброс региона отсечения производится методом холста GDI+:

```
function ResetClip: TStatus;
```

В итоге у холста восстанавливаются стандартные размеры региона отсечения. Кто угадает, что подразумевается под «стандартным размером»? Прав окажется тот читатель, который ответит, что это бесконечный регион, точ-

Рис. 32.5. Построение региона отсечения из траектории

нее, регион, в состав которого входят все точки используемого в данный момент координатного пространства.

Если возникает необходимость исследовать системный регион, то его описание можно скопировать, для этого предназначен метод:

```
function GetClip(Region: TGPRegion): TStatus;
```

Результаты возвращаются через параметр `Region` в формате класса `TGPRegion`.

Кроме того, при анализе региона отсечения можно выяснить координаты прямоугольной области, ограничивающей этот регион. С такой задачей с успехом справятся функции:

```
function GetClipBounds(out Rect: TGPRectF): TStatus; overload;
function GetClipBounds(out Rect: TGPRect): TStatus; overload;
```

Если же размеры региона не нужны, а лишь хочется узнать, определен ли он, то следует обратиться к методу:

```
function IsClipEmpty: Bool;
```

Если в состав региона отсечения не входит ни одна точка, а такая ситуация вполне вероятна, например, после выполнения строки кода `GPGraphics.SetClip(MakeRect(0,0,0,0))`, то функция вернет значение `true`. Кроме того, в распоряжении холста имеется еще более грамотный метод:

```
function IsVisibleClipEmpty: BOOL;
```

Он возвратит `true`, даже если регион отсечения определен, но ни одна из его точек не находится в видимой части окна.

Для получения прямоугольных координат видимой части региона отсечения холста (это пересечение региона отсечения окна и региона отсечения холста GDI+) следует воспользоваться методами:

```
function GetVisibleClipBounds(out Rect: TGPRectF): TStatus; overload;
function GetVisibleClipBounds(out Rect: TGPRect): TStatus; overload;
```

Результат окажется в параметре `Rect`.

Для сдвига существующего региона по горизонтальной и вертикальной осям следует вызвать любой из следующих методов:

```
function TranslateClip(dX, dY: Single): TStatus; overload;
function TranslateClip(dX, dY: Integer): TStatus; overload;
```

РЕЗЮМЕ

В GDI+ работа с регионами осуществляется при поддержке классов `TGPRegion` и `TGPGraphics`. Первый из них представляет собой объектно-ориентированное описание региона, а второй класс предназначен для обслуживания ключевого региона Windows – региона отсечения. К сожалению, возможности GDI+ по управлению регионами несколько сокращены, например, отсутствует возможность работы с метарегионом.

33

Графические примитивы и заливка областей в GDI+

В этой главе мы вновь возвращаемся к изучению возможностей холста GDI+. На этот раз предметом беседы станут методы класса `TGPGraphics`, нацеленные на рисование на поверхности контекста устройства простейших графических примитивов (линий, многоугольников, кривых и других элементарных фигур). Кроме того, мы рассмотрим способы заливки замкнутых областей.

Во многом перечень предназначенных для вывода графических примитивов методов класса `TGPGraphics` определется набором функций GDI. Однако есть и исключения, точнее, усовершенствования. В первую очередь – это появление принципиально нового способа работы со сплайнами Безье. Теперь отклоняющие точки явным образом определяют траекторию сплайна, находясь прямо на кривой.

Сразу отметим общие черты большей части рассматриваемых в главе методов. Так как каждый метод GDI+ должен быть самодостаточен, то кроме координат будущих отрезков, кривых и других фигур в состав параметров всегда входит описание пера или кисти. Поэтому если речь ведется о черчении линий, то первым параметром всех функций выступает перо `Pen: TGPPen`. Если же мы планируем закрасить замкнутую область, то первым параметром метода всегда будет кисть `Brush: TGPBrush`.

Прежде чем мы перейдем к изучению материала этой главы, позволю себе небольшое лирическое отступление. Иногда при проектировании приложений с применением GDI+ в голову невольно закрадывается крамольная мысль, что над созданием библиотеки трудилась весьма необычная группа разработчиков. Программисты никуда не спешили и даже немножко скучали. Когда подходил ответственный момент очередного отчета о проделанной работе, эта группа специалистов (для того чтобы убедить руководителя проекта в своей исключительной работоспособности) по четыре раза «создавала» один и тот же метод, наделяя его разными по типу, но абсолютно одинаковыми по смысловой нагрузке параметрами. Все остальное время эти ребя-

та тихонько дремали, раскладывали пасьянс и вполголоса сплетничали... Шутки шутками, но большинство объектов GDI+ в буквальном смысле слова перегружены одинаковыми методами. Судите сами...

Прямые и ломаные линии

Практически все классы GDI+ славятся большим количеством перегружаемых методов. В тройке лидеров находится холст GDI+. У него даже один из самых простейших методов, позволяющий провести линию из точки А в точку Б, и тот предложен в четырех вариантах:

```
function DrawLine(Pen: TGPPen; x1, y1, x2, y2: Integer): TStatus; overload;
function DrawLine(Pen: TGPPen; x1, y1, x2, y2: Single): TStatus; overload;
function DrawLine(Pen: TGPPen; const pt1, pt2: TGPPointF): TStatus; overload;
function DrawLine(Pen: TGPPen; const pt1, pt2: TGPPoint): TStatus; overload;
```

Для работы всем функциям необходимо передать перо `Pen` и координаты точек (форматом представления которых и различается каждый из членов перегружаемой четверки). Как видите, метод сразу устанавливает начальную и конечную точки прямой, за один раз объединяя обращения к функциям `MoveToEx()` и `LineTo()` из состава Windows API. Надо отметить, что в составе всех опубликованных методов `TGPGraphics` в принципе отсутствует выделенная функция, отвечающая за установку пера в заданных координатах поверхности графического устройства. Это объясняется тем, что любой из рисующих графические примитивы методов инкапсулирует обращение к позиционирующей функции `MoveToEx()`.

Метод `DrawLine()` *является наглядной демонстрацией системы программирования без информации о состоянии. В метод передаются все данные, требуемые для выполнения логически завершенной операции: перо и координаты начальной и конечной точки.*

Для вывода ломаной линии вместо многократного обращения к `DrawLine()` целесообразно выбрать функцию:

```
function DrawLines(Pen: TGPPen; pPoints: PGPPoint;
                   Count: Integer): TStatus; overload;
function DrawLines(Pen: TGPPen; pPoints: PGPPointF;
                   Count: Integer): TStatus; overload;
```

На этот раз координаты точек содержатся в массиве, на который ссылается указатель `pPoints`. Последний параметр функции `Count` определяет число подлежащих выводу точек.

```
Procedure GDI_PAINT(DC:HDC;ClientRect:TRect);
var   Graphics:TGPGraphics;
      Points: array[0..9] of TGPPoint;
      Pen:TGPPen;
      i : byte;
begin
Randomize; //инициализация генератора псевдослучайной последовательности
```

```
  for i:=0 to 9 do {заполняем массив случайными значениями}
  begin
   Points[i].X:=Random(ClientRect.Right);
   Points[i].Y:=Random(ClientRect.Bottom);
  end;

Graphics:=TGPGraphics.Create(DC);
Pen:=TGPPen.Create(aclBlack);
Graphics.DrawLines(Pen,pGPPoint(@Points),10);
Pen.Free; Graphics.free;
end;
```

Точно по такому же алгоритму осуществляется вывод многоугольника, только теперь вместо метода `DrawLines()` нам потребуется целочисленный или вещественный вариант функции:

```
function DrawPolygon(Pen: TGPPen; pPoints: PGPPoint;
                     Count: Integer): TStatus; overload;
function DrawPolygon(Pen: TGPPen; pPoints: PGPPointF;
                     Count: Integer): TStatus; overload;
```

Для того чтобы ломаная линия превратилась в многоугольник, метод соединит первую и последнюю точки из массива.

Многоугольник (как, впрочем, и любая другая замкнутая фигура) может быть закрашен. Для этого предназначена группа методов, название которых начинается со слова «Fill». В случае многоугольника следует применять методы:

```
function FillPolygon(Brush: TGPBrush; pPoints: PGPPoint;
                     Count: Integer): TStatus; overload;
function FillPolygon(Brush: TGPBrush; pPoints: PGPPointF;
                     Count: Integer): TStatus; overload;
function FillPolygon(Brush: TGPBrush; pPoints: PGPPoint;
                     Count: Integer; FillMode: TFillMode): TStatus; overload;
function FillPolygon(Brush: TGPBrush; pPoints: PGPPointF;
                     Count: Integer; FillMode: TFillMode): TStatus; overload;
```

Естественно, что в операции заливки главное действующее лицо – кисть GDI+. Методы отличаются лишь способом определения вершин фигуры. Кроме того, последняя пара функций предоставляет возможность управлять режимом заливки, для этого предназначен аргумент `FillMode`. Параметр может принимать одно из двух значений – `FillModeAlternate` или `FillModeWinding`. В зависимости от установленного значения будет выбран порядок принятия решения, о том принадлежит ли точка фигуре, иначе говоря – стоит закрашивать точку или нет. Алгоритм нам не нов, достаточно вспомнить функцию GDI `SetPolyFillMode` (за подробностями возвращайтесь к табл. 7.1 главы «Простейшие геометрические фигуры и заливка областей»). На рис. 33.1 продемонстрировано влияние режима `TFillMode` на особенности заливки внутренних областей многоугольника сложной формы.

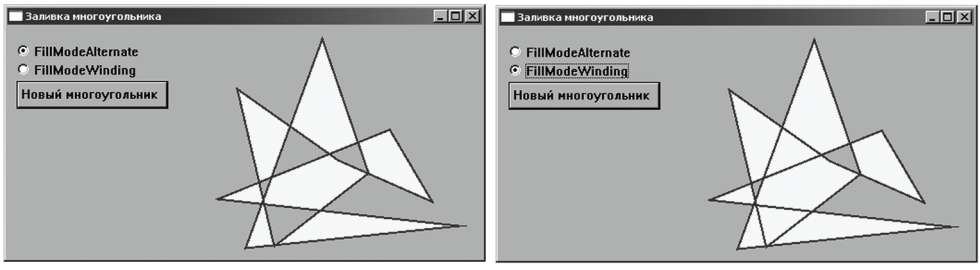


Рис. 33.1. Влияние режима TFillMode на закраску внутренних областей многоугольника сложной формы

Кривые

В GDI+ имеется две разновидности кривых линий. Это унаследованная со времен безраздельного царствования GDI кривая Безье и усовершенствованная кривая GDI+, в некоторой литературе называемая *каноническим сплайном* [5]. Для вывода сплайна в стиле GDI допустимо воспользоваться любым из предложенных ниже методов:

```
function DrawBezier(Pen: TGPPen;
                    x1, y1, x2, y2, x3, y3, x4, y4: Integer): TStatus; overload;
function DrawBezier(Pen: TGPPen;
                    x1, y1, x2, y2, x3, y3, x4, y4: Single): TStatus; overload;
function DrawBezier(Pen: TGPPen;
                    const pt1, pt2, pt3, pt4: TGPPoint): TStatus; overload;
function DrawBezier(Pen: TGPPen;
                    const pt1, pt2, pt3, pt4: TGPPointF): TStatus; overload;
```

Метод `DrawBezier()` является аналогом функции Windows API `PolyBezier()`. Здесь первая и четвертая точка определяют начало и окончание линии, вторая и третья точки отклоняют линию от оси, превращая ее в сплайн.

При одновременном выводе нескольких сплайнов применяются методы:

```
function DrawBeziers(Pen: TGPPen; pPoints: PGPPoint;
                     Count: Integer): TStatus; overload;
function DrawBeziers(Pen: TGPPen; pPoints: PGPPointF;
                     Count: Integer): TStatus; overload;
```

Описания сплайнов загружаются в массив точек, на которые ссылается указатель `pPoints`. Заключительный параметр определяет число точек в подлежащем выводу массиве.

Построенные таким образом кривые обладают всеми характерными чертами старого доброго сплайна: кривая размещается внутри выпуклой фигуры с вершинами в определяющих точках кривой, и прямая, проведенная из конечной в ближайшую отклоняющую точку, всегда представляет собой касательную к сплайну.

В GDI+ предусмотрен альтернативный механизм прорисовки сплайна Безье, здесь принципиально изменилась роль отклоняющих точек. В методах:

```
function DrawCurve(Pen: TGPPen; pPoints: PGPPoint;
                   Count: Integer): TStatus; overload;
function DrawCurve(Pen: TGPPen; pPoints: PGPPointF;
                   Count: Integer): TStatus; overload;
```

точки 2 и 3 не просто отводят кривую от прямолинейной траектории, а явным образом определяют траекторию кривой. Простейшие версии перегружаемого метода требуют от программиста трех параметров: пера `Pen`, указателя на массив точек кривой `pPoints` и числа точек, подлежащих выводу, – `Count`. Дальше становится интереснее – на свет появляется параметр `Tension`, влияющий на степень натяжения кривой:

```
function DrawCurve(Pen: TGPPen; pPoints: PGPPoint;
                   Count: Integer; Tension: Single): TStatus; overload;
function DrawCurve(Pen: TGPPen; pPoints: PGPPointF;
                   Count: Integer; Tension: Single): TStatus; overload;
```

Благодаря параметру `Tension` поведение канонического сплайна GDI+ можно сравнить с поведением туго натянутой струны. По умолчанию струна не напряжена (`Tension=0.5`). При нулевом натяжении кривая превращается в обычную ломаную, а при увеличении степени натяжения можно добиться весьма забавных визуальных эффектов (но порвать струну у нас не получится).

В апогее сложности метод потребует еще два параметра. Это отступ `Offset`, позволяющий начать вывод кривой с любой точки массива, и `NumberOfSegments`, указывающий количество сегментов в сплайне.

```
function DrawCurve(Pen: TGPPen; pPoints: PGPPoint;
                   Count, Offset, NumberOfSegments: Integer;
                   Tension: Single = 0.5): TStatus; overload;
function DrawCurve(Pen: TGPPen; pPoints: PGPPointF;
                   Count, Offset, NumberOfSegments: Integer;
                   Tension: Single = 0.5): TStatus; overload;
```

Если вы читаете эти строки за компьютером, то настоятельно рекомендую поэкспериментировать с обычным и каноническим сплайнами (точнее, с рисующими их методами). Особое внимание следует уделить различиям во взглядах методов `DrawBezier()` и `DrawCurve()` на роль отклоняющих точек в процессе построения сплайна. Если же под рукой компьютера нет, то некоторое представление об этих видах кривых можно получить, просмотрев экранные снимки форм (рис. 33.2).

Верхний ряд снимков демонстрирует поведение классических методов построения кривых – отклоняющие точки лишь оттягивают кривую от прямолинейной траектории. Совсем другой результат получается при применении метода `DrawCurve()`. В этом случае отклоняющие точки выступают не просто в роли «потусторонней» силы, а явным образом определяют форму канонического сплайна (см. нижний ряд экранных снимков на рис. 33.2). Полагаю, что новый способ формирования кривой в большей степени понятен обычно-

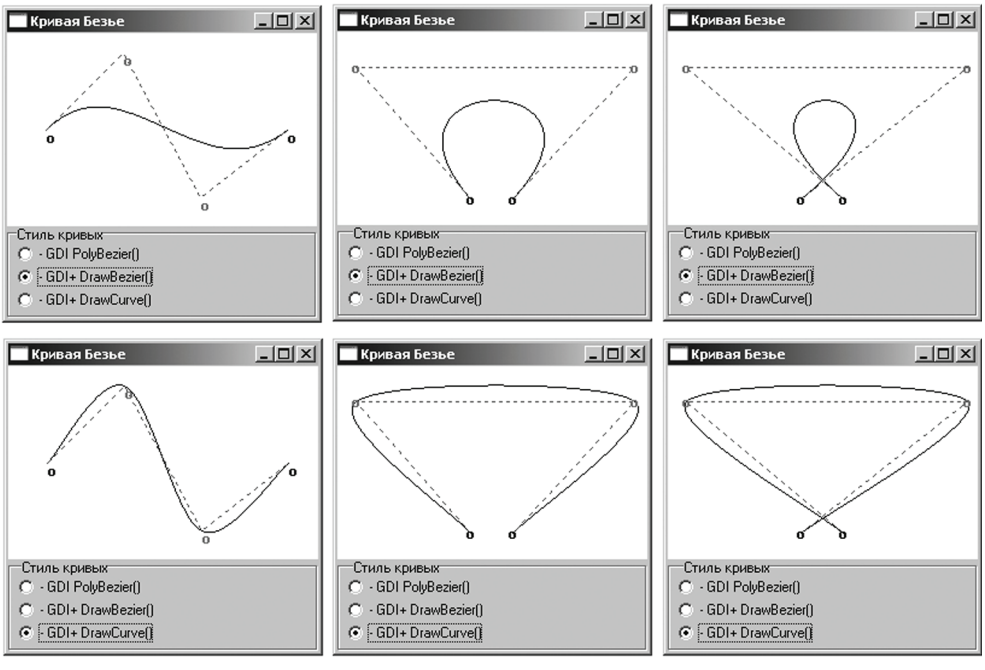


Рис. 33.2. Различия в способах вывода кривых Безье в GDI и GDI+

му пользователю. Отмечу, что нижний ряд рисунков получен с параметром `Tension`, равным 0,5 (значение по умолчанию), малейшие модификации значения степени натяжения кривой способны изменить форму кривой до неузнаваемости (рис. 33.3).

Не затрагивая положения основных и отклоняющих точек, мы управляем натяжением сплайна, изменяя коэффициент `Tension` в диапазоне от –3 до 3 с единичным шагом. Интересно поведение кривой при отсутствии напряженности (нулевое значение), в этом случае сплайн превращается в ломаную линию. При отрицательных значениях коэффициента наша кривая постепенно превращается в «бабочку», сворачиваясь вовнутрь области, ограниченной управляющими точками. И наоборот, переходя в положительный диапазон натяжения, сплайн постепенно выступает за пределы законной области кривой.

В отличие от кубических кривых Безье в GDI канонические сплайны в GDI+ не находятся внутри фигуры с вершинами в определяющих точках кривой. Эта черта сплайнов GDI+ становится особо заметна при увеличении степени натяжения.

На выводе замкнутой кривой (с характером натянутой струны) специализируются еще четыре метода:

```
function DrawClosedCurve(Pen: TGPPen; pPoints: PGPPoint;
                         Count: Integer): TStatus; overload;
```

```
function DrawClosedCurve(Pen: TGPPen; pPoints: PGPPointF;
                         Count: Integer): TStatus; overload;
function DrawClosedCurve(Pen: TGPPen; pPoints: PGPPoint;
                         Count: Integer; Tension: Single): TStatus; overload;
function DrawClosedCurve(Pen: TGPPen; pPoints: PGPPointF;
                         Count: Integer; Tension: Single): TStatus; overload;
```

Аргументы функций аналогичны параметрам `DrawCurve()`, поэтому на них мы останавливаться не станем.

Осталось отметить последний метод холста GDI+, имеющий прямое отношение к работе со сплайнами. На этот раз мы говорим о заливке области, ограниченной замкнутой кривой. С этой целью также реализовано 4 метода:

```
function FillClosedCurve(Brush: TGPBrush; pPoints: PGPPointF;
                         Count: Integer): TStatus; overload;
function FillClosedCurve(Brush: TGPBrush; pPoints: PGPPoint;
                         Count: Integer): TStatus; overload;
function FillClosedCurve(Brush: TGPBrush; pPoints: PGPPoint;
                         Count: Integer; FillMode: TFillMode;
                         Tension: Single = 0.5): TStatus; overload;
function FillClosedCurve(Brush: TGPBrush; pPoints: PGPPointF;
                         Count: Integer; FillMode: TFillMode;
                         Tension: Single = 0.5 ): TStatus; overload;
```

В двух последних методах требуется определить режим заливки `FillMode`.

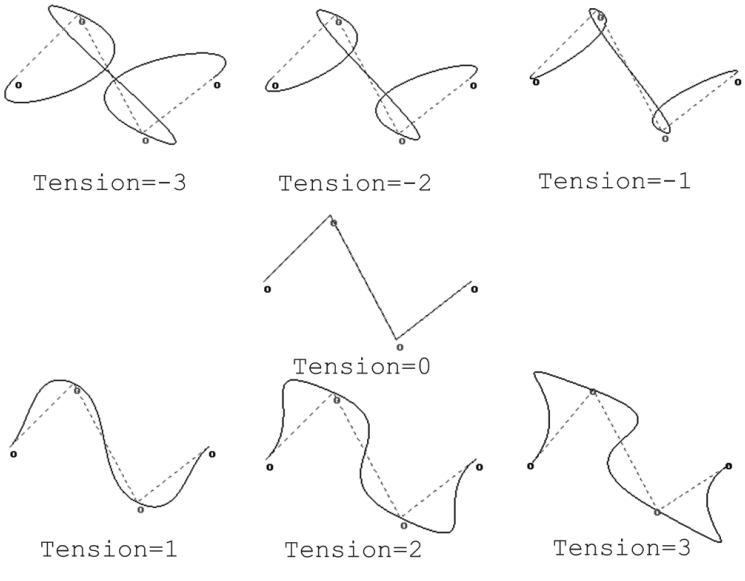


Рис. 33.3. Управление натяжением сплайна GDI+

Простейшие фигуры

Для указания координат вывода большинства простейших фигур, изображение которых вписывается в прямоугольную область, требуется однотипный перечень параметров. В GDI+ предусмотрено 4 варианта аргументов, определяющих местоположение ограничивающего прямоугольника:

1. Прямоугольная область с полями целых чисел — `Rect: TGPRect`.
2. Прямоугольная область с полями вещественных чисел — `Rect: TGPRectF`.
3. Координаты левой верхней точки, ширина и высота — `X`, `Y`, `Width`, `Height: Integer`.
4. Координаты левой верхней точки, ширина и высота в виде вещественных чисел — `X`, `Y`, `Width`, `Height: Single`.

Именно этими аргументами и отличаются друг от друга все четыре версии перегружаемых функций. В табл. 33.1 представлены названия методов, осуществляющих рисование и заливку прямоугольника и эллипса.

Таблица 33.1. Методы вывода прямоугольника и эллипса

Название метода	Описание
`DrawRectangle()`	Вывод контура прямоугольника.
`FillRectangle()`	Заливка прямоугольника.
`DrawEllipse()`	Вывод контура эллипса.
`FillEllipse()`	Заливка эллипса.

Вывод на поверхность холста нескольких прямоугольников осуществляет следующая группа методов:

```
function DrawRectangles(Pen: TGPPen;
                        pRects: PGPRect; Count: Integer): TStatus; overload;
function DrawRectangles(Pen: TGPPen;
                        pRects: PGPRectF; Count: Integer): TStatus; overload;
```

Координаты прямоугольников передаются через указатель на массив `pRects`. Последний параметр функции определяет количество рисуемых фигур. Точно такой же перечень аргументов содержат методы, осуществляющие заливку прямоугольников.

```
function FillRectangles(Brush: TGPBrush;
                        pRects: PGPRect; Count: Integer): TStatus; overload;
function FillRectangles(Brush: TGPBrush;
                        pRects: PGPRectF; Count: Integer): TStatus; overload;
```

Ниже вашему вниманию предложена процедура, демонстрирующая порядок работы с методами рисования группы прямоугольников.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var   Pen : TGPPen; Brush : TGPHatchBrush;
      i : integer;
```

```
      R : array [0..9] of TGPRect;
      Graphics : TGPGraphics;
begin
Graphics:=TGPGraphics.Create(DC);

for i:=0 to 9 do
  R[i]:=MakeRect(5+i*20,20*i+18,30,30);

Pen:=TGPPen.Create(aclRed,2);
Graphics.DrawRectangles(Pen,pGPRect(@R),10);
Pen.Free;

Brush:=TGPHatchBrush.Create(HatchStyleDashedDownwardDiagonal,aclGreen,aclBlue);
Graphics.FillRectangles(Brush,pGPRect(@R),10);
Brush.Free;

Graphics.free;
end;
```

Дуга и сектор окружности

В основу идеи построения дуги и сектора окружности также положены координаты ограничивающего прямоугольника. У каждого из перечисленных в табл. 33.2 метода есть четыре перегружаемых версии, отличающиеся друг от друга способом описания местоположения прямоугольника. Кроме того, у каждого из представленных в таблице методов предусмотрено еще два завершающих параметра: `StartAngle, SweepAngle: Single`, отвечающих за определение углов раскрытия дуги или сектора.

Таблица 33.2. Методы вывода дуги и сектора

Название метода	Описание
`DrawArc()`	Вывод дуги.
`DrawPie()`	Вывод сектора окружности.
`FillPie()`	Заливка сектора окружности.

Рассмотрим особенности работы с перечисленной группой функций на примере метода вывода дуги:

```
function DrawArc(Pen: TGPPen; X, Y, Width, Height,
                 StartAngle, SweepAngle: Single): TStatus; overload;
```

В процессе вывода дуги задействована функция `AngleArc()` классического GDI (см. главу 6 «Линии и кривые»), только несколько изменен порядок применения параметров. В рассматриваемом методе `DrawArc()` точка с координатами `(X,Y)` определяет левый верхний угол прямоугольника с шириной `Width` и высотой `Height`. Дуга рисуется по контуру вписанного в прямоугольник эллипса. Стартовая точка дуги определяется точкой пересечения контура эллипса с лучом, направляемым из центра эллипса под углом `StartAngle` (в градусах) к оси X. Размах дуги назначается другим лучом, также направленным из центра эллипса, но теперь под углом раскрытия `SweepAngle` отно-

сительно стартового луча (рис. 33.4). Ко всему сказанному осталось добавить, что величина угла возрастает по часовой стрелке, так что если необходимо изменить направление вывода – передавайте в последний параметр функции отрицательное значение.

Еще раз напомню, что вывод сектора окружности осуществляется функцией `DrawPie()`, методы сектора снабжены идентичным перечнем аргументов, роль которых повторяет назначение аргументов, использующихся в функции рисования дуги. Ко всему прочему, сектор круга (как замкнутую фигуру) допускается закрасить кистью с помощью метода `FillPie()`. В предложенном ниже примере продемонстрирован порядок работы с сектором окружности.

```
Procedure GDI_PAINT(DC:HDC);
var   Pen:TGPPen;
      Brush:TGPHatchBrush;
      Graphics:TGPGraphics;
const X=10;Y=10; Width=250; Height=150;
begin
Graphics:=TGPGraphics.Create(DC);
 Pen:=TGPPen.Create(aclBlack,1);
 Brush:=TGPHatchBrush.Create(HatchStyleHorizontalBrick,aclYellow,aclRed);

 Graphics.DrawRectangle(Pen,X,Y,Width,Height);
 Graphics.DrawPie( Pen,X,Y,Width,Height,280,90);
 Graphics.FillPie( Brush,X,Y,Width,Height,10,270);

Pen.Free;
Brush.Free;
Graphics.free;
end;
```

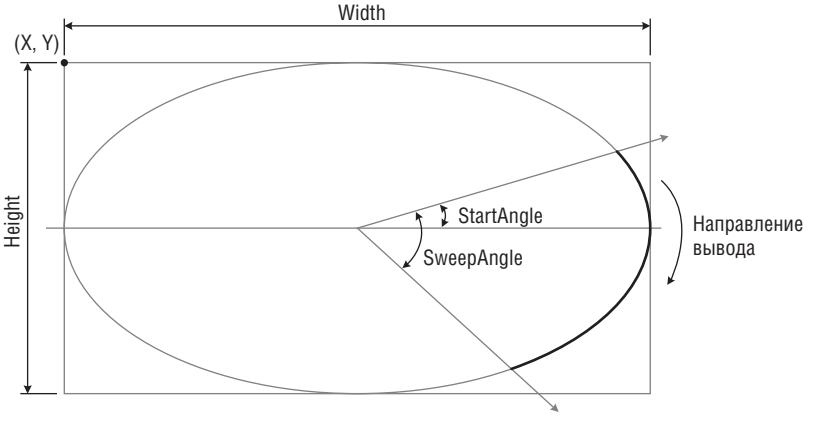


Рис. 33.4. Вывод дуги методом DrawArc( )

Рис. 33.5. Пример вывода сектора окружности

По ходу процедуры мы последовательно выводим контур ограничивающего прямоугольника, затем контур сектора и в заключение закрашиваем кистью с «кирпичным» стилем оставшуюся часть сектора окружности.

РЕЗЮМЕ

В GDI+ решение вопросов, связанных с выводом графических примитивов и заливкой замкнутых областей, возложено на холст `TGPGraphics`. Хотя функционал класса целиком и полностью построен на базе функций Windows API, холст GDI+ может похвастаться существенным новшеством – усовершенствованным способом описания кривых Безье.

34

Координатные системы и преобразования в GDI+

В этой главе мы вновь обратимся к теме координатных пространств, страничных и мировых преобразований. На этот раз нам предстоит рассмотреть, какими возможностями в работе со страничными и мировыми координатами может похвастаться GDI+. Но прежде чем мы приступим к рассмотрению нового материала, скажу, что если вы по каким-то причинам выпустили из своего внимания главы 11 «Системы координат и режимы отображения» и 12 «Мировые координаты», то настоятельно рекомендую хотя бы ненадолго к ним вернуться.

Расширенная версия GDI+ точно так же, как и ее коллега GDI, способна работать в одном из трех координатных пространств: мировом, страничном или в координатном пространстве устройства. Имеющееся в распоряжении операционной системы четвертое по счету пространство координат (называемое физическим) доступно лишь драйверу устройства. Обо всем этом мы уже подробно говорили на первых страницах главы 11. На тех же страницах отмечалось, что все четыре системы координат представляют собой многоуровневую взаимоувязанную систему. Вызвав метод, рисующий точку в мировых координатах, мы, сами не подозревая этого, запускаем сложный механизм, последовательно преобразующий координаты точки из мировых в страничные, затем из страничных в координаты устройства, в заключение – из координат устройства в физические координаты (см. рис. 11.3). Только после всего этого в требуемом месте экрана окажется ожидаемое изображение.

Одно из важных различий между графическими механизмами Windows 95/98/Me и Windows NT/2000/XP/Vista в том, что более ранние версии операционных систем не имеют представления о мировых координатах.

Страничная система координат в GDI+

Контекст графического устройства по умолчанию устанавливается в страничную систему координат. Это сделано для обеспечения совместимости со старыми версиями программ. Для проведения страничных преобразований у холста GDI+ имеется две пары методов, отвечающих, соответственно, за установку единицы измерения и коэффициента масштабирования (табл. 34.1):

Таблица 34.1. Методы класса TGPGraphics, отвечающие за страничные преобразования

Метод	Описание
`function SetPageUnit(Unit_: TUnit): TStatus;`	Метод устанавливает единицу измерения, которая используется при преобразовании из страничных координат в координаты устройства.
`function GetPageUnit: TUnit;`	Возвращает текущую единицу измерения.
`function SetPageScale(Scale: Single): TStatus;`	Метод назначает единицу масштабирования для проведения страничных преобразований.
`function GetPageScale: Single;`	Возвращает текущий коэффициент масштабирования.

Для идентификации единиц измерения введено специальное перечисление:

```
type TUnit_ = (
    UnitWorld,     //=0 логические единицы для работы в мировых координатах
    UnitDisplay,   //=1 переменная для мировых преобразований
    UnitPixel,     //=2 единица измерения – пиксел
    UnitPoint,     //=3 разрешение устройства (пикселей в одном дюйме)
    UnitInch,      //=4 единица измерения – дюйм
    UnitDocument, //=5 единица измерения – 1/300 дюйма
    UnitMillimeter//=6 единица измерения – миллиметр
  );
```

Страничные преобразования не оказывают влияния на физический размер текста, если он выводится шрифтом, созданным с четким определением размера в пунктах, дюймах или миллиметрах.

Кроме того, стоит выделить два метода, позволяющие нам судить о текущем разрешении, установленном в холсте GDI+:

```
function GetDpiX: Single;
function GetDpiY: Single;
```

Функции возвращают число точек в одном дюйме по горизонтали и вертикали. Несмотря на косвенное отношение к механизму страничных преобразований, перечисленные методы весьма полезны при осуществлении тонких настроек, например толщины пера. Это продемонстрировано в первом листинге к этой главе, в примере мы рисуем 10-дюймовую линейку с шагом в 1 дюйм.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    X,Y:Integer;
    Pen:TGPPen;
begin
GPGraphics:=TGPGraphics.Create(DC);

GPGraphics.SetPageUnit(UnitInch);
Pen:=TGPPen.Create(aclBlack,1/GPGraphics.GetDpiY);//перо толщиной 1 пиксел
X:=0; Y:=1;
GPGraphics.DrawLine(Pen,X,Y,X+110,Y);

  While X<10 do
  begin
    GPGraphics.DrawLine(Pen,X,Y,X,Y+0.1);//изображение рисок на линейке
    Inc(X,1);
  end;
GPGraphics.Free; Pen.Free;
end;
```

Для того чтобы при выводе изображения на дисплее толщина линии была минимальной, при создании пера мы выясняем вертикальное разрешение и устанавливаем толщину пера в 1 пиксел.

Управление коэффициентом масштабирования в страничной системе координат тоже весьма интересное занятие. Рассмотрим другой пример.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    X:integer;
    Scale:Single;
    GPPen:TGPPen;
begin
GPGraphics:=TGPGraphics.Create(DC);

GPGraphics.SetPageUnit(UnitMillimeter);
GPPen:=TGPPen.Create(aclSilver,1);

Scale:=1;
For X:=1 to 10 do
begin
GPGraphics.DrawLine(GPPen, X,1,X,10);
GPGraphics.SetPageScale( Scale);
Scale:=Scale+1; {увеличиваем коэффициент масштабирования}
end;

GPGraphics.free; GPPen.Free;
end;
```

Наша задача заключается в черчении десятка вертикальных линий на поверхности холста, настроенного на работу в миллиметрах. В самом начале процедуры мы создаем светло-серое перо единичной толщины, другими словами, толщиной в один миллиметр. Затем в теле цикла `for X:=1 to 10 do` проводим вертикальные линии, при этом в каждой итерации цикла увеличиваем коэффициент масштабирования `Scale` на единицу. Результат работы кода

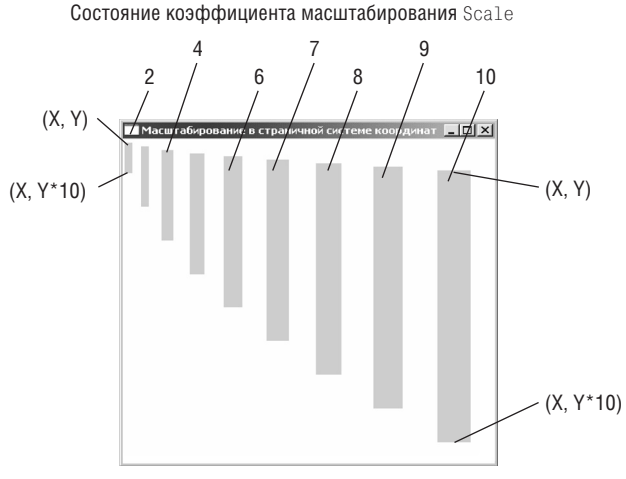


Рис. 34.1. Влияние масштабирования на толщину пера

представлен на рис. 34.1. Обратите внимание на то, что с каждым шагом цикла из-за приращения коэффициента `Scale` происходило не только смещение места вывода очередного отрезка и увеличение его протяженности, но и возрастала толщина пера. В результате последний отрезок был проведен пером толщиной 10 мм.

Вы наверняка удивитесь, когда узнаете, что механизм страничных преобразований в GDI+ не столь гибок, как в GDI! Во-первых, у GDI+ нет возможности перемещать начало координат, оно всегда будет расположено в верхней левой точке. Во-вторых, в GDI+ мы не сможем изменить направление координатных осей. В-третьих, здесь нет и намека на анизотропный режим отображения, в котором нам разрешалось использовать разные единицы измерения для вертикальной и горизонтальной осей. Видимо, разработчики GDI+ полагают, что перечисленные недостатки компенсируются весьма развитой системой мировых преобразований…

Мировые координаты, матрица TGPMatrix

Если бы мне поручили написать сказку о мировых координатах в GDI+, я бы ее начал примерно так: «В бескрайнем пространстве мировых координат GDI+ безраздельно господствуют двумерные аффинные преобразования. А их предводителем выступает грозная матрица из рода `TGPMatrix`». Сказка сказкой, но если разобраться, то действительно окажется, что стержнем, поддерживающим все мировые преобразования во всех без исключения объектах GDI+, действительно служит может быть и не столь грозная, но, что уж точно, весьма высокопоставленная матрица. Впрочем, судите сами.

В главе 12 уже говорилось о матрице преобразований (см. формулу 12.3). Хотя в математическом описании классическая матрица преобразования состоит из девяти элементов (матрица 3×3), неискоренимая тяга Microsoft

к упрощенчеству сделала свое дело, и в GDI+ матрица лишилась последнего ряда. Такое решение основано на том, что третий ряд матрицы всегда содержит значения 0, 0 и 1, и создатели GDI+, видимо, полагают, что мы с вами в состоянии удержать в памяти эти три цифры. Таким образом, экземпляр класса TGPMatrix нацелен на обслуживание массива размерностью 3×2.

$$\begin{bmatrix} eM11 & eM21 & eDx \\ eM12 & eM22 & eDy \end{bmatrix} \tag{34.1}$$

Три из четырех имеющихся в распоряжении TGPMatrix конструкторов в момент создания экземпляра класса матрицы требуют заполнения всех шести ячеек матрицы преобразования. Наибольшей наглядностью обладает конструктор, включающий шесть параметров с именами, эквивалентными названиям ячеек матрицы:

```
constructor Create(eM11, eM12,
                   eM21, eM22,
                   eDx,  eDy: Single); reintroduce; overload;
```

Видимо, для того, чтобы разработчики ПО не скучали во время работы с матрицей преобразования, им предложена еще пара конструкторов, решающих точно такую же задачу, но более изощренным способом:

```
constructor Create(const Rect: TGPRect;
                   const Dstplg: TGPPoint); reintroduce; overload;
constructor Create(const Rect: TGPRectF;
                   const Dstplg: TGPPointF); reintroduce; overload;
```

Здесь поле Rect.x соответствует ячейке $eM11$, Rect.y – $eM12$, Rect.width – $eM21$, Rect.height – $eM22$, поле Dstplg.x относится к eDx, Dstplg.y – к eDy. Не забыли, с чего все начиналось? Если да, то вернитесь к началу абзаца и прочтите все заново...

Четвертый по счету конструктор наиболее покладист и не требует от программиста немедленного заполнения матрицы.

```
constructor TGPMatrix.Create;
```

Самый скромный конструктор заполнит ячейки матрицы значениями по умолчанию: $eM11$ и $eM22$ единицами, а в остальные элементы передаст нулевые значения. Проведение аффинного преобразования в этом режиме не имеет смысла, так как результат преобразования окажется тождественным исходному изображению. Поэтому нам все равно придется отредактировать матрицу, например обратившись к методу:

```
function SetElements(eM11, eM12, eM21, eM22, dX, dY: Single): TStatus;
```

Описанное в матрице правило преобразований может применяться к перу, большинству кистей, траекториям или к холсту GDI+, например как в следующем фрагменте кода.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
```

```
    GPMatrix:TGPMatrix;
    GPPen:TGPPen;
begin
GPPen:=TGPPen.Create(aclRed);
GPGraphics:=TGPGraphics.Create(DC);
GPGraphics.DrawEllipse(GPPen,10,10,80,100);//исходный эллипс

GPMatrix:=TGPMatrix.Create;
GPMatrix.SetElements(1,0.1,1.4,1.15,20,0);

if GPGraphics.SetTransform(GPMatrix)=OK then //применяем преобразования
   begin
   GPPen.SetColor(aclBlue);
   GPGraphics.DrawEllipse(GPPen, 10,10,100,100);//вывод результата
   end;
GPGraphics.Free;  GPPen.Free; GPMatrix.Free;
end;
```

Сначала пером красного цвета мы рисуем эллипс. Затем создаем матрицу преобразований и заполняем ее так, чтобы слегка «помучить» рисунок. Мы одновременно увеличиваем, поворачиваем, деформируем и слегка смещаем по горизонтальной оси исходный эллипс. Результат эксперимента чертится пером синего цвета. В библиотеке на сайте издательства вы обнаружите существенно усложненный пример (экранный снимок которого представлен на рис. 34.2), позволяющий проводить более изощренные эксперименты.

Напомню, что в GDI ответственность за преобразование мировых координат в страничные возложена на функцию `SetWorldTransform()`. *Именно эта функция Windows API приводит в действие механизм аффинных преобразований. В GDI+ аналогичную задачу решает метод класса* `TGPGraphics`, *называемый* `SetTransform()`. *Все подробности о методах холста GDI+, связанных с мировыми преобразованиями, можно найти в табл. 34.2 в конце этой главы.*

Рис. 34.2. Применение матрицы преобразований

Методы поддержки аффинных преобразований

Предвижу удивление читателя, внимательно изучившего главу 12 этой книги: «Какая необходимость введения класса TGPMatrix, если он практически повторяет не вполне удобный стиль работы с мировыми координатами старого доброго GDI?» И действительно, в обычном GDI программисту приходится изрядно поломать голову, дабы получить запланированный результат! Чего только стоит табл. 12.1, в которой была представлена сводная информация о влиянии элементов матрицы на результирующее изображение.

Не спешите с выводами. В недрах класса TGPMatrix прячется более десятка методов, предназначенных как раз для упрощения нашей работы. Но прежде чем приступить к их изучению, познакомимся с функцией, призванной проинформировать программиста о значениях, находящихся в ячейках матрицы:

```
function GetElements(const m: TMatrixArray): TStatus;
```

Результат возвращается в массив из шести элементов:

```
type TMatrixArray = array[0..5] of Single;
```

Значение $eM11$ передается в нулевой элемент массива, $eM12$ – в первый и так далее.

Иногда вместо полного состава матрицы программисту достаточно выяснить только значения смещения, находящиеся в ячейках eDx и eDy. Для решения этой задачи не стоит тревожить метод GetElements(), вместо него рациональнее обратиться к функции:

```
function OffsetX: Single; //ячейка eDx
function OffsetY: Single; //ячейка eDy
```

Теперь приступим к рассмотрению методов, предназначенных для упрощения аффинных преобразований.

Перенос

Наиболее элементарная операция аффинных преобразований связана с обычным переносом изображения влево, вправо, вверх или вниз без внесения в него каких-либо искажений:

```
function Translate(OffsetX, OffsetY: Single;
                   Order: TMatrixOrder = MatrixOrderPrepend): TStatus;
```

Перенос производится за счет изменения состояния элементов матрицы eDx и eDy, в них передаются соответствующие значения смещения из параметров OffsetX и OffsetY. Это утверждение можно проверить на практике, воспользовавшись примером ex34_03.exe из библиотеки или повторив следующий код:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPMatrix:TGPMatrix;
    GPPen:TGPPen;
begin
```

```
  GPGraphics:=TGPGraphics.Create(DC);
  GPPen:=TGPPen.Create(aclRed);
  GPGraphics.DrawEllipse(GPPen,100,100,80,80);
  GPGraphics.DrawRectangle(GPPen,100,100,80,80);

  GPMatrix:=TGPMatrix.Create;
  ShowMatrixValues(DC,GPMatrix,0,0);
  GPMatrix.Translate(-50,40);

  if GPGraphics.SetTransform(GPMatrix)=OK then
     begin
     GPPen.SetColor(aclBlue);
     GPGraphics.DrawEllipse(GPPen,100,100,80,80);
     GPGraphics.DrawRectangle(GPPen,  100,100,80,80);
     end;
  GPGraphics.Free; GPMatrix.Free;
end;
```

В обычном режиме работы на поверхности формы пером красного цвета рисуется круг и обрамляющий его квадрат. Первоначально в ячейках только что появившейся на свет матрицы содержатся значения по умолчанию:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}$$

Затем обращаемся к методу `Translate()`, передав в него значения смещения –50 по оси X и 40 по оси Y. Для большей наглядности перед рисованием второй пары фигур изменяем цвет пера и вновь вызываем методы `DrawEllipse` и `DrawRectangle` с тем же набором аргументов. В заключение интересуемся значениями в ячейках матрицы и выводим ее содержание в панели с названием «Состояние матрицы» (рис. 34.3). Как и было ранее мною обещано, значения смещения оказались в ячейках матрицы eDx и eDy:

$$\begin{bmatrix} 1 & 0 & -50 \\ 0 & 1 & 40 \end{bmatrix}$$

Если вы планируете осуществить несколько аффинных преобразований, то перед этим обязательно вспомните о существовании опционального параметра `Order: TMatrixOrder = MatrixOrderPrepend` *(им снабжены все воздействующие на ячейки матрицы преобразования методы, например* `Translate`, `Scale` *и т. д.). Параметр определяет порядок проведения математических операций между исходной и новой матрицей преобразований, тем самым оказывая влияние на итоговое изображение. Дело в том, что в аффинных преобразованиях, в противовес обычной арифметике, перестановка мест слагаемых изменяет результат. Другими словами, изображение, подвергнутое повороту, деформации и инвертированию, в итоге будет отличаться от такого же рисунка, попавшего под идентичные воздействия, но в другой последовательности.*

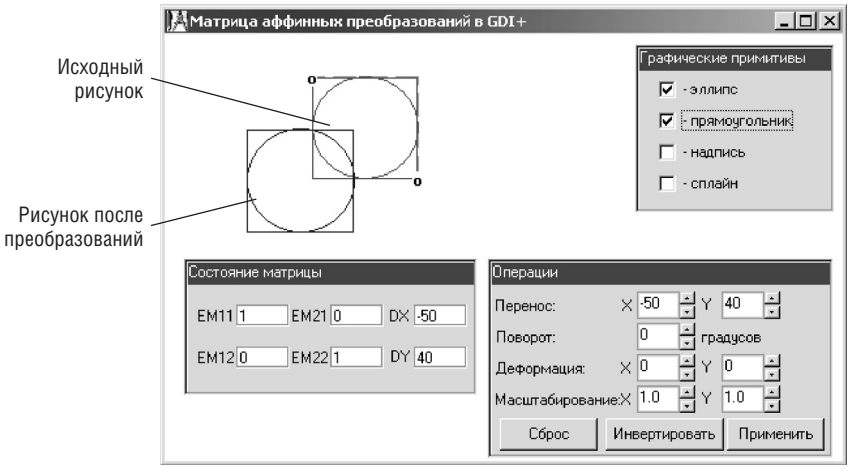


Рис. 34.3. Смещение изображения

Масштабирование

Операцию, связанную с изменением пропорций изображения, проще всего осуществить, обратившись к методу:

```
function Scale(ScaleX, ScaleY: Single;
               Order: TMatrixOrder = MatrixOrderPrepend): TStatus;
```

Множитель назначается раздельно для горизонтальной `ScaleX` и вертикальной осей `ScaleY`. Значение, соответствующее коэффициенту масштабирования по оси X, передается в ячейку *eM*11, коэффициент оси Y – в ячейку *eM*22 матрицы преобразований. На рис. 34.4 представлен результат масштабирования кривой Безье и эллипса с множителями `ScaleX=2` и `ScaleY=3`.

Если логика программы предполагает изменение масштаба только по одной из осей, то во второй множитель следует передавать единицу.

Обратите внимание на то, что изменение масштаба с помощью метода `Scale()` *приводит не только к изменению размера результирующего изображения, но и к смещению точки вывода, ведь она также попала под воздействие аффинных преобразований! Более того, во время аффинных преобразований изменения претерпевает и толщина пера.*

Проведение операций деформации, масштабирования, поворота приводит не только к изменению геометрии рисуемой фигуры, но и к смещению всех ее точек. Зачастую подобное поведение нежелательно, особенно, когда программист планирует изменить размер фигуры, но при этом сохранить местоположение ее левой верхней, центральной или какой либо другой точки. В таком случае необходимо проводить дополнительный расчет коэффициентов *eDx* и *eDy* матрицы преобразования (формулы пересчета предложены в табл. 12.1).

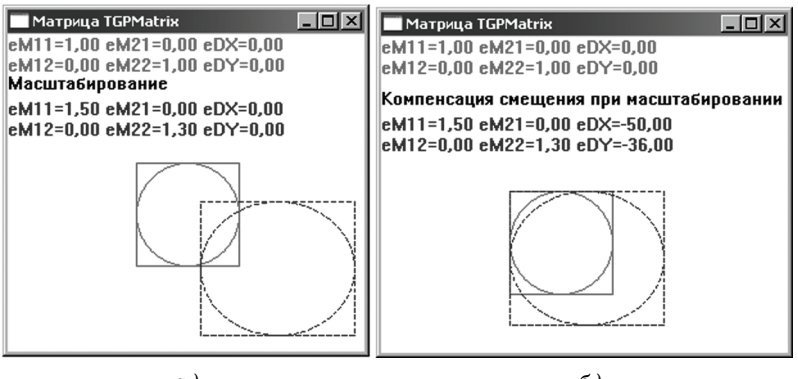
Рис. 34.4. Масштабирование

Ниже предложен листинг, демонстрирующий способ компенсации смещения изображения после операции масштабирования.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPMatrix:TGPMatrix;
    GPPen:TGPPen;
    Rect:TGPRect;
    eM11,eM22,eDx,eDy:single;
begin
GPGraphics:=TGPGraphics.Create(DC);
GPPen:=TGPPen.Create($FFFF0000);
Rect:=MakeRect(100,120,80,80);

GPGraphics.DrawEllipse(GPPen,Rect);
GPGraphics.DrawRectangle(GPPen,Rect);

GPMatrix:=TGPMatrix.Create;

eM11:=1.5;                //коэффициент масштабирования X
eM22:=1.3;                //коэффициент масштабирования Y
eDx:=Rect.X*eM11-Rect.X;  //компенсация по оси X
eDy:=Rect.Y*eM22-Rect.Y;  //компенсация по оси Y

GPMatrix.SetElements(eM11,0,0,eM22,-eDx,-eDy);

if GPGraphics.SetTransform(GPMatrix)=OK then
   begin
   GPPen.SetColor($FF0000FF); GPPen.SetDashStyle(DashStyleDot);
   GPGraphics.DrawEllipse(GPPen,Rect);
   GPGraphics.DrawRectangle(GPPen,Rect);
   end;
GPMatrix.Free; GPPen.Free;
```

```
GPGraphics.Free;
end;
```

Процедура рисует круг, вписанный в квадрат. Координаты этих геометрических фигур определяются прямоугольной областью `Rect:TGPRect`. В коде процедуры мы увеличиваем фигуру по оси X в полтора раза и по оси Y – в 1,3 раза. Но на этот раз перед вызовом метода `SetTransform()` мы производим перерасчет элементов матрицы *eDx* и *eDy*. Таким образом компенсируются вызванные мировыми преобразованиями смещения фигур по вертикали и горизонтали. В результате в правой части рис. 34.5 мы добились совпадения левых верхних точек фигур, нарисованных до и после аффинных преобразований.



а) *б)*

Рис. 34.5. Смещение без компенсации (а) и компенсация смещения (б) при проведении операции масштабирования

Поперечная деформация

Весьма занимательный эффект поперечной деформации достигается с помощью метода:

```
function Shear(ShearX, ShearY: Single;
               Order: TMatrixOrder = MatrixOrderPrepend): TStatus;
```

Деформирующие величины задаются параметрами `ShearX` и `ShearY`, а функция позаботится, чтобы они были переданы в ячейки *eM*21 и *eM*12 матрицы. Представленный ниже рисунок (рис. 34.6) получен за счет вызова метода с параметрами:

```
GPMatrix.Shear(0.4,0.2);
```

Мы подвергли вертикальной и горизонтальной деформации три простейших графических объекта: прямоугольник, текстовую надпись и сплайн (все четыре точки которого расположены вдоль одной линии).

Если логика программы предполагает сдвиг изображения только по одной оси, то во второй незадействованный параметр метода следует направить нулевое значение.

Рис. 34.6. Поперечная деформация

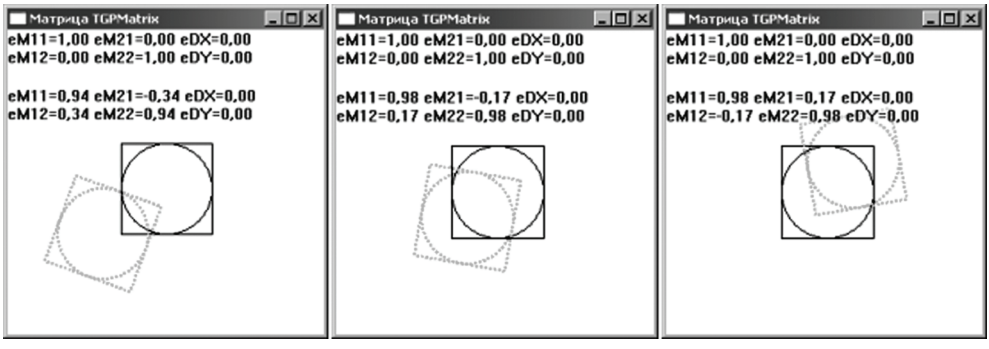
Поворот

Мы подошли к еще более сложному действию – повороту изображения. Поворот – это весьма непростая операция, при ее выполнении задействуются 4 из 6 ячеек матрицы. Если вы ненадолго вернетесь к табл. 12.1, то вспомните, что для организации этой операции в элементы *eM*11, *eM*12, *eM*21 и *eM*22 направляются синусы и косинусы угла поворота. Такие пересчеты несколько утомительны, поэтому для проведения трансформации подобного рода в GDI+ предусмотрены два весьма простых метода.

```
function Rotate(Angle: Single;
                Order: TMatrixOrder = MatrixOrderPrepend): TStatus;
function RotateAt(Angle: Single; const Center: TGPPointF;
                  Order: TMatrixOrder = MatrixOrderPrepend): TStatus
```

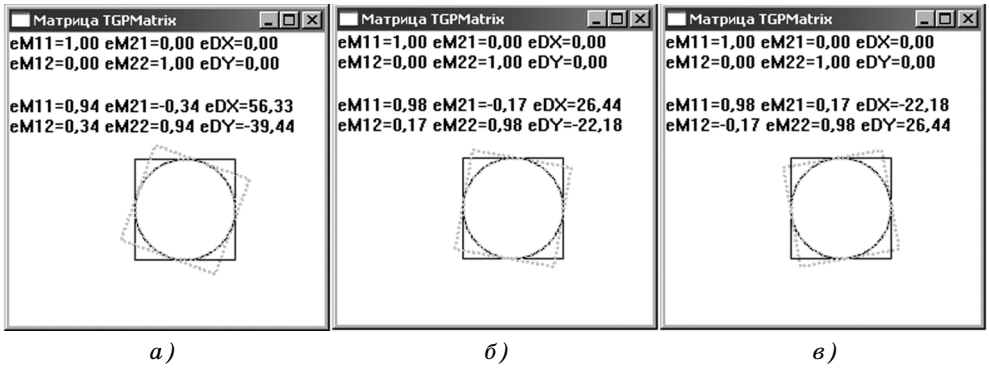
Поворот осуществляется против часовой стрелки на угол, переданный в параметр `Angle`. Первая из перечисленных функций может и не оправдать наших надежд. Изучите рис. 34.7. Здесь представлены три экранных снимка окна, в которых исходный рисунок расположен в центре окна, а его дубликат повернут на 20, 10 и –10 градусов соответственно. Как видите, результирующее изображение, хотя и действительно повернуто на заданную величину, но выведено практически в непредсказуемом месте окна. Такой отрицательный эффект объясняется тем, что аффинному преобразованию были подвержены все базовые точки прямоугольника и эллипса, поэтому новый рисунок смещен относительно центральной точки исходного изображения.

Метод `RotateAt()` призван устранить недостаток своего «неуклюжего» коллеги. Для этого в метод включен дополнительный параметр `Center`, определяющий центральную точку, относительно которой и осуществляется поворот изображения. Передав в `Center` координаты центра вращаемой фигуры, мы получим вполне приемлемый результат (рис. 34.8). Метод весьма умен, для



а) *б)* *в)*

Рис. 34.7. Повороты на 20 (а), 10 (б) и -10 градусов (в) с помощью метода Rotate( )



а) *б)* *в)*

Рис. 34.8. Повороты на 20 (а), 10 (б) и -10 градусов (в) с помощью метода RotateAt( )

того, чтобы изображение не «скакало» по окну, он без постороннего вмешательства рассчитывает необходимое смещение и передает его в ячейки *eDx* и *eDy*. Подтверждением этого служит то, что у нас полностью совпали контуры исходной и поворачиваемой окружности.

Инвертирование

В состав дополнительных фокусов класса `TGPMatrix` входит возможность инвертирования результата аффинных преобразований. Для этого достаточно вызвать метод:

```
function Invert: TStatus;
```

При проведении аффинных преобразований инвертирование – это далеко не всегда умножение элементов матрицы на минус единицу. GDI+ рассчитывает новые значения матрицы таким образом, чтобы мы получили обратный визуальный эффект. Рассмотрите рис. 34.9. В окне выводится два графических примитива: надпись и прямоугольник. Первоначально (верхний рису-

нок) мы задаем весьма сложное преобразование, включающее смещение по оси Y, поворот и горизонтальную деформацию. В итоге в ячейках матрицы преобразований оказываются следующие значения:

$$\begin{bmatrix} 0{,}94 & 0{,}79 & 0 \\ 0{,}34 & 1{,}35 & -30 \end{bmatrix}$$

Затем инициируется операция инвертирования (нижняя часть рис. 34.9). После этой операции весьма полезно изучить состояние матрицы аффинных преобразований. Значения в ячейках $EM21$ и $EM12$ изменили знак на противоположенный, а значения $EM11$ и $EM22$ даже обменялись местами:

$$\begin{bmatrix} 1{,}35 & -0{,}79 & -23{,}56 \\ -0{,}34 & 0{,}94 & 28{,}19 \end{bmatrix}$$

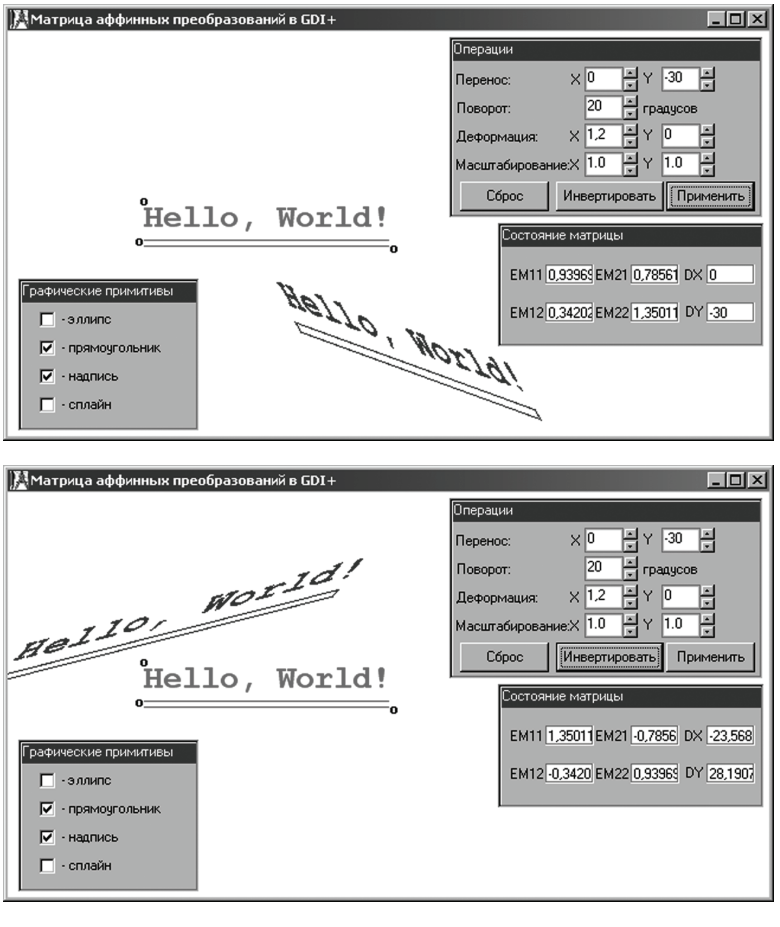


Рис. 34.9. Графические примитивы до и после инвертирования

Сброс настроек преобразования

Вдоволь наигравшись с матрицей аффинных преобразований, приводим ее в исходное состояние. Сброс осуществит метод:

```
function Reset: TStatus;
```

В результате в ячейках матрицы окажутся значения по умолчанию:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}.$$

Аффинные преобразования с точкой

Как правило, для осуществления аффинных преобразований программисты обращаются к методу `SetTransform()` холста GDI+. Вместе с тем предусмотрены альтернативные способы применения механизма аффинных преобразований. Непосредственно матрица `TGPMatrix` вооружена методами, позволяющими применять аффинные преобразования не для контекста графического устройства в целом, а для отдельной точки или нескольких точек массива:

```
function TransformPoints(Pts: PGPPointF; Count: Integer = 1): TStatus; overload;
function TransformPoints(Pts: PGPPoint; Count: Integer = 1): TStatus; overload;
```

В параметр `Pts` функции передается указатель на массив, содержащий одну или несколько точек (их количество определяется в параметре `Count`). В результате выполнения операции в массиве `Pts` окажутся новые координаты точек, значения которых пересчитаны в соответствии с описанным в матрице законом аффинных преобразований. Порядок работы с методом представлен в следующем листинге:

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics : TGPGraphics;
    GPMatrix:TGPMatrix;
    GPPen: TGPPen;
const  Points: array[0..4] of TGPPoint =((x:50; y:100),
                     (x:100 ; y:80 ), (x:150; y:125 ),
                     (x:200; y:100 ), (x:250; y:150));
begin
  GPGraphics := TGPGraphics.Create(DC);
  GPPen:= TGPPen.Create(aclRed);
  GPGraphics.DrawCurve(GPpen, PGPPoint(@Points), 5);

  GPMatrix:=TGPMatrix.Create(1, 0, 0, 2, 0, 0); //заполним матрицу
  GPMatrix.TransformPoints(PGPPoint(@Points), 5);//пересчет

  GPPen.SetColor(aclBlue);
  GPGraphics.DrawCurve(GPpen, PGPPoint(@Points), 5);

  GPGraphics.Free; GPPen.Free;  GPMatrix.Free;
end;
```

В результате работы процедуры аффинным преобразованиям были подвергнуты пять точек массива Points.

Аффинные преобразования с вектором

До сих пор при обсуждении процесса аффинных преобразований в своих примерах мы отталкивались от привычной нам сущности – точки. Вряд ли стоит напоминать, что в двухмерной графике (впрочем, как и в окружающем нас мире) точка однозначно отождествляется с местоположением в пространстве. Между тем решение математических задач, в особенности связанных с представлением таких физических понятий, как скорость или сила, требует принципиально другой сущности – вектора. Физический смысл вектора представляет собой силу и направление ее приложения.

Результаты аффинных преобразований над вектором несколько отличаются от подобных действий с точкой. Вектор может быть увеличен или уменьшен, может быть повернут или может поменять свое направление на противоположнное, но, в отличие от точки, вектор не может быть перемещен – он не реагирует на изменение значений полей *eDx* и *eDy* матрицы. Поэтому в GDI+ для работы с вектором предусмотрена отдельная группа методов:

```
function TransformVectors(Pts: PGPPointF; count: Integer = 1): TStatus; overload;
function TransformVectors(Pts: PGPPoint; count: Integer = 1): TStatus; overload;
```

Перечень и назначение параметров функций аналогичен методам по работе с точкой, поэтому сразу перейдем к примеру.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics : TGPGraphics;
    GPMatrix:TGPMatrix;
    GPPen: TGPPen;
    GPBrush:TGPSolidBrush;
const
    p: TGPPoint =(x:120; y:120); //координаты точки
    v: TGPPoint =(x:120; y:120); //координаты вектора
begin
  GPGraphics := TGPGraphics.Create(DC);
  GPPen:= TGPPen.Create($FF0000FF);
  GPBrush:=TGPSolidBrush.Create($FF0000FF);

  GPgraphics.FillEllipse(GPBrush, p.X - 4, p.Y - 4, 8, 8); //точка
  GPgraphics.DrawLine(GPPen, MakePoint(0, 0), v);  //вектор

  GPMatrix:=TGPMatrix.Create;
  GPMatrix.Rotate(20); //поворот на 20 градусов
  GPMatrix.Translate(50,0); //перенос по оси x
  GPMatrix.TransformPoints(pGPPoint(@p));
  GPMatrix.TransformVectors(pGPPoint(@v));

  GPPen.SetColor($FFFF0000);
  GPBrush.SetColor($FFFF0000);
  GPGraphics.FillEllipse(GPBrush, p.X - 4, p.Y - 4, 8, 8);
  GPGraphics.DrawLine(GPPen, MakePoint(0, 0), v);
```

```
    GPPen.Free;   GPBrush.Free;
    GPMatrix.Free;
    GPGraphics.Free;
  end;
```

В ходе примера мы объявляем две переменные p и v, хранящие координаты точки и вектора. Заметьте, что изначально их координаты идентичны, поэтому при их первом выводе экран (на рис. 34.10 точка изображается окружностью, а вектор линией) они сходятся в одном и том же месте. Затем мы формируем матрицу преобразований и делаем это в два этапа: сначала осуществляем поворот на 20 градусов по часовой стрелке (метод Rotate), а затем перенос (метод Translate) по оси X на 50 логических единиц. Первая операция задействует ячейки *eM*11, *eM*12, *eM*21 и *eM*22, а последняя окажет влияние только на ячейку *eDx* матрицы. Применяем к точке метод TransformPoints(), а к вектору – TransformVectors() и вновь смотрим на рис. 34.10. Теперь «пути» вектора и точки разошлись. Причина в том, что точка отреагировала и на команду Rotate(), и на команду Translate(). Вектор же был лишь повернут, а к команде переноса он остался безразличен.

Рис. 34.10. Аффинные преобразования с вектором

В процессе аффинных преобразований вектора не задействуются элементы eDx и eDy матрицы, поэтому он не может быть перемещен.

Дополнительные возможности класса TGPMatrix

Очень часто результирующее изображение формируется за счет нескольких последовательных преобразований. Один из вариантов построения итоговой матрицы заключается в проведении операции умножения матриц:

```
function Multiply(Matrix: TGPMatrix;
                  Order: TMatrixOrder = MatrixOrderPrepend): TStatus;
```

Термин «умножение» не вполне корректен и использован здесь лишь потому, что это дословный перевод названия метода. На самом деле правильнее

сказать, что мы обновляем матрицу на основе данных, передаваемых через параметр Matrix.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var
  GPGraphics : TGPGraphics;
  GPMatrix1,GPMatrix2:TGPMatrix;
  GPPen: TGPPen;
begin
  GPGraphics := TGPGraphics.Create(DC);
  GPPen:= TGPPen.Create($FF0000FF); //синее перо
  GPGraphics.DrawRectangle(GPPen, 0, 0, 100, 100);

  GPMatrix1:=TGPMatrix.Create(1, 1.2, 0, 1, 0, 0);
  GPMatrix2:=TGPMatrix.Create(1, -1.2, 0, 1, 0, 0);

  GPMatrix1.Multiply(GPMatrix2, MatrixOrderAppend); //умножение
  GPGraphics.SetTransform(GPMatrix1); //трансформация

  GPPen.SetColor($FFFF0000); //красное перо
  GPGraphics.DrawRectangle(GPPen, 0, 0, 100, 100);

  GPPen.Free;
  GPMatrix1.Free; GPMatrix2.Free;
  GPGraphics.Free;
end;
```

В примере встречаются две команды DrawRectangle() на вывод синего и красного прямоугольника. Синий прямоугольник рисуется до проведения преобразований (связанных с деформацией по оси X), красный – после. Угадайте, что мы увидим по завершению работы? Оба прямоугольника будут нарисованы в одном и том же месте. Операция «умножения» двух матриц GPMatrix1 и GPMatrix2 привела к взаимной компенсации аффинных преобразований. Причина в том, что при создании экземпляров матриц мы использовали идентичные параметры, за исключением второго аргумента. В первом случае в параметр, отвечающий за поле матрицы $eM12$, было передано положительное значение, а во втором случае – то же значение, только с отрицательным знаком.

В заключение разговора о матрице упомянем три информационных метода. Если матрица обратима, то метод:

```
function IsInvertible: BOOL;
```

возвращает значение true. Проверку на тождественность осуществляет метод:

```
function IsIdentity: BOOL;
```

Сравнение матриц производит метод:

```
function Equals(matrix: TGPMatrix): BOOL;
```

Если значение ячеек матриц совпадают, то метод вернет true.

Мировые преобразования, класс TGPGraphics

Хотя параметры преобразований хранятся в матрице, но непосредственно процессом преобразований заведует холст GDI+. На протяжении этой главы мы уже не раз пользовались услугами класса TGPGraphics, а теперь настал момент познакомиться с частью его методов, нацеленных на аффинные преобразования (табл. 34.2).

Таблица 34.2. Методы холста GDI+, отвечающие за мировые преобразования

Метод	Описание
`Function SetTransform(Matrix: TGPMatrix): TStatus;`	Применение аффинных преобразований, установленных в матрице Matrix к холсту GDI+.
`Function ResetTransform: TStatus;`	Обнуление настроек преобразований.
`Function MultiplyTransform(Matrix: TGPMatrix; Order: TMatrixOrder = MatrixOrderPrepend): TStatus;`	Обновляет текущие настройки холста за счет полей матрицы Matrix.
`Function TranslateTransform(dX, dY: Single; Order: TMatrixOrder = MatrixOrderPrepend): TStatus;`	Перенос по горизонтальной и вертикальной осям на расстояния dX и dY соответственно.
`Function ScaleTransform(dX, dY: Single; Order: TMatrixOrder = MatrixOrderPrepend): TStatus;`	Операция масштабирования, где X и Y – коэффициенты масштабирования по горизонтальной и вертикальной осям.
`function RotateTransform(Angle: Single; Order: TMatrixOrder = MatrixOrderPrepend): TStatus;`	Поворот на угол Angle.
`function GetTransform(Matrix: TGPMatrix): TStatus;`	Справочный метод. Заполняет поля матрицы Matrix текущими установками холста.
`function TransformPoints( DestSpace: TCoordinateSpace; SrcSpace: TCoordinateSpace; pPts: PGPPointF; Count: Integer): TStatus; overload;` `function TransformPoints( DestSpace: TCoordinateSpace; SrcSpace: TCoordinateSpace; pPts: PGPPoint; Count: Integer): TStatus; overload;`	Метод специализируется на мировых и страничных преобразованиях. Он конвертирует координаты точек, входящих в массив (указатель на массив pPts), между координатными пространствами DestSpace и SrcSpace. Количество подлежащих трансформации элементов массива определяется в параметре Count. `type TCoordinateSpace = ( CoordinateSpaceWorld, CoordinateSpacePage, CoordinateSpaceDevice);`

РЕЗЮМЕ

Решение задач проведения двумерных аффинных преобразований в библиотеке GDI+ возложено на матрицу преобразований – класс `TGPMatrix`. Экземпляр класса ведет «переговоры» с мировыми координатами на тему трансформации изображения. Класс `TGPMatrix` не может похвастаться какими-то уникальными способностями, он осуществляет те же самые операции (перенос, масштабирование, деформация и поворот), что и графическая система GDI. Вместе с тем у матрицы преобразований GDI+ есть одно неоспоримое преимущество – наличие интуитивно понятных методов, благодаря которым существенно упрощается процесс программирования и повышается читабельность кода.

35

Изображения в GDI+

В сравнении с GDI (вообще не имевшего универсального объекта для работы с изображениями) GDI+ порадовал программистов набором классов, предназначенных для работы с растровой и векторной графикой. Впечатляет расширившийся список поддерживаемых форматов. К еще большему удовольствию разработчиков программного обеспечения, к имеющимся в распоряжении GDI форматам BMP, ICON, WMF и EMF добавились GIF, JPEG, PNG, TIFF и обновленная версия EMF+. Кроме того, GDI+ умеет обрабатывать очень популярные в современной цифровой фотографии метаданные в формате EXIF.

С форматами BMP, ICON, JPEG, WMF и EMF мы уже достаточно хорошо знакомы по главам, посвященным GDI и VCL. Поэтому не стану повторяться и лучше представлю читателю TIFF, GIF и PNG.

Формат TIFF (*Tagged Image File Format* – тэговый формат файлов изображений) создавался как стандарт для сканированных изображений. Такое название раскрывает суть хранения данных – каждое изображение описывается с помощью набора полей (тегов). TIFF появился на свет в те старозаветные времена, когда в мире господствовали MS-DOS, OS/2 и Unix. Говорят, что в ту пору люди, встретив программиста на улице, не бросали в него камнями, а напротив – радушно снимали перед ним шляпы... Так вот уже тогда TIFF мог похвастаться весьма интересными способностями. Например, в одном файле TIFF допускалось хранить несколько изображений. Чудеса, да и только! Еще одна особенность формата заключается в поддержке сжатия без потерь за счет алгоритма LZW[1].

Формат GIF (*Graphic Interchange Format*) тоже не молод. Его первая спецификация была опубликована в 1987 году компанией Compuserve. Как следует из названия, GIF в первую очередь предназначался для сетевого обмена,

---

[1] Алгоритм сжатия назван по первым буквам фамилий его разработчиков (Lempel-Ziv-Welth).

поэтому разработчики формата приложили максимум усилий для сокращения размера файла. Именно поэтому максимальная поддерживаемая в формате глубина цвета составляет всего 8 бит, кроме того, изображение в GIF может быть сжато за счет метода LZW. Кроме легковесности GIF у него имеется еще одна положительная черта – поддержка многостраничных изображений.

Тяжело сказать, хорошо это или плохо, но из всех форматов растровой графики, рассматриваемой в этой книге, GIF долго относился к наиболее защищенным с юридической точки зрения. В первую очередь это произошло благодаря применяемому в GIF патентованному методу сжатия данных LZW. С 1994 года обладающая правами на LZW компания Unisys Corporation начала требовать от пользователей лицензионных выплат за использование этого алгоритма в GIF. И хотя сам GIF является открытым форматом, из-за юридических проблем с LZW любовь пользователей к GIF пошла на убыль. На сегодняшний день срок действия патента истек, так что GIF вновь на свободе!

Новейший из всех перечисленных форматов – *PNG* (*Portable Network Graphics*). Он вышел в свет осенью 1996 года. Если следовать дословному переводу PNG, это портативная сетевая графика. Чем знаменит PNG? Во-первых, он использует алгоритм сжатия без потерь, что в выгодную сторону отличает его от JPEG. Во-вторых, формат поддерживает 48-битную глубину цвета. Таким образом, PNG – единственный из всех рассматриваемых в этой книге форматов растровой графики, способный без постороннего вмешательства управлять прозрачностью! В-третьих, PNG характеризуется наилучшим цветовым соответствием (он поддерживает пять различных цветовых моделей).

Кодеры и декодеры изображений

В главе 16 этой книги мы изрядно потрудились, решая на первый взгляд банальную задачу загрузки растрового рисунка в формате BMP. В результате наш далеко не полный листинг занял более 60 строк кода, что способно привести в трепет даже весьма продвинутого программиста. К счастью, в GDI+ от нас уже не требуются такие подвиги. Поддержка графическими классами GDI+ разнообразных форматов рисунков реализуется не за счет наших бессонных ночей, а благодаря услугам заранее установленных в операционной системе Windows кодеров и декодеров – специальных программ, позволяющих, соответственно, кодировать и декодировать разнотипные изображения.

О числе имеющихся в распоряжении ОС кодеров проинформирует объявленная в модуле GDIPOBJ функция:

```
function GetImageEncodersSize(out NumEncoders, Size: UINT): TStatus;
```

Помимо данных о количестве кодеров (которые возвращаются через параметр `NumEncoders`) функция укажет, какой объем памяти потребуется для получения подробностей о каждом кодировщике. Эта информация помещается в параметр `Size`. Полученные значения могут быть востребованы функцией:

```
function GetImageEncoders(NumEncoders, Size: UINT;
                          pEncoders: PImageCodecInfo): TStatus;
```

Благодаря GetImageEncoders() мы сможем построить полный список установленных кодеров. Для этого в функции предусмотрен параметр pEncoders, представляющий собой указатель на массив записей:

```
type TImageCodecInfo = packed record
    Clsid             : TGUID; //идентификатор кодека
    FormatID          : TGUID; //идентификатор типа изображения
    CodecName         : PWCHAR;//название кодека
    DllName           : PWCHAR;//библиотека, в которой размещен кодек
    FormatDescription : PWCHAR;//название файлового формата
    FilenameExtension : PWCHAR;//расширения файлов
    MimeType          : PWCHAR;//мнемонический идентификатор типа
    Flags             : DWORD; //флаги ImageCodecFlags
    Version           : DWORD; //версия кодека
    SigCount          : DWORD; //число сигнатур, связанных с кодеком
    SigSize           : DWORD; //размер сигнатуры
    SigPattern        : PBYTE; //указатель на массив с шаблоном сигнатуры
    SigMask           : PBYTE; //указатель на массив с маской сигнатуры
end;
```

Обратная задача, связанная с получением списка декодеров, решается с помощью еще одной пары функций:

```
function GetImageDecodersSize(out NumDecoders, size: UINT): TStatus;
function GetImageDecoders(NumDecoders, Size: UINT;
                          pDecoders: PImageCodecInfo): TStatus;
```

Как видите, функции исследования декодеров обладают точно таким же набором параметров, как и функции кодеров. Повторив следующий пример, вы получите подробную информацию об имеющихся в нашем распоряжении декодерах изображений.

```
Procedure GetDecoders(LISTBOX :  HWnd);
var S:String;
    NumDecoders, Size, I: UINT;
    ICI:Array of TImageCodecInfo;
begin
GetImageDecodersSize(NumDecoders, Size);
SetLength(ICI,Size DIV SizeOf(TImageCodecInfo));
GetImageDecoders(NumDecoders, Size,PImageCodecInfo(ICI));

For I:=0 to NumDecoders-1 do
 begin
  S:=Format('======== Декодер %s =======',[GUIDToString(ICI[i].Clsid)]);
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
  S:='Название: '+ICI[i].CodecName;
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
  S:='Поддерживаемый формат: '+ICI[i].FormatDescription;
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
  S:='GUID формата: '+GUIDToString(ICI[i].FormatID);
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
  S:='Расширения файлов: '+ICI[i].FilenameExtension;
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
```

```
    S:='Тип MIME: '+ICI[i].MimeType;
    SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
   {пустая строка для разделения декодеров}
    SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(''+#0)));
  end;
end;
```

Обработанная информация сбрасывается в список, указанный в параметре LISTBOX (рис. 35.1).

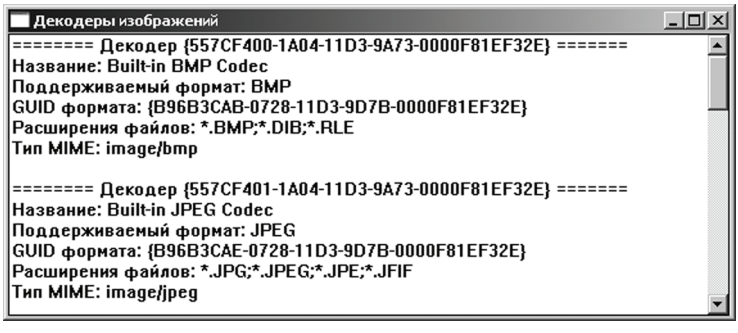


Рис. 35.1. Перечень декодеров изображений

В системе вы не обнаружите кодеры векторных форматов WMF и EMF. Дело в том, что построение любых векторных рисунков полностью зависит от особенностей разрабатываемого программного обеспечения и не может быть унифицировано.

В этой главе мы еще вернемся к кодерам при обсуждении параметров кодирования изображений.

Класс TGPImage

В рамках библиотеки GDI+ отправной точкой для работы с растровыми и векторными рисунками выступает класс TGPImage. Фактически TGPImage представляет собой универсальный контейнер, способный инкапсулировать векторные и растровые изображения. Несмотря на наличие двух прямых потомков (TGPBitmap и TGPMetafile), также специализирующихся на обработке изображений, опорный класс ни в коем случае не абстрактный и способен самостоятельно управляться с основными задачами по обслуживанию рисунков.

Жизнь контейнера начинается с вызова одного из представленных ниже конструкторов:

```
constructor Create(FileName: WideString;
            UseEmbeddedColorManagement: BOOL = FALSE); reintroduce; overload;
constructor Create(Stream: IStream;
            UseEmbeddedColorManagement: BOOL  = FALSE); reintroduce; overload;
```

Исходное изображение может быть получено из файла с именем FileName или загружено из памяти, для этого предназначен поток, описываемый интер-

фейсом Stream. Второй параметр в обоих методах одинаков. Передав в UseEmbeddedColorManagement значение true, мы активируем дополнительный механизм цветовой коррекции. В результате к созданному объекту подключается дополнительный профайл, стандартизированный Международным консорциумом по средствам обработки цветных изображений (*International Color Consortium*, ICC).

Наряду с конструкторами зачастую используются равноценные им по возможностям методы FromFile() *и* FromStream().

Исследование изображения

С точки зрения класса TGPImage едва ли не самым основным свойством изображения будет его тип. Информацию о типе предоставит функция:

```
function GetType: TImageType;
```

Именно тип рисунка определяет дальнейшее поведение компонента. Вариантов ответа всего три, они описываются перечислением:

```
type TImageType = (ImageTypeUnknown,   //тип неизвестен
                   ImageTypeBitmap,    //битовый образ
                   ImageTypeMetafile);//векторное изображение
```

Более детализированные данные предоставит функция:

```
function GetRawFormat(out Format: TGUID): TStatus;
```

Метод возвращает *глобальный уникальный идентификатор* (*Globally Unique IDentifier*, GUID) загруженного в контейнер изображения. Значения GUID хранятся в системном реестре Windows в ветви HKEY_CLASSES_ROOT. Все необходимые для нашей работы идентификаторы объявлены в модуле GDIPAPI, а соответствующие им константы представлены в табл. 35.1.

Таблица 35.1. Глобальные идентификаторы изображений

Формат	Константа	GUID
Неизвестный	ImageFormatUndefined	{B96B3CA9-0728-11D3-9D7B-0000F81EF32E}
BMP в памяти	ImageFormatMemoryBMP	{B96B3CAA-0728-11D3-9D7B-0000F81EF32E}
BMP	ImageFormatBMP	{B96B3CAB-0728-11D3-9D7B-0000F81EF32E}
EMF	ImageFormatEMF	{B96B3CAC-0728-11D3-9D7B-0000F81EF32E}
EXIF	ImageFormatEXIF	{B96B3CB2-0728-11D3-9D7B-0000F81EF32E}
GIF	ImageFormatGIF	{B96B3CB0-0728-11D3-9D7B-0000F81EF32E}
ICON	ImageFormatIcon	{B96B3CB5-0728-11D3-9D7B-0000F81EF32E}
JPEG	ImageFormatJPEG	{B96B3CAE-0728-11D3-9D7B-0000F81EF32E}
PNG	ImageFormatPNG	{B96B3CAF-0728-11D3-9D7B-0000F81EF32E}
TIFF	ImageFormatTIFF	{B96B3CB1-0728-11D3-9D7B-0000F81EF32E}
WMF	ImageFormatWMF	{B96B3CAD-0728-11D3-9D7B-0000F81EF32E}

После того как формат рисунка будет однозначно идентифицирован, можно обращаться за дополнительной информацией. Геометрические размеры рисунка в пикселах возвратит пара методов:

```
function GetWidth: UINT; //ширина
function GetHeight: UINT;//высота
```

Ровно такого же результата можно добиться за счет вызова функции:

```
function GetPhysicalDimension(out Size: TGPSizeF): TStatus;
```

На этот раз данные представляются в виде вещественных чисел и возвращаются через параметр Size.

За информацией о горизонтальном и вертикальном разрешении изображения стоит обратиться к функциям:

```
function GetHorizontalResolution: Single;
function GetVerticalResolution: Single;
```

В результате мы узнаем количество пикселов в одном дюйме. Для многих начинающих пользователей термин «пиксел» или выражение «пикселов в дюйме» сродни тарабарщине, поэтому подобные данные лучше конвертировать в более понятные единицы измерений, например в сантиметры. Для этого предлагаю воспользоваться простым выражением:

```
X:= 2.54*GPImage.GetWidth/GPImage.GetHorizontalResolution;//ширина в см
```

При работе с изображениями (в особенности, с легко масштабируемой векторной графикой) бывает полезно знать размеры ограничивающего прямоугольника:

```
function GetBounds(out SrcRect: TGPRectF; out SrcUnit: TUnit): TStatus;
```

Функция обладает двумя выходными параметрами. В структуру, определяемую параметром SrcRect, возвращаются размеры прямоугольной области, в SrcUnit передаются установленные в изображении единицы измерения.

Для выяснения глубины цвета обслуживаемого объекта предназначен метод:

```
function GetPixelFormat: TPixelFormat;
```

Возвращаемое функцией значение требует расшифровки. Тип данных TPixelFormat представляет собой 32-битное значение в котором биты 0..7 хранят индекс формата, биты 8..15 указывают, сколько бит необходимо для описания одного пиксела, биты 16..23 содержат специальные флаги и биты 24..31 ожидают лучших времен (они пока в резерве). В табл. 35.2 представлены заранее подготовленные константы, описывающие глубину цвета.

Много полезной информации об атрибутах изображения предоставит метод:

```
function GetFlags: UINT;
```

Функция возвращает комбинацию флагов, представленных в табл. 35.3.

Таблица 35.2. Константы глубины цвета изображения

Константа	Глубина цвета бит/пикс	Примечание
PixelFormat1bppIndexed	1	Черно-белое изображение.
PixelFormat4bppIndexed	4	Применяет палитру.
PixelFormat8bppIndexed	8	Применяет палитру.
PixelFormat16bppARGB1555	16	По 5 бит отводится на каналы RGB и 1 бит на альфа-канал.
PixelFormat16bppGrayScale	16	Оттенки серого.
PixelFormat16bppRGB555	16	По 5 бит отводится на каналы RGB. Альфа-канал не кодируется.
PixelFormat16bppRGB565	16	По 5 бит на каналы R и B, 6 бит – канал G.
PixelFormat24bppRGB	24	По 8 бит на каналы RGB. Альфа-канал не кодируется.
PixelFormat32bppARGB	32	По 8 бит на каждый канал ARGB.
PixelFormat32bppPARGB	32	По 8 бит на каждый канал ARGB. Улучшен механизм смешивания компонент RGB и альфа-канала.
PixelFormat32bppRGB	32	По 8 бит на каналы RGB. Альфа-канал не используется.
PixelFormat48bppRGB	48	По 16 бит на каналы RGB. Альфа-канал не используется.
PixelFormat64bppARGB	64	По 16 бит на каждый канал ARGB.
PixelFormat64bppPARGB	64	По 16 бит на каждый канал ARGB. Улучшен механизм смешивания компонент RGB и альфа-канала.

Таблица 35.3. Флаги, возвращаемые функцией GetFlags( )

Константа	Значение	Описание флага
ImageFlagsNone	h0	Флаги не установлены.
ImageFlagsScalable	h0001	Изображение может масштабироваться.
ImageFlagsHasAlpha	h0002	Наличие альфа-канала.
ImageFlagsHasTranslucent	h0004	Изображение прозрачно (величина альфа-канала находится между 0 и 255).
ImageFlagsPartiallyScalable	h0008	Изображение масштабируемо с некоторыми ограничениями.
ImageFlagsColorSpaceRGB	h0010	Цветовое пространство RGB.
ImageFlagsColorSpaceCMYK	h0020	Цветовое пространство CMYK.
ImageFlagsColorSpaceGRAY	h0040	Градации серого.

Константа	Значение	Описание флага
ImageFlagsColorSpaceYCBCR	h0080	Цветовое пространство YCbCr.
ImageFlagsColorSpaceYCCK	h0100	Цветовое пространство YCCK.
ImageFlagsHasRealDPI	h1000	Имеется информация о реальном разрешении.
ImageFlagsHasRealPixelSize	h2000	Имеется информация о размере пиксела.
ImageFlagsReadOnly	h00010000	Только для чтения.
ImageFlagsCaching	h00020000	Изображение может быть кэшировано для ускорения доступа.

Понимаю, что рискую вас немного утомить, но тем не менее отважусь предложить читателю код процедуры получения основных характеристик изображения. Полученные данные построчно передаются в список, идентифицируемый дескриптором LISTBOX.

```
Procedure GetImageInfo(GPImage :TGPImage; LISTBOX :  HWnd);
var S:String;
    G: TGUID;
    X,Y:real;
    SrcRect: TGPRectF;
    SrcUnit: TUnit;
begin
{1 - вид изображения}
  CASE GPImage.GetType OF
   ImageTypeUnknown: S:='Вид изображения неизвестен';
   ImageTypeBitmap : S:='Растровое изображение';
   ImageTypeMetafile: S:='Векторный рисунок';
  END;
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
{2 - по GUID узнаем формат рисунка}
  GPImage.GetRawFormat(G);
  S:='GUID объекта: '+GUIDToString(G);
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
  S:='Формат: ';
  IF IsEqualGUID(G,ImageFormatUndefined) THEN S:=S+'неизвестен' ELSE
  IF IsEqualGUID(G,ImageFormatMemoryBMP)
                      THEN  S:=S+'битовый образ в памяти' ELSE
  IF IsEqualGUID(G,ImageFormatBMP)       THEN  S:=S+'BMP'  ELSE
  IF IsEqualGUID(G,ImageFormatEMF)       THEN  S:=S+'EMF'  ELSE
  IF IsEqualGUID(G,ImageFormatWMF)       THEN  S:=S+'WMF'  ELSE
  IF IsEqualGUID(G,ImageFormatJPEG)      THEN  S:=S+'JPEG' ELSE
  IF IsEqualGUID(G,ImageFormatPNG)       THEN  S:=S+'PNG'  ELSE
  IF IsEqualGUID(G,ImageFormatGIF)       THEN  S:=S+'GIF'  ELSE
  IF IsEqualGUID(G,ImageFormatTIFF)      THEN  S:=S+'TIFF' ELSE
  IF IsEqualGUID(G,ImageFormatEXIF)      THEN  S:=S+'EXIF' ELSE
  IF IsEqualGUID(G,ImageFormatIcon)      THEN  S:=S+'ICON';

  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));

{3 - размер в пикселах}
```

```
  S:=Format('Размер в пикселах: %d x %d',[GPImage.GetWidth,GPImage.GetHeight]);
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
{4 - разрешение}
  S:=Format('Разрешение по вертикали:
             1/%f дюйм/пикс.',[GPImage.GetVerticalResolution]);
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
  S:=Format('Разрешение по горизонтали:
             1/%f дюйм/пикс.',[GPImage.GetHorizontalResolution]);
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
{6 - размер в дюймах и сантиметрах}
  X:=GPImage.GetWidth/GPImage.GetHorizontalResolution;
  Y:=GPImage.GetHeight/GPImage.GetVerticalResolution;
  S:=Format('Размер в дюймах: %f x %f',[X,Y]);
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
  S:=Format('Размер в сантиметрах: %f x %f',[X*2.54,Y*2.54]);
  SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
{7 - глубина цвета}
S:='Глубина цвета: ';
CASE GPImage.GetPixelFormat OF
   PixelFormat1bppIndexed   : S:='Монохромное изображение';
   PixelFormat4bppIndexed   : S:=S+'4 бита';
   PixelFormat8bppIndexed   : S:=S+'8 бит';
   PixelFormat16bppARGB1555 : S:=S+'16 бит + альфа-канал';
   PixelFormat16bppGrayScale: S:=S+'градации серого16 бит';
   PixelFormat16bppRGB555   :S:=S+'16 бит (R=5,G=5,B=5)';
   PixelFormat16bppRGB565   :S:=S+'16 бит (R=5,G=6,B=5)';
   PixelFormat24bppRGB      :S:=S+'24 бит';
   PixelFormat32bppARGB     :S:=S+'32 бит + альфа-канал';
   PixelFormat32bppPARGB    :S:=S+'32 бит + смешивание с альфа-каналом';
   PixelFormat32bppRGB      :S:=S+'32 бит без альфа-канала';
   PixelFormat48bppRGB      :S:=S+'48 бит без альфа-канала';
   PixelFormat64bppARGB     :S:=S+'64 бит + альфа-канал';
   PixelFormat64bppPARGB    :S:=S+'64 бит + смешивание с альфа-каналом';
END;
 SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
end;
```

На рис. 35.2 (см. цветную вклейку, с. 422) представлено окно программы с результатом исследования файла из библиотеки к этой книге, размещенной на сайте издательства.

Работа с палитрой цветов

Главное техническое отличие палитры цветов GDI+ от палитры, применявшейся в GDI, заключается в поддержке альфа-канала. Вследствие этого каждый элемент палитры GDI+ кроме цветовых составляющих RGB должен уметь хранить 8 бит канала прозрачности. В структуру палитры также внесены некоторые правки:

```
type TColorPalette = packed record
    Flags  : UINT ;                 // флаги палитры
```

```
    Count  : UINT ;                  // число элементов
    Entries: array [0..0] of ARGB;  // цветовая таблица
  end;
```

Как видите, число полей осталось таким же, как и в описывающей палитру в GDI записи TLogPalette. Но два поля из трех затронули функциональные изменения. В структуре TColorPalette первое поле хранит не номер версии палитры, а информационные флаги. В комбинации могут участвовать три флага:

```
PaletteFlagsHasAlpha  = $0001;{наличие альфа-составляющей}
PaletteFlagsGrayScale = $0002;{признак оттенков серого}
PaletteFlagsHalftone  = $0004;{палитра с равномерным распределением цветов}
```

Второго поля реформы не коснулись, и оно по-прежнему содержит число элементов в палитре. Третье поле хотя и сохранило свое назначение (ссылка на динамический массив, содержащий перечень цветов изображения), но не смогло уберечь себя от дуновений технического прогресса. Теперь его элементы описываются 32-битной структурой ARGB, а не записью TPaletteEntry, как это принято в древнем GDI.

Для доступа к полутоновой палитре холста GDI+ воспользуйтесь методом:

```
function GetHalftonePalette: HPALETTE;
```

Функция возвращает дескриптор палитры.

Начиная работу с современной версией палитры, программист поинтересуется ее размером:

```
function GetPaletteSize: Integer;
```

Функция возвращает число байт, которые необходимо выделить для размещения в ней структуры TColorPalette и таблицы цветов. Распределив память для палитры, обращаемся к методу:

```
function GetPalette(pPalette: PColorPalette; Size: Integer): TStatus;
```

Здесь pPalette – указатель на палитру; Size – размер палитры. Ниже представлен пример, демонстрирующий порядок работы с этими методами.

```
Procedure GDIPLUS_PAINT(DC:HDC;GPImage :TGPImage);
var GPGraphics:TGPGraphics;
    Size,I,EntriesCount:Cardinal;
    GPRect:TGPRectF;
    pPalette: PColorPalette;
    GPBrush:TGPSolidBrush;
begin
    GPGraphics:=TGPGraphics.Create(DC);
    Size:=GPImage.GetPaletteSize;
    GetMem(pPalette,Size);
    GPImage.GetPalette(pPalette,Size);
{узнаем число элементов}
    EntriesCount:=1+(Size-SizeOf(TColorPalette)) SizeOf(ARGB);

    GPBrush:=TGPSolidBrush.Create(aclBlack);
```

```
    GPRect:=MakeRect(380,20,16,16.0);
{выводим палитру}
    for I:=0 to EntriesCount-1 do
      begin
      GPBrush.SetColor(pPalette^.Entries[I]);
      GPGraphics.FillRectangle(GPBrush,GPRect);
      GPRect.X:=GPRect.X+18;
      If GPRect.X>380+15*18 then
         begin {новый ряд ячеек}
          GPRect.X:=380;
          GPRect.Y:=GPRect.Y+18;
         end;
      end;

   GPGraphics.Free; GPBrush.Free;
   FreeMem(pPalette,Size)
end;
```

Исследуемый рисунок содержит цветовую таблицу, состоящую из 256 элементов. Получив затребованные данные, мы выводим палитру рисунка в виде матрицы с размерностью 16×16 элементов (см. рис. 35.3 на цв. вклейке, с. 422).

Факт существования палитры цветов значительно упрощает процесс применения к изображению различного рода цветовых эффектов. Например, для повышения уровня той или иной цветовой составляющей достаточно обработать данные палитры, не затрагивая сам рисунок. Для изменения элементов цветовой палитры следует воспользоваться методом:

```
function SetPalette(pPalette: PColorPalette): TStatus;
```

Параметр `pPalette` должен быть указателем на новую палитру. Ниже предложена функция, позволяющая управлять степенью прозрачности и глубиной цвета всех трех компонентов RGB для содержащего палитру изображения `GPImage`. В функцию требуется передать приращения для каждого элемента ARGB.

```
Function ChangePalette(var GPImage :TGPImage;
                       A_Offset,R_Offset,G_Offset,B_Offset:Smallint):TStatus;
var pPalette : PColorPalette;
    Size,I,EntriesCount:Cardinal;
    A,R,G,B:byte;
begin
Size:=GPImage.GetPaletteSize;
GetMem(pPalette,Size);
GPImage.GetPalette(pPalette,Size);
EntriesCount:=1+(Size-SizeOf(TColorPalette)) DIV SizeOf(ARGB);

    for I:=0 to EntriesCount-1 do
      begin
       A:=GetAlpha(pPalette^.Entries[I]);
       R:=GetRed(pPalette^.Entries[I]);
       G:=GetGreen(pPalette^.Entries[I]);
       B:=GetBlue(pPalette^.Entries[I]);
```

```
{Альфа-канал}
       IF A_Offset>0 then
            IF A+A_Offset>255 Then A:=255 ELSE A:=A+A_Offset
       else IF A+A_Offset<0   Then A:=0   ELSE A:=A+A_Offset;
{Красный цвет}
       IF R_Offset>0 then
            IF R+R_Offset>255 Then R:=255 ELSE R:=R+R_Offset
       else IF R+R_Offset<0   Then R:=0   ELSE R:=R+R_Offset;
{Зеленый цвет}
       IF G_Offset>0 then
            IF G+G_Offset>255 Then G:=255 ELSE G:=G+G_Offset
       else IF G+G_Offset<0   Then G:=0   ELSE G:=G+G_Offset;
{Синий цвет}
       IF B_Offset>0 then
            IF B+B_Offset>255 Then B:=255 ELSE B:=B+B_Offset
       else IF B+B_Offset<0   Then B:=0 ELSE B:=B+B_Offset;
{Изменяем цвет в ячейке таблицы цветов}
       pPalette^.Entries[I]:=MakeColor(A,R,G,B);
      end;
Result:=GPImage.SetPalette(pPalette);
FreeMem(pPalette,Size);
end;
```

Во время перебора элементов цветовой палитры осуществляется контроль за тем, чтобы новое значение не вышло за допустимые пределы значений 0...255. На рис. 35.4 (см. цветную вклейку, с. 423) представлены четыре вариации одного и того же рисунка, отличающиеся друг от друга набором цветов в палитре. Этот же рисунок вы сможете найти среди примеров в библиотеке к этой книге на сайте издательства.

Поворот изображения

Поворот изображения производится с помощью метода:

```
function RotateFlip(RotateFlipType: TRotateFlipType): TStatus;
```

В качестве единственного параметра в функцию передается значение поворота.

```
type TRotateFlipType = (RotateNoneFlipNone=0, Rotate90FlipNone=1,
Rotate180FlipNone=2,Rotate270FlipNone=3,RotateNoneFlipX=4,
Rotate90FlipX=5, Rotate180FlipX=6, Rotate270FlipX=7,
RotateNoneFlipY=Rotate180FlipX, Rotate90FlipY=Rotate270FlipX,
Rotate180FlipY=RotateNoneFlipX, Rotate270FlipY=Rotate90FlipX,
RotateNoneFlipXY=Rotate180FlipNone,
Rotate90FlipXY=Rotate270FlipNone,
Rotate180FlipXY=RotateNoneFlipNone,
Rotate270FlipXY= Rotate90FlipNone);
```

Константы перечисления вряд ли нуждаются в дополнительных комментариях, их названия говорят сами за себя. Числовое значение определяет угол поворота, а наличие слова «Flip» и названия осей (X или Y) указывает на то, что будет произведен переворот изображения вокруг оси.

Многокадровые изображения

Программистами разработано несколько форматов графических объектов, способных содержать в одном файле несколько изображений. Подобные изображения-контейнеры допустимо классифицировать по размерам хранящихся в них рисунков. Первый класс многокадровых изображений составляют файлы, объединяющие несколько картинок одинакового разрешения. Последовательное воспроизведение подобных кадров (фреймов) позволяет получить анимированную картинку. Объекты такого рода обычно называют многостраничными – «*multiple-page image*». К классическим представителям многостраничных форматов относится GIF. Ко второму классу принадлежат файлы, хранящие рисунки с различным разрешением. Как правило, в них находятся основное изображение и его уменьшенная копия. Такой подход весьма эффективен при обмене данными в низкоскоростных сетях – пользователь может просмотреть рисунок, не дожидаясь его полной загрузки. В англоязычной литературе такие изображения называют «*multiple-resolution image*» – изображения с разным разрешением. В качестве примера формата, поддерживающего мультиразрешение, стоит привести TIFF.

В перечисленных многокадровых контейнерах с изображениями могут находиться три типа фреймов:

1. *Фрейм-страница*. Ему соответствует константа `FrameDimensionPage` с GUID `{7462DC86-6180-4C7E-8E3F-EE7333A7A483}`.
2. *Фрейм анимации*. Константа `FrameDimensionTime`, GUID `{6AEDBD6D-3FB5-418A-83A6-7F45229DC872}`.
3. *Фрейм с другим разрешением*. Константа `FrameDimensionResolution`, GUID `{84236F7B-3BD3-428F-8DAB-4EA1439CA315}`.

Для того чтобы выяснить, сколько типов фреймов хранится в контейнере, обращаемся к методу:

```
function GetFrameDimensionsCount: UINT;
```

Если все фреймы одинаковы, то на выходе функции окажется единица, если же имеются разнотипные фреймы – получим двойку или тройку.

Исследование продолжает метод:

```
function GetFrameDimensionsList(pDimensionIDs: PGUID; Count: UINT): TStatus;
```

Он позволит построить список типов фреймов и вернуть его через параметр-указатель `pDimensionIDs`. Второй параметр функции не что иное, как результат работы метода `GetFrameDimensionsCount()`.

Сколько именно фреймов определенного типа размещено в контейнере, расскажет метод:

```
function GetFrameCount(const DimensionID: TGUID): UINT;
```

Единственный параметр метода – тип кадра. Порядок применения всех трех функций представлен в рамках процедуры `GetMultuFrameInfo()`.

```
Procedure GetMultuFrameInfo(GPImage:TGPImage;LISTBOX:HWND);
var Count,DimensionCount,X:Cardinal;
    S:String;
    DimensionIDs: Array of TGUID;
begin
DimensionCount:=GPImage.GetFrameDimensionsCount;
SetLength(DimensionIDs, DimensionCount);
GPImage.GetFrameDimensionsList(@DimensionIDs[0], DimensionCount);
    For X:=0 to DimensionCount -1 do
    begin
    IF IsEqualGUID(DimensionIDs[X],FrameDimensionTime)
          Then S:='Фрейм анимации' ELSE
    IF IsEqualGUID(DimensionIDs[X],FrameDimensionResolution)
          Then S:='Фрейм с другим разрешением' ELSE
    IF IsEqualGUID(DimensionIDs[X],FrameDimensionPage)
          Then S:='Фрейм-страница';
    SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(S+#0)));

    Count:=GPImage.GetFrameCount(DimensionIDs[X]);
    S:='Число фреймов: '+IntToStr(Count);
    SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(S+#0)));
    end;
end;
```

Процедура исследует хранящееся в объекте GPImage изображение и передает результаты в элемент-список LISTBOX.

Процесс создания многокадрового изображения рассмотрен в подразделе «Сохранение многокадровых изображений» этой главы.

При работе с многокадровыми изображениями существенный интерес представляет метод, позволяющий активировать заданный фрейм:

```
function SelectActiveFrame(const DimensionID: TGUID; FrameIndex: UINT): TStatus;
```

Выбор кадра определяется двумя параметрами: типом DimensionID и номером фрейма FrameIndex. Указанная функция может стать основой кода, анимирующего рисунок.

Сохранение изображения

Одно из существенных достоинств объекта TGPImage в том, что обслуживаемое им изображение может быть не просто сохранено под другим именем, но и конвертировано в любой поддерживаемый GDI+ формат. Существуют две вариации одного и того же метода сохранения изображения:

```
function Save(Filename: WideString; const ClsidEncoder: TGUID;
              pEncoderParams: PEncoderParameters = nil): TStatus; overload;
function Save(Stream: IStream; const ClsidEncoder: TGUID;
              pEncoderParams: PEncoderParameters  = nil): TStatus; overload;
```

Отличие скрывается в первом параметре перегружаемого метода: Filename – имя файла или Stream – область в памяти, в которую будет сохранено изображение. Второй параметр ClsidEncoder – константа с GUID кодера, используе-

мого для сохранения данных. Заключительный параметр метода определяет особенности операции – параметры сохранения; если его не задействовать, параметры заполняются значениями по умолчанию.

Прежде чем мы перейдем к примеру, демонстрирующему процесс сохранения изображения, пройдем одну подготовительную ступень – разработаем функцию, позволяющую выяснить GUID кодера по расширению имени файла. Для этого функция вооружена двумя параметрами: расширением `FileExt` и уникальным идентификатором `EncoderCLSID`.

```
Function GetEncoder(FileExt:string; out EncoderCLSID:TGUID):TStatus;
var NumEncoders, Size, I: UINT;
    ICI:Array of TImageCodecInfo;
begin
Result:=UnknownImageFormat;
GetImageEncodersSize(NumEncoders, Size);
SetLength(ICI,Size DIV SizeOf(TImageCodecInfo));
GetImageEncoders(NumEncoders, Size,PImageCodecInfo(ICI));
For i:=0 to NumEncoders-1 do
  IF POS(UPPERCASE(FileExt),ICI[i].FilenameExtension)>0 then
  begin
    EncoderCLSID:=ICI[i].Clsid;
    Result:=OK;
    Break;
  end;
end;
```

При выяснении идентификатора GUID кодека вместо расширения файла можно воспользоваться мнемоническим идентификатором формата (см. поле `MIME` *структуры* `TImageCodecInfo`*). В GDI+ их всего пять: «Image/bmp», «Image/jpeg», «Image/gif», «Image/tiff» и «Image/png».*

Теперь все готово для конвертирования изображения в любой имеющийся в нашем распоряжении формат.

```
var GPImage:TGPImage; EncoderCLSID:TGUID;
begin
 GPImage :=TGPImage.Create('demo.bmp');
 IF GetEncoder('TIF',EncoderCLSID)=OK Then
               GPImage.Save('c:\new.tif',EncoderCLSID);
 GPImage.Free;
end;
```

Параметры кодирования изображения

В простейшем случае для сохранения изображения в определенном графическом формате программисту достаточно указать идентификатор кодека и не задействовать параметр `pEncoderParams` метода `Save()`. В этом случае кодек инициализируется параметрами по умолчанию и, как следствие, мы лишимся привилегий управлять такими важными характеристиками изображения, как, например, степень сжатия и глубина цвета. Для устранения не-

справедливости стоит разобраться с порядком определения параметров формата, и свою работу мы начнем с рассмотрения списка параметров кодека – структуры TEncoderParameters:

```
type TEncoderParameters = packed record
    Count     : UINT; // число параметров в этой структуре
    Parameter : array[0..0] of TEncoderParameter;  // значения параметров
end;
```

Как видите, это обычный список, состоящий из двух полей. В первом поле Count хранится число параметров, описываемых в массиве Parameter. В свою очередь отдельный элемент массива соответствует структуре:

```
type TEncoderParameter = packed record
    Guid           : TGUID;   // GUID параметра (см. таблицу 35.5)
    NumberOfValues : ULONG;   // число значений параметра
    Type_          : ULONG;   // тип параметра (см. таблицу 35.6)
    Value          : Pointer; // указатель на данные
end;
```

В зависимости от формата графического файла перечень задействованных параметров изменяется. Например, при обслуживании JPEG можно обойтись только EncoderTransformation и EncoderQuality.

Таблица 35.4. Вид параметров в структуре TEncoderParametr

Константа	GUID	Описание
EncoderCompression	{E09D739D-CCD4-44EE-8EBA-3FBF8BE4FC58}	Степень сжатия.
EncoderColorDepth	{66087055-AD66-4C7C-9A18-38A2310B8337}	Глубина цвета.
EncoderScanMethod	{3A4E2661-3109-4E56-8536-42C156E7DCFA}	Метод сканирования.
EncoderVersion	{24D18C76-814A-41A4-BF53-1C219CCCF797}	Версия кодека.
EncoderRenderMethod	{6D42C53A-229A-4825-8BB7-5C99E2B9A8B8}	Метод прорисовки.
EncoderQuality	{1D5BE4B5-FA4A-452D-9CDD-5DB35105E7EB}	Качество кодирования.
EncoderTransformation	{8D0EB2D1-A58E-4EA8-AA14-108074B7B6F9}	Различного рода преобразования изображения, например повороты.
EncoderLuminanceTable	{EDB33BCE-0266-4A77-B904-27216099E717}	Таблица яркости (в цветовой модели YCbCr – компонента Y).
EncoderChrominanceTable	{F2E455DC-09B3-4316-8260-676ADA32481C}	Таблица цветности (в цветовой модели YCbCr – компоненты Cb и Cr).
EncoderSaveFlag	{292266FC-AC40-47BF-8CFC-A85B89A655DE}	Флаг сохранения.
CodecIImageBytes	{025D1823-6C7D-447B-BBDB-A3CBC3DFA2FC}	Интерфейс заблокированных/разблокированных байт изображения.

Что качается типа параметра (поле type), то, на мой взгляд, разработчики GDI+ несколько перемудрили, в особенности с формой представления вещественных чисел (табл. 35.5).

Таблица 35.5. Типы параметров в структуре TEncoderParametr

Константа	Значение	Описание
EncoderParameterValueTypeByte	1	Байт.
EncoderParameterValueTypeASCII	2	Текст в формате ASCII.
EncoderParameterValueTypeShort	3	Короткое неотрицательное целое (16 бит).
EncoderParameterValueTypeLong	4	Длинное неотрицательное целое (32 бит).
EncoderParameterValueTypeRational	5	Массив из 2 длинных неотрицательных целых. Нулевой элемент представляет собой числитель, первый – знаменатель.
EncoderParameterValueTypeLongRange	6	Массив из 2 длинных неотрицательных целых для хранения диапазона значений. Нулевой элемент соответствует минимальному числу, первый – максимальному числу в диапазоне.
EncoderParameterValueTypeUndefined	7	Универсальный массив байт.
EncoderParameterValueTypeRationalRange	8	Массив из 4 длинных неотрицательных целых. Массив описывает диапазон, границы которого определяют вещественные числа. Минимальное значение = элемент_0 /элемент_1, максимальное = элемент_2/элемент_3.

Как узнать, какие параметры имеются у того или иного кодека? Ответ на этот вопрос предоставят два метода из арсенала класса TGPImage. Как всегда, все самое интересное начинается с выделения памяти под данные. Объем буфера выясняется при помощи метода:

```
function GetEncoderParameterListSize(const ClsidEncoder: TGUID): UINT;
```

В качестве параметра в метод передается GUID анализируемого кодека. Второй метод возвратит все параметры кодека:

```
function GetEncoderParameterList(const ClsidEncoder: TGUID; Size: UINT;
                                 pBuffer: pEncoderParameters): TStatus;
```

Результат работы окажется в буфере pBuffer (перед обращением к функции не забудьте выделить для него память), это указатель на структуру TEncoderParameters.

Процесс распознавания задействованных в кодере параметров – не столько сложная, сколько громоздкая задача.

```
Procedure GetEncoderParams(GPImage:TGPImage;
                           EncoderClsid: TGUID; LISTBOX :  HWnd);
type TArr1_2_7=Array of Byte; {все типы параметров}
     TArr3    =Array of WORD;
     TArr4    =Array of CARDINAL;
     TArr5_6  =Array[0..0,0..1] of CARDINAL;
     TArr8    =Array[0..0,0..3] of CARDINAL;
var Size,I,K:Cardinal;  S:String;
    pEP:pEncoderParameters; EParameter:TEncoderParameter;
BEGIN
Size:=GPImage.GetEncoderParameterListSize(EncoderClsid);

IF Size<=0 then
  Begin
  {если размер нулевой, то прекращаем обработку}
  SendMessage(LISTBOX,LB_ADDSTRING,0, INTEGER(pChar('Параметры кодера: НЕТ'+#0)));
  Exit;
  end;

GetMem(pEP,Size);
GPImage.GetEncoderParameterList(EncoderClsid,Size,pEncoderParameters(pEP));
FOR I:=0 TO pEP^.Count-1 DO //перебор всех параметров
   BEGIN
   EParameter:=pEP^.Parameter[i];
   //ВЫВОДИМ НАЗВАНИЯ КОНСТАНТ ПАРАМЕТРОВ
   IF IsEqualGUID(EParameter.Guid,EncoderCompression) then
      S:='EncoderCompression= -> '
   ELSE
{…
И так далее сверяем все GUID параметров до CodecIImageBytes
…}
   ELSE
   IF IsEqualGUID(EParameter.Guid,CodecIImageBytes) then
      S:='CodecIImageBytes -> ';

//перебор значений параметров
   IF EParameter.Value<>nil then
     For K:=0 TO EParameter.NumberOfValues-1 do
       CASE EParameter.Type_ OF
        1: S:=S+IntToStr(TARR1_2_7(EParameter.Value)[k])+';';
        2: S:=S+CHAR(TARR1_2_7(EParameter.Value)[k]);
        3: S:=S+IntToStr(TArr3(EParameter.Value)[k])+';';
        4: S:=S+IntToStr(TArr4(EParameter.Value)[k])+';';
        5: S:=S+Format('%f;',[TArr5_6(EParameter.Value^)[k,0]/
                              TArr5_6(EParameter.Value^)[k,1]]);
        6: S:=S+Format('диапазон %d-%d',[TArr5_6(EParameter.Value^)[k,0],
                                         TArr5_6(EParameter.Value^)[k,1]]);
        7: S:=S+IntToHex(TARR1_2_7(EParameter.Value)[k],2)+';';
        8: S:=S+Format('диапазон %f - %f',
```

```
            [TArr8(EParameter.Value^)[k,0]/TArr8(EParameter.Value^)[k,1],
             TArr8(EParameter.Value^)[k,2]/TArr8(EParameter.Value^)[k,3]]);
        end;
        SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar('        '+S+#0)));

    END;
    FreeMem(pEP,Size);
END;
```

В секции объявления типов мы декларируем перечень типизированных массивов, соответствующий типам из табл. 35.5. Затем, узнав число параметров, перебираем их в цикле `FOR I:=0 TO pEP^.Count-1 DO`. Внутри имеется еще один вложенный цикл `For K:=0 TO EParameter.NumberOfValues-1 DO`, он предназначен для получения значений параметров. На рис. 35.5 представлен экранный снимок окна с параметрами кодера JPEG, полученный с помощью нашей процедуры.

По рис. 35.5 видно, что JPEG позволяет управлять качеством результирующего изображения (параметр `EncoderQuality`), коэффициент качества должен находиться в диапазоне значений от 0 до 100. Технически это не что иное, как сжатие изображения: 100 – сжатие отсутствует, 0 – сжатие максимально. Попробуем поработать с параметром `EncoderQuality`. Для этого разработаем функцию, позволяющую конвертировать изображения из любого из поддерживаемых в GDI+ форматов в формат JPEG с заданной степенью сжатия:

```
Function SaveImageAsJPEG(GPImage:TGPImage; FileName:WideString;
                         var Quality:Cardinal):TStatus;
var FileExt:string;
    EncoderParams: TEncoderParameters;
    EncoderCLSID:TGUID;
begin
Result:=GenericError;
IF GetEncoder('JPEG',EncoderCLSID)=OK Then
begin
  EncoderParams.Count:=1;
  EncoderParams.Parameter[0].Guid:=EncoderQuality;
  EncoderParams.Parameter[0].Type_:=EncoderParameterValueTypeLong;
```

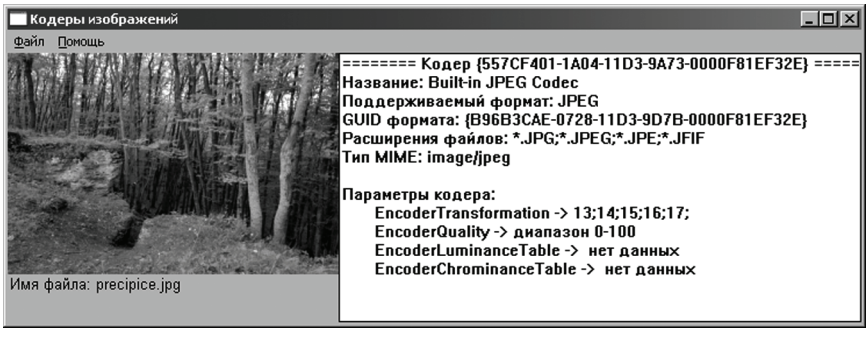


Рис. 35.5. Параметры кодера JPEG

```
    EncoderParams.Parameter[0].NumberOfValues:=1;
    {диапазон качества: от 0 до 100}
    If Quality>100 then Quality:=100;
    EncoderParams.Parameter[0].Value:=@Quality;
    Result:=GPImage.Save(FileName,EncoderCLSID,@EncoderParams);
  end;
  end;
```

Для получения GUID кодека в функции задействована выдуманная нами ранее функция `GetEncoder()`. Затем мы описываем отвечающий за сжатие изображения параметр и обращаемся методу `Save()`.

В JPEG сжатие осуществляется с потерей качества. Значение 100 соответствует минимальной степени сжатия (и, соответственно, наименьшим потерям), тогда как значение 0 максимально уменьшит размер файла и одновременно максимально ухудшит изображение.

Вновь вернемся к рис. 35.5. Обратите внимание на параметр `EncoderTransformation`. Если вы еще не забыли – он определяет, какие преобразования поддерживает формат. Согласитесь, что перечень значений 13..17 мало объясняет суть трансформаций. Давайте разбираться, а начнем с того, что я раскрою очередной секрет. Помимо числовых значений, например, определяющих степень сжатия JPEG-картинки, параметр кодера может содержать заранее предопределенные значения. Они описаны в модуле GDIAPI в виде перечисления `TEncoderValue`:

```
Type TEncoderValue =
    (EncoderValueColorTypeCMYK,             EncoderValueColorTypeYCCK,
     EncoderValueCompressionLZW,            EncoderValueCompressionCCITT3,
     EncoderValueCompressionCCITT4,         EncoderValueCompressionRle,
     EncoderValueCompressionNone,           EncoderValueScanMethodInterlaced,
     EncoderValueScanMethodNonInterlaced,   EncoderValueVersionGif87,
     EncoderValueVersionGif89,              EncoderValueRenderProgressive,
     EncoderValueRenderNonProgressive,      EncoderValueTransformRotate90,
     EncoderValueTransformRotate180,        EncoderValueTransformRotate270,
     EncoderValueTransformFlipHorizontal,   EncoderValueTransformFlipVertical,
     EncoderValueMultiFrame,                EncoderValueLastFrame,
     EncoderValueFlush,                     EncoderValueFrameDimensionTime,
     EncoderValueFrameDimensionResolution,  EncoderValueFrameDimensionPage);
```

Назначение констант легко понять, для этого надо лишь внимательно взглянуть на названия. Посему не станем заниматься расшифровкой имен, а вновь обратимся к таинственным значениям 13..17, которые возвратила наша программа. Для того чтобы понять, что это такое, в перечислении `TEncoderValue` отсчитайте константы с порядковыми номерами 13, 14, 15, 16 и 17 (не забывайте, что счет начинается с нуля). Ими окажутся:

1. `EncoderValueTransformRotate90` {поворот изображения на 90 градусов}.
2. `EncoderValueTransformRotate180` {поворот изображения на 180 градусов}.
3. `EncoderValueTransformRotate270` {поворот изображения на 270 градусов}.
4. `EncoderValueTransformFlipHorizontal` {переворот по горизонтальной оси}.
5. `EncoderValueTransformFlipVertical` {переворот по вертикальной оси}.

Таким образом, мы вычислили, что JPEG способен вращать изображение. Вот так, просто и со вкусом, разработчики GDI+ доказали, что они не лыком шиты и могут закрутить сюжет не менее лихо, чем Агата Кристи в своих детективах.

Сохранение многокадровых изображений

Для создания многокадрового изображения применяется перегружаемый метод SaveAdd(). В этом случае вновь приходится говорить о (судя по всему, проблемному для разработчиков GDI+) вопросе выдумки имен для своих функций и методов. Дело в том, что в GDI+ достаточно часто под одним и тем же названием встречаются методы, решающие разноплановые задачи. Судите сами, метод:

```
function SaveAdd(NewImage: TGPImage;
                 EncoderParams: PEncoderParameters): TStatus; overload;
```

предназначен для добавления новых фреймов к файлу либо потоку. В свою очередь метод:

```
function SaveAdd(EncoderParams: PEncoderParameters): TStatus; overload;
```

позволяет решить две задачи:

1. Сохранить активный фрейм из многокадрового изображения в файл (либо в поток).
2. Уведомить объект TGPImage о завершении работы над многокадровым изображением.

Что ж, пусть будет так. Когда мы с вами разработаем свою графическую библиотеку, вообще все методы назовем одинаково – пусть все остальные удивляются...

Ладно, возвратимся к SaveAdd(). Кроме названия объединяющей чертой обеих функций выступает указатель на параметры кодировки EncoderParams. С его помощью программист определяет порядок кодировки результирующего изображения. Кроме того, в первом методе имеется параметр NewImage. Это контейнер, содержащий добавляемый фрейм.

Добавление фреймов должно производиться только в предварительно сохраненное изображение.

Изучите листинг, демонстрирующий порядок создания многокадрового изображения из трех файлов:

```
var EncoderCLSID:TGUID;
    EncoderParams: TEncoderParameters;
    ParameterValue:TEncoderValue;
    MultiImage,TempImage:TGPImage;
begin
GetEncoder('TIF',EncoderCLSID);
  EncoderParams.Count:=1;
  EncoderParams.Parameter[0].Guid := EncoderSaveFlag;
```

```
  EncoderParams.Parameter[0].Type_:=EncoderParameterValueTypeLong;
  EncoderParams.Parameter[0].NumberOfValues:=1;
  EncoderParams.Parameter[0].Value:= @ParameterValue;

MultiImage:=TGPImage.Create('00.tif'); {нулевой кадр}
ParameterValue := EncoderValueMultiFrame; {1 изменение значения}
MultiImage.Save('MultiFrame.tif', EncoderClsid, @EncoderParams);

   ParameterValue := EncoderValueFrameDimensionPage; {2 изменение}
   TempImage:=TGPImage.Create('01.tif'); {первый кадр}
   MultiImage.SaveAdd(TempImage, @EncoderParams);
   TempImage.Free;

   TempImage:=TGPImage.Create('02.tif'); {второй кадр}
   MultiImage.SaveAdd(TempImage, @EncoderParams);
   TempImage.Free;

ParameterValue := EncoderValueFlush; {3 изменение}
MultiImage.SaveAdd(@EncoderParams);

MultiImage.Free;
end;
```

Обратите внимание на то, что во время выполнения программы мы трижды поменяли значение, содержащееся в поле `Value` параметра `EncoderParams.Parameter[0]`. Впервые это произошло перед сохранением нулевого кадра из объекта `MultiImage` на жесткий диск. В этот момент в `Value` устанавливается значение `EncoderValueMultiFrame`, указывающее кодеку на то, что мы создаем многостраничное изображение. Второе изменение значения поля `Value` происходит перед началом добавления к изображению остальных фреймов, теперь в нем установлено значение `EncoderValueFrameDimensionPage`, говорящее, что к созданному ранее изображению следует добавить очередной кадр. Заключительная перемена в настроении поля `Value` происходит на последних строках листинга, теперь здесь хранится значение `EncoderValueFlush`. Таким образом мы завершаем работу по формированию многостраничного изображения.

Роль хоста при выводе рисунков, метод DrawImage

Пальму первенства по части числа перегружаемых версий функции вне всякого сомнения держит метод `DrawImage()`. Не знаю, ведает ли об этом Книга рекордов Гиннесса, но в случае с `DrawImage()` разработчики GDI+ побили все рекорды и написали целых 18 разновидностей метода с идентичным названием!

Галерею знаменитостей открывают простейшие методы, для нормального самочувствия которых достаточно минимального набора параметров:

```
function DrawImage(Image: TGPImage; X, Y: Integer): TStatus; overload;
function DrawImage(Image: TGPImage; X, Y: Single): TStatus; overload;
function DrawImage(Image: TGPImage; const Point: TGPPointF): TStatus; overload;
function DrawImage(Image: TGPImage; const Point: TGPPoint): TStatus; overload;
```

Кроме собственно рисунка (параметр `Image`) функции рассчитывают получить координаты точки места вывода рисунка. Минимальный набор аргу-

ментов свидетельствует и о минимальных возможностях методов – они даже не способны изменить размер рисунка.

При выводе образа с помощью простейшей версии метода `DrawImage()` *можно столкнуться с проблемой некорректного определения размеров рисунков с разрешением, отличающимся от разрешения экрана.*

Для управления пропорциями изображения введен еще один квартет одноименных методов:

```
function DrawImage(Image: TGPImage;
                   X, Y, Width, Height: Integer): TStatus; overload;
function DrawImage(Image: TGPImage;
                   X, Y, Width,Height: Single): TStatus; overload;
function DrawImage(Image: TGPImage; const Rect: TGPRectF): TStatus; overload;
function DrawImage(Image: TGPImage; const Rect: TGPRect): TStatus; overload;
```

Но теперь вместо банальной координаты верхней левой точки нам предоставлена возможность определиться с конкретной прямоугольной областью вывода. Здесь имеются параметры на любой вкус: от одного-единственного `Rect` до вереницы `X`, `Y`, `Width`, `Height`...

Понимаю, что шутка с одинаковыми именами методов явно затянулась, но вывод изображения с предоставлением дополнительной услуги (обрезки лишних краев) производит все тот же метод `DrawImage`:

```
function DrawImage(Image: TGPImage;
                   X, Y, SrcX, SrcY, SrcWidth, SrcHeight: Integer;
                   SrcUnit: TUnit): TStatus; overload;
function DrawImage(Image: TGPImage;
                   X, Y, SrcX, SrcY, SrcWidth, SrcHeight: Single;
                   SrcUnit: TUnit): TStatus; overload;
```

Для того чтобы мы с вами лишились части рисунка или фотографии, необходимо определить смещение `SrcX`, `SrcY` от левого верхнего угла изображения и ширину с высотой `SrcWidth`, `SrcHeight`, все остальное будет беспощадно отрезано.

```
Procedure  GDIPLUS_PAINT(DC:HDC;GPImage :TGPImage);
var GPGraphics:TGPGraphics;
begin
GPGraphics:=TGPGraphics.Create(DC);
    GPGraphics.DrawImage(GPImage,0,0); //исходный рисунок
{обрезаем рисунок}
    GPGraphics.DrawImage(GPImage,300,0,350,0,
               GPImage.GetWidth -350, GPImage.GetHeight -220,UnitPixel);
GPGraphics.Free;
end;
```

В нашем примере мы в два счета расправились с летящей ракетой, оставив на второй картинке лишь изображение Сатурна (рис. 35.6).

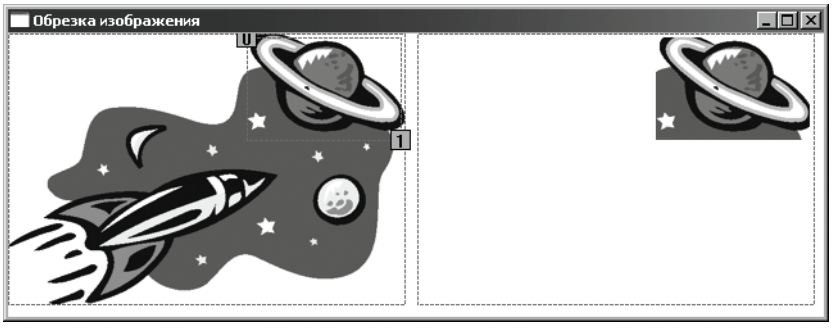


Рис. 35.6. Обрезка изображения

Дальше ирония становится неуместной. На смену безобидным методам приходят функции, с которыми шутки плохи. Следующая парочка позволяет деформировать картинку. Попробуйте за одну попытку угадать название метода, обеспечивающего эту операцию. Правильно, это все тот же `DrawImage`!

```
function DrawImage(Image: TGPImage; pDstPoints: PGPPoint;
                   Count: Integer): TStatus; overload;
function DrawImage(Image: TGPImage; pDstPoints: PGPPointF;
                   Count: Integer): TStatus; overload;
```

Результирующее изображение описывается тремя точками, определяющими три из четырех вершин параллелограмма. Отображение происходит следующим образом: левая верхняя точка исходного рисунка передается в нулевой элемент массива, правая верхняя – в первый, нижняя левая – во второй. Необходимость в определении четвертой точки отсутствует, так как она вычисляется автоматически. Но при желании вы можете ввести и ее, явным образом указав методу форму искаженного рисунка. На массив координат ссылается указатель `pDstPoints`. Заключительный параметр информирует метод о количестве точек в массиве. Порядок применения метода представлен в следующих строках кода:

```
Procedure  GDIPLUS_PAINT(DC:HDC;GPImage :TGPImage);
var GPGraphics:TGPGraphics;
    Points:Array[0..2] of TGPPointF;
    Pen:TGPPen;
begin
  GPGraphics:=TGPGraphics.Create(DC);
  GPGraphics.DrawImage(GPImage,0,0);

  Points[0]:=MakePoint(100,0.0); Points[1]:=MakePoint(300,150.0);
  Points[2]:=MakePoint(100,150.0);
{деформируем изображение}
  GPGraphics.DrawImage(GPImage,pGPPointF(@Points),3);
  GPGraphics.Free;
end;
```

Представленный на рисунке (рис. 35.7) снимок экранной формы демонстрирует возможности метода. При его помощи мы можем не только искажать форму исходного изображения (превращая прямоугольник в параллелограмм), но и изменять его пропорции и поворачивать рисунок вокруг горизонтальной и вертикальной осей. Весь секрет заключается в правильном выборе координат точек.

Можно предположить, что в основу работы метода `DrawImage()` *положена функция Windows API с неброским названием* `PlgBlt()`. *Косвенным подтверждением этой мысли является фирменный для* `PlgBlt()` *способ определения области, в которой будет выведено изображение, именно здесь предусмотрена возможность деформации картинки (см. главу 17 «Растровые операции»).*

Но и на этом создатели класса `TGPGraphics` не смогли остановиться. Не переводя дух, они разработали группу методов, объединяющих в себе практически все, о чем можно только мечтать: вывод картинки с одновременной возможностью ее обрезки и деформации, а также настройка единицы измерения и управление основными атрибутами цвета (гамма-коррекция, цветовая фильтрация и т. п.). Кроме того, методы способны обращаться к функции обратного вызова с целью более тонкой настройки процесса вывода. После недолгих (а может быть, долгих и мучительных) дебатов, скорее всего единогласным тайным голосованием анонсированную выше группу методов постановили назвать `DrawImage`! Решение всему чему только можно присваивать одно и то же имя может внести сумятицу в голову любому специалисту, но зато существенно упрощает жизнь студентов-второгодников. Согласитесь, что это очень удобно, когда абсолютно все называется одинаково. На любой заковыристый вопрос докучливого профессора можно давать один и тот же ответ...

```
function DrawImage(Image: TGPImage; const DestRect: TGPRect;
                   SrcX, SrcY, SrcWidth, SrcHeight: Integer;
                   SrcUnit: TUnit;
```

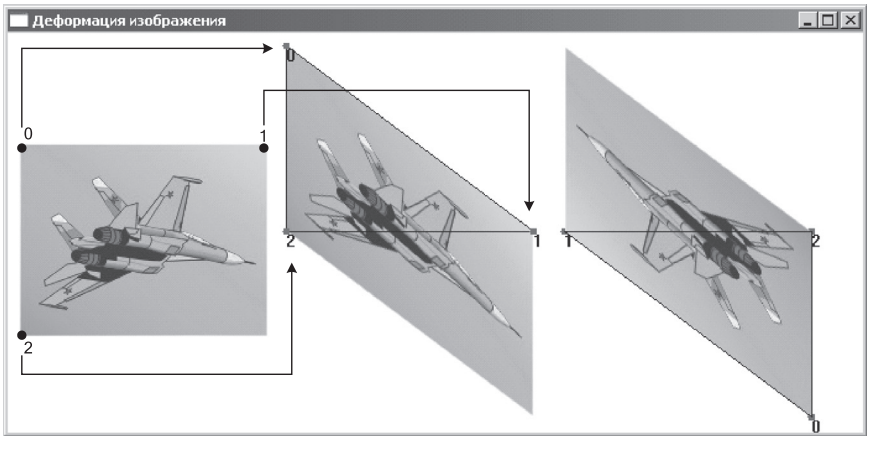


Рис. 35.7. Пример деформации и поворотов изображения

```
                    ImageAttributes: TGPImageAttributes = nil;
                    Callback: DrawImageAbort = nil;
                    CallbackData: Pointer = nil): TStatus; overload;
function DrawImage(Image: TGPImage; const DestRect: TGPRectF;
                    SrcX, SrcY, Srcwidth, SrcHeight: Single;
                    SrcUnit: TUnit;
                    ImageAttributes: TGPImageAttributes = nil;
                    Callback: DrawImageAbort = nil;
                    CallbackData: Pointer = nil): TStatus; overload;
```

Почти все параметры нам знакомы. Кроме собственно изображения `Image` требуется заполнить структуру `DestRect`, в ней определяются границы области вывода рисунка. Группа `SrcX`, `SrcY`, `SrcWidth`, `SrcHeight` несет ответственность за обрезку изображения. Аргумент `SrcUnit` определяет единицу измерения. Из новинок в первую очередь выделю `ImageAttributes`. Это экземпляр класса `TGPImageAttributes`, хранящий информацию о том, каким образом растровое либо векторное изображение должно поступать со своими цветовыми характеристиками. Например, с помощью этого параметра можно разложить картинку на каналы цветовой модели CMYK или сделать определенные цвета прозрачными. Следующий параметр хранит ссылку на необязательную функцию обратного вызова, которая обладает весьма тривиальной синтаксической конструкцией:

```
ImageAbort = function: BOOL; stdcall;
```

Впрочем, это не удивительно, так как указанная функция необязательна и используется лишь при необходимости прерывания вывода картинки. То же самое могу сказать и об указателе на дополнительные данные `CallbackData`.

Раз мы упомянули возможность декомпозиции изображения на составные части модели CMYK, то стоит подтвердить сказанное на практике.

```
Procedure GDIPLUS_PAINT(DC:HDC;GPImage :TGPImage);
var GPGraphics:TGPGraphics;
    DestRect:TGPRectF;
    IA: TGPImageAttributes;
begin
GPGraphics:=TGPGraphics.Create(DC);
  GPGraphics.DrawImage(GPImage,1,1); //исходное изображение
  IA:=TGPImageAttributes.Create;
{Канал C (голубой цвет)}
  DestRect:=MakeRect(210,50,GPImage.GetWidth*0.75,GPImage.GetHeight*0.75);
  IA.SetOutputChannel(ColorChannelFlagsC);
  GPGraphics.DrawImage(GPImage,DestRect,0,0,GPImage.GetWidth,GPImage.GetHeight,
                       UnitPixel,IA,nil,nil);
{и т. д. для оставшихся каналов:
IA.SetOutputChannel( ->
ColorChannelFlagsM, ColorChannelFlagsY и ColorChannelFlagsK)}

  IA.Free;
  GPGraphics.Free;
end;
```

Код примера несколько сокращен – нами выделен лишь канал, содержащий данные о голубой цветовой составляющей, а на рис. 35.8 (см. цветную вклейку, с. 423) представлен целостный результат – изображение Эйфелевой башни разложено на 4 канала. Для демонстрации дополнительных возможностей DrawImage() при выводе картинок для каждого из каналов мы их уменьшили.

Нам осталось рассмотреть последнюю пару методов DrawImage():

```
function DrawImage(Image: TGPImage; pDestPoints: PGPPointF; Count: Integer;
                   SrcX, SrcY, SrcWidth, SrcHeight: Single; SrcUnit: TUnit;
      ImageAttributes: TGPImageAttributes = nil; Callback: DrawImageAbort = nil;
      CallbackData: Pointer = nil): TStatus; overload;

function DrawImage(Image: TGPImage; pDestPoints: PGPPoint;
      Count, SrcX, SrcY, SrcWidth, SrcHeight: Integer; srcUnit: TUnit;
      ImageAttributes: TGPImageAttributes = nil; callback: DrawImageAbort = nil;
      CallbackData: Pointer = nil): TStatus; overload;
```

Их единственное отличие от предыдущего дуэта в том, что они позволяют деформировать результирующую картинку. С этой целью вместо аргумента DestRect введен указатель на массив точек pDestPoints, хранящий координаты деформирующих точек и их количество.

РЕЗЮМЕ

В GDI+, наряду с привычными для нас форматами изображений BMP, ICON, WMF и EMF, появились поддержка распространенных графических форматов GIF, TIFF, JPEG и PNG. Ключевым специалистом по обслуживанию всех перечисленных выше форматов считается универсальный класс TGPImage. Класс умеет загружать рисунки из файлов и сохранять их, при необходимости конвертируя в другой формат. Для вывода загруженного изображения на экран или печатающее устройство потребуются совместные усилия TGPImage и холста GDI+.

Класс TGPImage не лишен недостатков, на мой взгляд, основной из них заключается в неспособности работать с графическими объектами на языке, понятном GDI (с помощью дескрипторов). К счастью, этот пробел устранен у потомков TGPImage – специализированных классов TGPBitmap и TGPMetafile, с которыми мы познакомимся в главе 37.

36

Метаданные EXIF

Формат некоторых изображений допускает наличие метаданных. Например, современные электронные фотоаппараты присоединяют к файлу фотоснимка (как правило, кодируемого в формате JPEG) дополнительную информацию о модели камеры, фокусном расстоянии, параметрах экспозиции, использовании вспышки в момент срабатывания затвора, дате и времени снимка и многое-многое другое, зачастую даже недоступное для понимания обычного смертного.

К счастливым обладателям уникальной возможности работы с метаданными относятся не только популярные графические форматы (JPEG, TIFF и PNG), но и некоторые аудиоформаты данных (ряд звуковых файлов, закодированных по правилам импульсно-кодовой модуляции (Pulse Code Modulation, PCM)).

Формат данных EXIF (Exchangeable Image File) не столь стар. Точная дата рождения EXIF версии 1.0 мне неизвестна, могу лишь сказать, что это знаменательное событие произошло на рубеже 1995–1996 годов. Уже тогда была предусмотрена возможность хранения в одном файле данных кроме собственно изображения еще и его *уменьшенной копии* (*thumbnail*) и подраздела с текстовым описанием. Имеющиеся в нашем распоряжении заголовочные файлы GDI+ нацелены на работу с EXIF версии 2.1. Эта версия зарегистрирована Японской ассоциацией развития электронной промышленности (Japan Electronic Industry Development Association, JEIDA) в июне 1998. Затем ассоциация JEIDA в виде структурного подразделения влилась в Японскую ассоциацию электронных и информационных технологий (Japan Electronics and Information Technology Association, JEITA) и почти одновременно с этим (в 2002 году) предложила последнюю на сегодня версию EXIF 2.2.

Формат EXIF поддерживают большинство крупных производителей камер. Некоторые из них (Casio, Fuji, Konica, Minolta, Nikon, Olympus и др.) даже объединены в консорциум «EXIF Supporters Group».

Не знаю, существует ли хоть какая-нибудь классификация программистов по их психологическому портрету? Почему я об этом заговорил? Уверен, что эта тематика может стать толчком для двух-трех десятков диссертаций.

Когда я впервые проектировал приложение с применением EXIF, мне невольно пришла мысль, что над созданием структуры метаданных EXIF трудились не просто рядовые разработчики, а Программисты с большой буквы. Отбор в команду этих суперспециалистов наверняка осуществлялся только из числа людей, в принципе не искавших обходных путей и простых решений. Почему? Да потому, что только силами таких кудесников можно было разработать потрясающе профессионально запутанные операции чтения метаданных из цифровой фотографии...

Конечно же, все сказанное выше – не более чем шутка, тем более что в случае с EXIF программисты скорее всего столкнулись с неприятностями в лице аппаратных ограничений цифровых фотокамер, но, как говорится, судите сами...

Физически метаданные представляют собой некоторый перечень свойств. Для описания отдельного свойства применяется структура:

```
type TPropertyItem = record
   id       : PROPID;  // идентификатор свойства
   length   : ULONG;   // размер массива с данными
   type_    : WORD;    // тип свойства
   value    : Pointer; // указатель на массив с данными
 end;
```

Идентификатор свойства `id`, берется не с потолка, он должен соответствовать заранее зарегистрированному значению для изображений формата EXIF. Идентификаторы свойств (таковых более двух сотен) утверждаются в JEITA. В перечне идентификаторов найдется все, что вашей душе угодно, начиная с координат глобальной системы позиционирования (константа `PropertyTagGpsVer=$0000`) и модели фотокамеры (`PropertyTagEquipModel= $0110`) и заканчивая описанием использованного во время съемки массива цветных светофильтров (`PropertyTagExifCfaPattern= $A302`). По понятным причинам в книге все 200 идентификаторов перечислять не станем. Имена и значения констант имеются в модуле GDIAPI, еще больше ценной информации вы найдете в SDK и спецификации EXIF.

За подробностями о спецификациях формата EXIF стоит обратиться к интернет-сайту http://www.exif.org. Здесь находится описание последнего формата EXIF 2.2, датированное апрелем 2002 года. Это же описание вы обнаружите и в библиотеке к этой книге.

Свойства EXIF хранят разнотипные значения. Название камеры логично описывать набором символов, геометрический размер фотоснимка – в целых числах, параметры экспозиции – в вещественных. Для представления этих данных требуется различное число байт. Для того чтобы наши программы сумели корректно прочитать эти данные, им необходимо знать, что именно содержится в том или ином свойстве. Поэтому в состав структуры `TPropertyItem`

введено поле type – оно указывает, какой тип данных содержится в свойстве (табл. 36.1).

Таблица 36.1. Типы свойства (поле TPropertyItem.type)

Константа	Значение	Описание
PropertyTagTypeByte	1	Массив байт.
PropertyTagTypeASCII	2	Текст ASCII, завершающийся нулем.
PropertyTagTypeShort	3	Массив неотрицательных коротких целых (16-бит).
PropertyTagTypeLong	4	Массив неотрицательных длинных целых (32-бит).
PropertyTagTypeRational	5	Массив пар неотрицательных длинных целых (32-бит). Числитель числа хранится в первом элементе пары, знаменатель – во втором.
PropertyTagTypeUndefined	7	Неопределенный тип.
PropertyTagTypeSLONG	9	Массив длинных целых (32-бит).
PropertyTagTypeSRational	10	Массив пар длинных целых (32-бит). Целая часть числа хранится в первом элементе пары, дробная – во втором.

Как видите, создатели EXIF не боялись сложностей, чего только стоит идея описывать вещественные числа, например, не в формате стандартного типа Real, а в виде дроби, как в случае PropertyTagTypeRational и PropertyTagTypeSRational.

Дробями чудеса не заканчиваются. Обязательно отмечу, что при работе со свойствами изображения без SDK и документации спецификации не обойтись еще потому, что, на мой взгляд, проектировщики EXIF обладали незаурядным чувством юмора. Закрываю глаза и вижу группу японских специалистов, поставивших себе цель, чтобы представление данных в каждом очередном свойстве не было похоже на все предыдущие. Особых сюрпризов следует ожидать от свойств с неопределенным типом данных PropertyTagTypeUndefined. Хороший тому пример – свойство с идентификатором id=h9101 (константа PropertyTagExifCompConfig). Это не что иное, как описание цветовой схемы, используемой для представления изображения. Сколько таких моделей существует? Не более двух-трех десятков, даже если в список включить названия-синонимы и названия схем, не применяемых в индустрии цифрового фото и видео. На память приходят RGB, CMY, CMYK, HSV, HLS, HSB, CIE, YUV (и разновидности YUV, YIQ, YDbDr, YCbCr) и не забудем про самый древний стандарт XYZ. Казалось, чего проще? Если вы не хотите представлять эту информацию в формате ASCII, то отведите для хранения один байт (или даже два байта для будущих шестидесяти пяти тысяч поколений форматов) и утвердите правило: единице соответствует модель RGB, двойке – CMY и так далее. Не тут-то было! В ассоциации JEIDA не ищут простых путей! Для описания цветовой схемы отведено 4 байта. Если в байте вы обнаружили 0, то

это означает, что здесь данных нет. Единице соответствует символ «Y», 2 – символы «Cb», 3 – «Cr», 4 – «R», 5 – «G», 6 – «B». Таким образом, если мы работаем в классическом несжатом RGB, то в четверке байт окажутся следующие числа: 4, 5, 6 и 0. Я думаю, теперь читатель согласится со мной, что специалисты JEIDA не едят свой хлеб зря! Кстати, чуть не забыл – на значении 6 список заканчивается, так что кроме RGB свойство позволяет закодировать еще одну аббревиатуру – YCbCr (числа 1,2,3)... Нам остается только восхищаться специалистами, внедрившими в стандарт столь незаурядный способ кодирования!

Разобравшись с форматом отдельного свойства, перейдем к изучению способа получения метаданных.

Чтение метаданных EXIF

Поддержка формата EXIF включена в состав GDI+ в качестве опциональной части класса `TGPImage`. Класс способен читать и редактировать метаданные, а также извлекать из снимков их миниатюрные копии.

Сбор прикрепленной к файлу информации представляет собой многоходовую, практически шахматную комбинацию. Дебют посвящается выяснению количества свойств у исследуемого изображения. Непревзойденный специалист в этой области – метод:

```
function GetPropertyCount: UINT;
```

Если метаданные обнаружены (функция вернет значение больше нуля), настает черед построения списка идентификаторов свойств. Для этого предназначен метод:

```
function GetPropertyIdList(NumOfProperty: UINT; pList: pPropID): TStatus;
```

В параметр `NumOfProperty` передается число свойств, во второй – указатель на массив идентификаторов. Заметьте, что тип данных `PROPID` (он соответствует типу `Cardinal`) объявлен в модуле `ActiveX`. Распределив память для списка с идентификаторами, переходим к индивидуальной работе с отдельными свойствами.

Каждое из свойств изображения имеет различную размерность, посему нам необходима функция, предоставляющая нам такие данные:

```
function GetPropertyItemSize(PropId: PROPID): UINT;
```

Единственный параметр метода – числовой идентификатор исследуемого свойства. В ответ мы получим размер свойства в байтах. Этим значением необходимо распределить память для хранения данных о свойстве (речь о структуре `TPropertyItem`) и затем вызвать функцию:

```
function GetPropertyItem(PropId: PROPID;
                         PropSize: UINT; Buffer: PPropertyItem): TStatus;
```

Метод отыщет свойство, идентифицируемое параметром `PropId`, и передаст информацию в приемный буфер `Buffer`.

На практике порядок сбора данных EXIF выглядит следующим образом. На подготовительном этапе мы типизируем данные в соответствии с предпочтениями поля type структуры TPropertyItem (см. табл. 36.1). Например, для свойств снимка, описываемых неотрицательными короткими целыми значениями (PropertyTagTypeShort=3), будет удобно использовать массив типа TArr3. Из-за того что нам заранее не известен размер свойства, пришлось подстраховаться, зарезервировав в массиве место для 10 элементов.

```
type  TArr3 =Array[0..9] of WORD;          {PropertyTagTypeShort}
      TArr5 =Array[0..9,0..1] of Cardinal; {PropertyTagTypeRational}
      TArr10=Array[0..9,0..1] of Integer;  {PropertyTagTypeSRational}

      TCH=Array[0..3] of CHAR; //массив символов
      TBt=Array[0..3] of BYTE; //массив байт
```

Закончив с типизацией, готовим процедуру, содержащую два параметра: GPImage – ссылка на экземпляр изучаемого изображения и LISTBOX – список, в строки которого мы намерены поместить данные. В теле процедуры последовательно обращаемся к описанным ранее методам (узнаем число свойств NumOfProperty, заполняем список идентификаторов List и в цикле for .. to .. do последовательно перебираем все свойства).

```
Procedure GetImageMetaData(GPImage :TGPImage; LISTBOX :  HWnd);
var NumOfProperty,I,PropSize,X : UINT;
    List:Array of PROPID;
    pItem:pPropertyItem;
    S:String;

    pArr3:^TArr3;    pArr5:^TArr5;
    pArr10:^TArr10;  pCH:^TCH;     pBT:^TBt;
begin
    NumOfProperty:=GPImage.GetPropertyCount;
    SetLength(List,NumOfProperty);
    GPImage.GetPropertyIdList(NumOfProperty,PPropID(List));

   for i:=0 to NumOfProperty-1 do
   begin
     PropSize:=GPImage.GetPropertyItemSize(List[i]);
     GetMem(pItem,PropSize);
     GPImage.GetPropertyItem(List[i],PropSize,pItem);


     CASE  pItem^.id OF
     {0x010F} PropertyTagEquipMake: S:='Производитель: '+ StrPas(pItem^.value);
     {0x0110} PropertyTagEquipModel: S:='Модель: '+ StrPas(pItem^.value);
     {0x011A} PropertyTagXResolution: begin
                   {Type_=5,Length=8 байт}
                   pArr5:=(pItem^.value);
                   S:=Format('%s %d/%d',['Разрешение по X:
                                        ',pArr5^[0,0],pArr5^[0,1]]);
                                    end;
      //РАБОТА С ДРУГИМИ СВОЙСТВАМИ
      //…
```

```
        Else S:=Format('%X length=%d
                        type=%d',[pItem^.ID,pItem^.length,pItem^.type_]);
      END;

      SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
      FreeMem(pItem,PropSize);
    end;
end;
```

В предложенном фрагменте кода оператор `Case..of` обрабатывает далеко не все варианты идентификаторов `EXIF`, так сделано исключительно для экономии бумаги. Если хотите получить опыт работы с другими свойствами EXIF – просмотрите примеры к этой главе. Результат выполнения процедуры представлен на рис. 36.1 (см. цветную вклейку, с. 423).

Библиотека GDI+ не исключает возможности получения всех метаданных одним пакетом. В этом случае работа программиста начинается с обращения к методу:

```
function GetPropertySize(out TotalBufferSize, NumProperties : UINT): TStatus;
```

Функция содержит два выходных параметра, в которые направляются данные о необходимом размере приемного буфера в байтах `TotalBufferSize` и количестве свойств `NumProperties`. Полученные данные потребуются для инициализации очередного метода:

```
function GetAllPropertyItems(TotalBufferSize, NumProperties: UINT;
                             pAllItems: PPROPERTYITEM): TStatus;
```

Указатель на весь перечень свойств окажется в параметре `pAllItems`.

Редактирование метаданных

В основу процесса редактирования данных EXIF положено всего два действия: удаление и добавление свойства. Как всегда в таких случаях, наиболее простая операция отвечает за изъятие свойства из списка. Для этого необходимо обращение к функции:

```
function RemovePropertyItem(PropId: TPROPID): TStatus;
```

Единственное, что нам потребуется, – идентификатор ненужного свойства.

Добавление нового свойства или редактирование данных в уже существующем свойстве реализуется с помощью одного и того же метода:

```
function SetPropertyItem(const Item: TPropertyItem): TStatus;
```

Перед обращением к методу следует проинициализировать параметр `Item` так, как представлено в очередном примере.

```
Function SetTextPropertyItem(GPImage:TGPImage;ItemID:Cardinal;S:String):TStatus;
var Item:TPropertyItem;
    ValueSize:Cardinal;
begin
```

```
ValueSize:=Length(S);
If ValueSize<1 then
   begin
     Result:=InvalidParameter;
     Exit;
   end;
{Строка должна завершаться символом #0}
IF S[ValueSize]<>#0 then
begin
 S:=S+#0; INC(ValueSize);
end;

  With Item do
    begin
    id:=ItemID;
    Type_:=PropertyTagTypeASCII;
    length :=ValueSize;
    Value:=pChar(S);
    end;
Result:=GPImage.SetPropertyItem(Item);
end;
```

Наша функция позволяет добавлять свойства, содержащие текстовые данные. Для этого необходимо передать три параметра: ссылку на изображение `GPImage`, идентификатор `ItemID` и текстовые данные `S`.

Миниатюра изображения

В проводнике Windows XP (программа Explorer) предусмотрен весьма полезный сервис – чтение уменьшенных копий, находящихся в графическом файле. Благодаря этому пользователь может быстро просматривать содержание папок с рисунками. Результат достигается благодаря самоотверженным действиям метода:

```
function GetThumbnailImage(ThumbWidth, ThumbHeight: UINT;
                           Callback: GetThumbnailImageAbort = nil;
                           CallbackData: pointer = nil): TGPImage;
```

В тривиальной нотации нам достаточно указать горизонтальный `ThumbWidth` и вертикальный `ThumbHeight` размеры будущей миниатюры. Помощь функции обратного вызова (на которую ссылается параметр `Callback`), как правило, не требуется. Единственное исключение – случай, когда необходимо немедленно прекратить формирование миниатюрных копий. В этом случае с помощью функции обратного вызова:

```
ImageAbort = function: BOOL; stdcall;
```

система прерывает работу с копиями. Как видите, в простейшей нотации функция даже не содержит параметров, для передачи в нее каких-либо данных следует воспользоваться услугами указателя на область в памяти `CallbackData`.

Метод GetThumbnailImage не уменьшает исходный рисунок до заданных размеров, это было бы слишком банальное решение. В алгоритме метода предусмотрен предварительный поиск в обслуживаемом изображении фрейма с его уменьшенной копией. Такая возможность имеется в графических форматах JPEG, TIFF и PNG.

В соответствии с настоятельными пожеланиями стандарта метаданных EXIF размеры миниатюры для файла в формате JPEG должны составлять 160×120 пикселов.

Предложенный ниже упрощенный пример демонстрирует порядок создания миниатюр графических файлов, расположенных в каталоге DIR.

```
Function GetThumbnailList(DIR:String; const ThSize:byte=160):TStringList;
var GPImage, GPThumbImage : TGPImage;
    F:TSearchRec;
begin
IncludeTrailingBackslash(DIR); //путь должен завершаться символом «\»
Result:=TStringList.Create;
 If FindFirst(DIR+'*.*',faAnyFile,F)=0 Then {перебор файлов}
 Repeat
if ((F.Attr and NOT(faDirectory) and NOT(faHidden)) <> 0)
     AND (F.Size>0) Then
      begin
        GPImage:=TGPImage.Create(DIR+F.Name);
        GPThumbImage:=GPImage.GetThumbnailImage(ThSize, ThSize, nil, nil);
        Result.AddObject(F.Name,TObject(GPThumbImage));
        GPImage.Free;
      end;
 Until FindNext(F)<>0;
FindClose(f);
End;
```

В результате выполнения функции возвращается список TStringList, каждый элемент которого хранит имя файла и объект TGPImage с уменьшенной копией изображения.

Для извлечения миниатюры из списка достаточно указать ее индекс:

```
GPImage:=TGPImage(ThumbnailList.Objects[index]);
```

Все остальное – дело техники: просматриваем список и последовательно выводим миниатюры на экран.

В документации SDK утверждается, что при отсутствии в составе графического образа его уменьшенной копии метод сформирует ее самостоятельно. К сожалению, на практике метод GetThumbnailImage() *не всегда справляется со своими обязанностями и зачастую «спотыкается» на таких изображениях. В этом случае (вместо самостоятельной генерации уменьшенной копии) функция возвращает ссылку на полноразмерное изображение. В сопровождающей эту книгу библиотеке на сайте издательства все файлы в формате JPEG обладают интегрированной миниатюрой, исключение составляет файл «cat.jpg». Можете проверить поведение* GetThumbnailImage() *при обслуживании именно этого файла.*

РЕЗЮМЕ

Еще немногим более пятнадцати лет назад цифровой фотоаппарат многим был в диковинку. Только вид ценника такой «безделушки» вызывал у покупателя легкое головокружение, а внешний вид фотографий оставлял желать лучшего. Стремительное развитие цифровой техники, удешевление памяти, появление высококачественных светочувствительных матриц постепенно снизило себестоимость цифровой техники и одновременно повысило качество получаемых снимков. Программисты также не отстают от технического прогресса, в помощь владельцам цифровой фототехники японской ассоциацией JEITA был предоставлен формат метаданных EXIF, позволяющий сохранять в файле фотографии дополнительную информацию – описание снимка. Поддержка формата EXIF включена в состав GDI+ в качестве опциональной части класса `TGPImage`.

37

Особенности работы с битовыми образами и метафайлами

В подавляющем большинстве случаев для работы с изображениями в GDI+ достаточно услуг универсального класса `TGPImage`. Он способен инкапсулировать все поддерживаемые GDI+ форматы, умеет загружать и сохранять изображения, научен считывать большинство их характеристик, а в союзе с холстом `TGPGraphics` легко справится с выводом рисунков на поверхности графического устройства. Однако у указанного класса есть один весьма существенный недостаток – он прячет от нас особенности физической реализации изображения. Поищите среди методов `TGPImage` хотя бы один, возвращающий дескриптор изображения. Вы обречены на неудачу. Многие удивятся – разве это недостаток? Инкапсуляция – это один из краеугольных камней объектно-ориентированного программирования, а она как раз и предполагает сокрытие от программиста наиболее уязвимых сторон объекта. Таким образом снижается вероятность появления некорректного кода и, как следствие, повышается общая надежность программы. Разве здесь есть повод для печали? Отвечу, что да, повод имеется. Судите сами. При разработке нетривиального программного обеспечения, в основу которого положены не только библиотека VCL (или MFC для программистов Microsoft Visual Studio), но и как минимум функции API Windows, без дескрипторов никуда не деться – нравится нам это или нет, но они наша путеводная звезда. Как ни крути, GDI+ построен на фундаменте обычного GDI, посему классы модного сегодня GDI+ просто обязаны уметь делать все, что и их прародитель. Вы не согласны? Тогда попробуйте средствами `TGPImage` создать инициализированный аппаратно-зависимый битовый образ DDB или нарисовать векторный рисунок...

Парадоксально, но факт – классы GDI+, способные предоставить в распоряжение программиста интерфейс методов GDI, в цепочке наследования находятся не до, а после `TGPImage`*! Сразу видно, что стиль программирования разработчиков классов GDI+ не относится к школе Borland. В библиотеке VCL аналогов такому подходу не имеется. Здесь чем выше в иерархии класс, тем в большей степени он удален от*

фундамента системы – функций Windows API. И это вполне логично. Но в GDI+ все получилось наоборот – два наследника TGPImage *(классы* TGPBitmap *и* TGPMetafile*) позволяют программисту разговаривать с объектами на языке, понятном обычному GDI, на что их предок не способен. Не берусь судить, так ли это было задумано при проектировании GDI+. Вполне вероятно, что изначально рождение* TGPBitmap *и* TGPMetafile *вообще не планировалось, и они появились на свет после «обкатки» библиотеки по совету тестирующих программистов.*

Битовые образы, класс TGPBitmap

Класс TGPBitmap построен на основе TGPImage, его основное назначение – осуществление операций загрузки и сохранения растровых и векторных изображений с расширенной поддержкой интерфейса GDI. Цель существования класса TGPBitmap заключается в предоставлении объектно-ориентированной оболочки для работающих с растровыми образами структур и функций Windows GDI.

Вас не должно смущать название класса TGPBitmap*. Несмотря на то, что в его имени имеется слово bitmap, переводимое как «битовая карта», объект способен загружать также и векторную графику, но создавать векторный рисунок он не умеет.*

Создание экземпляра класса

В жилах TGPBitmap течет кровь предка – универсального класса для работы с изображениями. Посему не будем останавливаться на уже известных нам (унаследованных от TGPImage) конструкторах и методах и сразу перейдем к уникальным возможностям изучаемого класса.

Простейший из вновь введенных конструкторов позволяет создать неинициализированное изображение заданного размера и глубины цвета:

```
constructor Create(Width, Height: Integer;
                   Format: TPixelFormat = PixelFormat32bppARGB);
                   reintroduce; overload;
```

Прямоугольные размеры будущего рисунка определяются параметрами Width и Height, а глубина цвета – последним параметром Format (см. табл. 35.2). Позднее (по завершению работы конструктора) полученное изображение можно раскрасить на свой вкус и сохранить в поток либо в файл так, как представлено в следующих строках кода:

```
Procedure  DrawImage(DC:HDC);
var GPGraphics,GPGraphicsImage:TGPGraphics;
    GPBitmap :TGPBitmap;
    EncoderCLSID:TGUID;
    GPPen:TGPPen;
    GPBrush:TGPSolidBrush;
const fName='c:\redellipse.bmp';
begin
    GPGraphics:=TGPGraphics.Create(DC);
```

```
    GPBitmap :=TGPBitmap.Create(100,100,PixelFormat32bppRGB);
    GPGraphicsImage:=GPGraphics.FromImage(GPBitmap);

    GPPen:=TGPPen.create(aclRed,3);
    GPBrush:=TGPSolidBrush.Create(aclYellow);
    GPGraphicsImage.FillRectangle(GPBrush,0,0,100,100);
    GPGraphicsImage.DrawEllipse(GPPen,10,10,80,80);

    GetEncoder('bmp',EncoderCLSID);
    GPBitmap.Save(fName,EncoderCLSID); //сохраняем рисунок в файл

    ShellExecute(Wnd, {просмотрим созданный файл внешней программой}
             'Open',
             fName,
             PChar(''),
             PChar('c:\'),
             SW_SHOWNORMAL);

  GPPen.Free; GPBitmap.Free; GPBrush.Free;
  GPGraphics.Free;
  GPGraphicsImage.Free;
end;
```

В ходе примера мы создаем пустой битовый образ размерами 100×100 пикселов и с глубиной цвета 32 бита. Получив заказанный объект, рисуем на его поверхности; для этого мы обращаемся к методу `FromImage()` холста GDI+. Начертив красный эллипс на желтом фоне, сохраняем шедевр в файл.

Новый пустой битовый образ может быть построен на основе характеристик холста GDI+, описываемого параметром `Target`. Для этих целей GDI+ рекомендует использовать конструктор:

```
constructor Create(Width, Height: Integer;
                   Target: TGPGraphics); reintroduce; overload;
```

В первую очередь данный метод пригодится в ситуации, когда необходимо повторить такие важные характеристики битового образа, как глубина цветов и разрешение.

Очередной конструктор позволит создать инициализированный битовый образ:

```
constructor Create(Width, Height, Stride: Integer;
                   Format: TPixelFormat;
                   scan0: pBYTE); reintroduce; overload;
```

Здесь параметр `Stride` назначает размер скан-линии будущего растра. Обычно это значение соответствует числу байт, необходимых для описания одного пиксела (с требуемой глубиной цвета) и умноженных на ширину образа. При этом не забывайте, что скан-линия выравнивается по границе двойного слова, т. е. значение должно быть кратным 4. Последний параметр конструктора представляет собой указатель на массив байт, описывающих растр.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
```

```
    GPBitmap :TGPBitmap;
const scan0 : Array[0..7] of DWORD =($E7,$E7,$E7,$0,$0,$E7,$E7,$E7);
begin
    GPGraphics:=TGPGraphics.Create(DC);
    GPBitmap :=TGPBitmap.Create(8,8,4,PixelFormat1bppIndexed,@scan0);
    GPGraphics.DrawImage(GPBitmap,0,0);
    GPGraphics.Free;
    GPBitmap.Free;
end;
```

Основой будущего растрового рисунка способна стать поверхность DirectDraw:

```
constructor Create(Surface: IDirectDrawSurface7); reintroduce; overload;
```

Для реализации столь грандиозных планов потребуется не только интерфейс поверхности IDirectDrawSurface7, но и подключение к проекту библиотеки DirectDraw, что несколько утяжелит результирующий код.

А теперь настал черед группы конструкторов, позволяющих формировать объект TGPBitmap на основе заготовок GDI. В первую очередь познакомлю читателя с функцией, собирающей битовый образ из двух ингредиентов: классической структуры TBitmapInfo и данных рисунка, на который ссылается указатель pBitmapData:

```
constructor Create(var BitmapInfo: TBitmapInfo;
                   pBitmapData: Pointer); reintroduce; overload;
```

Напомню (см. главу 16 «Аппаратно-независимые растры»), что запись TBitmapInfo объединяет заголовок и цветовую таблицу растра.

Еще три конструктора позволяют нам «схватить за хвост» изображение, уже находящееся в памяти компьютера:

```
constructor Create(hBMP: HBITMAP; hPal: HPALETTE); reintroduce; overload;
constructor Create(hIcon: HICON); reintroduce; overload;
constructor Create(hInstance: HMODULE;
                   BitmapName: WideString); reintroduce; overload;
```

Для читателя, внимательно ознакомившегося с главами 15 и 16 этой книги, не составит особого труда идентифицировать параметры методов. Первый конструктор для построения объекта требует дескрипторы битового образа и палитры. Второй конструктор намерен создать экземпляр класса TGPBitmap на основе иконки, идентифицируемой дескриптором hIcon. Третий конструктор позволит достать растровый образ с именем BitmapName из внешнего ресурса hInstance (динамической библиотеки или исполняемого файла).

В качестве примечания упомянем о том, что совместно с конструкторами сосуществует группа методов, позволяющих создавать экземпляр объекта TGPBitmap. *Это* FromDirectDrawSurface7(), FromBITMAPINFO(), FromHBITMAP(), FromHICON(), FromResource().

Получение дескриптора образа

Технология GDI+ допускает не только создание экземпляра класса из растрового изображения, представленного в формате структур обычного GDI. Вполне реально решить обратную задачу – получить совместимые с функциями GDI дескрипторы образа, инкапсулированного в «броню» TGPBitmap:

```
function GetHBITMAP(ColorBackground: TGPColor; out hBmReturn: HBITMAP): TStatus;
function GetHICON(out hIcon: HICON): TStatus;
```

Как следует из названий, первая функция специализируется на битовых образах, а вторая – на иконках Windows.

Клонирование образа

У класса TGPBitmap имеется четыре метода клонирования, отличающихся друг от друга лишь способом определения копируемого участка:

```
function Clone(Rect: TGPRect; Format: TPixelFormat): TGPBitmap; overload;
function Clone(Rect: TGPRectF; Format: TPixelFormat): TGPBitmap; overload;
function Clone(X, Y, Width, Height: Integer;
               Format: TPixelFormat): TGPBitmap; overload;
function Clone(X, Y, Width, Height: Single;
               Format: TPixelFormat): TGPBitmap; overload;
```

Последний параметр функции определяет глубину цвета, назначаемую результирующему битовому образу. Уникальность методов заключается в возможности указания координат области, подлежащей клонированию.

Услуга выборочного копирования заданной части изображения окажется весьма кстати при работе со «склеенными» рисунками. Совсем недавно мы работали с компонентом TImageList, это контейнер из библиотеки VCL, предназначенный для хранения коллекции растровых рисунков. Единственным условием того, чтобы битовый образ смог попасть в коллекцию, было равенство геометрических размеров всех картинок. Внутри контейнера образы хранятся не по отдельности, а в виде единого блока (см. рис. 25.2). Когда возникает необходимость в обращении к конкретному рисунку, он извлекается из соответствующей части объединенного изображения.

Ниже предложена процедура, «разрезающая» на составные части коллекцию рисунков, экспортированную в файл из компонента TImageList.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPBitmap,CloneBitmap :TGPBitmap;
    X,Y:Integer;
    ColsCount,RowsCount:word;
    PixelFormat: TPixelFormat;
const W=16; H=16; //размеры 1 пиктограммы
begin
    GPGraphics:=TGPGraphics.Create(DC);

    GPBitmap :=TGPBitmap.Create('..\Graphics\group.bmp');
    GPGraphics.DrawImage(GPBitmap,0,0);
```

```
    ColsCount:=GPBitmap.GetWidth DIV W;    //количество колонок
    RowsCount:=GPBitmap.GetHeight DIV H;   //количество рядов
    PixelFormat:=GPBitmap.GetPixelFormat; //узнаем глубину цвета

    For Y:=0 to RowsCount do {перебираем «склеенные» рисунки}
       For X:=0 to ColsCount do
       begin
         CloneBitmap:=GPBitmap.Clone(X*W,Y*H,W,H, PixelFormat);
         GPGraphics.DrawImage(CloneBitmap,200+X*(W+2),Y*(H+2));
         CloneBitmap.Free;
       end;

    GPGraphics.Free;
    GPBitmap.Free;
end;
```

Изначально предполагается, что вертикальный и горизонтальный размеры атомарных рисунков равны 16 пикселам. Поэтому эта величина упакована в константы W и H. Перед «резкой» изображения мы выясняем число колонок и рядов в исходном образе и уточняем его глубину цвета. Все остальное – дело техники: во вложенном цикле последовательно перебираем все рисунки и выводим их на холст (рис. 37.1).

Рис. 37.1. «Разрезка» составного рисунка

Раскраска отдельных пикселов образа

В самом начале главы мы уже говорили о том, что одной из положительных сторон экземпляра класса TGPBitmap считается возможность рисования на поверхности битового образа. В простейших случаях нам могут пригодиться и методы, способные исследовать и закрашивать отдельные пикселы:

```
function GetPixel(X, Y: Integer; out Color: TGPColor): TStatus;
function SetPixel(X, Y: Integer; Color: TGPColor): TStatus;
```

На этот раз создатели класса не стали мудрить и присвоили методам имена, соответствующие названиям хорошо знакомых нам функций Windows API. Первый метод позволяет программисту узнать цвет пиксела с координатами

(X, Y), результат исследований окажется в выходном параметре Color. Второй метод изменит цвет указанного пиксела.

Для изменения разрешения битового образа достаточно воспользоваться методом:

```
function SetResolution(xDPI, yDPI: Single): TStatus;
```

Здесь параметры xDPI и yDPI определяют число пикселов в одном дюйме по ширине и высоте.

Блокировка и редактирование части образа

Любителям экстремальных способов редактирования битовых образов рекомендую познакомиться с парой методов LockBits() и UnlockBits(). Благодаря ним возможно организовать весьма эксцентричный способ воздействия на содержание рисунка. Первый из перечисленных методов позволяет направить весь битовый образ или его часть во внешний массив пикселов. Таким образом, мы приобретаем право читать или редактировать рисунок напрямую через ячейки массива. Завершив работу, обращаемся ко второму методу, который разблокирует изображение.

Синтаксическая конструкция метода, блокирующего образ, выглядит следующим образом:

```
function LockBits(Rect: TGPRect; Flags: UINT; Format: TPixelFormat;
                  out LockedBitmapData: TBitmapData): TStatus;
```

Функция захватывает участок изображения, координаты и размеры которого назначаются параметром Rect. Данные о пикселах, входящих в состав затронутой нашим вниманием области, копируются во временный буфер, содержимое которого можно получить, обратившись к выходному параметру LockedBitmapData. Данные представляются в виде структуры:

```
type TBitmapData = packed record
    Width       : UINT;        //ширина
    Height      : UINT;        //высота
    Stride      : Integer;     //расстояние между скан-линиями в байтах
    PixelFormat : PixelFormat; //глубина цвета
    Scan0       : Pointer;     //указатель на нулевую скан-линию
    Reserved    : UINT;        //не используется
end;
```

Если поле Stride *принимает положительное значение, то это говорит о том, что скан-линии образа следуют сверху вниз. Отрицательное значение соответствует обратному порядку скан-линий.*

Одной из полезных черт метода является то, что глубина цвета временного буфера может отличаться от глубины цвета обслуживаемого образа. Для этого предназначен параметр Format.

Степень монополизации участка зависит от набора применяемых флагов Flags:

```
ImageLockModeRead         = $0001; //доступ для чтения
ImageLockModeWrite        = $0002; //доступ для записи
ImageLockModeUserInputBuf = $0004; //определяет порядок блокировки буфера
```

Особых пояснений требует флаг ImageLockModeUserInputBuf. Он работает совместно с флагами чтения и записи. Если он установлен, то функция потребует, чтобы даже передаваемая в параметр LockedBitmapData структура TBitmapData не разделялась с другими процессами.

Завершив работу с пикселами части рисунка, необходимо разблокировать образ. Для этого предназначена функция:

```
function UnlockBits(var LockedBitmapData: TBitmapData): TStatus;
```

В единственный параметр метода передается отредактированное (если в этом есть необходимость) содержимое структуры TBitmapData.

Наш пример демонстрирует как порядок чтения данных, так и порядок записи новых значений в заблокированный участок битового образа. В качестве исходного рисунка воспользуемся абсолютно черным образом с 24-битной глубиной цвета.

```
Procedure GDIPLUS_PAINT(DC:HDC);
type TPixels=Array of DWORD;
var GPGraphics:TGPGraphics;
    GPBitmap :TGPBitmap;
    RCT:TGPRect;
    Value:DWORD;
    LockedBitmapData: TBitmapData ;
    Row,Col,X,Y,index:word;
    s:string;
    EncoderCLSID:TGUID;
const W=8; H=8;//габариты блокируемого прямоугольника
begin
GPGraphics:=TGPGraphics.Create(DC);

 GPBitmap :=TGPBitmap.Create('..\Graphics\black24bit.bmp');
 RCT:=MakeRect(10,10,W,H);
 GPBitmap.LockBits(Rct, ImageLockModeRead OR ImageLockModeWrite,
                     PixelFormat32bppARGB,lockedBitmapData);

 Y:=0;
 for row := 0 to H-1 do
  begin
  X:=150;
   for col:= 0 to W-1 do
   begin
    index:=row * lockedBitmapData.Stride div 4+col;//индекс
    Value:= TPixels(LockedBitmapData.Scan0)[index];
    s:=Format('$%.8x',[Value]);          //содержимое пиксела
    TextOut(DC,X,Y,PChar(s),Length(s)); //вывод пояснительного текста
    TPixels(LockedBitmapData.Scan0)[index]:=$FF0000FF;//синий цвет
    INC(X,80);
   end;
```

```
  INC(Y,20);
  end;

   GPBitmap.UnlockBits(lockedBitmapData);//разблокируем образ
   GPGraphics.DrawImage(GPBitmap,0,0);

   GetEncoder('bmp',EncoderCLSID);
   GPBitmap.Save('new.bmp',EncoderCLSID);//сохраняем новый образ

   GPGraphics.Free;
   GPBitmap.Free;
end;
```

В момент блокировки прямоугольной области указываем методу `LockBits()`, что мы намерены работать с 32-битным цветом. Для этого в функцию передается константа `PixelFormat32bppARGB`. Получив указатель на буфер с пикселами, сначала выводим их значения в шестнадцатеричном представлении на экран, а затем меняем их цвет на синий. В конце работы разблокируем образ (передав в него отредактированные данные) и сохраним результат в новый файл.

Метафайл, класс TGPMetafile

Точность представления изображений, великолепная масштабируемость, малый размер файлов, сравнительная простота редактирования как рисунка в целом, так и его части сделали векторную графику неотъемлемой составной частью огромного числа современных программных продуктов. Это и такие признанные фавориты, как графический редактор CorelDRAW корпорации Corel, программа промышленного проектирования AutoCAD компании Autodesk и Microsoft Visio корпорации Microsoft, так и сотни простейших графических редакторов, поддерживающих работу с метафайлами.

Вместе с GDI+ на свет появилась очередная версия метафайла, называемая EMF+. Основное отличие EMF+ от своего предшественника заключается в поддержке записей в формате GDI+. Различают две вариации EMF+. Первая версия может быть воспроизведена только средствами библиотеки GDI+, вторая версия ультрасовременного метафайла может быть просмотрена и средствами обычного GDI. Для идентификации версии метафайла в состав GDI+ введено перечисление:

```
type TMetafileType = (
 MetafileTypeInvalid,      //0- поврежденный метафайл
 MetafileTypeWmf,          //1- стандартный WMF
 MetafileTypeWmfPlaceable,//2- WMF с улучшенным заголовком
 MetafileTypeEmf,          //3- стандартный EMF
 MetafileTypeEmfPlusOnly, //4- EMF+ воспроизводимый только GDI+
 MetafileTypeEmfPlusDual  //5- EMF+ воспроизводится в GDI и в GDI+
 );
```

В GDI+ наряду с метафайлами классических форматов WMF и EMF поддерживается новый формат EMF+. Это дальнейшее развитие векторного рисунка, опирающееся на функции современной библиотеки GDI+.

Рассмотренный нами в прошлой главе класс TGPImage с легкостью справляется с воспроизведением метафайлов Windows, но не проявляет энтузиазма при решении вопросов, связанных с созданием или редактированием объектов векторной графики. Поэтому в состав библиотеки GDI+ был введен узкий специалист по метафайлам Windows – класс TGPMetafile. Указанный класс построен на фундаменте универсального TGPImage, а значительная часть его новых методов нацелена на организацию взаимодействия экземпляра класса со структурами метафайла, создаваемыми функциями обычного Windows GDI.

Основная нить, связывающая экземпляр класса TGPMetafile с «бренным» GDI, поддерживается за счет метода:

```
function GetHENHMETAFILE: HENHMETAFILE;
```

Полученный за счет метода дескриптор соответствует формату метафайла расширенной версии (EMF) и может применяться совместно с функциями, рассмотренными в главе 18 этой книги.

Свойства метафайла, заголовок TMetafileHeader

В любом случае работа с метафайлом должна начинаться со знакомства с его визитной карточкой – заголовком. В нем содержится основная описательная информация, изучив которую, программа получает ответы на вопросы о версии изображения, его размерах, разрешении и ряде других характеристик. Отмечу, что мы уже крупные специалисты по расширенному метафайлу EMF: в главе 18 этой книги нами подробно рассмотрены особенности внутренней организации метафайла EMF. Теперь настал черед обсудить тонкости работы с заголовком метафайла в GDI+.

Для упрощения взаимодействия с заголовком метафайла в GDI+ построен специальный класс TMetafileHeader, обладающий набором методов, предназначенных для информирования программиста об основных свойствах векторного рисунка. Класс универсален и способен работать как с современными метафайлами EMF и EMF+, так и с утратившим актуальность форматом WMF.

Считать экземпляр класса TMetafileHeader полноценным объектом было бы большой натяжкой. Я бы назвал его структурой с проблесками интеллекта. Умственные способности класса сводятся к тому, что он пытается запретить (правда, безуспешно) кому бы то ни было напрямую обращаться к его полям. Ниже представлено сокращенное описание класса, снабженное комментариями полей.

```
type TMetafileHeader = packed class;
public
 Type_        : TMetafileType; //тип метафайла
 Size         : UINT;          //размер метафайла в байтах
 Version      : UINT;   //версия: EMF+, EMF или WMF
 EmfPlusFlags : UINT;   //флаги для метафайла EMF+
 DpiX         : Single; //точек в дюйме по горизонтали
 DpiY         : Single; //точек в дюйме по вертикали
```

```
    X            : Integer;{координата X ограничивающего прямоугольника
                           в единицах измерения устройства}
    Y            : Integer;//координата Y ограничивающего прямоугольника
    Width        : Integer;//ширина ограничивающего прямоугольника
    Height       : Integer;//высота ограничивающего прямоугольника
    Header       : record  //вариант заголовка
       case integer of
         0: (WmfHeader: TMETAHEADER);   //заголовок WMF
         1: (EmfHeader: TENHMETAHEADER3);//заголовок EMF 3-й версии
       end;
    EmfPlusHeaderSize : Integer; // размер заголовка EMF+ в файле
    LogicalDpiX       : Integer; {разрешение по оси X в логических единицах}
    LogicalDpiY       : Integer; {разрешение по оси Y в логических единицах}
  Public
  property GetType: TMetafileType read Type;
  //…
  //другие методы доступа к полям класса
  End;
```

В описание не попали методы класса, но они этого и не заслуживают. Почему? С точки зрения «семейственности» класс `TMetafileHeader` является прямым потомком `TObject`, посему существенного наследства от небогатого предка он не получил. Все методы заголовка метафайла предназначены лишь для чтения данных, но так как в описании класса напрочь отсутствуют защищенные секции (`private` и `protected`), то ко всем полям объекта и так, без посредничества каких-либо методов, обеспечен прямой доступ.

Обратите внимание на то, что при работе с метафайлом EMF класс `TMetafileHeader` *возвращает заголовок третьей версии* `TENHMETAHEADER3`. *Это немного сокращенный вариант более современных заголовков четвертой и пятой версий* `TEnhMetaHeader`, *с которым мы познакомились в главе 18 «Расширенный метафайл».*

Доступ к заголовку метафайла

В рамках класса `TGPMetafile` предусмотрено пять версий перегружаемого метода, специализирующегося на доступе к заголовку метафайла. Простейший из них работает с заголовком метафайла, загруженного (вернее сказать, инкапсулированного) в экземпляр класса `TGPMetafile`:

```
function GetMetafileHeader(Header: TMetafileHeader): TStatus; overload;
```

Единственный параметр метода возвратит нам заголовок через выходной параметр `TMetafileHeader`.

Если метафайл находится во внешнем ресурсе (файл `Filename`, поток `Stream`) или в нашем распоряжении имеется дескриптор метафайла `hEmf`, полученный благодаря помощи функций Windows API (например, `GetEnhMetaFile` или `CopyEnhMetaFile`), то для считывания заголовка выбираем один из представленных ниже методов:

```
function GetMetafileHeader(Filename: WideString;
                           Header: TMetafileHeader): TStatus; overload;
```

```
function GetMetafileHeader(Stream: IStream;
                           Header: TMetafileHeader): TStatus; overload;
function GetMetafileHeader(hEmf: HENHMETAFILE;
                           Header: TMetafileHeader): TStatus; overload;
```

Дескриптор метафайла EMF может быть получен с помощью метода GetHENHMETAFILE() *или функции* GetEnhMetaFile() *из состава Windows API.*

Порядок работы с заголовком продемонстрирован в очередном примере. В задачи процедуры GetMetafileInfo **входит исследование векторного рисунка, хранящегося в файле** FileName**, результаты передаются в список с дескриптором** ListBox**.**

```
Procedure GetMetafileInfo(FileName:String;ListBox:HWND);
var MH:TMetafileHeader;
    s:string;
begin
SendMessage(LISTBOX,LB_ADDSTRING,0,
            INTEGER(pChar('Файл: '+ExtractFileName(FileName)+#0)));

MH:=TMetafileHeader.Create;
with TGPMetafile.Create(FileName) do
begin
GetMetafileHeader(MH);
free;
end;

S:=Format('Размер: %u байт',[MH.Size]);
SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(S+#0)));

{тип метафайла}
case MH.Type_ of
 MetafileTypeInvalid:      s:='поврежденный метафайл';
 MetafileTypeWmf:          s:='стандартный WMF';
            MetafileTypeWmfPlaceable: s:='WMF с улучшенным заголовком';
            MetafileTypeEmf:          s:='стандартный EMF';
            MetafileTypeEmfPlusOnly:  s:='EMF+ для GDI+';
            MetafileTypeEmfPlusDual:  s:='EMF+ для GDI+ и GDI';
           end;
SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar('Тип: '+s+#0)));
if MH.Type_=MetafileTypeInvalid then exit;

S:='$'+IntToHex(MH.Version,6);
SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar('Версия: '+s+#0)));
S:=Format('Разрешение по горизонтали: %f точек/дюйм',[MH.DpiX]);
SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
S:=Format('Разрешение по вертикали: %f точек/дюйм',[MH.DpiY]);
SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
S:=Format('Область вывода: (%d,%d,%d,%d) в единицах устройства',
[MH.X,MH.Y,MH.Width,MH.Height]);
SendMessage(LISTBOX,LB_ADDSTRING,0,INTEGER(pChar(s+#0)));
MH.Free;
end;
```

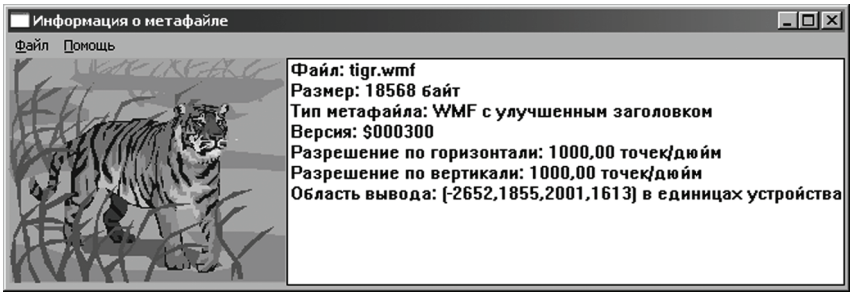


Рис. 37.2. Использование заголовка TMetafileHeader для изучения метафайла

На рис. 37.2 представлены результаты работы процедуры. Мы исследовали файл WMF с улучшенным заголовком из коллекции картинок программного пакета Microsoft Office. Заметьте, что область вывода содержит отрицательную координату. Скажу сразу, что такая ситуация вполне вероятна, если в составе записей метафайла имеется обращение к этой точке – например, вызов метода `DrawLine()` с отрицательной начальной координатой. Но в случае с форматом WMF все может быть гораздо печальнее, дело в том, что устаревший метафайл Windows не располагает информацией о размере изображения, посему данные об области вывода у старой версии WMF могут оказаться некорректными.

Начальная точка ограничивающего прямоугольника не оказывает влияния на положение рисунка при его выводе на поверхности контекста. Верхний угол метафайла будет отображен в координате, определяемой методом `DrawImage()`.

Идея подмены заголовка у устаревшего метафайла Windows появилась не на пустом месте. Хотя Microsoft настоятельно рекомендует вместо WMF применять обновленные форматы EMF и EMF+, но, несколько противореча сам себе, до сих пор включает коллекции файлов WMF в свои новейшие продукты, например, в библиотеку мультимедиа такого популярного пакета программ, как Microsoft Office. Проверьте сами, выбрав пункт меню «Вставка – Рисунок – Картинки» в текстовом процессоре Word. В каталоге обнаружатся «залежи» устаревших векторных рисунков. Почему разработчики офисных программ сломя голову не бросились конвертировать рисунки в формат EMF? Ответ прост – у всех этих файлов несколько усовершенствован заголовок. Здесь вместо явно неудачной структуры `TMetaHeader` применяется структура `TWmfPlaceableFileHeader`, в которой нашлось место для данных о графическом разрешении метафайла, благодаря чему устранился серьезный недостаток WMF – проблема с корректным масштабированием.

Класс `TGPMetafile`, учитывая факт существования WMF с улучшенным заголовком, выдал в наше распоряжение ряд методов с параметром, типизированным записью `TWmfPlaceableFileHeader` размерностью 22 байта. Взгляните на поля этой структуры:

```
type TWmfPlaceableFileHeader = packed record
```

```
  Key         : UINT32;      // всегда GDIP_WMF_PLACEABLEKEY ($9AC6CDD7)
  Hmf         : INT16;       // дескриптор (всегда 0)
  BoundingBox : PWMFRect16;  // ограничивающий прямоугольник
  Inch        : INT16;       // разрешение в пунктах
  Reserved    : UINT32;      // зарезервировано (всегда 0)
  Checksum    : INT16;       // контрольная сумма первых 10 слов заголовка
end;
```

К сожалению, исправить цветопередачу заголовок не в силах, но что касается масштабирования, то дела обстоят не так плохо. Нам позволено влиять на разрешение (поле `inch`) и размеры границ рисунка (поле `BoundingBox`). Разрешение настраивается в пунктах (TWIPS) на дюйм, обычно это число составляет 1440. Уменьшая или увеличивая указанное значение, мы сможем масштабировать изображение.

Основное отличие обычного метафайла Windows WMF от метафайла с улучшенным заголовком в том, что заголовок последнего содержит данные о разрешении и координаты ограничивающего прямоугольника. К сожалению, в распоряжении первой версии WMF (основанной на заголовке `TMetaHeader`*) такой информации не имеется.*

Хотя механизм GDI+ и способен общаться с файлами WMF, но по понятным причинам он предпочитает иметь дело с более продвинутым форматом EMF. Посему, загружая метафайл Windows, `TGPMetafile` автоматически приводит его к расширенному формату. Для преобразования заголовка к принятой у EMF форме используется метод:

```
function GetMetafileHeader(hWmf: HMETAFILE;
                           var wmfPlaceableFileHeader: TWmfPlaceableFileHeader;
                           Header: TMetafileHeader): TStatus; overload;
```

Указанный метод позволяет передавать в заголовок `Header` данные из улучшенного заголовка WMF, описываемого структурой `TWmfPlaceableFileHeader`. К улучшенному заголовку позднее мы вернемся еще раз, а пока перейдем к вопросу создания экземпляра метафайла средствами класса `TGPMetafile`.

В Windows GDI отсутствует поддержка метафайла WMF с улучшенным заголовком. Поэтому для воспроизведения таких файлов средствами GDI приходится «выкусывать» заголовок, копировать рисунок во временный файл (начиная с 22-го байта) и замещать новую версию заголовка устаревшей. Именно поэтому в англоязычной литературе улучшенный заголовок WMF называют замещаемым (англ. placeable).

Создание расширенного метафайла

Среди всех объектов GDI+ по количеству вариаций конструкторов метафайл `TGPMetafile` – безусловный лидер. Здесь можно найти все, что душе угодно, от загрузки метафайла из потока или файла до получения векторного рисунка по дескриптору объекта-метафайла Windows GDI. Чтобы не потеряться в дебрях одноименных методов, попробуем их систематизировать, для этой цели разделим имеющиеся в нашем распоряжении конструкторы на три группы:

1. Конструкторы, загружающие метафайл EMF из внешнего ресурса для его последующего воспроизведения.
2. Конструкторы, загружающие метафайл устаревшего формата WMF.
3. Конструкторы, создающие новый метафайл с целью записи.

Конструкторы, загружающие метафайл EMF

Первая группа конструкторов невелика, ее возглавляют простейшие методы, извлекающие векторный рисунок из файла и из потока.

```
constructor Create(FileName: WideString); overload;
constructor Create(Stream: IStream); overload;
```

При работе с перечисленными выше конструкторами сталкиваться с какими-то проблемами не придется. Подчеркну лишь то, что при работе с потоком речь идет об интерфейсе потока `IStream` (он объявлен в модуле `ActiveX`), а не о потомке класса `TStream` из модуля `Classes`.

Если подлежащий воспроизведению метафайл формировался средствами GDI и мы обладаем его дескриптором, то для создания экземпляра класса `TGPMetafile` обращаемся к конструктору:

```
constructor Create(hEmf: HENHMETAFILE; DeleteEmf: BOOL = FALSE); overload;
```

Помимо дескриптора `hEmf` функция снабжена опциональным параметром `DeleteEmf`. По умолчанию он установлен в `false`. Это означает, что созданный нами объект предназначен только для инкапсуляции метафайла Windows и разрушение нашего объекта никоим образом не отразится на самом изображении. При установке параметра в `true` экземпляр класса становится владельцем векторного рисунка, в этом случае вызов деструктора объекта `TGPMetafile` автоматически приводит к уничтожению инкапсулированного в объект метафайла.

Конструкторы, совместимые с форматом WMF

По большому счету все конструкторы `TGPMetafile` способны работать с метафайлом WMF первой версии. Вместе с тем разработчики GDI+ особо акцентировали внимание на метафайлах Windows с заголовком `TMetaHeader`. Любителям «старины глубокой» рекомендуется обращаться к такому метафайлу с помощью конструктора:

```
constructor Create(hWmf: HMETAFILE;
                   var WMFPlaceableFileHeader: TWmfPlaceableFileHeader;
                   DeleteEmf: BOOL = FALSE); overload;
```

Здесь `hWmf` – дескриптор метафайла Windows; `WMFPlaceableFileHeader` – улучшенный заголовок WMF; опциональный параметр `DeleteEmf` определяет дальнейшую судьбу дескриптора метафайла после уничтожения объекта `TGPMetafile` (по умолчанию дескриптор остается целым и невредимым, а в состоянии `true` освобождается вместе с метафайлом Windows в момент уничтожения экземпляра класса `TGPMetafile`).

Рассматриваемый выше метод специализируется на самом первом формате метафайла (не обладающем улучшенным заголовком). Он силовым приемом замещает штатный неудачный заголовок файла WMF (структура `TMetaHeader`) на структуру `TWmfPlaceableFileHeader` (размерностью 22 байта). Напомню, что необходимость подмены объясняется тем, что метафайл WMF с заголовком `TMetaHeader` не содержит никакой информации о размере и исходном разрешении изображения. Поэтому при воспроизведении устаревшего метафайла Windows на альтернативном устройстве изображение не сможет быть правильно отмасштабировано.

Для получения дескриптора метафайла WMF, хранящегося в файле, потребуются услуги функции Windows API `GetMetaFile()`.

Если совсем не хочется возиться с дескриптором `HMETAFILE`, то стоит воспользоваться упрощенным аналогом предыдущего конструктора:

```
constructor Create(Filename: WideString;
                   var wmfPlaceableFileHeader: TWmfPlaceableFileHeader); overload;
```

Указанному конструктору достаточно узнать имя обслуживаемого метафайла Windows.

В качестве примечания отмечу наличие варианта конвертировать записи метафайла с расширенными возможностями в записи формата метафайла WMF. Этот метод называется `EmfToWmfBits()`. *Преобразованные данные сохраняются в специальном буфере, из которого они позднее могут быть перенесены в отдельный файл. Нелишним станет напоминание, что указанный метод не сможет корректно обслуживать некоторые записи в формате EMF+, так как в WMF им просто нет аналогов.*

Конструкторы, создающие метафайл для записи

Все остальные конструкторы метафайла GDI+ предназначены для создания нового метафайла с целью организации записи векторного рисунка в память компьютера. Эти конструкторы можно разделить на три направления:

1. Запись метафайла в поток.
2. Запись метафайла в файл.
3. Запись метафайла в память.

Отличительной чертой конструкторов первого направления является наличие параметра для работы с потоком `Stream`. Конструкторы, нацеленные на запись данных в файл, вместо интерфейса потока снабжены параметром `FileName`. Конструкторы третьего направления не обладают перечисленными аргументами. В них нет необходимости, так как создаваемый метафайл не подлежит сохранению и предназначен для воспроизведения в рамках того же процесса, в котором он и появился на свет.

Простейший из имеющих дело с потоком конструктор метафайла выглядит следующим образом:

```
constructor Create(Stream: IStream; ReferenceHdc: HDC;
                   Type_: TEmfType = EmfTypeEmfPlusDual;
                   Description: pWCHAR = nil); overload;
```

Помимо интерфейса потока Stream в конструкторе имеется еще один обязательный параметр – ReferenceHdc. Это дескриптор контекста графического устройства, атрибуты которого используются при записи векторного изображения. Кроме того, у конструктора предусмотрена пара опциональных параметров: Type и Description. Первый из них уточняет тип создаваемого метафайла. Так, константа EmfTypeEmfOnly указывает на то, что записываемый векторный рисунок должен сохранить совместимость с метафайлом GDI. Константа EmfTypeEmfPlusOnly заставит сформировать метафайл, способный воспроизводиться только в GDI+. Наконец, наиболее универсальный тип рисунка (доступный всем современным версиям Windows) получится, если мы оставим значение по умолчанию – EmfTypeEmfPlusDual. Заключительный параметр предназначен для добавления к метафайлу текстовой строки с комментариями.

Более продвинутая версия конструктора метафайла, работающего с потоком, дополнена еще двумя аргументами: FrameRect и FrameUnit.

```
constructor Create(Stream: IStream; ReferenceHdc: HDC; FrameRect: TGPRectF;
                   FrameUnit: TMetafileFrameUnit = MetafileFrameUnitGdi;
                   Type_: TEmfType = EmfTypeEmfPlusDual;
                   Description: pWCHAR = nil); overload;
```

Параметр FrameRect явным образом определяет координаты обрамляющего прямоугольника (предусмотрены две версии конструктора, работающие со структурой TGPRectF или TGPRect).

Параметр FrameUnit определяет логическую единицу измерения, применяемую в векторном рисунке (табл. 37.1). По умолчанию это одна сотая миллиметра (константа MetafileFrameUnitGdi).

Таблица 37.1. Константы TMetafileFrameUnit

Константа	Значение	Единица измерения
MetafileFrameUnitPixel	2	Пиксел.
MetafileFrameUnitPoint	3	Логическая точка.
MetafileFrameUnitInch	4	Дюйм.
MetafileFrameUnitDocument	5	1/300 дюйма.
MetafileFrameUnitMillimeter	6	Миллиметр.
MetafileFrameUnitGdi	7	0,01 миллиметра.

Перечень и назначение параметров конструктора, отвечающего за создание векторного рисунка с его последующей записью в файл, абсолютно аналогичны параметрам конструктора, работающего с потоком:

```
constructor Create(FileName: WideString; ReferenceHdc: HDC;
                   FrameRect: TGPRectF;
```

```
                    FrameUnit: TMetafileFrameUnit = MetafileFrameUnitGdi;
                    type_: TEmfType = EmfTypeEmfPlusDual;
                    Description: pWCHAR = nil); overload;
```

Единственное исключение в том, что здесь конструктору вместо интерфейса потока требуется передать имя файла `FileName`.

Для создания метафайла с возможностью последующей записи и воспроизведения на поверхности графического устройства проще всего воспользоваться эталонным контекстом:

```
constructor Create(ReferenceHDC: HDC; FrameRect: TGPRectF;
                    FrameUnit: TMetafileFrameUnit = MetafileFrameUnitGdi;
                    type_: TEmfType = EmfTypeEmfPlusDual;
                    Description: pWCHAR = nil); overload;
```

Конструктор обладает двумя обязательными и тремя опциональными параметрами. Обязательны для передачи в функцию аргументы, определяющие дескриптор контекста `ReferenceHDC` (на поверхности которого и будет записываться метафайл), а также координаты обрамляющего прямоугольника `FrameRect` (существуют две версии конструктора работающие со структурой `TGPRectF` или `TGPRect`).

Предусмотрен еще более упрощенный конструктор, решающий такую же задачу:

```
constructor Create(ReferenceHDC: HDC; type_: TEmfType = EmfTypeEmfPlusDual;
                    Description: PWCHAR = nil); overload;
```

Здесь отсутствуют параметры, определяющие единицу измерения и границы рисунка.

Все вышеизложенное можно объединить в один пример. Функция `RecordMetafile()` создает векторный рисунок, состоящий из прямой и эллипса. Для рисования метафайла применяется контекст, совместимый с текущими настройками экрана.

```
Function RecordMetafile:TGPMetafile;
var GPGraphics:TGPGraphics;
    ReferenceHDC:HDC;
    GPPen:TGPPen;
begin
ReferenceHDC:=CreateCompatibleDC(0);

Result:=TGPMetafile.Create(ReferenceHDC,
                           MakeRect(0,0,100,100),MetafileFrameUnitPixel);
GPGraphics:=TGPGraphics.Create(Result);

GPPen:=TGPPen.Create(aclRed,1);
GPGraphics.DrawLine(GPPen,0,0,100,100);
GPGraphics.DrawEllipse(GPPen,5,5,50,100);

GPPen.Free;
GPGraphics.Free;
DeleteDC(ReferenceHDC);
end;
```

Комментарии к метафайлу

В распоряжении класса TGPGraphics имеется метод, позволяющий дополнять метафайл текстовым комментарием.

```
function AddMetafileComment(pData: PBYTE; SizeData: UINT): TStatus;
```

Содержание комментария передается в метод посредством указателя pData, количество байт данных необходимо указать во втором параметре функции.

Воспроизведение метафайла

В главе 18 «Расширенный метафайл EMF» мы уже рассмотрели способ построчного воспроизведения метафайла средствами Windows GDI. Тогда основу кода составляли такие функции Windows API, как EnumEnhMetaFile() и PlayEnhMetaFileRecord(), отвечающие, соответственно, за пошаговое перечисление записей метафайла и отправку каждой из записей на устройство графического вывода.

Экземпляр класса TGPMetafile лишен способности самостоятельно провести декомпозицию загруженного в него метафайла на записи. Указанный функционал разработчики GDI+ возложили на дружный коллектив методов EnumerateMetafile, принадлежащих холсту TGPGraphics. В анонсированном коллективе целых 12 вариантов перегружаемых версий метода, отличающихся друг от друга незначительными нюансами. Дабы не утомлять читателя, из 12 мы выберем всего 3 метода, отсортировав их по мере возрастания возможностей. Простейший вариант выглядит следующим образом:

```
function EnumerateMetafile(Metafile: TGPMetafile; const DestPoint: TGPPointF;
                           pCallback: EnumerateMetafileProc;
                           pCallbackData: pointer = nil;
                           IA: TGPImageAttributes = nil): TStatus; overload;
```

Здесь Metafile – подлежащий выводу метафайл; DestPoint – координата левой верхней точки выводимого рисунка; pCallback – указатель на процедуру обратного вызова; pCallbackData – указатель на дополнительные данные (через него, как правило, передается ссылка на объект TGPMetafile); IA – дополнительные атрибуты изображения. В рассмотренном методе совершенно не уделяется внимания одной из самых положительных черт метафайла – возможности масштабирования векторного рисунка без искажений. Поэтому сразу перейдем к версии функции, позволяющей устранить эту несправедливость:

```
function EnumerateMetafile(Metafile: TGPMetafile; const DestRect: TGPRectF;
                           const SrcRect: TGPRectF; SrcUnit: TUnit;
                           pCallback: EnumerateMetafileProc;
                           pCallbackData: Pointer = nil;
                           IA: TGPImageAttributes = nil): TStatus; overload;
```

В отличие от предыдущего метода (определяющего место вывода рисунка одной-единственной координатой верхнего левого угла), в данном случае требуется указать прямоугольную область DestRect, указывающую не только место, но и геометрические размеры результирующего изображения. Но на

этом особенности функции не заканчиваются. Разработчики пошли дальше и позволили нам выводить только необходимую часть исходного метафайла, для этого в параметр SrcRect необходимо передать границы интересующей нас области. Параметр SrcUnit определяет единицу измерения, используемую в метафайле.

К разряду наиболее сложных методов перебора записей метафайла стоит отнести функцию:

```
function EnumerateMetafile(Metafile: TGPMetafile; pDestPoints: PGPPointF;
                           Count: Integer;
                           Const SrcRect: TGPRectF; SrcUnit: TUnit;
                           pCallback: EnumerateMetafileProc;
                           CallbackData: Pointer = nil;
                           IA: TGPImageAttributes = nil): TStatus; overload;
```

Особенность метода в том, что здесь область-получатель может представлять собой параллелограмм, а не прямоугольник. Другими словами, нам предоставляется возможность деформировать рисунок. Для определения вершин достаточно трех точек (указатель pDestPoints), координаты четвертой точки функция вычислит сама. Число точек в массиве вершин следует передать в параметр Count.

Синтаксис заголовка прототипа функции обратного вызова как всегда строг и однозначен для всех перечисленных выше методов:

```
EnumerateMetafileProc = function(RecordType: EmfPlusRecordType;
                                 Flags: UINT; DataSize: UINT; Data: PBYTE;
                                 CallbackData: pointer): BOOL; stdcall;
```

Внутри функции обратного вызова обращаемся к еще одному методу класса TGPMetafile:

```
function PlayRecord(RecordType: TEmfPlusRecordType;
                    Flags,DataSize: UINT; pData: PBYTE): TStatus;
```

Он позволяет проигрывать метафайл построчно. Здесь параметр RecordType определяет тип записи метафайла, описываемый перечислением TEmfPlusRecordType, состоящим из двух с половиной сотен элементов. Такое количество элементов объясняется тем, что в перечислении учтены записи, применяемые в WMF, EMF и EMF+. Параметр Flags содержит множество флагов, определяющих атрибуты воспроизводимой записи. Аргумент DataSize определяет количество байт, необходимых для описания отдельной записи, на которую ссылается указатель pData.

Рассмотрим пример построчного воспроизведения. В ходе следующего листинга нами создается векторный рисунок в формате EMF+, он содержит всего три визуальных элемента: прямоугольник, эллипс и текстовую строку. Создав метафайл, мы дважды выводим его на графическом устройстве. Первое воспроизведение – обычное (за счет обращения к методу DrawImage), второе воспроизведение осуществляется с помощью метода EnumerateMetafile().

```
procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics,MetaGraphics:TGPGraphics;
```

```
    GPMetafile:TGPMetafile;
    GPPen:TGPPen;
    memDC:HDC;
    GPFont:TGPFont; GPSolidBrush:TGPSolidBrush;
    GPPoint:TGPPointF;
begin
//записываем метафайл
memDC:=CreateCompatibleDC(0);

  GPMetafile:=TGPMetafile.Create(memDC,EmfTypeEmfPlusOnly);
  MetaGraphics:=TGPGraphics.Create(GPMetafile);
  GPPen:=TGPPen.Create(aclBlue,1);
  MetaGraphics.DrawRectangle(GPPen,5,5,160,40); //прямоугольник
  MetaGraphics.DrawEllipse(GPPen,5,5,160,40);   //эллипс

  GPFont:=TGPFont.Create('Arial',20);
  GPSolidBrush:=TGPSolidBrush.Create(aclRed);
  GPPoint.X:=10; GPPoint.Y:=10;
  MetaGraphics.DrawString('Метафайл',8,GPFont,GPPoint,GPSolidBrush); //текст


  GPPen.Free; GPSolidBrush.Free; MetaGraphics.Free;
DeleteDC(memDC);
//конец записи метафайла

GPGraphics:=TGPGraphics.Create(DC);
GPGraphics.DrawImage(GPMetafile, 10, 10); //обычное воспроизведение
GPPoint.X:=200;
{построчное воспроизведение}
GPGraphics.EnumerateMetafile(GPMetafile,GPPoint,@EnumerateMetafileProc,
                             @GPMetafile,nil);

GPMetafile.Free;
GPGraphics.Free;
end;
```

Настала очередь функции обратного вызова. Именно она помогает методу `EnumerateMetafile()` перебирать содержимое метафайла.

```
Function EnumerateMetafileProc(RecordType: EmfPlusRecordType; Flags: UINT;
                               DataSize: UINT; data: PBYTE;
                               callbackData: pointer): BOOL; stdcall;
begin
if RecordType<>EmfPlusRecordTypeDrawEllipse then
{запрет на воспроизведение эллипса}
   TGPMetafile(callbackData^).PlayRecord(RecordType,flags,DataSize,Data);
end;
```

На этом отрезке кода демонстрируется один из способов управления процессом воспроизведения – мы отказываемся от вывода эллипса. Результаты работы представлены на рис. 37.3, это снимок окна, в левой части которого наш метафайл нарисован с помощью стандартного метода `DrawImage()`, а в правой – средствами функции обратного вызова.

Рис. 37.3. Управление воспроизведением метафайла

Методы построчного воспроизведения метафайла нужны не столько для вывода изображения на поверхности графического устройства, сколько для построения программных модулей, предназначенных для редактирования векторного рисунка.

Минимальное разрешение битовых образов метафайла

Современный метафайл способен хранить не только перечень обращений к функциям GDI и GDI+, но и при необходимости в состоянии включать битовые образы. Симбиоз между различными способами описания изображений необходим, например, при формировании кистей, основанных на растровом рисунке. Для управления минимальным разрешением образа используют метод:

```
function SetDownLevelRasterizationLimit(RasterizationLimitDpi: UINT): TStatus;
```

Функция устанавливает нижний предел разрешения в точках на дюйм одновременно для всех входящих в метафайл битовых образов.

Метод может быть применен для всех типов метафайлов EMF, за исключением метафайла в формате `EmfTypeEmfPlusOnly`.

Для выяснения текущих настроек метафайла обращайтесь к методу:

```
function GetDownLevelRasterizationLimit: UINT;
```

По умолчанию минимальное разрешение битовых образов соответствует экранному представлению 96 dpi.

РЕЗЮМЕ

Если разрабатываемая программа не просто использует готовые растровые и векторные изображения, но и должна уметь самостоятельно формировать эти изображения, то нам не обойтись без помощи объектов, построенных на основе `TGPBitmap` и `TGPMetafile`. Указанные классы представляют собой логическое развитие `TGPImage` и специализируются, соответственно, на работе с растровой и векторной графикой.

38

Работа со шрифтами в GDI+

Работа с текстом – едва ли не самая сложная задача из всех задач, встречающихся в двумерной графике. Для достижения, казалось бы, тривиальной цели – вывода в окне строки «Hello, World!» задействуется вся мощь графических библиотек Windows. В процесс вовлекаются как простейшие объекты и функции GDI, так и сложнейшие механизмы отсечения, переноса координат, мировых преобразований и, конечно же, самый необходимый ингредиент – шрифт.

К вопросу работы со шрифтами технология GDI+ подошла со свойственным ей размахом – обновленная библиотека включает пять классов, нацеленных на облегчение жизни программиста. Основной из них – класс `TGPFont` – специализируется на обслуживании экземпляра логического шрифта. Спрятав экземпляр шрифта в объектно-ориентированную оболочку, разработчики GDI+ не успокоились и реализовали ряд вспомогательных классов:

- Это класс `TGPFontFamily`, специализирующийся на обслуживании подмножества шрифтов одного семейства.
- Три коллекции (`TGPFontCollection`, `TGPInstalledFontCollection` и `TGPPrivateFontCollection`), способные хранить несколько подмножеств шрифтов разных семейств.

Удалось ли объектно-ориентированной модели GDI+ кардинально улучшить работу со шрифтами? Безусловно, но отголоски GDI по-прежнему весьма заметны. Так, при создании экземпляра логического шрифта опорный класс `TGPFont` (точно так же, как и знаменитые функции `CreateFont()` и `CreateFontInderect()` из состава обычного Windows API) по-прежнему опирается на структуру `TLongFont` с ее более чем десятком полей. От такой зависимости никуда не деться, ведь с технической точки зрения GDI+ – не больше, чем надстройка над «устаревшим» графическим механизмом. Правда, в ряде конструкторов связь с технологией GDI несколько закамуфлирована, но суть от этого не изменилась. Что касается вспомогательных классов, то у них явных аналогов в GDI нет. Благодаря им несколько упрощается процесс разра-

ботки приложений, для нормальной жизнедеятельности которых требуется более чем одно семейство шрифтов.

Шрифт, класс TGPFont

В рамках GDI+ экземпляр логического шрифта инкапсулируется в классе TGPFont. В технологию создания объекта шрифта GDI+ практически не внес никаких существенных усовершенствований – весь механизм полностью основан на базовых функциях GDI. Вполне естественно, что несколько упростился процесс программирования, но это скорее следствие применения объектно-ориентированного подхода, а не какого-то новаторского решения разработчиков класса. Впрочем, умение грамотно переносить наиболее положительные черты ООП на графический «движок» Windows само по себе заслуживает отдельных похвал.

Все как всегда начинается с конструкторов. Простейший из них для ввода в строй экземпляра класса TGPFont удовлетворится единственным параметром – дескриптором контекста устройства.

```
constructor Create(DC: HDC); reintroduce; overload;
```

Вновь рожденный экземпляр класса создаст логический шрифт по образу и подобию текущего шрифта, выбранного в контекст устройства GDI.

Еще один конструктор позволяет вдохнуть жизнь в объект, копирующий характеристики шрифта с дескриптором hFont:

```
constructor Create(DC: HDC; hFont: HFONT); reintroduce; overload;
```

Использование представленной выше пары конструкторов имеет скорее теоретическое, чем прикладное значение – слишком многоходовая комбинация необходима для получения элементарного результата.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPFont:TGPFont;
    GPSolidBrush:TGPSolidBrush;
    GPPoint:TGPPointF;
begin
GPGraphics:=TGPGraphics.Create(DC);

SelectObject(DC,GetStockObject(ANSI_FIXED_FONT));//выбор шрифта в контекст
GPFont:=TGPFont.Create(DC);//создание экземпляра шрифта
GPSolidBrush:=TGPSolidBrush.Create(aclRed);
GPPoint.X:=10; GPPoint.Y:=10;
GPGraphics.DrawString('Hello, World!',13,GPFont,GPPoint,GPSolidBrush);

GPFont.Free; GPSolidBrush.free;
GPGraphics.Free;
end;
```

Согласитесь, что для того чтобы вывести строку системным шрифтом, проще воспользоваться только функциями GDI, чем будить более ресурсоемкий GDI+.

Создание объекта может производиться за счет классической структуры TLogFont, о которой неоднократно упоминалось в главе 19 «Шрифты». Для этого в рамках GDI+ используют структуры TLogFontA (для набора ANSI, в котором отдельный символ кодируется одним байтом) или TLogFontW (двухбайтовая кодировка для набора Unicode). Описав поля структуры соответствующего типа, направляйте указатель на запись во второй параметр конструктора:

```
constructor Create(DC: HDC; Logfont: PLogFontA); reintroduce; overload;
constructor Create(DC: HDC; Logfont: PLogFontW); reintroduce; overload;
```

Кроме огромной, описывающей шрифт структуры, конструктор рассчитывает получить дескриптор контекста устройства. На мой взгляд, такое решение разработчиков GDI+ ненамного эффективнее, чем формирование логического шрифта средствами обычной функции GDI CreateFontIndirect(), к тому же последняя обходится единственным параметром... Но критиковать проще всего, поэтому продолжим исследование класса, задавшись целью найти положительные стороны у двух оставшихся конструкторов. К счастью, на этот раз это не так сложно.

Начнем с конструктора, создающего экземпляр логического шрифта, опираясь на поддержку вспомогательного класса TGPFontFamily:

```
constructor Create(Family: TGPFontFamily; emSize: Single;
                   Style: TFontStyle = FontStyleRegular;
                   Unit_: TUnit = UnitPoint); reintroduce; overload;
```

Экземпляр класса TGPFontFamily *описывает множество шрифтов одного семейства. В простейшем случае для создания объекта достаточно передать в его конструктор название семейства.*

В конструкторе имеются два обязательных и два опциональных параметра. К обязательным относятся параметр Family (назначающий семейство для вновь создаваемого шрифта) и параметр emSize (указывающий высоту шрифта). Единица измерения зависит от состояния опциального параметра Unit (табл. 38.1), который по умолчанию установлен в состояние UnitPoint.

Таблица 38.1. Элементы перечисления TUnit

Элемент	Значение	Описание
UnitWorld	0	Логическая единица в текущих мировых координатах.
UnitDisplay	1	Экранный пиксел.
UnitPixel	2	Пиксел.
UnitPoint	3	Пункт (1/72 дюйма).
UnitInch	4	Дюйм.
UnitDocument	5	1/300 дюйма.
UnitMillimeter	6	Миллиметр.

Особенности начертания создаваемого логического шрифта определяются параметром Style. Он набирается из флагов FontStyle:

```
type FontStyle = Integer;
  const
    FontStyleRegular    = Integer(0); //нормальный
    FontStyleBold       = Integer(1); //утолщенный
    FontStyleItalic     = Integer(2); //курсив
    FontStyleBoldItalic = Integer(3); //утолщенный + курсив
    FontStyleUnderline  = Integer(4); //подчеркнутый
    FontStyleStrikeout  = Integer(8); //перечеркнутый
type TFontStyle = FontStyle;
```

Практический пример по созданию логического шрифта средствами рассмотренного выше конструктора представлен ниже.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPFontFamily :TGPFontFamily;
    GPFont:TGPFont;
    GPBrush:TGPLinearGradientBrush;
begin
GPGraphics:=TGPGraphics.Create(DC);

GPBrush:= TGPLinearGradientBrush.Create(MakePoint(0,0),MakePoint(0,100),
                                        aclBlue, aclLightGreen);

GPFontFamily:=TGPFontFamily.Create('ARIAL');
GPFont:=TGPFont.Create(GPFontFamily,80);
GPGraphics.DrawString('Класс TGPFont',-1,GPFont,MakePoint(0,0.0),GPBrush);
GPFont.Free; GPFontFamily.Free; GPBrush.Free;
GPGraphics.Free;
end;
```

Нами формируется шрифт гарнитуры «Arial» высотой 80 пунктов. Для того чтобы вы не забыли о существовании кистей с градиентной заливкой, вывод текстовой строки производится именно такой кистью.

Последний (и наиболее функциональный) из имеющихся в распоряжении класса TGPFont конструкторов не требует от программиста явным образом задействовать вспомогательный класс TGPFontFamily.

```
constructor Create(FamilyName: WideString;
                   emSize: Single;
                   Style: TFontStyle = FontStyleRegular;
                   Unit_: TUnit = UnitPoint;
                   FontCollection: TGPFontCollection = nil);
                   reintroduce; overload;
```

В метод разрешено сразу направлять имя семейства шрифта (параметр FamilyName), а создание экземпляра TGPFontFamily осуществляется в недрах конструктора точно так, как было продемонстрировано в предыдущем примере.

Все имеющиеся в распоряжении класса TGPFont методы (за исключением Clone) справочные – они информируют разработчика о характеристиках вновь созданного экземпляра логического шрифта (табл. 38.2).

Таблица 38.2. Информационные методы класса TGPFont

Метод	Описание
`function IsAvailable: BOOL;`	Проверка на доступность объекта шрифта.
`function GetFamily(Family: TGPFontFamily): TStatus;`	Метод возвращает семейство шрифта.
`function GetLogFontA(G: TGPGraphics; out LogFontA: TLogFontA): TStatus;`	Заполняет поля структуры `TLogFontA` данными шрифта.
`function GetLogFontW(G: TGPGraphics; out LogFontW: TLogFontW): TStatus;`	Заполняет поля структуры `TLogFontW` данными шрифта.
`function GetStyle: Integer;`	Стиль шрифта.
`function GetUnit: TUnit;`	Информирует о единице измерения шрифта (см. табл. 38.1).
`function GetSize: Single;`	Возвращает размер шрифта в единицах измерения данного объекта `TGPFont`.
`function GetHeight(graphics: TGPGraphics): Single; overload;`	Размер между базовыми линиями двух текстовых строк, размещенных друг над другом. Единица измерения выясняется методом `GetUnit()`.
`function GetHeight(DPI: Single): Single; overload;`	Размер между базовыми линиями двух текстовых строк, расположенных друг над другом. Единица измерения – число точек в дюйме.

Семейство шрифтов, класс TGPFontFamily

Экземпляр класса `TGPFontFamily` специализируется на хранении множества шрифтов одного семейства. Класс обладает двумя конструкторами:

```
constructor Create; reintroduce; overload;
constructor Create(Name: WideString;
                   FontCollection: TGPFontCollection = nil); reintroduce; overload;
```

Первый из конструкторов формирует пустое семейство (этой особенностью класса мы уже пользовались в прошлом примере). Вторая вариация конструктора создает реальное семейство. Для этого методу необходимо указать название гарнитуры (параметр `Name`) и при желании передать ссылку на экземпляр коллекции шрифтов `FontCollection`.

Кроме конструкторов в процессе создания нового семейства не прочь поучаствовать трио функций класса:

```
class function GenericMonospace: TGPFontFamily;
class function GenericSansSerif: TGPFontFamily;
class function GenericSerif: TGPFontFamily;
```

Функции помогут нам при формировании экземпляра шрифта заранее предопределенного вида (рис. 38.1). С программной реализацией проблем воз-

Рис. 38.1. Начертание предопределенных шрифтов

никнуть не должно. Для обращения к функциям подобного рода даже нет необходимости в создании экземпляра класса. Например, следующая строка листинга формирует моноширинный шрифт высотой в 40 единиц за счет вызова метода `GenericMonospace()`:

```
GPFont:=TGPFont.Create(TGPFontFamily.GenericMonospace,40);
```

В табл. 38.3 представлена информация о методах семейства шрифтов, информирующих программиста о наиболее востребованных метриках семейства.

Таблица 38.3. Информационные методы класса TGPFamily

Метод	Описание
`function IsAvailable: BOOL;`	Проверяет факт доступности экземпляра класса.
`function GetFamilyName(out Name: String; language: LANGID = 0): TStatus;`	Возвращает название семейства в параметр `Name`.
`function IsStyleAvailable(style: Integer): BOOL;`	Проверяет факт доступности в семействе шрифта с заданным стилем. Стиль задается в соответствии с типом `TFontStyle`.
`function GetEmHeight(style: Integer): UINT16;`	Высота шрифта заданного стиля.
`function GetCellAscent(style: Integer): UINT16;`	Величина надстрочного интервала.
`function GetCellDescent(style: Integer): UINT16;`	Величина подстрочного интервала.
`function GetLineSpacing(style: Integer): UINT16;`	Расстояние между базовыми линиями двух строк, расположенных друг над другом.

Коллекции шрифтов

Если приложение одновременно нуждается в услугах нескольких шрифтов, то наиболее рациональным решением станет применение в проектах массивов или списков, способных одновременно обслуживать некоторый перечень однотипных объектов. Чтобы несчастные программисты не ломали голову над изобретением очередного велосипеда, разработчики GDI+ предложили нам три класса коллекций шрифтов (рис. 38.2).

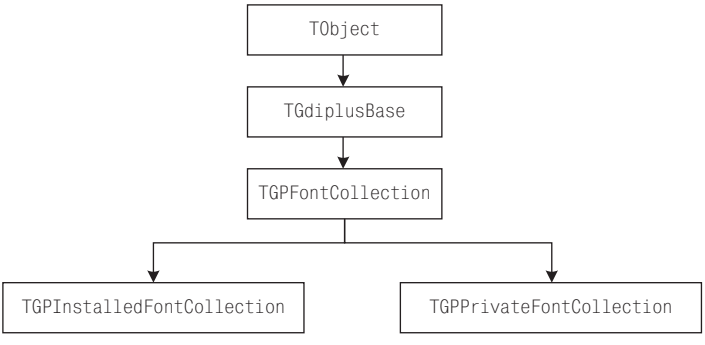


Рис. 38.2. Коллекции шрифтов GDI+

Основу иерархии составляет абстрактный класс TGPFontCollection, а на его основе построены коллекция TGPInstalledFontCollection (специализирующаяся на хранении установленных в ОС шрифтов) и коллекция TGPPrivateFontCollection (позволяющая нам строить индивидуальный список наиболее предпочтительных шрифтов).

На уровне абстрактного класса нас интересуют только два метода. Это функция:

```
function GetFamilyCount: Integer;
```

информирующая нас о количестве элементов, упрятанных в коллекцию. И функция:

```
function GetFamilies(NumSought: Integer; out GPfamilies: array of TGPFontFamily;
                     out NumFound: Integer): TStatus;
```

позволяющая извлечь из коллекции данные о содержащихся в ней шрифтах и поместить их в массив GPFamilies. Перед обращением к методу необходимо позаботиться об инициализации массива (установке размеров и заполнении каждой ячейки экземпляром класса TGPFontFamily). Параметр NumSought укажет, сколько семейств шрифтов мы намерены собрать. По завершении работы метода параметр NumFound возвратит количество реально возвращенных семейств.

Заполнение коллекции шрифтами происходит по-разному. Так, класс TGPInstalledFontCollection приобретает тайные знания при обращении к конструктору. В этот момент создаваемый объект автоматически собирает данные обо всех доступных в Windows семействах шрифтов. Порядок работы с данным классом представлен в очередном фрагменте кода.

```
VAR //…
  FC:TGPInstalledFontCollection;
  i,Count,NumFound: integer;
  s:string;
  GPfamilies: array of TGPFontFamily;

function WindowProc(Window: HWND; Msg: UINT;
```

```
                    wParam: WPARAM; lParam: LPARAM) : LongInt; stdcall;
begin
case Msg of
  WM_CREATE : begin
          COMBOBOX:= CreateWindow('COMBOBOX', '',
                WS_CHILD OR WS_BORDER OR WS_VISIBLE OR BS_LEFT OR LBS_STANDARD,
                0, 0, 0, 0, Window, 0, HInstance, nil);

                FC:=TGPInstalledFontCollection.Create;
                Count:=FC.GetFamilyCount;
                SetLength(gpfamilies,Count);

                 for i:=0 to Count-1 do gpfamilies[i]:=TGPFontFamily.Create;

                 FC.GetFamilies(Count,gpfamilies,NumFound);

                 for i:=0 to Count-1 do
                 begin
                  gpfamilies[i].GetFamilyName(s);
                  SendMessage(COMBOBOX,CB_ADDSTRING,0,INTEGER(pChar(s+#0)));
                 end;
                  SendMessage(ComboBox,CB_SETCURSEL,0,0);
                  FC.Free;
                 for i:=0 to Count-1 do gpfamilies[i].Destroy;
                 end;
//…
```

Код сосредоточен в рамках обработки сообщения `WM_CREATE`. Нами создается комбинированный список с дескриптором `COMBOBOX` и экземпляр класса `TGPInstalledFontCollection`. Затем, выяснив число элементов в коллекции, мы устанавливаем размеры массива `GPFamilies` и заполняем его пустыми объектами `TGPFontFamily`. На заключительном этапе осталось обратиться к методу `GetFamilies()` и передать имена найденных семейств шрифтов в строки комбинированного списка (рис. 38.3).

В отличие от класса, посвятившего все свое существование обслуживанию установленных шрифтов, частная коллекция `TGPPrivateFontCollection` специализируется на хранении пользовательского перечня шрифтов. Поэтому после вызова конструктора коллекция создается пустой, а ее заполнение производится в ручном режиме. Для этого объявлены два метода:

```
function AddFontFile(Filename: WideString): TStatus;
function AddMemoryFont(pMemory: Pointer; length: Integer): TStatus;
```

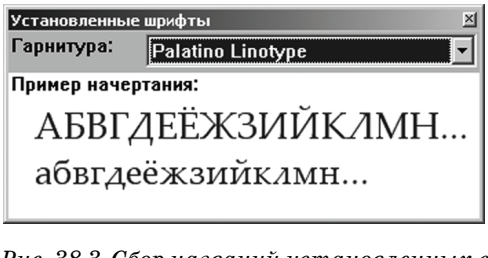


Рис. 38.3. Сбор названий установленных семейств шрифтов

Первая функция внедряет в коллекцию шрифт из файла с именем `Filename`. Вторая функция подгружает шрифт из области памяти, на которую ссылается указатель `pMemory`. Параметр `length` определяет размер загружаемых данных.

Основное достоинство частной коллекции шрифтов `TGPPrivateFontCollection` *в том, что она способна инкапсулировать не только инсталлированные в ОС, но и незарегистрированные в системе шрифты. Таким образом, программист приобретает возможность научить приложение пользоваться индивидуальными шрифтами.*

РЕЗЮМЕ

Для работы со шрифтами в GDI+ создано пять классов. Основной из них – `TGPFont`; он предназначен для инкапсуляции логического шрифта GDI. Кроме того, в нашем распоряжении имеются классы, играющие вспомогательную роль. Это класс `TGPFontFamily`, отвечающий за инкапсуляцию шрифтов одного семейства, и коллекции шрифтов `TGPFontCollection`, `TGPInstalledFontCollection` и `TGPPrivateFontCollection`, которые позволят нашим приложениям работать с некоторым перечнем шрифтов.

39

Операции с текстом в GDI+

В предыдущей главе мы познакомились со способами описания характеристик логических шрифтов в GDI+. Благодаря им определяются гарнитура, стиль начертания, высота и многие другие, в конечном счете влияющие на внешний вид текстовых данных, параметры. Но шрифт – это всего лишь инструмент, который сам по себе бесполезен. Результата можно ожидать лишь тогда, когда экземпляр класса `TGPFont` попадет в руки к холсту GDI+.

Именно холст инкапсулирует функции GDI, отвечающие за вывод текстовых данных. Получив столь ответственную партию первой скрипки в графическом оркестре GDI+, класс `TGPGraphics` не остался простым исполнителем. Возможности GDI+ по работе с текстовыми данными значительно усовершенствованы, это работа в мировой системе координат, поддержка прозрачного вывода, улучшенное качество отображения и многое-многое другое. Но если внимательно вслушаться в, казалось, новую симфонию, то можно отчетливо услышать мотивы, звучавшие еще во времена обычного GDI (касательно вывода текста это отголоски функции `DrawTextEx`). Это еще одно подтверждение, что и сегодня Microsoft не торопится расстаться с написанными еще во времена Windows 95 партитурами...

Методы TGPGraphics по выводу текста

Принцип самодостаточности функций GDI+ сполна реализован и в методах, нацеленных на вывод текстовых строк. Они требуют от программиста определить достаточно большой набор параметров, начиная собственно с текста и заканчивая описаниями логического шрифта и кисти, которой будут залиты контуры символов. Для вывода текстовой строки необходимо воспользоваться услугами перегружаемого метода `DrawString()`. Указанный метод работает с тремя вариантами параметров. Первая по счету вариация метода наиболее проста, для нормального самочувствия ей требуется 5 параметров:

```
function DrawString(S _: WideString; Length: Integer;
```

```
    Font: TGPFont; const Origin: TGPPointF;
    Brush: TGPBrush): TStatus; overload;
```

Это подлежащий выводу текст S, количество отображаемых символов Length, характеристики шрифта Font, координаты начала вывода Origin и кисть Brush, которой будут залиты глифы символов. Фактически это аналог хорошо нам знакомой функции TextOut(), с той лишь разницей, что в нашем методе для вывода текста применяется обладающая весьма внушительным спектром заливок и узоров кисть. Так что по навыкам по «покраске» текста эта версия метода существенно превосходит своего дальнего родственника.

Передав в параметр Length *метода* DrawString() *значение -1, мы укажем функции, что хотим вывести все символы строки.*

Ниже представлена процедура, использующая двумерное аффинное преобразование для вывода текста с тенью с помощью двух обращений к методу DrawString().

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPFont:TGPFont;
    GPSolidBrush:TGPSolidBrush;
    GPPoint:TGPPointF;
    GPMatrix:TGPMatrix;
const S='Hello, World!';
begin
GPGraphics:=TGPGraphics.Create(DC);
GPFont:=TGPFont.Create('Arial',50);

GPPoint:=MakePoint(10,50.0); //координата вывода
{ВЫВОД ТЕНИ}
GPSolidBrush:=TGPSolidBrush.Create(aclSilver);    //кисть для тени
GPMatrix:=TGPMatrix.Create(1,0,-1.1,1.5,120,-60); //преобразование
GPGraphics.SetTransform(GPMatrix);                //применяем матрицу
GPGraphics.DrawString(S,13,GPFont,GPPoint,GPSolidBrush); //вывод текста
GPSolidBrush.free;                 //кисть для тени больше не нужна
{ВЫВОД ТЕКСТА}
GPMatrix.Reset;                    //сброс матрицы
GPGraphics.SetTransform(GPMatrix); //восстанавливаем обычный режим холста
GPSolidBrush:=TGPSolidBrush.Create(aclBlack); //кисть для вывода текста
GPGraphics.DrawString(S,13,GPFont,GPPoint,GPSolidBrush); //текст
GPSolidBrush.Free; GPFont.Free; GPGraphics.Free;
end;
```

Все весьма элегантно и одновременно просто. Сначала мы имитируем тень, для этого применяем к холсту GDI+ матрицу аффинных преобразований с параметрами, деформирующими и сдвигающими текстовую строку. Нарисовав тень, выводим поверх нее текстовую строку в обычном режиме (рис. 39.1).

Рис. 39.1. Мировые преобразования при выводе текста

Вновь возвращаемся к методу `DrawString()`, точнее, к двум оставшимся его реализациям. Они позволяют осуществить сложное форматирование текста, по своим возможностям сопоставимое с функцией `DrawTextEx()` из состава GDI:

```
function DrawString(S _: WideString; Length: Integer; Font: TGPFont;
                    const Origin: TGPPointF; StringFormat: TGPStringFormat;
                    Brush: TGPBrush): TStatus; overload;

function DrawString(S_: WideString; Length: Integer; Font: TGPFont;
                    const LayoutRect: TGPRectF; StringFormat: TGPStringFormat;
                    Brush: GPBrush): TStatus; overload;
```

В этих методах `DrawString()` появляется дополнительный параметр `StringFormat`. Это экземпляр класса `TGPStringFormat`, предназначенный для описания особенностей форматирования текстовой строки (ему посвящен следующий подраздел этой главы). Вместо координат точки, определяющих местоположение левой верхней точки строки, разрешено задавать прямоугольную область `LayoutRect` (третья версия метода), в пределах которой будет производиться вывод.

```
Procedure GDI_PAINT(DC:HDC);
var GPFont:TGPFont;
    GPGraphics:TGPGraphics;
    Brush:TGPSolidBrush;
    GPPointF:TGPPointF;
const S='Hello, World!';
begin
GPGraphics:=TGPGraphics.Create(DC);
Brush:=TGPSolidBrush.Create(aclBlack);

  GPPointF:=MakePoint(10,10.0);
  GPFont:=TGPFont.Create(TGPFontFamily.GenericMonospace,20);
  GPGraphics.DrawString(s,Length(s),gpFont,GPPointF,Brush);
  GPFont.Free;

Brush.Free; GPGraphics.free;
end;
```

Теперь настал черед поговорить о порядке форматирования текста, выводимого с помощью метода `DrawString()`.

Форматирование текстовой строки, класс TGPStringFormat

По сути, экземпляр класса TGPStringFormat представляет собой хранилище атрибутов форматирования, которые могут быть применены к текстовой строке. Атрибуты схожи с флагами форматирования, которые встретились нам в функции DrawText() из состава GDI. Основной конструктор класса содержит два параметра:

```
Constructor Create(FormatFlags: Integer = 0;
                   Language: LANGID = LANG_NEUTRAL); reintroduce; overload;
```

Первый параметр представляет собой комбинацию форматирующих флагов (табл. 39.1). В момент создания объекта ему присваиваются атрибуты по умолчанию – все флаги сняты. Второй параметр рассчитывает получить 16-битный языковой идентификатор. По умолчанию предлагается сохранить системные настройки (константа LANG_NEUTRAL). Для смены языка следует воспользоваться макросом MAKELANGID(), он позволяет создать шестнадцатиразрядную комбинацию, содержащую данные об основном и дополнительном языке Windows. Например, для большинства российских компьютеров подойдет идентификатор, полученный следующим образом:

```
Language:=MAKELANGID(LANG_RUSSIAN, SUBLANG_ENGLISH_US);
```

Это означает, что основной язык ввода русский, а дополнительный – английский (в американской интерпретации).

Таблица 39.1. Флаги форматирования строки FormatFlags

Константа	Значение	Описание
StringFormatFlagsDirectionRightToLeft	$0001	Строка выводится справа налево.
StringFormatFlagsDirectionVertical	$0002	Вертикальное направление вывода строки.
StringFormatFlagsNoFitBlackBox	$0004	Определяет, будут ли при выводе отсечены части глифа, выступающие за пределы ограничивающего прямоугольника. Например, включение флага может привести к отказу системы рисовать «хвосты» символов «р» и «у», так как они находятся за пределами базовой линии (см. рис. 19.5).
StringFormatFlagsDisplayFormatControl	$0020	Разрешает использовать подстановочный символ для вывода не установленного в системе символа в кодировке Unicode (см. флаг $0400).
StringFormatFlagsNoFontFallback	$0400	Определяет альтернативный шрифт, который будет использован при выводе специальных символов, отсутствующих в основном шрифте.

Константа	Значение	Описание
StringFormatFlagsMeasureTrailing-Spaces	$0800	Работает совместно с методом MeasureString(), позволяющим рассчитать ширину и высоту текста. По умолчанию метод не принимает во внимание часть текста, вышедшую за пределы ограничивающего прямоугольника. Включив флаг, мы заставим метод исследовать даже «отрезаемую» часть строки.
StringFormatFlagsNoWrap	$1000	Запрет переноса слов, текст выводится в одну строку.
StringFormatFlagsLineLimit	$2000	Запрет на вывод очередной строки, если она не помещается в рамках области вывода.
StringFormatFlagsNoClip	$4000	Прекращает обрезку очередной строки, даже если она не помещается в области вывода.

Макрос MAKELANGID() *объявлен в заголовочном файле WinNT.h. К сожалению, в составе Delphi (даже включая современную версию от CodeGear) аналога данного файла нет, поэтому при необходимости указанный макрос придется написать самим. Ничего сложного в этом нет, важно лишь знать правило формирования итогового идентификатора: значение основного языка должно находиться с 0 по 9 бит, а вторичного – с 10 по 15 биты шестнадцатеричной комбинации.*

```
Function MAKELANGID(Primary_Language_ID,Sublanguage_ID:Word):Word;
begin
Result:=(Sublanguage_ID SHL 10) OR Primary_Language_ID;
end;
```

Ниже предложен фрагмент процедуры, демонстрирующей порядок создания объекта форматирования строки.

```
Procedure GDI_PAINT(DC:HDC; S : WideString);
var GPFont:TGPFont;
    LayoutRect: TGPRectF;
    GPGraphics:TGPGraphics;
    BrushRect,BrushFont : TGPSolidBrush;
    SF:TGPStringFormat;
begin
GPGraphics:=TGPGraphics.Create(DC);
BrushRect:=TGPSolidBrush.Create(aclWhite); //кисть для заливки области
BrushFont:=TGPSolidBrush.Create(aclBlack); //кисть для вывода текста
GPFont:=TGPFont.Create(TGPFontFamily.Create('Times'),18);//шрифт

LayoutRect:=MakeRect(160,5,150,100.0); {область вывода}
GPGraphics.FillRectangle(BrushRect,LayoutRect); {выделяем область белым цветом}

SF:=TGPStringFormat.Create(StringFormatFlagsDirectionVertical);
GPGraphics.DrawString(s,Length(s),gpFont,LayoutRect,SF,BrushFont);
```

```
SF.Free; GPFont.Free; BrushFont.Free; BrushRect.Free; GPGraphics.free;
End;
```

Экспериментируя с отдельными флагами и их комбинациями, можно получить разнообразные варианты форматирования текстовой строки. Влияние флагов класса `TStringFormat` на процесс вывода текста весьма схоже с ролью параметра форматирования `Format` из состава наиболее продвинутых функций GDI: `DrawText()` и `DrawTextEx()`. Однако для вывода текста по вертикали в GDI приходилось «колдовать» с характеристиками шрифта, а здесь поворот текста на 90 градусов осуществляется без особых сложностей (рис. 39.2).

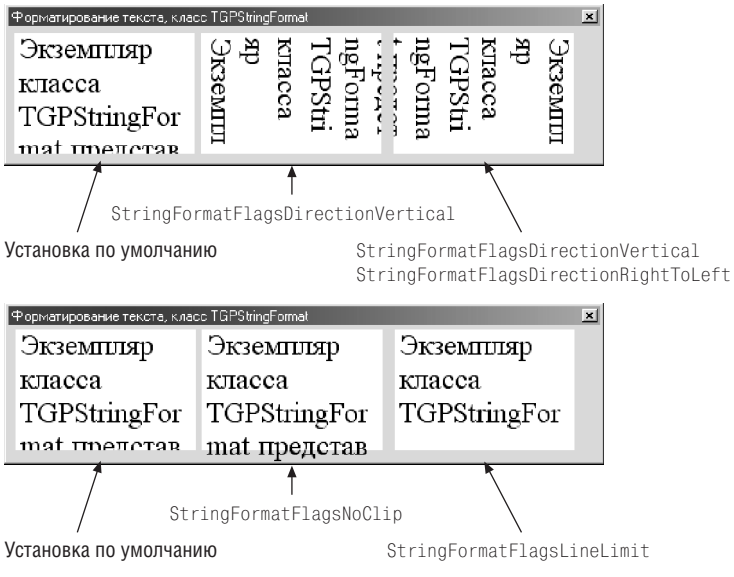


Рис. 39.2. Варианты вывода текста в прямоугольной области

Состав флагов не является неизменным и вполне может быть изменен во время жизни объекта. Для этого предназначен метод:

```
function SetFormatFlags(Flags: Integer): TStatus;
```

Рука об руку с `SetFormatFlags()` шествует метод, предназначенный для исследования уже установленных флагов объекта:

```
function GetFormatFlags: Integer;
```

Предопределенное форматирование

Исключительно для удобства разработчика предусмотрены две функции класса, позволяющие создать объект `TGPStringFormat` с заранее определенными характеристиками форматирования. Таких функций две:

```
class function GenericDefault: TGPStringFormat;
class function GenericTypographic: TGPStringFormat;
```

Первая из них формирует объект с параметрами по умолчанию (флаги сняты, выравнивание строки по левому верхнему краю ограничивающей области и т. д.). Вторая функция предлагает наиболее общие (на взгляд разработчиков класса) параметры форматирования: установлены флаги StringFormatFlagsLineLimit, StringFormatFlagsNoClip и StringFormatFlagsNoFitBlackBox, активировано выравнивание по левому краю, табулирование выключено и т. д.

Отсечение элементов строки

В той ситуации, когда текст целиком не помещается в отведенную ему прямоугольную область, программист имеет возможность отдать предпочтение одному из вариантов отсечения «лишних» элементов. Способ избавления от символов или даже слов устанавливается посредством обращения к методу:

```
function SetTrimming(Trimming: TStringTrimming): TStatus;
```

В качестве параметров SetTrimming() принимает любой из шести элементов, входящих в перечисление TStringTrimming (табл. 39.2).

Таблица 39.2. Описание перечисления TStringTrimming

Константа	Значение	Описание
StringTrimmingNone	0	Механизм отсечения по умолчанию.
StringTrimmingCharacter	1	Отсекается не вмещающийся символ.
StringTrimmingWord	2	Полностью отсекается слово.
StringTrimmingEllipsisCharacter	3	Не вмещающиеся символы слова заменяются многоточием.
StringTrimmingEllipsisWord	4	Не вмещающееся слово заменяется многоточием.
StringTrimmingEllipsisPath	5	В области выводится начало и окончание текстовой строки, многоточием замещаются символы, находящиеся в середине строки.

Способы отсечения строк GDI+ целиком и полностью позаимствовал у наиболее продвинутых функций GDI – DrawText() и DrawTextEx(). Можете проверить, но для этого придется ненадолго вернуться к табл. 20.3 и перечитать назначение флагов DT_END_ELLIPSIS, DT_WORD_ELLIPSIS, DT_PATH_ELLIPSIS и т. п. На рис. 39.3 функция SetTrimming() продемонстрирована во всей своей красе, справа от строк выведены числовые значения констант отсечения.

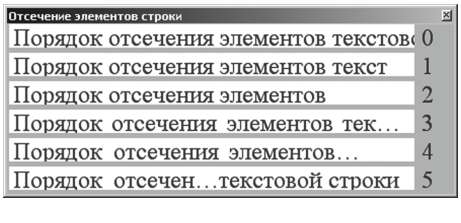


Рис. 39.3. Варианты отсечения элементов строки

В завершение истории об укрощении строк больших размеров отметим, что для изучения текущих настроек отсечения следует воспользоваться функцией:

```
function GetTrimming: TStringTrimming;
```

Выравнивание текста

Едва ли не самый популярный (и одновременно простейший) способ форматирования связан с выравниванием текста по горизонтали и вертикали. В первом случае применяется следующая пара методов:

```
function SetAlignment(Align: TStringAlignment): TStatus;
function GetAlignment : TStringAlignment;
```

Функция `Set…` устанавливает новый режим выравнивания, а функция `Get…` информирует о текущих настройках. Для выравнивания по вертикали предназначена другая пара методов:

```
function SetLineAlignment(Align: TStringAlignment): TStatus;
function GetLineAlignment: TStringAlignment;
```

Порядок выравнивания назначает параметр `Align`, представляющий собой перечисление из трех элементов:

```
type TStringAlignment =
         (StringAlignmentNear,   //по левому краю или по верху
          StringAlignmentCenter, //по центру
          StringAlignmentFar);   //по правому краю или по низу
```

Работа с префиксом «&»

По давно сложившейся традиции при разработке пользовательского интерфейса программисты реализуют возможность быстрого доступа к ключевым элементам меню и кнопкам по нажатию комбинации клавиш Alt+<Клавиша>[1]. Для того чтобы пользователь догадался, клавишу с каким символом необходимо нажать, чтобы добраться до требуемого пункта меню, этот символ в заголовке элемента управления выводится подчеркнутым, например «File». При разработке приложения для предупреждения Windows, что именно эта буква должна выводиться с нижним подчеркиванием, перед ней ставят символ амперсанда «&», например «&File».

Вместе с тем вопрос вывода подчеркнутого символа в заголовке элемента управления – дело творческое. А вдруг амперсанд «&» является полноправным членом текстовой строки, а не признаком клавиши быстрого доступа? Специально для таких случаев предназначен метод:

```
function SetHotkeyPrefix(HotkeyPrefix: THotkeyPrefix): TStatus;
```

Функция позволяет устанавливать три варианта реакции системы на амперсанд.

---

[1] Обычно такие комбинации называют «горячими клавишами».

```
type THotkeyPrefix = (
    HotkeyPrefixNone, //воспринимать «&» как обычный символ строки
    HotkeyPrefixShow, //подчеркивать следующий за префиксом символ
    HotkeyPrefixHide  //не подчеркивать следующий за префиксом символ
  );
```

По умолчанию в момент создания экземпляра класса TGPStringFormat выбирается первый режим HotkeyPrefixNone().

Для исследовательских целей предназначен метод:

```
function GetHotkeyPrefix: THotkeyPrefix;
```

Он расскажет нам, какой режим восприятия амперсанда установлен в данное время.

Вывод табулированного текста

Табулирование предоставляет простой механизм позиционирования слов в строке, например, табуляция позволяет легко превратить текст в многоколоночную таблицу:

```
function SetTabStops(FirstTabOffset: Single;
                     Count: Integer; pTabStops: PSingle): TStatus;
```

Здесь параметр FirstTabOffset определяет начальное положение для вывода, если это ноль, то вывод текста начинается непосредственно с левой границы области вывода. Значения отступов табуляции хранятся в массиве вещественных чисел, на который ссылается указатель pTabStops. Число элементов в массиве передается через второй параметр Count. Для того чтобы функция DrawString() поняла, к каким именно элементам строки требуется применять табуляцию, в соответствующие места подлежащего выводу текста вставляются коды табуляции (в таблице символов ASCII – #9).

Заметьте, что здесь система определения отступов несколько отличается от идеи, заложенной в GDI (функция TabbedTextOut). Напомню, что в GDI каждый элемент массива отступов содержал расстояние от начальной точки вывода табулированной строки до очередного фрагмента помеченного таб-стопом текста. Другими словами, позиция каждого последующего отступа являлась суммой значений предыдущих элементов массива плюс собственное значение. В GDI+ все гораздо удобнее – содержимое массива отступов представляет собой ширину колонок невидимой таблицы. Убедиться в этом несложно, достаточно воспользоваться следующим примером:

```
Procedure GDI_PAINT(DC:HDC; LayoutRect:TGPRectF);
var GPFont:TGPFont;
    GPGraphics:TGPGraphics;
    BrushFont:TGPSolidBrush;
    SF:TGPStringFormat;
const S='Табуляция:'+#9+'№ 0'+#9+'№ 1'+#9+'№ 2';
      TS:Array[0..2] of single=(100,100,50);
begin
GPGraphics:=TGPGraphics.Create(DC);
```

```
BrushFont:=TGPSolidBrush.Create(aclBlack);
GPFont:=TGPFont.Create(TGPFontFamily.Create('Times'),8);

SF:=TGPStringFormat.Create(StringFormatFlagsNoWrap);
SF.SetTabStops(0,3,@TS);
GPGraphics.DrawString(s,Length(s),gpFont,LayoutRect,SF,BrushFont);
SF.Free;

GPFont.Free; BrushFont.Free;
GPGraphics.free;
end;
```

В усложненном примере, который вы найдете в библиотеке к книге, отступы табуляции определяются тремя метками. Горизонтальные координаты этих меток передаются в ячейки массива табуляции. Смещая метки по оси X, вы сможете изменять позиции текста. Результат работы кода представлен на рис. 39.4. Экспериментируя с примером, обязательно обратите внимание на поведение метода `DrawString()` в случае совпадения позиций табуляции. Подобная ситуация продемонстрирована в нижней части рисунка.

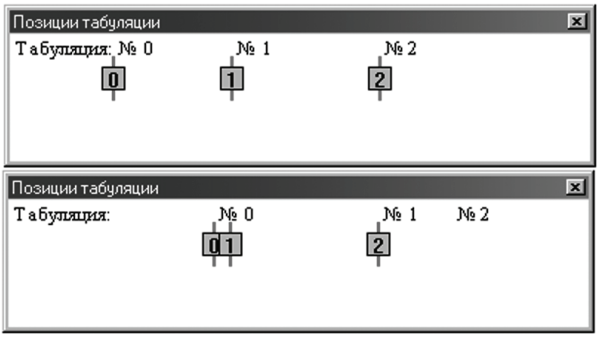


Рис. 39.4. Работа метода SetTabStops( )

Для выяснения текущих настроек экземпляра класса `TGPStringFormat` по выводу табулированной строки стоит познакомиться с методами:

```
function GetTabStopCount: Integer;
function GetTabStops(Count: Integer;
                     FirstTabOffset, pTabStops: PSingle): TStatus;
```

Первый из наших новых знакомых просветит нас относительно числа внедренных в объект позиций табуляции. Это значение должно быть передано в первый параметр метода `GetTabStops()`. Метод, в свою очередь, возвратит значение начального отступа `FirstTabOffset` и указатель на массив значений `pTabStops`.

Определение диапазона символов

В наборе вспомогательных возможностей класса форматирования строк стоит выделить метод:

```
function SetMeasurableCharacterRanges(RangeCount: Integer;
                                      pRanges: PCharacterRange): TStatus;
```

Необходимость в использовании метода возникает только при решении исследовательских задач, связанных с выяснением, в каком месте области вывода окажется тот или иной текстовый символ. Функция определяет диапазон интересующих исследователя символов (указатель на массив `pRanges`) и количество этих диапазонов (аргумент `RangeCount`). Метод работает совместно с входящим в состав класса `TGPGraphics` методом `MeasureCharacterRanges()`. Собственно, он и произведет требуемые расчеты.

Вопрос практического применения `SetMeasurableCharacterRanges()` *мы рассмотрим при изучении метода* `MeasureCharacterRanges()`.

К решению той же самой задачи имеет отношение метод:

```
function GetMeasurableCharacterRangeCount: Integer;
```

Он проинформирует программиста о числе элементов в массиве исследуемых диапазонов.

Вывод символа в точной позиции

Кроме того, в состав холста GDI+ внедрены методы, позволяющие отобразить символ в строго отведенной позиции. За воплощение идеи в жизнь отвечает прецизионный метод:

```
function DrawDriverString(pTxt: pUINT16; Length: Integer; Font: TGPFont;
                          Brush: TGPBrush; pPositions: PGPPointF;
                          Flags: Integer; Matrix: TGPMatrix): TStatus;
```

Сразу бросается в глаза несколько необычный для GDI+ способ определения подлежащего выводу текста – содержимое строки передается в массив, на который ссылается указатель `pTxt`. Каждая ячейка массива состоит из двух байт, позволяя хранить данные в формате Unicode. На первый взгляд несколько эксцентричное решение по передаче текста принято не для того, чтобы внести хотя бы немного разнообразия в будничную жизнь программиста. Особенность метода в том, что при определенной комбинации флагов `Flags` (табл. 39.3) вместо кода символа в массив разрешено передавать индексы интересующих нас глифов. Таким образом, мы получаем возможность вывести не просто символ (который может представлять комбинацию из нескольких глифов), но и любой отдельный глиф, например только точки над символом «ё». Точное место вывода символа или отдельного глифа определяется параметром `pPositions`.

В составе файла, описывающего шрифт TrueType, имеется таблица глифов (таблица «glyf»). В ней хранится определение контура глифа и инструкции по его выводу с жесткой привязкой к сетке. Для того чтобы система поняла, какие глифы принадлежат тому или иному символу, в файл введена таблица соответствия (таблица «cmap»). Для получения данных шрифта требуется обратиться к функции Windows API `GetFontData()`.

Таблица 39.3. Флаги метода DrawDriverString

Константа	Описание
DriverStringOptionsCmapLookup	Установленный флаг определяет, что во входном массиве хранятся коды символов в таблице Unicode. В противном случае метод полагает, что в массиве находятся индексы глифов.
DriverStringOptionsVertical	Указывает на то, что строка должна выводиться по вертикали.
DriverStringOptionsRealizedAdvance	Каждая очередная позиция глифа привязана к местоположению изображения первого символа в строке. При снятии флага координаты вывода глифа определяются программистом.
DriverStringOptionsLimitSubpixel	Флаг экономии памяти. Но при его включении снижается качество изображения выводимых глифов.

Рассмотрим способ просмотра таблицы глифов шрифта гарнитуры Arial. Итак, для того чтобы метод понял, что нам требуется организовать вывод глифа по его индексу, необходимо снять флаг DriverStringOptionsCmapLookup.

```
Procedure GDIPLUS_PAINT(DC:HDC;ClientRect:TRect);
var GPGraphics:TGPGraphics;
    GlyphFont:TGPFont;
    BlackBrush:TGPSolidBrush;
    GlyphNum:Word;            {индекс начального глифа}
    P:TGPPointF;              {точка вывода}
    GlyphFontHeight:single;   {высота символа}
begin
GPGraphics:=TGPGraphics.Create(DC);
BlackBrush:=TGPSolidBrush.Create(aclBlack);
GlyphFont:=TGPFont.Create('Arial',14);
GlyphFontHeight:=GlyphFont.GetHeight(GPGraphics);  //высота шрифта

P:=MakePoint(-32,GlyphFontHeight);
GlyphNum:=0;
REPEAT
 P:=MakePoint(P.X+32,P.Y);//точка вывода
 If P.X+32>ClientRect.Right then P:=MakePoint(0,P.Y+GlyphFontHeight);
 GPGraphics.DrawDriverString(@GlyphNum,1,GlyphFont,BlackBrush,@P,0,nil);
 INC(GlyphNum);
UNTIL P.Y>ClientRect.Bottom;

GlyphFont.Free;  BlackBrush.free; GPGraphics.free;
end;
```

На рис. 39.5 представлен экранный снимок окна более сложного приложения (вы его обнаружите в библиотеке). Оно позволяет строить таблицу глифов для любой указываемой пользователем гарнитуры шрифта. В данном случае вы видите первые 200 глифов шрифта с мудреным названием «Haettenschweiler».

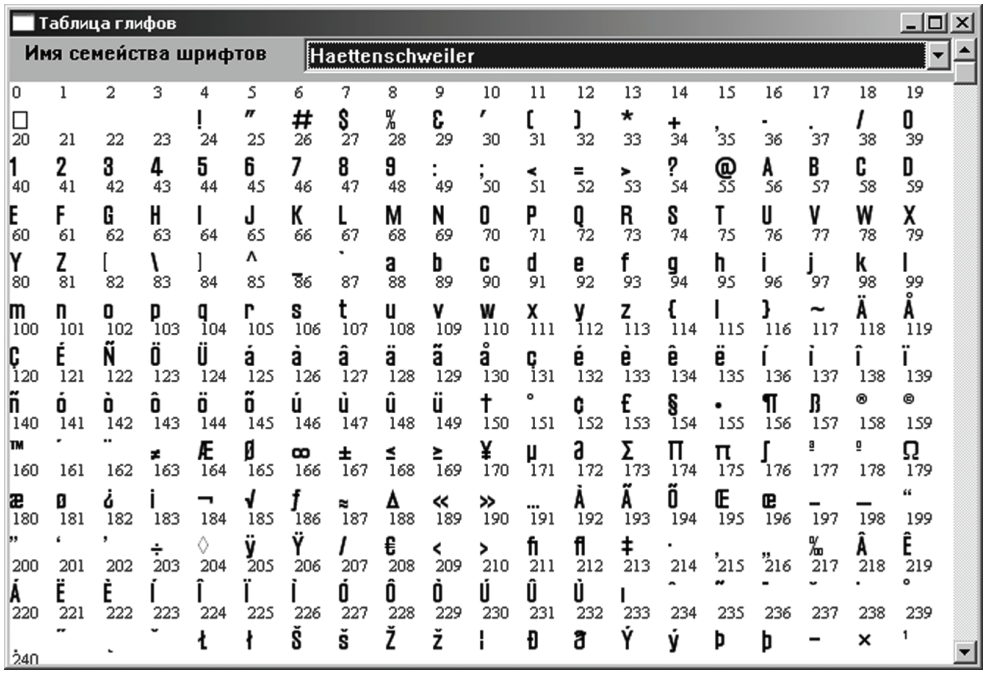


Рис. 39.5. Таблица глифов семейства шрифтов «Haettenschweiler»

Исследование строки

Среди всей когорты функций, работающих с текстом, верхом совершенства по праву считаются методы `MeasureString()`, `MeasureCharacterRanges()` и `MeasureDriverString()`. Они предназначены для проведения скрупулезных исследований по поведению текстовой строки при ее выводе на поверхности холста GDI+. Первый из методов с точностью до логической единицы укажет размер ограничивающего прямоугольника, необходимый для вывода текстовой строки. Второй метод предсказывает, в каком месте холста разместится каждый из подлежащих выводу текстовых символов. Впечатление усиливает еще тот факт, что методы чувствительны к особенностям форматирования текста. Третий по счету метод нацелен на работу с отдельным глифом, он узнает все о его местоположении. Конечно, все сверхъестественные способности методов опираются не на магию, а на точный математический расчет. Но это нисколько не умаляет тот эффект, который они производят на программистов, которые хотя бы раз в своей карьере пытались решить аналогичные задачи «дедовскими» способами, получая метрики шрифтов средствами GDI.

Метод MeasureString()

Факт исключительной полезности `MeasureString()` подтвердит любой разработчик программного обеспечения, который потратил не один час времени

в тщетных попытках найти способ рационального отображения в ограниченной области, например в диалоговом окне, текста с заранее неизвестным размером. Если содержимое строк (количество символов, слов, предложений и т. д.) изменяется динамически, то вместить все это в отведенную область весьма проблематично.

Простейшая реализация так разрекламированного мною метода MeasureString() выглядит следующим образом:

```
function MeasureString(S _: WideString; Length: Integer; Font: TGPFont;
                       const Origin: TGPPointF;
                       out BoundingBox: TGPRectF): TStatus; overload;
```

Здесь S – исследуемая строка, Length – число подлежащих изучению символов (если передать -1, то будет обработан весь текст), Font – применяемый для вывода шрифт, Origin – начальная координата вывода, а BoundingBox представляет собой выходной параметр, в который будут возвращены координаты и геометрические размеры прямоугольной области, необходимой для вывода строки.

Ниже предложен вариант применения рассмотренной версии метода. Наша задача – подобрать размер шрифта так, чтобы выводимый в одну строку текст S полностью поместился в пределах клиентской области ClientRect.

```
Function GetMaxFontSize(DC:HDC; ClientRect:TRect; S:Widestring):single;
var GPGraphics:TGPGraphics;
    GPFont:TGPFont;
    BoundingBox:TGPRectF;
    FontSize:Single;
begin
FontSize:=100.5; //начальная высота шрифта

GPGraphics:=TGPGraphics.Create(DC);
 repeat
    FontSize:=FontSize-0.5;
    GPFont:=TGPFont.Create('Arial',FontSize);//создаем экземпляр шрифта
    GPGraphics.MeasureString(S,-1,GPFont,MakePoint(0,0.0),BoundingBox);
    Result:=FontSize;
    GPFont.Free;
until BoundingBox.Width<=ClientRect.Right;

GPGraphics.free;
end;
```

Разработанная нами функция прямым перебором (с целью упрощения кода) в цикле repeat..until выбирает наиболее подходящий размер шрифта, проверяя результат обращением к методу MeasureString(). Как уже отмечалось ранее, такой способ адаптации размера шрифта под размеры области вывода может оказаться весьма полезным, например, при разработке диалоговых окон (рис. 39.6).

Как всегда, создав одну опорную функцию, разработчики класса TGPGraphics не смогли остановиться и реализовали целую обойму одноименных методов

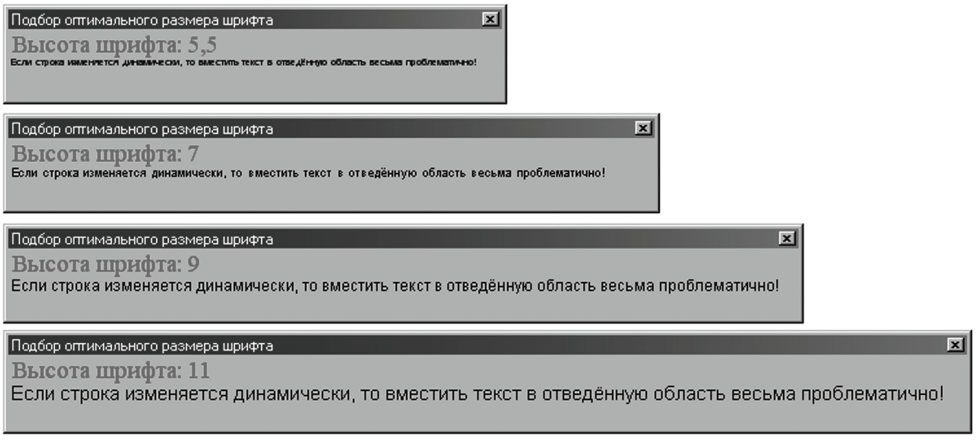


Рис. 39.6. Снимки экранных форм с адаптивным размером шрифта

`MeasureString()`, отличающихся друг от друга незначительными нюансами в определении параметров. От такого «разнообразия» может закружиться голова, но, впрочем, я надеюсь, если вы дочитали книгу до этой главы, то к особенностям именования перегружаемых версий методов в GDI+ уже привыкли...

Итак, вторая вариация метода выглядит следующим образом:

```
function MeasureString(S _: WideString; Length: Integer; Font: TGPFont;
                       const LayoutRect: TGPRectF;
                       out BoundingBox: TGPRectF): TStatus; overload;
```

Как видите, здесь несколько усложнен порядок определения рекомендуемой области вывода, теперь вместо одной координаты начала вывода задается прямоугольная структура `LayoutRect`.

Очередная версия метода `MeasureString()` отличается от предыдущих тем, что в ней учитываются пожелания программиста по форматированию строки:

```
function MeasureString(S _: WideString ; Length: Integer; Font: TGPFont;
                    const Origin: TGPPointF;
                    SFormat: TGPStringFormat;
                    out BoundingBox: TGPRectF): TStatus; overload;
```

Для этого в функцию введен очередной параметр `SFormat` для данных об особенностях форматирования текста.

Наибольшим числом аргументов обладают два заключительных метода:

```
function MeasureString(S _: WideString; Length: Integer; Font: TGPFont;
                   const LayoutRectSize: TGPSizeF;
                   SFormat: TGPStringFormat;
                   out Size: TGPSizeF; pCodePointsFitted: PInteger = nil;
                   pLinesFilled: PInteger = nil): TStatus; overload;
```

и

```
function MeasureString(S _: WideString; Length: Integer; Font: TGPFont;
                  const LayoutRect: TGPRectF;
                  SFormat: TGPStringFormat;
                  out BoundingBox: TGPRectF; pCodepointsFitted: PInteger = nil;
                  pLinesFilled: PInteger = nil): TStatus; overload;
```

Перечень параметров дополнился указателем на переменную, в которую метод возвратит число символов (параметр pCodepointsFitted), свободно вмещающихся в область вывода LayoutRect, и указателем (параметр pLinesFilled) на переменную, принимающую количество строк, входящих в область. Разница между методами чисто косметическая. Один из них возвращает только ширину и высоту требуемой области (параметр Size), а второй – прямоугольные координаты BoundingBox.

Метод MeasureCharacterRanges()

Наконец мы добрались до кульминационного момента. Если бы исследующим строки методам присваивались ученые звания, то метод MeasureCharacterRanges() в два счета заслужил бы профессорский статус. Возможности его таковы, что он с легкостью определит местоположение любого символа текстовой строки на поверхности графического устройства.

```
function MeasureCharacterRanges(S_: WideString;
                        Length: Integer;
                        Font: TGPFont;
                        const LayoutRect: TGPRectF;
                        SFormat: TGPStringFormat;
                        RegionCount: Integer;
                        const Regions: array of TGPRegion): TStatus; overload;
```

Большая часть аргументов метода нам уже хорошо знакома, так что, не останавливаясь на пройденном, сразу направимся к главному. Начнем с того, что метод должен работать в паре с методом SetMeasurableCharacterRanges() экземпляра класса TGPStringFormat. В рамках SetMeasurableCharacterRanges() заполняется массив с индексами символов строки, подлежащих изучению. Указанные данные передаются через параметр форматирования SFormat. Два заключительных аргумента метода возвращают результаты исследования. Так, RegionCount расскажет о числе полученных регионов, а Regions возвратит массив регионов, описывающих точное местонахождение запрошенных символов.

Рассмотрим пример процедуры, демонстрирующей возможности метода на практике. Нами исследуется местонахождение трех групп символов в строке S. Для этого порядковый номер и число символов в группе заносятся в соответствующие ячейки массива ChrRange.

```
Procedure GDIPLUS_PAINT(DC:HDC; S:WideString);
var GPGraphics:TGPGraphics;
    GPFont:TGPFont;
    FntBrush,RBrush:TGPSolidBrush;
```

```
    GPPen:TGPPen;
    Rect_L,Rect_R:TGPRectF;
    SF:TGPStringFormat;
    ChrRange:array[0..2] of TCharacterRange;
    CharRangeRegions : array of TGPRegion;
    i,count:integer;
begin
//РАЗЫСКИВАЕМЫЕ СИМВОЛЫ
ChrRange[0]:=MakeCharacterRange(2, 5);  //1-я группа символов
ChrRange[1]:=MakeCharacterRange(33, 4); //2-я группа символов
ChrRange[2]:=MakeCharacterRange(73, 22);//3-я группа символов

GPGraphics:=TGPGraphics.Create(DC);

FntBrush:=TGPSolidBrush.Create(aclNavy);
RBrush:=TGPSolidBrush.Create($7FFF0000);
GPPen:=TGPPen.Create(aclBlack);

Rect_L:=MakeRect(5, 10, 205, 160.0);  //координаты левой области вывода
Rect_R:=MakeRect(215, 10, 205, 160.0);//координаты правой области

GPFont:=TGPFont.Create('Times New Roman', 14.0);

SF:=TGPStringFormat.Create(0);
SF.SetMeasurableCharacterRanges(3, @ChrRange);//задаем диапазоны исследования
count := SF.GetMeasurableCharacterRangeCount; //уточняем число диапазонов

SetLength(CharRangeRegions,count); //распределяем память под массив регионов
for i := 0 to count-1 do CharRangeRegions[i]:=TGPRegion.Create;

//проводим исследования и выводим их в левой области
GPGraphics.MeasureCharacterRanges(S, -1,GPFont, Rect_L,
                                  SF, Count, CharRangeRegions);
GPGraphics.DrawString(S, -1,GPFont, Rect_L, SF, FntBrush);
GPGraphics.DrawRectangle(GPPen, Rect_L);

for i := 0 to count-1 do GPgraphics.FillRegion(RBrush, CharRangeRegions[i]);

//правая область
SF.SetFormatFlags(StringFormatFlagsMeasureTrailingSpaces);
GPGraphics.MeasureCharacterRanges(S, -1,GPFont, Rect_R,
                                  SF, Count, CharRangeRegions);
GPGraphics.DrawString(S, -1,GPFont, Rect_R, SF, FntBrush);
GPGraphics.DrawRectangle(GPPen, Rect_R);

for i := 0 to count-1 do GPgraphics.FillRegion(RBrush, CharRangeRegions[i]);

for i := 0 to count-1 do CharRangeRegions[i].Free;
GPFont.Free;  FntBrush.free; RBrush.free; GPPen.free;
SF.Free; GPGraphics.free;
end;
```

Читатель, внимательно изучивший код, заметил, что мы разыскиваем три группы символов. В группу 1, обозначенную `ChrRange[0]`**, входят 5 символов (со второго по седьмой). Учитывая то, что отсчет символов в строке берет начало с нулевого элемента, в первую группу вошли «л», «и», символ пробела,**

«б» и «ы». Вторая и третья группы просят возвратить данные о местоположении четырех символов, начиная с 33-й позиции, и 22 символов, начиная с 73-й позиции. Результаты анализа местоположения символов представлены на рис. 39.7, интересующие нас диапазоны закрашены прозрачной кистью.

Постарайтесь найти два отличия между левой и правой областями вывода. Если сдаетесь, то подскажу – разница в выделении последнего символа «к» в слове «строк» и символа «s» в названии метода `MeasureCharacterRanges`. На левом рисунке область выделения завершилась сразу по окончании глифа символа, а в правой части окна (из-за активизации флага `StringFormatFlagsMeasureTrailingSpaces`) в область выделения вошло и пустое пространство за глифом.

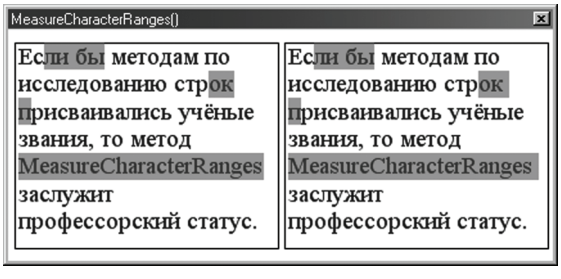


Рис. 39.7. Демонстрация метода MeasureCharacterRanges

Метод MeasureDriverString()

Не так давно мы уже сталкивались с задачей отображения отдельного глифа. Теперь познакомимся со способом «предсказания» места вывода контура целого символа. С ролью прорицателя превосходно справляется метод:

```
function MeasureDriverString(Txt: PUINT16;
                             Length: Integer;
                             Font: TGPFont;
                             pPositions: PGPPointF;
                             Flags: Integer;
                             Matrix: TGPMatrix;
                             out BoundingBox: TGPRectF): TStatus;
```

Перечень аргументов метода аналогичен параметрам `DrawDriverString()`, но за единственным исключением. Здесь добавлен выходной параметр `BoundingBox`, в который функция поместит координаты прямоугольной области, в которой и окажется глиф (рис. 39.8).

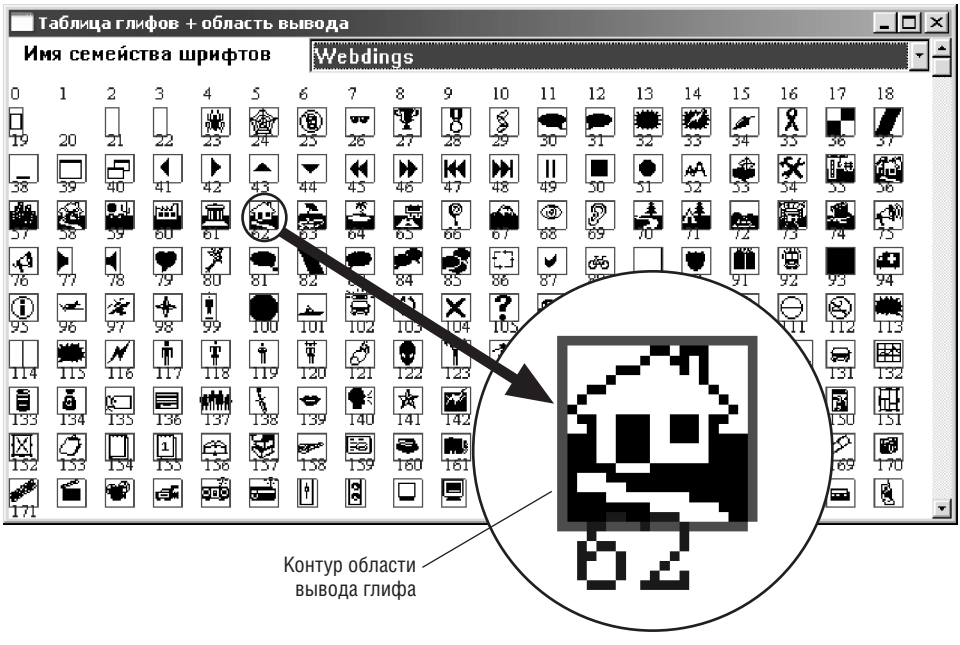


Рис. 39.8. Предсказание области вывода глифов шрифта гарнитуры «Webdings»

РЕЗЮМЕ

Вывод текстовых данных в GDI+ опирается на применение метода холста `DrawString()`. Указанный метод превосходно справляется как с отображением простейшей текстовой строки, так и с выводом текста со сложным форматированием. Правила форматирования задаются с помощью класса `TGPStringFormat`. Помимо вывода текста холст GDI+ обладает рядом методов, позволяющих спрогнозировать местоположение любого символа или даже его части – отдельного глифа.

40

Качество вывода и коррекция цвета

Одно из ключевых преимуществ расширенной версии графической библиотеки над ее предком GDI – наличие простых в применении методов, позволяющих управлять качеством вывода изображения. Вполне естественно, что никто не может нам запретить самостоятельно (используя лишь Windows API) написать свои версии функций, оказывающих положительное влияние на процесс графического вывода, и обращаться к ним в проектах, но подобный подход, безусловно, потребует много дополнительных часов работы даже для весьма квалифицированного программиста. Так что, если мы с вами принимаем решение – включить в состав проекта модуль GDI+, то вместо изобретения велосипедов можем воспользоваться готовым решением.

В этой главе мы рассмотрим ряд принадлежащих классу `TGPGraphics` методов, нацеленных исключительно на совершенствование вывода графических примитивов, изображений и текстовых строк. Кроме того, мы посвятим много времени вопросам управления цветовыми атрибутами, изучив класс `TGPImageAttributes`.

Сглаживание

Полагаю, что вы не раз обращали внимание на одну неприятную особенность растровых систем, работающих при низком разрешении (до 300 dpi). При выводе на таком графическом устройстве наклонных линий их поверхность как бы покрывается ступеньками. В англоязычной литературе по теории дискретизации такой эффект называется *aliasing*. Конечно, в перспективе, с постепенным повышением качества дисплеев за счет уменьшения размера пиксела, данный недостаток станет просто незаметен человеческому глазу, но пока программистам приходится заниматься мошенничеством – обманывать зрение доверчивого пользователя. Процесс устранения ступенчатого эффекта называется *antialiasing*. В самом тривиальном случае суть «надувательства» сводится к следующему: наиболее далеко отстоящие от центральной оси линии ячейки растра закрашиваются не цветом пера, которым

проводится линия, а усредненным значением между цветом фона и цветом пера (рис. 40.1). В результате пикселы линии «размазываются», за счет чего поверхность линий и кривых выравнивается.

В желании порадовать пользователя качественным изображением разработчики GDI+ пошли еще дальше и предложили несколько режимов сглаживания ступенек. Наиболее совершенный из них основан на идее локальной фильтрации. В ней цвет обрабатываемого пиксела строится на основе взвешенного суммирования цветов целой группы окружающих пикселов. Естественно, качество результирующей картинки стоит в прямой зависимости от числа анализируемых пикселов, так, в GDI+ окно фильтрации может достигать размера 8×4 пикселов.

Существенное достоинство алгоритма локальной фильтрации в том, что он может быть применен при обработке уже нарисованного изображения, например фотографии. В современных графических редакторах такой прием называется сглаживанием (smoothing).

Нам, к счастью, нет необходимости проводить ночи над разработкой своих алгоритмов обработки пикселов. Один из наиболее простых и одновременно весьма эффективных приемов борьбы со ступеньками, зубцами и прочими заусенцами на проведенных под углом линиях предоставляет холст `TGPGraphics`:

```
function SetSmoothingMode(SmoothingMode: TSmoothingMode): TStatus;
```

Степень сглаживания определяется параметром `SmoothingMode`. Это диапазон значений от -1 до 4.

```
const
  SmoothingModeInvalid     = QualityModeInvalid; //=-1 не используется
  SmoothingModeDefault     = QualityModeDefault; //= 0 по умолчанию
  SmoothingModeHighSpeed   = QualityModeLow;     //= 1 низкое качество
  SmoothingModeHighQuality = QualityModeHigh;    //= 2 высокое качество
  SmoothingModeNone        = 3;
  SmoothingModeAntiAlias   = 4;
```

Наилучшее качество сглаживания обеспечивают режимы `SmoothingModeHighQuality` и `SmoothingModeAntiAlias`. Получив такую команду, GDI+ активизирует специальный фильтр, основанный на алгоритме локальной фильтра-

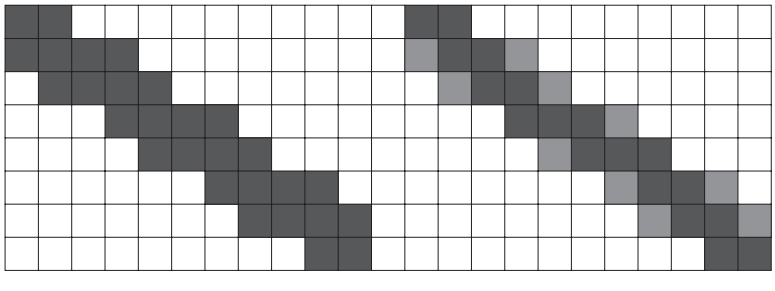


Рис. 40.1. Простейший способ устранения ступенчатого эффекта

ции с анализируемым окном размерностью 8×4 пикселов. Согласно SDK по GDI+ компромиссный вариант между качеством фильтрации и ресурсными затратами достигается при установке режима SmoothingModeHighSpeed, однако на деле внешний вид полученной в данном режиме линии мало чем отличается от линии, проведенной при выключенном сглаживании (рис. 40.2). Зато плавность контура линии, нарисованной в режиме сглаживания с высоким качеством SmoothingModeHighQuality, вполне заметна.

Для уточнения текущих настроек холста по сглаживанию необходимо обратиться к методу:

```
function GetSmoothingMode: TSmoothingMode;
```

Прямое отношение к процессу сглаживания имеет метод:

```
function SetPixelOffsetMode(PixelOffsetMode: TPixelOffsetMode): TStatus;
```

Он позволяет незначительно изменить внешний вид границы визуального объекта за счет небольшого сдвига закрашенных пикселов. Из всего перечня значений, которые может воспринимать метод, стоит использовать PixelOffsetModeHighQuality, подбирающий значение наилучшего сдвига, и PixelOffsetModeNone, отключающий режим подбора отступа. Осталось добавить, что текущее значение сдвига легко выяснить, воспользовавшись услугами метода:

```
function GetPixelOffsetMode: TPixelOffsetMode;
```

Порядок наложения

При работе с GDI+ изредка возникает впечатление, что библиотека несколько сыровата. С одной стороны, в классы GDI+ упаковано даже то, что в общем-то можно было и не трогать (взять хотя бы корректировку эффекта красных глаз на фотографиях или масштабирование цвета). С другой стороны, разработчики GDI+ не стали утомлять себя инкапсуляцией в холст методов, поддерживающих простейшие бинарные растровые операции (не говоря уже о тернарных и кватернарных)! Видимо, в Microsoft полагают, что если обычный программист знает о существовании ROP-операций, то он в состоянии самостоятельно разобраться с вопросом их использования в GDI+

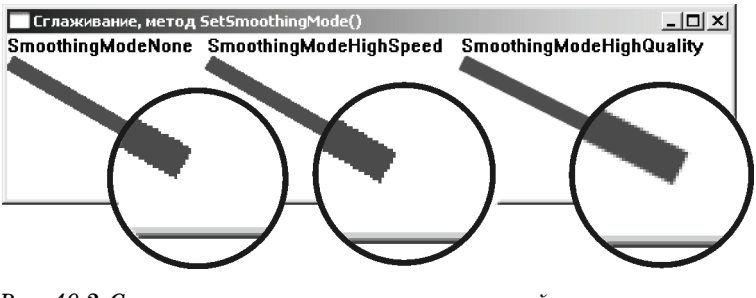


Рис. 40.2. Сглаживание ступенек на наклонной линии с помощью метода SetSmoothingMode

без помощи класса TGPGraphics. Такое доверие вдвойне приятно после того, как, изрядно порывшись в недрах библиотеки расширенной графики, все-таки добираешься до скромной группы методов, оказывающих некоторое влияние на логику сложения пикселов холста и рисунка. В данном случае речь ведется о функциях холста:

```
function SetCompositingMode(CompositingMode: TCompositingMode): TStatus;
function GetCompositingMode: TCompositingMode;
```

Первая из них назначает формат наложения, а вторая информирует программиста об установленном режиме. Режимов всего два, они определяются перечислением:

```
type TCompositingMode =
               (CompositingModeSourceOver, CompositingModeSourceCopy);
```

Начальный элемент перечисления указывает, что при наложении рисунка на холст осуществляется смешивание цветов пикселов фона и рисунка с учетом значения альфа-канала. Второй же режим требует, чтобы рисунок просто стер фоновые пикселы. Надо полагать, что в основу идеи работы SetCompositingMode() положена домашняя заготовка Microsoft, появившаяся на свет еще во времена Windows 98, – одна из наиболее мощных функций GDI AlphaBlend(). Но это не просто повторение старой разработки, GDI+ развивает возможности функции, так, в режиме CompositingModeSourceOver обновленная версия функции позволяет влиять на качество наложения. Для этих целей в состав методов холста введены функции:

```
function SetCompositingQuality(
            CompositingQuality: TCompositingQuality): TStatus;
function GetCompositingQuality: TCompositingQuality;
```

Несмотря на то, что имеется существенный перечень констант, определяющих требуемую степень качества, наиболее заметный эффект будет получен только при работе в режимах CompositingQualityHighSpeed и CompositingQualityHighQuality (см. рис. 40.3 на цв. вклейке, с. 424).

Интерполяция растра

Все операции, связанные с масштабированием или поворотом растрового рисунка, так или иначе пагубно сказываются на результирующем изображении. В арсенале возможностей холста GDI+ имеется возможность снизить остроту проблемы, возникающей при изменении размеров изображения. Нам позволено повысить качество результирующего рисунка за счет управления алгоритмом интерполяции, добавляющим в изображение новые пикселы. Из общеизвестных алгоритмов этой категории следует упомянуть три:

1. *Алгоритм копирования ближайшего пиксела* (*nearest neighbor interpolation*) сводится к тому, что в изображение добавляются новые пикселы того же цвета, что и расположенные рядом. Точки дублируются, их создается тем больше, чем больше должно быть изображение. Метод максимально прост в реализации, но на этом его достоинства заканчиваются.

2. *Алгоритм билинейной интерполяции* (*bilinear interpolation*) выполняется на основе матрицы из четырех соседних пикселов. Метод достаточно уверенно устраняет ступеньки, но снижает резкость увеличенного изображения.
3. *Алгоритм бикубической интерполяции* (*bicubic interpolation*) схож с билинейным алгоритмом, но теперь матрица включает 16 соседних пикселов. Как следствие, из всех перечисленных алгоритмов он дает наилучшие результаты и является наиболее ресурсоемким.

Все перечисленные алгоритмы нашли свое место в библиотеке GDI+. Предпочтение тому или иному способу интерполяции можно отдать, обратившись к методу:

```
function SetInterpolationMode(InterpolationMode: TInterpolationMode): TStatus;
```

Возможные режимы представлены в табл. 40.1. Для уточнения текущих настроек холста следует обратиться к методу:

```
function GetInterpolationMode: TInterpolationMode;
```

Таблица 40.1. Константы интерполяции изображения

Константа	Описание
InterpolationModeInvalid	Не используется.
InterpolationModeDefault	Режим работы по умолчанию.
InterpolationModeLowQuality	Низкокачественная интерполяция.
InterpolationModeHighQuality	Высококачественная интерполяция.
InterpolationModeBilinear	Билинейная интерполяция. Результат среднего качества, так как не используется предварительная цифровая фильтрация обслуживаемого изображения. Метод не подходит для рисунков, которые будут уменьшены меньше чем на 50% от их исходного размера.
InterpolationModeBicubic	Бикубическая интерполяция. Результат среднего качества. Метод не подходит для рисунков, которые будут уменьшены меньше чем на 25% от их исходного размера.
InterpolationModeNearest-Neighbor	Интерполяция, использующая для определения цвета нового пиксела цвета его ближайших соседей.
InterpolationModeHighQuality-Bilinear	Высококачественная билинейная интерполяция.
InterpolationModeHighQuality-Bicubic	Высококачественная бикубическая интерполяция.

Вклад метода `SetInterpolationMode()` в формирование итогового изображения особенно заметен в режиме высококачественной бикубической интерполяции. На рис. 40.4 представлен результат шестикратного увеличения изображения в наихудшем (`InterpolationModeLowQuality`) и наилучшем (`InterpolationModeHighQualityBicubic`) режимах.

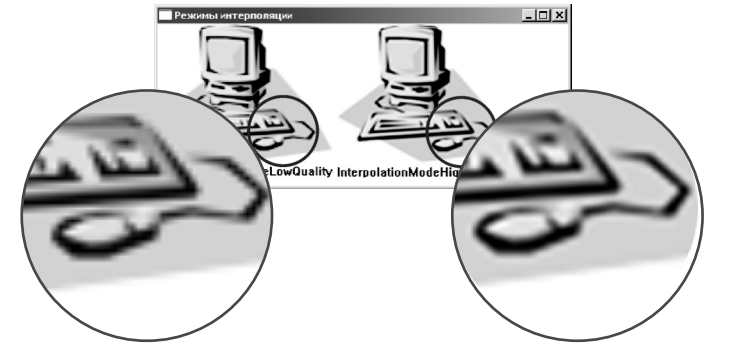


Рис. 40.4. Влияние режима интерполяции на увеличение растрового изображения

Повышение качества вывода текста

Несмотря на то, что современные контурные шрифты TrueType и OpenType отличаются покладистым характером и превосходно масштабируются, все равно существует проблема с внешностью текстовых строк. Причин появления зубцов на контуре глифа великое множество. Это зависит от физического размера пиксела экрана, кривизны экрана, несовершенства драйверов, сложности непропорционального масштабирования (в особенности при поворотах символов) и от многих других причин. Именно поэтому в GDI+ уделено существенное внимание вопросу улучшения качества изображения текста за счет сглаживания глифов шрифта.

В Windows 2003, XP и Vista внедрена усовершенствованная технология вывода текста Microsoft ClearType, труды которой особенно заметны на современных жидкокристаллических дисплеях.

Степень сглаживания определяется методом:

```
function SetTextRenderingHint(NewMode: TTextRenderingHint): TStatus;
```

Метод позволит выбрать один из шести вариантов режима прорисовки, определенных в перечислении `TTextRenderingHint`.

```
type TTextRenderingHint= (
  TextRenderingHintSystemDefault,           //=0
  TextRenderingHintSingleBitPerPixelGridFit,//=1
  TextRenderingHintSingleBitPerPixel,       //=2
  TextRenderingHintAntiAliasGridFit,        //=3
  TextRenderingHintAntiAlias,               //=4
  TextRenderingHintClearTypeGridFit         //=5
  );
```

В перечислении каждый очередной элемент выдвигает все более возрастающие требования к качеству прорисовки глифов. Исключение составляет пер-

вый элемент перечисления TextRenderingHintSystemDefault, он указывает на то, что при выводе текста должны использоваться текущие установки системы. Наилучшему режиму соответствует константа TextRenderingHintClearTypeGridFit (в результате активизируется Microsoft ClearType), наихудший режим включается константой TextRenderingHintSingleBitPerPixelGridFit. В последнем случае внешний вид символа оставляет желать лучшего, но зато нагрузка на процессор ничтожна. Вполне естественно, что за наилучшее качество сглаживания приходится расплачиваться дополнительными ресурсными затратами компьютера.

Для выяснения текущих установок предназначен метод:

```
function GetTextRenderingHint: TTextRenderingHint;
```

Ниже вашему вниманию предложен пример, демонстрирующий порядок работы с методом.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPFont:TGPFont;
    Brush:TGPSolidBrush;
const S1='Сглаживание выключено';
      S2='Сглаживание включено';
begin
GPGraphics:=TGPGraphics.Create(DC);
Brush:=TGPSolidBrush.Create(aclBlue);
GPFont:=TGPFont.Create('Arial',32);
GPGraphics.SetTextRenderingHint(TextRenderingHintSingleBitPerPixel);
GPGraphics.DrawString(S1, -1, GPFont, MakePoint(10.0, 10.0), Brush);
GPGraphics.SetTextRenderingHint(TextRenderingHintAntiAlias);
GPGraphics.DrawString(S2, -1, GPFont, MakePoint(10.0, 80.0), Brush);
GPFont.Free; Brush.Free; GPGraphics.Free;
end;
```

Результат выполнения предложенного кода представлен на рис. 40.5. Верхняя строка соответствует наиболее непритязательному режиму работы, когда технология улучшения привлекательности шрифта отключена. В нижней строке глифы символов уже «приглажены» средствами GDI+.

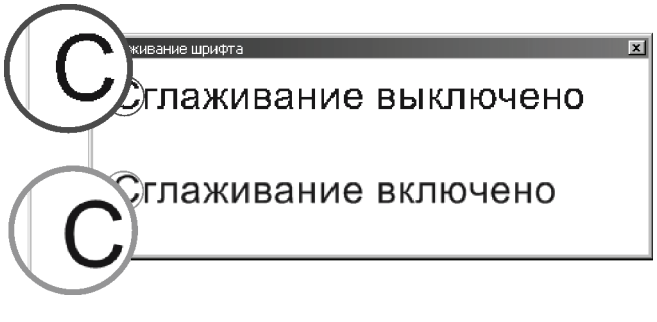


Рис. 40.5. Улучшение качества вывода текста за счет сглаживания

Если верить документации SDK к GDI+, то дополнительную настройку степени сглаживания контуров глифов можно осуществить посредством установки контрастности текста:

```
function SetTextContrast(Contrast: UINT): TStatus; // 0..12
```

Функция может принимать значения в пределе диапазона от 0 до 12. Имеется и парная функция, информирующая о текущих настройках холста:

```
function GetTextContrast: UINT;
```

К сожалению, практика несколько противоречит теории, все мои попытки изменить значения контрастности с помощью метода `SetTextContrast()` не привели к сколько-нибудь заметному изменению контрастности или сглаживанию контуров символов...

Сохранение состояния холста GDI+

Зачастую при построении относительно сложного изображения программисту приходится неоднократно изменять текущие настройки холста GDI+. Очень часто воздействию подвергаются такие существенные характеристики, как регион отсечения, аффинные преобразования и качество вывода (речь идет о методах, которым посвящена первая часть этой главы). Три перечисленных выше атрибута контекста устройства в терминах GDI+ называются *графическим состоянием* холста (*graphics state*). Если изменения в состоянии холста носят временный характер, то было бы неплохо иметь механизм, позволяющий осуществить откат к некоторому, заранее отмеченному состоянию.

Простейший способ сохранения текущих настроек заключается в обращении к методу:

```
function Save: GraphicsState;
```

Функция возвратит целочисленный идентификатор запомненного состояния (тип `GraphicsState` тождественен типу `Unit`). Для восстановления сохраненных атрибутов контекста обращаемся к методу:

```
function Restore(gstate: GraphicsState): TStatus;
```

Указанная функция рассчитывает получить идентификатор подлежащего восстановлению состояния.

Холст допускает построение вложенных конструкций, основанных на обращении к методам сохранения и восстановления, но при этом необходимо помнить, что информация о графическом состоянии помещается в стек. И если команда `Save` заносит в стек некоторый блок данных, то `Restore` извлекает этот блок из стека. Так как стек построен по принципу *LIFO* (*Last Input First Output*, пришел последним – первым вышел), то попытка восстановить хранимое на дне стека состояние контекста приведет к стиранию всей информации, расположенной выше. Другими словами, будут удалены все сохраненные данные, которые были внесены в стек с помощью более поздних вызовов метода `Save()`.

Продемонстрируем процесс сохранения-восстановления графического состояния холста на практике. Допустим, нам необходимо получить эффект зеркального отражения при выводе текстовых данных (рис. 40.6).

Подобную задачу легко решить, задействовав механизм аффинных преобразований. Для этого достаточно передать в поле eM22 матрицы аффинных преобразований значение -1 (см. табл. 12.1). Ниже предложена процедура, позволяющая перевернуть текстовую строку вокруг оси Y и вывести ее в точке с координатами Origin.

```
Procedure DrawString_MirrorY(GPGraphics:TGPGraphics;
                    S:String;GPFont:TGPFont;Origin:TGPPointF;Brush:TGPBrush);
var GRState:Cardinal;
    GPMatrix:TGPMatrix;
    eDy:single;
begin
GRState:=GPGraphics.Save;    //запоминаем состояние холста

{при зеркальном отражении надо рассчитать смещение по оси Y,
иначе перевернутый текст окажется за пределами области отсечения}
eDy:=Origin.Y*2+GPFont.GetHeight(GPGraphics);

  GPMatrix:=TGPMatrix.Create();
  GPMatrix.SetElements(1,0,0,-1,0, eDy);
  GPGraphics.SetTransform(GPMatrix);

GPGraphics.DrawString(S,-1,GPFont,Origin,Brush);
GPGraphics.Restore(GRState); //восстанавливаем исходное состояние холста
end;
```

Суть проводимых преобразований раскрыта на рис. 40.7. При заполнении матрицы GPMarix необходимо учитывать, что операция зеркального отражения текста переместит строку в область с отрицательными значениями координаты Y, другими словами, оставит текст за пределами видимой области. Для того чтобы строка вновь оказалась в запланированном месте (определенном в параметре Origin), в коде процедуры нам пришлось компенсировать перенос значением eDy.

Для того чтобы применяемые при выводе строки аффинные преобразования не оказали никакого пагубного воздействия на весь остальной код программы, мы воспользовались методами Save() и Restore().

Рис. 40.6. Зеркальное отражение текстовой строки

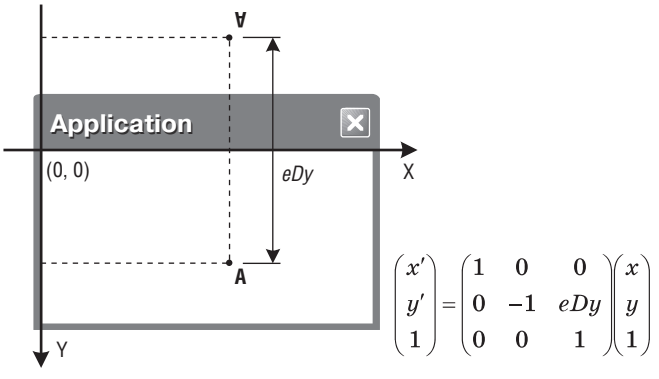


Рис. 40.7. Состояние матрицы преобразований при осуществлении зеркального отражения

Напомню, что у обычного контекста графического устройства, полученного средствами GDI, также предусмотрена возможность сохранения текущего состояния (см. функции `SaveDC` *и* `RestoreDC` *во второй главе книги). Причем в этом случае будет сохранен более широкий спектр атрибутов контекста: перо, кисть, шрифт, палитра, регион отсечения, режим вывода…*

Графический контейнер

Для сохранения и восстановления состояния холста `TGPGraphics` можно воспользоваться еще одной группой методов – методами построения графического контейнера. С точки зрения программирования под графическим контейнером понимается программный блок, вовнутрь которого программист вносит операции, в дальнейшем расцениваемые холстом GDI+ как временные.

В простейшем случае о факте начала области контейнера извещает метод:

```
function BeginContainer: GraphicsContainer;
```

В результате выполнения функция возвращает значение, идентифицирующее новый контейнер (тип данных `GraphicsContainer` является не чем иным, как `UNIT`). Полученный идентификатор понадобится в тот момент, когда нам потребуется вернуть холст GDI+ в исходное состояние. Тогда мы обращаемся к методу:

```
function EndContainer(ID: GraphicsContainer): TStatus;
```

При работе с холстом GDI+ допускается создавать несколько контейнеров, более того, ничто не запрещает нам формировать сложные вложенные конструкции, когда контейнеры погружены друг в друга наподобие русской матрешки. При этом должно выполняться единственное ограничение – операции сохранения и восстановления данных контейнеров должны вестись последовательно, так как информация хранится в стеке LIFO.

```
Conteiner1:=GPGraphics.BeginContainer();
//операции в рамках первого контейнера
```

```
        Conteiner2:=GPGraphics.BeginContainer();
        //операции в рамках второго контейнера
        GPGraphics.EndContainer(Conteiner2);
//операции в рамках первого контейнера
GPGraphics.EndContainer(Conteiner1);
```

Механизм графического контейнера позволяет сохранить только настройки контекста, связанные с регионом отсечения, двумерными аффинными преобразованиями и опциями, определяющими качество графического вывода.

Коррекция цвета, класс TGPImageAttributes

Одним из интереснейших нововведений GDI+, нацеленных на улучшение качества изображений, безусловно, можно считать выдающиеся возможности библиотеки по коррекции цвета графических объектов. Благодаря этому сервис, которым ранее могли похвастаться только дорогостоящие графические редакторы, стал доступен и в проектах Delphi.

В качестве основного подарка, который компания Microsoft предоставила разработчику программного обеспечения, выступает уникальный, не имеющий аналога ни в наборе функций GDI, ни в библиотеке VCL, класс `TGPImageAttributes`. К полученному из указанного класса объекту корректнее относится как к контейнеру, способному хранить описание неких цветовых атрибутов и порядка их преобразования, в результате применения которого (например, к растровому изображению) мы получим тот или иной визуальный эффект. Вопросами применения цветового фильтра ведает не контейнер, а холст `TGPGraphics`. Например, одна из наиболее сложных реализаций метода `DrawImage` (см. главу 35) рассчитывает получить от нас параметр `ImageAttributes:TGPImageAttributes` с описанием цветовых атрибутов.

Нет ничего проще, чем создать экземпляр класса `TGPImageAttributes`, способ до такой степени неприхотлив, что даже вызывает некоторое разочарование.

```
constructor Create; reintroduce; overload;
```

Как видите, конструктор не требует от программиста приложения умственных усилий. Но спешу вас заверить, что первоначальное впечатление о `TGPImageAttributes` поверхностное, оно проходит очень скоро, сразу после того, как мы взглянем на его методы.

После создания объекта цветовых атрибутов следует определиться, к каким графическим объектам мы намерены применять цветовую фильтрацию. Для этого предназначен метод:

```
function SetToIdentity(
        Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

По умолчанию функция указывает контейнеру, что коррекция цвета затронет и растровые, и векторные изображения. Но это не окончательное решение, и нам позволено внести в него некоторые уточнения. Для этого предназначен параметр `Type`, типизированный перечислением `TColorAdjustType`.

```
type TColorAdjustType = (
    ColorAdjustTypeDefault,//универсальные настройки по умолчанию
    ColorAdjustTypeBitmap, //коррекция растровых образов
    ColorAdjustTypeBrush,  //коррекция только кисти метафайла
    ColorAdjustTypePen,    //коррекция только пера метафайла
    ColorAdjustTypeText,   //коррекция только текста метафайла
    ColorAdjustTypeCount,  //не используется
    ColorAdjustTypeAny     //не используется);
```

Если мы обслуживаем метафайлы, то допускается внести дополнительную детализацию и конкретизировать, что в качестве подопытных объектов выступят только кисти, перья или текст векторного рисунка.

Коррекция цвета объектов метафайла (идентификаторы `ColorAdjustTypeBrush`*,* `ColorAdjustTypePen`*,* `ColorAdjustTypeText`*) обычно применяется во время редактирования векторного рисунка для воздействия на его отдельные записи либо при его построчном воспроизведении (метод* `EnumerateMetafile`*). Если же речь идет об обычном отображении метафайла на поверхности графического устройства, то стоит удовлетвориться настройками по умолчанию.*

Завершая тему работы с метафайлами, упомянем еще две функции. Чтобы отключить цветовую коррекцию для определенной категории объектов (кисть, перо, текст), используем метод:

```
function SetNoOp(Type _: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Чтобы восстановить режим фильтрации цвета для ранее отключенного объекта, вызываем:

```
function ClearNoOp(Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

На этом подготовительные операции закончились, и наступил черед попробовать класс в деле. Начнем с наиболее простого метода, позволяющего заместить один или несколько цветов исходного графического объекта другими.

```
function SetRemapTable(MapSize: Cardinal;
                       Map: PColorMap;
                 Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Данные о цветах передаются во второй параметр функции. Это указатель на массив, каждый элемент которого описывается записью:

```
type TColorMap = packed record
    OldColor: TGPColor; //значение старого цвета
    NewColor: TGPColor; //значение нового цвета
end;
```

Первый элемент структуры хранит старое, а второй – новое значение цвета. В простейшем случае, когда нам требуется изменить всего один цвет, например черный белым, за основу берем следующий фрагмент кода:

```
var GPGraphics:TGPGraphics;
    ColorMap:TColorMap;
    IA:TGPImageAttributes;
```

```
begin
GPGraphics:=TGPGraphics.Create(DC);
  IA:=TGPImageAttributes.Create; //создаем объект атрибутов
  ColorMap.OldColor:=aclWhite;   //цвет замены – черный
  ColorMap.NewColor:=aclBlack;   //новый цвет – белый
  IA.SetRemapTable(2,@ColorMap); //передаем данные о цветах в атрибуты
//другие операции по выводу графического объекта
```

Для сброса настроек замены цветов вызываем функцию:

```
function ClearRemapTable(
            Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Контейнер цветовых атрибутов не забывает и о том, что многие графические образы зачастую работают с цветовой палитрой. В этой палитре тоже допускается заменить один или несколько цветов. Для этого предназначена функция:

```
function GetAdjustedPalette(ColorPalette: PColorPalette;
                            ColorAdjustType: TColorAdjustType): TStatus;
```

Здесь первый параметр ColorPalette – указатель на описание палитры цветов. Перед обращением к методу необходимо инициализировать контейнер цветовых атрибутов, например, как в следующем фрагменте кода с замещением цвета.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    CP:PColorPalette;
    IA:TGPImageAttributes;
    Br:TGPSolidBrush;
    Rct:TGPRect;
    i:byte;
    ColorMap:TColorMap;
begin
//создаем палитру из 4 цветов
  GetMem(CP,SizeOf(TColorPalette)+SizeOf(ARGB)*4);
  CP^.Flags:=0; CP^.Count:=4;
  i:=0; CP^.Entries[i]:=aclRed;
  i:=1; CP^.Entries[i]:=aclBlue;
  i:=2; CP^.Entries[i]:=aclGreen;
  i:=3; CP^.Entries[i]:=aclBlack;

  GPGraphics:=TGPGraphics.Create(DC);
  Rct:=MakeRect(0,0,30,30);
{выводим старую палитру на экран}
  Br:=TGPSolidBrush.Create(aclBlack);
  for i:= 0 to CP^.Count-1 do
    begin
      Br.SetColor(CP^.Entries[i]);
      GPGraphics.FillRectangle(Br,Rct);
      Rct.X:=Rct.X+Rct.Width+5;
    end;

IA:=TGPImageAttributes.Create; {создаем контейнер цветовых атрибутов}
```

```
  ColorMap.oldColor:=aclRed; ColorMap.newColor:=aclYellow;
  IA.SetRemapTable(1,@ColorMap); {заменяем красный цвет на желтый}
  IA.GetAdjustedPalette(CP,ColorAdjustTypeBitmap); {обновляем палитру}

  Rct:=MakeRect(0,35,30,30);
  for i:= 0 to CP^.Count-1 do {выводим палитру с замещенным цветом}
    begin
      Br.SetColor(CP^.Entries[i]);
      GPGraphics.FillRectangle(Br,Rct);
      Rct.X:=Rct.X+Rct.Width+5;
    end;

  FreeMem(CP);
  Br.Free; GPGraphics.Free; IA.Free;
end;
```

Задача подмены одного цвета другим – просто семечки в сравнении с другими возможностями контейнера цветовых атрибутов. Для начала познакомимся с эффектом гаммы – фильтром, позволяющим проявлять детали в слабоконтрастном изображении. С настройкой коэффициента гаммы с большим удовольствием сразится метод:

```
function SetGamma(Gamma: Single;
            Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Значение устанавливаемой гаммы определяется первым параметром `Gamma`. Как вы уже заметили, у всех методов класса `TGPImageAttributes`, начинающихся на приставку «`Set`», имеется антипод «`Clear`». В данном случае я говорю о функции:

```
function ClearGamma(
            Type _: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

снимающей гамму.

Управление результирующим цветом на основе устранения малозначащих цветовых составляющих и максимизации уровня основных цветов осуществляет пороговый фильтр, он вызывается функцией:

```
function SetThreshold(Threshold: Single;
                  Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Метод производит усиление и/или удаление цветов в соответствии с определенным в параметре `Threshold` коэффициентом отсечения. Передаваемое в аргумент число должно находиться в диапазоне значений от 0 до 1. Принцип отсечения таков. Допустим, исходный цвет определяется следующими цветовыми составляющими: R=160, G=200 и B=100. Значение коэффициента отсечки равно 0,6. Функция осуществляет следующие расчеты:

1. Вычисляется порог отсечения. Для этого максимальное значение интенсивности 255 умножается на коэффициент отсечения 0,6. Результат 255×0,6=153.
2. Полученное пороговое число сравнивается со значениями каждой из трех компонент RGB (байт прозрачности A не учитывается). Цветовые состав-

ляющие, не сумевшие превысить порог, беспощадно отбрасываются, а более успешные компоненты, наоборот, усиливаются до предельного значения. В нашем случае R=160>153, G= 200>153, B=100<153. В результате синий канал отключается, а красный и зеленый усиливаются до 255.

В библиотеке на сайте издательства вы найдете пример управления отсечением цветовых каналов.

```
Procedure GDIPLUS_PAINT(DC:HDC);
var GPGraphics:TGPGraphics;
    GPImage:TGPImage;
    IA:TGPImageAttributes;
    W,H:integer;
    DestRect:TGPRectF;
    Threshold:Single;
begin
  GPGraphics:=TGPGraphics.Create(DC);
  GPImage:=TGPImage.Create('..\Graphics\NWAG.png');
  W:=GPImage.GetWidth; H:=GPImage.GetHeight; //размеры рисунка

  GPGraphics.DrawImage(GPImage,0,0,W,H);  //вывод обычного рисунка
  DestRect:=MakeRect(0,H+20.0,W,H);

  Threshold:=1;
  IA:=TGPImageAttributes.Create;

  While Threshold>0 Do
  begin
  Threshold:=Threshold-0.1;
  IA.SetThreshold(Threshold,ColorAdjustTypeBitmap);
  GPGraphics.DrawImage(GPImage,DestRect,0,0,
                       GPImage.GetWidth,GPImage.GetHeight,UnitPixel,IA,Nil);
  DestRect.X:=DestRect.X+GPImage.GetWidth; //смещаем область вывода
  end;

  GPGraphics.Free; GPImage.Free; IA.Free;
end;
```

Загрузив исходное изображение с ничего не подозревающим попугаем, мы немного поэкспериментировали с порогом отсечения `Threshold`, вследствие чего птичка неоднократно «линяла». Результаты бескровных опытов над нарисованным пернатым представлены на рис. 40.8 (см. цветную вклейку, с. 424).

Для сброса настроек порогового фильтра обращаемся к методу:

```
function ClearThreshold(
          Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Еще один эффект, позволяющий управлять прозрачностью результирующего изображения, обеспечит метод:

```
function SetColorKey(ColorLow, ColorHigh: TGPColor;
         Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Функция не просто выбирает один из цветов для превращения его в прозрачный. Функция `SetColorKey()` устанавливает диапазон прозрачности для об-

служиваемого графического объекта. Диапазон цветов определяется параметрами ColorLow, ColorHigh. Сброс цветовых ключей осуществляет функция-антипод:

```
function ClearColorKey(
            Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Зачастую на помощь контейнеру цветовых атрибутов приходит еще один уникальный класс из состава GDI + – матрица цветов TColorMatrix.

Матрица цветов TColorMatrix

Ряд методов класса TGPImageAttributes в своей работе пользуется услугами особой структуры TColorMatrix, которую мы станем именовать матрицей цветов. Матрица цветов TColorMatrix находит применение при построении целой плеяды эффектов, начиная с простейших преобразований отдельного цветового оттенка и заканчивая сложными операциями по масштабированию, трансляции и даже повороту цветового вектора. С физической точки зрения наша матрица – это массив вещественных чисел размерностью 5×5 ячеек.

```
type TColorMatrix = packed array[0..4, 0..4] of Single;
```

С точки зрения графической подсистемы Windows матрица позволяет воспринимать цвет как вектор в четырехмерном пространстве (каналы RGB и канал прозрачности A). Первые четыре колонки матрицы (рис. 40.9) хранят управляющие значения для красной, зеленой, синей составляющей и канала прозрачности. В ячейку на пересечении 5-го ряда и 5-го столбца всегда установлена единица, оставшиеся 4 элемента пятого столбца хранят нули.

Исходное состояние матрицы следующее: главная диагональ заполнена единицами, а в остальных ячейках находятся нулевые значения. Умножение цветового вектора на такую матрицу не приведет ни к каким изменениям –

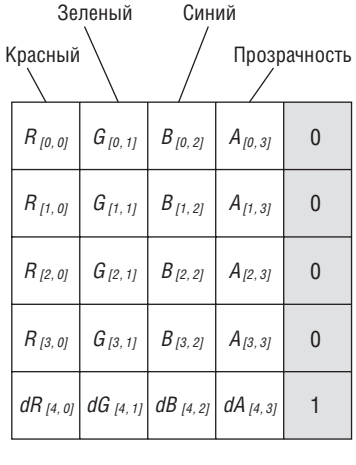


Рис. 40.9. Структура цветовой матрицы TColorMatrix

на выходе получим вектор, идентичный входному. Но любое отклонение в ячейках матрицы от исходного приводит к искажению цветового вектора.

Рассмотрим одну из простейших операций преобразования цвета – *масштабирование*. В этом случае основная роль отводится значениям, хранящимся на главной диагонали матрицы, – они рассматриваются как множители масштабирования. Поступающие на вход матрицы цветовые каналы R, G, B и A умножаются на соответствующий множитель. Если в ячейке матрицы находится единица, то цвет не поменяется, умножение на ноль отключит цветовой канал, на 0,5 – вдвое ослабит, на 2 – вдвое усилит и т. д.

Ответственность за масштабирование (усиление и ослабление цвета) возложена на элементы главной диагонали матрицы. Элементы $R_{[0,0]}$, $G_{[1,1]}$, $B_{[2,2]}$ и $A_{[3,3]}$ выполняют роль множителей.

Еще одна, весьма распространенная операция коррекции цвета называется *трансляцией*. Для трансляции цвета к поступающим на вход матрицы данным ARGB добавляются находящиеся в нижней строке матрицы значения смещения.

За трансляцию (перенос) значений цвета отвечает пятая строка матрицы `TColorMatrix` *– элементы $dR_{[4,0]}$, $dG_{[4,1]}$, $dB_{[4,2]}$ и $dA_{[4,2]}$.*

А теперь попробуем математически смоделировать эффект, одновременно включающий операции масштабирования и трансляции цветов. Допустим, на вход матрицы поступает чистый белый цвет (255; 255; 255; 255). В матричной нотации этот цвет записывается следующим образом (1,0; 1,0; 1,0, 1,0; 1,0). На главной диагонали матрицы хранятся следующие множители (0,2; 1,0; 0,7; 1,0), а в пятой строке – (0,2; 0,0; 0,0; 1,0). Проведем нехитрые вычисления:

$$(1{,}0;\ 1{,}0;\ 1{,}0;\ 1{,}0) \times \begin{bmatrix} 0{,}2 & 0{,}0 & 0{,}0 & 0{,}0 & 0{,}0 \\ 0{,}0 & 1{,}0 & 0{,}0 & 0{,}0 & 0{,}0 \\ 0{,}0 & 0{,}0 & 0{,}7 & 0{,}0 & 0{,}0 \\ 0{,}0 & 0{,}0 & 0{,}0 & 1{,}0 & 0{,}0 \\ 0{,}2 & 0{,}0 & 0{,}0 & 0{,}0 & 1{,}0 \end{bmatrix} = (0{,}4;\ 1{,}0;\ 0{,}7;\ 1{,}0)$$

В итоге мы получим следующий оттенок: (1,0×0,2+0,2; 1×1+0; 1×0,7+0; 1×1+0)=(0,4; 1; 0,7; 1). Если вам привычнее описание цвета, нормированного не к единице, а к числу 255, то результат таков – (0,4×255; 1×255; 0,7×255; 1×255)=(102; 255; 179; 255).

При проведении экспериментов с `TColorMatrix` *не ставьте матрице запредельных задач. Не стоит забывать, что результирующие значения цветовых составляющих имеют непреодолимые ограничения – предельное значение усиливаемого цвета не может превысить 1, а минимальное не упадет ниже 0.*

Матрица способна и на более значительные подвиги. Например, операция по *повороту цвета* ввергнет в трепет любого любителя острых ощущений. Вре-

менно забудем о существовании канала прозрачности и вновь (по аналогии с главой 13 этой книги) рассмотрим цветовое пространство как куб с осями Red, Green и Blue, выходящими из начала координат (0; 0; 0), а значения цветов станем нормировать к единице. Каждый, кто хотя бы немного знаком с геометрией, подтвердит, что любую из точек, находящуюся на поверхности или внутри куба, можно адресовать не только по значениям цветовых составляющих (например (1; 0,4; 0)), но и в полярных координатах. В этом случае нам требуется знать величину вектора (отвечающего за интенсивность света) и углы проекции вектора к трем цветовым осям. Введение в описание цвета такого математического понятия, как вектор, позволяет нам говорить не только о масштабировании или трансляции, но и о поворотах цвета (поворотах вектора) – за счет изменения углов наклона вектора к цветовым плоскостям.

На рис. 40.10 представлено состояние ячеек матрицы при проведении поворотов цвета. Допустим, что нам требуется повернуть вектор на угол α градусов от оси красного цвета R к оси зеленого цвета G. В этом случае задействуем элементы матрицы $R_{[0,0]}$, $R_{[0,1]}$, $G_{[0,1]}$ и $G_{[1,1]}$. В перечисленные ячейки передаются косинусы и синусы угла α в соответствии с матрицей, изображенной на рис. 40.10.

Возможности матрицы далеко не ограничиваются вопросами масштабирования, трансляции и поворотами цветового вектора. Программисты из Microsoft поработали над ней так, что зачастую предсказать результат трансформации цветов простому смертному не получится. Чтобы почувствовать себя высшим существом, можете день-другой поэкспериментировать с соответствующим примером к этой главе (см. рис. 40.11 на цв. вклейке, с. 424), а мы вновь вернемся практической стороне вопроса программирования.

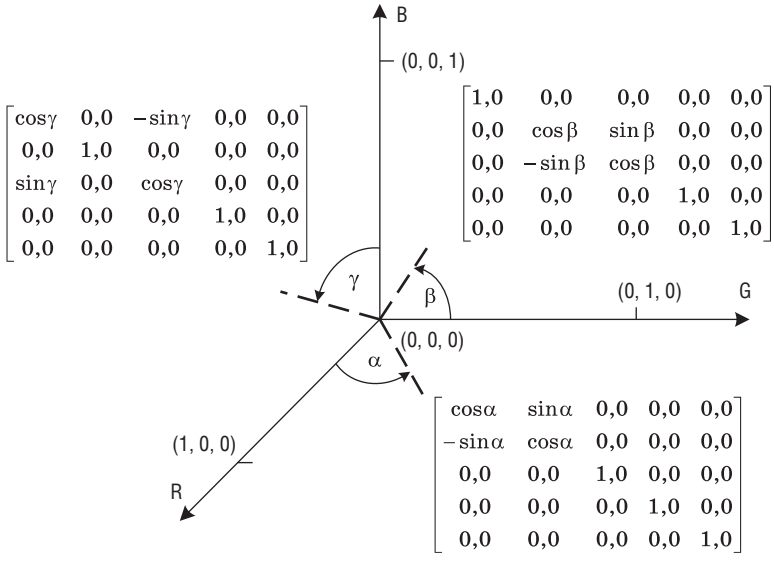


Рис. 40.10. Состояние матрицы при осуществлении поворотов цветов

Для того чтобы проверить наши теоретические изыскания относительно матрицы цветов на практике, стоит вновь обратиться к классу TGPImageAttributes. Именно он ведает вопросом переноса информации из матрицы в атрибуты изображения, для этого контейнер цветовых атрибутов обзавелся методом:

```
function SetColorMatrix(const ColorMatrix: TColorMatrix;
                        Mode: TColorMatrixFlags = ColorMatrixFlagsDefault;
                  type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Описание матрицы передается в первый параметр функции. Кроме этого, метод не прочь узнать наше отношение к оттенкам серого цвета. Для этого следует поработать со вторым параметром Mode, он имеет право принимать одно из трех значений.

```
type TColorMatrixFlags = (ColorMatrixFlagsDefault,
                          ColorMatrixFlagsSkipGrays,
                          ColorMatrixFlagsAltGray);
```

В состоянии по умолчанию (значение ColorAdjustTypeDefault) коррекция цвета затронет все цвета. Направив в параметр значение ColorMatrixFlagsSkipGrays, мы укажем графической системе, что во время проведения коррекции изображения оттенки серого (например, тени) не должны быть затронуты. Значение ColorMatrixFlagsAltGray говорит GDI+, что описание преобразования оттенков серого хранится в другой матрице.

Сброс матрицы в исходное состояние производится с помощью метода:

```
function ClearColorMatrix(
                type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Многие изображения состоят не только из цветных областей, но и содержат черные и серые элементы. Обычно это контуры рисунка, штриховка, тени, текстовые подписи. Иногда применение цветовых фильтров к окрашенным в градации серого элементам приведет не к улучшению, а наоборот, к искажению смысла изображения. Поэтому контейнер цветных атрибутов TGPImageAttributes обучен не только различать цветные элементы рисунка, но и выделять из них участки, закрашенные оттенками серого. Как следствие, цветовую коррекцию также допускается проводить избирательно, для этого предназначен метод:

```
function SetColorMatrices(const ColorMatrix: TColorMatrix;
                          const GrayMatrix: TColorMatrix;
                          Mode: TColorMatrixFlags  = ColorMatrixFlagsDefault;
                 Type_: TColorAdjustType  = ColorAdjustTypeDefault): TStatus;
```

На вход функции поступает информация о двух матрицах: матрице цвета ColorMatrix и матрице оттенков серого GrayMatrix. Возврат к исходному состоянию на этот раз обеспечит метод:

```
function ClearColorMatrices(
               Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Разложение цвета в модель CMYK

Вопросами преобразования цветовой схемы в соответствии с предпочтениями модели CMYK ведает метод:

```
function SetOutputChannel(ChannelFlags: TColorChannelFlags;
                  Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Для получения запланированного результата программисту требуется поколдовать с параметром ChannelFlags. Здесь указывается цвет канала, который мы рассчитываем получить.

```
type  TColorChannelFlags = (
    ColorChannelFlagsC, // голубой цвет
    ColorChannelFlagsM, // пурпурный цвет
    ColorChannelFlagsY, // желтый цвет
    ColorChannelFlagsK, // черный цвет
    ColorChannelFlagsLast
  );
```

Многие создатели программного обеспечения и графических устройств стараются, чтобы их мониторы, принтеры, сканеры и прочее оборудование работали с эксклюзивными настройками. Цветовые профайлы этих производителей (файлы с расширением «icc») прячутся в папке Windows\System32\Spool\Drivers\Color. Для того чтобы графический вывод стал осуществляться под управлением настроек производителя, название файла передается в первый параметр метода:

```
function SetOutputChannelColorProfile(ColorProfileFilename: WideString;
                  type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Второй параметр нам уже знаком, он встречался в функции SetOutputChannel. Получив все необходимое, метод возвратит любой цветовой канал схемы CMYK в соответствии с указаниями профайла.

Названия методов, сбрасывающих настройки по разложению изображения в цветовую схему CMYK, традиционно начинаются с приставки «Clear»:

```
function ClearOutputChannel(
           Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
function ClearOutputChannelColorProfile(
           Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Мозаика

Один из немногих методов класса TGPImageAttributes, не обладающих параметром Type, отвечает за управление повторами графических объектов, что позволяет создавать из них простейшие мозаичные полотна.

```
function SetWrapMode(Wrap: TWrapMode;
                     Color: TGPColor = aclBlack;
                     Clamp: BOOL = FALSE): TStatus;
```

Кроме знакомого еще по главе о кистях GDI+ параметра Wrap (собственно, и определяющего способы повторения изображения) читатель встретит необязательные параметры Color и Clamp. Первый из них работоспособен только в режиме WrapModeClamp, здесь он назначает цвет фона. О назначении второго аргумента Clamp, как мне кажется, до конца не знают даже в Microsoft, так что предположим, что он найдет себе применение в будущих версиях GDI+.

Так сильно «намучавшись» с атрибутами, очень хочется все вернуть в исходное. К счастью, это не запрещено, надо только попросить метод:

```
function Reset(Type_: TColorAdjustType = ColorAdjustTypeDefault): TStatus;
```

Метод сбросит все установки, связанные с преобразованиями цветов с учетом состояния параметра Type. Здесь определены «заводские» регулировки для основных графических объектов. Ко всему прочему можно и выбрать универсальное состояние ColorAdjustTypeDefault (оно же по умолчанию).

Особенности цветовой коррекции кистей

Класс TImageAttributes не оставил без своего внимания и такого ключевого объекта всей графической системы, как кисть. Здесь (по аналогии с методом SetColorKey) нам позволено замещать цвета в рисунке кисти.

```
function SetBrushRemapTable(MapSize: Cardinal; Map: PColorMap): TStatus;
```

Число замещаемых цветов определяется в аргументе MapSize, ссылка на цветовой массив передается во второй параметр Map. Отмена эффекта обеспечивается функцией:

```
function ClearBrushRemapTable: TStatus;
```

РЕЗЮМЕ

Одно из неоспоримых преимуществ обновленной графической библиотеки над обычным GDI в том, что она располагает набором инструментов, позволяющих управлять качеством вывода графических примитивов, изображений и текстовых строк. Подавляющее большинство методов улучшения качества нацелены на борьбу с главным врагом растровых устройств вывода – ступенчатым эффектом и основаны на различных вариантах *технологии сглаживания* (*antialiasing*).

Еще одно существенное нововведение GDI+ заключается в наличии широкого спектра методов, осуществляющих коррекцию цвета изображений. В перечень поддерживаемых цветовых фильтров входят как простейшие методы замещения цветов, так и сложные эффекты, позволяющие масштабировать, транслировать и поворачивать цветовой вектор.

Заключение

На сегодняшний день на рынке сред разработки программного обеспечения сложилась парадоксальная ситуация! Выход в свет высокоэффективных объектно-ориентированных сред разработки (речь не только о Delphi) привел к появлению большого количества плохо спроектированных, неэффективных и малопригодных к использованию приложений. Число слабых во всех отношениях программ множится в геометрической прогрессии от года к году и, кажется, нет этому нашествию конца и края. Причин тому великое множество, и, как бы это ни показалось странным, в первую очередь всему виной дружелюбность и совершенство современных систем проектирования. Они позволяют написать жизнеспособное приложение «программисту», который состоит в конфронтации даже с таблицей умножения. Что уж говорить о знании архитектуры компьютера, операционной системы, основ разработки эффективных алгоритмов и самого главного инструмента – языка программирования...

Задумав написать книгу, посвященную графической подсистеме Windows, я преследовал цель не только раскрыть возможности двумерной деловой графики в проектах Delphi. Кроме того, я рассчитывал расширить кругозор начинающего программиста. Судите сами. Подавляющее большинство книг, посвященных программированию в Delphi, концентрируют внимание читателя исключительно на возможностях библиотеки визуальных компонентов среды разработки. Возможно, в этом есть свой резон, ведь с помощью Delphi для решения поставленной задачи достаточно сделать несколько щелчков мышью и написать пару строк кода. Шаг влево или вправо от визуальной библиотеки компонентов в подобных книгах классифицируется как хакерство! Если быть последовательным, то в век электроники способность человека считать в уме очень скоро будет рассматриваться, как чародейство... Но вернемся к VCL. Действительно, библиотека компонентов позволяет разработать простейшую программу любому новичку, но он должен понять, что пределы возможностей языка программирования не ограничиваются VCL, это лишь одна сторона медали. На самом деле Delphi – это мощный и совершенный инструмент, позволяющий создавать высокоэффективные программные продукты, опирающиеся не только на библиотеку компонентов, но и непосредственно на прикладной интерфейс программирования Windows API.

Есть еще один момент, на который я хотел бы обратить внимание. Печально, но факт – на книжных полках магазинов практически отсутствует литература по Delphi, ориентированная на профессионалов. А им не нужны книги, в которых глубоко и обстоятельно объясняется порядок размещения кнопки на рабочей форме проекта. Именно поэтому я постарался убрать со страниц всю «воду» и сформировать материал о GDI и GDI+ таким образом, чтобы изданием можно было пользоваться и как справочником для специалиста.

Что дальше? Технологии GDI и GDI+ вовсе не являются пределом возможностей Windows, но они представляет собой отправную точку для понимания других механизмов работы с графикой, например таких, как OpenGL и DirectDraw. К счастью, процесс познания бесконечен.

Литература

1. Рихтер Дж. «Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows», 4-е изд. – Пер. с англ. – СПб.: Питер; М.: Русская Редакция, 2001.
2. Руссинович М., Соломон Д. «Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер-класс», 4-е изд. – Пер. с англ. – М.: Русская редакция; СПб.: Питер, 2005.
3. Саймон Р. «Windows 2000 API. Энциклопедия программиста», – К.; СПб.: ДиаСофтЮП, 2002.
4. Петзольд Ч. «Программирование для Windows 95», в двух томах. – Пер. с англ. – СПб.: БХВ–Петербург, 1997.
5. Петзольд Ч. «Программирование для Microsoft Windows на C#», в двух томах. – Пер. с англ. – М.: Русская редакция, 2002.
6. Юань Фень. «Программирование графики для Windows». – СПб.: Питер, 2002.
7. Ласло М. «Вычислительная геометрия и компьютерная графика на C++». – Пер. с англ. – М.: БИНОМ, 1997.
8. Кузан Д.Я., Шапоров В.Н. «Программирование Win32 API в Delphi». – СПб.: БХВ–Петербург, 2005.
9. Пореев В.Н. «Компьютерная графика». – СПб.: БХВ-Петербург, 2004.
10. Миано Дж. «Форматы и алгоритмы сжатия изображений в действии». – М.: Триумф, 2003.
11. Краснов М. «OpenGL графика в проектах Delphi». – СПб.: БХВ–Петербург, 2000.
12. Румянцев П.В. «Азбука программирования в Win 32 API», 2-е изд. – М.: Радио и связь, Горячая линия – Телеком, 1999.
13. Евченко А.И. «OpenGL и DirectX: программирование графики. Для профессионалов». – СПб.: Питер, 2006.
14. Осипов Д. «Delphi. Профессиональное программирование». – СПб.: Символ-Плюс, 2006.
15. Робинсон С. и др. «C# для профессионалов. Том II. – Пер. с англ. – М.: Лори, 2003.

16. Ганеев Р. «Проектирование интерфейса пользователя средствами Win32 API», 2-е изд.– М.: Горячая Линия – Телеком, 2006.
17. Справочная система Microsoft Developers Network (MSDN Library). *http://msdn.microsoft.com*
18. Справочная система Delphi.
19. Бакнелл Дж. «Функциональные алгоритмы и структуры данных в Delphi. Библиотека программиста». – М.: ДиаСофтЮП; СПб.: Питер, 2006.
20. Вигерс Карл «Разработка требований к программному обеспечению». – Пер. с англ. – М.: Русская редакция, 2004.
21. Боровский А. «Программирование в Delphi 2005». – СПб.: БХВ-Петербург, 2005.

Алфавитный указатель

A
antialiasing, 616

C
class, 314

D
DIB-растр, 220, 345
DIB-секция, 229

E
EXIF, 557

G
Windows GDI
  AbortPath, 124
  AddFontResource, 280
  AddFontResourceEx, 281
  AlphaBlend, 300
  AngleArc, 94, 108
  Arc, 93
  ArcTo, 94
  BeginPath, 121
  BitBlt, 241
  ChangeDisplaySettings, 65
  ChangeDisplaySettingsEx, 67
  Chord, 110
  ClientToScreen, 155
  CloseEnhMetaFile, 258
  CloseFigure, 124
  CombineRgn, 136
  CombineTransform, 181
  CreateBitmap, 211
  CreateBitmapIndirect, 212
  CreateBrushIndirect, 75, 320
  CreateCompatibleBitmap, 214
  CreateCompatibleDC, 47
  CreateDC, 41
  CreateDIBPatternBrush, 75
  CreateDIBPatternBrushPt, 75
  CreateDIBSection, 230
  CreateDiscardableBitmap, 215
  CreateEllipticRgn, 131
  CreateEllipticRgnIndirect, 131
  CreateEnhMetaFile, 257
  CreateFont, 274, 322
  CreateFontInderect, 275
  CreateFontIndirectEx, 276
  CreateHalftonePalette, 203
  CreateHatchBrush, 73
  CreateIC, 50
  CreatePalette, 205
  CreatePatternBrush, 74
  CreatePen, 79
  CreatePenIndirect, 81
  CreatePolygonRgn, 132
  CreatePolyPolygonRgn, 132
  CreateRectRgn, 131
  CreateRectRgnIndirect, 131
  CreateRoundRectRgn, 131
  CreateSolidBrush, 73
  DeleteDC, 46
  DeleteEnhMetaFile, 257
  DeleteObject, 52
  DrawCaption, 116
  DrawEdge, 112
  DrawFocusRect, 115
  DrawFrameControl, 113
  DrawState, 117
  DrawText, 291
  DrawTextEx, 293
  EndDoc, 385
  EndPage, 385
  EndPaint, 42
  EndPath, 122
  EnumDisplayDevices, 62
  EnumDisplaySettings, 64
  EnumEnhMetaFile, 260
  EnumFontFamiliesEx, 272
  EqualRect, 102
  EqualRgn, 136
  ExcludeClipRect, 148
  ExtCreatePen, 82, 318
  ExtCreateRegion, 139
  ExtFloodFill, 106
  ExtSelectClipRgn, 148
  ExtTextOut, 284
  FillPath, 124
  FillRect, 105
  FillRgn, 133
  FlattenPath, 125
  FrameRgn, 134
  GDIComment, 264
  GetArcDirection, 94
  GetBkColor, 287
  GetBkMode, 287
  GetBrushOrgEx, 76
  GetCharABCWidths, 279
  GetCharWidth32, 279
  GetClipBox, 145
  GetClipRgn, 145, 146
  GetCurrentObject, 50
  GetDC, 43
  GetDCBrushColor, 70
  GetDCEx, 44
  GetDeviceCaps, 50
  GetDIBColorTable, 232
  GetDIBits, 228
  GetEnhMetaFile, 256
  GetEnhMetaFileBits, 259
  GetEnhMetaFileDescription, 263
  GetEnhMetaFileHeader, 256
  GetEnhMetaFilePaletteEntries, 254
  GetGraphicsMode, 175, 176
  GetMapMode, 159
  GetMetaRgn, 145
  GetNearestColor, 207
  GetNearestPaletteIndex, 207
  GetObject, 53
  GetObjectType, 53
  GetPaletteEntries, 205
  GetPath, 126
  GetPixel, 192
  GetPolyFillMode, 112
  GetRandomRgn, 146
  GetRegionData, 138
  GetRgnBox, 137
  GetROP2, 237
  GetStockObject, 48
  GetStretchBltMode, 226
  GetSysColorBrush, 72
  GetTextAlign, 288
  GetTextColor, 286
  GetTextFace, 278
  GetTextMetrics, 278
  GetViewportExtEx, 169
  GetWindowDC, 44
  GetWindowExtEx, 169
  GetWindowOrgEx, 164
  GetWorldTransform, 181
  GradientFill, 305
  GrayString, 297
  InflateRect, 102
  IntersectClipRect, 148
  IntersectRect, 103
  InvalidateRect, 24
  InvalidateRgn, 151

Windows GDI
IsRectEmpty, 101
LineDDA, 89
LoadBitmap, 216
MapWindowPoints, 157
MaskBlt, 246
ModifyWorldTransform, 181
MoveToEx, 87
OffsetClipRgn, 149
OffsetRect, 103
OffsetRgn, 136
PaintDesktop, 106
PaintRgn, 133
PatBlt, 239
Pie, 109
PlayEnhMetaFile, 256
PlayEnhMetaFileRecord, 261
PlgBlt, 248, 249
PolyBezierTo, 95
PolyDraw, 98
Polygon, 110
Polyline, 91
PolyPolygon, 111
PolyPolyline, 91
PolyTextOut, 285
PtInRect, 103
PtInRegion, 135
PtVisible, 149
RealizePalette, 204
Rectangle, 108
RectInRegion, 136
RectVisible, 149
ReleaseDC, 47
RemoveFontResource, 280
RemoveFontResourceEx, 281
ResizePalette, 206
RestoreDC, 52
SaveDC, 51
ScaleWindowExtEx, 174
ScreenToClient, 156
SelectClipRgn, 147
SelectObject, 52
SelectPalette, 203
SetArcDirection, 94
SetBkColor, 287
SetBkMode, 287
SetBrushOrgEx, 76
SetDCBrushColor, 70
SetDCPenColor, 79
SetDIBColorTable, 232
SetDIBits, 228
SetDIBitsToDevice, 225
SetEnhMetaFileBits, 259
SetMapMode, 159
SetMetaRgn, 149
SetMiterLimit, 83
SetPixel, 190
SetPolyFillMode, 112
SetRect, 101
SetRectEmpty, 101
SetRectRgn, 137
SetROP2, 237
SetStretchBltMode, 226
SetSystemPaletteUse, 201
SetTextAlign, 288
SetTextCharacterExtra, 287
SetTextColor, 286
SetTextJustification, 288
SetViewportOrgEx, 163, 164
SetWindowExtEx, 167
SetWindowOrgEx, 164
SetWorldTransform, 176
StartDoc, 384
StartPage, 385
StretchBlt, 245
StretchDIBits, 224
StrokeAndFillPath, 123, 296
StrokePath, 296
SubtractRect, 104
TabbedTextOut, 290
TextOut, 283
TransparentBlt, 247
UnionRect, 104
UnrealizeObject, 76, 204
UpdateColors, 207
ValidateRect, 24
ValidateRgn, 151
WidenPath, 125
WindowFromDC, 46
GDI+
ARGB, 411
ColorRefToARGB, 413
GetAlpha, 413
GetBlue, 413
GetGreen, 413
GetImageDecoders, 532
GetImageDecodersSize, 532
GetImageEncoders, 531
GetImageEncodersSize, 531
GetLastStatus, 414
GetRed, 413
MakeColor, 412
MakePoint, 414
MakeRect, 414
TGPColor, 411

P

Printer, 386
GetPrinter, 387

R

RTTI, 316

T

TBevel
Shape, 379
Style, 380
TBitmap
Dormant, 347
FreeImage, 347
Handle, 347
HandleType, 346
IgnorePalette, 347
LoadFromResourceID, 347
LoadFromResourceName, 347
Mask, 348
MaskHandle, 348
Monochrome, 346
Palette, 347
PixelFormat, 346
ReleaseHandle, 347
ReleasePalette, 347
ScanLine, 346
TransparentColor, 347
TransparentMode, 347
TBrush, 320
Assign, 322
Bitmap, 322
Color, 321
Handle, 320
Style, 320
TBrushRecall
Create, 326
TCanvas
Arc, 336
Brush, 332
BrushCopy, 337
Chord, 336
ClipRect, 333
CopyMode, 333, 337
CopyRect, 337
Draw, 339
DrawFocusRect, 337
Ellipse, 336
FillRect, 337
FloodFill, 337
Font, 332
FrameRect, 336
Handle, 331
LineTo, 334
Lock, 340
LockCount, 340
MoveTo, 334
Pen, 332
PenPos, 333
Pie, 336
Pixels, 333
PolyBexierTo, 335
Polygon, 334
Polyline, 334
Rectangle, 336
RoundRect, 336
StretchDraw, 339
TextExtent, 339
TextFlags, 339
TextHeight, 339
TextOut, 338
TextRect, 339
TextWidth, 339
TryLock, 340
Unlock, 340
TCommonDialog
Execute, 397

TComponent
Name, 368
Tag, 368
TControl
Align, 371
Anchors, 372
BoundsRect, 370
BringToFront, 372
ClientHeight, 371
ClientOrigin, 371
ClientRect, 371
ClientWidth, 371
Constraints, 371
Enabled, 373
Height, 370
Hide, 372
Left, 370
Parent, 369
SendToBack, 372
Show, 372
Top, 370
Visible, 372
TFont, 322
Charset, 323
Color, 323
FontAdapter, 323
Handle, 322
Height, 323
Name, 322
Orientation, 323
Pitch, 323
PixelsPerInch, 323
Size, 323
Style, 323
TGPBitmap
Clone, 570
Create, 567, 568, 569
GetHBITMAP, 570
GetHICON, 570
GetPixel, 571
LockBits, 572
SetPixel, 571
SetResolution, 572
TGPBrush
Clone, 426
GetLastStatus, 426
GetType, 425
TGPCustomLineCap
Create, 456
GetBaseInset, 456
GetWidthScale, 457
SetBaseInset, 456
SetWidthScale, 457
TGPFont
Create, 589, 590, 591
GetFamily, 592
GetHeight, 592
GetLogFontA, 592
GetLogFontW, 592
GetSize, 592
GetStyle, 592
GetUnit, 592
IsAvailable, 592
TGPFontCollection
AddFontFile, 595
AddMemoryFont, 595
GetFamilies, 594
GetFamilyCount, 594
TGPFontFamily
Create, 592
GenericMonospace, 592
GenericSansSerif, 592
GenericSerif, 592
GetCellAscent, 593
GetCellDescent, 593
GetEmHeight, 593
GetFamilyName, 593
GetLineSpacing, 593
IsAvailable, 593
IsStyleAvailable, 593
TGPGraphics
AddMetafileComment, 584
BeginContainer, 625
Create, 410
DrawArc, 507
DrawBezier, 502
DrawBeziers, 502
DrawClosedCurve, 504
DrawCurve, 503
DrawEllipse, 506
DrawImage, 551
DrawLine, 500
DrawLines, 500
DrawPath, 462
DrawPie, 507, 508
DrawPolygon, 501
DrawRectangle, 506
DrawRectangles, 506
EnumerateMetafile, 584
ExcludeClip, 497
FillClosedCurve, 505
FillEllipse, 506
FillPath, 462
FillPie, 507, 508
FillPolygon, 501
FillRectangles, 506
FillRegion, 491
FromHDC, 411
FromHWND, 411
FromImage, 411
GetBlend, 441
GetBlendCount, 440
GetClip, 498
GetClipBounds, 498
GetCompositingMode, 619
GetCompositingQuality, 619
GetDpiX, 511
GetDpiY, 511
GetHalftonePalette, 539
GetInterpolationMode, 620
GetPageScale, 511
GetPageUnit, 511
GetPixelOffsetMode, 618
GetSmoothingMode, 618
GetTextContrast, 623
GetTextRenderingHint, 622
GetTransform, 528
GetVisibleClipBounds, 498
GraphicsContainer, 625
IntersectClip, 497
IsClipEmpty, 498
IsVisibleClipEmpty, 498
MultiplyTransform, 528
ResetClip, 497
ResetTransform, 528
Restore, 623
RotateTransform, 528
Save, 623
ScaleTransform, 528
SetBlendBellShape, 441
SetBlendTriangularShape, 441
SetClip, 495, 496
SetCompositingMode, 619
SetCompositingQuality, 619
SetInterpolationMode, 620
SetPageScale, 511
SetPageUnit, 511
SetPixelOffsetMode, 618
SetRenderingOrigin, 430
SetSmoothingMode, 617
SetTextContrast, 623
SetTextRenderingHint, 621
SetTransform, 528
TransformPoints, 528
TranslateClip, 498
TranslateTransform, 528
TGPGraphicsPath
AddArc, 467
AddBezier, 468
AddBeziers, 468
AddClosedCurve, 469
AddCurve, 468
AddEllipse, 467
AddLine, 466
AddLines, 466
AddPie, 467
AddPolygon, 467
AddRectangle, 466
AddRectangles, 467
AddString, 469
ClearMarkers, 479
CloseAllFigures, 464
CloseFigure, 464
Create, 462, 463
Flatten, 470
GetBounds, 477
GetFillMode, 464
GetLastPoint, 476
GetPathData, 476
GetPathPoints, 474
GetPathTypes, 475
GetPointCount, 474
IsOutlineVisible, 478
IsVisible, 477
Outline, 471
Reset, 466
Reverse, 466
SetFillMode, 464

SetMarker, 479
StartFigure, 464
Transform, 470
Warp, 473
Widen, 471
TGPHatchBrush
Create, 428
GetBackgroundColor, 429
GetForegroundColor, 429
GetHatchStyle, 429
TGPImage
Create, 533
FromFile, 534
FromStream, 534
GetAllPropertyItems, 562
GetBounds, 535
GetEncoderParameterList, 546
GetEncoderParameterListSize, 546
GetFlags, 535
GetFrameCount, 542
GetFrameDimensionsCount, 542
GetFrameDimensionsList, 542
GetHeight, 535
GetHorizontalResolution, 535
GetPalette, 539
GetPaletteSize, 539
GetPhysicalDimension, 535
GetPixelFormat, 535
GetPropertyCount, 560
GetPropertyIdList, 560
GetPropertyItem, 560
GetPropertyItemSize, 560
GetPropertySize, 562
GetRawFormat, 534
GetThumbnailImage, 563
GetType, 534
GetVerticalResolution, 535
GetWidth, 535
RemovePropertyItem, 562
RotateFlip, 541
Save, 543
SaveAdd, 550
SelectActiveFrame, 543
SetPalette, 540
SetPropertyItem, 562
TGPImageAttributes
ClearBrushRemapTable, 636
ClearColorKey, 631
ClearColorMatrices, 634
ClearColorMatrix, 634
ClearGamma, 629
ClearNoOp, 627
ClearOutputChannel, 635
ClearOutputChannelColorProfile, 635
ClearRemapTable, 628
ClearThreshold, 630
Create, 626
Reset, 636
SetBrushRemapTable, 636
SetColorKey, 630
SetColorMatrices, 634
SetColorMatrix, 634
SetGamma, 629
SetNoOp, 627
SetOutputChannel, 635
SetOutputChannelColorProfile, 635
SetRemapTable, 627
SetThreshold, 629
SetToIdentity, 626
SetWrapMode, 635
TGPLinearGradientBrush
Create, 436, 437, 438
GetGammaCorrection, 439
GetInterpolationColorCount, 442
GetInterpolationColors, 442
GetLinearColors, 437
GetRectangle, 437
SetBlend, 439
SetGammaCorrection, 439
SetInterpolationColors, 441
SetLinearColors, 437
TGPMatrix
Create, 514
Equals, 527
GetElements, 516
Invert, 522
IsIdentity, 527
IsInvertible, 527
Multiply, 526
OffsetX, 516
OffsetY, 516
Reset, 524
Rotate, 521
RotateAt, 521
Scale, 518
SetElements, 514
Shear, 520
TransformPoints, 524
TransformVectors, 525
Translate, 516
TGPMetafile
Create, 580, 581
EmfToWmfBits, 581
GetHENHMETAFILE, 575
TGPPathGradientBrush
Create, 442, 443
GetCenterColor, 444
GetCenterPoint, 444
GetFocusScales, 446
GetGraphicsPath, 445
GetPointCount, 443
GetSurroundColorCount, 444
GetSurroundColors, 444
SetCenterColor, 444
SetCenterPoint, 444
SetFocusScales, 445
SetSurroundColors, 443
TGPPen
Create, 448
GetBrush, 449
GetColor, 450
GetCompoundArray, 460
GetCompoundArrayCount, 460
GetCustomEndCap, 456
GetCustomStartCap, 456
GetDashOffset, 452
GetDashPattern, 451
GetDashPatternCount, 451
GetDashStyle, 450
GetEndCap, 454
GetLineJoin, 458
GetMiterLimit, 458
GetPenType, 449
GetStartCap, 454
GetWidth, 449
SetAlignment, 452
SetBrush, 449
SetColor, 450
SetCompoundArray, 459
SetCustomEndCap, 455
SetCustomStartCap, 455
SetDashCap, 453
SetDashOffset, 452
SetDashPattern, 450
SetDashStyle, 450
SetEndCap, 454
SetLineCap, 455
SetLineJoin, 457
SetMiterLimit, 458
SetStartCap, 454
SetWidth, 449
TGPRegion
Complement, 488
Create, 485
Equals, 487
Exclude, 488
FromHRGN, 486
GetBounds, 490
GetData, 493
GetDataSize, 493
GetHRGN, 486
GetRegionScans, 493
GetRegionScansCount, 493
Intersect, 488
IsEmpty, 487
IsInfinite, 487
IsVisible, 490, 492
MakeEmpty, 487
MakeInfinite, 486
Transform, 492
Translate, 492
Union, 488
Xor_, 488
TGPSolidBrush
Create, 426
GetColor, 427
SetColor, 427
TGPTextureBrush
Create, 430, 432

GetImage, 431
GetTransform, 434
GetWrapMode, 432
MultiplyTransform, 434
ResetTransform, 434
RotateTransform, 434
ScaleTransform, 434
SetTransform, 434
SetWrapMode, 432
TranslateTransform, 434
TGraphic
Empty, 343
Height, 343
LoadFromClipboardFormat, 342
LoadFromFile, 342
LoadFromStream, 342
Modified, 343
OnProgress, 342
Palette, 343
PaletteModified, 343
SaveToClipboardFormat, 342
SaveToFile, 342
SaveToStream, 342
SetSize, 344
Transparent, 343
TGraphicControl
Canvas, 376
Paint, 376
TGraphicsObject
OnChange, 318
OwnerCriticalSection, 318
TGraphicsPathIterator
CopyData, 482
Create, 480
Enumerate, 482
GetCount, 482
HasCurve, 482
NextMarker, 480
NextPathType, 482
NextSubpath, 482
Rewind, 481
TIcon
Handle, 344
Height, 344
ReleaseHandle, 344
Width, 344
TImage
Center, 377
IncrementalDisplay, 377
Picture, 376
Stretch, 377
Transparent, 377
TImageList
Add, 359
AddIcon, 359
AddImages, 360
AddMasked, 360
AllocBy, 360
BkColor, 364
BlendColor, 363
Clear, 362
Delete, 362
Draw, 363
DrawingStyle, 362
DrawOverlay, 363
FileLoad, 362
GetBitmap, 365
GetIcon, 365
GetImageBitmap, 364
GetImages, 365
GetInstRes, 361
GetMaskBitmap, 364
GetResource, 361
Height, 357
ImageIndex, 365
ImageType, 358
Masked, 364
Move, 360
OnChange, 362
Overlay, 364
ResInstLoad, 362
ResourceLoad, 361
StateIndex, 365
Width, 357
TJPEGImage
Assign, 353
Compress, 353
CompressionQuality, 353
DIBNeeded, 353
JPEGNeeded, 353
Performance, 354
PixelFormat, 354
ProgressiveDisplay, 353
ProgressiveEncoding, 353
Scale, 354
Smoothing, 354
TLabel
Alignment, 381
AutoSize, 381
Caption, 381
FocusControl, 382
Layout, 381
ShowAccelChar, 382
WordWrap, 381
TMetafile
CreatedBy, 351
Description, 351
Enhanced, 350
Handle, 351
Height, 351
Inch, 351
MMHeight, 351
MMWidth, 351
Palette, 351
Width, 351
TMetafileCanvas
Create, 351
TMetafileCanvas CreateWithComment, 352
TObject, 314, 315
Constructor, 316
Destructor, 316
Free, 316
NewInstance, 316
TPageSetupDialog
MarginLeft, 400
MinMarginBottom, 401
MinMarginLeft, 401
MinMarginRight, 401
MinMarginTop, 401
Options, 401
PageHeight, 401
PageWidth, 401
Units, 401
TPaintBox
OnPaint, 378
Paint, 378
TPen, 318
Color, 319
Handle, 319
Width, 319
TPenRecall
Create, 326
TPersistent, 316
Assign, 317
AssignTo, 317
TPicture
Bitmap, 355
Icon, 355
LoadFromClipboardFormat, 355
LoadFromFile, 355
Metafile, 355
SaveToClipboardFormat, 355
SaveToFile, 355
TPrintDialog
Copies, 400
FromPage, 399
MaxPage, 399
MinPage, 399
Options, 398
PrintRange, 399
PrintToFile, 399
ToPage, 399
TPrinter
Abort, 390
Aborted, 390
BeginDoc, 390
Canvas, 389
Capabilities, 389
EndDoc, 390
Fonts, 392
NewPage, 390
Orientation, 389
PageHeight, 389
PageNumber, 390
PageWidth, 389
PrinterIndex, 387
Printers, 386
SetPrinter, 388
Title, 390
TPrinterSetupDialog
Options, 398
TRecall
Forget, 326
Store, 325

TScreen
ActiveControl, 329
ActiveCustomForm, 329
ActiveForm, 329
Cursor, 328
Cursors, 328
CustomFormCount, 329
CustomForms, 329
DataModuleCount, 329
DataModules, 329
Fonts, 328
FormCount, 329
Forms, 329
Height, 327
HintFont, 329
IconFont, 329
MenuFont, 329
MonitorCount, 328
Monitors, 328
PixelsPerInch, 328
ResetFonts, 328
Width, 327
TShape
Brush, 378
Pen, 378
Shape, 377
TSpeedButton
AllowAllUp, 381
Down, 380
Flat, 380
Glyph, 380
GroupIndex, 380
Transparent, 380
TSplitter
AutoSnap, 379
MinSize, 378
OnCanResize, 379
OnMoved, 379
ResizeStyle, 379

W

Win32 API
ExtractAssociatedIcon, 344
Windows API
CallWindowProc, 36
ChooseFont, 276
CreateIconIndirect, 345
CreateWindow, 22
ExtractAssociatedIcon, 344
GetMessage, 25
GetSysColor, 193, 202
PostMessage, 28
RegisterClass, 21
SendMessage, 28
SetSysColors, 194, 202
SetWindowLong, 35
SetWindowRgn, 144
ShowWindow, 23
UpdateWindow, 23
WindowProc, 28

А

абстрактный класс, 315
аппаратно-независимый растр, 220
аффинные преобразования, 176, 434, 513

В

вывод текста, 338
выравнивание текста, 288

Д

деструктор, 316

З

запись
TDeviceMode, 197

И

инкапсуляция, 313
интерполяция растра, 619

К

кернинг, 287
кисть, 68, 320, 425
компонент
TBevel, 379
TImage, 376
TLabel, 381
TPageSetupDialog, 400
TPaintBox, 378
TPrintDialog, 398
TPrinterSetupDialog, 397
TShape, 377
TSpeedButton, 380
TSplitter, 378
константа
AD_CLOCKWISE, 94
AD_COUNTERCLOCKWISE, 94
ALTERNATE, 112
ANSI_FIXED_FONT, 49
ANSI_VAR_FONT, 49
BDR_RAISEDINNER, 113
BDR_RAISEDOUTER, 113
BDR_SUNKENINNER, 113
BDR_SUNKENOUTER, 113
BI_BITFIELDS, 223
BI_JPEG, 223
BI_PNG, 223
BI_RGB, 223
BI_RLE4, 223
BI_RLE8, 223
BLACK_BRUSH, 49, 69
BLACKNESS, 241
BLACKONWHITE, 226
BLACK_PEN, 49, 78
BS_DIBPATTERN, 71
BS_DIBPATTERN8X8, 71
BS_DIBPATTERNPT, 71
BS_HATCHED, 71
BS_HOLLOW, 71
BS_NULL, 71
BS_PATTERN, 71
BS_PATTERN8X8, 71
BS_SOLID, 71
COLORONCOLOR, 226
COMPLEXREGION, 137
DC_ACTIVE, 116
DC_BRUSH, 49, 69
DC_BUTTONS, 116
DC_GRADIENT, 116
DC_ICON, 116
DC_INBUTTON, 116
DC_PEN, 49, 78
DC_SMALLCAP, 116
DC_TEXT, 116
DCX_CACHE, 45
DCX_CLIPSCHILDREN, 45
DCX_CLIPSIBLINGS, 45
DCX_PARENTCLIP, 45
DCX_VALIDATE, 45
DCX_WINDOW, 45
DEFAULT_GUI_FONT, 49
DEFAULT_PALETTE, 49
DEVICE_DEFAULT_FONT, 49
DFC_BUTTON, 114
DFC_CAPTION, 114
DFC_MENU, 114
DFC_POPUPMENU, 114
DFCS_BUTTON3STATE, 114
DFCS_BUTTONCHECK, 114
DFCS_BUTTONPUSH, 114
DFCS_BUTTONRADIO, 114
DFCS_BUTTONRADIOIMAGE, 114
DFCS_BUTTONRADIOMASK, 114
DFC_SCROLL, 114
DIB_PAL_COLORS, 71, 75, 216
DIB_RGB_COLORS, 71, 75, 216
DISPLAY_DEVICE_ATTACHED_TO_DESKTOP, 63
DISPLAY_DEVICE_MIRRORING_DRIVER, 63
DISPLAY_DEVICE_MODESPRUNED, 63
DISPLAY_DEVICE_PRIMARY_DEVICE, 63
DISPLAY_DEVICE_REMOVABLE, 63
DISPLAY_DEVICE_VGA_COMPATIBLE, 63
DKGRAY_BRUSH, 49, 69
DSS_DISABLED, 118
DSS_HIDEPREFIX, 118
DSS_MONO, 118
DSS_NORMAL, 118
DSS_PREFIXONLY, 118
DSS_RIGHT, 118
DSS_UNION, 118
DST_BITMAP, 117

константа
DST_COMPLEX, 117
DST_ICON, 117
DSTINVERT, 239, 241
DST_PREFIXTEXT, 117
DST_TEXT, 117
DT_BOTTOM, 292
DT_CENTER, 292
DT_EDITCONTROL, 292
DT_END_ELLIPSIS, 293
DT_EXPANDTABS, 292
DT_HIDEPREFIX, 293
DT_LEFT, 292
DT_NOCLIP, 292
DT_NOFULLWIDTHCHAR-BREAK, 293
DT_NOPREFIX, 293
DT_PATH_ELLIPSIS, 293
DT_PREFIXONLY, 293
DT_RIGHT, 292
DT_RTLREADING, 292
DT_SINGLELINE, 292
DT_TABSTOP, 292
DT_TOP, 292
DT_VCENTER, 292
DT_WORDBREAK, 292
DT_WORD_ELLIPSIS, 293
EDGE_BUMP, 113
EDGE_ETCHED, 113
EDGE_RAISED, 113
EDGE_SUNKEN, 113
GRAY_BRUSH, 49, 69
GWL_WNDPROC, 35
HALFTONE, 226
HOLLOW_BRUSH, 49, 69
HS_BDIAGONAL, 71
HS_CROSS, 71
HS_DIAGCROSS, 71
HS_FDIAGONAL, 71
HS_HORIZONTAL, 71
HS_VERTICAL, 71
LTGRAY_BRUSH, 49, 69
MERGECOPY, 241
MERGEPAINT, 241
MM_ANIISOTROPIC, 173
MM_HIENGLISH, 158
MM_HIMETRIC, 158
MM_ISOTROPIC, 167
MM_LOENGLISH, 158
MM_LOMETRIC, 158
NOTSRCCOPY, 241
NOTSRCERASE, 241
NULL_BRUSH, 49, 69, 108
NULL_PEN, 78, 108
NULLREGION, 137
OBJ_BITMAP, 50, 53
OBJ_BRUSH, 50, 53
OBJ_COLORSPACE, 50, 53
OBJ_DC, 53
OBJ_ENHMETADC, 53
OBJ_ENHMETAFILE, 53
OBJ_EXTPEN, 53
OBJ_FONT, 50, 53
OBJ_MEMDC, 53
OBJ_METADC, 53
OBJ_METAFILE, 53
OBJ_PAL, 50, 53
OBJ_PEN, 50, 53
OBJ_REGION, 53
OEM_FIXED_FONT, 49
OPAQUE, 287
PATCOPY, 239, 241
PATINVERT, 239, 241
PATPAINT, 241
PC_EXPLICIT, 198
PC_NOCOLLAPSE, 198
PC_RESERVED, 198
PS_ALTERNATE, 79, 82
PS_COSMETIC, 82
PS_DASH, 79
PS_DASHDOT, 79
PS_DOT, 79
PS_ENDCAP_FLAT, 82
PS_ENDCAP_ROUND, 82
PS_ENDCAP_SQUARE, 82
PS_GEOMETRIC, 82
PS_INSIDEFRAME, 79
PS_JOIN_BEVEL, 82
PS_JOIN_MITER, 82
PS_JOIN_ROUND, 82
PS_NULL, 79
PS_SOLID, 79
PS_USERSTYLE, 79, 82
PT_BEZIERTO, 98
PT_CLOSEFIGURE, 98
PT_LINETO, 98
PT_MOVETO, 98
R2_BLACK, 237
R2_MASKNOTPEN, 237
R2_MASKPEN, 237
R2_MASKPENNOT, 237
R2_MERGENOTPEN, 237
R2_MERGEPEN, 237
R2_NOP, 237
R2_NOT, 237
R2_NOTCOPYPEN, 237
R2_NOTMASKPEN, 237
R2_NOTMERGEPEN, 237
R2_WHITE, 237
R2_XORPEN, 237
RGN_AND, 136
RGN_COPY, 136
RGN_DIFF, 136
RGN_OR, 136
RGN_XOR, 136
SIMPLEREGION, 137
SRCAND, 242
SRCCOPY, 242
SRCERASE, 242
SRCINVERT, 242
SRCPAINT, 242
STRETCH_ANDSCANS, 226
STRETCH_DELETES-CANS, 226
STRETCH_HALFTONE, 226
STRETCH_ORSCANS, 226
SYSPAL_NOSTATIC, 202
SYSPAL_STATIC, 202
SYSPAL_STATIC256, 202
SYSTEM_FIXED_FONT, 49
SYSTEM_FONT, 49
TRANSPARENT, 287
WHITE_BRUSH, 49, 69
WHITENESS, 239, 242
WHITEONBLACK, 226
WHITE_PEN, 49, 78
WINDING, 112
конструктор, 316
координаты
логические, 153
устройства, 153, 174
коррекция цвета, 626
кривая Безье, 335

М

макрос
GetBValue, 190
GetGValue, 190
GetRValue, 190
MAKEROP4, 246
PALETTEINDEX, 199
PALETTERGB, 199
RGB, 190
массив
Slice, 335
масштабирование, 174
матрица аффинных преобразований
TGPMatrix, 513
матрица цветов, 631
миниатюра изображения, 563
модуль
Jpeg, 353
Printers, 386

Н

наследование, 314

О

оконная процедура, 28
оконный элемент управления, 367

П

перо, 77, 318, 447
геометрическое, 81
косметическое, 79
поле объекта, 312
принтер, 383

Р

регион отсечения, 141, 495
режим отображения
MM_ANIISOTROPIC, 173
MM_ANISOTROPIC, 158
MM_HIENGLISH, 158
MM_HIMETRIC, 158

MM_ISOTROPIC, 158, 167
MM_LOENGLISH, 158
MM_LOMETRIC, 158

С

сглаживание, 616
событие
OnClick, 373
OnContextPopup, 373
OnDblClick, 373
OnDragDrop, 374
OnDragOver, 374
OnEndDock, 376
OnEndDrag, 374
OnMouseActivate, 373
OnMouseDown, 373
OnMouseEnter, 373
OnMouseLeave, 373
OnMouseUp, 373
OnPaint, 378
OnProgress, 342
OnStartDock, 376
OnStartDrag, 374
сообщение
WM_CHAR, 26
WM_COMMAND, 33
WM_FONTCHANGE, 280
WM_MOUSEMOVE, 193
WM_NCCLASSSIZE, 26
WM_NCPAINT, 26
WM_PAINT, 376
WM_PALETTECHANGED, 206
WM_PALETTEISCHANGING, 207
WM_QUIT, 26
WM_SIZE, 26
WM_SYSCOLORCHANGE, 202
структура
LOGBRUSH, 70
tagLogBrush, 84
TBitmapData, 572
TBitmapFileHeader, 222
TBitmapInfo, 224, 569
TBitmapInfoHeader, 222
TBlendFunction, 300
TChooseFont, 277
TColorMap, 627
TColorMatrix, 631
TColorPalette, 538
TDeviceMode, 64
TDeviceModeA, 387
TDIBSection, 230
TDisplay_DeviceA, 62
TDocInfo, 385
TDrawTextParams, 294
TEMR, 253
TEMREOF, 254
TEncoderParameter, 545
TEncoderParameters, 545
TEnhMetaHeader, 255
TEnhMetaRecord, 253
TGPPoint, 413
TGPPointF, 413
TGPRect, 414
TGPRectF, 414
TGPSize, 414
TGPSizeF, 414
TGradientRect, 306
TGradientTriangle, 306
THandleTable, 260
TIconInfo, 345
TImageCodecInfo, 532
TImageType, 534
TLogBrush, 70, 84
TLogFont, 271
TLogPalette, 198
TLOGPEN, 81
TMetafileHeader, 575
TPaintStruct, 42
TPoint, 101
TPolyText, 285
TPropertyItem, 558
TRect, 100
TRGBQuad, 223
TRGBTriple, 223
TRgnData, 138
TRgnDataHeader, 138
TSmallRect, 101
TTextMetric, 278
TTriVertex, 305
TWmfPlaceableFileHeader, 578
TWndClass, 19
TXForm, 177
субклассирование, 33

Т

тип данных
TAlign, 371
TAnchorKind, 372
TBevelShape, 379
TBrushStyle, 320
TColorAdjustType, 626
TCombineMode, 495
TCompositingMode, 619
TCoordinateSpace, 528
TDashCap, 453
TFillMode, 462
TFillStyle, 338
TFontPitch, 323
TFontStyle, 323
TJPEGPixelFormat, 354
TJPEGQualityRange, 353
TJPEGScale, 354
TLineJoin, 457
TMetafileType, 574
TPageSetupDialogOptions, 401
TPathPointType, 462
TPenMode, 320
TPenType, 449
TPixelFormat, 346
TPrintDialogOptions, 398
TProgressStage, 343
TShapeType, 378
TSmoothingMode, 617
TTransparentMode, 347
TUnit, 511
траектория
GDI, 120
GDI+, 461

Ф

файл
*.bmp, 346, 567
*.dib, 346
*.emf, 251, 350, 574, 580
*.gif, 531
*.jpg, 352
*.png, 531
*.tif, 530, 550
*.wmf, 350, 580
флаг
CDS_FULLSCREEN, 65, 197
CDS_GLOBAL, 65
CDS_NORESET, 65
CDS_RESET, 65
CDS_SET_PRIMARY, 65
CDS_TEST, 65
CDS_UPDATEREGISTRY, 65
ETO_CLIPPED, 284
ETO_IGNORELANGUAGE, 284
ETO_INDEX, 284
ETO_NUMERICSLATIN, 284
ETO_NUMERICSLOCAL, 284
ETO_OPAQUE, 284
ETO_PDY, 284
ETO_RTLREADING, 285
форматирование текста, 289
функции GraphUtil
GradientFillCanvas, 338
ScaleImage, 350

Х

холст GDI+, 409
холст VCL, 331

Ц

цветовая модель
CMY, 195, 397
CMYK, 195, 397, 635
HLS, 195
HSB, 195
RGB, 187
цветовые атрибуты, 626
цикл обработки сообщений, 24
цифровая фотография, 557

Ш

шрифт, 322