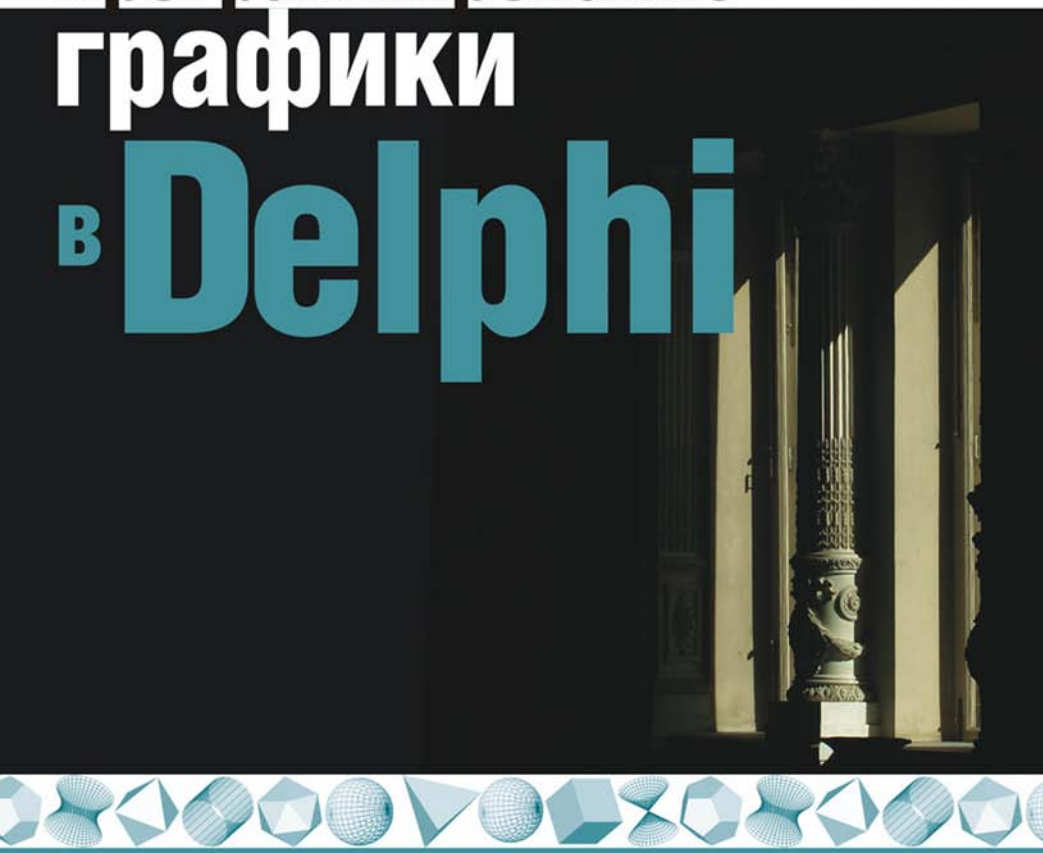
Николай Тюкачев,
Игорь Илларионов,
Виктор Хлебостроев

Программирование графики в Delphi



+CD

- Цветовые модели и форматы графических файлов
- Средства для разработки мультимедийных приложений
- Компоненты для работы с деловой графикой
- Методы построения кривых в задачах интерполяции, сглаживания, аппроксимации, методы Эрмита, Безье и B-сплайнов
- Создание растровых и векторных графических редакторов
- Работа с трехмерными изображениями
- Задачи триангуляции, проектирование и реализация ГИС

bhv

Николай Тюкачёв,
Игорь Илларионов,
Виктор Хлебостроев

Программирование графики в Delphi

Санкт-Петербург
«БХВ-Петербург»
2008

УДК 681.3.068+800.92Delphi
ББК 32.973.26-018.1
Т98

Тюкачѐв, Н. А.

Т98 Программирование графики в Delphi / Н. А. Тюкачѐв, И. В. Илларионов, В. Г. Хлебостроев. — СПб.: БХВ-Петербург, 2008. — 784 с.: ил. + CD-ROM

ISBN 978-5-9775-0253-5

Книга написана на базе курса лекций, читаемых авторами. Рассмотрены основные классы и функции среды Delphi, которые используются для создания графических и мультимедийных приложений. Описаны цветовые модели, основные форматы графических файлов, а также методы построения кривых в задачах интерполяции, сглаживания, аппроксимации, методы Эрмита, Безье и B-сплайнов. Приведены алгоритмы триангуляции поверхностей в трехмерном пространстве. На конкретных примерах показан весь процесс разработки основных типов приложений — пакетов деловой графики, работы с трехмерными объектами, растровых и векторных графических редакторов, геоинформационных систем. Каждый раздел сопровождается задачами различной сложности для самостоятельного решения. На прилагаемом компакт-диске представлено более 30 проектов, описанных в книге.

Для программистов

УДК 681.3.068+800.92Delphi
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Наталья Баскакова*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.06.08.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 63,21.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.60.953.Д.003650.04.08
от 14.04.2008 г. выдано Федеральной службой по надзору
в сфере защиты прав потребителей и благополучия человека.

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

ISBN 978-5-9775-0253-5

Оглавление

Введение

Эта книга написана на основе лекций для студентов, обучающихся по специальности "Информационные технологии", и не является введением в визуальную среду программирования Delphi. Предполагается, что читатель уже преодолел несколько первых ступеней, знаком с объектно-ориентированным программированием, со средой Delphi и умеет создавать в ней простые проекты. Это позволит нам не останавливаться на подробном описании последовательности нажатия клавиш и больше внимания уделять принципиальным вопросам разработки проектов.

Для читателей, делающих первую попытку открыть для себя мир Delphi, опубликовано большое количество прекрасных книг. Все книги этого направления можно разбить на три больших группы. Первая группа — с условным названием "Математические основы машинной графики", вторая — "Описание и использование библиотек типа OpenGL или Direct", третья — "Описание методов рисования Windows".

Из числа книг первой группы наиболее полно математические основы графики изложены в книге Д. Роджерса и Д. Адамса "Математические основы машинной графики" [98]. В ней, а также в книге Д. Роджерса "Алгоритмические основы машинной графики" [97], подробно описаны модели и алгоритмы растровой графики:

- алгоритмы Брезенхейма;
- модели освещения, тени, прозрачности, фактуры;
- двумерные и трехмерные преобразования и проекции;
- плоские кривые, пространственные кривые: кубические сплайны, кривые Безье, B-сплайны;
- поверхности, квадратичные поверхности, бикубические поверхности, поверхности Безье, B-сплайн поверхности.

Книга Л. Аммерала "Принципы программирования в машинной графике" [58] содержит меньше материала, чем у Д. Роджерса, но в ней дается введение в язык Си. Майкл Абраш в книге "Программирование графики. Таинства" [56] излагает основы графики еще под MS-DOS: алгоритмы Брезенхейма для отрезка и окружности, заполнение многоугольника, устранение ступенчатости.

В книге Е. В. Шикина и А. В. Борескова "Компьютерная графика. Динамика, реалистические изображения" [120] освещаются те же вопросы, что и у Д. Роджерса и Дж. Адамса, и, кроме того, описывается пакет 3D Studio.

Книга Е. А. Никулина [87] содержит подробное изложение геометрических и алгоритмических основ компьютерной графики — математических моделей графических элементов на плоскости и в пространстве, методов геометрических преобразований и визуализации, способов моделирования оптических объектов. Однако все рассматривающиеся в ней алгоритмы описаны словесно либо в виде блок-схем.

Список второй группы книг может быть открыт книгой Ю. В. Тихомирова "Программирование трехмерной графики в Visual C++" [105], в которой освещаются возможности применения в приложениях на языке Си библиотеки OpenGL. Подробно описаны:

- вершины, примитивы;
- спецэффекты: глубина, туман, цвет;
- геометрические преобразования;
- реалистические изображения: свойства материала, источники света, прозрачность, тени, текстура.

М. В. Краснов в книге "OpenGL в Delphi" [80] переложил для Delphi тот же материал что и Ю. В. Тихомиров, добавив описание графического редактора, использующего OpenGL.

Наиболее ярким представителем третьей группы является книга Юань Феня "Программирование графики для Windows" [125]. Ч. Петзолд в главе 4 книги "Программирование для Windows 95" [92] также описывает графические API-функции Windows, а А. Я. Архангельский в одной из глав своей книги "Программирование в Delphi 6" [60] перечисляет свойства и методы канвы. Впрочем, наиболее полно API-функции, свойства и методы канвы описаны в справочной системе Delphi.

Таким образом, между математическими основами графики и готовыми графическими библиотеками типа OpenGL/Direct, в которых эти основные алгоритмы реализованы, образовался разрыв. Назначение нашей книги — частично заполнить этот разрыв. Средством для заполнения является описание многочисленных приложений.

Мы попытались объединить различные подходы, собрав и систематизировав в *главах 1–3* весь доступный и необходимый нам справочный материал о графических возможностях Delphi, а в *главах 4–14* представив многочисленные примеры проектов. Отметим, что при разработке проектов иногда существенную помощь оказывали интернет-ресурсы. В первую очередь это относится к проблемам моделей цветов, рисования линий произвольным стилем, алгоритмам Брезенхейма.

Мы осознаем, что чтение первых трех глав достаточно утомительное занятие, но без знания (хотя бы поверхностного) возможностей графических классов сложно приступать к разработке самостоятельных графических проектов. После беглого чтения этих глав можно заняться примерами, возвращаясь, по мере необходимости, к справочному материалу.

Delphi опирается на мощный набор классов, предназначенных для работы с графикой. Значительная часть этих классов содержится в модуле `Graphics`. Описанию модуля `Graphics` посвящена *глава 2.* В ней описаны классы инструментов `TFont`, `TPen`, `TBrush` и класс канвы `TCanvas`. Там же приводится описание всех методов рисования на канве.

В *главе 2* обсуждаются также различные цветовые модели: RGB, CMY, CMYK, HSB, Lab, HSV, HLS. Поскольку все цветовые модели являются математическими,

они легко конвертируются одна в другую по простым формулам. Такие конверторы встроены во многие графические программы. Примеры процедур, которые можно использовать для перехода из одной модели в другую, приведены в *разд. 2.4*.

Можно нарисовать сколь угодно сложную картинку, вставляя в код программы вызов методов с заданными значениями координат точек на канве. Но приложения должны быть максимально независимы от конкретных данных. Программа, описанная в *разд. 2.10*, рисует различные рисунки на канве. Для реализации этой идеи нужно записать в текстовый файл данные о рисунке в специальном формате.

Метод канвы `TextOut` позволяет печатать только горизонтально расположенную строку. В *разд. 2.11* описан простой проект, позволяющий печатать строку под любым углом `Angle` в десятых долях градуса.

Из приложения, как это показано в *разд. 2.12*, можно получить доступ не только к канве компонента, имеющего опубликованное свойство `Canvas`, но и к канве любого компонента или ко всему экрану.

Среда программирования Delphi поддерживает некоторые форматы графических файлов. Во-первых, это "родной" формат Windows — формат растровых изображений `BMP` (класс `TBitmap`). Во-вторых, форматы метафайлов `EMF` и `WMF` (класс `TMetaflle`). В-третьих, формат пиктограмм `ICO` (класс `TIcon`). В четвертой версии Delphi добавлен класс, позволяющий работать со сжатыми файлами в формате `JPG` (класс `TJPEGImage`).

В *главе 3* описан абстрактный класс `TGraphic`, являющийся родительским для этих четырех видов изображений, и сами классы `TBitmap`, `TMetaflle`, `TIcon`.

В *главе 4* рассмотрены средства, которые предоставляет среда программирования Delphi для разработки мультимедийных приложений. Сначала приводятся примеры использования компонентов Animate и MediaPlayer, затем продемонстрировано, как можно использовать стандартные функции MCI и WinAPI для решения задач воспроизведения и записи звука, работы с мультимедийными файлами. Подробно рассмотрена структура wave-файла и ее применение для разработки приложений, оперирующих низкоуровневыми структурами.

Программы построения диаграмм составляют неотъемлемую часть офисных программ. Delphi включает в себя несколько специальных компонентов для работы с деловой графикой. Описание свойств и методов этих компонентов представлено в *главе 5*. Подробно рассмотрены методы и свойства компонента Chart. Показаны возможности создания приложений, как с использованием прямого обращения к методам компонента в тексте программы, так и с применением собственных средств компонента.

В *главе 6* рассмотрены вопросы моделирования изображений: преобразование системы координат, проецирование. В этой же главе обсуждаются методы построения кривых линий в задачах интерполяции, сглаживания, аппроксимации, методы Эрмита, Безье и B-сплайнов. Затронуты вопросы математического моделирования освещения и цвета.

В *главе 7* приводятся примеры проектов, использующих методы и свойства графических классов: просмотр файлов форматов BMP, ICO, WMF и JPG, вывод изображений на канву принтера, использование мультимедийных ресурсов, деформация изображений в формате BMP, реализация простейшего графического редактора, проектирование плоских схем.

В этой же главе обсуждаются способы создания мультипликации. Изменение изображений со временем возможно двумя способами. Во-первых, можно воспользоваться процедурой временной задержки Sleep из модуля SysUtils, во-вторых, воспользоваться компонентом Timer, имеющим обработчик события OnTimer. Возможности, связанные с использованием процедуры Sleep, демонстрируются на примере сортировки элементов массива, а использование компонента Timer — на примере морфинга одного предмета (стула) в другой (стол) и в проекте "Движение велосипеда".

Свойство Style определяет стиль линии, рисуемой пером. Во многих случаях узкий диапазон возможных значений этого свойства не устраивает программистов. Для применения стиля, заданного пользователем, необходимо использовать API-процедуру LineDDA, определенную в модуле Windows. Проект, описанный в *разд. 7.4*, показывает, как это реализовать.

Далее обсуждается работа с принтером, для использования которого должен быть подключен модуль Printers. С принтером можно работать, как с текстовым файлом, или как с полотном.

В многочисленных графических редакторах используются различные фильтры, изменяющие первоначальные изображения. Некоторые из этих фильтров меняют геометрию изображения. В *разд. 7.5* описывается проект, решающий задачу деформации изображения из прямоугольной области (0,0), (I2,J2) в произвольный четырехугольник (u0,v0), (u1,v1), (u2,v2), (u3,v3) двумя способами.

В *разд. 7.6* описан проект, реализующий многооконный растровый редактор типа редактора PaintBrush, который позволяет рисовать линию, отрезок прямой, прямоугольник, выделять и перемещать прямоугольник с частью изображения, вставлять изображение.

В *разд. 7.7–7.9* обсуждаются задачи проектирования плоских схем. К этому классу задач относятся задачи построения сложных диаграмм и электрических схем, структурных схем соединения компьютеров в сети и структурных схем программ, проектирования реляционных баз данных, задачи макетирования векторных карт и множество других проектов, в которых требуется установить связи между объектами, изображенными на экране в виде условных обозначений. Программы, предназначенные для проектирования плоских схем, должны, прежде всего, уметь вставлять в проект условные обозначения объектов, устанавливать связи между объектами, перемещать объекты и связи, удалять объекты и связи, менять свойства объектов.

В качестве примера решения такой задачи предлагается проект построения изображения произвольного графа с возможностями добавлять, удалять и перемещать узлы, создавать дуги, а также выбирать вид представления узлов.

Вторым примером подобной задачи является разработка проекта газификации домов. Предполагается, что проект дома готов и отсканирован в формате `BMP` или `JPG`. Программа позволяет подготовить эскизы газификации дома, трассы трубы от газопровода до дома, расставить газовое оборудование на эскизе газификации дома (есть возможность выбора газового оборудования из списка). Имеется возможность редактирования (перемещения, удаления) оборудования и труб. При построении эскиза трассы трубы от газопровода до дома можно учитывать профиль поверхности земли, расставить бетонные трубы и показать глубину их прокладки. После завершения редактирования программа позволяет распечатать подготовленные эскизы и список необходимого оборудования.

Сохранение графической информации существенно зависит от характера изображения. Для издательских систем и рекламных роликов характерно использование растровой графики, при этом в памяти хранится информация о цвете каждой точки. Такое представление информации, естественно, требует большого объема оперативной памяти и дискового пространства, а также эффективных алгоритмов сжатия. В системах автоматизированного проектирования таких, например, как AutoCAD, информация сохраняется в векторном виде. Так, в векторном представлении, для хранения отрезка прямой линии можно записать в файл координаты его начала и конца, информацию о цвете и стиле линии. Информация о плоских схемах, в том числе о графах, также хранится в векторном виде в одном нетипизированном файле.

В *главе 8* описывается векторный редактор типа редактора Corel DRAW, реализация которого сложнее растрового. Поэтому в проекте реализована только часть функциональных возможностей, а именно: построение изображений прямоугольника, эллипса, кривой Безье и фрагмента текста; перемещение, масштабирование и поворот этих объектов.

В *главе 9* продемонстрировано использование графических возможностей Delphi для решения некоторых геометрических задач. Описаны проекты построения графиков функций в двумерном и трехмерном пространствах, интерполяция функций (интерполяционный многочлен Лагранжа, метод наименьших квадратов, кубические сплайны, кривые Безье).

В *разд. 9.1* обсуждаются дополнительные возможности реализации задачи построения графика функции одной переменной. В проекте на инструментальной форме помещены компоненты, позволяющие менять размеры окна "на бумаге", рисовать или не рисовать координатную сетку и строить график функции в обычной или логарифмической системе координат.

Далее рассматривается задача построения в трехмерном пространстве поверхности, описываемой функцией $z = \mathrm{f}(x,y)$ с возможностью управления вращением системы координат указателем мыши.

На практике часто встречаются задачи построения интерполирующих и аппроксимирующих функций для заданного на плоскости множества точек (x_i,y_i), $i = 0, 1,..., n$. В *разд. 9.3* рассмотрены четыре способа решения этой задачи:

интерполяционный многочлен Лагранжа, метод наименьших квадратов, интерполяция кубическими сплайнами и интерполяция кривыми Безье.

В *разд. 9.4* демонстрируется простой проект для рисования семи трансцендентных кривых, заданных параметрически и зависящих от шести параметров.

В *разд. 9.5* рассматривается задача построения графика функции по выражению, заданному в виде строки, т. е. конструкции из переменных, констант, бинарных операций и функций.

Глава 10 посвящена задаче визуального создания сложных отчетов, которые могут включать строковые надписи, таблицы и графики. Надписи могут быть статическими или вычисляться в процессе заполнения отчета. Таблицы могут быть горизонтальными или вертикальными. Они могут содержать статические заголовки колонок и поля, вычисляемые в процессе заполнения отчета. Результатом работы генератора отчетов является набор файлов, в которых содержатся следующие данные:

- общие данные об отчете (название отчета, поля на листе бумаги, шаг сетки);
- данные о строковых надписях (положение на листе бумаги, ширина, высота, шрифт, отметка о статичности или номер функции);
- данные о таблицах (положение на листе бумаги, ширина, высота, шрифт, номер таблицы);
- данные о строках таблиц (название, номер таблицы, отметка о статичности или номер функции).

Генератор отчетов должен уметь создавать и удалять объекты (надписи и таблицы), настраивать свойства объектов, перемещать объекты, изменять общие параметры отчета.

В *главе 11* "Геометрия трехмерных тел" собраны проекты построения трехмерных изображений с учетом перспективы и тени, построения изображения трехмерного тела по трем проекциям, рисования полутонами, бинарные операции над множествами.

Первый проект — "Платоновы тела", предназначен для рисования трех Платоновых тел: тетраэдра, гексаэдра и октаэдра и состоит из двух форм: главной формы и формы для настройки параметров. В проекте реализованы следующие функции: выбор одного из тел; вращение системы координат или тела; рисование тела гранями или ребрами; изменение размера тела; изменение масштаба; окрашивание граней полутонами; рисование тени на плоскости $z = const$; изменение положения точек схода на осях OZ и OY.

Второй проект — "Квадратичные поверхности" предназначен для рисования 10 поверхностей (эллипсоида, различных типов гиперболоидов, параболоидов и цилиндров). В общем случае уравнения поверхностей, заданных полиномами второй степени, имеют вид $Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz + K = 0$. Но, после приведения этого уравнения к каноническому виду, число различных квадратичных поверхностей существенно сокращается. В этом проекте, как и в

проекте "Платоновы тела", реализовано стереографическое изображение многогранников.

В начертательной геометрии и черчении одна из основных задач состоит в том, что необходимо спроецировать трехмерное тело на координатные плоскости OX, OY, OZ. Обратная задача, обсуждаемая в третьем проекте *главы 11*, более сложна: по трем (или даже по двум) проекциям нужно восстановить объемное изображение тела. Предлагается простой алгоритм решения этой задачи. Заметим, что в общем случае задача не имеет однозначного решения.

Операции объединения, вычитания, пересечения двумерных и трехмерных тел используются при проектировании в архитектуре и машиностроении. В общем случае эти задачи являются достаточно сложными. Однако бинарная нумерация операций над множествами, используемая в четвертом проекте, позволяет сформулировать общий алгоритм для ЭВМ.

В *главе 12* сначала обсуждаются различные способы моделирования трехмерных тел. Выбор модели трехмерных тел определяет способ решения таких проблем, как выделение видимых частей объекта и назначение цвета каждому элементу объекта. Моделирование может быть реализовано разложением объекта на элементы, трехмерные тела в этом случае будут представляться объединением некоторых базовых блоков. Тип базовых объектов определяет различные методы моделирования этим способом. Самый простой способ описания тела заключается в сопоставлении с ним набора одинаковых кубов (вокселей). Можно уменьшить объем памяти, необходимой для хранения информации об объекте, если хранить информацию только о блоках, относящихся к объекту. В этом случае число элементов пропорционально площади его поверхности, т. е. пропорционально N^2. С этой целью можно использовать октарные или бинарные деревья. В пространственной геометрии объект может задаваться набором примитивов и операций над ними. Такое представление называется твердотельным моделированием. Примитивы являются "строительными блоками" объекта. Под операциями понимаются булевы операции над примитивами, а также геометрические преобразования, такие, как передвижение, поворот, изменение размеров. В отличие от ранее перечисленных моделей, поверхностное представление определяет сплошное тело неявно, путем описания ограничивающей его поверхности. Наиболее простой способ поверхностного описания тел — это описание его набором плоских граней.

В той же главе описан проект, предназначенный для визуальной работы с многогранниками. Он позволяет: читать и записывать данные; перемещать окна просмотра и тело; масштабировать изображение на любой координатной проекции; перемещать вершины на любой проекции; выделять грань и, добавляя к ней новую вершину, увеличивать число граней. Предполагается, что все грани представляют собой треугольники.

В геоинформационных системах (ГИС) существует проблема описания двумерных и трехмерных тел (многоугольников на плоскости и многогранников в пространстве), топологически связанных между собой. Это означает, что граничные вершины

и ребра многоугольников (на плоскости) или граничные вершины, ребра и грани (в трехмерном пространстве) должны быть общими для соседних тел. В этом случае изменение координат общей вершины двух соседних тел не приводит к появлению "ничейной" области между телами. В *разд. 12.3* предлагается структура данных, описывающая систему топологически связанных тел для трехмерного пространства. Стандартное описание системы тел в трехмерном пространстве выглядит следующим образом: проект состоит из массива тел-многогранников. Для каждого тела определены свои независимые массивы вершин, ребер и граней, опирающихся на вершины. В предлагаемом проекте для всех тел используются общие массивы вершин, ребер и граней.

В *главе 13* описывается работа с библиотекой OpenGL, основные команды, а также очень простой проект, демонстрирующий использование плоских примитивов OpenGL. А для демонстрации работы с трехмерными объектами предназначен второй проект "3D OpenGL". Проект позволяет рисовать многогранники модуля `DGLUT` или иные многогранники, квадратичные поверхности, заданные параметрическими уравнениями, поверхность, заданную таблично, различные фигуры, данные о которых записываются в текстовых файлах программами, использующими DirectX, и накладывать на эти поверхности текстуру из файлов BMP или JPG.

В заключение *главы 13* обсуждается проект "Редактор многогранников", в котором реализован ряд функций манипулирования простейшими многогранниками — пирамидами и параллелепипедами. Проект позволяет добавлять тела или группы тел, менять размеры выбранного тела, менять положение выбранного тела или группы тел, менять видимость тел или группы тел, изменять цвет тел, изменять цвет и положение источника света, поворачивать сцену.

Глава 14 посвящена некоторым задачам триангуляции, многочисленные варианты которых возникают при разработке геоинформационных систем. В частности, рассматриваются следующие задачи триангуляции: триангуляция по заданным точкам; триангуляция слоев; сглаживание поверхности; изолинии.

Здесь же рассматривается еще одна задача, связанная с триангуляцией — построение триангуляции невыпуклого многоугольника. Эта задача порождает определенные проблемы при рисовании, так как библиотека OpenGL, например, не может изображать невыпуклые многоугольники.

В заключение главы рассматривается часто возникающая задача построения изолиний и рисования топографической карты поверхности. Эта поверхность может быть дневной поверхностью земли или поверхностью слоя породы или значением объемных геофизических измерений, сделанных на определенной глубине.

В тексте приведены многочисленные примеры программ и процедур. В книге приведены условия задач, решение которых достаточно просто реализуется в среде Delphi.

Авторы с благодарностью примут все замечания и пожелания в адрес предлагаемой книги (**E-mail: nik@cs.vsu.ru**).

Глава 1

Рисование в Delphi

1.1. Моделирование цветов

Мы видим мир цветным, и это одна из самых больших биологических загадок. Человеческий глаз может воспринимать цветовые волны с длинами от 380 нм до 780 нм. Это лишь незначительный диапазон спектра электромагнитных волн. Зрительные пигменты глаза состоят из колбочек трех типов, чувствительных к синему, зеленому и красному цвету. Эффективность поглощения световых волн существенно различается для различных типов колбочек. Особенно хорошо воспринимается зеленый свет, красный свет воспринимается несколько хуже, а чувствительность глаза к синему свету еще ниже. Многочисленные психологические тесты позволяют считать, что яркость можно вычислить по формуле [124]:

Яркость= 0,59×Зеленый + 0,3×Красный + 0,11×Синий

Цвет представляет собой индивидуальное ощущение, не позволяющее судить о его спектральном составе. Для обработки изображений на современной технике такая субъективность нежелательна. Именно для цели обработки изображений были разработаны математические методы точного описания цвета, называемые цветовыми моделями. Одна из них носит название `RGB` (`Red`, `Green`, `Blue`). В рамках этой модели предполагается получение цветов смешением красного, зеленого и синего цветов. Смесь синего и зеленого, например, дает голубой цвет, а смесь красного и синего — пурпурный.

Операционные системы, такие как Windows, позволяют устанавливать различные графические режимы:

- 16-цветный режим;
- 256-цветный режим, при котором для хранения цвета каждой точки (пиксела) выделяется один байт;
- режим `High Color`, при котором для хранения цвета каждой точки выделяется два байта, что позволяет использовать до 65000 цветов;
- режим `True Color`, при котором для хранения каждого пиксела выделяется четыре байта (32 бита). Чаще всего из этих 32 битов используется 24, что позволяет изменять доли красного, синего и зеленого цветов в диапазоне от 0 до 255.

Математически систему RGB можно представить в виде куба, каждая точка которого однозначно определяется координатами R, G и B. Так как в системе RGB цвета определяются смешением основных цветов, то она особенно удобна для устройств, излучающих цветовые волны (видеомонитор, цветной телевизор).

Цветовая система RGB очень проста, но при ее использовании возникают две проблемы. Первая — зависимость от аппаратуры, вторая — невозможность получить все цвета смешением красного, зеленого и синего цветов.

Другой, часто используемой цветовой системой, является модель CMYK. Название этой модели образовано из первых букв названий цветов Cyan, Magenta, Yellow (голубой, пурпурный, желтый) и последней буквы слова blacK (черный) (используется буква K, так как буква B уже задействована в названии цветовой модели RGB). Данная модель предназначена для описания цвета отражающих поверхностей. Именно эта модель служит теоретической основой цветной печати. В отличие от модели RGB, цвета в модели CMYK получаются не аддитивно (суммированием), а субтрактивно (вычитанием). Например, поверхность голубого цвета отражает синий и зеленый, но поглощает красный, пурпурный поглощает зеленый, а желтый поглощает синий цвет.

Применяемые на практике цветные краски далеко не идеальны, и смешением всех красок, как правило, не удается получить черный цвет. Поэтому для повышения контрастности применяется чисто черный краситель.

1.2. Полотно компонентов

Рисовать в Delphi проще, чем на бумаге. При разработке проекта в вашем распоряжении находятся полотно (свойство Canvas), карандаш (свойство Pen), кисть (свойство Brush) и некоторое количество примитивов (линий, прямоугольников, эллипсов и т. д.). Правда, опубликованным свойством Canvas обладают далеко не все компоненты. В частности, этим свойством обладают компоненты форма (класс TForm), таблица (класс TStringGrid), растровые изображения (класс TImage), принтер (класс TPrinter). У карандаша и кисти можно менять цвет (Color) и стиль (Style). Этот набор инструментов позволяет создавать достаточно сложные рисунки математического и инженерного содержания. Кроме того, Delphi позволяет использовать многие ресурсы Windows: графические файлы, фильмы и звуковые файлы.

Полотно — это прямоугольная сетка, состоящая из маленьких квадратов, называемых пикселами (свойство Pixels[X,Y]: TColor). Каждый пиксел имеет свой номер, точнее два номера. Первый номер указывает на горизонтальное расположение пиксела, а второй — на вертикальное. Левый верхний пиксел полотна имеет координаты 0,0 — Pixels[0,0]. Общее количество пикселов по горизонтали доступно через свойство Width, а по вертикали — через свойство Height. Каждый пиксел может быть закрашен любым доступным для Windows цветом. Рисование пиксела на по-

лотне происходит в тот момент, когда мы присваиваем элементу массива `Pixels` номер цвета. Например, присваивание:

```
Image1.Canvas.Pixels[100,100]:= clRed
```

приведет к рисованию красной точки с координатами `100,100`. Мы можем узнать номер цвета любого пиксела обратным действием:

```
Color:= Image1.Canvas.Pixels[100,100].
```

Класс цвета точки `TColor` определен как `longint`:

```
TColor = - $80000000 ..$7FFFFFFF.
```

Четыре байта переменных этого типа содержат информацию о долях синего (B), зеленого (G) и красного (R) цветов и устроены следующим образом: `$00BBGGRR`. Доля каждого цвета может меняться от 0 до 255. Поэтому для рисования красной точки, например, мы должны выбрать цвет с номером `$0000FF`. В Delphi определен набор констант для цветов. Список этих констант можно увидеть в инспекторе объектов или в модуле `Grahics`.

Впрочем, обращение к точкам полотна через двумерный массив `Pixels[X,Y]` — это самый медленный способ рисования. Для более быстрого рисования используют специальные методы канвы `Canvas`: например, `LineTo` — линии, `Arc` — дуги, `RectAngle` — прямоугольники, `TextOut` — вывод текста и т. д.

1.3. Пример использования графики

Для демонстрации возможностей использования графики, рассмотрим простой проект, в котором на полотне компонента `Image1` будем рисовать кривые Лиссажу (рис. 1.1). Проект приведен на компакт-диске (полное содержание которого приведено в *прил. 2*) в папке **Примеры | Глава 1 | Кривые Лиссажу**.

Все помнят из школьного курса физики, что если на вход X и вход Y осциллографа подать синусоидальные сигналы, то на экране появится отрезок прямой, но при некотором сдвиге фаз отрезок расслоится и превратится в эллипс, который превращается в окружность при сдвиге фаз π/2 и одинаковых амплитудах. Это обстоятельство используется для измерения сдвига фаз в любительских радиоизмерениях. В общем виде кривые Лиссажу задаются параметрическими уравнениями:

$$x = A \times \sin(w_x \times t + w_1),$$

$$y = A \times \sin(w_y \times t + w_2)$$

и зависят от четырех параметров: двух частот w_x, w_y и двух фазовых смещений w_1 и w_2. В проекте функции, определяющие кривые Лиссажу, обозначены как `Fx` и `Fy` (листинг 1.1).

Листинг 1.1. Функции, определяющие кривые Лиссажу

```
function TForm1.Fx(t: real): real;
begin
  Fx:=sin(wx*t+w1);
end;
function TForm1.Fy(t: real): real;
begin
  Fy:=sin(wy*t+w2);
end;
```

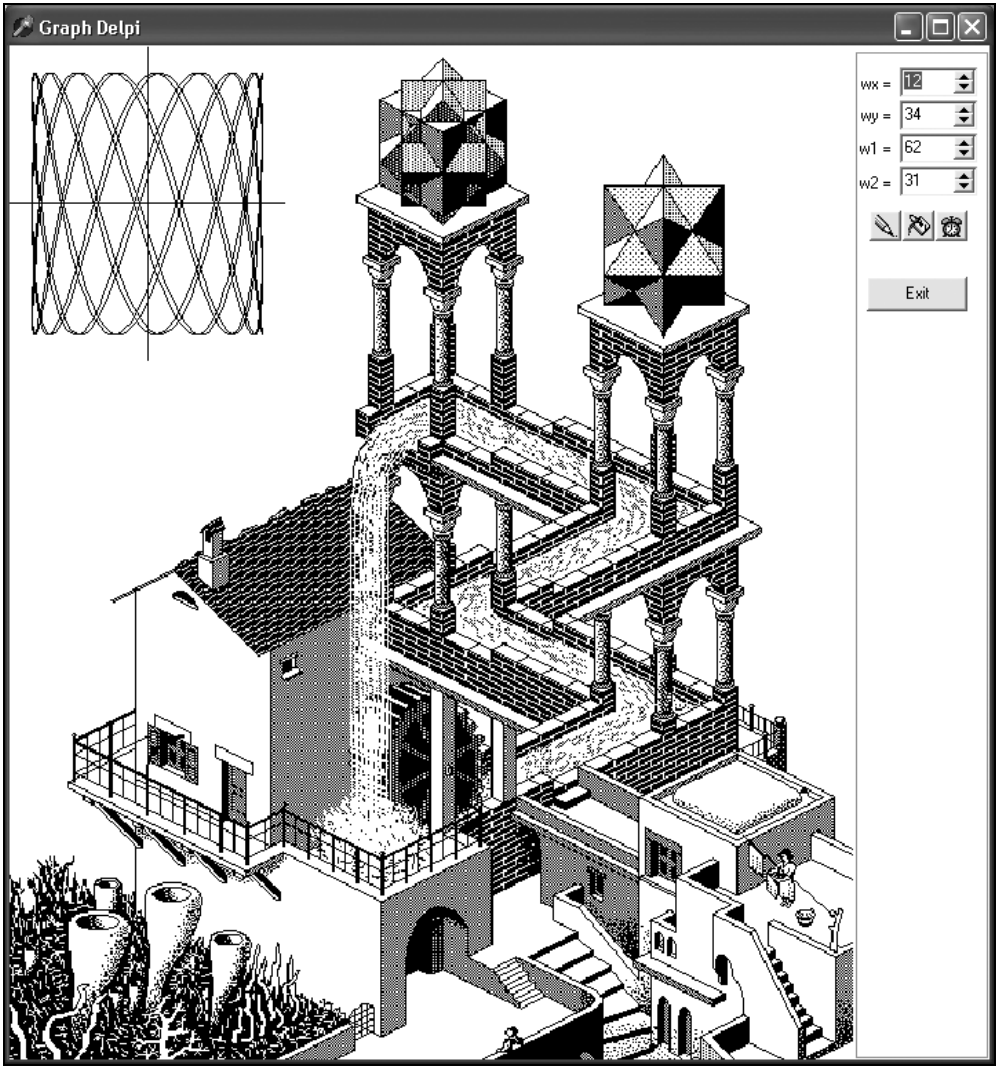


Рис. 1.1. Проект "Кривые Лиссажу" с копией картины Мориса Эшера

Для изменения параметров `wx`, `wy`, `w1` и `w2` во время выполнения проекта поместим на единственную форму (рис. 1.1) четыре компонента `TSpinEdit`. Для более плавного изменения значений параметров используем коэффициент масштабирования равный 1/10. Например:

```
wx:=SpinEditWx.Value/10.
```

Прежде чем рисовать график функции на бумаге, мы выбираем на ней окно рисования, задавая интервалы изменения переменных `x∈[x1,x2]` и `y∈[y1,y2]`. При переходе от "бумажного" окна `(x1,y1),(x2,y2)` к окну на экране `(I1,J1),(I2,J2)` придется масштабировать координаты точки с помощью функций `II(x)` и `JJ(y)` (листинг 1.2).

Листинг 1.2. Функции масштабирования

```
function TForm1.II(x: real): integer;
// функция масштабирования по оси OX
begin
  II:=I1+Trunc((x-X1)*(I2-I1)/(X2-X1));
end;

function TForm1.JJ(y: real): integer;
// функция масштабирования по оси OY
begin
  JJ:=J2+Trunc((y-Y1)*(J1-J2)/(Y2-Y1));
end;
```

Размеры окна на экране и на бумаге зададим при запуске проекта (листинг 1.3).

Листинг 1.3. Инициализация переменных

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Bitmap:=TBitmap.Create;
  Bitmap.Transparent := true;       //false;
  Bitmap.TransparentMode := tmAuto; //tmFixed;
  BitMap.LoadFromFile('Waterfal.bmp');
  BitMap.TransParentColor := clWhite;

  n:=200;
  X1:=-1.2; X2:=1.2; Y1:=-1.2; Y2:=1.2;
  // Задание окна на экране
  I1:=0; J1:=0; I2:=2*Width div 5; J2:=2*Height div 5;
  // вычисление шага по оси OX
  h:=2*Pi/n;
  DrawGraphic;
end;
```

Более подробное обсуждение функций масштабирования отложим до следующих глав, а пока займемся рассмотрением основного для данного проекта метода `DrawGraphic` (листинг 1.4). Первые четыре строчки метода `DrawGraphic` присваивают значения компонентов `SpinEdit` соответствующим параметрам функций. Все остальные события происходят на полотне компонента `Image1`. Сначала назначается белый цвет пера `Pen.Color:=clWhite` и кисти `Brush.Color:= clWhite` и вызовом метода `RectAngle(0,0,Width,Height)` рисуется белый прямоугольник размером во все полотно, т. е. полотно `Image1` очищается.

Листинг 1.4. Рисование графика

```
procedure TForm1.DrawGraphic;
var i: integer;
    t: real;
begin
  wx:=SpinEditWx.Value/10;
  wy:=SpinEditWy.Value/10;
  w1:=SpinEditW1.Value/10;
  w2:=SpinEditW2.Value/10;
  with Canvas do begin
    RectAngle(0,0,Width,Height);
    FillRect(Rect(0,0,Width,Height));
    StretchDraw(Rect(0,0,W,Height),BitMap);
    // Построение осей координат
    MoveTo(II(0),JJ(Y1)); LineTo(II(0),JJ(Y2));
    MoveTo(II(x1),JJ(0)); LineTo(II(x2),JJ(0));
    // Построение графика функции отрезками
    t:=0; x:=Fx(t); y:=Fy(t); MoveTo(II(x),JJ(y));
    for i:=1 to 5*n do begin
      t:=t+h; x:=Fx(t);
      y:=Fy(t); LineTo(II(x),JJ(y));
    end;
  end;
end;
```

Затем, изменив цвет карандаша на черный `Pen.Color:=clBlack`, рисуем оси координат — прямые линии от точки `(x1,0)` до `(x2,0)` и от точки `(0,y1)` до `(0,y2)`, масштабируя "бумажные" координаты в экранные с помощью функций `II(x)` и `JJ(y)`. Прорисовку отрезка прямой производим следующим образом: сначала методом полотна `MoveTo(x1,y1)` переводим экранный указатель в точку `(x1,y1)`, потом методом `LineTo(x2,y2)` перемещаем его в точку `(x2,y2)`.

Переходим теперь к рисованию графика. При активизации формы был вычислен шаг изменения `h:=2*Pi/n` параметра `t`, который на периоде `2*Pi` принимает `n` значений. Перед началом рисования параметру `t` присвоим значение `0` и переместим указатель в точку `(x,y)`.

```
t:=0; x:=Fx(t); y:=Fy(t); MoveTo(II(x),JJ(y));
```

Будем увеличивать `t` на `h` в цикле и рисовать отрезок до следующей точки.

```
t:=t+h; x:=Fx(t); y:=Fy(t); LineTo(II(x),JJ(y));
```

Для компонента `SpinEditWx` назначим обработчик события, который при изменении свойства `SpinEditWx.Value` перерисует график (листинг 1.5).

Листинг 1.5. Событие при изменении *SpinEditWx*

```
procedure TForm1.SpinEditWxChange(Sender: TObject);
begin
  DrawGraphic;
end;
```

Эту же процедуру используем в качестве обработчика события `onChange` для остальных компонентов.

Проект для рисования графика функции готов. Осталось нажать клавишу <F9> для запуска проекта на выполнение.

Усложним наш проект, введя возможность изменения цвета пера и кисти. Для этого поставим на форму две кнопки `SpeedButton1` и `SpeedButton2` и используем метод `ColorDialog1`, вызывающий диалоговое окно **Цвет** для выбора цвета. При нажатии кнопки `SpeedButton1` будем вызывать процедуру обработки события `SpeedButton1Click` (листинг 1.6).

Листинг 1.6. Изменение цвета кисти

```
procedure TForm1.SpeedButton1Click(Sender: TObject);
begin
  if ColorDialog1.Execute then begin
    Canvas.Brush.Color:=ColorDialog1.Color;
    DrawGraphic;
  end;
end;
```

После выбора в диалоговом окне **Цвет** нового цвета, цвет кисти изменится `Image1.Canvas.Brush.Color:=ColorDialog1.Color` и будет перерисован график вызовом метода `DrawGraphic`.

При нажатии кнопки `SpeedButton2` тот же метод `ColorDialog1` используется для изменения цвета пера (листинг 1.7).

Листинг 1.7. Изменение цвета пера

```
procedure TForm1.SpeedButton2Click(Sender: TObject);
begin
  if ColorDialog1.Execute then begin
    Canvas.Pen.Color:=ColorDialog1.Color;
    DrawGraphic;
  end;
end;
```

Поставим на форму компонент Timer1 и две кнопки SpeedButton3 и SpeedButton4. Единственное событие счетчика времени Timer1Timer(Sender) вызывается через интервал времени, определенный свойством Timer1.Interval, если Timer1.Enabled=true. До начала выполнения проекта зададим значение свойства Timer1.Enabled=false, а для кнопки SpeedButton3 определим событие при щелчке кнопки мыши (листинг 1.8).

Листинг 1.8. Подключение таймера

```
procedure TForm1.SpeedButton3Click(Sender: TObject);
begin
  Timer1.Enabled:=not Timer1.Enabled;
end;
```

При нажатии кнопки SpeedButton3 будет запущен механизм таймера, который через 1000 мс будет вызывать обработчик события, который случайным образом меняет значения SpinEdit и цвета кисти и пера (листинг 1.9).

Листинг 1.9. Событие, контролируемое таймером

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  SpinEditWx.Value:=Random(100);
  SpinEditWy.Value:=Random(100);
  SpinEditW1.Value:=Random(100);
  SpinEditW2.Value:=Random(100);
end;
```

В этой процедуре параметры линии и цвета меняются случайным образом.

Для кнопки SpeedButton4 назначим событие выключения таймера (листинг 1.10).

Листинг 1.10. Выключение таймера

```
procedure TForm1.SpeedButton4Click(Sender: TObject);
begin
  Timer1.Enabled:=false;
end;
```

Введем еще одно изменение в проект. На полотно формы выведем изображение из графического файла Waterfal.BMP. Нам потребуется переменная Bitmap: TBitmap, которую инициализируем при запуске проекта в процедуре FormCreate(Sender).

```
Bitmap:=TBitmap.Create;
Bitmap.Transparent := true; //false;
Bitmap.TransparentMode := tmAuto; //tmFixed;
BitMap.LoadFromFile('Waterfal.bmp');
BitMap.TransParentColor := clWhite;
```

Вносим изменения: в этой же процедуре назначаем значение свойства прозрачности изображения `Bitmap.Transparent := true`. Это означает, что точки одного из цветов (в нашем случае `BitMap.TransParentColor := clWhite`) не будут выводиться. Методом `BitMap.LoadFromFile('Waterfal.bmp')` загрузим изображение из файла `Waterfal.BMP` в `BitMap`.

Осталось ввести последнее изменение: в процедуру рисования `DrawGraphic` добавить строчку

```
Canvas.StretchDraw(Rect(0,0,W,Height),BitMap); ,
```

после чего изображение из файла `Waterfal.BMP`, ранее загруженное в `BitMap` методом `LoadFromFile`, переносится в прямоугольник `Rect(0,0,W,Height)` полотна формы.

1.4. Мультимедийные ресурсы Windows

В Delphi включен компонент `TMediaPlayer`, позволяющий легко получить доступ к мультимедийным ресурсам Windows. Этот компонент может воспроизводить аудио- и видеофайлы в форматах WAV, MIDI и AVI. Для нормальной работы компонента `TMediaPlayer` необходимы специальные драйверы и апплеты, но в среде Windows они, обычно, инсталлируются автоматически.

Для создания приложения, реализующего доступ к мультимедийным ресурсам Windows, достаточно разместить на форме (рис. 1.2) компонент `TMediaPlayer` и создать обработчик события для нажатия кнопки, запускающей этот компонент (листинг 1.11).

Листинг 1.11. Воспроизведение аудио- и видеофайлов

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  MediaPlayer1.Close;
  if OpenDialog1.Execute then begin
    MediaPlayer1.FileName:=OpenDialog1.FileName;
    MediaPlayer1.Display := Panel1;
    MediaPlayer1.Open;
    MediaPlayer1.Play;
  end;
end;
```

Если работа метода `OpenDialog1` заканчивается выбором AVI-файла, то начинается воспроизведение этого файла.

Воспроизведение звуковых файлов возможно, естественно, при наличии звуковых колонок и звуковой карты.

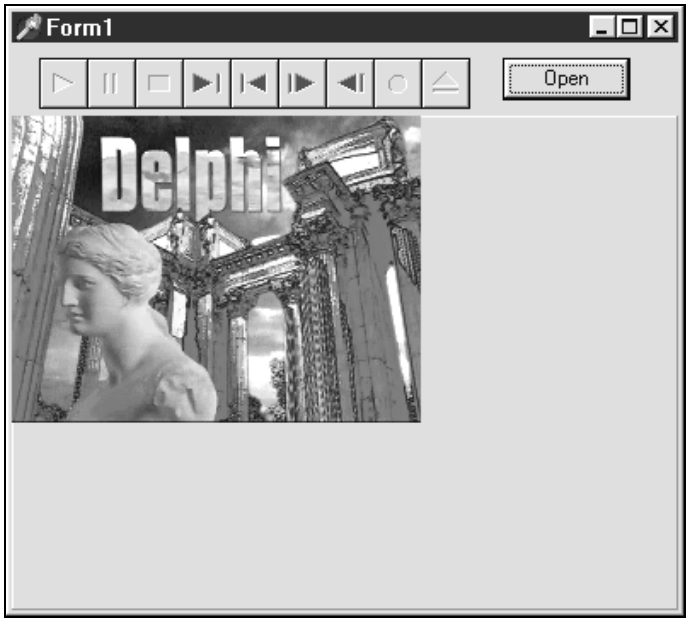


Рис. 1.2. Воспроизведение AVI-файла

На этом мы заканчиваем краткий обзор графических возможностей и переходим к более последовательному изучению графических структур Delphi.

Глава 2

Модуль *Graphics* и специальные приемы рисования

2.1. Структура классов

Основным источником информации о графических классах Delphi, их методах и свойствах является справочная система Delphi. Однако часть информации можно найти, например, в книгах А. Я. Архангельского [60], П. Дарахвелидзе [70].

В приложениях, разработанных в среде Delphi, рисование возможно только на компонентах, обладающих свойством `Canvas` (холстом, канвой, полотном). В частности, этим свойством обладают компоненты `TForm`, `TStringGrid`, `TImage`, `TPrinter`. Родительским для класса `TCanvas` и для всех остальных графических классов является класс `TPersistent`.

В листинге 2.1 представлена структура классов и их основных свойств, используемых при рисовании. Потомками класса `TPersistent` являются:

- класс `TGraphicsObject`, порождающий (инкапсулирующий) классы инструментов;
- класс `TCanvas`, содержащий инструменты и методы рисования;
- класс `TGraphic`, порождающий четыре класса изображений со своим форматом файлов (`TBitmap`, `TIcon`, `Tmetafile`, `TJPEGImage`);
- класс `TPicture`, надстройка над `TGraphic`, точнее, над его потомками. Он содержит поле `Graphic`, которое может содержать `TBitmap`, `Ticon`, `TMetafile` или `TJPEGImage`.

Листинг 2.1 достаточно велик, но мы решили включить его в настоящий раздел для того, чтобы более отчетливо отобразить иерархию классов и перечислить основные свойства объектов.

Листинг 2.1. Структура графических классов *TPersistent*

```
TGraphicsObject
    Tfont                    // класс шрифта
    PixelsPerInch: Integer   // число точек на дюйм
```

```
  Color: TColor           // цвет шрифта
  Height: Integer         // высота шрифта
  Name: TFontName         // имя шрифта
  Pitch: TfontPitch       // ширина символов шрифта
  Size: Integer           // размер
  Style: TFontStyles      // стиль
TPen                      // класс карандаша
  Color: TColor           // цвет пера
  Mode: TPenMode          // режим рисования линии
  Style: TPenStyle        // стиль линии
  Width: Integer          // толщина линии
TBrush                    // класс кисти
  Bitmap: TBitmap         // Bitmap для кисти
  Handle: HBrush          // указатель кисти
  Color: TColor           // цвет кисти
  Style: TBrushStyle      // стиль заливки
TCanvas                   // класс канвы
  ClipRect: TRect         // область отсечения канвы
  Handle: HDC             // указатель канвы
  PenPos: TPoint          // позиция указателя пера
  Pixels[X, Y: Integer]: TColor // массив пикселов канвы
  OnChange: TnotifyEvent  // событие после изменения
  OnChanging: TNotifyEvent// событие перед изменением
  Brush: TBrush           // свойство кисть
  CopyMode: TCopyMode     // режим копирования
  Font: TFont             // свойство шрифт
  Pen: TPen               // свойство перо
TControlCanvas            // родительский класс для канвы
  TBitmapCanvas           // канва класса TBitmap
  TPrinterCanvas          // канва принтера
  TMetafileCanvas         // канва метафайла
TGraphic
  TMetafile               // класс метафайлов
    CreatedBy: String     // имя автора
    Description: String   // описание файла
    Enhanced: Boolean     // способ сохранения на диске
    Handle: HENHMETAFILE  // указатель на экземпляр
    MMWidth: Integer      // ширина в сотых долях миллиметров
    MMHeight: Integer     // высота в сотых долях миллиметров
    Inch: Word            // число единиц на дюйм
  TBitmap                 // класс, содержащий изображение
    Canvas: TCanvas       // свойство полотно
    Handle: HBITMAP       // указатель на экземпляр
    HandleType: TBitmapHandleType // признак аппаратной зависимости
    Height                // высота в пикселах
```

```
    IgnorePalette: Boolean// игнорировать палитру
    MaskHandle: HBITMAP   // указатель на битовый массив GDI
    Monochrome: Boolean   // монохромность
    Palette               // палитра
    PixelFormat           // формат пикселов
    ScanLine[Row: Integer]: Pointer // указатель к линиям пикселов
    TransparentColor: TColor // прозрачный цвет
    TransparentMode: TTransparentMode // режим прозрачности
    Width                 // ширина канвы
    Modified              // модифицированность
    PaletteModified       // модифицированность палитры
    Transparent           // прозрачность
  TJPEGImage              // класс для файлов формата JPG
    CompressionQuality  // отношение между качеством и размером файла
    Empty                 // содержит ли JPG-графику
    GrayScale             // оттенки серого цвета
    Height                // высота в пикселах
    Palette               // палитра
    Performance           // отношение между качеством и скоростью
                          // декомпрессии
    PixelFormat: TPixelFormat // формат пиксела
    ProgressiveDisplay    // коэффициент уменьшения изображения
    ProgressiveEncoding   // показывает, уменьшенное ли изображение
    Scale                 // определяет размер изображения
    Smoothing             // сглаживание краев
    Width                 // ширина
    Modified              // модифицированность
    PaletteModified       // модифицированность палитры
    Transparent           // прозрачность
  TIcon                   // класс для файлов формата ICO
  Handle: HICON           // указатель на экземпляр класса
  TPicture                // класс изображений
    Bitmap: TBitmap       // свойство TBitmap
    Graphic: TGraphic     // свойство TGraphic
    PictureAdapter: IChangeNotifier // интерфейс OLE
    Height: Integer       // высота
    Icon: TIcon           // свойство типа TIco
    Metafile: TMetafile   // свойство типа метафайл
    Width: Integer        // ширина
    OnChange: TNotifyEvent// событие после изменения
    OnProgress: TprogressEvent // событие в процессе изменения
```

Для рисования используются методы класса `TCanvas` и три класса инструментов: `TFont` (шрифты), `TPen` (карандаш, перо), `TBrush` (кисть).

Абстрактный класс TGraphic является родительским для четырех видов изображений, соответствующих четырем форматам графических файлов: пиктограммы — формат ICO (класс TIcon), метафайлы — форматы EMF, WMF (класс TMetaflle), растровые изображения — формат BMP (класс TBitmap) и сжатые фотоизображения — формат JPG (класс TJPEGImage). Из этих четырех графических классов только TBitmap обладает опубликованным свойством TCanvas.

Эти классы, постоянные и некоторые функции описаны в модуле Graphics.

2.2. Цвет

Класс цвета точки TColor определен как длинное целое longint:

```
Tcolor = -$80000000..$7FFFFFFF.
```

Четыре байта переменных этого типа содержат информацию о долях синего B, зеленого G и красного R цветов и устроены следующим образом: $00BBGGRR, т. е. цвет можно вычислить через доли R, G, B по формуле:

$$Color = 256^2 \times B + 256 \times G + R,$$

или, используя битовую операцию "сдвиг налево" shl,

$$Color = B\ shl\ 16 + G\ shl\ 8 + R.$$

В модуле Graphics приведены некоторые константы для цветов (листинг 2.2).

Листинг 2.2. Предопределенные константы для цветов

```
Const
       clBlack = TColor($000000);
       clMaroon = TColor($000080);
       clGreen = TColor($008000);
       clOlive = TColor($008080);
       clNavy = TColor($800000);
       clPurple = TColor($800080);
       clTeal = TColor($808000);
       clGray = TColor($808080);
       clSilver = TColor($C0C0C0);
       clRed = TColor($0000FF);
       clLime = TColor($00FF00);
       clYellow = TColor($00FFFF);
       clBlue = TColor($FF0000);
       clFuchsia = TColor($FF00FF);
       clAqua = TColor($FFFF00);
       clLtGray = TColor($C0C0C0);
```

```
clDkGray = TColor($808080);
clWhite = TColor($FFFFFF);
clNone = TColor($1FFFFFFF);
clDefault = TColor($20000000);
```

В тексте программы цвет может быть выбран с помощью стандартного метода `TColorDialog`, вызываемого методом `Execute` (листинг 2.3). Метод `TColorDialog` отображает диалоговое окно **Цвет**, используемое для выбора цвета.

Листинг 2.3. Вызов компонента *TColorDialog*

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  if ColorDialog1.Execute then begin
    Color:= ColorDialog1.Color;
    …
  end;
end;
```

Внешний вид диалогового окна **Цвет** представлен на рис. 2.1.

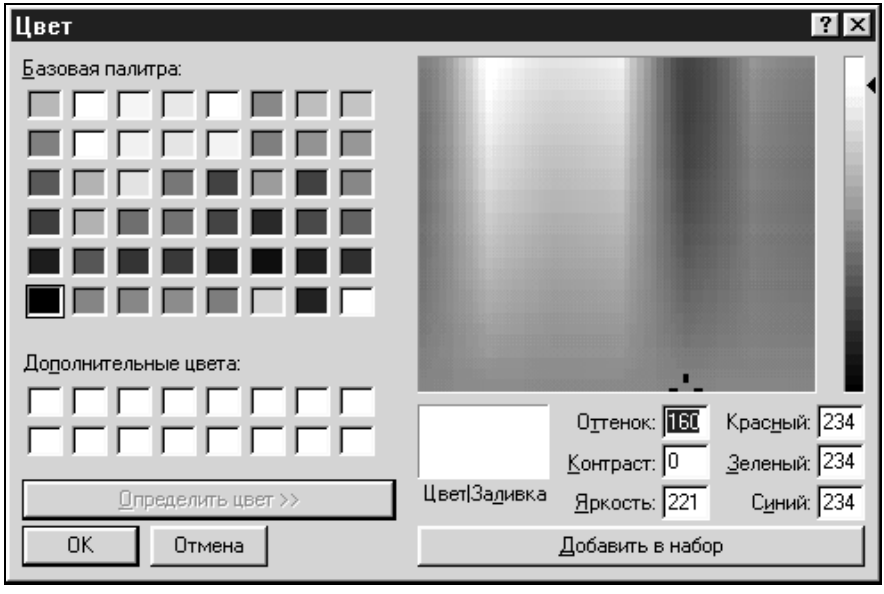


Рис. 2.1. Диалоговое окно **Цвет**

Класс `TgraphicsObject` порождает три класса инструментов: шрифт, перо и кисть. Обсуждение этих классов отложим до *разд. 2.5*, а пока рассмотрим математические методы точного описания цвета, называемые цветовыми моделями.

2.3. Цветовые модели

Цвет — это форма световой энергии, передаваемая в виде волн. На восприятие конкретного цвета влияют следующие факторы:

- источник света;
- информация об окружающих предметах;
- свойства глаз.

Способы образования цвета в природе:

- источники света (солнце, лампочка и т. д.) излучают свет различных длин волн спектра. Этот свет воспринимается глазом как цветной;
- свет отражается и поглощается, попадая на поверхность несветящихся предметов. Отраженное излучение воспринимается глазом как окраска предметов.

Как уже говорилось в *главе 1*, для описания излучаемого и отраженного цвета используются разные математические модели. Их называют цветовыми моделями. В каждой модели определенный диапазон цветов представляют в виде трехмерного пространства. В этом пространстве каждый цвет существует в виде набора числовых координат.

Большинство цветовых моделей аппаратно-зависимы, например, RGB и CMYK. Примером аппаратно-независимой модели является модель Lab. В большинстве графических пакетов, например, в Photoshop можно преобразовывать изображение из одной цветовой модели в другую.

Перечислим основные цветовые модели:

- RGB (Red, Green, Blue) — красный, зеленый и синий цвета;
- CMY (Cyan, Magenta, Yellow) — голубой, пурпурный и желтый цвета;
- CMYK (Cyan, Magenta, Yellow, blacK) — голубой, пурпурный, желтый и черный цвета;
- HSB (Hue, Saturation, Brightness) — тон, насыщенность, яркость;
- Lab (модель, основанная на восприятии изображения глазом человека);
- HSV (Hue, Saturation, Value) — тон, насыщенность, значение;
- HLS (Hue, Lightness, Saturation) — тон, яркость, насыщенность;
- HSI (Hue, Saturation, Intensity) — тон, насыщенность, интенсивность;

другие.

2.3.1. Модель RGB

Модель RGB (Red, Green, Blue) описывает излучаемые мониторами цвета и образована из трех базовых цветов: красного (`red`), зеленого (`green`) и синего (`blue`) (рис. 2.2). Все остальные цвета образуются смешиванием этих трех основных в разных пропорци-

ях. При смешении основных цветов результирующий цвет будет светлее составляющих. Модель является аддитивной и аппаратно-зависимой, так как значения базовых цветов определяются качеством примененного в мониторе люминофора.

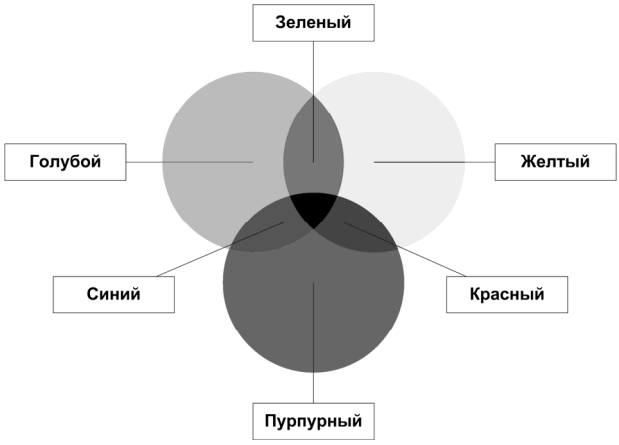


Рис. 2.2. Модель RGB

Как уже упоминалось в *главе 1*, в модели RGB информация о долях синего (B), зеленого (G) и красного (R) в цвете пиксела хранится в четырехбайтовых переменных, устроенных как `$00BBGGRR`. Доля каждого цвета может меняться от 0 до 255. В табл. 2.1 приведены значения некоторых цветов в числовой модели RGB.

Таблица 2.1. *Значения некоторых цветов в числовой модели RGB*

Цвет	R	G	B
Красный (red)	255	0	0
Зеленый (green)	0	255	0
Синий (blue)	0	0	255
Фуксин (magenta)	255	0	255
Голубой (cyan)	0	255	255
Желтый (yellow)	255	255	0
Белый (white)	255	255	255
Черный (black)	0	0	0

2.3.2. Модель CMY

В субтрактивной аппаратно-зависимой модели CMY (Cyan, Magenta, Yellow) основные цвета образуются путем вычитания из белого цветов основных аддитивных цветов модели RGB (рис. 2.3).

Основные цвета этой модели:

- голубой $FFFF00;
- фуксин (пурпурный) $FF00FF;
- желтый $00FFFF.

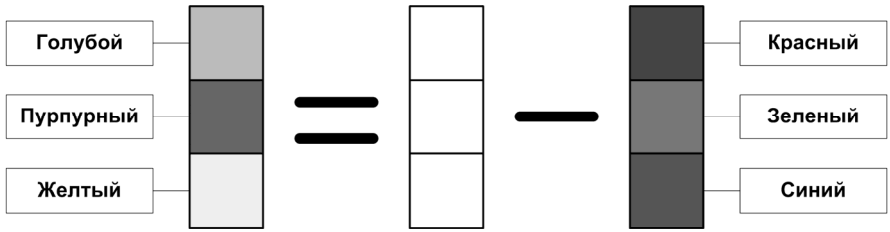


Рис. 2.3. Модель CMY=White-RGB

Эти цвета могут быть легко воспроизведены полиграфическими машинами. При смешении двух субтрактивных цветов результат затемняется (рис. 2.4). При нулевом значении всех компонентов образуется белый цвет. Эта модель представляет отраженный цвет.

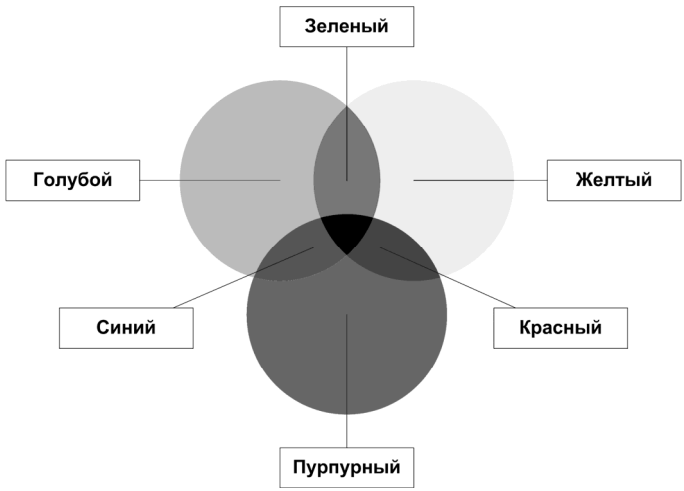


Рис. 2.4. Модель CMY

2.3.3. Модель CMYK

Модель CMYK (Cyan, Magenta, Yellow, blacK — голубой, пурпурный, желтый и черный цвета) является улучшенным вариантом модели CMY с включением чет-

вертого — черного цвета. Типографские краски имеют примеси, их цвет не совпадает в точности с теоретически рассчитанным голубым, желтым и пурпурным цветом. Особенно трудно получить из этих красок черный цвет. Поэтому в модели CMYK к трем основным краскам добавляют черный цвет. Черный цвет зашифрован как символ K (blacK), так как символ B уже задействован в цветовой модели RGB. Модель также является аппаратно-зависимой.

2.3.4. Модели HSB и HSV

Аппаратно-зависимая модель HSB (Hue, Saturation, Brightness — тон, насыщенность, яркость) так же, как и эквивалентная ей модель HSV (Hue, Saturation, Value — тон, насыщенность, значение), предложена в 1978 году и построена на основе субъективного восприятия цвета человеком. Эта модель основана на цветах модели RGB, но любой цвет в ней определяется своим цветом (тоном), насыщенностью (то есть добавлением к нему белой краски) и яркостью (то есть добавлением к нему черной краски). Фактически любой цвет получается из спектрального добавлением серой краски. Эта модель аппаратно-зависима и не соответствует восприятию человеческого глаза, так как глаз воспринимает спектральные цвета как цвета с разной яркостью (синий кажется более темным, чем красный), а в модели HSB им всем приписывается яркость 100%.

На рис. 2.5 величина H определяет тон (соответствующую частоту электромагнитных волн) и принимает значение от 0 до 360 градусов. Величина V — значение, изменяется в пределах от 0 до 1. Эквивалентная ей величина B — яркость, определяющая уровень белого света, изменяется в пределах от 0 до 100%. Величины V или B соответствуют высоте конуса. Величина S определяет насыщенность цвета. Ее значение соответствует радиусу конуса. Рисунок 2.6 представляет собой сечение конуса из рис. 2.5 (цветовой круг) при значениях S=1 и V=1.

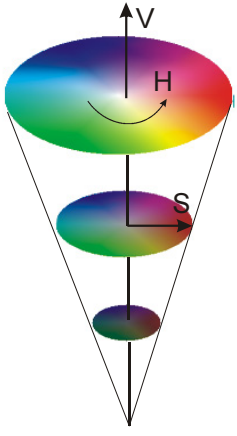


Рис. 2.5. Модели HSB и HSV

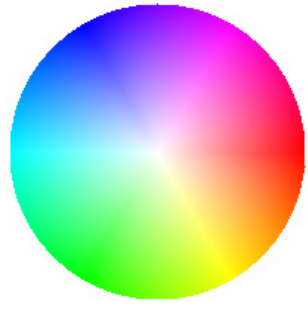


Рис. 2.6. Цветовой круг при S=1 и V=1

2.3.5. Модель Lab

Данная модель является аппаратно-независимой. Экспериментально доказано, что восприятие цвета зависит от наблюдателя и условий наблюдения. Члены Международной комиссии по освещению CIE (Commission Internationale de l'Eclairage) в 1931 г. стандартизировали условия наблюдения цветов и исследовали восприятие цвета у большой группы людей. В результате были экспериментально определены базовые компоненты цветовой модели XYZ. Эта модель аппаратно независима, поскольку описывает цвета так, как они воспринимаются "стандартным наблюдателем CIE". Цветовая модель Lab, использующаяся в компьютерной графике, является разновидностью цветовой модели XYZ.

Модель Lab получила название от своих базовых компонентов L, a и b. Компонент L несет информацию о яркости изображения, а компоненты a и b — о его цветах (т. е. a и b — хроматические компоненты). Компонент a изменяется от зеленого до красного, а компонент b — от синего до желтого.

Яркость в этой модели отделена от цвета, что удобно для регулирования контрастности и резкости. Однако, будучи абстрактной и сильно математизированной, эта модель неудобна для практической работы.

Поскольку все цветовые модели являются математическими, они легко конвертируются одна в другую по простым формулам. Такие конверторы встроены во многие графические программы.

2.4. Проект "Цветовые модели"

На компакт-диске (полное содержание которого приведено в *прил. 1*) в папке **Примеры | Глава 2 | Цветовые модели** приведен проект, предназначенный для иллюстрации работы с различными цветовыми моделями. Единственная форма этого проекта представлена на рис. 2.7.

Помимо формы `FormMain` в проект включен модуль `UnitModelColor`, включающий в себя процедуры перехода от одной цветовой модели к другой (листинг 2.4).

Листинг 2.4. Список процедур модуля *UnitModelColor*

```
procedure ColorToRGB(Color: integer; var R,G,B: real);
function RGBtoColor(R,G,B: real): integer;

procedure HueToRGB(m0,m2,H: real; var R,G,B: real);
function RGBtoHue(R,G,B: real): real;

// Hue, Lightness, Saturation = тон, яркость, частота света,
procedure HLStoRGB(H,L,S: real; var R,G,B: real);
procedure RGBtoHLS(R,G,B: real; var H,L,S: real);
```

```
// HSV (or HSB)
// Hue Saturation Brightness = Тон Насыщенность Яркость
procedure HSVtoRGB(H,S,V: real; var R,G,B: real);
procedure RGBtoHSV(R,G,B: real; var H,S,V: real);
// H определяет частоту света
// и принимает значение от 0 до 360 градусов.
// V - значение (принимает значения от 0 до 1) или B - яркость,
// определяющая уровень белого света является высотой конуса.
// S - определяет насыщенность цвета.
// Значение S является радиусом конуса.

// Hue Saturation Intensity
procedure HSItoRGB(H,S,I: real; var R,G,B: real);
procedure RGBtoHSI(R,G,B: real; var H,S,I: real);
```

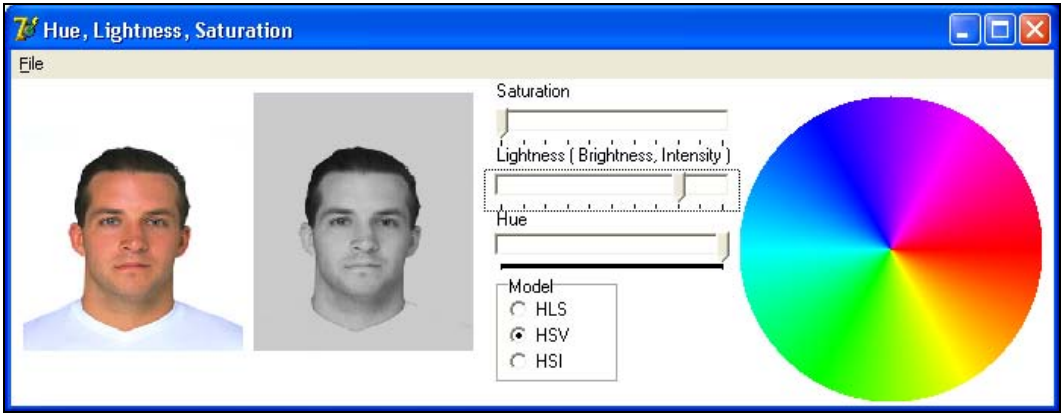


Рис. 2.7. Проект "Цветовые модели"

2.4.1. Процедуры для модели RGB

Для работы с моделью RGB предназначены две процедуры: `ColorToRGB` и `RGBtoColor`, позволяющие перейти от параметров R, G, B к цвету и обратно (листинг 2.5).

Листинг 2.5. Процедуры работы с моделью RGB

```
procedure ColorToRGB(Color: integer; var R,G,B: real);
begin
  R:=(Color and $0000FF)/255;
  G:=((Color and $00FF00) shr 8)/255;
  B:=((Color and $FF0000) shr 16)/255;
end;
```

```
function RGBtoColor(R,G,B: real): integer;
begin
  Result:=Trunc(B*255) shl 16+Trunc(G*255) shl 8+Trunc(R*255);
end;
```

2.4.2. Процедуры для модели HSV

Для работы с моделью HSV (или HSB) предназначены две следующие процедуры конвертации (листинг 2.6).

Листинг 2.6. Процедуры работы с моделью HSV (или HSB)

```
procedure HSVtoRGB(H,S,V: real; var R,G,B: real);
var m0: real;
begin
  R:=-1; G:=-1; B:=-1;
  if not Verify(V) or not Verify(S) then EXIT;
  if V<=Eps2 then begin
    R:=0; G:=0; B:=0; EXIT;
  end;
  m0:=(1-S)*V;
  if Abs(V-m0)<=Eps2 then begin
    R:=V; G:=V; B:=V; EXIT;
  end;
  HueToRGB(m0,V,H,R,G,B);
end;

procedure RGBtoHSV(R,G,B: real; var H,S,V: real);
var m0,m2: real;
begin
  H:=0; S:=-1; V:=-1;
  if not Verify(R) or not Verify(G) or not Verify(B) then EXIT;
  m0:=Minimum(R,G,B); m2:=Maximum(R,G,B);
  if m2<=Eps2 then begin
    S:=0; V:=0; EXIT;
  end;
  V:=m2; S:=(m2-m0);
  if S<=Eps2 then S:=0 else S:=S/m2;
  if S<=Eps2 then begin
    S:=0; EXIT;
  end;
  H:=RGBtoHue(R,G,B);
end;
```

Эти две процедуры используют пять вспомогательных процедур: `Verify` (листинг 2.7), `HueToRGB` (листинг 2.8), `Minimum(R,G,B: real): real`, `Maximum(R,G,B: real): real` и `RGBtoHue` (листинг 2.9). Текст функций `Minimum` и `Maximum` очевиден и здесь не приводится.

Листинг 2.7. Принадлежность переменной *V* интервалу (0,1)

```
function Verify(var v: real): boolean;
begin
  Result := (V>=-Eps) and (V<=1+Eps);
  if Result then begin
    if v<=Eps then v:=0;
    if v>=1-Eps then v:=1;
  end;
end;
```

Листинг 2.8. Преобразование тона (Hue) в RGB

```
procedure HueToRGB(m0,m2,H: real; var R,G,B: real);
var mu,md,F: real; n: integer;
begin
  while H<0 do H:=H+360;
  n:=Trunc(H/60); F:=(H-n*60)/60;
  n:=n mod 6; mu:=m0+(m2-m0)*F; md:=m2-(m2-m0)*F;
  case n of
    0: begin R:=m2; G:=mu; B:=m0; end;
    1: begin R:=md; G:=m2; B:=m0; end;
    2: begin R:=m0; G:=m2; B:=mu; end;
    3: begin R:=m0; G:=md; B:=m2; end;
    4: begin R:=mu; G:=m0; B:=m2; end;
    5: begin R:=m2; G:=m0; B:=md; end;
  end;
end;
```

Листинг 2.9. Преобразование RGB в тон (Hue)

```
function RGBtoHue(R,G,B: real): real;
var F,m0,m1,m2: real; n: integer;
begin
  m2:=Maximum(R,G,B);
  m0:=Minimum(R,G,B);
  if Abs(m2-m0)<=Eps then begin
    Result:=0; EXIT;
  end;
  m1:=Centre(R,G,B);
```

```
  if Abs(R-m2)<=Eps then
    if Abs(B-m0)<=Eps then n:=0 else n:=5
  else
    if Abs(G-m2)<=Eps then
      if Abs(B-m0)<=Eps then n:=1 else n:=2
    else
      if Abs(R-m2)<=Eps then n:=3 else n:=4;

  if (n and 1)=0 then F:=m1-m0 else F:=m2-m1;
  F:=F/(m2-m0);
  Result:=60*(n+F);
end;
```

Функция `RGBtoHue` использует функцию `Centre`, вычисляющую среднее значение из трех (листинг 2.10).

Листинг 2.10. Вычисление среднего значения из трех

```
function Centre(R,G,B: real): real;
begin
  if (R<G) and (R<B) then
    if G<B then Result:=G else Result:=B
  else
    if (G<R) and (G<B) then
      if R<B then Result:=R else Result:=B
    else
      if R<G then Result:=R else Result:=G;
end;
```

2.4.3. Процедуры для модели HSI

Для работы с моделью HSI (Hue, Saturation, Intensity — тон, насыщенность, интенсивность) предназначены процедуры `HSItoRGB` и `RGBtoHSI` (листинг 2.11).

Листинг 2.11. Процедуры *HSItoRGB* и *RGBtoHSI*

```
procedure HSItoRGB(H,S,I: real; var R,G,B: real);
var m0,m2: real;
begin
  R:=-1; G:=-1; B:=-1;
  if not Verify(S) or not Verify(I) then EXIT;
  m0:=I*(1-S); m2:=I;
  if m2<=Eps2 then begin
    R:=0; G:=0; B:=0; EXIT;
  end;
  if S<=Eps2 then begin
```

```
    R:=I; G:=I; B:=I; EXIT;
  end;
  if (m2-m0)<=Eps2 then begin
    R:=I; G:=I; B:=I; EXIT;
  end;
  HueToRGB(m0,m2,H,R,G,B);
end;

procedure RGBtoHSI(R,G,B: real; var H,S,I: real);
var m0,m2: real;
begin
  H:=0; S:=-1; I:=-1;
  if not Verify(R) or not Verify(G) or not Verify(B) then EXIT;
  m2:=Maximum(R,G,B); m0:=Minimum(R,G,B);
  I:=m2;
  if I<=Eps2 then begin
    I:=0; S:=0; EXIT;
  end;
  S:=m2-m0;
  if S<=Eps2 then S:=0 else S:=S/m2;
  if S<Eps2 then begin
    S:=0; EXIT;
  end;
  H:=RGBtoHue(R,G,B);
end;
```

В проекте "Цветовые модели" оригинал изображения хранится на `Bitmap`. Цвет каждой точки раскладывается с помощью процедуры `ColorToRGB` в доли R, G, B. Затем, в зависимости от выбранной модели, осуществляется преобразование R, G, B в параметры S, L, V, I. Эти параметры изменяются в соответствии со значениями компонентов `TrackBar` и снова преобразуются в R, G, B. Наконец R, G, B преобразуется в цвет точки на `Bitmap2` (листинг 2.12).

Листинг 2.12. Преобразование цветов с помощью моделей HLS, HSV и HSI

```
procedure TFormMain.N1Click(Sender: TObject);
var i,j: integer;
    R,G,B: real;
    H,S,V,L,II: real;
begin
  with Bitmap,Canvas do begin
    for i:=0 to Bitmap.Width-1 do
    for j:=0 to Bitmap.Height-1 do begin
      ColorToRGB(Pixels[i,j],R,G,B);
      case RadioGroup1.ItemIndex of
        0: RGBtoHLS(R,G,B,H,L,S);
```

```
        1: RGBtoHSV(R,G,B,H,S,V);
        2: RGBtoHSI(R,G,B,H,S,II);
      end;
      S:=S*TrackBar1.Position/10;
      case RadioGroup1.ItemIndex of
        0: L:=L*TrackBar2.Position/10;
        1: V:=V*TrackBar2.Position/10;
        2: II:=II*TrackBar2.Position/10;
      end;
      L:=L*TrackBar2.Position/10;
      //H:=TrackBar3.Position;
      case RadioGroup1.ItemIndex of
        1: HSVtoRGB(H,S,V,R,G,B);
        0: HLStoRGB(H,L,S,R,G,B);
        2: HSItoRGB(H,S,II,R,G,B);
      end;
      Bitmap2.Canvas.Pixels[i,j]:=RGBtoColor(R,G,B);
    end;
  end;
  Image1.Canvas.StretchDraw(Rect(0,0,Image1.Width,Image1.Height), Bitmap2);
end;
```

Напомним, что полностью проект приведен на компакт-диске (*см. прил. 1*) в папке **Примеры | Глава 2 | Цветовые модели**.

2.5. Класс *TFont*

Класс `TFont=class(TGraphicsObject)` предназначен для описания шрифтов Windows. К сожалению, методы канвы модуля `Graphics` позволяют рисовать только горизонтально расположенные шрифты. В модуле `Graphics` приводится следующее (листинг 2.13) описание класса `TFont`. Здесь и далее приводятся в алфавитном порядке только доступные свойства (`property`) и методы (`procedure`) класса.

Листинг 2.13. Описание класса *TFont*

```
Type
  TFont = class(TGraphicsObject)
  public
    procedure Assign(Source: TPersistent); override;
    property FontAdapter: IChangeNotifier;
    property Handle: HFont;
    property PixelsPerInch: Integer;
  published
    property Charset: TFontCharset;
    property Color: TColor;
```

```
    property Height: Integer;
    property Name: TFontName;
    property Pitch: TFontPitch;
    property Size: Integer read;
    property Style: TFontStyles;
  end;
```

Многие свойства класса `TFont` могут быть настроены с помощью стандартного метода `TFontDialog`, вызываемого методом `Execute` (листинг 2.14). Метод `TFontDialog` отображает диалоговое окно **Шрифт**, используемое для выбора шрифта и задания параметров символов.

Листинг 2.14. Вызов компонента *TFontDialog*

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  if FontDialog1.Execute then begin
    Font:= FontDialog1.Font;
    …
  end;
end;
```

Внешний вид диалогового окна **Шрифт** представлен на рис. 2.8.

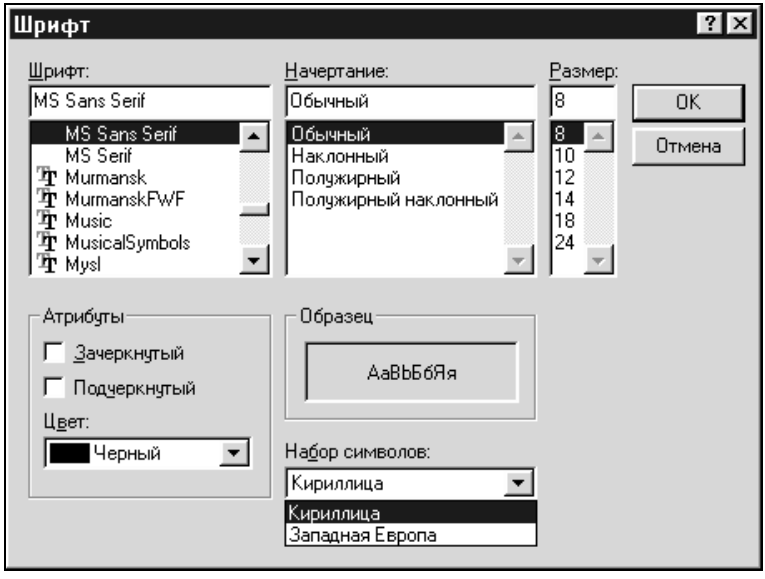


Рис. 2.8. Диалоговое окно **Шрифт**

Рассмотрим более подробно свойства класса `TFont`. Далее в тексте будут приводиться многочисленные примеры.

Для определенности будем использовать в них компонент `Image1`, обладающий канвой для рисования `Canvas`.

- Свойство `Handle` содержит дескриптор шрифта.

  ```
  property Handle: HFont
  ```

- Свойство `Name` определяет вид используемого шрифта.

  ```
  property Name: TFontName
  ```

  Это свойство содержит имя шрифта, например,

  ```
  Image1.Canvas.Font.Name: ='MS Sans Serif'.
  ```

  Список имен всех доступных шрифтов легко получить в любом пользовательском приложении. Так для того, чтобы передать в компонент `ListBox1` список зарегистрированных в Windows шрифтов, достаточно воспользоваться одним из свойств класса `TScreen: ListBox1.Items := Screen.Fonts` (рис. 2.9).

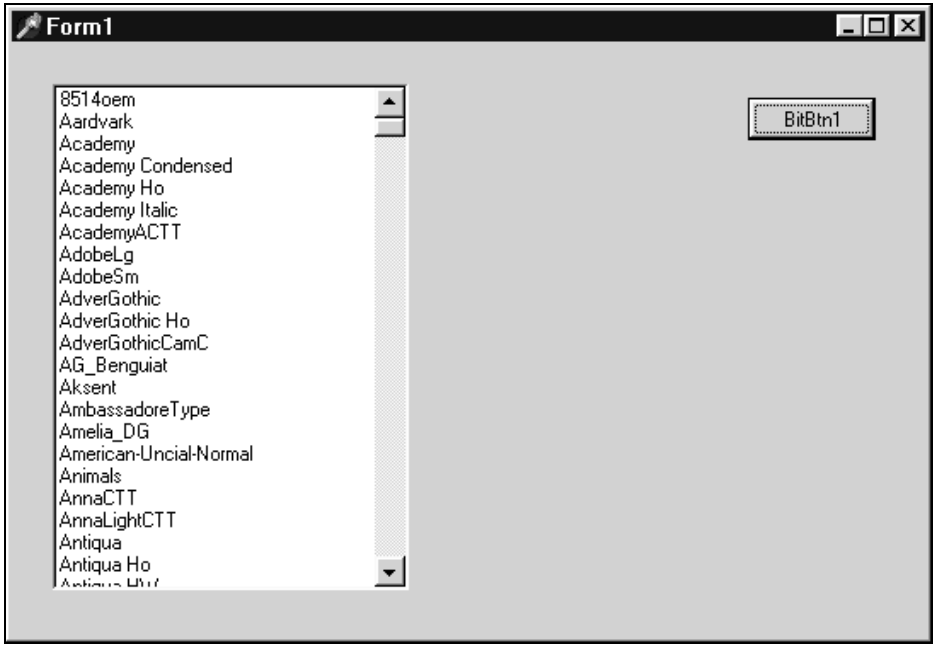


Рис. 2.9. Доступ к списку шрифтов через свойство *Screen.Fonts*

- Свойство `Style` содержит стиль начертания символов шрифта: жирный (`fsBold`), курсив (`fsltalic`), подчеркнутый (`fsUnderline`) и перечеркнутый (`fsStrikeOut`).

  ```
  property Style: TFontStyles;
  TFontStyle = (fsBold, fsltalic, fsUnderline, fsStrikeOut) ;
  TFontStyles = set of TFontStyle;
  ```

Для установки жирного курсива, например на канве компонента `Image1`, необходимо выполнить следующее действие: `Image1.Canvas.Font.Style:=[fsBold, fsltalic]`. Эти установки будут использованы при печати строки методом `TextOut`. Так, например, для вывода строки `Font.Style!` в точке с координатами `(100,100)` канвы компонента `Image1` необходимо использовать следующий код: `Image1.Canvas.TextOut(100,100,'Font.Style')`. Строка занимает на экране некоторый прямоугольник. Точка `(100,100)` — левый верхний угол этого прямоугольника.

- Свойство `Color` определяет цвет шрифта.

```
property Color: TColor
```

Например, для установки красного цвета символов необходимо выполнить следующее присваивание: `Image1.Canvas.Font.Color:=clRed`. Из программы цвет шрифта может быть задан предопределенной константой (список таких констант приведен в листинге 2.2) или целым значением.

- Свойство `Charset` определяет набор символов шрифта.

```
property Charset: TFontCharset nodefault;
type TFontCharset = 0..255;
```

Каждый тип шрифта (определенный свойством `Name`) поддерживает один или более наборов символов. Таблица 2.2 содержит встроенные константы, предусмотренные для стандартных наборов символов.

Таблица 2.2. *Константы, определяющие стандартные наборы символов*

Описание	Значение	Назначение
ANSI_CHARSET	0	Символы ANSI
DEFAULT_CHARSET	1	Шрифт выбран по умолчанию. Если описанный шрифт не доступен системе, Windows использует другой шрифт
SYMBOL_CHARSET	2	Установленные стандартные символы
MAC_CHARSET	77	Macintosh-символы. Не доступно на NT 3.51
SHIFTJIS_CHARSET	128	Японские символы
HANGEUL_CHARSET	129	Корейские символы (Wansung)
JOHAB_CHARSET	130	Корейские символы (Johab). Не доступно на NT 3.51
GB2312_CHARSET	134	Упрощенные китайские символы (материковый Китай)
CHINESEBIG5_CHARSET	136	Традиционные китайские символы (Тайвань)
GREEK_CHARSET	161	Греческие буквы. Не доступно на NT 3.51
TURKISH_CHARSET	162	Турецкие символы. Не доступно на NT 3.51
VIETNAMESE_CHARSET	163	Вьетнамские символы. Не доступно на NT 3.51
HEBREW_CHARSET	177	Еврейские символы. Не доступно на NT 3.51

Таблица 2.2 *(окончание)*

Описание	Значение	Назначение
ARABIC_CHARSET	178	Арабские символы. Не доступно на NT 3.51
BALTIC_CHARSET	186	Балтийские символы. Не доступно на NT 3.51
RUSSIAN_CHARSET	204	Символы кириллицы. Не доступно на NT 3.51
THAI_CHARSET	222	Тайские символы. Не доступно на NT 3.51
EASTEUROPE_CHARSET	238	Включает диакритические маркеры для восточных европейских стран. Не доступно на NT 3.51
OEM_CHARSET	255	Зависит от кодовой страницы операционной системы

Для того чтобы быть уверенным в том, что будет использована именно русская часть таблицы шрифтов, рекомендуется установить ее:

```
Image1.Canvas.Font.CharSet:=RUSSIAN_CHARSET.
```

- Свойство `Pitch` определяет ширину символов шрифта.

```
property Pitch: TFontPitch;
TFontPitch = (fpDefault, fpVariable, fpFixed)
```

Значение `fpFixed` соответствует моноширинным шрифтам, `fpVariable` — шрифтам с переменной шириной символа. Использование значения `fpDefault` означает принятие ширины символов, определенной начертанием. Равноширинных шрифтов совсем немного (например, `MS Sans Serif` или `Courier`). Но любой неравноширинный шрифт можно сделать равноширинным:

```
Image1.Canvas.Font.Pitch:=fpFixed.
```

- Свойство `Height` определяет высоту шрифта в пикселах.

```
property Height: Integer
```

Например, `Image1.Canvas.Font.Height:=-10`. Свойство `Height` связано со свойством `Size`, о котором пойдет речь далее.

- Свойство `PixelsPerInch` определяет число точек на дюйм.

```
property PixelsPerInch: Integer
```

Это свойство желательно не изменять, т. к. оно используется системой для приведения изображения на экране и принтере к одному виду.

- Свойство `Size` содержит размер шрифта в пунктах.

```
property Size: Integer
```

Это свойство связано со свойством `Height` соотношением:

```
Font.Size := -Font.Height * 72 / Font.PixelsPerInch;.
```

Примечание

Один пункт равен 1/72 дюйма.

2.6. Класс *TPen*

Класс `TPen=class(TGraphicsObject)` предназначен для описания свойств пера (карандаша). Этот инструмент используется канвой для рисования линий. По умолчанию создается непрерывное (`psSolid`) черное перо шириной в один пиксел. В модуле `Graphics` приводится следующее описание класса `Tpen` (листинг 2.15).

Листинг 2.15. Описание класса *TPen*

```
TPen = class(TGraphicsObject)
  public
    procedure Assign(Source: TPersistent); override;
    property Handle: HPen;
  published
    property Color: TColor;
    property Mode: TPenMode;
    property Style: TPenStyle;
    property Width: Integer;
end;
```

Рассмотрим свойства класса `TPen`. Перо обладает цветом (`Color`), модой (`Mode`), стилем (`Style`) и шириной (`Width`).

- Свойство `Handle` описывает дескриптор пера.

  ```
  property Handle: HPen
  ```

- Свойство `Color` определяет цвет пера.

  ```
  property Color: TColor
  ```

  Для изменения цвета пера во время выполнения проекта можно выполнить следующее присваивание:

  `Image1.Canvas.Pen.Color: =clRed` или `Image1.Canvas. Pen.Color: =$0000FF`.

- Свойство `Mode` (мода, режим) описывает одну из бинарных растровых операций.

  ```
  property Mode: TPenMode;
  ```

  ```
  TPenMode = (pmBlack, pmWhite, pmNop, pmNot, pmCopy, pmNotCopy, pmMergePenHot, pmMaskPenNot, pmMergeNotPen, pmMaskNotPen, pmMerge, pmHotMerge, pmMask, pmNotMask, pmXor, pmNotXor);
  ```

  Эти операции определяют взаимодействие цвета пера с цветом экрана и соответствуют стандартным операциям, определенным в Windows (табл. 2.3).

Таблица 2.3. *Возможные значения свойства Mode*

Мода (*Mode*)	Логическая операция перо-экран	Результирующий цвет (*Color*)
`PmBlack`	`Black`	Всегда черный
`PmWhite`	`White`	Всегда белый

Таблица 2.3 *(окончание)*

Мода (*Mode*)	Логическая операция перо-экран	Результирующий цвет (*Color*)
PmNop	Нет	Неизменный
PmNot	Not Screen	Инверсия цвета экрана
PmCopy	Pen	Цвет пера, указанный свойством Pen.Color
PmNotCopy	Not Pen	Инверсия цвета пера
PmMergePen-Not	Pen or not Screen	Дизъюнкция цвета пера и инверсии цвета экрана
PmMaskPenNot	Pen and not Screen	Конъюнкция цвета пера и инверсии экрана
PmMergeNot-Pen	Not Pen or Screen	Дизъюнкция цвета экрана и инверсии цвета пера
PmMaskNotPen	Not Pen and Screen	Конъюнкция цвета экрана и инверсии пера
PmMerge	Pen or Screen	Дизъюнкция цвета пера и цвета экрана
PmNotMerge	Not (Pen or Screen)	Инверсия дизъюнкции цвета пера и цвета экрана
PmMask	Pen and Screen	Конъюнкция цветов пера и экрана
PmNotMask	Not(Pen and Screen)	Инверсия конъюнкции цветов пера и экрана
PmXor	Pen xor Screen	Цвета пера и экрана связаны операцией "исключающее или"
PmNotXor	Not(Pen xor Screen)	Инверсия операции "исключающее или" для цветов пера и экрана

Логические битовые операции (Pen op Screen) представлены в табл. 2.4.

Таблица 2.4. *Возможные значения свойства Mode в битовом представлении*

Операция (op)	Pen	0	1	0	1
	Screen	0	0	1	1
pmBlack	0	0	0	0	0
pmMask	1	0	0	0	1
pmMaskNotPen	2	0	0	1	0
pmNop	3	0	0	1	1
pmMaskPenNot	4	0	1	0	0
pmCopy	5	0	1	0	1
pmXor	6	0	1	1	0
pmMerge	7	0	1	1	1
pmNotMerge	8	1	0	0	0

Таблица 2.4 *(окончание)*

Операция (op)	Pen	0	1	0	1
	Screen	0	0	1	1
pmNotXor	9	1	0	0	1
pmNotCopy	10	1	0	1	0
pmMergeNotPen	11	1	0	1	1
pmNot	12	1	1	0	0
pmMergePenNot	13	1	1	0	1
pmNotMask	14	1	1	1	0
pmWhite	15	1	1	1	1

По умолчанию устанавливается режим `Mode=pmCopy`, при котором цвет линий определяется свойством `Pen.Color`, то есть линия пера закрашивает все под собой своим цветом. Другим, часто используемым значением свойства `Mode`, является значение `pmNotXor`. Этот режим замечателен тем, что при повторном рисовании цвет точек экрана восстанавливается. Этот прием используется далее при рисовании линий методом "резиновых нитей" в графическом редакторе.

Обсудим более подробно бинарные битовые операции с растром. В табл. 2.5 приведены результаты выполнения битовых операций `or`, `and` и `xor` с операндами A и B, которые могут принимать значения нуль или единица. Битовая операция `not` меняет нулевые биты на единичные, а единичные — на нулевые. Бинарные битовые операции выполняются по правилам, представленным в табл. 2.5. Полный список всех шестнадцати бинарных операций приведен далее *(см. гл. 10)*.

Таблица 2.5. *Бинарные битовые операции*

A	0	0	1	1
B	0	1	0	1
A and B	0	0	0	1
A or B	0	1	1	1
A xor B	0	1	1	0

Если, например, установлен режим `Mode=pmXor`, цвет точки на экране имеет бинарный номер `B00101`, а цвет пера имеет бинарный номер `B01001`, то в результате растровой операции цвет точки будет равен `B00101 Xor B01001=B01100`.

- Свойство `Style` определяет стиль линии, рисуемой пером.

```
property Style: TPenStyle;
```

```
TPenStyle = (psSolid, psDash, psDot, psDashDot, psDashDotDot, psClear,
psInsideFrame);
```

Возможные значение этого свойства представлены в табл. 2.6.

Таблица 2.6. *Возможные значения стиля линии, рисуемой пером*

Стиль (*Style*)	**Назначение**
`PsSolid`	Перо рисует сплошную линию
`PsDash`	Перо рисует линию, составленную из ряда пунктиров
`PsDot`	Перо рисует линию, составленную из ряда точек
`PsDashDot`	Перо рисует штрихпунктирную линию
`PsDashDotDot`	Перо рисует линию, составленную из комбинаций пунктир-точка-точка
`PsClear`	Перо рисует невидимые линии
`PsInsideFrame`	Перо рисует сплошные цветные линии с шириной больше 1 и с возможностью выбора цвета вне таблицы цветов Windows

На рис. 2.10 изображены линии всех возможных стилей.

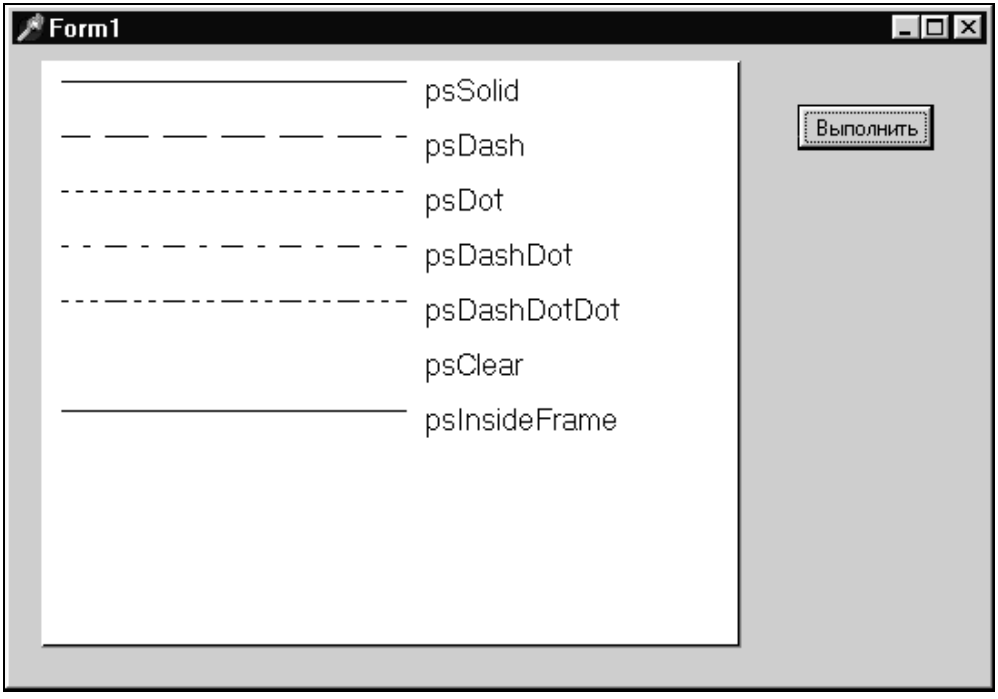


Рис. 2.10. Возможные стили пера

Примечание

Только стиль `psInsideFrame` дает возможность произвольно выбирать цвет. Все другие стили выбирают цвет из цветовой таблицы Windows.

При смене стиля линии необходимо воспользоваться таким присваиванием: `Image1.Canvas.Pen.Style:=psDashDot`. К сожалению, стиль можно менять только для пера шириной в 1 пиксел.

- Свойство `Width` содержит значение толщины пера в пикселах.

  ```
  property Width: Integer
  ```

2.7. Класс *TBrush*

Класс `TBrush = class(TGraphicsObject)` описывает кисть, то есть битовый шаблон, используемый при закрашивании областей. В модуле `Graphics` приводится следующее описание класса `TBrush` (листинг 2.16).

Листинг 2.16. Описание класса *TBrush*

```
TBrush = class(TGraphicsObject)
  public
    procedure Assign(Source: TPersistent); override;
    property Bitmap: TBitmap;
    property Handle: HBrush;
  published
    property Color: TColor;
    property Style: TBrushStyle;
end;
```

Опишем более подробно свойства класса `TBrush`. Класс `TBrush` обладает цветом (`Color`), предопределенным стилем (`Style`) и произвольным стилем (`Bitmap`), загружаемым из файла.

- Свойство `Handle` содержит описание дескриптора кисти.

  ```
  property Handle: HBrush.
  ```

- Свойство `Color` определяет цвет кисти.

  ```
  property Color: Tcolor.
  ```

  Изменение цвета во время выполнения проекта можно выполнить так: `Image1.Canvas.Brush.Color:=clRed.`

- Свойство `Style` содержит стандартный стиль кисти, т. е. определяет фактуру закраски.

  ```
  property Style: TBrushStyle;
  ```

  ```
  TBrushStyle = (bsSolid, bsClear, bsHorizontal, bsVertical, bsFDiagonal, bsBDiagonal, bsCross, bsDiagCross) ;
  ```

В табл. 2.7 представлены возможные значения свойства `Style`.

Таблица 2.7. *Возможные значения стиля кисти*

Стиль (*Style*)	Назначение
`bsSolid`	Сплошной
`bsClear`	Прозрачный
`bsHorizontal`	Горизонтальные линии
`bsVertical`	Вертикальные линии
`bsFDiagonal`	Диагональные линии
`bsBDiagonal`	Диагональные линии
`bsCross`	Решетка
`bsDiagCross`	Диагональная решетка

На рис. 2.11 приведены возможные стили кисти.

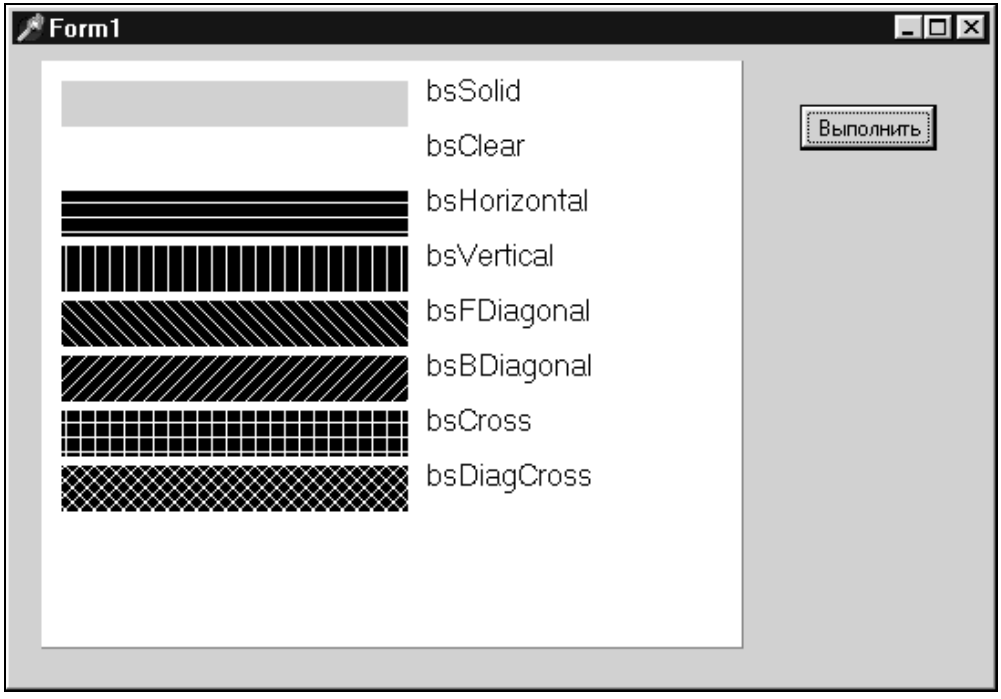


Рис. 2.11. Возможные стили кисти

Далее приводится код (листинг 2.17), в котором устанавливается `Brush.Style:=bsDiagCross` и рисуется прямоугольник.

Листинг 2.17. Установка стиля кисти

```
with Image1.Canvas do
begin
  Brush.Color:= clRed;
  Brush.Style:=bsDiagCross;
  RectAngle(100,100,200,200);
End.
```

- Свойство `Bitmap` может содержать битовую карту, определенную пользователем для закраски областей.

  ```
  property Bitmap: TBitmap
  ```

  Если это свойство определено, то свойства кисти `Color` и `Style` не используются.

Код, представленный в листинге 2.18, загружает битовое изображение из файла `Edit.bmp` и назначает его в качестве стиля для кисти канвы `Form1`. Оператор `try … finally … end` позволяет сразу же после использования переменной `Bitmap` освободить память, выделенную ей под изображение.

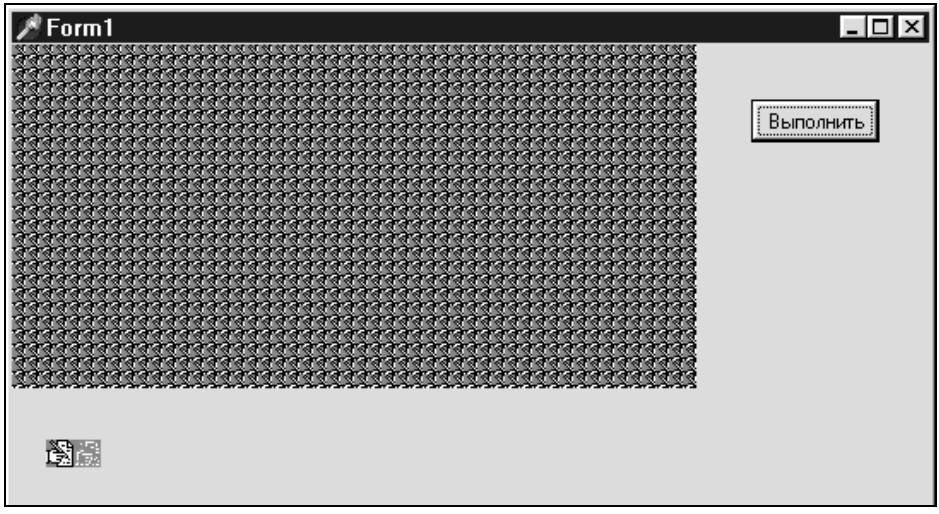


Рис. 2.12. Использование шаблона кисти из файла

Листинг 2.18. Назначение битового стиля кисти

```
var Bitmap: TBitmap;
begin
  Bitmap := TBitmap.Create;
  try
    Bitmap.LoadFromFile('Edit.bmp');
    Form1.Canvas.Brush.Bitmap := Bitmap;
```

```
    Form1.Canvas.FillRect(Rect(0,0,100,100));
  finally
    Form1.Canvas.Brush.Bitmap := nil;
    Bitmap.Free;
  end;
end
```

На рис. 2.12 представлен результат работы программы с таким кодом.

Обратите внимание на то, что для шаблона из файла используется только часть пикселов — блок размером 8×8 пикселов из левого верхнего угла рисунка формата BMP.

2.8. Класс *TCanvas*

Класс `TCanvas=class(TPersistent)` объединяет в себе устройство графического интерфейса `Graphics Device Interface (GDI)`, инструменты (перо, кисть, шрифт) и набор методов для рисования простейших геометрических фигур.

Хотя класс `TCanvas` не является компонентом, но он входит в качестве свойства во многие другие визуальные компоненты. Описание доступных свойств и методов этого класса в модуле `Graphics` выглядит так (листинг 2.19).

Листинг 2.19. Описание класса *TCanvas*

```
TCanvas = class(TPersistent)
public
  procedure Arc(X1,Y1,X2,Y2,X3,Y3,X4,Y4:Integer);
  procedure BrushCopy(const Dest: TRect; Bitmap:
    TBitmap; const Source: TRect; Color: TColor);
  procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);
  procedure CopyRect(const Dest: TRect; Canvas:
    TCanvas; const Source: TRect);
  procedure Draw(X, Y: Integer; Graphic: TGraphic);
  procedure DrawFocusRect(const Rect: TRect);
  procedure Ellipse(X1, Y1, X2, Y2: Integer);
  procedure FillRect(const Rect: TRect);
  procedure FloodFill(X, Y: Integer; Color: TColor;
    FillStyle: TFillStyle);
  procedure FrameRect(const Rect: TRect);
  procedure LineTo(X, Y: Integer);
  procedure Lock;
  procedure MoveTo(X, Y: Integer);
  procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer);
  procedure Polygon(const Points: array of TPoint);
  procedure Polyline(const Points: array of TPoint);
```

```
  procedure PolyBezier(const Points: array of TPoint);
  procedure PolyBezierTo(const Points: array of TPoint);
  procedure Rectangle(X1, Y1, X2, Y2: Integer);
  procedure Refresh;
  procedure RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer);
  procedure StretchDraw(const Rect: TRect; Graphic: TGraphic);
  function TextExtent(const Text: string): TSize;
  function TextHeight(const Text: string): Integer;
  procedure TextOut(X, Y: Integer; const Text: string);
  procedure TextRect(Rect: TRect; X, Y: Integer;
    const Text: string);
  function TextWidth(const Text: string): Integer;
  function TryLock: Boolean;
  procedure Unlock;
  property ClipRect: TRect
  property Handle: HDC
  property LockCount: Integer
  property CanvasOrientation: TCanvasOrientation
  property PenPos: TPoint
  property Pixels[X, Y: Integer]: TColor
  property TextFlags: Longint
  property OnChange: TNotifyEvent
  property OnChanging: TNotifyEvent
published
  property Brush: TBrush
  property CopyMode: TCopyMode
  property Font: TFont
  property Pen: TPen
end;
```

Рассмотрим более подробно свойства класса `TCanvas`.

- Дескриптор канвы `Handle` — это указатель на область памяти, занимаемую совместимым контекстом графического устройства. Так называют структуру данных, содержащую всю информацию, которая необходима для вывода изображения на экран монитора или принтер.
- Свойства `Brush`, `Pen`, `Font` из раздела `published` уже описаны, обсудим свойство `CopyMode`.

  ```
  TCopyMode = Longint
  ```

  Свойство `CopyMode` определяет способ взаимодействия холста с изображением, скопированным с другого холста. По умолчанию принимается, что `CopyMode=cmSrcCopy`, а это означает, что пикселы другого холста копируются на холст, перекрывая любое изображение. Изменяя `CopyMode`, можно создавать различные эффекты. Таблица 2.8 показывает возможные значения свойства `CopyMode` и описывает каждое из них.

Таблица 2.8. *Возможные значения свойства CopyMode*

Название	Значение	Логическая операция	Результат
CmBlackness	$00000042	BLACK	Превращает все пикселы на полотне в черные
CmDstInvert	$00550009	NOT dest	Инвертирует цвета всех пикселов
CmMergeCopy	$00C000CA	source AND pattern	Объединяет шаблон и исходное полотно, используя логический оператор AND
CmMergePaint	$00BB0226	(NOT source) OR dest	Объединяет инвертированное исходное полотно с результирующим полотном, используя логический оператор OR
CmNotSrcCopy	$00330008	NOT source	Копирует инвертированное исходное полотно
CmNotSrcErase	$001100A6	(NOT src) AND (NOT dest)	Инвертирует результат объединения результирующего полотна и исходного полотна, используя логический оператор OR
CmPatCopy	$00F00021	Pattern	Копирует шаблон
CmPatInvert	$005A0049	pattern XOR dest	Объединяет результирующее полотно с шаблоном, используя логический оператор XOR
CmPatPaint	$00FB0A09	DPSnoo	Объединяет инвертированное исходное полотно с шаблоном, используя логический оператор OR
CmSrcAnd	$008800C6	source AND dest	Объединяет точки результирующего полотна и исходного полотна, используя логический оператор AND
CmSrcCopy	$00CC0020	Source	Копирует исходное полотно в результирующее полотно
CmSrcErase	$00440328	Source AND (NOT dest)	Инвертирует результирующее полотно и объединяет результат с исходным полотном, используя логический оператор AND
CmSrcInvert	$00660046	Source XOR dest	Объединяет точки результирующего полотна и исходное полотно, используя логический оператор XOR
CmSrcPaint	$00EE0086	source OR dest	Объединяет точки результирующего полотна и исходное полотно, используя логический оператор OR
CmWhiteness	$00FF0062	WHITE	Превращает все пикселы на полотне в белые

2.9. Методы канвы

Приведем в алфавитном порядке описание методов рисования, описанных в разделе `public` класса `TCanvas`.

- Метод `Arc` рисует дугу на холсте по периметру эллипса, ограниченного прямоугольником `(X1,Y1,X2,Y2)`.

  ```
  procedure Arc(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer)
  ```

  Начальная точка сегмента лежит на пересечении эллипса и луча, проведенного из его центра через точку `(X3,Y3)`. Конечная точка сегмента лежит на пересечении эллипса и луча, проведенного из его центра через точку `(X4,Y4)`.

В следующем примере (листинг 2.20) на канве формы рисуется четверть дуги, ограниченная текущим окном.

Листинг 2.20. Пример рисования дуги

```
procedure TForm1.FormPaint(Sender: TObject);
var R: TRect;
begin
  R:=GetClientRect;
  Canvas.Arc(R.Left, R.Top, R.Right, R.Bottom, R.Right,
    R.Top, R.Left, R.Top);
end;
```

Тип `TRect` определен в модуле `Types`:

```
TRect = packed record
  case Integer of
    0: (Left, Top, Right, Bottom: Longint);
    1: (TopLeft, BottomRight: TPoint);
  end;
```

Тип `TPoint` также определен в модуле `Types` и имеет два поля:

```
TPoint = packed record
  X: Longint;
  Y: Longint;
end;
```

На рис. 2.13 в левом верхнем углу нарисованы дуга, сектор, эллипс, пересечение линии с эллипсом и линия.

- Метод `BrushCopy` копирует прямоугольник `Source` из битовой карты `Bitmap` в прямоугольник `Dest` на канве, при этом для точек, имеющих цвет `Color`, назначается цвет кисти (`Brush.Color`).

  ```
  procedure BrushCopy(const Dest: TRect; Bitmap: TBitmap; const Source:
  TRect; Color: TColor)
  ```

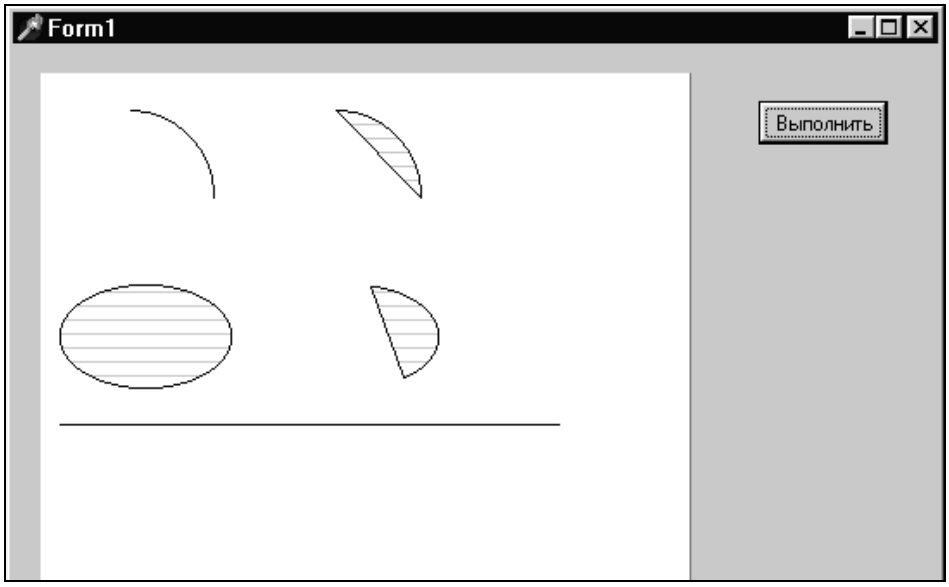


Рис. 2.13. Примитивы канвы: дуга, сектор, эллипс, пересечение линии с эллипсом, линия

Следующий пример (листинг 2.21) иллюстрирует различия между методами `CopyRect` и `BrushCopy`. Графический файл TARTAN.BMP загружается в `Bitmap` и выводится на канву формы `Form1`. Метод `BrushCopy` заменяет черный цвет цветом кисти холста, а `CopyRect` оставляет все цвета.

Листинг 2.21. Сравнение методов *BrushCopy* и *CopyRect*

```
var Bitmap: TBitmap;
    MyRect, MyOther: TRect;
begin
  yRect.Top := 10;  MyRect.Left := 10;
  MyRect.Bottom := 100;  MyRect.Right := 100;
  MyOther.Top := 111;  MyOther.Left := 10;
  MyOther.Bottom := 201;  MyOther.Right := 100;
  try
    Bitmap := TBitmap.Create;
    Bitmap.LoadFromFile('tartan.bmp');
    Form1.Canvas.BrushCopy(MyRect,Bitmap,MyRect,clBlack);
    Form1.Canvas.CopyRect(MyOther,
    Bitmap.Canvas,MyRect);
  finally
    Bitmap.Free;
  end;
end;
```

- Метод `Chord` рисует линию на холсте, соединяющую две точки на эллипсе, ограниченном указанным прямоугольником, и заливает отсекаемую ею часть эллипса.

  ```
  procedure Chord(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer)
  ```

  Эллипс ограничен прямоугольником `(X1,Y1,X2,Y2)`. Начальная точка хорды лежит на пересечении эллипса и луча, проведенного из его центра через точку `(X3,Y3)`. Конечная точка хорды лежит на пересечении эллипса и луча, проведенного из его центра через точку `(X4,Y4)` (рис. 2.13).

- Метод `CopyRect` копирует прямоугольник `Source` из канвы `Canvas` в прямоугольник `Dest` канвы метода.

  ```
  procedure CopyRect(const Dest: TRect; Canvas: TCanvas; const Source: TRect)
  ```

- Метод `Draw` выводит графический объект `Graphic` на холсте в прямоугольной области с верхним левым углом `(X, Y)`.

  ```
  procedure Draw(X, Y: Integer; Graphic: TGraphic)
  ```

  Параметр `Graphic` может быть файлом форматов BMP, ICO или метафайлом.

Следующий пример (листинг 2.22) рисует графический файл TARTAN.BMP в центре канвы формы `Form1`.

Листинг 2.22. Пример использования метода *Draw*

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Bitmap1: TBitmap;
begin
  Bitmap1 := TBitmap.Create;
  Bitmap1.LoadFromFile('tartan.bmp');
  Form1.Canvas.Draw(
    (Form1.Width div 2)- Bitmap1.Widthdiv 2),
    (Form1.Height div 2) - (Bitmap1.Height div 2), Bitmap1);
end;
```

- Метод `DrawFocusRect` рисует прямоугольник из точек (рис. 2.14).

  ```
  procedure DrawFocusRect(const Rect: TRect)
  ```

  Такой прямоугольник, в частности, выводится на элемент, получающий фокус ввода. Так как метод использует логическую операцию `not Xor`, повторный вызов того же метода приводит к стиранию прямоугольника.

В следующем примере (листинг 2.23) рисуется прямоугольник `DrawFocusRect` вокруг компонента `RadioButton`.

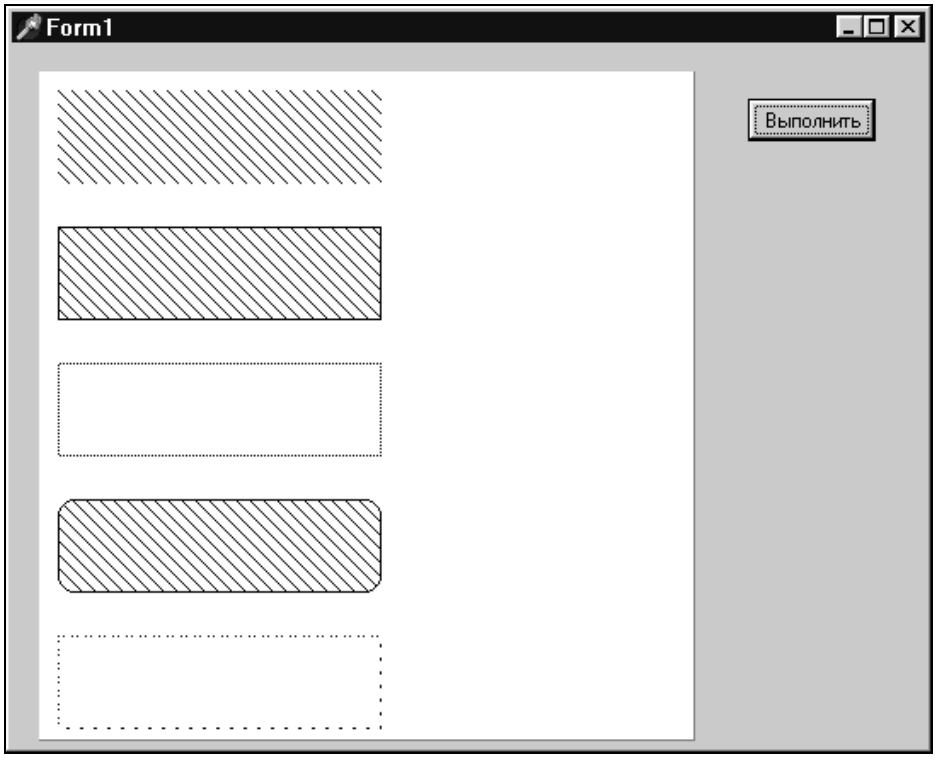


Рис. 2.14. Примитивы канвы:
FillRect, *RectAngle*, *DrawFocusRect*, *RoundRect*, *FrameRect*

Листинг 2.23. Пример использования метода *DrawFocusRect*

```
procedure TForm1.Button1Click(Sender: TObject);
var  NewRect: TRect;
begin
  NewRect := RadioButton1.BoundsRect;
  with NewRect do begin
    Left := Left - 10; Top := Top - 10;
    Right := Right + 10; Bottom := Bottom + 10;
  end;
  Form1.Canvas.DrawFocusRect(NewRect);
end;
```

- Метод `Ellipse` рисует эллипс (рис. 2.13), вписанный в прямоугольник `(X1,Y1),(X2,Y2)`.

  ```
  procedure Ellipse(X1, Y1, X2, Y2: Integer)
  ```

  Если точки прямоугольника образуют квадрат, то рисуется круг.

В следующем примере (листинг 2.24) рисуется эллипс, заполняющий форму.

Листинг 2.24. Пример использования метода *Ellipse*

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Ellipse(0, 0, ClientWidth, ClientHeight);
end;
```

- Метод `FillRect` заполняет указанный прямоугольник `Rect` на холсте, используя текущую кисть (рис. 2.14).

  ```
  procedure FillRect(const Rect: TRect)
  ```

В следующем примере (листинг 2.25) рисуется красный прямоугольник (рис. 2.14) на холсте формы.

Листинг 2.25. Пример использования метода *FillRect*

```
procedure TForm1.ColorRectangleClick(Sender: TObject);
var  NewRect: TRect;
begin
  NewRect := Rect(20, 30, 50, 90);
  Form1.Canvas.Brush.Color := clRed;
  Form1.Canvas.FillRect(NewRect);
end;
```

- Метод `FloodFill` заливает область текущей кистью.

  ```
  procedure FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle)
  ```

  Процесс начинается с точки `(X,Y)`. Если параметр `FillStyle` равен `fsSurface`, то заливка продолжается до тех пор, пока есть соседние точки с цветом `Color`. Если же параметру присваивается значение `fsBorder`, то заливка прекращается при достижении границ с цветом `Color`. Заливка со значением параметра `Color=fsBorder` приводит к заливке всей канвы компонента, если ограничивающий контур "проколот" точкой другого цвета.

  В следующем примере `FloodFills` заливает из центра формы `Form1` до черной границы:

  ```
  Form1.Canvas.FloodFill(ClientWidth div 2, ClientHeight div 2, clBlack, fsBorder); .
  ```

- Метод `FrameRect` рисует контур прямоугольника, используя кисть холста.

  ```
  procedure FrameRect(const Rect: TRect)
  ```

  Данный метод не заполняет внутренность прямоугольника кистью `Brush` (рис. 2.14).

Следующий пример (листинг 2.26) показывает строку `FrameRect` в прямоугольнике, определенном координатами `(10,10)` и `(100,100)`. После вывода строки методом `TextRect`, рисуется черная рамка стилем `bsVertical` вокруг прямоугольника.

Листинг 2.26. Пример использования методов *FrameRect* и *TextRect*

```
var  TheRect: TRect;
begin
  Canvas.Brush.Color := clBlack;
  Canvas.Brush.Style := bsVertical;
  TheRect.Top := 10;  TheRect.Left := 10;
  TheRect.Bottom := 100;  TheRect.Right := 100;
  Canvas.TextRect(TheRect,10,10,'FrameRect');
  Canvas.FrameRect(TheRect);
end;
```

- Метод `LineTo` рисует линию (рис. 2.13) текущим пером от точки, в которой находится указатель, до точки `(X,Y)`, перемещая указатель в точку `(X,Y)`.

  ```
  procedure LineTo(X, Y: Integer)
  ```

В следующем примере (листинг 2.27) рисуется линия от верхнего левого угла формы к точке, в которой щелкают указателем мыши.

Листинг 2.27. Пример использования метода *LineTo*

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;  Shift:
TShiftState; X, Y: Integer);
begin
  Canvas.MoveTo(0, 0);
  Canvas.LineTo(X, Y);
end;
```

- Метод `Lock` вызывается в мультизадачных приложениях для того, чтобы запретить другим потокам использовать холст, пока не вызван метод `Unlock`.

  ```
  procedure Lock
  ```

  Вложенные запросы `Lock` увеличивают свойство `LockCount`, для того чтобы холст не открыли до тех пор, пока не будет освобожден последний замок. В мультизадачных приложениях, использующих `Lock` для защиты холста, все запросы, использующие холст, должны быть защищены запросом `Lock`. Любому потоку, не захватывающему холст перед использованием, грозят потенциальные ошибки.

- Метод `MoveTo` перемещает текущее положение пера в точку `(X,Y)`.

  ```
  procedure MoveTo(X, Y: Integer)
  ```

  Использование `MoveTo` устанавливает текущее положение быстрее, чем непосредственная установка свойства `PenPos`.

- Метод `Pie` рисует сектор эллипса (рис. 2.13), описываемого прямоугольником `(X1,Y1)`, `(X2,Y2)`.

  ```
  procedure Pie(X1, Y1, X2, Y2, X3, Y3, X4, Y4: Integer)
  ```

  Стороны сектора лежат на лучах, проходящих из центра эллипса через точки `(X3,Y3)` и `(X4,Y4)`.

В следующем примере (листинг 2.28) рисуется сектор эллипса (рис. 2.13) на холсте формы, когда пользователь нажимает кнопку `Button1`.

Листинг 2.28. Пример использования метода *Pie*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Canvas.Pie(10, 10, 200, 200, 61, 3, 200, 61);
end;
```

- Метод `Polygon` рисует замкнутый многоугольник, используя массив координат точек `Points`.

  ```
  procedure Polygon(const Points: array of TPoint)
  ```

  Последняя точка соединяется с первой, и внутренняя область заливается, используя текущие свойства кисти (рис. 2.15).

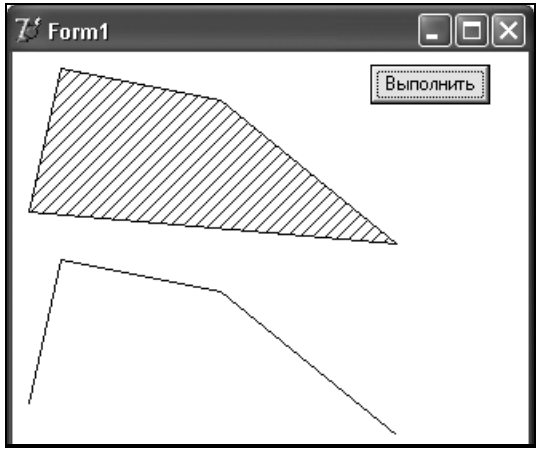


Рис. 2.15. Примитивы *Polygon* и *Polyline*

Функция `Slice` может использоваться для того, чтобы передать часть массива точек в метод `Polygon`. Например, если имеется массив, содержащий 100 точек, а формируется многоугольник из первых 10, можно использовать функцию `Slice`:

```
Canvas.Polygon.(Slice(PointArray, 10)).
```

Следующий пример (листинг 2.29) рисует многоугольник на форме и заполняет его цветом `clTeal`.

Листинг 2.29. Пример использования метода *Polygon*

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  Canvas.Brush.Color := clTeal;
```

```
  Canvas.Brush.Style:=bsBDiagonal;
  Canvas.Polygon([Point(10,100),
    Point(30,10),Point(130,30), Point(240,120)]);
end;
```

- Метод `Polyline` строит ломаную линию, координаты вершин которой определяются массивом `Points` (рис. 2.15).

  ```
  procedure Polyline(const Points: array of TPoint)
  ```

В следующем примере (листинг 2.30) рисуется красная ломаная линия.

Листинг 2.30. Пример использования метода *Polyline*

```
procedure TForm1.FormPaint(Sender: TObject);
begin
  Canvas.Pen.Color := clRed;
  Canvas.PolyLine([Point(10,100+120),
    Point(30,10+120),Point(130,30+120), Point(240,120+120)]);
end;
```

- Метод `PolyBezier` используется для рисования кубических кривых Безье.

  ```
  procedure PolyBezier(const Points: array of TPoint)
  ```

  Концевые и промежуточные точки указываются параметром `Points`. Первая кривая выходит из первой точки до четвертой точки, используя вторую и третью точки как промежуточные. Каждая последующая кривая в последовательности нуждается в трех точках: концевая точка предыдущей кривой используется как отправная точка, следующие две точки в последовательности — промежуточные, и третья — концевая точка. Этот метод рисует линии, используя текущее перо.

- Метод `PolyBezierTo`, как и предыдущий, рисует набор кубических кривых Безье.

  ```
  procedure PolyBezierTo(const Points: array of TPoint)
  ```

  Как и в методе `PolyBezier`, концевые и промежуточные точки указываются параметром `Points`. Метод `PolyBezierTo`, как и предыдущий, рисует линии, используя текущее перо, но, в отличие от метода `PolyBezier`, изменяет значение `PenPos` к значению последней точки.

- Метод `Rectangle` рисует прямоугольник (рис. 2.14) на холсте с верхним левым углом в точке `(X1,Y1)` и нижним правым углом в точке `(X2,Y2)`, используя текущие свойства кисти `(TBrush)` и карандаша `(TPen)`.

  ```
  procedure Rectangle(X1, Y1, X2, Y2: Integer)
  ```

Следующий пример (листинг 2.31) рисует несколько прямоугольников различных размеров и цветов на форме. Один такой прямоугольник изображен на рис. 2.14.

Листинг 2.31. Пример использования метода *Rectangle*

```
var  X, Y: Integer;
procedure TForm1.FormActivate(Sender: TObject);
```

```
begin
  WindowState := wsMaximized;
  Canvas.Pen.Width := 5;
  Canvas.Pen.Style := psDot;
  Timer1.Interval := 50;
  Randomize;
end;
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  X := X + 4;
  Y := Y + 4;
  Canvas.Pen.Color := Random(65535);
  Canvas.Rectangle(X,Y,X + Random(400),Y + Random(400));
  if X > 700 then  Timer1.Enabled := False;
end;
```

- Метод `Refresh` устанавливает инструменты шрифт, перо и кисть со стандартным набором свойств (`BLACK.PEN`, `HOLLOW.BRUSH`, `SYSTEM.FONT`).

  ```
  procedure Refresh
  ```

- Метод `RoundRect` рисует прямоугольник с закругленными углами (рис. 2.14).

  ```
  procedure RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer)
  ```

  Закругления изображаются дугами эллипса с размерами полуосей по горизонтали и вертикали `X3` и `Y3`.

- Метод `StretchDraw` осуществляет рисование объекта `Graphic` в заданном прямоугольнике `Rect`.

  ```
  procedure StretchDraw(const Rect: TRect; Graphic: TGraphic)
  ```

  Если размеры `Rect` и `Graphic` не совпадают, то `Graphic` масштабируется и вписывается в `Rect`. Этот метод более гибкий, чем метод `Draw`, в котором указывается не прямоугольник `Rect`, а только верхний левый угол, и `Graphic` всегда выводится без масштабирования.

  Следующий пример растягивает битовое изображение для того, чтобы заполнить клиентскую область формы `Form1`:

  ```
  Form1.Canvas.StretchDraw(Form1.ClientRect, TheGraphic).
  ```

- Метод `TextExtent` используется для того, чтобы определить область, которую займет строка на холсте.

  ```
  function TextExtent(const Text: string): TSize;
  TSize = record  cx: Longint;  cy: Longint; end;
  ```

  Данный метод возвращает ширину и высоту в пикселах строки, предоставленной в текущем шрифте. Для определения только высоты строки можно использовать метод `TextHeight`, только ширины — `TextWidth`.

- Метод `TextHeight` возвращает высоту строки `Text` в пикселах.

  ```
  function TextHeight(const Text: string): Integer
  ```

Следующий пример (листинг 2.32) показывает высоту строки текста в текущем шрифте холста в поле компонента `Edit1`.

Листинг 2.32. Пример использования метода *TextHeight*

```
procedure TForm1.FormCreate(Sender: TObject);
var  L: LongInt;
begin
  L := Canvas.TextHeight('Привет!');
  Edit1.Text := 'Высота = '+IntToStr(L) + ' пикселов ';
end;
```

- Метод `TextOut` рисует строку `Text`. Левый верхний угол помещается в точку канвы `(X,Y)` (рис. 2.16).

  ```
  procedure TextOut(X,Y: Integer; const Text: string)
  ```

Рис. 2.16. Вывод строк методами *TextOut* и *TextRect*

Следующий пример (листинг 2.33) показывает строку текста в указанной позиции на форме, если пользователь щелкает кнопкой `Button1`.

Листинг 2.33. Пример использования метода *TextOut*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Canvas.TextOut(20, 20, ' procedure TForm1.Button ');
end;
```

- Метод `TextRect` производит вывод текста с отсечением (рис. 2.16).

  ```
  procedure TextRect(Rect: TRect; X, Y: Integer; const Text: string)
  ```

  Как и в `TextOut`, строка `Text` выводится с позиции `(X,Y)`, при этом часть текста, лежащая вне пределов прямоугольника `Rect`, отсекается и не будет видна.

В примере (листинг 2.34) текст "`TextRect`" выводится в прямоугольнике, определенном координатами `(10, 10)` и `(100, 100)`. Вершина и левые грани текста будут подрезаны прямоугольником (рис. 2.16).

Листинг 2.34. Пример использования метода *TextRect*

```
var  TheRect: TRect;
begin
  TheRect.Top := 10;  TheRect.Left := 10;
  TheRect.Bottom := 100;  TheRect.Right := 100;
  Form1.Canvas.TextRect(TheRect,0,0, 'TextRect');
end;
```

- Метод `TextWidth` возвращает ширину строки `Text` в пикселах.

  ```
  function TextWidth(const Text: string): Integer
  ```

В примере листинга 2.35 определяется ширина указанной строки и, если строка слишком широка, чтобы показать ее в `Edit1`, компонент `Edit1` расширяется для того, чтобы поместить строку.

Листинг 2.35. Пример использования метода *TextWidth*

```
procedure TForm1.Button1Click(Sender: TObject);
var
  T: Longint;
  S: string;
begin
  S := ' procedure TForm1.Button1Click';
  T := Canvas.TextWidth(S);
  if T > Edit1.Width then  Edit1.Width := T + 10;
  Edit1.Text := S;
end;
```

- Метод `TryLock` захватывает холст, если он в настоящее время доступен.

  ```
  function TryLock: Boolean
  ```

  Вызовите метод `TryLock` для того, чтобы запретить другим потокам вывод на поверхность холста, до тех пор, пока не будет вызван метод `Unlock`. В отличие от метода `Lock`, организованного так, чтобы каждый запрос `Lock` был согласован с последующим запросом `Unlock`, метод `TryLock` должен быть согласован только с запросом `Unlock`, если холст не "заперт".

Если холст "отпирают", TryLock захватывает холст, приравнивает значение свойства LockCount к 1 и возвращает значение True. Если холст уже захвачен, Trylock возвращает False и не позволяет делать никаких изменений.

Свойство TryLock используется перед попыткой изменений на холсте, которые могут конфликтовать с другой программой, использующей Lock ... Unlock.

- Метод Unlock уменьшает значение свойства LockCount на 1, позволяя другим потокам выполняться, если LockCount достигает 0.

```
procedure Unlock
```

Вызывайте Unlock в конце фрагмента, начатого вызовом метода Lock. Каждый запрос Lock должен быть согласован с вызовом Unlock.

- Свойство ClipRect определяет область отсечения канвы.

```
property ClipRect: TRect
```

То, что при рисовании попадает за пределы этого прямоугольника, не будет изображено. Свойство доступно только для чтения. Любой рисунок, координаты которого находятся вне ClipRect, не появляется на экране. Например, ClipRect холста формы имеют те же самые размеры, что и клиентская область формы.

Если окно, содержащее рисунок, было перекрыто другим окном, то соответствующая часть рисунка оказывается испорченной и операционная система сообщает о необходимости перерисовки изображения. Такое сообщение приводит к возникновению события OnPaint. При этом свойство Canvas.ClipRect указывает область канвы, с испорченной частью изображения. Обработчик события OnPaint может использовать ClipRect, для того чтобы перерисовать только эту часть, ускорив тем самым выполнение приложения. Свойство ClipRect выгодно использовать в том случае, когда изменения на канве происходят в небольшой области, так как вне ClipRect рисование не выполняется.

- Свойство Handle — дескриптор холста.

```
property Handle: HDC
```

- Свойство LockCount указывает количество захватов холста (счетчик замков), для того чтобы предотвратить вмешательство других потоков.

```
property LockCount: Integer
```

Читайте LockCount, для того чтобы определить, захвачен ли холст. Всякий раз, когда метод Lock вызывается для блокирования от других потоков, значение LockCount увеличивается на 1. Всякий раз, когда метод Unlock вызывается в конце фрагмента рисования, значение LockCount уменьшается на 1. Другие потоки могут рисовать на холсте при LockCount = 0.

- Свойство CanvasOrientation определяет ориентацию холста слева-направо (coLeftToRight) или справа-налево (coRightToLeft). Это свойство предназначено только для чтения.

```
property CanvasOrientation: TCanvasOrientation read GetCanvasOrientaion;
TCanvasOrientation = (coLeftToRight, coRightToLeft);
```

По умолчанию, свойство `CanvasOrientation` принимает значение `coLeftToRight`, а это означает, что точка с координатами `0,0` находится в левом верхнем углу холста. Для изображений с элементами текста на языках стран, где принято написание текста справа-налево, например, стран Ближнего Востока, ориентация холста может быть `coRightToLeft`, т. е. точка `0,0` находится в верхнем правом углу холста. Эта свойство важно, например, для компонента `Grid`, потому что оно изменяет ориентацию холста сетки во время рисования.

- Свойство `PenPos` содержит и позволяет изменять текущую позицию пера канвы.

  ```
  property PenPos: TPoint read GetPenPos write SetPenPos
  ```

  Впрочем, позиция быстрее изменяется посредством метода `MoveTo`.

- Свойство `Pixels` позволяет получить непосредственный доступ к любой точке на холсте, устанавливать или читать ее цвет.

  ```
  property Pixels[X, Y: Integer]: TColor read GetPixel write SetPixel;
  ```

  Каждый элемент в `Pixels` содержит цвет точки. Индексы `X` и `Y` этого двумерного массива определяют, соответственно, горизонтальные и вертикальные координаты точки. Левый верхний угол канвы, как правило, соответствует элементу `Pixels[0,0]`.

> ***ЗАМЕЧАНИЕ***
>
> Использование свойства `Pixels` — это самый медленный способ рисования изображения.

В следующем примере (листинг 2.36) очень медленно рисуется красная горизонтальная линия.

Листинг 2.36. Пример использования свойства *Pixels*

```
procedure TForm1.FormActivate(Sender: TObject);
var i: Word;
begin
  for i := 10 to 200 do
    Canvas.Pixels[i,10] := clRed;
end;
```

- Свойство `TextFlags` определяет способ вывода текста на холст.

  ```
  property TextFlags: Longint
  ```

  Используйте свойство `TextFlags` для того, чтобы влиять на способ вывода текста при вызовах методов `TextOut` или `TextRect`. `TextFlags` — целое число, получаемое объединением с помощью битовых операций любого набора из следующих констант (табл. 2.9).

  Например, `TextFlags:= ETO_CLIPPED and ETO_OPAQUE`.

Таблица 2.9. *Константы, влияющие на вывод текста*

Константа	Значение
ETO_CLIPPED	Показан только текст, попадающий в указанный прямоугольник. Этот флаг автоматически устанавливается при вызове метода TextRect. При выводе текста, использующего метод TextOut, этот флаг не оказывает никакого влияния, потому что границы рисунка определены свойством TextExtent
ETO_OPAQUE	Текст выводится с непрозрачным цветом фона. Использование этого флага улучшает читаемость, но не позволяет видеть изображения позади прямоугольника текста
ETO_RTLREADING	Строки текста показаны справа-налево. Это удобно только в ближневосточных версиях Windows, где текст пишется справа-налево. Часто используется в комбинации с CanvasOrientation = coRightToLeft
ETO_GLYPH_INDEX	Текст — массив кодов положения символов, соответствующий синтаксису Windows GDI. Glyph-индексация применима только к шрифтам TrueType, но флаг может использоваться и с другими шрифтами, для того чтобы указать, что GDI должен обработать строку текста непосредственно без обработки языка. См. документацию Windows GDI
ETO_IGNORELANGUAGE	недокументированный флаг Microsoft
ETO_NUMERICSLOCAL	недокументированный флаг Microsoft
ETO_NUMERICSLATIN	недокументированный флаг Microsoft

- Свойства OnChange и OnChanging используются при изменениях свойств, например, свойств Color, Mode, Style или Width объекта Tpen, и вызовах методов TCanvas, меняющих вид канвы (то есть при любом рисовании).

  ```
  property OnChange: TNotifyEvent
  property OnChanging: TNotifyEvent
  ```

  При использовании метода перемещения пера MoveTo они не используются. Метод OnChanging вызывается до начала изменений, метод OnChange вызывается после их завершения.

2.10. Чтение данных из текстового файла

В предыдущем разделе мы привели достаточно много примеров рисования графических примитивов методами канвы. Можно нарисовать сколь угодно сложный рисунок, вставляя в код программы вызов методов с заданными значениями координат точек на канве. Но программы должны быть максимально независимы от конкретных данных. В нашем случае программа должна уметь рисовать различные картинки на канве. Для реализации этой идеи мы можем записать в текстовый

файл данные о рисунке, используя, например, следующий формат: в каждой строке файла находятся данные для одного метода рисования и в начале каждой строки содержится символ, предназначенный для обозначения метода канвы (табл. 2.10).

Таблица 2.10. *Символьные обозначения методов канвы*

Символ	Метод канвы	Символ	Метод канвы
C	Pen.Color	S	Brush.Style
c	Brush.Color	s	Text.Size
R	RectAngle	T	TextOut
F	FillRect		

Далее в строке записываются значения параметров, используемых для данного метода. Так, например, для изменения цвета пера на красный требуется одно целое значение `Pen.Color:=$0000FF`, а для рисования прямоугольника — четыре целых значения `RectAngle(100,100,400,200)`.

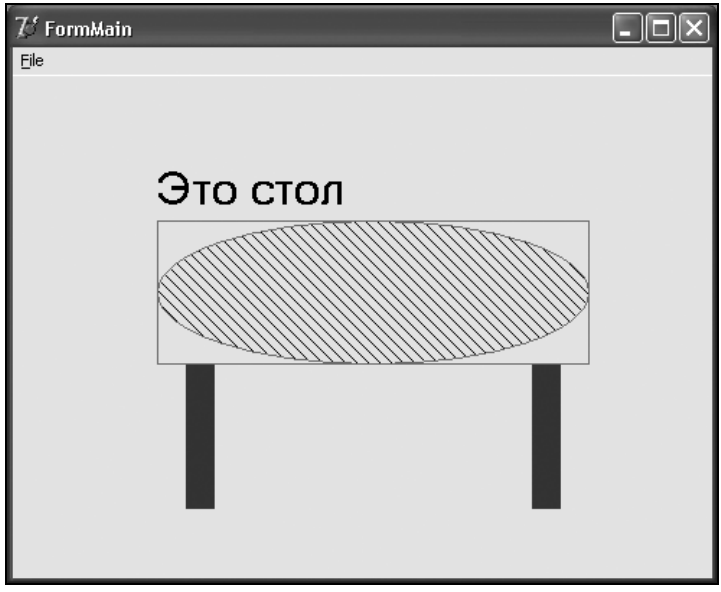


Рис. 2.17. Рисование при чтении данных из файла

В файле File1.TXT, текст которого приводится далее, приведен пример данных, используемых для создания изображения, приведенного на рис. 2.17.

```
C $0000FF
R 100 100 400 200
c $FF0000
F 120 200 140 300
```

```
F 360 200 380 300
c $00FF00
s bsFDiagonal
E 100 100 400 200
S 24
s bsClear
T 100 60 Это стол
```

Чтение текстового файла, разбор и рисование осуществляется процедурой `ReadTXT` (листинг 2.37).

Листинг 2.37. Чтение текстового файла и рисование

```
procedure TFormMain.ReadTXT(FileName: string);
var
  f          : textfile;
  x1,y1,x2,y2: integer;
  PenColor,
  BrushColor,
  FontSize   : integer;
  ch         : char;
  s          : string;
begin
  AssignFile(f,FileName); Reset(f);
  with Canvas do
  while not EOF(f) do begin
    Read(f,ch);
    case ch of
      'C': begin
             Readln(f,PenColor);
             Pen.Color:=PenColor;
           end;
      'c': begin
             Readln(f,BrushColor);
             Brush.Color:=BrushColor;
           end;
      'R': begin
             Readln(f,x1,y1,x2,y2);
             RectAngle(x1,y1,x2,y2);
           end;
      'E': begin
             Readln(f,x1,y1,x2,y2);
             Ellipse(x1,y1,x2,y2);
           end;
      'F': begin
```

```
            Readln(f,x1,y1,x2,y2);
            FillRect(Rect(x1,y1,x2,y2));
          end;
      'T': begin
            Readln(f,x1,y1,s);
            s:=Erase(s);
            TextOut(x1,y1,s);
          end;
      'S': begin
            Readln(f,FontSize);
            Font.Size:=FontSize;
          end;
      's': begin
            Readln(f,s);
            s:=EraseSpace(s);
            Brush.Style:=SetBrushStyle(S);
          end;
    end;
  end;
  CloseFile(f);
end;
```

В процедуре `ReadTXT` после выбора файла и его открытия в каждой строке считывается первый символ, анализируется оператором `case`, затем считываются остальные данные из строки, и рисуется один из примитивов на канве.

Если после первого символа считывается строка, то в начале этой строки могут быть пробелы. Для удаления пробелов в начале строки предназначена процедура `EraseSpace` (листинг 2.38).

Листинг 2.38. Удаление пробелов в начале строки

```
function EraseSpace(s: string): string;
begin
  while (Length(s)>0) and (s[1]=' ') do Delete(s,1,1);
  Result:=s;
end;
```

Задания стиля кисти в файле осуществляется строкой. Для преобразования этой строки в стиль кисти типа `TBrushStyle` предназначена процедура `SetBrushStyle` (листинг 2.39).

Листинг 2.39. Установка стиля кисти

```
function SetBrushStyle(S: string): TBrushStyle;
begin
  if S = 'bsSolid' then Result := bsSolid
  else if S = 'bsClear' then Result := bsClear
```

```
  else if S = 'bsHorizontal' then Result := bsHorizontal
  else if S = 'bsVertical' then Result := bsVertical
  else if S = 'bsFDiagonal' then Result := bsFDiagonal
  else if S = 'bsBDiagonal' then Result := bsBDiagonal
  else if S = 'bsCross' then Result := bsCross
  else if S = 'bsDiagCross' then Result := bsDiagCross;
end;
```

Проект приведен на компакт-диске (*см. прил. 1*) в папке **Примеры | Глава 2 | Данные из текстового файла**.

2.11. Вывод строки под углом

2.11.1. Установка угла для печати строки

При рисовании строки в Windows нет никакого способа указать направление шрифта: кажется, что Windows умеет рисовать строку только горизонтально. Однако чтобы быть точным, Windows рисует строку в направлении, соответствующем шрифту, являющемуся горизонтальным по умолчанию. Метод канвы `TextOut` умеет выводить строку только горизонтально. Но, тем не менее, существует возможность печать строку под любым углом. В книге Юань Феня "Программирование графики для Windows" [125] описан тип логического шрифта `TlogFont`, позволяющий реализовать такую возможность.

В листинге 2.40 приведена процедура, устанавливающая угол вывода текста под углом `Angle` в десятых долях градуса. Ее работа основана на том, что тип логического шрифта `TlogFont`, описанный в модуле Windows, содержит поля `lfEscapement` и `lfOrientation`, позволяющие задавать угол для направления строки и поворот каждого символа.

Листинг 2.40. Изменение угла направления печати строки

```
procedure SetAngle(Canvas: TCanvas; Angle: integer);
var LogFont: TLogFont; // Объявляем переменную логического шрифта
begin
  GetObject(Canvas.Font.Handle,SizeOf(LogFont),Addr(LogFont));
  LogFont.lfEscapement := Angle;
  LogFont.lfOrientation := Angle;
  Canvas.Font.Handle := CreateFontIndirect(LogFont);
end;
```

В качестве параметров для функции `SetAngle` мы передаем процедуре `Canvas` и угол поворота строки. Угол поворота `Angle` может меняться от 0 до 3600, т. е. через 0,1 градуса.

Рассмотрим по частям первую строчку:

- `GetObject` — возвращает информацию о графическом объекте `LogFont`;
- `Canvas.Font.Handle` — указатель на объект;
- `SizeOf(LogFont)` — возвращает размер переменной;
- `Addr(LogFont)` — адрес переменной.

С помощью функции `GetObject`, мы получаем информацию о шрифте, используемом нами для рисования. Вторая и третья строчки процедуры `SetAngle` изменяют значения полученной информации. Четвертая строка устанавливает измененный шрифт.

Теперь перейдем к процедуре рисования. Пример обращения к процедуре `SetAngle` приведен в листинге 2.41, а результат рисования показан на рис. 2.18.

Листинг 2.41. Рисование строки под углом 45°

```
with Bitmap.Canvas do begin
  Font.Size:=12;  Font.Name:='Times'; Font.Style:=[fsBold];
  SetAngle(Bitmap.Canvas, 450);
  TextOut(50, 150, 'Алгоритмы на деревьях');
end;
```

Процедура `SetAngle` устанавливает измененную информацию, и метод `TextOut` рисует строку под углом 45 градусов.

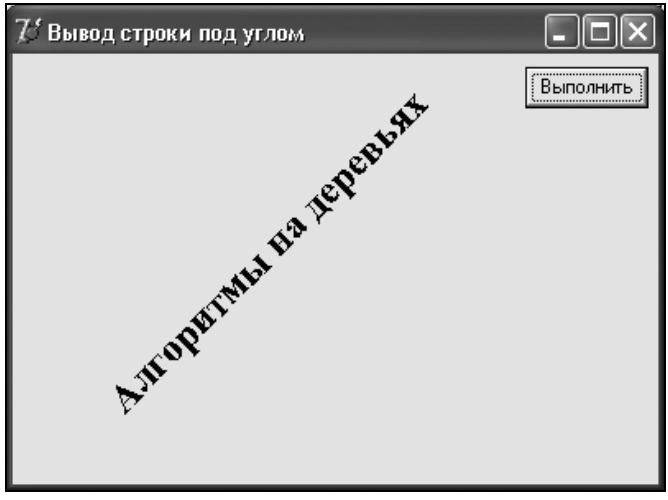


Рис. 2.18. Рисование строки под углом 45°

Этот маленький проект приведен на компакт-диске (*см. прил. 2*) в папке **Примеры | Глава 2 | Вывод строки**.

2.11.2. Тип логического шрифта

Поговорим более подробно о типе логического шрифта `TLogFont`. Тип `TLogFont=tagLogFontA` описан в модуле Windows и определяет атрибуты шрифта (листинг 2.42).

Листинг 2.42. Структура логического шрифта

```
tagLOGFONTA = packed record
  lfHeight: Longint;
  lfWidth: Longint;
  lfEscapement: Longint;
  lfOrientation: Longint;
  lfWeight: Longint;
  lfItalic: Byte;
  lfUnderline: Byte;
  lfStrikeOut: Byte;
  lfCharSet: Byte;
  lfOutPrecision: Byte;
  lfClipPrecision: Byte;
  lfQuality: Byte;
  lfPitchAndFamily: Byte;
  lfFaceName: array[0..LF_FACESIZE - 1] of AnsiChar;
end;
```

- Поле `lfHeight` определяет высоту ячейки символа шрифта или символа. Процедуры рисования шрифтов интерпретируют значение, указанное в `lfHeight` следующим способом (табл. 2.11).

Таблица 2.11. *Возможные значения поля lfHeight*

Значение	Описание
> 0	Процедуры рисования шрифтов преобразуют это значение в высоту матрицы символа доступных шрифтов
0	Процедуры рисования шрифтов используют заданное по умолчанию значение высоты
< 0	Процедуры рисования шрифтов преобразуют это значение в абсолютное значение высоты символов доступных шрифтов

Для режима отображения `MM_TEXT` можно использовать следующую формулу для определения высоты шрифта с заданным размером пункта:

```
lfHeight = -MulDiv(PointSize, GetDeviceCaps(hDC, LOGPIXELSY), 72);
```

- Поле `lfWidth` определяет среднюю ширину символов в шрифте. Если значение `lfWidth` является нулевым, коэффициент сжатия устройства согласовывается

с коэффициентом сжатия доступных шрифтов для определения наиболее близкого соответствия.

- Поле `lfEscapement` определяет угол (в десятых долях градуса) между вектором направления и осью `X` устройства. Вектор направления параллелен основной строке текста.

  Если в Windows NT установлен графический режим `GM_ADVANCED`, то можно определить угол направления строки независимо от угла ориентации символов строки. Если же установлен графический режим `GM_COMPATIBLE`, то `lfEscapement` определяет и направление и ориентацию. Необходимо установить `lfEscapement` и `lfOrientation` равными одному и тому же значению.

- Поле `lfOrientation` определяет угол (в десятых долях градуса) между направлением каждого символа и осью `X` устройства.

- Поле `lfWeight` определяет вес шрифта в диапазоне 0 до 1000 (см. табл. 2.12). Например, при значении 400 шрифт имеет нормальное начертание, а при значении 700 является полужирным. Если значение является нулевым, то используется заданный по умолчанию вес.

Для удобства определены следующие значения.

Таблица 2.12. *Возможные значения поля lfWeight*

Значение	Вес	Значение	Вес
`FW_DONTCARE`	0	`FW_SEMIBOLD`	600
`FW_THIN`	100	`FW_DEMIBOLD`	600
`FW_EXTRALIGHT`	200	`FW_BOLD`	700
`FW_ULTRALIGHT`	200	`FW_EXTRABOLD`	800
`FW_LIGHT`	300	`FW_ULTRABOLD`	800
`FW_NORMAL`	400	`FW_HEAVY`	900
`FW_REGULAR`	400	`FW_BLACK`	900
`FW_MEDIUM`	500		

- Поле `lfItalic` определяет курсивный шрифт, если поле установлено в значение `true`.
- Поле `lfUnderline` определяет подчеркнутый шрифт, если поле установлено в значение `true`.
- Поле `lfStrikeOut` определяет зачеркнутый шрифт, если поле установлено в значение `true`.
- Поле `lfCharSet` определяет набор символов. Значение `OEM_CHARSET` определяет набор символов операционной системы. Можно использовать значение `DEFAULT_CHARSET`, для того чтобы по имени и размеру шрифта полностью описывать логический шрифт. Если указанное имя шрифта не существует, то указан-

ный шрифт может быть заменен любым шрифтом. Поэтому необходимо осторожно использовать DEFAULT_CHARSET.

Параметр lfCharSet важен в процессе подстановки шрифтов. Задайте определенный набор символов, для того чтобы гарантировать непротиворечивые результаты. Если имя шрифта определяется в lfFaceName, проверьте, что значение lfCharSet соответствует набору символов шрифта, указанного в lfFaceName.

- Поле lfOutPrecision определяет точность вывода. Точность вывода определяет, как близко вывод должен соответствовать высоте требуемого шрифта, ширине, символьной ориентации, направлению, шагу и типу шрифта. Определены следующие значения поля lfOutPrecision (табл. 2.13).

Таблица 2.13. *Возможные значения поля lfOutPrecision*

Значение	Назначение
OUT_CHARACTER_PRECIS	Не используется
OUT_DEFAULT_PRECIS	Определяет заданное по умолчанию поведение процедур рисования шрифтов
OUT_DEVICE_PRECIS	Инструктирует процедуры рисования шрифтов выбирать шрифт устройства, когда система содержит несколько шрифтов с одним и тем же именем
OUT_OUTLINE_PRECIS	Это значение предписывает процедурам рисования шрифтов выбирать из TrueType и других шрифтов на основе иерархической структуры
OUT_RASTER_PRECIS	Предписывает процедурам рисования шрифтов выбирать растровый шрифт, если система содержит несколько шрифтов с одним и тем же именем
OUT_STRING_PRECIS	Это значение не используется процедурами рисования шрифтов, но возвращается при использовании растровых шрифтов
OUT_STROKE_PRECIS	Windows NT: это значение не используется процедурами рисования шрифтов, но возвращается, если используется TrueType, другие шрифты на основе иерархической структуры и векторные шрифты. Windows 95: это значение используется при изображении векторных шрифтов, и возвращается, если используется TrueType или векторные шрифты
OUT_TT_ONLY_PRECIS	Предписывает процедурам рисования шрифтов выбирать только из шрифтов TrueType. Если шрифтов нет TrueType, установленных в системе, то возвращает процедуры рисования шрифтов к заданному по умолчанию поведению
OUT_TT_PRECIS	Предписывает процедурам рисования шрифтов выбирать шрифт TrueType, если система содержит несколько шрифтов с одним и тем же именем

Приложения могут использовать значения `OUT_DEVICE_PRECIS`, `OUT_RASTER_PRECIS` и `OUT_TT_PRECIS` для того, чтобы управлять выбором процедур рисования шрифтов, если операционная система содержит более одного шрифта с данным именем. Например, если операционная система содержит растровый шрифт с именем `Symbol` в форме `TrueType`, то значение `OUT_TT_PRECIS` принуждает процедуры рисования шрифтов выбирать версию `TrueType`. Значение `OUT_TT_ONLY_PRECIS` вынудило бы процедуры рисования шрифтов выбирать шрифт `TrueType`, даже если бы пришлось заменить его другим шрифтом `TrueType`.

- Поле `lfClipPrecision` определяет точность отсечения, т. е. определяет, как отсечь символы, частично появляющиеся вне области отсечения. Это поле может принимать одно из следующих значений (табл. 2.14).

Таблица 2.14. *Возможные значения поля lfClipPrecision*

Значение	Назначение
`CLIP_DEFAULT_PRECIS`	Определяет значение по умолчанию
`CLIP_CHARACTER_PRECIS`	Не используется
`CLIP_STROKE_PRECIS`	Не используется процедурами рисования шрифтов, но возвращается, если назначен растровый, векторный или шрифт TrueType. В Windows NT для совместимости это значение всегда возвращается при перечислении шрифтов
`CLIP_MASK`	Не используется
`CLIP_EMBEDDED`	Необходимо определить этот флаг для использования пользовательского шрифта только для чтения
`CLIP_LH_ANGLES`	Если используется это значение, то вращение для всех шрифтов зависит от того, является ли ориентация системы координат левосторонней или правосторонней. Если не используемые шрифты устройства всегда вращаются против часовой стрелки, то вращение других шрифтов зависит от ориентации системы координат
`CLIP_TT_ALWAYS`	Не используется

- Поле `lfQuality` определяет качество вывода. Качество вывода определяет, как тщательно графический интерфейс устройства должен соответствовать логическому шрифту. Поле может принимать одно из следующих значений (табл. 2.15).

Таблица 2.15. *Возможные значения поля lfQuality*

Значение	Назначение
`DEFAULT_QUALITY`	Вид шрифта не имеет значения
`DRAFT_QUALITY`	Вид шрифта менее важен, чем при использовании `PROOF_QUALITY`.

Таблица 2.15 *(окончание)*

Значение	Назначение
	Для растровых шрифтов графического интерфейса устройств допускается масштабирование. Это означает, что допускается увеличение размеров шрифта, но качество может быть далее. Полужирный, курсивный, подчеркнутый и зачеркнутый шрифты синтезируются в случае необходимости
PROOF_QUALITY	Символьное качество шрифта более важно, чем точное соответствие атрибутов логического шрифта. Для растровых шрифтов графического интерфейса устройств, масштабирование заблокировано, и выбирается шрифт, наиболее близкий по размеру. Хотя выбранный размер шрифта не может быть отображен точно, если используется PROOF_QUALITY, но качество шрифта высокое и нет искажения. Полужирный, курсивный, подчеркнутый и зачеркнутый шрифты синтезируются в случае необходимости

- Поле lfPitchAndFamily определяет шаг (pitch) и семейство шрифта. Два младших бита определяют шаг шрифта и могут принимать одно из следующих значений: DEFAULT_PITCH, FIXED_PITCH, VARIABLE_PITCH. Биты 4–7 определяют семейство шрифта и могут принимать одно из следующих значений: FF_DECORATIVE, FF_DONTCARE, FF_MODERN, FF_ROMAN, FF_SCRIPT, FF_SWISS. Для объединения константы шага с константой семейства может быть использована логическая операция OR.

  Семейства шрифта описывают просмотр шрифта общим способом. Они предназначены для того, чтобы определить шрифты, когда желаемый шрифт не доступен. Значения поля lfPitchAndFamily для семейств шрифта приведены в табл. 2.16.

Таблица 2.16. *Возможные значения поля lfPitchAndFamily*

Значение	Назначение
FF_DECORATIVE	Новые шрифты. Например, Old English
FF_DONTCARE	Неизвестный
FF_MODERN	Шрифты с фиксированной шириной, с засечками или без них. Шрифты фиксированной ширины, обычно современные. Например, Pica, Elite и CourierNew ®
FF_ROMAN	Пропорциональные шрифты с засечками. Например, MS ® Serif
FF_SCRIPT	Рукописные шрифты. Например, Script и Cursive
FF_SWISS	Пропорциональные шрифты без засечек. Например, MS ® Sans Serif

- Поле `lfFaceName` — строка с нулевым символом на конце, определяющая имя шрифта. Длина этой строки не должна превысить 32 символа, включая нулевой признак конца. Функция `EnumFontFamilies` может использоваться для того, чтобы перечислить имена всех доступных шрифтов. Если `lfFaceName` — пустая строка, то графический интерфейс устройств использует первый шрифт, соответствущий другим указанным атрибутам.

2.12. Рисование на экране

Из приложения можно получить доступ не только к канве компонента, имеющего свойство `Canvas`, но и к канве любого компонента или ко всему экрану. Для этого объявим переменную `ScreenDC` типа `HDC` (см. листинг 2.43). Напомним, что тип `HDC` — это тип контекста рисования в Windows.

Листинг 2.43. Рисование эллипса на экране

```
procedure TFormMain.Button1Click(Sender: TObject);
var ScreenDC:HDC;
begin
  ScreenDC := GetDC(0);
  Ellipse(ScreenDC, 10, 10, 120, 120);
  ReleaseDC(0,ScreenDC);
end;
```

С помощью функции `GetDC()` можно взять контекст окна, указанного в скобках. Ноль означает, что используется контекст экрана. Метод `Ellipse` отличается от метода `TCanvas.Ellipse` только первым параметром — контекстом устройства. Это связано с тем, что ранее рисование шло через объект `TCanvas`, а при рисовании на экране — средствами GDI модуля Windows. Процедура `TCanvas.Ellipse` всего лишь вызывает `Ellipse` из Windows API и подставляет нужный контекст устройства и размеры, поэтому в ней на один параметр меньше. Вот так выглядит, например, метод `Ellipse` в модуле `Graphics`:

```
procedure TCanvas.Ellipse(X1, Y1, X2, Y2: Integer);
begin
  Changing;
  RequiredState([csHandleValid, csPenValid, csBrushValid]);
  Windows.Ellipse(FHandle, X1, Y1, X2, Y2);
  Changed;
end;
```

Аналогично переопределяет Delphi и остальные методы канвы через API-функции из модуля Windows.

После рисования через контекст экрана необходимо освободить контекст процедурой `ReleaseDC`. Для рисования не на экране, а внутри определенного окна или

на компоненте, в этой процедуре необходимо поправить только первую строчку: в качестве параметра `GetDC` передавать указатель на окно или на компонент.
Если необходимо менять свойства пера или кисти, то удобно ввести переменную `Canvas: TCanvas`, создать ее и направить контекст экрана на `Canvas` (листинг 2.44).

Листинг 2.44. Рисование эллипса на экране с использованием канвы

```
var ScreenDC:HDC;
    Canvas: TCanvas;
begin
  Canvas:=TCanvas.Create;
  ScreenDC := GetDC(0);
  Canvas.Handle:=ScreenDC;
  Canvas.Brush.Color:=clRed;
  Canvas.Ellipse(10, 10, 120, 120);
  ReleaseDC(0,ScreenDC);
  Canvas.Free;
end;
```

В следующем примере (листинг 2.45) обработчик события нажатия кнопки `Button3` на форме `Form2` с помощью процедуры `FindWindow` ищет указатель на форму `Form1` и рисует на ней прямоугольник.

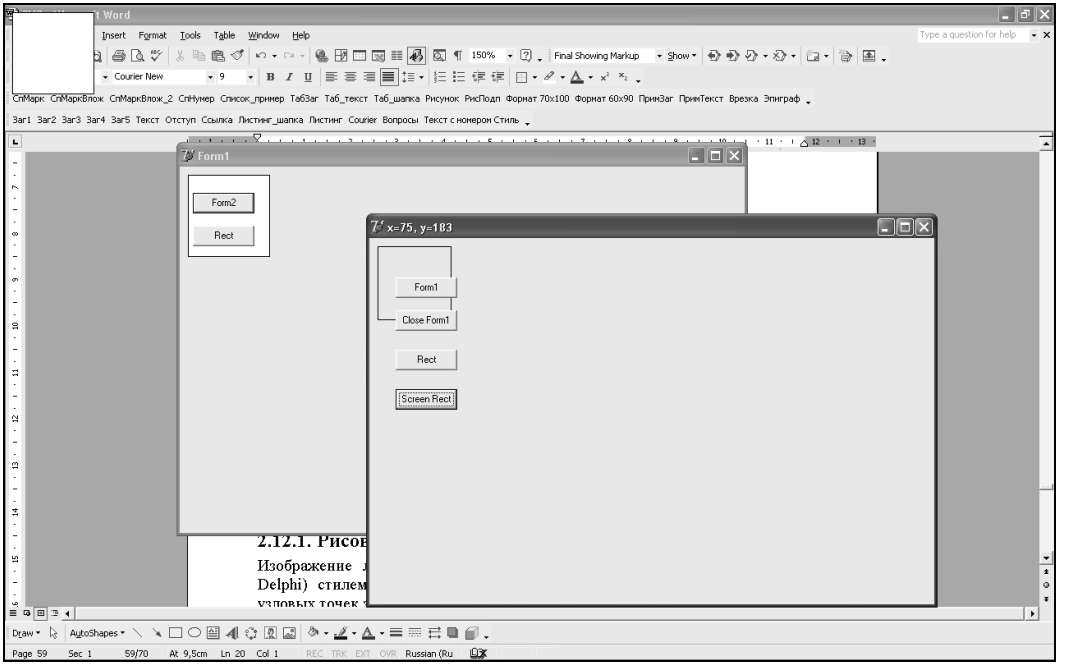


Рис. 2.19. Рисование на экране

Листинг 2.45. Рисование прямоугольника на экране с использованием канвы

```
procedure TForm2.Button3Click(Sender: TObject);
var
  dc: HDC; Window: HWND;
begin
  Window := FindWindow('TForm1', 'Form1');
  If Window <> 0 then begin // окно найдено
    dc := GetDC (Window);   // ссылка на найденное окно
    Rectangle (dc, 10, 10, 110, 110);
    // квадрат на чужом окне
    ReleaseDC (Window, dc); // освобождение ссылки
    DeleteDC (dc);          // удаление ссылки
  end;
end;
```

Результат выполнения листинга 2.45 представлен на рис. 2.19.

Проект приведен на компакт-диске (*см. прил. 2*) в папке **Примеры | Глава 2 | На экране**.

ГЛАВА 3

Графические классы

3.1. Класс *TGraphic*

Как уже говорилось в *главе 2*, абстрактный класс `TGraphic` является родительским для четырех видов изображений: пиктограммы (класс `TIcon`), метафайлы (класс `TMetaflle`), растровые картинки (класс `TBitmap`) и сжатые фотоизображения (класс `TJPEGImage`) [64]. Из этих четырех графических классов только `TBitmap` обладает свойством `TCanvas`. Методы класса `TGraphic` предназначены, прежде всего, для обмена графической информацией с файлами, потоками и буфером обмена. В листинге 3.1 приводится описание доступных методов и свойств этого класса.

Листинг 3.1. Описание класса *TGraphic*

```
TGraphic = class(TPersistent)
public
  procedure LoadFromFile(const Filename: string); virtual;
  procedure SaveToFile(const Filename: string); virtual;
  procedure LoadFromStream(Stream: TStream); virtual; abstract;
  procedure SaveToStream(Stream: TStream); virtual; abstract;
  procedure LoadFromClipboardFormat(AFormat: Word;
    AData: THandle; APalette: HPALETTE); virtual; abstract;
  procedure SaveToClipboardFormat(var AFormat: Word;
    var AData: THandle; var APalette: HPALETTE); virtual;abstract;
  property Empty: Boolean;
  property Height: Integer;
  property Modified: Boolean;
  property Palette: HPALETTE;
  property PaletteModified: Boolean;
  property Transparent: Boolean;
  property Width: Integer;
  property OnChange: TNotifyEvent;
  property OnProgress: TprogressEvent;
end;
```

Рассмотрим более подробно эти свойства и методы.

- Метод `LoadFromFile` создает файловый поток `FileStream` и читает изображение из файла, вызывая метод `LoadFromStream`.

  ```
  procedure LoadFromFile (const Filename: string)
  ```

  Старое содержание `Graphic` теряется. Если файл не имеет допустимого формата, то возникает состояние ошибки.

  Обычно метод `LoadFromFile` (листинг 3.2) используется вместе с методом открытия графических файлов `OpenPictureDialog1`, отображающим диалоговое окно **Открытие файла**. Внешний вид диалогового окна **Открытие файла** представлен на рис. 3.1.

Листинг 3.2. Пример использования метода *LoadFromFile*

```
procedure TForm1.Button1Click(Sender: TObject);
var BitMap1,BitMap2 : TBitMap;
begin
  if OpenPictureDialog1.Execute then begin
    BitMap2 := TBitMap.Create;
    BitMap1 := TBitMap.Create;
    try
      BitMap1.LoadFromFile(OpenPictureDialog1.FileName);
      BitMap2.Assign(BitMap1);
      // копирует BitMap1 в BitMap2
      BitMap2.Dormant;
      // освобождает ресурсы GDI
      BitMap2.FreeImage;
      // освождает память
      Canvas.Draw(20,20,BitMap2);
      BitMap2.Monochrome := true;
      Image1.Canvas.Draw(80,80,BitMap2);
      BitMap2.ReleaseHandle;
    finally
      BitMap1.Free;
      BitMap2.Free;
    end;
  end;
end;
```

Обратите внимание на то, что переменная `BitMap: TBitMap` захватывает много памяти и, поэтому, как только необходимость в ней пропадет, нужно срочно освободить память методом `BitMap.Free`.

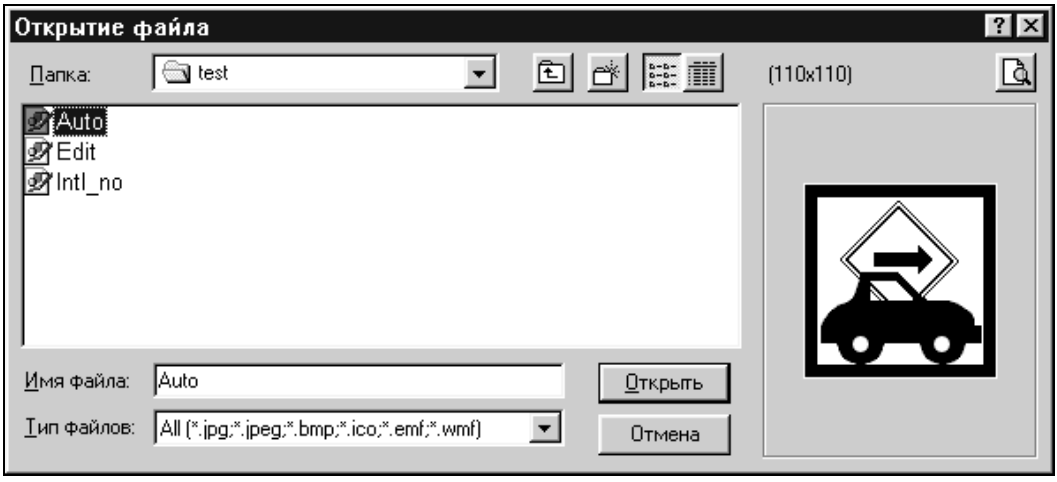


Рис. 3.1. Диалоговое окно **Открытие файла**

- Метод `SaveToFile` записывает объект `Graphic` в файл.

  ```
  procedure SaveToFile (const Filename: string); virtual
  ```

  Обычно сохранение информации в графический файл реализуется с использованием метода `SavePictureDialog1`, отображающего диалоговое окно **Сохранение файла**, например:

  ```
  if SavePictureDialog1.Execute then
    BitMap1.SaveToFile(SavePictureDialog1.FileName).
  ```

  Внешний вид диалогового окна **Сохранение файла** представлен на рис. 3.2.

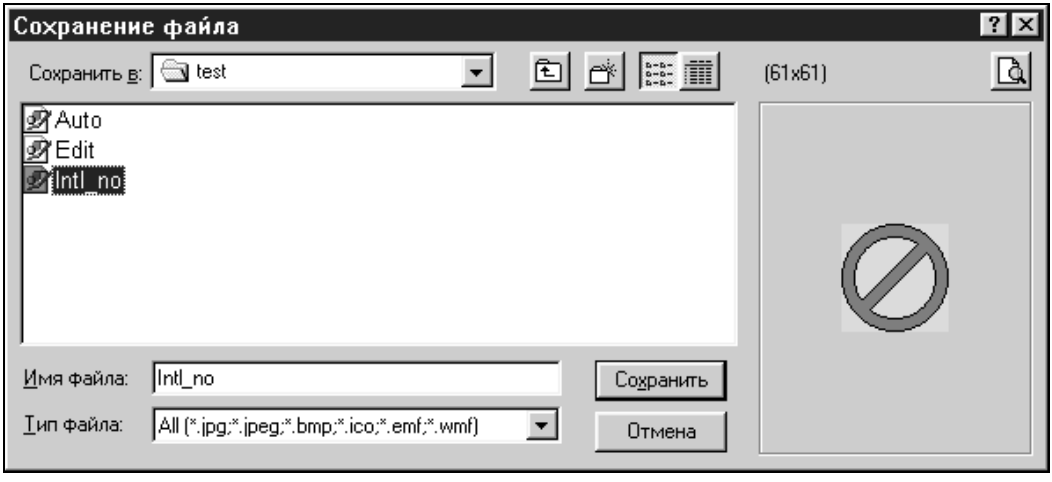


Рис. 3.2. Диалоговое окно **Сохранение файла**

- Виртуальный абстрактный метод `LoadFromStream` загружает графический объект из потока.

  ```
  procedure LoadFromStream (Stream: TStream) ; virtual; abstract
  ```

- Виртуальный абстрактный метод `SaveToStream` сохраняет объект в поток.

  ```
  procedure SaveToStream (Stream: TStream) ; virtual; abstract
  ```

Два метода: `LoadFromClipboardFormat` и `SaveToClipboardFormat` осуществляют взаимодействие с буфером обмена (`clipboard`).

- Метод `LoadFromClipboardFormat` заменяет текущее изображение данными из буфера обмена. Если `TGraphic` не поддерживает формат данных из буфера обмена, то возникает состояние ошибки.

  ```
  procedure LoadFromClipboardFormat (AFormat: Word; AData: THandle; APa-
  lette: HPALETTE).
  ```

- Метод `SaveToClipboardFormat` преобразует изображение к формату буфера обмена. Если изображение не поддерживается буфером обмена, то возникает состояние ошибки.

  ```
  procedure SaveToClipboardFormat (var AFormat: Word; var AData: THan-
  dle; var APalette: HPALETTE)
  ```

- Свойство `Empty` устанавливается равным `true`, если графический объект не содержит данные.

  ```
  property Empty: Boolean
  ```

- В свойстве `Height`, как обычно, содержится высота графического объекта в пикселах.

  ```
  property Height: Integer
  ```

- Свойство `Modified` устанавливается равным `true` внутри обработчика события `OnChange`, если графический объект модифицировался.

  ```
  property Modified: Boolean
  ```

- Свойство `Palette` указывает цветную палитру графического изображения, т. е. позволяет управлять цветами битовой карты. Понятие палитры определено для изображений, использующих до 256 цветов.

  ```
  property Palette: HPALETTE
  ```

  Если графика не нужна или не использует палитру, свойство `Palette` равно нулю.

- Свойство `PaletteModified` используется для того, чтобы определить, изменилась ли палитра. `PaletteModified` меняется при событии `OnChange`.

  ```
  property PaletteModified: Boolean
  ```

  Данное свойство остается равным `true`, пока какой-нибудь компонент, ответственный за реализацию этой новой палитры, например, `Timage`, не установит его равным `false`.

- Свойство `Transparent` указывает, закрывает ли изображение всю свою прямоугольную область.

  ```
  property Transparent: Boolean
  ```

Если при рисовании изображения требуется не выводить часть точек одного цвета (сделать их прозрачными), то необходимо установить свойство `Bitmap.TransparentMode:=tmAuto`, назначить прозрачный цвет `Bitmap.TransParentColor` (в примере, который приведен далее, это цвет точки с координатами `50,50`) и установить флаг прозрачности `Transparent := true` (см. листинг 3.3).

Используйте свойство `Transparent` для вывода прозрачной графики. Некоторые потомки `TGraphic` типа `TIcon` и `TMetafile` всегда прозрачны (в качестве прозрачного цвета используется цвет левой нижней точки), так что переустановка свойства для этих объектов не изменяет их поведение. Однако рисунок на `TBitmap` зависит от этого свойства. Компонент `TImage` имеет такое же значение свойства `Transparent`.

Листинг 3.3. Пример использования метода *Transparent*

```
procedure TForm1.Button1Click(Sender: TObject);
var Bitmap : TBitMap;
begin
  Bitmap := TBitmap.Create;
  try
    with Bitmap do begin
      LoadFromFile('factory.bmp');
      Transparent := true;
      TransParentColor := BitMap.Canvas.Pixels[50,50];
      Form1.Canvas.Draw(100,100,BitMap);
      TransparentMode := tmAuto;
      // прозрачный цвет по умолчанию устанавливается
      // в clDefault = TColor($20000000);
      Form1.Canvas.Draw(50,50,BitMap);
    end;
  finally
    Bitmap.Free;
  end;
end;
```

- Свойство `Width` отвечает за ширину графического объекта в пикселах.

  ```
  property Width: Integer
  ```

- Событие `OnProgress` происходит, когда графическое изображение находится в процессе изменения.

  ```
  property OnProgress: TProgressEvent;
  type TProgressEvent = procedure (Sender: TObject; Stage: TProgressStage; PercentDone: Byte; RedrawNow: Boolean; const R: TRect; const Msg: string) of object;
  TProgressStage = (psStarting, psRunning, psEnding);
  ```

Для некоторых потомков TGraphic событие OnProgress существенно при медленных процессах типа загрузки, сохранения или преобразования данных изображения. Оно позволяет приложениям обеспечить обратную связь с пользователем (как правило, в виде индикаторов) при медленных процессах.

Пишущие компоненты могут генерировать события OnProgress для потомков TGraphic, вызывая защищенный метод Progress. Эти события распространяются на объекты TImage и TPicture.

Параметр Stage указывает, начинается ли действие (psStarting), продолжается ли оно (psRunning) или заканчивается (psEnding). Если процедура обработки события OnProgress использует индикатор выполнения, последний может быть создан при Stage = psStarting, меняться при Stage = psRunning и удаляться при Stage = psEnding. Индикатор PercentDone удобно использовать как параметр, показывающий процент выполнения.

Параметр RedrawNow указывает, возможен ли вывод изображения. Параметр R типа TRect определяет ту часть изображения, которая изменилась и должна быть перерисована.

Параметр Msg содержит строку, которая описывает происходящее действие. Например, значением Msg могла бы быть строка 'Loading', 'Storing' или 'Reducing'. Строка Msg может быть пустой.

В примере, приведенном далее (листинг 3.4), при работе с компонентом Image1 вызывается метод ProgressUpdate, который в зависимости от значения параметра Stage или делает индикатор видимым Gauge1.Visible:=true, или изменяет значение индикатора Gauge1.Progress:=PercentDone, или прячет индикатор Gauge1.Visible:=false.

Листинг 3.4. Пример использования свойства *OnProgress*

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Image1.OnProgress := ProgressUpdate;
end;

procedure TForm1.ProgressUpdate(Sender: TObject; Stage:
       TProgressStage; PercentDone:Byte; RedrawNow:Boolean;
       const R:TRect; const Msg: string);
begin
  case Stage of
    psStarting : Gauge1.Visible:=true;
    psRunning  : Gauge1.Progress:=PercentDone;
    psEnding   : Gauge1.Visible:=false;
  end;
end;
```

3.2. Класс *TPicture*

Класс `TPicture = class(TPersistent)` является контейнером класса `TGraphic`, имея соответствующее свойство — объект `Graphic` [79]. Этот класс используется вместо класса `TGraphic` при работе с графикой, принадлежащей любому классу — наследнику `TGraphic`. Например, если `TPicture` содержит битовое изображение, то графику определяет свойство `Bitmap`. Если `TPicture` содержит пиктограмму, то графику определяет свойство `Icon`. Если же `TPicture` содержит графику метафайла, то графику определяет свойство `Metafile`. Свойства `TPicture` указывают тип графики и ее размер. Методы `TPicture` используются для загрузки, сохранения и манипулирования графикой и предназначены для управления вызовами соответствующих методов.

В листинге 3.5 приведено описание доступных методов и свойств класса `TPicture`.

Листинг 3.5. Описание класса *TPicture*

```
TPicture = class(TPersistent)
public
  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);
  class function SupportsClipboardFormat(AFormat: Word): Boolean;
  procedure Assign(Source: TPersistent); override;
  class procedure RegisterFileFormat(const AExtension,
    ADescription: string; AGraphicClass: TGraphicClass);
  class procedure RegisterFileFormatRes(const AExtension: String;
    ADescriptionResID: Integer; AGraphicClass: TGraphicClass);
  class procedure RegisterClipboardFormat(AFormat:
    Word; AGraphicClass: TGraphicClass);
  class procedure UnregisterGraphicClass(AClass: TGraphicClass);
  property Bitmap: TBitmap
  property Graphic: TGraphic
  property PictureAdapter: IChangeNotifier
  property Height: Intege
  property Icon: TIcon
  property Metafile: TMetafile
  property Width: Integer
  property OnChange: TNotifyEvent
  property OnProgress: TProgressEventgress;
end;
```

- Виртуальный метод `Assign` определен в классе-предке `TPersistent` и используется для передачи свойств или других атрибутов объекта `Source` вызывающему объекту.

  ```
  procedure Assign (Source: TPersistent)
  ```

  В классе `TPicture` метод предка переопределяется, делая возможным полиморфное присваивание графических объектов. Действия, выполненные `Assign`, зависят от фактических типов свойств `Graphic` и `Source`. Например, если свойства `Graphic` и `Source` — битовая карта (`TBitmap`), то битовая карта, содержащаяся в `Source`, копируется в `Graphic`. Такие же преобразования будут выполнены, например, для `TIcon` или `TMetafile`. Если параметр `Source` равен `nil`, то поле `Graphic` очищается с удалением прежнего объекта.

  Типичное использование метода `Assign`: `DestObject.Assign(Source)`. При этом объекту `DestObject` передается содержимое объекта `Source`. В качестве примера можно привести оператор, копирующий одно битовое изображение `BitMap1` в другое — `BitMap2: BitMap2.Assign(BitMap1)`. Другим примером может служить загрузка изображения из буфера: `Image1.Picture.Assign(Clipboard)`.

- Метод `LoadFromFile` используется для чтения изображения с диска.

  ```
  procedure LoadFromFile (const Filename: string)
  ```

  Класс `TGraphic` узнает определенные файловые расширения. Если файловое расширение не распознано, то возникает исключительная ситуация `EInvalidGraphic`. Стандартными расширениями являются ICO, WMF, BMP и JPG.

- Метод `SaveToFile` сохраняет графику в файле `Filename`, вызывая соответствующий метод объекта `Graphic`.

  ```
  procedure SaveToFile (const Filename: string)
  ```

- Метод `LoadFromClipboardFormat` используется для чтения графических данных из буфера обмена.

  ```
  procedure LoadFromClipboardFormat (AFormat: Word; AData: THandle; APa-
  lette: HPALETTE)
  ```

  Если формат не поддерживается, то возникает исключительная ситуация `EInvalidGraphic`. Зарегистрированные форматы: `CF_BITMAP` для битовых изображений, `CF_METAFILEPICT` и `CF_ENHMETAFILE` для метафайлов [85].

- Метод `SaveToClipboardFormat` используется для сохранения графики в буфере обмена в формате, поддерживаемом буфером обмена (`CF_BITMAP` для файлов формата BMP, `CF_METAFILE` и `CF_ENHMETAFILE` для метафайлов).

  ```
  procedure SaveToClipboardFormat (var AFormat: Word; var AData: THan-
  dle; var APalette: HPALETTE)
  ```

- Метод класса `SupportsClipboardFormat` возвращает значение `true`, если формат `AFormat` зарегистрирован в системе и поддерживается классом `TPicture`.

  ```
  class function SupportsClipboardFormat (AFormat: Word) : Boolean
  ```

- Метод `RegisterFileFormat` регистрирует новый класс `TGraphic` для использования в `LoadFromFile`.

  ```
  class procedure RegisterFileFormat (const AExtension, ADescription: string; AGraphicClass: TGraphicClass)
  ```

  Параметр `AExtension` определяет расширение имени файла. Параметр `Adescription` — условное название графики. Параметр `AGraphicClass` — имя нового графического класса.

- Методы `RegisterFileFormatRes` и `RegisterClipboardFormat` регистрируют новый класс `TGraphic` для использования в `LoadFromClipboardFormat`.

  ```
  class procedure RegisterFileFormatRes (const AExtension: String; ADescriptionResID : Integer; AGraphicClass: TGraphicClass);
  class procedure RegisterClipboardFormat(AFormat: Word; AGraphicClass: TGraphicClass)
  ```

  Оба этих метода предназначены для создания новых графических классов. Они позволяют зарегистрировать формат файла для буфера обмена и связать его с созданным классом — потомком `TGraphic`, который умеет читать и записывать информацию в этом формате. Параметры процедур такие же, как у метода `RegisterFileFormat`.

- Метод `UnregisterGraphicClass` отменяет регистрации классов. Он удаляет все ссылки на указанный класс `TGraphic` и всех его потомков из внутренних списков форматов файлов и форматов буфера обмена (`clipboard`).

  ```
  class procedure UnregisterGraphicClass (AClass: TGraphicClass)
  ```

  Метод `UnregisterGraphicClass` вызывается для того, чтобы сделать графический класс недоступным для всех объектов, обладающих канвой. `UnregisterGraphicClass` полностью отменяет регистрацию, выполненную методами `RegisterFileFormat`, `RegisterFileFormatRes` или `RegisterClipboardFormat`.

  Если графический класс зарегистрирован, то глобальные функции `GraphicFilter`, `GraphicExtension` и `GraphicFileMask` могут возвращать строки для фильтра диалога, расширения файла по умолчанию или регистрировать фильтры для графического класса. Вызывайте метод `UnregisterGraphicClass`, если эти значения должны стать недоступными.

- Свойства `Width` и `Height`, описанные ранее в *разд. 3.1*, доступны и для класса `TPicture`. Эти свойства определяют ширину и высоту картинки в пикселах.

  ```
  property Width: Integer
  property Height: Integer
  ```

- Свойство `Graphic` используется для конкретизации информации о том, какой именно графический объект содержит `TPicture`.

  ```
  property Graphic: TGraphic
  ```

  В качестве `Graphic` могут быть классы `Bitmap`, `Icon`, `Metafile` или определенный пользователем графический класс.

В следующем примере в верхнем левом углу холста Form1 рисуется графика из Picture1:

```
Form1.Canvas.Draw(0,0 Picture1.Graphic).
```

- Свойство Bitmap определяет содержание объекта с канвой как графическую битовую карту (формат BMP).

  ```
  property Bitmap: TBitmap
  ```

  Это свойство используется для того, чтобы сослаться на объект с канвой, если он содержит битовую карту (bitmap). Если Bitmap используется для изображения, содержащего Metafile или ICO, то графика не будет преобразована. Первоначальное изображение удаляется, а Bitmap возвращает новую чистую битовую карту.

- Свойство Icon определяет содержание объекта TPicture как графику Icon (формат ICO).

  ```
  property Icon: Ticon
  ```

  Если используется свойство Icon, когда TPicture содержит графику Bitmap или Metafile, преобразований не проводится. Первоначальное содержание TPicture очищается, и Icon возвратит новое незаполненное изображение.

- Свойство Metafile определяет содержание объекта TPicture как графику Enhanced Windows metafile (формат EMF).

  ```
  property Metafile: TMetafile
  ```

  Если используется свойство Metafile, когда TPicture содержит графику Bitmap или Icon, преобразований не проводится. Первоначальное содержание TPicture будет очищено, и свойство Metafile возвратит новое незаполненное изображение.

- Свойство PictureAdapter представляет интерфейс OLE для изображения. Это свойство предназначено только для внутреннего использования.

  ```
  property PictureAdapter: IChangeNotifier
  ```

- Свойство OnChange вызывается при изменениях графического объекта.

  ```
  property OnChange: TNotifyEvent
  ```

- Описанное ранее в *разд. 3.1* свойство OnProgress доступно и для класса Tpicture.

  ```
  property OnProgress: TProgressEventgress;
  TProgressStage = (psStarting, psRunning, psEnding);
  TProgressEvent = procedure (Sender: TObject; Stage: TProgressStage;
  PercentDone: Byte; RedrawNow: Boolean; const R: TRect; const Msg:
  string) of object;
  ```

3.3. Класс *TBitmap*

Рассмотрим теперь предопределенные графические классы.

Формат файлов Windows Bitmap является родным растровым форматом для Windows (файлы с расширением BMP или DIB). Более подробно формат BMP описан в *главе 11*.

Класс `TBitmap` — инкапсуляция битовой карты (`HBITMAP`) и палитры (`HPALETTE`). Этот класс содержит внутреннее изображение битовой карты (`bitmap`) графики и автоматически управляет реализацией палитры. `TBitmap` — мощный графический объект, используемый для создания, манипулирования (масштабирования, прокрутки, вращения и закраски) и хранения изображения в виде файла на диске.

Для рисования битового изображения на холсте вызывайте методы `Draw` или `StretchDraw` объекта `TCanvas`, передавая `TBitmap` как параметр.

Создание копий `TBitmap` происходит очень быстро, так как указатель копируется быстрее, чем изображение. Если изображение изменено и указатель разделен более чем одним объектом `TBitmap`, то изображение скопируется прежде, чем будет выполнена модификация.

В этом классе перекрываются методы `Assign`, `LoadFromClipboardFormat`, `LoadFromStream`, `SaveToClipboardFormat`, `SaveToStream`.. Объект взаимодействует с буфером обмена в формате `CF_BITMAP`. В листинге 3.6 приводится описание доступной части класса.

Листинг 3.6. Описание класса *TBitmap*

```
TBitmap = class(TGraphic)
public
  procedure Assign(Source: TPersistent); override;
  procedure Dormant;
  procedure FreeImage;
  procedure LoadFromClipboardFormat(AFormat: Word;
    AData: THandle; APalette: HPALETTE); override;
  procedure LoadFromStream(Stream: TStream); override;
  procedure LoadFromResourceName(Instance: THandle;
    const ResName: String);
  procedure LoadFromResourceID(Instance: THandle; ResID: Integer);
  procedure Mask(TransparentColor: TColor);
  function ReleaseHandle: HBITMAP;
  function ReleaseMaskHandle: HBITMAP;
  function ReleasePalette: HPALETTE;
  procedure SaveToClipboardFormat(var Format: Word;
    var Data: Thandle;   var APalette: HPALETTE); override;
  procedure SaveToStream(Stream: TStream); override;
  property Canvas: TCanvas;
  property Handle: HBITMAP;
  property Palette: HPALETTE;
  property HandleType: TBitmapHandleType;
  property IgnorePalette: Boolean;
  property MaskHandle: HBITMAP;
  property Monochrome: Boolean;
  property PixelFormat: TPixelFormat;
```

```
  property ScanLine[Row: Integer]: Pointer;
  property TransparentColor: TColor;
  property TransparentMode: TTransparentMode;
end;
```

При обращении к дескриптору битовой карты и любой попытке рисовать на ее канве, разделение одного изображения несколькими объектами прерывается, и каждый объект получает свою отдельную копию содержимого дескриптора. У класса `TBitmap` есть свои дополнительные методы. Рассмотрим их описание.

- Метод `Dormant` (`procedure Dormant`) используется для уменьшения объема ресурсов *интерфейса графических устройств* — GDI (Graphics Device Interface), захваченных приложением под битовое изображение. Этот метод создает образ битового изображения в памяти. Он сохраняет образ, при этом битовое изображение освобождает указатель `HBITMAP`, доступный через свойство `Handle`.
- Метод `FreeImage` (`procedure FreeImage`) используется для того, чтобы уменьшить объем памяти, занятой битовым изображением. Когда битовое изображение загружается, Delphi создает изображение в памяти. Если битовое изображение не изменяется, то изображение в памяти используется при сохранении битового изображения для проверки (см. листинг 3.2).
- Метод `LoadFromResourceName` загружает битовую карту из ресурса с именем `ResName` в объект `bitmap` (листинг 3.7).

  ```
  procedure LoadFromResourceName (Instance: THandle; const ResName: String)
  ```

Листинг 3.7. Пример использования метода *LoadFromResourceName*

```
procedure TForm1.Button1Click(Sender: TObject);
var BitMap1 : TBitMap;
begin
  BitMap1 := TBitMap.Create;
  try
    BitMap1.LoadFromResourceName(HInstance,'THEBITMAP');
    Canvas.Draw(12,12,BitMap1);
  finally
    BitMap1.Free;
  end;
end;
```

- Метод `LoadFromResourceID` загружает битовую карту из ресурса с номером `ResID` в объект `bitmap`.

  ```
  procedure LoadFromResourceID (Instance: THandle; ResID: Integer)
  ```

- Метод `Mask` конвертирует битовую карту в одноцветное изображение, заменяя прозрачный цвет `TransparentColor` белым, а все другие цвета — черным.

  ```
  procedure Mask (TransparentColor: TColor)
  ```

Используйте `Mask` для того, чтобы заготовить одноцветную маску битового изображения, основанную на данном прозрачном цвете `TransparentColor`. Метод `Mask`, в отличие от свойства `MaskHandle`, описанного далее, заменяет изображение в объекте `TBitmap`, поэтому можно вызывать `Mask` только однажды. Если необходим указатель на другую битовую карту с изображением маски, используйте `MaskHandle`.

- Метод `ReleaseHandle` возвращает дескриптор битового изображения и обнуляет его.

  ```
  function ReleaseHandle: HBITMAP
  ```

- Канва битовой карты доступна через свойство `Canvas`.

  ```
  property Canvas: TCanvas
  ```

  Другие потомки `TGraphic` доступной канвы не имеют. С ее помощью можно быстро рисовать на поверхности изображения. Именно это свойство используется в примере деформации прямоугольника с изображением в произвольный четырехугольник (*см. гл. 9*).

- Свойство `Handle` — дескриптор битовой карты.

  ```
  property Handle: HBITMAP
  ```

- Свойство `Palette` (дескриптор палитры) позволяет управлять набором цветов битовой карты.

  ```
  property Palette: HPALETTE
  ```

  `Palette` битовой карты содержит до 256 цветов, используемых для вывода изображения на экран.

- Свойство `HandleType` используется для определения или изменения типа битовой карты.

  ```
  property HandleType: TbitmapHandleType;
  TBitmapHandleType = (bmDIB, bmDDB);
  ```

  Свойство `HandleType` указывает, используется ли битовая карта типа `DDB` (Device Dependent Bitmap — зависимая от устройства битовая карта) или типа `DIB` (Device Independent Bitmap — независимая от устройства битовая карта).

- Свойство `IgnorePalette` определяет, использует ли битовая карта палитру при рисовании изображения.

  ```
  property IgnorePalette: Boolean
  ```

  Можно изменять свойство `IgnorePalette` для повышения скорости вывода рисунка. При `IgnorePalette=true` битовая карта не использует палитру при рисовании, что приводит к более низкому качеству изображения на 256-цветных видеодрайверах, но к более быстрому выводу изображения.

- Свойство `MaskHandle` используется для того, чтобы вызывать API-функции Windows, которые требуют указателя на объект `bitmap`. Можно передать этим функциям `MaskHandle` как параметр, указывающий на `bitmap`.

  ```
  property MaskHandle: HBITMAP
  ```

- Свойство `Monochrome` определяет, какая битовая карта выводится — одноцветная (монохромная) или цветная.

  ```
  property Monochrome: Boolean
  ```

  Если это свойство имеет значение `true`, то битовая карта одноцветна. Значение `false` соответствует цветной битовой карте.

Следующий пример (листинг 3.8) создает `Bitmap1` и устанавливает свойство `Monochrome` равным `true`.

Листинг 3.8. Пример использования свойства *Monochrome*

```
var  Bitmap1: TBitmap;
begin
  Bitmap1 := TBitmap.Create;
  Bitmap1.Monochrome := true;
end;
```

- Свойство `PixelFormat` определяет число битов, используемых при форматировании изображения.

  ```
  property PixelFormat: TPixelFormat;
  type TPixelFormat = (pfDevice, pf1bit, pf4bit, pf8bit, pf15bit,
  pf16bit, pf24bit, pf32bit, pfCustom);
  ```

  Используйте свойство `PixelFormat`, например, для того, чтобы установить восьмибитовый формат изображения битовой карты для видеодрайверов, которые не могут показывать 24-битовый формат изображения битовой карты. Свойство `PixelFormat` изображения формата `JPEG` обратится к битовой карте, если изображение `JPEG` скопировано в нее.

- Свойство `ScanLine` обеспечивает быстрый индексированный доступ к любой линии пикселов.

  ```
  property ScanLine [Row: Integer]: Pointer
  ```

  Это свойство используется только с битовой картой типа `DIB` (Device Independent Bitmaps) для инструментов редактирования изображения, которые выполняют низкоуровневую работу с пикселами (см. листинг 3.9).

Листинг 3.9. Пример использования метода *ScanLine*

```
procedure TForm1.Button1Click(Sender: TObject);
var  x,y    : integer;
     BitMap : TBitMap;
     P      : PByteArray;
begin
  BitMap := TBitMap.create;
  try
    BitMap.LoadFromFile('factory.bmp');
    for y := 0 to BitMap.height -1 do begin
```

```
      P := BitMap.ScanLine[y];
      for x := 0 to BitMap.width -1 do  P[x] := y;
    end;
    Canvas.draw(0,0,BitMap);
  finally
    BitMap.free;
  end;
end;
```

В этом примере перебираются все точки рисунка из файла factory.BMP и точки каждой строки закрашиваются в свой цвет. Поведение программы с данным кодом зависит от установленного графического режима: если установлен 8-битовый режим, то будут перебираться все точки строки (на каждый пиксел отводится по 1 байту), в 24-битовом режиме на каждый пиксел выделяется 3 байта и, поэтому, цикл `for x := 0 to BitMap.width -1 do` пройдет только треть ширины картинки. При этом для любой точки RGB-составляющие цвета получают одинаковые значения, и пикселы окрашиваются в оттенки серого цвета.

- Свойство `TransparentColor` используется для назначения прозрачного цвета битового изображения.

  ```
  property TransparentColor: TColor
  ```

  Если свойство `TransparentMode` равно `tmAuto`, то свойство `TransparentColor` назначает прозрачным цвет нижнего левого пиксела битового изображения. Если `TransparentMode=tmFixed`, то свойство прозрачного цвета `TransparentColor` может быть назначено произвольно.

- Свойство `TransparentMode` определяет, вычисляется ли значение свойства `TransparentColor` автоматически или определяется битовой картой.

  ```
  property TransparentMode: TTransparentMode;
  type TTransparentMode = (tmAuto, tmFixed);
  ```

  Если свойство `TransparentMode` имеет значение `tmAuto`, то свойство `TransparentColor` возвратит цвет нижнего левого пиксела изображения битовой карты. Если же `TransparentMode` имеет значение `tmFixed`, свойство `TransparentColor` обращается к цвету, сохраненному в объекте битовой карты.

В листинге 3.10 приведен пример построения прозрачного изображения с использованием свойств `Transparent`, `TransparentColor` и `TransparentMode`.

Листинг 3.10. Пример рисования прозрачного изображения

```
procedure TForm1.Button1Click(Sender: TObject);
var Bitmap : TBitMap;
begin
  Bitmap := TBitmap.Create;
  try
    with Bitmap do begin
      LoadFromFile('factory.bmp');
```

```
      Transparent := true;
      TransparentColor := BitMap.canvas.pixels[50,50];
      Form1.Canvas.Draw(0,0,BitMap);
      TransparentMode := tmAuto;
      Form1.canvas.Draw(50,50,BitMap);
    end;
  finally
    Bitmap.Free;
  end;
end;
```

3.4. Класс *TMetafile*

Файлы в формате Microsoft Windows Metafile (файлы с расширением WMF, EMF, WMD) используются для хранения векторных и растровых изображений в оперативной памяти или на диске с последующим их воспроизведением на устройстве вывода [85]. Данный формат может поддерживать максимум 24-битовые цвета изображений.

Для описания метафайлов введен класс `TMetafile` (листинг 3.11).

Листинг 3.11. Описание класса *TMetafile*

```
TMetafile = class(TGraphic)
public
  procedure Clear;
  procedure LoadFromStream(Stream: TStream); override;
  procedure SaveToFile(const Filename: String); override;
  procedure SaveToStream(Stream: TStream); override;
  procedure LoadFromClipboardFormat(AFormat: Word;
    AData: THandle; APalette: HPALETTE); override;
  procedure SaveToClipboardFormat(var AFormat: Word;
   var AData: THandle; var APalette: HPALETTE); override;
  procedure Assign(Source: TPersistent); override;
  function ReleaseHandle: HENHMETAFILE;
  property CreatedBy: String;
  property Description: String;
  property Enhanced: Boolean;
  property Handle: HENHMETAFILE;
  property MMWidth: Integer;
  property MMHeight: Integer;
  property Inch: Word;
end;
```

Класс `TMetafile` инкапсулирует свойства метафайла Windows. В нем перекрываются методы `Assign`, `LoadFromStream`, `SaveToStream`, `LoadFromClipboardFormat`, `SaveToClipboardFormat`. В буфер обмена объект помещает свое содержимое в формате `CF_METAFILEPICT`.

Помимо общих, класс имеет следующие собственные свойства и методы.

- Метод `Clear` (`procedure Clear`) удаляет изображение метафайла. Используйте метод `Clear` для того, чтобы удалить старое изображение метафайла и создать новое.
- Метод `ReleaseHandle` предназначен для того, чтобы освободить использованные ресурсы, представленные метафайлу.

  ```
  function ReleaseHandle: HENHMETAFILE
  ```

  Метод `ReleaseHandle` устанавливает свойство `Handle` равным `nil`. Для того чтобы ресурсы, связанные с метафайлом, не были потеряны, вызовите `ReleaseHandle` раньше, чем установите свойство `Handle`.
- Свойство `CreatedBy` представляет собой необязательное имя автора или приложения, которое создало метафайл.

  ```
  property CreatedBy: String
  ```
- Свойство `Description` представляет собой необязательное текстовое описание метафайла.

  ```
  property Description: String
  ```

Можно устанавливать свойства `CreatedBy` и `Description` нового метафайла вызовом метода `TMetafileCanvas.CreateWithComment`.

- Свойство `Enhanced` определяет, как метафайл будет сохранен на диске.

  ```
  property Enhanced: Boolean
  ```

  Если `Enhanced = true` (по умолчанию), то метафайл сохраняется в формате EMF (Win32 Enhanced Metafile), если же `Enhanced = false`, то метафайл сохраняется в формате WMF (Windows 3.1 Metafile). Формат в памяти — всегда EMF. При сохранении файла в формате WMF будет теряться информация, которая была бы сохранена в формате EMF.

  Это свойство позволяет поддерживать совместимость файлов 16-ти разрядного Delphi, если файл загружен в формате WMF, а затем сохраняется в формате EMF.
- Свойства `MMWidth` и `MMHeight` показывают ширину и высоту изображения в сотых долях миллиметра. Эти свойства масштаба используются EMF-метафайлом.

  ```
  property MMWidth: Integer
  property MMHeight: Integer
  ```
- Свойство `Inch` показывает размеры в дюймах.

  ```
  property Inch: Word
  ```

Это устаревшее свойство используется для изменения масштаба при записи в формате WMF. Метафайлы формата EMF поддерживают полную информацию о масштабе.

- Свойство `Handle` представляет собой дескриптор метафайла.

```
property Handle: HMETAFILE
```

3.5. Класс *TIcon*

Класс `TIcon` инкапсулирует пиктограмму Windows. В листинге 3.12 приводится описание этого класса.

Листинг 3.12. Описание класса *TIcon*

```
TIcon = class(TGraphic)
public
  procedure Assign(Source: TPersistent); override;
  procedure LoadFromClipboardFormat(AFormat: Word;
    AData: THandle; APalette: HPALETTE); override;
  procedure LoadFromStream(Stream: TStream); override;
  function ReleaseHandle: HICON;
  procedure SaveToClipboardFormat(var Format: Word;
    var Data: THandle; var APalette: HPALETTE); override;
  procedure SaveToStream(Stream: TStream); override;
  property Handle: HICON;
end;
```

Размеры пиктограмм постоянны и равны 32×32 пиксела. Они устанавливаются с помощью вызова функций `GetSystemMetrics(SM_CXICON)` и `GetSystemMetrics(SM_CYICON)`. При попытке изменить размеры пиктограммы возникает исключительная ситуация `EInvalidGraphicOperation`. В классе `TIcon` перекрываются методы класса `TGraphic`: `LoadFromClipboardFormat`, `SaveToClipboardFormat`, `Assign`, `LoadFromStreain` и `SaveToStream`.

Дополнительно также определены следующие свойства и методы.

- Свойство `Handle` представляет собой дескриптор пиктограммы.

```
property Handle: HICON
```

- Метод `ReleaseHandle` вызывается для того, чтобы освободить ресурсы, использованные для представления изображения. Этот метод устанавливает свойство `Handle` равным `nil`.

```
function ReleaseHandle: HICON
```

3.6. Функции для работы с графикой

В заключение, приведем описание некоторых функций, включенных в модуль Graphics [64].

- Функция GraphicFilter возвращает значения для свойства Filter диалогов OpenPictureDialog или SavePictureDialog.

  ```
  function GraphicFilter (GraphicClass: TGraphicClass): string
  ```

  Параметр GraphicClass может принимать одно из следующих значений: TBitmap, TGraphic, TIcon, TMetafile, TJPEGImage или показывать на другой, определенный пользователем, потомок класса TGraphic. Функция возвращает свою строку для каждого встроенного графического класса (табл. 3.1).

Таблица 3.1. *Строки для функции Filter*

Графический класс	Возвращаемая строка для функции Filter
Tbitmap	Bitmaps (*.bmp)|*.bmp
TIcon	Icons (*.ico)|*.ico
TMetafile	All(*.emf;*.wmf)|*.emf;*.wmf|Enhanced Metafiles (*.emf)|*.emf|Metafiles(*.wmf)|*.wmf
TJPEGImage	All(*.jpeg;*.jpg)|*.jpeg;*.jpg|JPEG Image File (*.jpeg)|*.jpeg|JPEG Image File (*.jpg)|*.jpg
TGraphic	All (*.jpeg;*.jpg;*.bmp;*.ico;*.emf;*.wmf)|*.jpeg;*.jpg; *.bmp;*.ico;*.emf; *.wmf|JPEG Image File (*.jpeg)|*.jpeg|JPEG Image File (*.jpg)|*.jpg|Bitmaps (*.bmp)|*.bmp|Icons (*.ico)|*.ico|Enhanced Metafiles (*.emf)|*.emf|Metafiles(*.wmf)|*.wmf

  Расширения для графики JPEG доступны, если модуль JPEG включен в проект.

- Функция GraphicExtension вызывается для того, чтобы получить расширения имен файлов для свойства DefaultExt компонента SavePictureDialog.

  ```
  function GraphicExtension (GraphicClass: TGraphicClass): string
  ```

  Эта функция возвращает встроенные расширения для каждого типа графического объекта, определенного параметром GraphicClass.

  В табл. 3.2 представлены файловые расширения, возвращаемые для каждого встроенного графического класса.

Таблица 3.2. *Расширения имен графических файлов*

Графический класс	Расширения имен файлов	Графический класс	Расширения имен файлов
Tbitmap	BMP	Tmetafile	EMF
Ticon	ICO	TJPEGImage	JPG

Если новые пользовательские графические классы регистрируют свое файловое расширение, то функция GraphicExtension возвратит это расширение, при условии, что пользовательский графический класс указан как аргумент функции.

- Функция ColorToRGB возвращает RGB-представление цвета для использования в API-функциях.

```
function ColorToRGB (Color: TColor): Longint
```

- Функция ColorToIdent возвращает имя строковой константы, которая представляет цвет Color, например, clBlack или clWindow. Полный список строковых имен цветов приведен в *главе 2* (листинг 2.1). Параметр Ident возвращает строковое имя цвета. Если нет строковой константы для указанного цвета, то функция ColorToIdent вернет значение false.

```
function ColorToIdent (Color: Longint; var Ident: string): Boolean
```

- Функция IdentToColor обратна функции ColorToIdent.

```
function IdentToColor (const Ident: string; var Color: Longint): Boolean
```

Эта функция полезна для преобразования строк, сгенерированных процедурой GetColorValues, в значения типа TColor. Функция IdentToColor возвращает значение true, если параметр Ident успешно преобразовывается в цвет. IdentToColor возвращает значение false, если параметр Ident не является именем одной из встроенных цветовых констант.

Примечание

Если идентификатор строки для цвета представляет собой шестнадцатеричную величину или цветовую константу, используйте вместо этой функции функцию StringToColor.

- Функция ColorToString используется для того, чтобы получить строку, которая соответствует цвету Color.

```
function ColorToString (Color: Tcolor): string
```

Если есть строковая константа, определенная для цвета (например, clBlack или clWindow), то ColorToString возвращает это имя. В противном случае, ColorToString возвращает шестнадцатеричный номер цвета, сформатированный как строка.

- Функция StringToColor восстанавливает перевод, выполненный функцией ColorToString.

```
function StringToColor (S: string): TColor
```

Эта функция полезна для преобразования строк, введенных пользователем, в пригодные величины типа TColor.

Параметр S может быть также именем встроенной цветовой константы, например, 'clBtnFace', или правильным числовым представлением строковой величины типа TColor, например, '$02FF8800'. Функция StringToColor возвращает величину типа TColor, которая соответствует параметру S.

Примечание

Если идентификатор строки является именем цветовой константы, то используйте функцию `IdentToColor`.

Для преобразования битовой карты из зависимого от устройства формата DDB в независимый формат DIB предназначено два метода: `GetDIBSizes` и `GetDIB`.

- Процедура `GetDIBSizes` используется для определения объема необходимой памяти для преобразования битового изображения, зависимого от устройства, в битовое изображение, независимое от устройства (DIB).

  ```
  procedure GetDIBSizes(Bitmap: HBITMAP; var InfoHeaderSize: Integer;
  var ImageSize: Longint)
  ```

  Передайте дескриптор зависимого битового изображения как параметр `Bitmap`, тогда `GetDIBSizes` возвратит размеры заголовка `InfoHeaderSize` и изображения `ImageSize`, соответствующего DIB.

- Функция `GetDIB` используется для того, чтобы преобразовать зависимое от устройства битовое изображение в независимое от устройства (DIB) битовое изображение.

  ```
  function GetDIB (Bitmap: HBITMAP; Palette: HPALETTE; var Bitmaplnfo;
  var Bits): Boolean
  ```

  Передайте дескриптор зависимого битового изображения и дескриптор палитры как параметры `Bitmap` и `Palette`. Тогда `GetDIB` заполнит параметры `BitmapInfo` и `Bits` заголовком и изображением, соответствующими формату DIB. `GetDIB` возвращает `true`, если битовое изображение успешно преобразовалось в DIB и `false` в противном случае.

  До вызова `GetDIB`, необходимо распределить память для заголовка и изображения. Вызовите `GetDIBSizes` для того, чтобы определить размер необходимой для этого памяти.

- Функция `GraphicFileMask` возвращает строку маски, которая определяет допустимое расширение для указанного графического класса.

  ```
  function GraphicFileMask (GraphicClass: TGraphicClass): string
  ```

  Параметр `GraphicClass` может принимать одно из значений: `TBitmap`, `TGraphic`, `TIcon`, `TMetafile`, `TJPEGImage` или определенный пользователем потомок класса `TGraphic`. Функция возвращает строку для каждого встроенного графического класса (табл. 3.3).

Таблица 3.3. *Маски для графических файлов*

Графический класс	Строка фильтра
Tbitmap	*.bmp
Ticon	*.ico
TMetafile	*.emf;*.wmf

Таблица 3.3 *(окончание)*

Графический класс	Строка фильтра
TJPEGImage	*.jpeg;*.jpg
TGraphic	*.jpeg;*.jpg;*.bmp;*.ico;*.emf;*.wmf

> ***ЗАМЕЧАНИЕ***
>
> Расширения для графики JPEG доступны, если модуль JPEG включен в проект.

- Процедура GetCharsetValues производит вызов определенной пользователем процедуры Proc для всех имеющихся в списке цветов. В качестве параметра S в такую процедуру передается строка с именем цвета.

```
procedure GetCharsetValues (Proc: TGetStrProc);
type TGetStrProc = procedure(const S: string) of object;
```

> ***ЗАМЕЧАНИЕ***
>
> Для того чтобы преобразовывать строки в тип TFontCharset, используют функцию IdentToCharset.

В листингах 3.13 и 3.14 приводится пример использования процедуры GetCharsetValues для заполнения списка строк именами всех наборов символов, поддерживаемых Delphi. Этот список строк может использоваться для выбора набора символов, при этом используется процедура, которая добавляет строку к списку строк.

Листинг 3.13. Процедура добавления строки к списку строк

```
procedure AddCharacterSet(const S: string);
begin
  CharSetList.Add(S);
end;
```

Приведем пример кода, который создает список строк, заполняет его и сортирует.

Листинг 3.14. Создание списка строк

```
CharSetList := TStringList.Create;
GetCharSetValues(@AddCharacterSet);
CharSetList.Sort;
```

- Функция CharsetToIdent используется для того, чтобы получить строковое имя для набора символов шрифта.

```
function CharSetToIdent (Charset: Longint; var Ident: string): Boolean
```

Эта функция вернет true, если параметр Ident успешно получил определяющее имя для набора символов. Более подробно о наборах символов шрифтов *см. разд. 2.2*.

Замечание

Значение CharSetToIdent не указывает, допустим ли параметр CharSet для данного шрифта. Существует много шрифтов, которые поддерживают только ограниченное число наборов символов. Некоторые шрифты могут поддерживать менее распространенные наборы символов, которые не могут отображаться в идентификаторах CharsetToIdent.

- Функция IdentToCharset переводит имя набора символов в соответствующий набор символов.

```
function IdentToCharset (const Ident: string; var Charset: Longint):
Boolean
```

Вызовите IdentToCharset для того, чтобы обратить перевод, выполненный функцией CharSetToIdent. Этот метод полезен для преобразования строк, сгенерированных процедурой GetCharsetValues, в пригодные наборы символов. IdentToCharset возвращает true, если параметр Ident успешно преобразовывается в набор символов, который возвращается параметром Charset. IdentToCharset возвращает false, если параметр Ident не является встроенным именем набора символов.

- Функция CopyPalette создает новый объект палитры, который полностью соответствует существующей палитре.

```
function CopyPalette (Palette: HPALETTE): HPALETTE
```

Если параметр Palette ссылается на неправильную палитру Windows или возникает проблема при копировании палитры, то CopyPalette возвращает значение, равное нулю.

- Функция GetDefFontCharset определяет набор символов, который будет использоваться, если шрифт не назначен для данного компонента. Этот набор символов можно применить для инициализации шрифта.

```
function GetDefFontCharSet: TFontCharSet;
type TFontCharset = 0..255;
```

- Функция CreateMappedBmp используется, чтобы сделать выборочные изменения в карте цветов битового изображения.

```
function CreateMappedBmp (Handle: HBITMAP; const OldColors, NewColors:
array of TColor): HBITMAP
```

Эта функция заменяет цвета, указанные параметром OldColors, цветами, указанными параметром NewColors, и возвращает указатель для нового устройства битовой карты (DDB) с новой цветовой таблицей. Параметр Handle — указатель на битовую карту, у которой цветовая таблица должна быть изменена. Массивы OldColors и NewColors должны содержать одинаковое число элементов. Функция CreateMappedBmp создает новую битовую карту. При этом не изменяется оригинал битовой карты с указателем Handle.

Замечание

Функция `CreateMappedBmp` не работает с битовыми картами, которые используют более 256 цветов. Для битовых карт, использующих более 256 цветов, эта функция возвращает ту же самую битовую карту.

- Функция `CreateMappedRes` используется для выборочного изменения в цветовой карте ресурсов битового изображения.

```
function CreateMappedRes (Instance: THandle; ResName: PChar; const
OldColors, NewColors: array of TColor): HBITMAP
```

Эта функция заменяет цвета, указанные параметром `OldColors`, цветами, указанными параметром `NewColors`, и возвращает указатель для нового устройства битовой карты (DDB) с новой цветовой таблицей. Параметры `Instance` и `ResName` используются для идентификации ресурса битовой карты, у которого должна быть изменена цветовая таблица. Массивы `OldColors` и `NewColors` должны содержать одинаковое число элементов.

Замечание

Функция `CreateMappedRes` не работает с битовыми картами, использующими более 256 цветов. Для битовых карт, использующих более 256 цветов, эта функция возвращает ту же самую битовую карту.

- Функция `CreateGrayMappedBmp` превращает стандартные серые цвета битовой карты в системные серые.

```
function CreateGrayMappedBmp (Handle: HBITMAP): HBITMAP
```

Вызовите `CreateGrayMappedBmp` для того, чтобы создать битовую карту, использующую системные серые цвета (`clBtnHightLight`, `clBtnFace`, `clBtnShadow` и `clBtnText`), из битовой карты, использующей стандартные серые масштабные цвета (`clWhite`, `clSilver`, `clGray` и `clBlack`). Эта функция позволяет приложению динамически создавать образы, которые сочетают панель управления установочных параметров пользователя с данным базовым образом. Функция `CreateGrayMappedBmp` не изменяет исходную битовую карту, а создает новую битовую карту с указателем, определенным параметром `Handle`.

Замечание

Функция `CreateGrayMappedBmp` не работает с битовыми картами, которые используют более 256 цветов. Для битовых изображений, использующих более 256 цветов, эта функция возвращает исходное битовое изображение.

- Функция `CreateGrayMappedRes` превращает стандартные серые цвета в ресурсе битовой карты в системные серые.

```
function CreateGrayMappedRes (Instance: THandle; ResName: PChar):
HBITMAP
```

Вызывайте CreateGrayMappedRes для того, чтобы из битовой карты ресурса, использующей стандартные серые масштабные цвета (clWhite, clSilver, clGray и clBlack), создать битовую карту, использующую системные серые цвета (clBtnHightLight, clBtnFace, clBtnShadow и clBtnText) из битовой карты ресурса.

ЗАМЕЧАНИЕ

Функция CreateMappedRes не работает с битовыми картами, которые используют более 256 цветов. Для битовых изображений, использующих более 256 цветов, эта функция возвращает исходное битовое изображение.

3.7. Класс *TImage*

Компоненты класса TImage (unit ExtCtrls) используются для вывода графического изображения [79]. Они предназначены для показа изображений Bitmap, Icon, Metafile или других графических объектов. Класс TImage представляет несколько свойств, позволяющих определить, как изображение будет показано в пределах границ объекта TImage.

Для представления доступа к изображениям нескольким компонентам можно использовать компонент TImageList вместо TImage.

В листинге 3.15 приводится сокращенное описание класса TImage.

Листинг 3.15. Класс *TImage*

```
TImage = class(TGraphicControl)
public
  property Canvas: TCanvas;
published
  property Align;
  property Anchors;
  property AutoSize;
  property Center: Boolean;
  property Constraints;
  property DragCursor;
  property DragKind;
  property DragMode;
  property Enabled;
  property IncrementalDisplay: Boolean;
  property ParentShowHint;
  property Picture: TPicture;
  property PopupMenu;
  property ShowHint;
  property Stretch: Boolean;
```

```
  property Transparent: Boolean;
  property Visible;
  property OnClick;
  property OnDblClick;
  property OnDragDrop;
  property OnDragOver;
  property OnEndDock;
  property OnEndDrag;
  property OnMouseDown;
  property OnMouseMove;
  property OnMouseUp;
  property OnProgress: TProgressEvent;
  property OnStartDock;
  property OnStartDrag;
end;
```

Рассмотрим более подробно основные свойства этого класса.

- Методы свойства `Canvas` (холст) используются для того, чтобы изменять изображение, указанное свойством `Picture`. Например, метод `TextOut` используется для вывода строк.

  ```
  property Canvas: TCanvas
  ```

  ЗАМЕЧАНИЕ

  Метод `Canvas` доступен только в том случае, когда свойство `Picture` содержит `Bitmap`. Попытка использовать `Canvas`, когда `Picture` указывает иной тип графического изображения, приведет к исключительной ситуации `EInvalidOperation`.

В следующем примере (листинг 3.16) при нажатии кнопки `Button1` на канве `Image1` рисуется заштрихованный эллипс.

Листинг 3.16. Рисование на канве *TImage*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with Image1.Canvas do begin
    Brush.color := clRed;
    Brush.style := bsDiagCross;
    Ellipse(0, 0, Image1.Width, Image1.Height);
  end;
end;
```

- Свойство `Center` указывает, центруется ли графический объект в компоненте изображения.

  ```
  property Center: Boolean
  ```

Если изображение не соответствует размерам компонента, свойство `Center` используется для того, чтобы определить, как будет изображение выведено. Если `Center=true`, то изображение центруется на компоненте. При `Center=false`, верхний левый угол изображения совмещается с верхним левым углом компонента.

> ***ЗАМЕЧАНИЕ***
>
> Значение свойства `Center` не оказывает никакого влияния, если свойство `AutoSize=true` или если свойство `Stretch=true` и свойство `Picture` не содержит `Icon`.

- Свойство `IncrementalDisplay` определяет, должны ли выводиться промежуточные изменения изображения в течение медленных действий.

  ```
  property IncrementalDisplay: Boolean
  ```

  Свойство `IncrementalDisplay` имеет смысл устанавливать равным `true` для больших сжатых изображений, вывод которых выполняется долго, или для изображений, изменяемых в медленном вычислительном процессе. При значении `IncrementalDisplay = true` промежуточные изменения изображения выводятся периодически в течение медленных действий. Выбор значения `IncrementalDisplay = false` заставляет ждать полного завершения действия над изображением.

  Для некоторых классов графических объектов событие `OnProgress` вызывается в промежуточных точках в течение медленных действий. Если `IncrementalDisplay = true`, то событие `OnProgress` происходит немедленно после вывода промежуточных изменений изображения. Если `IncrementalDisplay = false`, то событие `OnProgress` происходит, но изображение не обновляется. В *главе 7* в проекте "Просмотр файлов BMP и JPG" приведен пример использования свойства `IncrementalDisplay`.

- Свойство `Picture` служит контейнером для графического объекта одного из классов: `icon`, `metafile`, графики битового изображения или определенной пользователем графики.

  ```
  property Picture: TPicture
  ```

  При установке `Picture` во время проектирования можно использовать редактор `Picture`.

Следующий пример (листинг 3.17) использует на форме компонент `Image1`, кнопку `Button1` и компонент `Shape1`. Если пользователь нажимает кнопку, то изображение формы запоминается в переменной `FormImage` и копируется в `Clipboard`. Затем запомненное изображение формы копируется в компонент `Image1`, производя интересный эффект повторного воспроизведения, если многократно нажимать кнопку (рис. 3.3).

Листинг 3.17. Рисование с использованием свойства *Picture*

```
procedure TForm1.Button1Click(Sender: TObject);
var FormImage: TBitmap;
begin
  FormImage := GetFormImage;
  try
    Clipboard.Assign(FormImage);
    Image1.Picture.Assign(Clipboard);
  finally
    FormImage.Free;
  end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  Shape1.Shape := stEllipse;
  Image1.Stretch := true;
  Clipboard:=TClipboard.Create;
end;
```

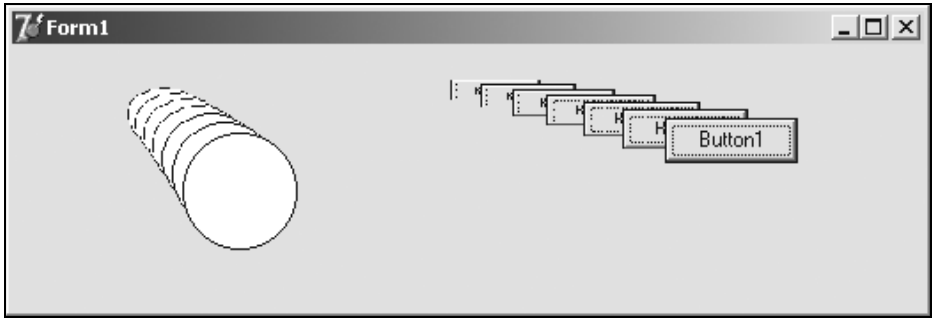


Рис. 3.3. Рисование с использованием свойства Picture

- Если свойство `Stretch=true`, то выводимое изображение масштабируется, изменяя свои размеры до размеров `Image`.

  `property Stretch: Boolean`

  Если свойство `Stretch = false`, то, если размеры `Picture` больше размеров `Image`, выводится часть изображения. Если размеры `Picture` меньше размеров `Image`, выводится все изображение в соответствии со свойством `Center`.

> ***ЗАМЕЧАНИЕ***
>
> Свойство `Stretch` не оказывает никакого влияния, если свойство `Picture` содержит `Icon`.

- Выбор значения свойства `Transparent = true` позволяет видеть объекты позади `TImage` через фон битовой карты (`bitmap`).

  ```
  property Transparent: Boolean
  ```

  Фоном считаются все точки, цвет которых назначен прозрачным. Выбор значения `Transparent = false` делает фон `bitmap` непрозрачным.

  ЗАМЕЧАНИЕ

  Свойство `Transparent` оказывает влияние только в том случае, если свойство `Picture` содержит `bitmap`.

- Свойство `AutoSize` используется для того, чтобы определить, будет ли компонент устанавливать свои размеры автоматически. По умолчанию, `AutoSize = false`.

  ```
  property AutoSize: Boolean
  ```

- Свойство `BiDiMode` используется для того, чтобы определить направление чтения текста (слева направо или справа налево), расположение вертикальных полос прокрутки (слева или справа), а также управлять возможностью выравнивания текста.

  ```
  property BiDiMode: TbiDiMode;
  type TBiDiMode = (bdLeftToRight, bdRightToLeft, bdRightToLeftNoAlign,
  bdRightToLeftReadingOnly);
  ```

  Выравнивание не влияет на компонент, содержащий число, дату, время и денежные значения. Для данных, имеющих средства управления, выравнивание не изменяется для следующих типов полей: `ftSmallInt`, `ftInteger`, `ftWord`, `ftFloat`, `ftCurrency`, `ftBCD`, `ftDate`, `ftTime`, `ftDateTime`, `ftAutoInc`.

- Свойство `BoundsRect` определяет прямоугольник ограничения компонента в системе координат родительского компонента.

  ```
  property BoundsRect: TRect
  ```

  Это свойство можно использовать для быстрого получения координат местоположения всех углов компонента.

  Например, выражение `R:=Control.BoundsRect` соответствует операторам:

  ```
  R.Left := Control.Left;
  R.Top := Control.Top;
  R.Right := Control.Left + Control.Width;
  R.Bottom := Control.Top + Control.Height; .
  ```

  Начало системы координаты находится в верхнем левом углу родительского окна.

3.8. Класс *TJPEGImage*

Изображения в формате JPEG (Joint Photographic Expert Groups — формат объединенной группы экспертов по фотографии) записываются в файлы с расширениями

JPG или JIF (JPEG + TIFF). JPEG — это метод сжатия, который может быть реализован программно или аппаратно практически на любой компьютерной системе.

Формат JPEG обладает отличными характеристиками при сжатии изображений. При коэффициенте сжатия, равном 20, по сравнению с файлами формата BMP, практически не происходит потери качества изображения.

Класс `TJPEGImage` инкапсулирует графику `JPEG` и используется для того, чтобы читать и записывать изображения, сжатые в формате JPEG. Класс пользуется цифровым сжатием и декомпрессией изображения.

Данный класс имеет внутреннюю битовую карту, которая представляет собой изображение `JPEG`. И это внутреннее изображение, и первоначальный источник изображения `JPEG` предназначены только для чтения.

Объект `TJPEGImage` имеет следующие характерные свойства:

данный объект не имеет опубликованного свойства холста (так что рисовать на холсте нельзя). Однако `TJPEGImage` осуществляет защищенный метод рисования, представленный в `TGraphic`, поэтому он может рисовать себя на холсте другого объекта;

объект не обеспечивает никакого доступа к внутреннему изображению битовой карты;

подсчет ссылок и разделение указателя ведется через объект `TJPEGData`. Несколько задач могут обратиться к одному и тому же изображению `TJPEGData`. Объект `TJPEGData` — фактический владелец указателя файла к источнику данных изображения `JPEG`.

Рассмотрим основные свойства класса `TJPEGImage`.

- Свойство `CompressionQuality` указывает отношение между качеством изображения и размером файла.

  ```
  property CompressionQuality: TJPEGQualityRange;
  type TJPEGQualityRange = 1..100;
  ```

  Используйте свойство `CompressionQuality` для того, чтобы установить качество сжатия при выводе изображения `JPEG`. Это свойство не используется при чтении из файла. Более высокое сжатие приводит к более низкому качеству изображения, но к меньшему размеру файла.

- Свойство `Empty` определяет, содержит ли объект изображения графику `JPEG`.

  ```
  property Empty: Boolean
  ```

  Свойство используется для определения, было ли изображение загружено в объект изображения `JPEG`. Значение свойства `Empty` равно `true`, если объект содержит изображение, и равно `false`, если нет.

- Свойство `Grayscale` определяет, является изображение `JPEG` черно-белым или цветным.

  ```
  property Grayscale: Boolean
  ```

Свойство `Grayscale` используется при чтении и записи, то есть при декомпрессии и компрессии изображения `JPEG`.

Свойство `Grayscale` используется для ускорения вывода. При значении `Grayscale = true` цвет отделен от яркости, и изображение будет содержать 255 оттенков серого цвета.

- Свойство `Palette` определяет цветную палитру изображения `JPEG`.

```
property Palette: HPALETTE
```

Используйте свойство `Palette` для того, чтобы получить или установить цветную палитру изображения `JPEG`. Если изображение `JPEG` не нуждается или не использует палитру, свойство `Palette` равно нулю. Изменение свойства `Palette` установит значение `PaletteModified` равным `true` только в том случае, если изображение загружено в объект `TJPEGImage`.

- Свойство `Performance` управляет отношением между качеством цвета и скоростью декомпрессии.

```
property Performance: TJPEGPerformance;
type TJPEGPerformance = (jpBestQuality, jpBestSpeed);
```

Свойство используется для того, чтобы ускорить работу при чтении, то есть при декомпрессии файла, и не используется при записи в файл.

Свойство `Performance` воздействует на показ внутреннего изображения битовой карты. Это изображение — 256-цветная битовая карта. Установка `Performance = jpBestSpeed` может вызвать некоторые искажения в изображении битовой карты.

- Свойство `PixelFormat` определяет, в 24-битовом или 8-битовом формате отображается изображение `JPEG`.

```
property PixelFormat: TJPEGPixelForm;
type TJPEGPixelFormat = (jf24Bit, jf8Bit);
```

С помощью свойства `PixelFormat` можно установить 8-битовый формат пиксельного изображения `JPEG` для видеоадаптеров, которые не могут пользоваться 24 битами. Родной формат изображения `JPEG` — 24-битовый. Свойство `PixelFormat` используется для декомпрессии, то есть при чтения из файлов. `Данное свойство` влияет на битовую карту, если изображение скопировано в нее.

- Свойство `ProgressiveDisplay` управляет выводом изображения при декомпрессии файла.

```
property ProgressiveDisplay: Boolean
```

Свойство используется для того, чтобы разрешить или запретить показ изображения при чтении из файла. Например, если файл источника данных `JPEG` находится на жестком диске (HD), то показ его по мере чтения замедлит скорость вывода. В этом случае, установка `ProgressiveDisplay` в значение `false` ускорит показ изображения. Свойство `ProgressiveDisplay` используется только при де-

компрессии, но воздействует способом, которым изображение было сжато. Изображения, которые были сжаты с `ProgressiveEncoding = false`, не могут воспользоваться преимуществом свойства `ProgressiveDisplay`. Установка `ProgressiveDisplay` в значение `true` для таких изображений не окажет никакого влияния.

- Свойство `ProgressiveEncoding` определяет, будет ли изображение постепенно показываться при декомпрессии.

```
property ProgressiveEncoding: Boolean
```

Свойство используется при записи изображения в файл со сжатием для того, чтобы оно могло постепенно выводиться при декомпрессии. Установка свойства `ProgressiveDisplay = true` не будет оказывать никакого результата, если свойство `ProgressiveEncoding` не было установлено в значение `true` при сжатии изображения.

- Свойство `Scale` определяет размер изображения `JPEG` при выводе.

```
property Scale: TJPEGScale
type TJPEGScale = (jsFullSize, jsHalf, jsQuarter, jsEighth);
```

Используйте свойство `Scale` для того, чтобы ускорить вывод при предварительном просмотре изображений. При `Scale = JsFullSize` выводится изображение максимального размера, а при `Scale = jsEight` изображение занимает лишь одну восьмую часть от истинного размера изображения, но выводится в 8 раз быстрее.

Свойство `TJPEGScale` определяет тип свойства `Scale`. В табл. 3.4 приведен список значений свойства `TJPEGScale`.

Таблица 3.4. *Значения свойства TJPEGScale*

Значение	Действие
`jsFullSize`	Показывает полный размер изображения
`jsHalf`	Показывает изображение половинного размера за половину времени, необходимого для показа изображения полного размера
`jsQuarter`	Показывает изображение 1/4 размера за 1/4 времени, необходимого для показа изображения полного размера
`jsEighth`	Показывает изображение 1/8 размера за 1/8 времени, необходимого для показа изображения полного размера

Свойство `Scale` использует эти значения при чтении из файлов, то есть при декомпрессии.

- Свойство `Smoothing` (сглаживание) используется, если свойство постепенного вывода `ProgressiveDisplay` равно `true`, и определяет, показывать ли изображение `JPEG` в маленьких блоках или с размытыми гранями.

```
property Smoothing: Boolean;
```

Если `Smoothing = true`, то грани размыты в процессе постепенного вывода изображения. Если же `Smoothing = false`, то при выводе изображения используются размытые блоки цвета.

- Свойство `Modified` указывает, был ли графический объект изменен или отредактирован.

  ```
  property Modified: Boolean
  ```

  Если `Modified = true`, то графический объект был изменен. Если же `Modified = false`, то графический объект находится в первоначальном состоянии.

  Свойство `Modified` не может быть равно `true`, если графический объект содержит графику `Icon` или `Metafile`, даже если они были изменены.

  Если графический объект был изменен, сохраните изменения в файле методом `SaveToFile`.

- Свойство `PaletteModified` указывает, изменялась ли палитра.

  ```
  property PaletteModified: Boolean
  ```

  Используйте свойство `PaletteModified` для определения, была ли изменена палитра, используемая для графического изображения. `PaletteModified` меняется в обработчике события `OnChange`. Значение `PaletteModified` остается равным `true`, пока компонент, ответственный за использование этой новой палитры (например, `TImage`), не устанавливает это свойство в значение `false`.

- Свойство `Transparent` используется при выводе прозрачной графики, для которой один цвет считается прозрачным, и точки этого цвета не выводятся.

  ```
  property Transparent: Boolean
  ```

  Некоторые потомки `TGraphic` типа `TIcon` и `TMetafile` всегда прозрачны, так что переустановка свойства для этих объектов не изменяет их поведение. Однако изображение `TBitmap` зависит от этого свойства. Компонент `TImage` устанавливает такое же значение свойства `Transparent`. Более подробно установка прозрачного цвета для изображения уже обсуждалась ранее, в *разд. 3.1*, *3.3*, *3.7*.

Листинг 3.18 демонстрирует пример построения прозрачных изображений, аналогичный приведенному в *разд. 3.3*.

Листинг 3.18. Рисование прозрачных изображений

```
procedure TForm1.Button1Click(Sender: TObject);
var Bitmap : TBitMap;
begin
  Bitmap := TBitmap.Create;
  try
    with Bitmap do begin
      LoadFromFile('MyBmp.bmp');
      Transparent := true;
```

```
      TransParentColor := BitMap.Canvas.Pixels[50,50];
      Form1.Canvas.Draw(0,0,BitMap);
      TransparentMode := tmAuto;
      Form1.Canvas.Draw(50,50,BitMap);
    end;
  finally
    Bitmap.Free;
  end;
end
```

Далее, в *главе 7* будет приведен пример проекта, использующего многие свойства класса `TJPEGImage`.

3.9. Класс *TPrinter*

Класс `TPrinter` (`unit Printers`) инкапсулирует интерфейс принтера Windows [79].

Этот класс используется для того, чтобы управлять печатью, выполняемой приложением. Доступ к `TPrinter` можно получить, вызывая функцию `Printer` модуля `Printers`.

Доступная часть свойств этого класса приведена в листинге 3.19.

Листинг 3.19. Доступные свойства и методы класса *TPrinter*

```
TPrinter = class(TObject)
public
  procedure Abort;
  procedure BeginDoc;
  procedure EndDoc;
  procedure NewPage;
  procedure GetPrinter(ADevice, ADriver, APort: PChar;
    var ADeviceMode: THandle);
  procedure SetPrinter(ADevice, ADriver, APort: PChar;
    ADeviceMode: THandle);
  property Aborted: Boolean;
  property Canvas: TCanvas;
  property Capabilities: TPrinterCapabilities;
  property Copies: Integer;
  property Fonts: TStrings;
  property Handle: HDC;
  property Orientation: TPrinterOrientation;
  property PageHeight: Integer;
  property PageWidth: Integer;
  property PageNumber: Integer;
  property PrinterIndex: Integer;
```

```
  property Printing: Boolean;
  property Printers: TStrings;
  property Title: string;
end;
```

Свойство `PrintScale` компонента `TForm` определяет, как напечатается изображение формы.

Печать реализуется через обращение к файловой переменной `Text` или холсту принтера. Печать не начинается, пока не вызывается метод `EndDoc`. Имя, используемое в `Print Manager`, определяется свойством `Title`. Для того чтобы начать работу печати, вызывают метод `BeginDoc`. Чтобы закончить печать, выполняемую принтером, вызывают метод `EndDoc`. Для аварийного завершения печати вызывают метод `Abort`.

Рассмотрим свойства и методы этого класса более подробно.

- Свойство `Aborted` определяет, прервана ли работа печати вызовом метода `Abort`.

  ```
  property Aborted: Boolean
  ```

  Если `Aborted = true`, то работа печати была прервана. Если `Aborted = false`, то пользователь не прерывал работу печати.

  Следующий код выведет диалоговое окно с сообщением, если работа печати будет прервана:

  ```
  if Printer.Aborted then
    MessageDlg('Не закончена печать', mtInformation, [mbOK], 0);
  ```

- Свойство `Canvas` представляет поверхность страницы печати.

  ```
  property Canvas: TCanvas
  ```

  Используйте свойства `Brush`, `Font` и `Pen`, для того чтобы определить, в каком виде рисунок или текст появляется на странице.

> ***ЗАМЕЧАНИЕ***
>
> Некоторые принтеры не поддерживают графику. Поэтому методы `Draw`, `StretchDraw` или `CopyRect` могут не печатать на этих принтерах.

В следующем примере (листинг 3.20), при нажатии кнопки будет напечатан `bitmap`. Чтобы воспользоваться этим примером, необходимо подключить модуль `Printers`.

Листинг 3.20. Пример печати на канве принтера

```
procedure TForm1.Button1Click(Sender: TObject);
var  Bmp: TBitmap;
begin
  Bmp := TBitmap.Create;
  try
```

```
    Bmp.LoadFromFile('MyBitmap.bmp');
    with Printer do begin
      BeginDoc;
      Canvas.Draw((PageWidth-Bmp.Width) div 2,
                  (PageHeight-Bmp.Height) div 2, Bmp);
      EndDoc;
    end;
  finally
    Bmp.Free;
  end;
end;
```

- Свойство `Capabilities` указывает текущие параметры драйвера принтера.

  ```
  property Capabilities: TPrinterCapabilities;
  type  TPrinterCapability = (pcCopies, pcOrientation, pcCollation);
          TPrinterCapabilities = set of TPrinterCapability;
  ```

  Свойство содержит параметры принтера, указывает ориентацию страницы, число копий и т. д.

- Свойство `Copies` определяет число печатаемых копий.

  ```
  property Copies: Integer
  ```

- Свойство `Fonts` используется, чтобы получить список шрифтов, поддерживаемых принтером.

  ```
  property Fonts: TStrings
  ```

  Список содержит шрифты `TrueType`, даже если принтер не поддерживает их внутренне, потому что интерфейс устройства графики Windows (GDI) может рисовать шрифты `TrueType` во время печати.

- Свойство `Orientation` определяет ориентацию страницы — вертикальную или горизонтальную печать на странице.

  ```
  property Orientation: TPrinterOrientation;
  type TPrinterOrientation = (poPortrait, poLandscape);,
  ```

  где `TPrinterOrientation` — тип свойства `Orienation`. Он определяет возможные значения для ориентации страницы при печати (табл. 3.5).

Таблица 3.5. *Возможные значения TPrinterOrientation*

Значение	Действие
`PoPortrait`	Печатает вертикально на странице
`PoLandscape`	Печатает горизонтально на странице

Следующий пример (листинг 3.21) использует две радиокнопки на форме для выбора режима печати `Landscape` или `Portrait`.

Листинг 3.21. Пример использования свойства *Orientation*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Printer.BeginDoc;
  Printer.Canvas.TextOut(100,100,'Hi there');
  Printer.EndDoc;
end;

procedure TForm1.PortraitClick(Sender: TObject);
begin
  Printer.Orientation := poPortrait;
end;

procedure TForm1.LandscapeClick(Sender: TObject);
begin
  Printer.Orientation := poLandscape;
end;
```

- Свойство `PageHeight` указывает высоту печатаемой страницы в пикселах.
  ```
  property PageHeight: Integer
  ```
- Свойство `PageNumber` указывает номер печатаемой страницы.
  ```
  property PageNumber: Integer
  ```
  Значение `PageNumber` увеличивается, если вызывается метод `NewPage`.

В следующем примере (листинг 3.22) при щелчке кнопкой одна строка текста будет напечатана на шести отдельных страницах. После печати каждой страницы, сообщение, указывающее номер напечатанной страницы, появляется в строке статуса.

Листинг 3.22. Пример использования свойства *PageNumber*

```
procedure TForm1.Button1Click(Sender: TObject);
var  I: Integer;
begin
  StatusBar1.SimplePanel := true;
  Printer.BeginDoc;
  for I := 1 to 6 do begin
    Printer.Canvas.TextOut(100, 100, 'Object Pascal is great');
    StatusBar1.SimpleText:='Печать страницы '+
      IntToStr(Printer.PageNumber));
    Printer.NewPage;
  end;
  Printer.EndDoc;
end;
```

- Свойство PageWidth указывает значение ширины печатаемой страницы в пикселах.

  ```
  property PageWidth: Integer
  ```

- Свойство PrinterIndex определяет, какой из принтеров, внесенный в список свойства Printers, является в настоящее время выбранным.

  ```
  property PrinterIndex: Integer
  ```

  Если это значение изменяется, то автоматически вызывается метод EndDoc. Для того чтобы выбирать принтер по умолчанию, установите значение PrinterIndex = -1.

- Список установленных принтеров зафиксирован в свойстве Printers. Список шрифтов, поддерживаемых текущим принтером, зафиксирован в свойстве Fonts.

  ```
  property Printers: TStrings
  ```

- Свойство Printing используется для того, чтобы определить, идет ли печать. Значение Printing = true, если приложение вызвало метод BeginDoc, но метод EndDoc (или метод Abort) еще не вызывался.

  ```
  property Printing: Boolean
  ```

Следующий код (листинг 3.23) прерывает работу печати, если нажата клавиша <Esc>. Обратите внимание на то, что надо назначить Form1.KeyPreview = true для того, чтобы процедура OnKeyDown вызывалась из Form1.

Листинг 3.23. Пример использования свойства *Printing*

```
procedure TForm1.FormKeyDown(Sender:TObject; var
  Key:Word; Shift: TShiftState);
begin
  if (Key=VK_ESCAPE) and Printer.Printing then begin
    Printer.Abort;
    MessageDlg('Printing aborted', mtInformation, [mbOK],0);
  end;
end;
```

- Метод Abort (procedure Abort) заканчивает печать, пропуская все ненапечатанные данные.

  Используйте метод Abort для того, чтобы закончить печать прежде, чем она закончится, используя метод EndDoc.

  Когда вызывается этот метод, устройство устанавливается для следующей печати.

- Метод BeginDoc (procedure BeginDoc) используется для инициализации печати. Если печать прошла успешно, то приложение вызывает EndDoc для того, чтобы закончить печать. Фактически печать не будет начинаться до вызова EndDoc.

- Метод `EndDoc` (`procedure EndDoc`) заканчивает печать и закрывает открытый в настоящее время текст. После того как приложение вызывает `EndDoc`, принтер начинает печатать.
- Метод `GetPrinter` устанавливает текущий принтер.

  ```
  procedure GetPrinter (ADevice, ADriver, APort: PChar; var ADeviceMode: THandle)
  ```

  Нет необходимости непосредственно вызвать `GetPrinter`. Вместо этого можно выбрать принтер из массива свойств `Printers`.
- Метод `NewPage` (`procedure NewPage`) начинает новую страницу и увеличивает свойство `PageNumber` на 1. `NewPage` начинает печатать на новой странице и увеличивает значение свойства `PageNumber`, устанавливая указатель свойства `Pen` в `(0, 0)`.

Следующий пример (листинг 3.24) использует кнопку, `PageControl` и метод `Print`, отображающий диалоговое окно **Печать**. После нажатия кнопки будет показано диалоговое окно **Печать**, в котором пользователь может выбрать любой набор страниц.

Листинг 3.24. Пример использования метода *NewPage*

```
procedure TForm1.Button1Click(Sender:TObject);
var  I, Start, Stop: Integer;
begin
  PrintDialog1.Options := [poPageNums, poSelection];
  PrintDialog1.FromPage := 1;
  PrintDialog1.MinPage := 1;
  PrintDialog1.ToPage := PageControl1.PageCount;
  PrintDialog1.MaxPage := PageControl1.PageCount;
  if PrintDialog1.Execute then begin
    { определение области печати }
    with PrintDialog1 do begin
      if PrintRange = prAllPages then begin
        Start := MinPage - 1;
        Stop := MaxPage - 1;
    end
    else
    if PrintRange = prSelection then begin
      Start := PageControl1.ActivePage.PageIndex;
      Stop := Start;
    end
    else { PrintRange = prPageNums } begin
      Start := FromPage - 1;
      Stop := ToPage - 1;
    end;
  end;
```

```
  with Printer do  begin
    BeginDoc;
    for I := Start to Stop do  begin
      PageControl1.Pages[I].PaintTo(Handle, 10, 10);
      if I <> Stop then NewPage;
    end;
    EndDoc;
  end;
  end;
end;
```

- Метод `SetPrinter` определяет принтер как текущий.

  ```
  procedure SetPrinter (ADevvice, ADriver, APort: PChar; ADeviceMode:
  THandle);
  ```

  Не вызывайте `SetPrinter` непосредственно. Вместо этого, выберете принтер из массива свойств `Printers`.

3.10. Заключение

В *главе 3* мы рассмотрели только небольшую часть компонентов, обладающих канвой. Здесь не нашлось места, например, для очень важного компонента `TPaintBox`, который часто используется для создания "своих" графических классов и компонентов. Общим для всех этих компонентов является наличие холста `Canvas`, на котором можно рисовать.

ГЛАВА 4

Мультимедиа

Одна из особенностей современного развития информационно-коммуникационных технологий — интенсивное использование в различных приложениях мультимедийной информации. Сам термин мультимедиа ("`multi`" — много, "`media`" — среда) подразумевает использование в рамках одного приложения виды информации с различной структурой. Строго говоря, к мультимедийной информации относятся все виды информации, с которыми может работать компьютер:

- изображения (`Graphics` и `Images`);
- звук (`Audio`);
- человеческая речь (`Speech`);
- анимация (`Animation`);
- видео (`Video`).

Иногда в эту классификацию включаются текстовые файлы. На практике рамки термина "мультимедиа" обычно ограничивают. Поскольку компьютерная графика сама по себе представляет огромную область, которая давно выделилась в самостоятельную науку, как правило, к мультимедиа относят все вопросы, связанные с обработкой звука и видео. Именно в таком смысле мы и будет в дальнейшем изложении понимать этот термин.

Как и любая современная среда, Delphi позволяет разработчику реализовать мультимедийные (MM) приложения, обладающие практически неограниченной функциональностью. При разработке приложений этого типа надо представлять структуру MM-подсистемы Windows. Практически MM-функции сосредоточены в библиотеке MMSYSTEM.DLL. В свою очередь, они делятся на функции низкоуровнего программирования (low-level functions) и функции Media Control Interface (MCI). В совокупности эти средства позволяют решить практически любую задачи из области разработки MM-приложений. Отметим, что в настоящее время, кроме указанных, широкое распространение получила разработка приложений с использованием различных MM Frameworks (DirectX, OpenML, OpenAL). Далее рассмотрение будет ограничено рамками средств, имеющихся в составе MMSYSTEM.DLL.

Входящая в состав среды программирования Delphi библиотека визуальных компонентов предоставляет пользователю для работы с мультимедийной информацией два компонента — `Animate` и `MediaPlayer`. Рассмотрим подробнее их возможности.

4.1. Компонент *Animate*

Компонент `Animate` [60] изначально располагается на линейке компонентов `Win32` и предназначен для решения простейших задач, требующих использования мультимедийных средств.

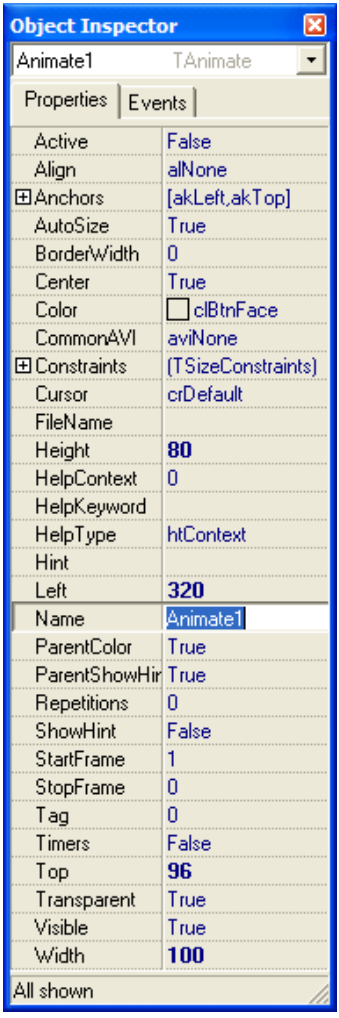


Рис. 4.1. Вкладка **Properties** диалогового окна **Object Inspector** для компонента *Animate*

Он может воспроизводить "немые" (неозвученные) AVI-файлы, в которых отсутствует звуковая дорожка. При попытке воспроизведения озвученных файлов выдается сообщение об ошибке (выполнение основной программы при этом не прекращается).

На рис. 4.1 и 4.2 представлен внешний вид вкладок **Properties** (Свойства) и **Events** (События) диалогового окна **Object Inspector** (Инспектор объектов) для компонента `Animate`.

Рассмотрим далее только те свойства компонента, которые имеют отношение к работе с мультимедийной информацией.

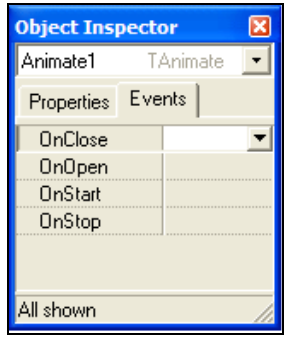


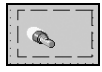
Рис. 4.2. Вкладка **Events** диалогового окна **Object Inspector** для компонента *Animate*

Остальные свойства, приведенные в инспекторе объектов, являются стандартными для большинства объектов Visual Component Library (VCL). Прежде всего, для реализации основной функции — воспроизведения анимации, пользователь должен определить, что именно он хочет воспроизвести на форме. Предоставляется два варианта действий:

1. Использовать набор предопределенных анимаций.
2. Использовать произвольно выбранные пользовательские файлы.

Первый вариант использует различные значения свойства `CommonAvi` и обычно применяется для визуализации стандартных системных событий, исчерпывающий список которых приведен в табл. 4.1.

Таблица 4.1. *Системные события и соответствующие им анимации*

Значение свойства *CommonAvi*	**Событие**	**Графическое изображение на форме, соответствующее событию**
`AviCopyFile`	Копирование файлов	
`aviDeleteFiles`	Удаление файлов	
`aviEmptyRecycle`	Пустая корзина	
`aviFindComputers`	Поиск компьютера в сети	
`AviFindFile`	Поиск файла	
`aviFindFolder`	Поиск Папки	
`aviRecycleFile`	Восстановление содержимого корзины	
`aviNone`	Пользовательская анимация	

Видно, что каждому событию соответствует определенное значение свойства `CommonAVI` компонента `Animate`. Любой из перечисленных вариантов можно выбрать, раскрыв поле списка, появляющееся при выборе этого свойства в диалоговом окне **Object Inspector** (Инспектор объектов) (рис. 4.3).

После этого пользователь должен щелкнуть указателем мыши на пиктограмме, соответствующей компоненту линейки Win32, и на форме (рис. 4.4). В результате на ней появится изображение, графика которого будет соответствовать необходимому стандартному событию (табл. 4.1).

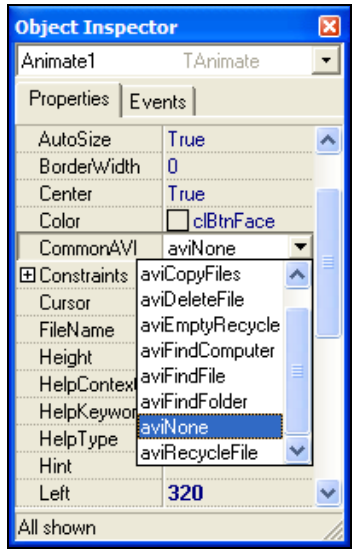


Рис. 4.3. Возможные значения свойства *CommonAVI* компонента *Animate*

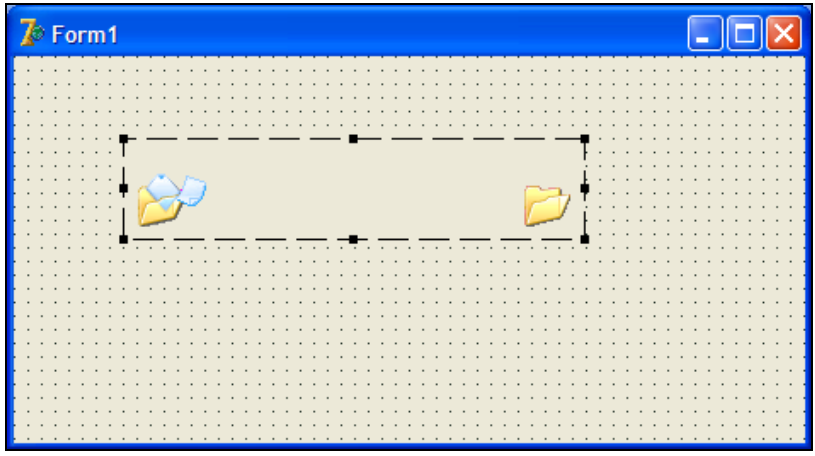


Рис. 4.4. Форма с расположенным на ней компонентом *Animate*

Второй вариант действий допускает использование произвольного файла с расширением AVI, для чего достаточно щелчка указателем мыши в поле свойства `FileName` компонента `Animate` (рис. 4.5). Это приведет к появлению стандартного диалогового окна **Открытие файла**. Значение свойства `CommonAVI` при этом должно быть установлено равным `aviNone`.

Рассмотрим теперь некоторые дополнительные возможности, предоставляемые компонентом. После размещения компонента на форме, свойство `Active` может находиться в двух состояниях — статическом (`false`) и динамическом (`true`).

В первом случае изображение будет выглядеть, как на рис. 4.4. Во втором — пользователь увидит на форме динамическую картинку, которая будет затем воспроизводиться в приложении.

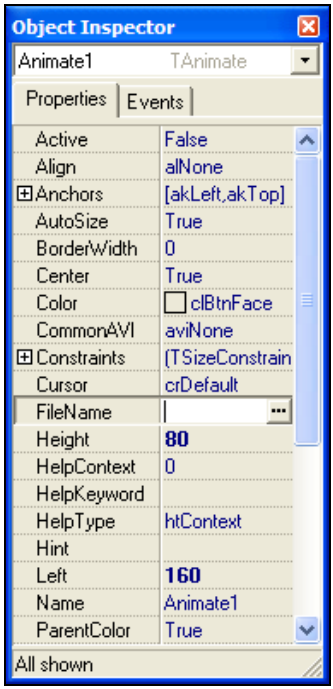


Рис. 4.5. Выбор произвольного AVI-файла для воспроизведения с помощью компонента *Animate*

Рис. 4.6. Форма к проекту "Использование стандартной мультипликации *FindComputer*"

Режим воспроизведения анимации в готовом приложении задается свойствами `Repetition`, `StartFrame`, `StopFrame`. Первое из них — `Repetition`. Значение этого свойства представляет собой целое число, которое задает количество повторов при проигрывании файла (значение 0 соответствует бесконечному повторению). Свойства `StartFrame` и `StopFrame` (их значения — тоже целые числа) задают начальный и конечный кадры анимации, которые требуется воспроизвести. Таким образом, можно проигрывать только необходимую пользователю часть анимационного файла, задавая начальный и конечный кадры.

Сразу после определения анимации значение свойства `StartFrame=0`, а значение свойства `StopFrame` равняется номеру последнего кадра (кстати, это позволяет определить число кадров анимации).

Методы компонента `Animate` — стандартные.

Из рассмотренного очевидно, что область применения компонента `Animate` крайне ограничена. В основном это анимирование системных событий, небольшие анимации, анимированные элементы.

Рассмотрим некоторые проекты с использованием компонента `Animate`. Проекты приведены на компакт-диске в папке **Примеры | Глава 4 | Использование компонента Animate**.

В первом проекте в приложении используется стандартная мультипликация `FindComputer`, причем ее воспроизведение начинается с заданного кадра. При окончании

воспроизведения выдается сообщение, продублированное звуковым сигналом. На рис. 4.6 приведено изображение формы к проекту, а в листинге 4.1 содержится код данного проекта.

Листинг 4.1. Воспроизведение стандартной мультипликации *FindComputer*

```
procedure TFormMain.FormShow(Sender: TObject);
  begin
     Animator.CommonAVI:=aviFindComputer;
     Animator.Center:=True;
     Animator.StartFrame:=1;
  end;

procedure TFormMain.BPlayClick(Sender: TObject);
  begin
   Animator.Play
              (StrToInt(EStartFrame.Text),Animator.FrameCount,1);
  end;

procedure TFormMain.BStopClick(Sender: TObject);
  begin
    Animator.Stop;
  end;

procedure TFormMain.AnimatorStop(Sender: TObject);
  begin
    Application.MessageBox('Playback finished','Notification',0);
    Beep;
 end;
```

В методе `Play` компонента `Animate` строка, введенная в поле `TEdit`, преобразуется в значение целого числа, которое, в свою очередь, задает начальный кадр воспроизведения. По окончании анимации выдается `MessageBox` с сообщением, заданным пользователем.

Во втором проекте в приложении используются стандартные мультипликации, каждая из которых воспроизводится определенное время. По окончании воспроизведения каждой из мультипликаций выдается сообщение, и воспроизводятся различные звуковые сигналы.

В приложении имеются два модуля `MainUnit` и `SettingsUnit`. Первый из них реализует функции главной формы (рис. 4.7) (вызов меню настроек, проигрывание, остановка, выход). Второй модуль задает настройки воспроизведения (рис. 4.8).

Код данного проекта представлен в листингах 4.2, 4.3, 4.4. Сразу после запуска (листинг 4.2) необходимо настроить воспроизведение, указав для каждой анимации число повторений и путь к звуковому файлу, который будет звучать при воспроизведении анимации.

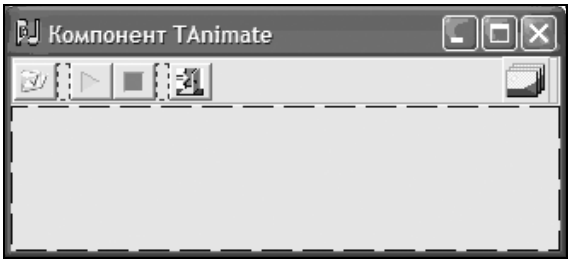


Рис. 4.7. Главная форма проекта "Воспроизведение стандартных мультипликаций"

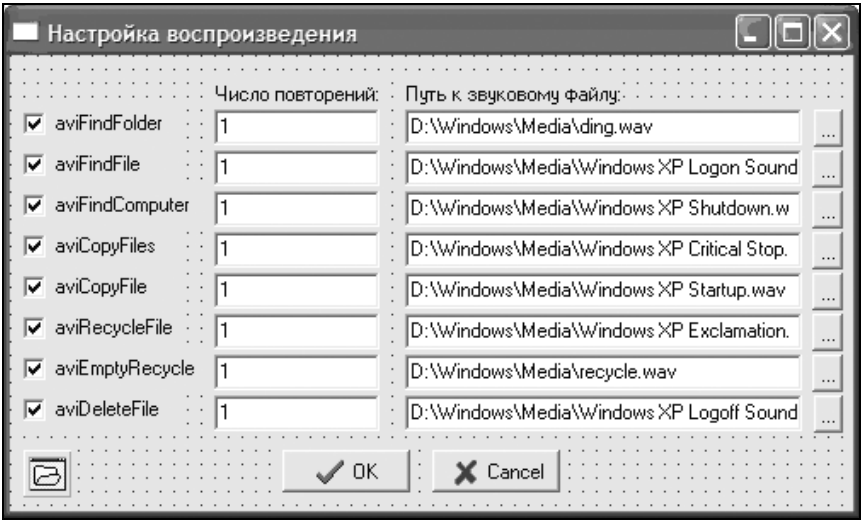


Рис. 4.8. Форма **Настройка воспроизведения**

Листинг 4.2. Стандартный диалог открытия файла анимации *FindFile*

```
procedure TSettingsForm.FindFolderBtnClick(Sender: TObject);
  begin
    if OpenDialog.Execute then
      FindFolderSoundPath.Text := OpenDialog.FileName;
  end;
```

После открытия файла формируются собственно настройки воспроизведения анимации (листинг 4.3).

Листинг 4.3. Формирование настроек воспроизведения анимации *FindFile*

```
procedure TSettingsForm.cbFindFolderClick(Sender: TObject);
  begin
    if cbFindFolder.Checked then
```

```
      begin
        FindFolderRepetitionsEdit.Enabled := true;
        FindFolderSoundPath.Enabled := true;
        FindFolderBtn.Enabled := true;
      end
    else
      begin
        FindFolderRepetitionsEdit.Enabled := false;
        FindFolderSoundPath.Enabled := false;
        FindFolderBtn.Enabled := false;
      end;
  end;
```

И, наконец, по нажатию кнопки воспроизведения на главной форме (листинг 4.4) идет воспроизведение файлов.

Листинг 4.4. Запуск анимаций на воспроизведение

```
procedure TMainForm.PlayBtnClick(Sender: TObject);
  begin
    StopAll:= false;
    Animate.CommonAVI:= usefulAVI[0].CommonAVI;
    num:=0;
    Animate.Visible:= true;
    Animate.Active:= true;
  end;
```

4.2. Компонент *MediaPlayer*

Класс `TMediaPlayer` [60, 92, 113], описанный в модуле `MPLAYER`, управляет устройствами, которые обеспечивают интерфейс с драйвером Media Control Interface (MCI).

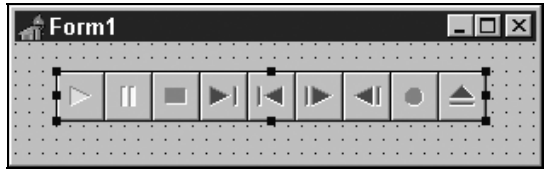


Рис. 4.9. Кнопки компонента *TmediaPlayer*

Внешне (рис. 4.9) компонент `TMediaPlayer` представляет собой набор кнопок (Play, Stop, Eject и т. д.), с помощью которых осуществляется управление устройствами мультимедиа, например, устройством воспроизведения компакт-дисков CD-ROM, MIDI-проигрывателем или кассетным видеомагнитофоном VCR. Устройства муль-

тимедиа могут быть реализованы, как аппаратные средства ЭВМ или как программные средства (приложения).

В табл. 4.2 приведено описание всех кнопок компонента `TMediaPlayer`.

Таблица 4.2. *Кнопки компонента TMediaPlayer*

Кнопка	Код	Действие
Play	`btPlay`	Запускает `TMediaPlayer`
Pause	`btPause`	Приостанавливает проигрывание или запись. Если действие уже приостановлено, то запускает проигрывание или запись
Stop	`btStop`	Останавливает проигрывание или запись
Next	`btNext`	Выполняет переход на следующую дорожку или к концу, если устройство не использует дорожек
Prev	`btPrev`	Выполняет переход к предыдущей дорожке или к началу, если устройство не использует дорожек
Step	`btStep`	Перемещает вперед на следующий фрагмент (фрейм)
Back	`btBack`	Перемещает назад на предыдущий фрагмент (фрейм)
Record	`btRecord`	Начинает запись
Eject	`BtEject`	Выталкивает носитель

Мультимедийное устройство начинает воспроизведение, приостанавливает его, останавливает и выполняет другие действия, когда пользователь нажимает соответствующую кнопку на компоненте `TMediaPlayer` во время выполнения проекта. Устройство также может управляться методами, которые соответствуют кнопкам (`Play`, `Pause`, `Stop`, `Next`, `Previous`, `Step`, `Back`, `StartRecording` и `Eject`).

- Свойство `DeviceType` определяет тип мультимедийного устройства. Если устройство хранит информацию в файле, имя этого файла определяется свойством `FileName`. Если свойство `DeviceType` имеет значение `dtAutoSelect`, то `TMediaPlayer` пытается определить тип устройства по расширению файла. Для того чтобы компонент `TmediaPlayer`, созданный во время выполнения проекта, пытался автоматически открывать устройство, указанное в свойстве `DeviceType`, установите значение свойства `AutoOpen` равным `true`.
- Метод `Open (procedure Open)` (листинг 4.5) открывает мультимедийное устройство для компонента `TMediaPlayer`. Тип устройства должен быть определен свойством `DeviceType` прежде, чем устройство будет открыто. При ошибочном завершении метод `Open` запоминает числовой код ошибки в свойстве `Error` и создает соответствующее сообщение в свойстве `ErrorMessage`.

  Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `Open` завершился. Свойство `Notify` определяет, сгенерировал ли метод `Open` событие `OnNotify`.

Листинг 4.5. Пример использования метода *Open*

```
var sWinDir: String; iLength: Integer;
begin
  // инициализация переменных
  iLength := 255;
  setLength(sWinDir, iLength);
  iLength := GetWindowsDirectory(PChar(sWinDir),iLength);
  SetLength(sWinDir, iLength);
  With MediaPlayer1 do begin
    Filename := sWinDir + '\Clock.avi';
    // установить имя видеофайла
    DeviceType := dtAVIVideo;
    // установить тип устройства AVI
    Display := Panel1;
    // установить устройство вывода TPanel
    Open;
    Play;
  end;
end;
```

- Метод `Play` (`procedure Play`) начинает воспроизводить медиаинформацию, загруженную в открытое мультимедийное устройство. Именно этот метод вызывается нажатием кнопки **Play** на панели управления медиаплеером во время выполнения.

  После завершения проигрывания, метод `Play` запоминает числовой код ошибки в свойстве `Error` и создает соответствующее сообщение в свойстве `ErrorMessage`.

  Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `Play` завершился. Свойство `Notify` определяет, сгенерировал ли метод `Play` событие `OnNotify`.

  Если установлено свойство `StartPos`, воспроизведение начинается с позиции, заданной свойством `StartPos`. В противном случае воспроизведение начинается с текущей позиции, заданной свойством `Position`. Если установлено свойство `EndPos`, воспроизведение завершается в заданной им позиции, в противном случае проигрывание заканчивается в конце медиа. Перемотка перед воспроизведением зависит от установки свойства `AutoRewind`.

- Метод `Stop` (`procedure Stop`) останавливает воспроизведение или запись. Этот метод вызывается нажатием кнопки **Stop** на панели управления медиаплеера во время выполнения. После завершения воспроизведения или записи, метод `Stop` запоминает числовой код ошибки в свойстве `Error` и создает соответствующее сообщение в свойстве `ErrorMessage`.

  При этом, как обычно, свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `Stop` завершился. Свойство `Notify` определяет, сгенерировал ли `Stop` событие `OnNotify`.

- Метод `Pause` (`procedure Pause`) приостанавливает работу мультимедийного устройства. Если устройство уже приостановлено вызовом `Pause`, продолжение проигрывания или записи обеспечивается вызовом метода `Resume`. Метод `Pause` вызывается нажатием кнопки **`Pause`** на панели управления медиаплеера.

  После завершения, метод `Pause` запоминает числовой код ошибки в свойстве `Error` и соответствующее сообщение в свойстве `ErrorMessage`.

  Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `Pause` завершился. Свойство `Notify` определяет, сгенерировал ли метод `Pause` событие `OnNotify`.

Далее приводится пример (листинг 4.6), в котором используется медиаплеер, таймер и кнопка на форме. Кнопка видима только во время выполнения приложения. Если пользователь нажимает кнопку, то проигрывается WAV-файл. Если пользователь нажимает кнопку снова, начинается пауза. Надпись на кнопке изменяется в зависимости от того, проигрывается WAV-файл, находится в состоянии паузы или остановлен.

Для выполнения этого примера нужно иметь файл CHIMES.WAV в текущем каталоге и устройство, умеющее проигрывать WAV-аудиофайлы.

Листинг 4.6. Пример использования метода *Pause*

```
procedure TForm1.FormActivate(Sender: TObject);
var WinDir: PChar;
begin
  MediaPlayer1.Visible := False;
  GetMem(WinDir, 144);
  GetWindowsDirectory(WinDir, 144);
  StrCat(WinDir, 'chimes.wav');
  MediaPlayer1.FileName := StrPas(WinDir);
  MediaPlayer1.Open;
  FreeMem(WinDir, 144);
  Button1.Caption := 'Play';
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  if Button1.Caption = 'Play' then begin
    Button1.Caption := 'Pause';
    MediaPlayer1.Play;
  end
  else begin
    Button1.Caption := 'Play';
    MediaPlayer1.Pause;
  end;
end;
```

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if MediaPlayer1.Mode = mpStopped then
    Button1.Caption := 'Play';
end;
```

- Метод `Step` (`procedure Step`) перемещает указатель проигрывателя вперед на некоторое число фрагментов (фреймов), определенное свойством `Frames` в текущем загруженном медиаконтенте. Метод `Step` вызывается нажатием кнопки **`Step`** на панели управления медиаплеера во время выполнения.

  После завершения, метод `Step` запоминает числовой код ошибки в свойстве `Error` и создает соответствующее сообщение в свойстве `ErrorMessage`.

  Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `Step` завершился. Свойство `Notify` определяет, сгенерировал ли метод `Step` событие `OnNotify`.

- Метод `Back` (`procedure Back`) перемещает указатель проигрывателя назад на некоторое число фрагментов (фреймов), определенное свойством `Frames` в текущем загруженном медиаконтенте. Метод `Back` вызывается нажатием кнопки **Back** на панели управления медиаплеера во время выполнения.

  После завершения, метод `Back` запоминает числовой код ошибки в свойстве `Error` и создает соответствующее сообщение в свойстве `ErrorMessage`.

  Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `Back` завершился. Свойство `Notify` определяет, сгенерировал ли метод `Back` событие `OnNotify`.

В следующем примере (листинг 4.7) пользователь выбирает AVI-видеофайл, применяя метод `OpenDialog1`, и открывает этот файл в компоненте `MediaPlayer1`. Кнопки **Forward** и **Back** могут использоваться для перемещения по фрагментам AVI-клипа в любом направлении.

Листинг 4.7. Пример использования метода *Back*

```
procedure TForm1.OpenClick(Sender: TObject);
begin
  OpenDialog1.DefaultExt := 'AVI';
  OpenDialog1.Filename := '*.avi';
  if OpenDialog1.Execute then begin
    MediaPlayer1.Filename := OpenDialog1.Filename;
    MediaPlayer1.Open;
  end;
end;

procedure TForm1.BackClick(Sender: TObject);
begin
  MediaPlayer1.Back;
end;
```

```
procedure TForm1.ForwardClick(Sender: TObject);
begin
  MediaPlayer1.Next;
end;
```

- Метод `Previous` (`procedure Previous`) устанавливает текущую позицию на начало предыдущей дорожки, если исходная позиция находилась в начале дорожки. Если исходная позиция находилась в месте, отличном от начала дорожки, то `Previous` устанавливает текущую позицию на начало текущего трека. Если устройство не использует треки, `Previous` устанавливает текущую позицию в начало медиаконтента, которое определено свойством `Start`. Метод `Previous` вызывается нажатием кнопки **Previous** панели управления медиаплеера.

  После завершения работы, метод `Previous` запоминает числовой код ошибки в свойстве `Error` и создает соответствующее сообщение в свойстве `ErrorMessage`.

  Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `Previous` завершился. Свойство `Notify` определяет, сгенерировал ли `Previous` событие `OnNotify`.

  В следующем примере включается воспроизведение медиаплеера. Если `AutoRewind = False` (автоперемотка выключена), вызывается метод `Previous`, чтобы после завершения проигрывания выполнить перемотку к началу медиаконтента.

  ```
  MediaPlayer.Wait := True;
  MediaPlayer.Play;
  if not MediaPlayer.AutoRewind then MediaPlayer.Previous;
  ```

- Метод `Next` (`procedure Next`) перемещает текущую позицию в начало следующего трека (дорожки) загруженного в настоящее время медиаконтента.

  Если текущая позиция установлена на последний трек, `Next` перемещает текущую позицию на начало этого трека. Если мультимедийное устройство не использует треки, `Next` переводит текущую позицию к концу медиаконтента. Метод `Next` вызывается нажатием кнопки **Next** на панели управления медиаплеера.

  После завершения, метод `Next` запоминает числовой код ошибки в свойстве `Error` и создает соответствующее сообщение в свойстве `ErrorMessage`.

  Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `Next` завершился. Свойство `Notify`, как обычно, определяет, сгенерировал ли `Next` событие `OnNotify`.

- Метод `StartRecording` (`procedure StartRecording`) выполняет запись, начиная с текущей позиции, или от позиции, указанной в свойстве `StartPos`. Этот метод вызывается нажатием кнопки **Record** на панели управления медиаплеера.

  После завершения, `StartRecording` запоминает числовой код ошибки в свойстве `Error` и создает соответствующее сообщение в свойстве `ErrorMessage`.

Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `StartRecording` завершился. Свойство `Notify` определяет, сгенерировал ли `StartRecording` событие `OnNotify`.

По умолчанию `Notify = True`, а свойство `Wait` становится равным `False` после завершения метода `StartRecording`. Однако, если значения этих свойств были установлены до вызова `StartRecording`, они остаются неизменными.

- Метод `Eject` (`procedure Eject`) освобождает загруженный носитель из открытого мультимедийного устройства и закрывает файл. Метод `Eject` вызывается нажатием кнопки **Eject** на панели управления медиаплеера.

  После завершения `Eject` запоминает числовой код ошибки в свойстве `Error` и соответствующее сообщение в свойстве `ErrorMessage`.

  Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `Eject` завершился. Свойство `Notify`, по-прежнему, определяет, сгенерировал ли `Eject` событие `OnNotify`.

  В примере, приведенном далее (листинг 4.8), компакт-диск освобождается из проигрывателя CD-ROM и через 10 секунд медиаплеер выключается. Для того чтобы программа выполнялась правильно, надо иметь установленное CD-аудиоустройство с программной поддержкой выбрасывателя.

Листинг 4.8. Пример использования метода *Eject*

```
var TimerOver: Word;
procedure TForm1.FormClick(Sender: TObject);
begin
  MediaPlayer1.DeviceType := dtCDAudio;
  MediaPlayer1.Open;
  MediaPlayer1.Play;
end;

procedure TForm1.Timer1Timer(Sender: TObject);
begin
  if TimeOver = 10 then begin
    MediaPlayer1.Eject;
    MediaPlayer1.Close;
    Timer1.Enabled := False;
  end
  else Inc(TimeOver);
end;
```

- Метод `Save` (`procedure Save`) сохраняет загруженную мультимедийную информацию в файле, указанном в свойстве `FileName`. Метод `Save` игнорируется при использовании устройств, не поддерживающих запись в файлах, например, видеодисков.

После завершения, `Save` запоминает числовой код ошибки в свойстве `Error` и создает соответствующее сообщение в свойстве `ErrorMessage`.

Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `Save` завершился. Свойство `Notify` определяет, сгенерировал ли `Save` событие `OnNotify`.

- Метод `PauseOnly` (`procedure PauseOnly`) приостанавливает работу мультимедийного устройства. Вызов `PauseOnly`, если устройство уже приостановлено, приводит к тому, что устройство остается в состоянии паузы.

  После завершения, метод `PauseOnly` запоминает числовой код ошибки в свойстве `Error` и создает соответствующее сообщение в свойстве `ErrorMessage`.

  Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `PauseOnly` завершился. Свойство `Notify` определяет, сгенерировал ли `PauseOnly` событие `OnNotify`.

Следующий код программы (листинг 4.9) иллюстрирует различие между методами `Pause` и `PauseOnly`. После второго обращения к методу `Pause`, `MediaPlayer1` продолжает проигрывание. После второго обращения к `PauseOnly`, `MediaPlayer1` будет оставаться в состоянии паузы.

Листинг 4.9. Пример использования метода *PauseOnly*

```
with MediaPlayer1 do
begin
  MediaPlayer1.Play;
  MediaPlayer1.Pause;
  MediaPlayer1.Pause;
  MediaPlayer1.PauseOnly;
  MediaPlayer1.PauseOnly;
end;
```

- Метод `Resume` (`procedure Resume`) возобновляет воспроизведение или запись на приостановленном мультимедийном устройстве. Метод вызывается нажатием кнопки **Pause** панели управлении медиа во время выполнения, когда устройство приостановлено.

  После завершения, `Resume` запоминает числовой код ошибки в свойстве `Error` и создает соответствующее сообщение в свойстве `ErrorMessage`.

  Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `Resume` завершился. Свойство `Notify` определяет, сгенерировал ли метод `Resume` событие `OnNotify`.

- Метод `Rewind` (`procedure Rewind`) устанавливает текущую позицию на начало медиаконтента, которое определено в свойстве `Start`.

  После завершения, метод `Rewind` запоминает числовой код ошибки в свойстве `Error` и создает соответствующее сообщение в свойстве `ErrorMessage`.

Свойство `Wait` определяет, было ли управление возвращено приложению, прежде чем метод `Rewind` завершился. Свойство `Notify` определяет, сгенерировал ли `Rewind` событие `OnNotify`.

В следующем примере (листинг 4.10) используется медиаплеер и кнопка на форме. Если пользователь нажимает кнопку, то звуковой файл формата WAV переходит к началу и начинает воспроизводиться.

Листинг 4.10. Пример использования метода *Rewind*

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  with MediaPlayer1 do begin
    DeviceType := dtWaveAudio;
    AutoRewind := False;
    FileName := 'CHIMES.WAV';
  end;
  Button1.Caption := 'Rewind and Play';
  Button1.Width := 130;
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  MediaPlayer1.Rewind;
  MediaPlayer1.Play;
end;
```

- Свойство `TrackLength` содержит значение длины дорожки.

  ```
  property TrackLength[TrackNum: Integer]: Longint
  ```

  Значение `TrackLength` соответствует длине дорожки, указанной индексом `TrackNum`, и определено согласно текущему формату времени, который задается свойством `TimeFormat`.

В следующем примере (листинг 4.11) звуковой WAV-файл проигрывается от начала и только до середины.

Листинг 4.11. Пример использования метода *TrackLength*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
    FileName := 'CARTOON.WAV';
    Open;
    EndPos := TrackLength[1] div 2;
    Play;
  end;
end;
```

- Свойство `TrackPosition` сообщает стартовое положение на дорожке, указанной индексом `TrackNum`.

  ```
  property TrackPosition[TrackNum: Integer]: Longint
  ```

  Значение `TrackPosition` определено согласно текущему формату времени, который определен свойством `TimeFormat`.

- Свойство `Capabilities` определяет характеристики открытого мультимедийного устройства.

  ```
  property Capabilities: TMPDevCapsSet
  type  TMPDevCaps = (mpCanStep, mpCanEject, mpCanPlay,
  mpCanRecord, mpUsesWindow);
  TMPDevCapsSet = set of TMPDevCaps;
  ```

  Возможные значения этих характеристик приведены в табл. 4.3.

Таблица 4.3. *Характеристики мультимедийного устройства*

Значение	Характеристика
`MpCanEject`	Может выталкивать (выбрасывать) носитель
`MpCanPlay`	Может воспроизводить медиаинформацию
`MpCanRecord`	Может выполнять запись
`MpCanStep`	Может пошагово перемещаться вперед или назад
`MpUsesWindow`	Использует окно для воспроизведения

В настоящее время нет никакого способа проверить, может ли устройство пошагово перемещаться вперед или назад. Характеристика `mpCanStep` известна только для устройств, тип которых указан в свойстве `DeviceType` (`Animation`, `AVI Video`, `Digital Video`, `Overlay` или `VCR`).

Следующий код определяет, использует ли открытое устройство `MediaPlayer1` окно для воспроизведения. Если это так, то прокрутка выводится на форме с именем `Form2`.

```
if mpUsesWindow in MediaPlayer1.Capabilities then
MediaPlayer1.Display := Form2;
```

- Свойство `Error` определяет `MCI`-код ошибки, возвращенный методом управления.

  ```
  property Error: Longint
  ```

  Значение свойства `Error` соответствует `MCI`-коду ошибки, возвращенным самым последним методом управления (`Back`, `Close`, `Eject`, `Next`, `Open`, `Pause`, `PauseOnly`, `Play`, `Previous`, `StartRecording`, `Resume`, `Rewind`, `Step` или `Stop`). Свойство `ErrorMessage` содержит описание ошибки.

  Код ошибки совпадает со значением, возвращаемым функцией `mciSendCommand`. Сообщение с описанием ошибки запоминается в свойстве `ErrorMessage`.

Значение свойства `Error` равно `0` в случае отсутствия ошибки и отлично от `0` при наличии ошибки. Если ошибка происходит при открытии устройства, возникает исключительная ситуация `EMCIDeviceError`.

Следующий код (листинг 4.12) открывает `MediaPlayer1`. Если возникает ошибка, в окне сообщения выдается номер ошибки и сообщение с ее описанием.

Листинг 4.12. Пример использования свойства *Error*

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var MyErrorString: String;
begin
  try
    MediaPlayer1.Open;
  except
    MyErrorString:='ErrorCode:'+IntToStr(Error)+#13#10;
    MessageDlg(MyErrorString+MediaPlayer1.ErrorMessage,
      mtError,[mbOk],0);
  end;
end;
```

- Свойство `ErrorMessage` описывает код ошибки, записанный в свойстве `Error`.

  ```
  property ErrorMessage: string
  ```

  Это свойство создает сообщение об ошибке, описывающее код ошибки, возвращенный от самого последнего метода управления медиа (`Back`, `Close`, `Eject`, `Next`, `Open`, `Pause`, `PauseOnly`, `Play`, `Previous`, `StartRecording`, `Resume`, `Rewind`, `Step` или `Stop`).

- Свойство `Start` определяет стартовое положение в пределах загруженного в настоящее время медиаконтента.

  ```
  property Start: Longint
  ```

  Свойство `Start` определяет начальную позицию для устройств, которые не используют дорожки или начало первой дорожки для устройств, их использующих. Свойство определено, когда мультимедийное устройство открыто методом `Open`. Данное свойство определено согласно текущему формату времени, описанному в свойстве `TimeFormat`. Свойство имеет атрибут "только для чтения" во время выполнения и недоступно на стадии проектирования.

- Свойство `Length` определяет длину медиаконтента в открытом мультимедийном устройстве. Длина соответствует текущему формату времени, который определен свойством `TimeFormat`.

  ```
  property Length: Longint
  ```

В следующем примере (листинг 4.13) объявлена запись `HMSRec` с четырьмя полями типа `byte`. Если значение свойства `TimeFormat` равно `tfHMS`, первое поле определяет часы, второе определяет минуты, третье поле определяет секунды, а четвертое поле

соответствует старшему неиспользованному байту формата времени `tfHMS`. Переменная типа `LongInt` переопределяет поля в записи `HMSRec`. Отсчет часов, минут и секунд от начальной позиции загруженного медиаконтента отображается в метках после нажатия кнопки.

Листинг 4.13. Пример использования свойства *Length*

```
type
  HMSRec = record
    Hours: byte;
    Minutes: byte;
    Seconds: byte;
    NotUsed: byte;
  end;

procedure TForm1.Button1Click(Sender: TObject);
var  TheLength: LongInt;
begin
  { Установите формат времени - имейте в виду, что
  некоторые устройства не поддерживают tfHMS }
  MediaPlayer1.TimeFormat := tfHMS;
  { Сохраните длину загруженного носителя в TheLength }
  TheLength := MediaPlayer1.Length;
  with HMSRec(TheLength) do
    { Приведение типа TheLength как записи HMSRec }
  begin
    Label1.Caption := IntToStr(Hours);
    { Отобразите часы в Label1 }
    Label2.Caption := IntToStr(Minutes);
    { Отобразите минуты в Label2 }
    Label3.Caption := IntToStr(Seconds);
    { Отобразите секунды в Label3 }
  end;
end;
```

- Свойство `Tracks` указывает, сколько дорожек доступно в открытом мультимедийном устройстве. Свойство не определено для устройств, которые не используют дорожки.

  ```
  property Tracks: Longint
  ```

- Свойство `Frames` определяет число фрагментов (фреймов) метода `Step` при движении вперед или метода `Back` при пошаговом движении назад. По умолчанию значение `Frames` соответствует десяти процентам длины загруженного медиафайла, определенной свойством `Length`.

  ```
  property Frames: Longint
  ```

Примечание

Определение фрейма зависит от типа мультимедийного устройства. Для дисплейных устройств фрейм — это одна картинка.

- Свойство `TMPModes` определяет состояние мультимедийного устройства, используемого с `TMediaPlayer`.

```
property Mode: TMPModes
type TMPModes = (mpNotReady, mpStopped, mpPlaying, mpRecording,
mpSeeking, mpPaused, mpOpen);
```

Таблица 4.4 содержит возможные значения свойства `TMPModes`.

Таблица 4.4. *Возможные значения свойства TMPModes*

Значение	Состояние
MpNotReady	Не готово
MpStopped	Остановлено
MpPlaying	Проигрывание
MpRecording	Запись
MpSeeking	Поиск
MpPaused	Пауза
MpOpen	Устройство открыто

В следующем примере (листинг 4.14) объявляется массив строк с именем `ModeStr`, проиндексированный типом `TMPModes`. В заголовок формы выносится строка, описывающая текущее состояние устройства. Обратите внимание, что перед выполнением, значение свойства `Notify` компонента `TMediaPlayer1` устанавливается равным `true`.

Листинг 4.14. Пример использования свойства *TMPModes*

```
const
  ModeStr: array[TMPModes] of string = ('Not ready','Stopped',
    'Playing', 'Recording', 'Seeking','Paused', 'Open');

procedure TForm1.MediaPlayer1Notify(Sender:
TObject);
begin
  with Sender as TMediaPlayer do begin
    Form1.Caption := ModeStr[Mode];
    { Мы должны восстановить значение свойства Notify = Верно }
    Notify := True;
  end;
end;
```

- Свойство `Position` определяет текущее положение в пределах загруженного медиаконтента.

  ```
  property Position: Longint
  ```

  Значение `Position` определено по текущему формату времени, который определен в свойстве `TimeFormat`.

  По умолчанию значение свойства `Position` соответствует началу. Если устройство поддерживает несколько дорожек, то значение `Position` по умолчанию соответствует началу первой дорожки.

- Свойство `Wait` определяет, вернул ли метод управления медиаконтентом (`Back`, `Close`, `Eject`, `Next`, `Open`, `Pause`, `PauseOnly`, `Play`, `Previous`, `StartRecording`, `Resume`, `Rewind`, `Step` или `Stop`) управление приложению по завершении метода. Свойство `Wait` доступно только во время выполнения программы.

  ```
  property Wait: Boolean
  ```

  Если `Wait = True`, то медиаплеер ждет завершения следующего метода управления перед возвращением в приложение. Если `Wait = False`, приложение не будет ждать завершения следующего метода управления.

  Свойство `Wait` воздействует только на следующий метод управления, вызванный после установки его значения. Значение `Wait` должно быть повторно установлено для того, чтобы воздействовать на любой последующий запрос к методу управления медиаконтентом.

  По умолчанию, для метода `Play` и функции `StartRecording` устанавливается `Wait = False`. Значение `Wait` должно быть установлено равным `True` перед вызовом `Play` или `StartRecording`, чтобы предотвратить возврат в приложение до завершения проигрывания или записи. По умолчанию все другие методы управления устанавливают `Wait = True`.

> ***Примечание***
>
> Значение свойства `Wait` обычно устанавливается равным `false` только в случае, если ожидается, что следующий метод управления медиаконтентом будет занимать длительное время, для того, чтобы приложение могло выполнять другой код прежде, чем завершится метод. Если значение `Wait` установлено равным `false`, рекомендуется установить `Notify = true`, чтобы уведомить приложение о завершении метода управления.

В следующем примере (листинг 4.15) дважды проигрывается звуковой WAV-файл с именем NI!.WAV. Первый запрос на проигрывание не возвращает управление приложению до тех пор, пока не завершился файл. Обратите внимание на то, что, если будет удалена строка кода `Wait:=True`, то проигрывание выполнится один раз.

Листинг 4.15. Пример использования свойства *Wait*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
```

```
  with MediaPlayer1 do begin
    FileName := 'ni!.wav';
    AutoRewind := True;
    Open;          //Открыть медиаплеер
    try
      Wait := True;
      Play;
      Play;
//Проигрывание снова после начала прогона завершено
    finally
      Close; // Закрыть медиаплеер
    end;
  end;
end;
```

- Свойство `Notify` определяет, сгенерировал ли следующий вызов метода управления медиаконтентом (`Back`, `Close`, `Eject`, `Next`, `Open`, `Pause`, `PauseOnly`, `Play`, `Previous`, `StartRecording`, `Resume`, `Rewind`, `Step` или `Stop`) событие `OnNotify` по завершении метода.

  ```
  property Notify: Boolean
  ```

  Если `Notify = True`, следующий метод управления приводит к событию `OnNotify` после завершения и запоминает сообщение уведомления в свойстве `NotifyValue`. Если `Notify = False`, метод не вызывает событие `OnNotify`, а значение свойства `NotifyValue` остается неизменным.

  Свойство воздействует только на следующий запрос метода управления. После события `OnNotify` значение `Notify` должно быть установлено повторно, чтобы воздействовать на любые последующие вызовы методов управления медиаконтентом.

  По умолчанию, для методов `Play` и `StartRecording` установлено значение `Notify = True`.

  Установите значение `Notify = False` перед вызовом методов `Play` или `StartRecording` для того, чтобы игнорировать событие `OnNotify` до завершения проигрывания или записи. По умолчанию все другие функции методов управления медиаконтентом соответствуют значению `Notify = False`.

Замечание 1

Установите `Notify = True`, если ожидается, что следующий метод управления медиаконтентом будет занимать длительное время, и приложение будет уведомлено, когда метод управления завершится. Если `Notify = True`, рекомендуется установить `Wait = False` для того, чтобы управление вернулось приложению прежде, чем завершится метод управления медиаконтентом.

Замечание 2

При попытке возобновить состояние устройства, которое не поддерживает метод `Resume`, устройство восстанавливается так, как будто вызывался метод `Play`. Если `Notify = True` перед вызовом `Resume` или любого другого метода управления, то `Notify` не вызывает метод `Resume`. Метод `Resume` не вызывает события `OnNotify` после завершения и значение `NotifyValue` остается неизменным.

В следующем примере (листинг 4.16) объявляется массив строк с именем `ModeStr`, индексированный типом `TMPModes`. В заголовке формы отображается строка, описывающая текущее состояние устройства.

Обратите внимание: перед выполнением примера следует установить значение свойства `Notify` для `TMediaPlayer1` равным `true`.

Листинг 4.16. Пример использования свойства *Notify*

```
const
  ModeStr: array[TMPModes] of string = ('Not ready','Stopped',
    'Playing', 'Recording', 'Seeking','Paused', 'Open');
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  with Sender as TMediaPlayer do begin
    Form1.Caption := ModeStr[Mode];
    { мы должны восстановить свойство Notify в True }
    Notify := True;
  end;
end;
```

- Свойство `NotifyValue` сообщает результат последнего метода управления медиаконтентом (`Back`, `Close`, `Eject`, `Next`, `Open`, `Pause`, `PauseOnly`, `Play`, `Previous`, `StartRecording`, `Resume`, `Rewind`, `Step` или `Stop`), который требовал уведомления.

  ```
  property NotifyValue: TMPNotifyValues
  type TMPNotifyValues = (nvSuccessful, nvSuperseded,
  nvAborted, nvFailure);
  ```

  Для того чтобы запрашивать уведомление, установите `Notify = True` перед вызовом метода управления медиа.

  Свойство `TMPNotifyValues` определяет значения уведомления для мультимедийного устройства, используемого с `TMediaPlayer`. Возможные значения этого свойства приведены в табл. 4.5.

***Таблица 4.5.** Возможные значения свойства NotifyValue*

Значение	Результат
`NvSuccessful`	Команда завершилась успешно
`NvSuperseded`	Команда была заменена другой командой

Таблица 4.5 *(окончание)*

Значение	Результат
NvAborted	Команда была прервана пользователем
NvFailure	Команда не выполнена

- Свойство StartPos определяет в пределах загруженного медиаконтента позицию, с которой надо начать проигрывание или запись. Значение StartPos определяется с использованием текущего формата времени, определенного в свойстве TimeFormat.

  ```
  property StartPos: Longint
  ```

  Свойство StartPos действует только на следующий метод Play или StartRecording, вызываемый после установки StartPos. Должна быть выполнена переустановка StartPos для того, чтобы воздействовать на любые последующие запросы Play или StartRecording. Свойство StartPos не воздействует на текущую позицию, указанную в свойстве Position, пока не будет вызван следующий метод Play или StartRecording.

Далее приводится код, в котором устанавливается формат времени "часы-минуты-секунды" (TimeFormat:=tfHMS), отрывается звуковой файл Eug.WAV и устанавливаются начало и конец звучания в миллисекундах.

```
try
  MediaPlayer1.TimeFormat:= tfHMS;
  MediaPlayer1.FileName:='Eug.wav';
  MediaPlayer1.Open;
  MediaPlayer1.StartPos:=9000;
  MediaPlayer1.EndPos:=32000;
  MediaPlayer1.Play;
except
  on EMCIDeviceError do;
end;
```

- Свойство EndPos определяет положение в пределах загруженного медиаконтента, в котором следует прекратить воспроизведение или запись. EndPos определяется с использованием текущего формата времени, определенного в свойстве TimeFormat.

  ```
  property EndPos: Longint
  ```

  Свойство EndPos воздействует только на следующий метод Play или StartRecording, вызываемый после установки EndPos. Переустановите EndPos, для того, чтобы воздействовать на любые последующие вызовы методов Play или StartRecording.

В следующем примере (листинг 4.17) проигрывается звуковой WAV-файл с начала до середины.

Листинг 4.17. Пример использования свойства *EndPos*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
    FileName := 'D:\WINAPPS\SOUNDS\CARTOON.WAV';
    Open;
    EndPos := TrackLength[1] div 2;
    Play;
  end;
end;
```

- Свойство `DeviceID` — идентификатор открытого устройства. Значение этого свойства определено для устройства, открытого методом `Open`.

  ```
  property DeviceID: Word
  ```

  Если нет открытого устройства, то `DeviceID = 0`.

- Свойство `TimeFormat` определяет формат позиционирования внутри медиаконтента.

  ```
  property TimeFormat: TMPTimeFormats;
  type TMPTimeFormats = (tfMilliseconds, tfHMS, tfMSF, tfFrames,
  tfSMPTE24, tfSMPTE25, tfSMPTE30, tfSMPTE30Drop, tfBytes, tfSamples,
  tfTMSF);
  ```

  Свойство `TimeFormat` определяет, как интерпретируются свойства `StartPos`, `Length`, `Position`, `Start` и `EndPos`. Например, если значение свойства `Position = 180` и значение `TimeFormat` равно `tfMilliseconds`, текущее положение — 180 миллисекунд от начальной позиции медиаконтента. Если значение свойства `Position = 180` и значение `TimeFormat` равно `tfMSF`, то текущее положение — 180 минут.

  Конкретное устройство может поддерживать не все возможные форматы позиционирования. При попытке установить неподдерживаемый формат, назначение игнорируется.

  Текущая временная информация всегда представляется целым числом длиной 4 байта. В некоторых форматах возвращенная временная информация представляет собой не одно целое число, а отдельные байты информации, упакованные в длинном целом числе.

- Свойство `TMPTimeFormats` определяет форматы времени для устройства мультимедиа, используемого с `TMediaPlayer`. Таблица 4.6 содержит список возможных значений свойства `TimeFormat`.

Таблица 4.6. *Возможные значения свойства TimeFormat*

Значение	Формат времени
`tfMilliseconds`	Миллисекунды, представленные как переменная целого типа длиной 4 байта

Таблица 4.6 *(продолжение)*

Значение	Формат времени
tfHMS	Часы, минуты и секунды, упакованные как переменная целого типа длиной 4 байта. Значения от младшего к старшему байту соответственно: часы (младший байт); минуты; секунды; неиспользованный (старший байт)
tfMSF	Минуты, секунды и фреймы (фрагменты), упакованные как переменная целого типа длиной 4 байта. Значения от младшего к старшему байту соответственно: Минуты (младший байт) Секунды Фреймы (фрагменты) Неиспользованный (старший байт)
tfFrames	Фреймы (фрагменты), представленные как переменная целого типа длиной 4 байта
tfSMPtE24	24-фреймовый формат SMPTE упаковывает значения в виде переменной целого типа длиной 4 байта. Значения от младшего к старшему байту соответственно: Часы (младший байт) Минуты Секунды Фреймы (фрагменты) (старший байт) SMPTE — (Society of Motion Oicture and Television Engineers — Общество кино- и телевизионных инженеров). Стандарт SMPTE предполагает три варианта: 24, 25, и 30 фреймов в секунду. Время выражается в абсолютном формате времени (часы, минуты, секунды и фреймы)
tfSMPtE25	25-фреймовый формат SMPTE упаковывает значения в виде переменной целого типа длиной 4 байта так же
tfSMPtE30	30-фреймовый формат SMPTE упаковывает значения в виде переменной целого типа длиной 4 байта так же
tfSMPtE30Drop	Формат 30-drop-frame SMPTE упаковывает значения в виде переменной целого типа длиной 4 байта в том же порядке, как SMPTE с 24 фреймами
tfBytes	Байтовый формат как переменная целого типа длиной 4 байта
tfSamples	Образцы, представленные как переменная целого типа длиной 4 байта

Таблица 4.6 *(окончание)*

Значение	Формат времени
tftMSF	Дорожки, минуты, секунды и фреймы, упакованные как переменная целого типа длиной 4 байта. Значения от младшего к старшему байту соответственно: Дорожки (младший байт) Минуты Секунды Фреймы (старший байт)

Замечание 1

MCI использует непрерывную нумерацию дорожек.

Замечание 2

Функции, обеспечивающие MCI для расшифровки целого числа с 4 байтами, указанного в данных форматах времени, представлены в MCI Macros для Encoding and Decoding Time Data в файле справки MMSYSTEM.HLP.

Следующий код (листинг 4.18) содержит запись `HMSRec` четырьмя байтовыми полями. Если значение свойства `TimeFormat равно tfHMS`, первое поле определяет часы, второе — минуты, третье — секунды и четвертое поле соответствует старшему неиспользованному байту временного формата `tfHMS`. Переменная типа `LongInt` переопределяет запись `HMSRec`, тогда отсчет часов, минут и секунд от начала загруженного медиаконтента отображается в метках при нажатии кнопки.

Листинг 4.18. Пример использования свойства *TimeFormat*

```
type
  HMSRec = record
    Hours: byte;
    Minutes: byte;
    Seconds: byte;
    NotUsed: byte;
  end;

procedure TForm1.Button1Click(Sender: TObject);
var TheLength: LongInt;
begin
//   Установите формат времени - имейте в виду, что
//   некоторые устройства не поддерживают tfHMS
  MediaPlayer1.TimeFormat := tfHMS;
//   Сохраните длину загруженного носителя в переменной
  TheLength := MediaPlayer1.Length;
```

```
  with HMSRec(TheLength) do
//   Приведение типа TheLength к HMSRec
  begin
    Label1.Caption := IntToStr(Hours);
//   Отобразить часы в Label1
    Label2.Caption := IntToStr(Minutes);
//   Отобразить минуты в Label2
    Label3.Caption := IntToStr(Seconds);
//   Отобразить секунды в Label3
  end;
end;
```

- Свойство `DisplayRect` определяет прямоугольную область экрана, указанную свойством `Display`, которая используется для воспроизведения медиаинформации.

  ```
  property DisplayRect: TRect
  ```

  Воспроизведение осуществляется в заданной прямоугольной области на форме, определенной записью `TRect` в `DisplayRect`. Функция `Rect` может использоваться для создания записи `TRect`. Качество воспроизведения обычно бывает лучше, если прямоугольник для показа воспроизведения имеет размеры, принятые по умолчанию. Чтобы установить размеры, принятые по умолчанию, поместите прямоугольник в верхнем левом углу экрана и используйте значение `0`, `0` для нижнего правого угла.

  Свойство `DisplayRect` игнорируется, если `Display = nil`.

  Примеры средств мультимедиа, использующих для воспроизведения окно: мультипликация `(Animation)`, AVI-видео (`AVI Video`), цифровое видео (`Digital Video`), оверлей (`Overlay`), и `VCR` (VCR) — кассетный видеомагнитофон.

  ПРИМЕЧАНИЕ

  Значение *DisplayRect* может быть установлено после того, как устройство медиа открыто.

- Свойство `ColoredButtons` определяет, какие кнопки управления медиаплеером окрашены.

  ```
  property ColoredButtons: TButtonSet
  type TMPBtnType = (btPlay, btPause, btStop, btNext, btPrev, btStep,
  btBack, btRecord, btEject);
  TButtonSet = set of TMPBtnType;
  ```

  Если кнопка не окрашивается свойством `ColoredButtons`, то она отображается в черно-белом цвете. По умолчанию все кнопки управления проигрывателем окрашены. Возможные значения свойства `ColoredButtons` соответствуют принятым обозначениям кнопок компонента `TmediaPlayer`. Описание всех кнопок компонента `TmediaPlayer` было приведено в табл. 4.2.

- Свойство `EnabledButtons` определяет, какие кнопки управления медиаплеером задействованы и используются.

```
property EnabledButtons: TButtonSet
type TMPBtnType = (btPlay, btPause, btStop, btNext, btPrev, btStep,
btBack, btRecord, btEject);
TButtonSet = set of TMPBtnType;
```

  Задействованная кнопка окрашивается и может использоваться. Незадействованная кнопка притемняется и не используется. Если кнопка не задействована с помощью `EnabledButtons`, она недоступна. По умолчанию все кнопки задействованы.

  Если свойство `AutoEnable=True`, то оно подавляет действие свойства `EnabledButtons`. Кнопки автоматически задействуются медиаплеером, и любые назначения свойства `EnabledButtons` аннулируются.

В следующем примере (листинг 4.19) все кнопки `MediaPlayer1` переводятся в незадействованное состояние, когда кнопка `BitBtn1` нажата.

Листинг 4.19. Пример использования свойства *EnabledButtons*

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
    AutoEnable := False;
    EnabledButtons := [];
  end;
end;
```

- Свойство `VisibleButtons` (листинг 4.20) определяет, какие из кнопок медиаплеера являются видимыми.

```
property VisibleButtons: TButtonSet
type TMPBtnType = (btPlay, btPause, btStop, btNext, btPrev, btStep,
btBack, btRecord, btEject);
TButtonSet = set of TMPBtnType;
```

  Если видимость кнопки не определена с помощью свойства `VisibleButtons`, кнопка не появляется на панели управлении медиаплеера. По умолчанию при добавлении компонента медиаплеера в форму все кнопки видимы.

Листинг 4.20. Пример использования свойства *VisibleButtons*

```
begin
  With MediaPlayer1 do begin
    Filename := 'c:\win95\roaring.wav';
    // определите звуковой файл
    ShowHint := True; //показать подсказки
    Shareable := True;
```

```
    // позвольте другим использовать тот же файл .wav
    VisibleButtons := [btPlay, btStop];
    // показать две кнопки
    Open; // поиграйте это
  end;
end;
```

- Свойство `AutoEnable` определяет, может ли `TMediaPlayer` автоматически задействовать или выводить отдельные кнопки.

  ```
  property AutoEnable: Boolean
  ```

  Используйте `AutoEnable`, чтобы автоматически задействовать или выводить отдельные кнопки управления в компоненте медиаплеера. Если `AutoEnable = True`, медиаплеер автоматически задействует или выводит свои управляющие кнопки. Медиаплеер определяет, какие кнопки задействованы, по текущему состоянию, определенному свойством `Mode`, и текущему типу устройства, определенному свойством `DeviceType`. Свойство `AutoEnable` аннулирует (подавляет) свойство `EnabledButtons`.

  Если `AutoEnable = false`, медиаплеер не позволяет автоматически задействовать или выводить кнопки. Кнопки должны задействоваться или выводиться свойством `EnabledButtons`.

В табл. 4.7 показано, могут ли кнопки автоматически задействоваться или выводиться при разных состояниях устройства.

Таблица 4.7. *Возможные значения свойства AutoEnable*

Кнопка/ Состояние	Play	Record	Pause	Stop	Not Open
Back	Enabled	Enabled	Enabled	Enabled	Disabled
Eject	Enabled	Enabled	Enabled	Enabled	Disabled
Next	Enabled	Enabled	Enabled	Enabled	Disabled
Pause	Enabled	Enabled	Enabled	Disabled	Disabled
Play	Disabled	Disabled	Enabled	Enabled	Disabled
Prev	Enabled	Enabled	Enabled	Enabled	Disabled
Record	Disabled	Disabled	Enabled	Enabled	Disabled
Step	Enabled	Enabled	Enabled	Enabled	Disabled
Stop	Enabled	Enabled	Disabled	Disabled	Disabled

В следующем примере (листинг 4.21) все кнопки `MediaPlayer1` становятся незадействованными после нажатия кнопки `BitBtn1`.

Листинг 4.21. Пример использования свойства *AutoEnable*

```
procedure TForm1.BitBtn1Click(Sender: TObject);
begin
  with MediaPlayer1 do begin
    AutoEnable := False;
    EnabledButtons := [];
  end;
end;
```

- Свойство `AutoOpen` определяет, открывается ли медиаплеер автоматически при запуске приложения.

  ```
  property AutoOpen: Boolean
  ```

  Если `AutoOpen=True`, то медиаплеер пытается открыть мультимедийное устройство, определенное свойством `DeviceType` (или `FileName`, если `DeviceType = dtAutoSelect`), когда форма, содержащая компонент медиаплеера, создается в момент выполнения программы. Если `AutoOpen = false`, то устройство должно открываться вызовом метода `Open`.

  В случае ошибки при открытии устройства, возникает исключительная ситуация `EMCIDeviceError`, которая содержит сообщение об ошибке. По завершении, цифровой код ошибки запоминается в свойстве `Error`, а соответствующее сообщение об ошибке хранится в свойстве `ErrorMessage`.

- Свойство `AutoRewind` определяет, выполняется ли перемотка медиаплеера перед проигрыванием или записью `property AutoRewind: Boolean`.

  Если `AutoRewind = True` и текущая позиция — в конце медиаконтента, `Play` или `StartRecording` перемещает текущую позицию в начало медиаконтента перед проигрыванием или записью. Если `AutoRewind = False`, пользователь должен нажать кнопку **Prev** или написать код, в котором необходимо вызвать `Previous` для перехода к началу.

  > ***ПРИМЕЧАНИЕ***
  >
  > Если свойствам `StartPos` или `EndPos` были присвоены значения или мультимедийное устройство использует дорожки, `AutoRewind` не окажет действия на проигрывание или запись: при вызовах методов `Play` или `StartRecording` текущая позиция остается в конце медиаконтента.

  Следующий код запускает воспроизведение медиаплеера. Если `AutoRewind = False`, то вызывается `Previous` для того, чтобы выполнить перемотку после завершения `Play`.

  ```
  MediaPlayer.Wait := True;
  MediaPlayer.Play;
  If not MediaPlayer.AutoRewind then MediaPlayer.Previous;
  ```

- Свойство `DeviceType` (листинг 4.22) определяет тип устройства мультимедиа для того, чтобы открыть медиаплеер методом `Open`.

```
property DeviceType: TMPDeviceTypes;
type TMPDeviceTypes = (dtAutoSelect, dtAVIVideo, dtCDAudio, dtDAT,
dtDigitalVideo, dtMMMovie, dtOther, dtOverlay, dtScanner, dtSequencer,
dtVCR, dtVideodisc, dtWaveAudio);
```

  Тип `TMPDeviceTypes` соответствует типам устройств мультимедиа, которые могут открываться компонентом `TMediaPlayer`. По умолчанию его значение равно `dtAutoSelect`.

  Если `DeviceType = dtAutoSelect`, то тип устройства определяется расширением файла `FileName`. Если с расширением не связан ни один из типов устройств, то правильный тип устройства должен явно определяться установкой `DeviceType` в значение, отличное от `dtAutoSelect`.

  Если устройство установлено, оно обычно связывается с соответствующим расширением имени файла. Связывание отмечается в реестре или в файле SYSTEM.INI. (Как связать расширение файлового имени с устройством, изложено в [102]).

Листинг 4.22. Пример использования свойства *DeviceType*

```
var sWinDir: String;
    iLength: Integer;
begin
  iLength := 255;
  setLength(sWinDir, iLength);
  iLength:=GetWindowsDirectory(PChar(sWinDir),iLength);
  setLength(sWinDir, iLength);
  With MediaPlayer1 do begin
    Filename := sWinDir + '\Clock.avi';
    // определите видео файл
    DeviceType := dtAVIVideo;
    // установите Device совместимым с AVI
    Display := Panel1;
    // Установите дисплейное устройство на TPanel
    Open;
    Play;
  end;
end;
```

- Свойство `Display` (листинг 4.23) используется для создания окна отображения вывода.

```
property Display: TWinControl
```

  Назначьте имя оконного управления как форму или панель в `Display` для того, чтобы отобразить вывод на этом элементе.

По умолчанию `Display=nil` и это значит, что устройство создает свое собственное окно для отображения вывода.

Примеры устройств мультимедиа, которые используют окно для отображения вывода — мультипликация (`Animation`), видео-`AVI` (`AVI Video`), цифровое видео (`Digital Video`), оверлей (`Overlay`) и `VCR`-видеомагнитофон (`VCR`).

Листинг 4.23. Пример использования свойства *Display*

```
var sWinDir: String;
              iLength: Integer;
begin
  iLength := 255;
  setLength(sWinDir, iLength);
  iLength:=GetWindowsDirectory(PChar(sWinDir),iLength);
  setLength(sWinDir, iLength);
  With MediaPlayer1 do begin
    Filename := sWinDir + '\Clock.avi';
    // определите видеофайл
    DeviceType := dtAVIVideo;
    // установите Device совместимым с AVI
    Display := Panel1;
    // Установите дисплейное устройство на TPanel
    Open;
    Play;
  end;
end;
```

- Свойство `FileName` определяет имя файла на носителе медиаинформации, который должен открываться методом `Open`, или имя файла, сохраняемого методом `Save`.

  ```
  property FileName: string
  ```

  Чтобы определить значение `FileName`, используйте диалог `file open dialog box`.

Следующий пример (листинг 4.24) позволяет пользователю выбирать в диалоговом окне **Открытие файла** видеофайл формата AVI, используя метод `OpenDialog1`, и открывать этот файл в компоненте `MediaPlayer1`. Затем используются кнопки **Forward** и **Back**, чтобы пошагово пройти через фреймы клипа в каждом направлении.

Листинг 4.24. Пример использования свойства *FileName*

```
procedure TForm1.OpenClick(Sender: TObject);
begin
  OpenDialog1.DefaultExt := 'AVI';
  OpenDialog1.Filename := '*.avi';
```

```
  if OpenDialog1.Execute then begin
     MediaPlayer1.Filename := OpenDialog1.Filename;
     MediaPlayer1.Open;
  end;
end;

procedure TForm1.BackClick(Sender: TObject);
begin
  MediaPlayer1.Back;
end;

procedure TForm1.ForwardClick(Sender: TObject);
begin
  MediaPlayer1.Next;
end;
```

- Свойство `Shareable` (листинг 4.25) определяет, может ли более чем одно приложение иметь общий доступ к устройству мультимедиа.

  ```
  property Shareable: Boolean
  ```

  Если значение `Shareable = false`, никакие другие компоненты или приложения не могут иметь доступ к устройству. Если `Shareable = true`, то более чем один компонент или приложение могут иметь доступ к устройству. По умолчанию `Shareable = false`.

  Значение `Shareable` устанавливается до открытия устройства. Некоторые устройства являются не разделяемыми. Для таких устройств, в случае, если установлено значение `Shareable = true`, при попытке открыть устройство другим компонентом метод `Open` вызывает исключительную ситуацию, а код ошибки возвращается свойством `Error`.

Листинг 4.25. Пример использования свойства *Shareable*

```
begin
  With MediaPlayer1 do begin
    Filename := 'c:\win95\roaring.wav';
    // определите звуковой файл
    ShowHint := True;
    Shareable := True;
    // позвольте другим использовать тот же файл .wav
    VisibleButtons := [btPlay, btStop];
    Open;
  end;
end;
```

- Событие `OnClick` происходит при нажатии и отпускании кнопки мыши, если указатель мыши находится над одной из управляющих кнопок, или при нажа-

тии клавиши пробел (Spacebar), если фокус расположен на управлении медиаплеера.

```
property OnClick: EMPNotify;
type  TMPBtnType = (btPlay, btPause, btStop, btNext, btPrev, btStep,
btBack, btRecord, btEject);
  EMPNotify = procedure (Sender: TObject; Button: TMPBtnType; var Do-
Default: Boolean) of object;
```

Если фокус установлен на управлении медиаплеера и клавиша Spacebar нажата, пользователь может использовать левую или правую стрелку, чтобы выбрать управляющую кнопку для нажатия.

Тип EMPNotify является указателем метода, который вызывается, когда возникает событие OnClick для компонентов TMediaPlayer. Аргумент Button может принимать одно из следующих значений: btBack, btEject, btNext, btPause, btPlay, btPrev, btRecord, btStep или btStop.

По умолчанию аргумент DoDefault = True. В этом случае вызовы методов управления медиаплеером соответствуют нажатой кнопке. Например, если пользователь нажимает кнопку **Play** (btPlay), вызывается метод Play.

Если DoDefault = False и нажимается кнопка управления медиаплеера, то пользователь должен написать код, который выполняется на OnClick обработчиком события. В табл. 4.8 представлены встроенные методы, соответствующие кнопкам управления медиаплеера.

Таблица 4.8. *Встроенные методы управления медиаплеера*

Кнопка управления	Значение	Вызываемый метод
Play	BtPlay	Play
Record	BtRecord	StartRecording
Stop	BtStop	Stop
Next	BtNext	Next
Prev	BtPrev	Previous
Step	BtStep	Step
Back	BtBack	Back
Pause	BtPause	Pause
Eject	BtEject	Eject

В следующем примере (листинг 4.26) используются метка и медиаплеер на форме. Если пользователь нажимает одну из кнопок медиаплеера, то заголовок метки указывает, какая кнопка была нажата. Для выполнения этого примера надо иметь установленное звуковое CD-устройство.

Листинг 4.26. Пример использования свойства *OnClick*

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  MediaPlayer1.DeviceType := dtCDAudio;
  MediaPlayer1.Open;
  MediaPlayer1.Left := 20;
  MediaPlayer1.Top := 12;
  Label1.Top := 44;
  Label1.Left := 20;
  Label1.Color := clYellow;
  Label1.Font.Name := 'Arial';
  Label1.Caption := 'Click Me';
end;

procedure TForm1.MediaPlayer1Click(Sender: TObject;
  Button: TMPBtnType; var DoDefault: Boolean);
begin
  case Button of
    btPlay : begin
               Label1.Caption := 'Playing';
               Label1.Left := 20;
             end;
    btPause: begin
               Label1.Caption := 'Paused';
               Label1.Left := 48;
             end;
    btStop:  begin
               Label1.Caption := 'Stopped';
               Label1.Left := 76;
             end;
    btNext:  begin
               Label1.Caption := 'Next';
               Label1.Left := 104;
             end;
    btPrev:  begin
               Label1.Caption := 'Previous';
               Label1.Left := 132;
             end;
    btStep:  begin
               Label1.Caption := 'Step';
               Label1.Left := 160;
             end;
    btBack:  begin
               Label1.Caption := 'Back';
```

```
            Label1.Left := 188;
          end;
    btRecord:begin
            Label1.Caption := 'Record';
            Label1.Left := 216;
          end;
    btEject: begin
            Label1.Caption := 'Eject';
            Label1.Left := 244;
          end;
  end;
end;
```

- Событие `OnEnter` (`property OnEnter`) происходит, когда управление получает фокус ввода. Используйте обработчик события `OnEnter`, чтобы выполнить любую специальную обработку, когда визуальный компонент становится активным. Событие `OnEnter` не возникает при переключении между формами или между приложением, которое имеет визуальное управление, и другими приложениями `Windows`.
- Событие `OnPostClick` генерируется после того, как вызван код обработчика события `OnClick`.

  ```
  property OnPostClick: EMPPostNotify;
  type TMPBtnType = (btPlay, btPause, btStop, btNext, btPrev, btStep,
  btBack, btRecord, btEject);
  EMPPostNotify = procedure (Sender: TObject; Button: TMPBtnType) of ob-
  ject;
  ```

  Если во время запуска медиаплеера значение свойства `Wait = True`, то обработчик `OnPostClick` не будет вызван до завершения кода `OnClick`. Если `Wait = False`, то управление может возвратиться в приложение перед завершением кода `OnClick` и событие `OnPostClick` может произойти перед завершением действий, инициализированных событием `OnClick`.

  Например, если пользователь нажимает кнопку `Play` и параметр `DoDefault` обработчика события `OnClick` для медиаплеера равен `True`, то воспроизведение мультимедиа завершилось. Если медиаконтент достаточно длинный, он все еще будет воспроизводиться, когда генерируется событие `OnPostClick`, если `Wait = True`. Если `Wait = False`, то событие `OnPostClick` не произойдет, пока не завершится воспроизведение.

  Метод `EMPPostNotify` вызывается при возникновении события `OnPostClick` для компонента `TMediaPlayer`. Аргумент `Button` этого метода может принимать одно из следующих значений: `btBack`, `btEject`, `btNext`, `btPause`, `btPlay`, `btPrev`, `btRecord`, `btStep` или `btStop`.
- Событие `OnNotify` происходит при завершении метода управления медиаконтентом (`Back`, `Close`, `Eject`, `Next`, `Open`, `Pause`, `PauseOnly`, `Play`, `Previous`,

`Resume`, `Rewind`, `StartRecording`, `Step` или `Stop`), если перед вызовом метода управления свойство `Notify` устанавливалось равным `True`.

```
property OnNotify: TNotifyEvent
```

После возникновения события `OnNotify` свойство `Notify` должно переустанавливаться в значение `True` для следующего возникновения события `OnNotify`.

4.3. Проект с использованием компонента *MediaPlayer*

Проект реализует нечто похожее на презентацию — позволяет проигрывать мультимедийные файлы. Программа работает в двух режимах:

1. Файлы выбираются из диалогового окна и проигрываются.
2. При нажатии кнопки в окне появляется изображение и воспроизводится связанная с ним мелодия или другое звуковое сопровождение. По завершении одной мелодии включается другая, соответственно меняется и изображение в окне вывода.

На рис. 4.10 представлено окно данного проекта **View Media**, предназначенное для воспроизведения медиаинформации.

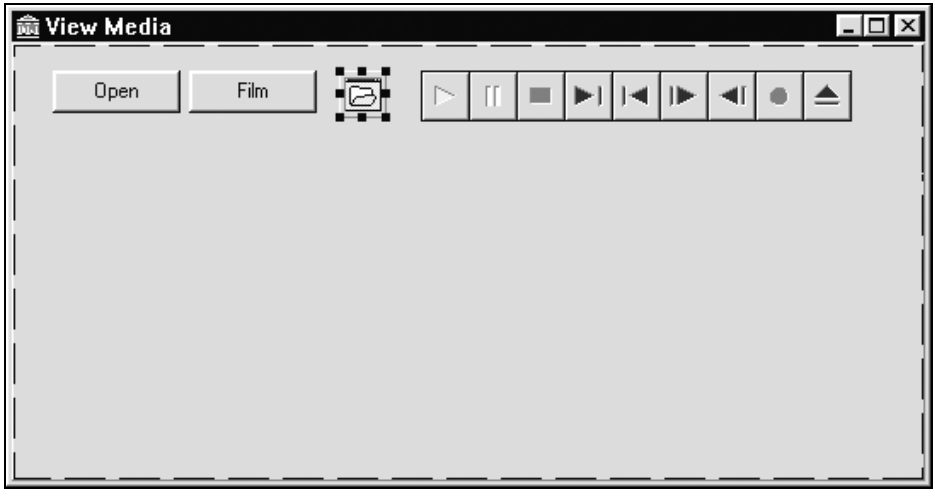


Рис. 4.10. Окно **View Media**

В первом режиме проекта при нажатии кнопки **Open** вызывается метод `OpenDialog1`, который позволяет выбрать мультимедийные файлы (с расширением AVI, WAV или MID).

Во втором режиме при нажатии кнопки **Film** проект работает автоматически, показывая графические файлы 1.BMP, 2.BMP, 3.BMP до тех пор, пока воспроизводятся соответствующие им звуковые файлы 1.WAV, 2.WAV, 3.WAV.

В первом режиме (листинг 4.27) вызывается процедура `Button1Click`, в которой назначается имя файла мультимедиа, назначается `Panel1` для вывода информации (`MediaPlayer1.Display := Panel1`), и запускается видеофильм `MediaPlayer1.Play`.

Листинг 4.27. Вывод файлов в форматах AVI, WAV, MID

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MediaPlayer1.Close;
  if OpenDialog1.Execute then begin
    MediaPlayer1.FileName:=OpenDialog1.FileName;
    MediaPlayer1.Open;
    MediaPlayer1.Display:=Panel1;
    MediaPlayer1.DisplayRect:=
    Rect(2,2,Panel1.Width-4,Panel1.Height-4);
    MediaPlayer1.Play;
  end;
end;
```

Во втором режиме вызывается процедура `Button2Click` (листинг 4.28), с помощью метода `SetImage` (листинг 4.29) выводится первый кадр ролика (графический файл 1.BMP на экране в сопровождении звукового файла 1.WAV), который выполняется, пока метод `MediaPlayer1Notify` (листинг 4.30) не сменит кадр, увеличив значение переменной `Num` на 1.

Листинг 4.28. Обработчик события "нажатие кнопки *Button2*"

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  Num: =1; SetImage;
end;
```

Листинг 4.29. Вывод мультимедиаинформации

```
procedure TForm1.SetImage;
begin
  Bitmap.LoadFromFile(IntToStr(Num)+'.bmp');
  Image1.Canvas.Draw(0,0,Bitmap);
  MediaPlayer1.FileName: =IntToStr(Num)+'.wav';
  MediaPlayer1.Open;
  MediaPlayer1.Play;
end;
```

Листинг 4.30. Обработка события *Notify*

```
procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
  with MediaPlayer1 do
  if (NotifyValue=nvSuccessful) and (Mode=mpStopped)
  then begin
    Inc(Num);
    MediaPlayer1.Close;
    if Num<=3 then SetImage;
  end;
end;
```

Полный текст основного модуля проекта приведен в листинге 4.31.

Листинг 4.31. Полный текст основного модуля проекта с использованием компонента *MediaPlayer*

```
unit Unit1;
interface
uses
       Windows, Messages, SysUtils, Classes, Graphics,
       Controls,Forms, Dialogs, StdCtrls, MPlayer, ExtCtrls;
type
       TForm1 = class(TForm)
              MediaPlayer1: TMediaPlayer;
              OpenDialog1: TOpenDialog;
              Button1: TButton;
              Panel1: TPanel;
              Image1: TImage;
              Button2: TButton;
              procedure Button1Click(Sender: TObject);
              procedure FormCreate(Sender: TObject);
              procedure Button2Click(Sender: TObject);
              procedure MediaPlayer1Notify(Sender: TObject);
       private
              { Private declarations }
              Num: byte;
              procedure SetImage;
       public
             { Public declarations }
       end;
var  Form1: TForm1;
            Bitmap: TBitmap;
implementation
{$R *.DFM}
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
       MediaPlayer1.Close;
       if OpenDialog1.Execute then begin
              MediaPlayer1.FileName:=OpenDialog1.FileName;
              MediaPlayer1.Open;
              MediaPlayer1.Display:=Panel1;
              MediaPlayer1.DisplayRect:=
                     Rect(2,2,Panel1.Width-4,Panel1.Height-4);
              MediaPlayer1.Play;
       end;
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
       Bitmap:=TBitmap.Create;
end;

procedure TForm1.SetImage;
begin
       Bitmap.LoadFromFile(IntToStr(Num)+'.bmp');
       Image1.Canvas.Draw(0,0,Bitmap);
       MediaPlayer1.FileName:=IntToStr(Num)+'.wav';
       MediaPlayer1.Open;
       MediaPlayer1.Play;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
       Num:=1; SetImage;
end;

procedure TForm1.MediaPlayer1Notify(Sender: TObject);
begin
       with MediaPlayer1 do
       if (NotifyValue=nvSuccessful) and (Mode=mpStopped)
       then begin
              Inc(Num);
              MediaPlayer1.Close;
              if Num<=3 then SetImage;
       end;
end;

end.
```

Подход, использованный при реализации этого проекта, может быть использован при создании презентаций и обучающих программ. Естественно, сценарий должен быть записан во внешнем файле в виде списка имен мультимедийных файлов и некоторых текстов.

4.4. Процедуры воспроизведения звуков *Beep*, *MessageBeep* и *PlaySound*

- Простейшей процедурой, управляющей звуком, является процедура `Beep`. Она не имеет параметров и воспроизводит стандартный звуковой сигнал, установленный в Windows, если компьютер имеет звуковую карту и стандартный сигнал задан [92, 102]. (Он устанавливается с помощью пункта главного меню `Windows` **Панель управления — Звуки и аудиоустройства**.) Если звуковой карты нет или стандартный сигнал не установлен, звук воспроизводится через динамик компьютера в виде короткого щелчка. Создадим простейшее приложение, использующее только одну кнопку, в обработчике которой будет содержаться следующий код:

  ```
  Beep; .
  ```

  При запуске такого приложения будет слышен стандартный звук Windows или просто щелчок динамика, если стандартный звук не установлен.

- Процедура `MessageBeep` используется для воспроизведения звуков, сопровождающих те или иные события Windows.

  ```
  Function MessageBeep (uType: word): boolean;
  ```

  Параметр `uType` указывает воспроизводимый звук как идентификатор раздела `[sounds]` реестра, в котором записаны звуки, сопровождающие те или иные события Windows.

  С помощью панели управления (Control Panel) пользователь может удалить или установить соответствующие звуки. В табл. 4.9 приведены возможные значения параметра `uType` и соответствующие им системные события.

Таблица 4.9. *Возможные значения параметра uType*

Значение параметра *uType*	**Системное событие**
`MB_ICONASTERISK`	`SystemAsterisk` — звездочка
`MB_ICONEXCLAMATION`	`SystemExclamation` — восклицание
`MB_ICONHAND`	`SystemHand` — критическая ошибка
`MB_ICONQUESTION`	`SystemQuestion` — вопрос
`MB_OK`	`SystemDefauIt` — стандартный звук

После запроса звука процедура `MessageBeep` возвращает управление вызывающей функции и воспроизводит звук асинхронно (во время воспроизведения звука приложение продолжает выполняться). Если невозможно воспроизвести указанный в процедуре звук, делается попытка воспроизвести стандартный системный звук, установленный по умолчанию. Если и это невозможно, то воспроизводится стандартный сигнал через динамик. При успешном выполнении процедуры возвращается ненулевое значение. При аварийном завершении возвращается нуль.

В приложении введем новую кнопку с кодом обработчика:

```
MessageBeep(MB_OK);
```

Будет слышен тот же стандартный звук `Windows`, что и при выполнении процедуры `Beep`.

- Процедура `PlaySound` [92] предназначена для выполнения ряда операций со звуковыми файлами.

```
function  PlaySound (pszSound: PChar; hmod: HINST; wSound: Cardinal):
boolean;
```

Параметр `pszSound` определяет воспроизводимый звук. Он представляет собой переменную типа `Pchar` (строка с нулевым символом в конце), которая задает имя файла, псевдоним, имя ресурса, запись в `реестре` (в разделе [sounds] файла WIN.INI) или указатель на данные WAV, расположенные в памяти.

Параметр `hmod` — указатель на исполняемый файл, содержащий ресурс для загрузки. Он должен быть равен нулю, если параметр `fdwSound` не равен `snd_Resource`. Параметр используется только в случае, если звук берется из ресурса.

Параметр `fdwSound` — множество флагов, которые определяют режим воспроизведения и тип источника звука. В табл. 4.10 приведены некоторые из флагов, наиболее важные для воспроизведения произвольных волновых файлов.

Таблица 4.10. *Описание возможных значений параметра fwdSound*

Флаг	Описание
`SND_APPLICATION`	Звук воспроизводится приложением, ассоциированным с данным типом звуковой информации
`SND_ALIAS`	Используется имя системного события в регистре или в файле WIN.INI. Нельзя использовать его совместно с `snd_filename` или `snd_resource`, так как они взаимно исключают друг друга
`SND_FILENAME`	Имя файла
`SND_NOWAITE`	Если драйвер занят, немедленный выход без воспроизведения
`SND_PURGE`	Прекращается воспроизведение всех звуков. Если `pszSound` не равен нулю, все звуки, определенные текущим заданием, прекращаются. Необходимо указать правильный дескриптор для прекращения события `snd_resource`

Таблица 4.10 *(окончание)*

Флаг	Описание
SND_RESOURCE	Идентификатор ресурса. При использовании этого флага параметр hmod должен содержать дескриптор экземпляра, включающего определяемый ресурс
SND_ASYNC	Асинхронное воспроизведение звука; вызывает практически мгновенный возврат из функции. Позволяет достичь эффекта фонового звучания
SND_LOOP	Многократное воспроизведение звука в цикле — пока вы не остановите его самостоятельно. При использовании этого флага необходимо использовать флаг SND_ASYNC
SND_MEMORY	Воспроизводит WAV-файл из области памяти с указателем, определенным параметром pszsound
SND_NODEFAULT	Если звук не найден, PlaySound прекращает свое выполнение без воспроизведения звука, указанного по умолчанию в Registry
SND_NOSTOP	Воспроизводит звук только в том случае, если в данный момент никакой звук не воспроизводится. PlaySound возвращает значение True, если звук воспроизводится, и False — если не воспроизводится. Если этот флаг не определен, Win32 приостановит воспроизведение любых других звуков, чтобы воспроизвести звук, заданный параметром pszsound
SND_SYNC	Воспроизводит звук в синхронном режиме. Возврата из функции не происходит, пока не завершится воспроизведение звука

Рассмотрим примеры использования функции PlaySound.

```
PlaySound ('c:\WINDOWS\Media\tada.wav', 0, SND_ASYNC);
```

В данном примере будет воспроизведен звук из файла tada.WAV, расположенного в папке c:\WINDOWS\Media\ . Ключ SND_ASYNC задает асинхронный характер звучания (во время воспроизведения возможно выполнение других операций).

Звук может задаваться не только прямым указанием пути WAV-файла, но и указанием соответствующего системного события

```
PlaySound (' SystemExclamation', 0, SND_ASYNC);
```

или прямым заданием имени файла через стандартный диалог

```
PlaySound (PChar(OpenDialogl.Filename), 0, SND_ASYNC);
```

Комбинация флагов SND_ASYNC or SND_LOOP обеспечивает циклическое асинхронное воспроизведение звука, которое будет продолжаться до бесконечности, если в программе не предусмотрены программные средства прерывания, например

```
PlaySound(0,0,   SND_PURGE)
```

или

```
PlaySound(Nil,0,0)
```

При использовании синхронного режима воспроизведения

```
PlaySound ('c:\WINDOWS\Media\tada.wav', 0, SND_SYNC);
```

приложение будет блокировано до тех пор, пока файл не будет воспроизведен полностью. Комбинации флагов `SND_SYNC or SND_LOOP` или `SND_SYNC or SND_NOSTOP` являются недопустимыми, поскольку приводят к бесконечному воспроизведению звука и зацикливанию системы.

Несмотря не простоту, возможностей функций `Beep`, `MessageBeep` и `PlaySound` достаточно для реализации несложных проигрывателей звуковых файлов.

4.5. Интерфейс управления мультимедийными устройствами — MCI

Интерфейс управления мультимедийными устройствами — MCI (Media Control Interface [77, 102]) представлен набором команд, ориентированных на работы с различными мультимедийными устройствами. Взаимодействие приложения и мультимедийной подсистемы организуется посредством командных строк и (или) командных сообщений. Перечень основных устройств, поддерживаемых MCI, представлен в табл. 4.11.

Таблица 4.11. *Основные MCI-устройства*

Тип устройства	Описание
`cdaudio`	Лазерный проигрыватель
`sequencer`	Устройства вывода аудиоинформации в формате MIDI
`videodisc`	Проигрыватель видеодисков
`waveaudio`	Устройства вывода аудиоинформации в формате WAV
`avivideo`	Устройство вывода информации в формате AVI

Среди перечисленных устройств иногда выделяют две группы — простые и составные устройства. Первые не требуют для работы указания имени конкретного файла (`cdaudio`, `videodisc`), вторые являются устройствами с файловой структурой (`sequencer`, `avivideo`, `waveaudio`).

Общий порядок работы одинаков практически для всех типов устройств, поддерживаемых MCI, и состоит из следующих шагов:

1. Открыть устройство.
2. Выполнение функционала (запись, воспроизведение и т. д.).
3. Закрыть устройство.

Наряду с перечисленными основными командами MCI, имеется целый ряд дополнительных команд, которые будут рассмотрены далее по мере необходимости.

Перейдем к рассмотрению основных команд MCI.

Основные средства, которые реализуют создание приложений с использованием структур MCI, представлены всего лишь двумя командами: mciSendCommand и mciSendString.

- Команда mciSendCommand предназначена для передачи команд системе

```
MCIERROR mciSendCommand( MCIDEVICEID IDDevice, UINT uMsg, DWORD
fdwCommand, DWORD dwParam );
```

Рассмотрим назначение аргументов команды mciSendCommand:

IDDevice — идентификатор устройства MCI, которому посылается данная команда. Этот аргумент не используется при посылке команды MCI_OPEN;

uMsg — посылаемое сообщение;

fdwCommand — флаги сообщения;

dwParam — указатель на объект структуры, содержащий параметры передаваемого сообщения.

- Команда mciSendString также предназначена для передачи команд системе.

```
MCIERROR mciSendString( LPCTSTR lpszCommand, LPTSTR lpszReturnString,
UINT cchReturn, HANDLE hwndCallback );
```

Рассмотрим назначение аргументов команды mciSendString:

lpszCommand — указатель на строку, заканчивающуюся нулем, определяющую командную строку MCI;

lpszReturnString — указатель на буфер, в который будет записана возвращаемая информация. Если возвращение информации не предполагается, этот аргумент может иметь нулевое значение;

cchReturn — размер (в байтах) буфера, на который указывает аргумент lpszReturnString;

hwndCallback — дескриптор окна, которое будет получать извещения, если в командной строке установлен соответствующий флаг.

Эти два типа команд взаимосвязаны. При использовании командных строк в тексте программы операционная система преобразует их в командные сообщения и только после этого результат пересылается драйверу MCI для исполнения.

Указанные команды в случае успешного завершения возвращают нулевое значение, в противном случае младшее слово возвращаемого значения содержит код ошибки.

- Если требуется получить текстуальное описание возникшей ошибки, можно использовать функцию mciGetErrorString. Эта функция позволяет получить текстуальное описание по указанному коду ошибки MCI. Длина каждой строки, возвращаемой MCI, независимо от того, содержит она данные или описание ошибки, не может превышать 128 байт. В случае успешного завершения работы функция возвращает значение TRUE, если код ошибки не распознан — то значение FALSE.

```
BOOL mciGetErrorString( DWORD fdwError, LPTSTR lpszErrorText, UINT
cchErrorText );
```

Рассмотрим назначение аргументов команды `mciGetErrorString`:

`fdwError` — код ошибки, возвращенный функциями `mciSendCommand` или `mciSendString`;

`lpszErrorText` — указатель на буфер, в который будет записана заканчивающаяся нулем строка, содержащая текстуальное описание указанной ошибки;

`cchErrorText` — размер (в байтах) буфера, на который указывает аргумент `lpszErrorText`.

4.5.1. Проект "Консольное выполнение команд MCI"

В проекте "Консольное выполнение команд MCI" создается приложение, которое позволяет вводить из командной строки любые допустимые команды `MCI` и выполнять их. Проект приведен на компакт-диске (полное содержание которого приведено в *прил. 2*) в папке **Примеры | Глава 4 | Консольное выполнение команд MCI**. На рис 4.11 представлена форма этого проекта.

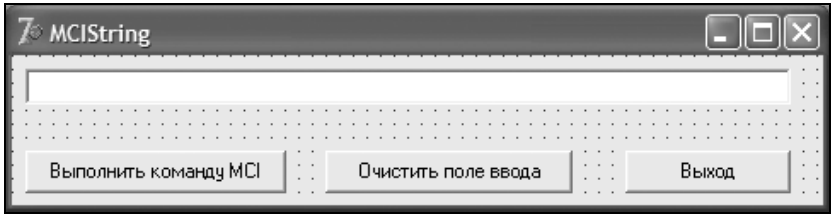


Рис. 4.11. Форма проекта "Консольное выполнение команд MCI"

На форме присутствует поле ввода, в котором пользователь может в текстовом виде вводить команды. При нажатии кнопки **Выполнить команду MCI** команда будет отработана и приложение будет готово к вводу и выполнению следующей команды. Вся функциональность программы сосредоточена в обработчике этой кнопки (листинг 4.32).

Листинг 4.32. Обработчик кнопки *Выполнить команду MCI*

```
procedure TForm1.WriteButtonClick(Sender: TObject);
var
  mcistring :String ;
  buf       :array[0..255]  of char ;
  res       :integer;
begin
  mcistring:=Edit1.Text; //Получаем введенную команду
```

```
  if mcistring='' then
    MessageBox (0,'Нет строки ввода,nil,MB_OK);
    //Посылаем команду драйверу MCI
  res:=mciSendString(PChar(mcistring),buf,256,0);
  if (res<>0) then begin
    mciGetErrorString (res,buf,256);
    MessageBox (0,buf,'Ошибка !',MB_OK or MB_ICONSTOP);
  end
  else
    if StrLen(buf)<>0 then
      MessageBox(0,buf,'Код возврата:',MB_OK);
end;
```

Убедившись, что в поле ввода не пустая строка, передаем ее в качестве первого аргумента функции `mciSendString`. Проверяем возвращенное функцией значение и, если оно отлично от нуля, выдаем сообщение об ошибке, иначе выдается результат работы команды. Рассмотрим функционирование созданного в проекте "Консольное выполнение команд MCI" приложения на нескольких примерах.

В первом примере рассмотрим воспроизведение звукового `WAV`-файла с помощью устройства `waveaudio`. В табл. 4.12 рассмотрены основные команды MCI для этого устройства.

Таблица 4.12. *Основные команды MCI для устройства waveaudio*

Команда MCI	**Назначение**	
`capability`	Запрашивает информацию о возможностях устройства управления цифровым звуком	
	Аргументы	**Комментарий**
	`can play`	Возвращает `TRUE`, если устройство может проигрывать
	`can record`	Возвращает `TRUE`, если устройство может записывать
	`can save`	Возвращает `TRUE`, если устройство может сохранить данные
`close`	Закрывает устройство управления цифровым звуком	
`delete`	Удаляет сегмент из WAV-файла	
	`from <position>`	Определяет позицию, с которой необходимо начать удаление данных
	`to <position>`	Определяет позицию, на которой необходимо остановить удаление данных
`info`	Выдает информацию об устройстве	
	`file`	Возвращает имя файла, которое использует устройство

Таблица 4.12 *(продолжение)*

Команда MCI	**Назначение**	
open	Открывает устройство	
pause	Приостанавливает воспроизведение файла	
play	Начинает проигрывание файла	
	Аргументы	**Комментарий**
	from <position>	Определяет позицию, с которой необходимо начать воспроизведение
	to <position>	Определяет позицию, на которой необходимо остановить воспроизведение
record	Начинает запись звука, вставляя новые данные в текущую позицию	
	insert	Определяет, что новые данные вставляются в файл
	from <position>	Определяет позицию, с которой необходимо начать запись
	to <position>	Определяет позицию, на которой необходимо остановить запись
close	Закрывает устройство	
delete	Удаляет фрагмент файла	
	from <position>	Позиция, с которой начинается удаление данных
	to <position>	Позиция, на которой прекращается удаление данных
save	Сохраняет звук в формате WAV	
	File	Имя файла (стандартное расширение WAV)
seek	Позиционирует текущий указатель воспроизведения	
	to <position>	Точное место позиционирования
set	Устанавливает параметры работы устройства	
	audio all on/ off	Разрешает/запрещает вывод аудиосигнала
	time format bytes/ milliseconds/ samples	Устанавливает формат времени в байтах/миллисекундах/сэмплах
	bltspersample <integer>	Устанавливает число битов на проигранную или записанную выборку
	audio left/right on/off	Разрешает/запрещает вывод аудиосигнала к левому/правому каналу
status	Возвращает информацию о состоянии устройства	
	channels	Возвращает число имеющихся в наличии каналов (1 для моно и 2 для стерео)

Таблица 4.12 *(окончание)*

Команда MCI	Назначение	
	Аргументы	**Комментарий**
	lenght	Возвращает полную длину WAV-файла
	level	Возвращает текущее значение уровня сигнала
	mode	Режим, в котором в данный момент находится устройство: NOT READY (не готов), PAUSED (временная остановка), PLAYING (проигрывание), RECORDING (запись) или STOPPED (остановлен)
	position	Возвращает текущую позицию
	time format	Возвращает текущий формат времени

В соответствии с приведенным ранее порядком работы, откроем устройство waveaudio. Команда открытия выглядит так:

```
open tada.wav type waveaudio alias z
```

В данной команде после ключевого слова open указывается полное имя файла — tada.WAV. Далее следует указание на то, что работа будет осуществляться с устройством waveaudio — type waveaudio. Далее следует alias z — необязательный параметр, устанавливающий псевдоним для файла (это удобно в случае длинных имен, чтобы не повторять их многократно при использовании в других командах).

После того как устройство открыто, мы можем:

1. Выполнить команду play z — воспроизвести открытый файл.
2. Выполнить команду close z — закрыть устройство.

Обратите внимание на выдаваемые после выполнения каждой команды коды (они должны быть отличны от нуля).

Во втором примере рассмотрим воспроизведение CD-диска с помощью устройства cdaudio без использования каких-либо дополнительных программных средств. Подчеркнем, что речь идет о диске в формате cdaudio, а не о диске с набором файлов mp3.

В табл. 4.13 рассмотрены основные команды MCI для устройства воспроизведения CD-дисков cdaudio. Приводятся только те команды, которые специфичны именно для этого устройства.

Таблица 4.13. *Основные команды MCI для устройства cdaudio*

Команда MCI	Назначение	
Capability	Возвращает информацию о возможностях устройства проигрывания компакт-дисков	
	Аргументы	**Комментарии**
	can eject	Возвращает TRUE, если устройство может выбрасывать диск

Таблица 4.13 *(окончание)*

Команда MCI	Назначение	
	Аргументы	**Комментарии**
	`can record`	Возвращает `TRUE`, если `cdaudio`-устройство не может записывать
`set`	Устанавливает параметры проигрывания дисков	
	`Door closed`	Выдвигает лоток привода
`status`	Выдает информацию об устройстве	
	`Current track`	Номер текущей дорожки
	`Length`	Длина диска в установленном формате
	`Number of tracks`	Возвращает число дорожек на диске

Воспроизведение осуществляется с использованием следующих команд MCI.

Команда `open cdaudio alias a` открывает устройство. При работе с устройством `cdaudio`, в отличие от `waveaudio`, в данной команде не надо указывать имя файла, поскольку `cdaudio` не является устройством с файловой структурой.

На рис. 4.12 приводится окно с кодом возврата, выводящееся в случае корректного выполнения команды, введенной в командной строке в проекте "Консольное выполнение команд MCI".

Возвращено значение, отличное от нуля, что подтверждает факт корректного выполнения команды.

Команда `set a time format tmsf` устанавливает временной формат `TMSF`. Время в формате `TMSF` записывается в двойное слово, младший байт которого хранит информацию о дорожках, следующий байт — о минутах, следующий за ним байт — о секундах. Старший байт двойного слова содержит информацию о кадрах.

Команда `status a number of tracks` выводит окно с информацией (рис. 4.13) о числе дорожек на диске (в нашем случае 23 дорожки).

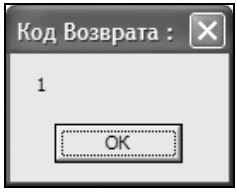


Рис. 4.12. Код возврата в проекте "Консольное выполнение команд MCI"

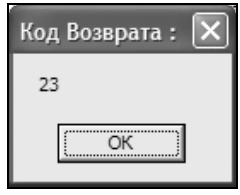


Рис. 4.13. Количество дорожек на диске

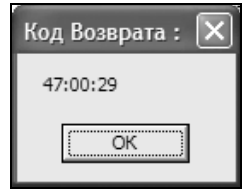


Рис. 4.14. Общая длительность звучания диска

Команда `status a length` выводит окно с информацией (рис. 4.14) об общей длительности звучания диска. Время задается в формате, который был задан предыдущей командой (в нашем случае `tmsf`).

Команда `seek a to 4` позиционирует головку устройства на дорожку с номером 4. Если следующей будет команда `play`, то начнется воспроизведение выбранной дорожки.

Команда `play a from 0:30:00 to 0:31:00 wait` воспроизводит музыкальный фрагмент в заданном временном промежутке.

Команда `stop a` останавливает воспроизведение.

Команда `set a door open` открывает приемный лоток привода.

Команда `set a door close` закрывает приемный лоток привода.

Команда `close a` закрывает устройство `cdaudio`.

Таким образом, приведенная последовательность команд реализует функциональность проигрывателя компакт-дисков.

4.5.2. Проект "Проигрыватель аудио-CD"

Рассмотрим теперь использование в программе функции `mciSendCommand`. В качестве проекта поставим задачу реализовать проигрыватель компакт-дисков (проект "Проигрыватель аудио-CD"). Проект приведен на компакт-диске в папке **Примеры | Глава 4 | Проигрыватель аудио CD**. На рис. 4.15 показан внешний вид формы этого проекта.

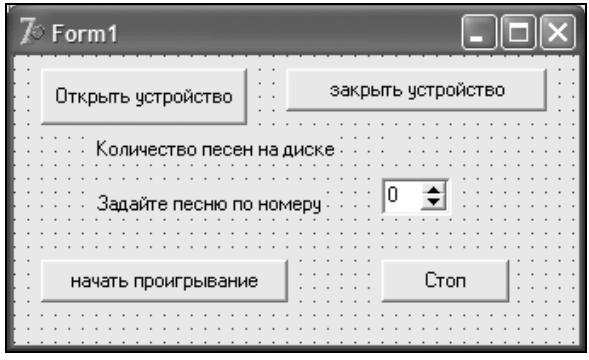


Рис. 4.15. Форма для проекта "Проигрыватель аудио-CD"

Опишем работу проигрывателя компакт-дисков, содаваемого в рамках данного проекта. Сначала необходимо, вставив диск в дисковод, открыть устройство (кнопка **Открыть устройство**). В листинге 4.33 приведен код обработчика этой кнопки. После открытия устройства на форме появляется информация о количестве имею-

щихся на диске дорожек (треков) (**Количество песен на диске**). Затем, выбрав одну из них по номеру (поле **Задайте песню по номеру**), ее можно воспроизвести (кнопка **Начать проигрывание**). В любой момент можно прекратить проигрывание (кнопка **Стоп**). Перед выходом из программы устройство надо закрыть (кнопка **Закрыть устройство**).

Листинг 4.33. Обработчик кнопки *Открыть устройство*

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  MciOpenParams.lpStrDeviceType:='cdaudio';
  res:=mciSendCommand(0,mci_open,mci_open_type,
    LongInt(@MciOpenParams));
  Id:=MciOpenParams.wDeviceID;
  MciSetParams.dwTimeFormat:=mci_Format_tmsf;
  res:=MciSendCommand(id,mci_set,mci_set_time_format,
    LongInt(@MciSetParams));
  MciStatusParams.dwItem:=mci_status_number_of_tracks;
  res:=MciSendCommand(id,mci_status,mci_status_item,
    LongInt(@MciStatusParams));
  MaxSong:=MciStatusParams.dwReturn;
  Label2.Caption:=IntToStr(MaxSong);
end;
```

Основная функциональность кода, приведенного в листинге 4.33, заключена в выполнении команды `MciSendCommand`. В отличие от проекта "Консольное выполнение команд MCI" (где была использована команда `mciSendString`), в данном проекте вместо команд (например, `play`) указываются соответствующие сообщения (`mci_play`). При использовании такого способа вместо аргументов требуется определить соответствующую команде структуру, задав значение ее полей.

В проекте использованы структуры: `MCI_OPEN_PARMS`, `MCI_SET_PARMS`, `MCI_STATUS_PARMS`, `MCI_PLAY_PARMS`, `MCI_SEEK_PARMS`. Рассмотрим описание этих структур.

- Структура `MCI_OPEN_PARMS`, используемая в команде открытия устройства, имеет следующий вид.

```
tMCI_OPEN_PARMS = record
    dwCallback: DWORD;
    wDeviceID: MCIDEVICEID;
    lpstrDeviceType: PAnsiChar;
    lpstrElementName: PAnsiChar;
    lpstrAlias: PAnsiChar;
  end;
```

  Рассмотрим значения отдельных полей структуры `MCI_OPEN_PARMS`.

dwCallback — младшее слово данной переменной содержит дескриптор окна, которому будет послано извещение.

wDeviceID — идентификатор, возвращаемый приложению.

lpstrDeviceType — имя или константа идентификатора типа устройства. (Имя устройства обычно берется из системного реестра или файла SYSTEM.INI). Для задания типа устройства могут использоваться следующие константы (табл. 4.14).

Таблица 4.14. *Допустимые типы устройств*

Значение константы	Устройство
MCI_ALL_DEVICE_ID	Любое устройство
MCI_DEVTYPE_ANIMATION	Устройство анимационного воспроизведения
MCI_DEVTYPE_CD_AUDIO	Лазерный проигрыватель
MCI_DEVTYPE_DAT	Цифровой аудиомагнитофон
MCI_DEVTYPE_DIGITAL_VIDEO	Устройство воспроизведения видеоинформации
MCI_DEVTYPE_OTHER	Тип устройства не определен
MCI_DEVTYPE_OVERLAY	Устройство наложения видеоинформации
MCI_DEVTYPE_SCANNER	Сканер
MCI_DEVTYPE_SEQUENCER	Устройства воспроизведения аудиоинформации в формате MIDI
MCI_DEVTYPE_VCR	Кассетный видеомагнитофон
MCI_DEVTYPE_VIDEODISC	Проигрыватель видеодисков
MCI_DEVTYPE_WAVEFORM_AUDIO	Устройства воспроизведения аудиоинформации в формате PCM

lpstrElementName — элемент устройства (обычно это путь к файлу).

lpstrAlias — необязательное альтернативное имя устройства.

Команда MCI_SET изменяет параметры устройств и производит их установку в различные режимы. Эту команду можно посылать лазерному проигрывателю, устройству воспроизведения и записи видеоинформации, устройству воспроизведения аудиоинформации в формате MIDI, видеомагнитофону, видеодиску, устройству наложения видеоинформации и устройству вывода аудиоинформации в формате ИКМ (Импульсно-кодовая модуляция).

- Структура MCI_SET_PARMS, используемая в команде установки параметров, имеет следующий вид.

```
tagMCI_SET_PARMS = record
  dwCallback: DWORD;
  dwTimeFormat: DWORD;
  dwAudio: DWORD;
end;
```

Значение отдельных полей структуры `MCI_SET_PARMS`:

`dwCallback` — младшее слово данной переменной содержит дескриптор окна, которому будет послано извещение;

`dwTimeFormat` — временной формат устройства;

`dwAudio` — канал вывода аудиоинформации.

- Структура `MCI_STATUS_PARMS`, используемая в команде получения статуса устройства, имеет следующий вид.

```
tMCI_STATUS_PARMS = record
    dwCallback: DWORD;
    dwReturn: DWORD;
    dwItem: DWORD;
    dwTrack: DWORD;
  end;
```

Рассмотрим значения отдельных полей структуры `MCI_STATUS_PARMS`:

`dwCallback` — младшее слово данной переменной содержит дескриптор окна, которому будет послано извещение;

`dwReturn` — буфер, в который будет записана запрашиваемая информация;

`dwItem` — определяет запрашиваемый параметр;

`dwTrack` — длина или число дорожек.

- Структура `MCI_PLAY_PARMS`, используемая в команде воспроизведения, имеет следующий вид.

```
tMCI_PLAY_PARMS = record
    dwCallback: DWORD;
    dwFrom: DWORD;
    dwTo: DWORD;
  end;
```

Рассмотрим значения отдельных полей структуры `MCI_PLAY_PARMS`:

`dwCallback` — младшее слово данной переменной содержит дескриптор окна, которому будет послано извещение;

`dwFrom` — начальная позиция воспроизведения;

`dwTo` — конечная позиция воспроизведения.

При работе с устройством вывода видеоинформации, переменной `lpSet` передается указатель на объект структуры `MCI_DGV_SET_PARMS`, при работе с устройством воспроизведения аудиоинформации в формате `MIDI` — указатель на объект структуры `MCI_SEQ_SET_PARMS`, а при работе с устройством вывода аудиоинформации в формате ИКМ — указатель на объект структуры `MCI_WAVE_SET_PARMS`.

Команда MCI_STATUS позволяет получить информацию об устройстве MCI. Эта команда может посылаться всем устройствам MCI. Запрашиваемая информация возвращается в переменной dwReturn объекта структуры, на который указывает аргумент lpStatus.

В листинге 4.34 представлен код обработчика кнопки **Начать проигрывание** проекта "Проигрыватель аудио-CD".

Листинг 4.34. Обработчик кнопки *Начать проигрывание*

```
procedure TForm1.Button2Click(Sender: TObject);
begin
  MciSeekParams.dwTo:=SpinEdit1.Value;
  res:=MciSendCommand(id,mci_seek,mci_to,LongInt(@MciSeekParams));
  res:= mciSendCommand(id,mci_play,0,longInt(@MciPlayParams));
end;
```

Рассмотрим подробнее функцию

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_SEEK, DWORD dwFlags,
(DWORD) (LPMCI_SEEK_PARMS) lpSeek); .
```

Рассмотрим аргументы этой функции.

wDeviceID — идентификатор устройства MCI, которому посылается данное командное сообщение.

MCI_SEEK — команда, изменяющая текущую позицию воспроизведения. В процессе выполнения этой команды прекращается вывод аудио- и видеоинформации. После завершения поиска устройство останавливается. Эту команду можно посылать лазерному проигрывателю, устройству воспроизведения и записи видеоинформации, устройству воспроизведения аудиоинформации в формате MIDI, видеомагнитофону, видеодиску и устройству воспроизведения аудиоинформации в формате PCM. Для сообщения MCI_SEEK определены следующие значения (табл. 4.15).

***Таблица 4.15.** Допустимые значения сообщения MCI_SEEK*

Значение сообщения	Комментарий
MCI_SEEK_TO_END	Поиск конца воспроизводимого фрагмента
MCI_SEEK_TO_START	Поиск начала воспроизводимого фрагмента
MCI_TO	В переменной dwTo объекта структуры MCI_SEEK_PARMS, на который указывает аргумент lpSeek, содержится новое значение текущей позиции воспроизведения. Размерность величины, содержащейся в данной переменной, определяется флагом MCI_SET_TIME_FORMAT сообщения MCI_SET. Этот флаг несовместим с флагами MCI_SEEK_TO_END или MCI_SEEK_TO_START

`dwFlags` — флаги сообщения `MCI_NOTIFY`, `MCI_WAIT` или `MCI_TEST` (для видеомагнитофона и устройства вывода видеоинформации).

`lpSeek` — указатель на объект структуры `MCI_SEEK_PARMS`. Устройства, имеющие расширенный набор команд, используют вместо этой структуры собственные структуры.

- Структура `MCI_SEEK_PARMS`, используемая в команде изменения текущей позиции воспроизведения, имеет следующий вид.

```
tMCI_SEEK_PARMS = record
    dwCallback: DWORD;
    dwTo: DWORD;
  end;
```

Значения отдельных полей структуры `MCI_SEEK_PARMS`:

`dwCallback` — младшее слово данной переменной содержит дескриптор окна, которому будет послано сообщение;

`dwTo` — новое значение текущей позиции воспроизведения.

Объект структуры `MCI_SEEK_PARMS` содержит информацию о новом значении текущей позиции воспроизведения, устанавливаемой сообщением `MCI_SEEK`. При присвоении значений переменным данного объекта структуры следует установить соответствующие флаги в аргументе `fdwCommand` функции `mciSendCommand` для того, чтобы установленные значения не игнорировались.

В листинге 4.35 представлен код обработчика кнопки **Стоп** проекта "Проигрыватель аудио-CD".

Листинг 4.35. Обработчик кнопки *Стоп*

```
procedure TForm1.Button3Click(Sender: TObject);
begin
res:=mciSendCommand(id,mci_stop,0,0);
end;
```

Команда `MCI_STOP` останавливает воспроизведение и запись на устройстве, очищает буфер воспроизведения и прекращает вывод видеоинформации. Эту команду можно посылать лазерному проигрывателю, устройству воспроизведения и записи видеоинформации, устройству воспроизведения аудиоинформации в формате `MIDI`, видеомагнитофону, видеодиску и устройству воспроизведения аудиоинформации в формате `waveaudio`.

Функция, осуществляющая команду остановки устройства, имеет следующий вид.

```
MCIERROR mciSendCommand(MCIDEVICEID wDeviceID, MCI_STOP, DWORD dwFlags,
(DWORD) (LPMCI_GENERIC_PARMS) lpStop);
```

Рассмотрим аргументы этой функции:

`wDeviceID` — идентификатор устройства MCI, которому посылается данное командное сообщение;

dwFlags — флаги сообщения MCI_NOTIFY, MCI_WAIT или MCI_TEST для видеомагнитофона и устройства вывода видеоинформации;

lpStop — указатель на объект структуры MCI_GENERIC_PARMS.

Различие между командами MCI_STOP и MCI_PAUSE зависит от особенностей устройства. Если есть такая возможность, команда MCI_PAUSE только приостанавливает операцию воспроизведения или записи и оставляет устройство готовым для немедленного возобновления этой операции.

Для лазерных проигрывателей команда MCI_STOP устанавливает текущую позицию воспроизведения на начало дорожки, а команда MCI_PAUSE сохраняет текущую позицию в пределах дорожки.

При возобновлении процесса воспроизведения данная команда не изменяет текущей позиции, установленной командами MCI_PLAY и MCI_RECORD.

4.5.3. Проект "Запись звука с использованием команд MCI"

Рассмотрим разработку еще одного проекта под названием "Запись звука с использованием команд MCI". Проект приведен на компакт-диске в папке **Примеры | Глава 4 | Запись звука с использованием команд MCI**. На рис. 4.16 показан внешний вид формы этого проекта.

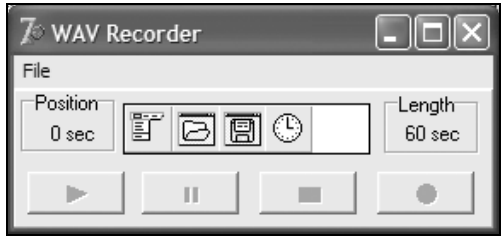


Рис. 4.16. Форма проекта "Запись звука с использованием команд MCI"

Разрабатываемое приложение должно осуществлять запись сигнала с микрофона в WAV-файл и поддерживать следующие функции: запись, воспроизведение, остановка записи и воспроизведения (стоп), приостановка записи и воспроизведения (пауза), сохранение записанного файла.

Запись и сохранение звука в виде WAV-файла с использованием интерфейса MCI реализуется такой последовательностью команд.

```
open new type waveaaudio alias z
record z
```

```
stop z
save z  test.wav
cloze z
```

Отличие этого примера от рассмотренных ранее состоит в том, что используемая в нем команда `open` содержит флаг `new` (поскольку файл к моменту открытия еще не существует). Команда `record` осуществляет процедуру записи (листинг 4.36). Команда `save` записывает на диск файл с заданным именем.

Листинг 4.36. Процедура записи звука в проекте "Запись звука с использованием команд MCI"

```
procedure Tfwav.bbtRecordClick(Sender: TObject);
begin
   bbtStop.Enabled := true;
   is_saved := false;
   mciSendString('record snd',Buf, 256, 0);
   tm := 0;
   Timer1.Enabled :=True ;
   bbtRecord.Enabled := false;
end;
```

4.6. Программирование мультимедийных приложений с использованием WinAPI

Как показано в предыдущих разделах, средства MCI позволяют решить целый ряд задач, стоящих перед разработчиком. Однако принципиальным недостатком этих средств, обойти который в рамках библиотеки MMSYSTEM.DLL невозможно, является работа с мультимедийным контентом только на уровне файлов. Вместе с тем, большая группа приложений требует именно работы с мультимедийным файлом на низком уровне. Примерами задач такого класса являются создание музыкального редактора, перекодировка файла из одного формата в другой, использование в приложении потокового мультимедиа.

Все перечисленные задачи могут быть решены с использованием мультимедийных функций низкоуровнего программирования [68, 77, 102, 116]. В состав библиотек Windows входят несколько групп функций, каждая из которых ориентирована на решение отдельной группы таких задач. По общим правилам именования функций WinAPI они начинаются с префиксов, которые и определяют область их применения. К ним относятся:

- `wave*` — группа функций для работы со звуковой информацией, представленной в формате WAV;

- `midi*` — группа функций для работы с музыкальной информацией, представленной в формате MID;
- `avi*` — группа функций для работы с видеоинформацией, представленной в формате AVI;
- `acm*` — группа функций для работы с кодеками и фильтрами;
- `aux*` — группа функций для работы с вспомогательными устройствами;
- `mmio*` — функции ввода/вывода для файлов мультимедийной структуры.

4.6.1. Структура RIFF-файла

Поскольку при разработке низкоуровневых приложений требуется знание структуры соответствующего мультимедийного файла [75] (`riff`, `wave`, `avi` и т. д.), подробнее рассмотрим такую структуру на примере звукового файла.

Файл формата WAV представляет собой одну из реализаций файлового формата RIFF (Resource Interchange File Format — формат файлов передачи ресурсов), который изначально был предложен фирмами Microsoft и Intel специально для хранения мультимедийных данных. Он позволяет хранить в едином файле не только содержательную часть информации (звук, музыку, видео), но и практически любую дополнительную информацию (формат, названия произведений и т. д.). Основным элементом RIFF-файла в соответствии со стандартом является `chunk` (чанк, сегмент). В листинге 4.37 представлена структура данных такого элемента.

Листинг 4.37. Структура данных сегмента файла RIFF

```
RIFF_struct = record
  ckID: DWORD;
  ckSize: DWORD;
  ckData[ckSize]: BYTE;
  End;
```

В первом поле структуры содержится идентификатор сегмента. Для однозначной идентификации сегментов стандартом предусмотрены как предопределенные названия, так и пользовательские. Второе поле — число, указывающее размер сегмента в байтах. Третье содержит раздел данных. Самое главное, что обеспечило распространение файлов такой структуры — это предусмотренная стандартом возможность содержать внутри сегментов вложенные сегменты той же структуры. Таким образом, RIFF-файл может рассматриваться как иерархическая структура данных произвольной сложности. Такая структура как раз и характерна для различных типов мультимедийных данных.

Формат WAVE файла имеет вид:

```
<WAVE-формат> = 'WAVE' + <fmt > + <data >,
```

где:

'WAVE' — набор символов, подтверждающий, чтот файл имеет данный формат;

<fmt> — сегмент с информацией о звуковом сигнале;

<data> — сегмент данных, содержащий сигнал.

В свою очередь,

```
<fmt> = 'fmt ' + <ckSize> + <WaveFormat> + <fmt-specific>,
```

где:

ckSize — размер сегмента;

WaveFormat — стандартная структура WinAPI WaveFormatEx;

fmt-specific — дополнительная информация о формате.

Структура WaveFormatEx очень существенна для всех приложений, работающих со звуком на низком уровне:

```
tWAVEFORMATEX = record
    wFormatTag: Word;
    nChannels: Word;
    nSamplesPerSec: DWORD;
    nAvgBytesPerSec: DWORD;
    nBlockAlign: Word;
    wBitsPerSample: Word;
    cbSize: Word;
  end;
```

Рассмотрим смысл отдельных полей структуры.

- wFormatTag — идентификатор используемого аудиоформата. Полный список идентификаторов аудиоформатов содержится в файле заголовка MMREG.H.
- nChannels — число используемых каналов вывода аудиоинформации. При монофоническом воспроизведении используется один канал, а при стереофоническом — два.
- nSamplesPerSec — частота дискретизации. Если в переменной wFormatTag содержится идентификатор WAVE_FORMAT_PCM, то стандартными значениями переменной nSamplesPerSec являются 8,0, 11,025, 22,05 и 44,1 кГц. Для форматов, не использующих импульсно-кодовую модуляцию, это значение вычисляется, исходя из спецификации разработчика соответствующего формата.
- nAvgBytesPerSec — скорость передачи данных, выраженная в байтах в секунду, используемая при воспроизведении информации в данном формате. Если в переменной wFormatTag содержится идентификатор WAVE_FORMAT_PCM, то значение переменной nAvgBytesPerSec равняется произведению значений переменных

nSamplesPerSec и nBlockAlign. Для форматов, не использующих импульсно-кодовую модуляцию, это значение вычисляется, исходя из спецификации разработчика соответствующего формата.

Значение переменной nAvgBytesPerSec используется программным обеспечением для определения размера буфера, использующегося для записи и воспроизведения сигналов.

- nBlockAlign — граница выравнивания блока, выраженная в байтах. В этой переменной хранится минимальный размер фрагмента данных, используемый в данном режиме компрессии. Если в переменной wFormatTag содержится идентификатор WAVE_FORMAT_PCM, то значение переменной nBlockAlign равняется произведению значений переменных nChannels и wBitsPerSample, деленному на 8 (количество бит в байте). Для форматов, не использующих импульсно-кодовую модуляцию, это значение вычисляется, исходя из спецификации разработчика соответствующего формата.

  Размер буфера для записи и воспроизведения сигналов должен быть кратным значению переменной nBlockAlign. Запись и воспроизведение информации должны всегда начинаться с начала блока. Недопустимо, например, начинать воспроизведение аудиоинформации, записанной в формате с импульсно-кодовой модуляцией, с середины отсчета (то есть не с границы выравнивания блока).

- wBitsPerSample — количество битов, используемых для кодирования одного отсчета данных в формате, определенном в переменной wFormatTag. Если в переменной wFormatTag содержится идентификатор WAVE_FORMAT_PCM, то переменная wBitsPerSample может принимать только два значения: 8 и 16. Для форматов, не использующих импульсно-кодовую модуляцию, это значение вычисляется, исходя из спецификации разработчика соответствующего формата. В некоторых схемах компрессии данных значение переменной не может быть корректно определено, в этих случаях в данную переменную записывается нулевое значение.

- cbSize — размер блока дополнительной информации, записываемой после объекта структуры WAVEFORMATEX, выраженный в байтах. Эта информация может использоваться форматами, не использующими импульсно-кодовую модуляцию, для хранения необходимой им информации. Если для заданного в переменной wFormatTag режима компрессии не требуется хранения дополнительной информации, в эту переменную заносится нулевое значение. В формате WAVE_FORMAT_PCM (и только в этом формате) значение данной переменной игнорируется.

Формат WAV предусматривает работу с многоканальным звуком, причем количество каналов стандартом не ограничено. Интрепретация отсчетов в разделе данных зависит при этом только от числа каналов, указанных в структуре.

Итог вышесказанному подведен в табл. 4.16, которая содержит описание структуры WAVE-файла.

Таблица 4.16. *Структура WAVE-файла*

Смещение от начала в байтах	Содержимое	Размер в байтах
Начало RIFF-чанка		
00	'RIFF'	4
04	DWORD — размер RIFF-чанка	4
Начало WAVE-формы		
08	'WAVE'	4
Начало fmt–чанка		
0C	'fmt '	4
10	DWORD — размер fmt-чанка (10h или 12h)	4
Структура WaveFormat (или WaveFormatEx)		
14	WORD wFormatTag = 1 (PCM)	2
16	WORD nChannels = 1	2
18	DWORD nSamplesPerSec = 11025	4
1C	DWORD nAvgBytesPerSec = 11025	4
20	WORD nBlockAlign = 1	2
22	WORD nBitsPerSample = 8	2
24	WORD cbSize (=0 или отсутствует для PCM)	2
Далее в круглых скобках приведены смещения для случая без cbSize		
Конец fmt-чанка		
Начало data-чанка		
26 (24)	'data'	4
2A (28)	DWORD размер data-чанка	4
2E (2C)	Sample0,Sample1,Sample2,...	
Конец WAVE-формы		
Конец RIFF-чанка		

Исходя из структуры файла, можно предложить следующий алгоритм работы с ним на низком уровне:

1. Проверяем сигнатуру 'RIFF' по смещению 0.
2. Проверяем сигнатуру 'WAVE' по смещению 8.
3. Проверяем wFormatTag=1 по смещению 14.

4. Читаем `nChannels`, `nBitsPerSample` по смещениям 16 и 22.
5. Если надо, читаем `nSamplesPerSec` по смещению 18.
6. Начиная со смещения 24, начинаем искать data-чанк.

4.6.2. Проект "Низкоуровневое чтение файла"

Проект осуществляет низкоуровневое чтение файла, при котором информация считывается из файла побитно и интерпретируется в соответствии с его структурой. Проект приведен на компакт-диске в папке **Примеры | Глава 4 | Низкоуровневое чтение файла**. Форма проекта приведена на рис. 4.17. В верхнем поле отображается название выбранного пользователем файла, в левом нижнем — структура файла, в правом нижнем — содержание выбранного элемента структуры.

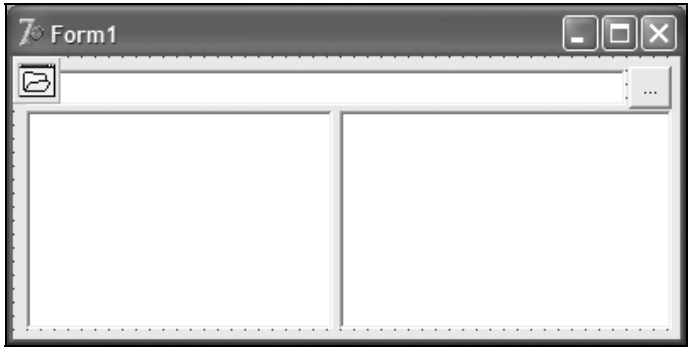


Рис. 4.17. Форма проекта "Низкоуровневое чтение файла"

В проекте "Низкоуровневое чтение файла" используются функции мультимедийного ввода-вывода `mmio`. Для открытия файла используется функция `mmioOpen`, выполняющая операцию открытия мультимедийного файла со структурой `RIFF`. Функция вовращает в программу дескриптор типа `HMMIO`, используемый далее другими функциями. Введем в разделе описаний следующий набор переменных.

```
mmio_hd     : HMMIO;      // дескриптор открытого ММ-файла
mmio_parent : TMMCKInfo;  //описатель раздела родительского чанка
mmio_subch  : TMMCKInfo;  // описатель раздела дочернего чанка
```

После того как имя файла будет указано в стандартном методе `OpenDialog`, осуществляется открытие файла:

```
mmio_hd := mmioOpen(PChar(edtFile.Text), nil, MMIO_READ);
```

При благополучном завершении этой операции (дескриптор `mmio_hd` должен быть отличен от нуля), пытаемся установить, с каким типом файла мы имеем дело. Как показано ранее, файл формата WAVE должен содержать в заголовке соответствующую сигнатуру.

Считываем ее:

```
mmio_parent.fccType := mmioStringToFOURCC('WAVE', MMIO_TOUPPER),
```

используя функцию `mmioStringToFOURCC`, которая возвращает четырехбуквенный код, созданный из переданной строки. При этом используется специальный тип поля `FOURCC` (`Four-Character Code`). Такой код продставляет собой строку длиной не более четырех символов (только цифры и буквы), содержащую текстовое название объекта (в нашем случае — `'WAVE'`). В случае если считанная сигнатура отличается от требуемой, очевидно, что открытый файл не принадлежит к файлам типа `'WAVE'`.

Далее, используя функцию `mmioDescend`, переходим к следующему по иерархии чанку (вложенному), который должен иметь значение `'fmt'`. Если и это условие выполнено, то можно начинать считывание информации (листинг 4.38) и отображение ее на форме.

Листинг 4.38. Процедура считывания содержания WAVE-файла на низком уровне с использованием функции *mmio**

```
procedure TForm1.btnOpenClick(Sender: TObject);
  mmio_hd     : HMMIO;
  mmio_parent : TMMCKInfo;
  mmio_subch  : TMMCKInfo;
  Fmt: TCommWaveFmtHeader;
  node, node_sub : TTreeNode;
  pinf : PRInfo;
  SamplesNumber, SampleRate : LongInt;
  Channels, BitsPerSample   : Word;
begin
  if OpenDialog.Execute then
    begin
      trwMain.Items.Clear;
      mem.Clear;
      edtFile.Text := OpenDialog.FileName;
      mmio_hd := mmioOpen(PChar(edtFile.Text), nil, MMIO_READ);
      if (mmio_hd = 0) then ShowMessage('Файл не удалось открыть')
      else
        begin
          mmio_parent.fccType := mmioStringToFOURCC('WAVE',
                                                MMIO_TOUPPER);
          if (mmioDescend(mmio_hd, PMMCKINFO(@mmio_parent), nil,
             MMIO_FINDRIFF) <> 0)
then ShowMessage('Это не WAVE файл.')
          else
            begin
              node      := trwMain.Items.AddFirst(nil, 'WAVE');
```

```
            new(pinf);  // указатель на строки
            node.Data := pinf;
            pinf.Lines := TStringList.Create;
            pinf.Lines.Add('Это WAVE форма.');
            mmio_subch.ckid := mmioStringToFOURCC('fmt ', 0);
            if (mmioDescend(mmio_hd, @mmio_subch, @mmio_parent,
                MMIO_FINDCHUNK) <> 0)
then ShowMessage('Не могу найти подраздел fmt.')
            else
              begin
                node_sub  := trwMain.Items.AddChild(node,'fmt');
                new(pinf);
                node_sub.Data := pinf;
                pinf.Lines := TStringList.Create;
                mmioRead(mmio_hd, PChar(@fmt),
                        Longint(SizeOf(TCommWaveFmtHeader)));
                pinf.Lines.Add('Идентификатор формата : ' + Int
                        ToStr(fmt.wFormatTag));
                Channels := fmt.nChannels;
                pinf.Lines.Add('Количество каналов : ' + In
                         ToStr(Channels));
                SampleRate := fmt.nSamplesPerSec;
                pinf.Lines.Add('Отсчетов в секунду : ' + Int
                         ToStr(SampleRate));
                BitsPerSample := fmt.nBitsPerSample;
                pinf.Lines.Add('Бит в отсчете : ' + Int
                              ToStr(BitsPerSample));
                mmioAscend(mmio_hd, @mmio_subch, 0);
              end;
            mmio_subch.ckid := mmioStringToFOURCC('data', 0);
            if (mmioDescend(mmio_hd, @mmio_subch, @mmio_parent,
                 MMIO_FINDCHUNK) <> 0)
            then ShowMessage('Не могу найти подраздел data.')
            else
              begin
                node_sub:= trwMain.Items.AddChild(node, 'data');
                new(pinf);
                node_sub.Data := pinf;
                pinf.Lines := TStringList.Create;
                SamplesNumber := (mmio_subch.cksize * 8) div
                                  BitsPerSample div Channels;
                pinf.Lines.Add('Размер (байты) : ' +
                        IntToStr(mmio_subch.cksize));
                pinf.Lines.Add('Количество отсчетов : ' +
                        InToStr(SamplesNumber));
```

```
            pinf.Lines.Add('Длительность (секунды) : ' +
                    IntToStr(SamplesNumber div SampleRate));

            mmioAscend(mmio_hd, @mmio_subch, 0);
          end;

      end;

    mmioClose(mmio_hd, 0);
  end;
  end;
  end;
```

4.6.3. Проект "Низкоуровневое воспроизведение файла"

Проект приведен на компакт-диске в папке **Примеры | Глава 4 | Низкоуровневое воспроизведение файла**. В этом проекте продемонстрировано, как можно реализовать низкоуровневое воспроизведение звукового файла, когда информация считывается побитно и интерпретируется в соответствии с описанной структурой файла. На форме проекта (рис. 4.18) присутствуют: кнопка вызова стандартного диалогового окна **Открытие файла**, стандартные кнопки управления мультимедийным устройством (**Play**, **Pause**, **Stop**), окно, где будет отображаться осциллограмма воспризводимого звука, регуляторы громкости и таймер.

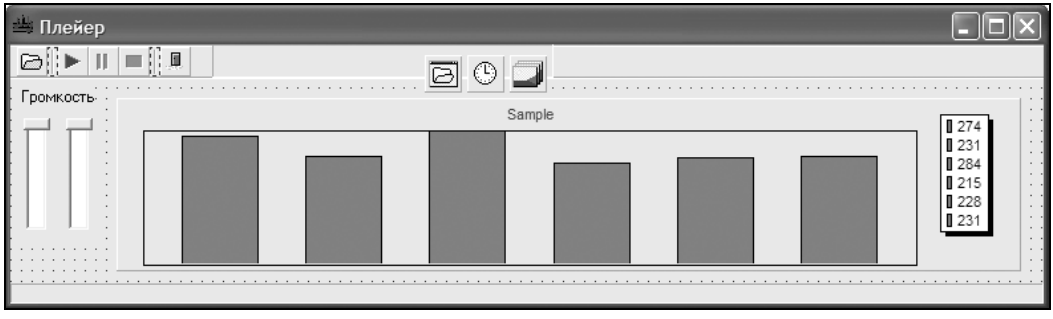


Рис. 4.18. Форма для проекта "Низкоуровневое воспроизведение файла"

Прежде чем приступить к выполнению проекта, необходимо получить информацию о возможностях звукового устройства. Для этого используется функция `waveOutGetDevCaps`.

```
function waveOutGetDevCaps(
 uDeviceID: UINT;     //Указатель на устройство воспроизведения
 lpCaps: PWaveOutCaps;//Структура, куда будут записаны данные
```

```
    uSize: UINT          //Размер структуры
  ): MMRESULT; stdcall;
```

Структура `WAVEOUTCAPS` имеет следующий вид.

```
  WAVEOUTCAPS = record
    wMid: Word;                 //Идентификатор производителя
    wPid: Word;                 //Идентификатор устройства
    vDriverVersion: MMVERSION;  //Версия драйвера
    szPname: array[0..MAXPNAMELEN-1] of AnsiChar; //Название устройства
    dwFormats: DWORD;           //Поддерживаемые форматы
    wChannels: Word;            //Количество каналов
    dwSupport: DWORD;           //Свойства устройства воспроизведения
  end;
```

Наиболее интересен параметр `dwSupport`, содержащий полную информацию о возможностях устройства воспроизведения, которые могут быть использованы в приложении.

Определены следующие значения этого параметра:

- `WAVECAPS_LRVOLUME` — допускается изменение громкости правого и левого каналов;
- `WAVECAPS_PITCH` — допускается изменение частоты воспроизводимого звука;
- `WAVECAPS_PLAYBACKRATE` — допускается изменение темпа воспроизведения звукового файла (`playback rate`), при этом не меняется частота дискретизации, меняется только время воспроизведения;
- `WAVECAPS_SYNC` — используется сихронный драйвер, который будет блокировать работу, пока воспроизводится звук;
- `WAVECAPS_VOLUME` — допустима регулировка громкости;
- `WAVECAPS_SAMPLEACCURATE` — возвращает позицию текущего указателя.

Параметр `wChannels` может принимать значение 1 (для монофонического звукового устройства) и 2 (для стереофонического звукового устройства).

Параметр `dwFormats` может принимать значения, приведенные в табл. 4.17.

Таблица 4.17. *Допустимые значения параметра dwFormats*

Значение параметра	Параметры звукового сигнала
`WAVE_FORMAT_1M08`	11,025 kHz, mono, 8 bit
`WAVE_FORMAT_1M16`	11,025 kHz, mono, 16 bit
`WAVE_FORMAT_1S08`	11,025 kHz, stereo, 8 bit
`WAVE_FORMAT_1S16`	11,025 kHz, stereo, 16 bit
`WAVE_FORMAT_2M08`	22,05 kHz, mono, 8 bit
`WAVE_FORMAT_2M16`	22,05 kHz, mono, 16 bit
`WAVE_FORMAT_2S08`	22,05 kHz, stereo, 8 bit

Таблица 4.17 *(окончание)*

Значение параметра	Параметры звукового сигнала
WAVE_FORMAT_2S16	22,05 kHz, stereo, 16 bit
WAVE_FORMAT_4M08	44,1 kHz, mono, 8 bit
WAVE_FORMAT_4M16	44,1 kHz, mono, 16 bit
WAVE_FORMAT_4S08	44,1 kHz, stereo, 8 bit
WAVE_FORMAT_4S16	44,1 kHz, stereo, 16 bit

После получения информации о возможностях устройства воспроизведения, открываем устройство с помощью функции waveOutOpen.

```
waveOutOpen
  function waveOutOpen(
   lphWaveOut: PHWaveOut;
   uDeviceID: UINT;
   lpFormat: PWaveFormatEx;
   dwCallback, dwInstance,
   dwFlags: DWORD
  ): MMRESULT;
```

Рассмотрим аргументы этой функции:

- lphWaveOut — адрес, по которому будет записан указатель на устройство воспроизведения;
- uDeviceID — идентификатор открываемого устройства. Значение WAVE_MAPPER соответствует выбору устройства по умолчанию, т. е. ситуации, когда функция сама выбирает устройство для записи аудиоинформации;
- lpFormat — указатель на структуру типа WAVEFORMATEX;
- dwCallback — указатель на функцию-семафор;
- dwFlags — параметры открываемого устройства.

  Допустимы следующие значения параметра dwFlags:

  CALLBACK_EVENT — в указателе dwCallback содержится событие THandle, через которое будет происходить информирование о ходе воспроизведения;

  CALLBACK_THREAD — в указателе dwCallback находится идентификатор потока;

  CALLBACK_FUNCTION — в dwCallback находится указатель на функцию;

  CALLBACK_WINDOW — в dwCallback находится указатель на окно, которому будут посылаться сообщения;

  CALLBACK_NULL — в dwCallback отсутствует содержимое;

  WAVE_ALLOWSYNC — можно открыть устройство в синхронном режиме;

WAVE_FORMAT_DIRECT — запрещается преобразование данных с помощью ACM-драйвера;

WAVE_FORMAT_QUERY — проверяется возможность открытия устройства с заданными параметрами. Если нет ошибок, то возвращает значение MMSYSERR_NOERROR. Если заданные параметры недопустимы, вернется код ошибки. В любом случае реального открытия устройства не произойдет. Этот флаг можно использовать как тест на допустимость настроек.

Функция waveOutOpen может вернуть следующие значения:

- MMSYSERR_ALLOCATED — устройство уже открыто;
- MMSYSERR_BADDEVICEID — указан неправильный идентификатор устройства uDeviceID;
- MMSYSERR_NODRIVER — драйвер отсутствует;
- MMSYSERR_NOMEM — не могу выделить память, или она заблокирована;
- WAVERR_BADFORMAT — указан неправильный формат;
- WAVERR_SYNC — устройство вывода синхронно, но оно вызвано без флага WAVE_ALLOWSYNC.

Теперь можно приступать непосредственно к процедуре воспроизведения звука. Фрагмент, который необходимо воспроизвести, разбивается на отдельные блоки, из которых формируется очередь воспроизведения. Каждый такой блок имеет заголовок, в котором содержится информация, необходимая для воспроизведения. Необходимо заполнить все поля заголовка и вызвать функцию waveOutPrepareHeader , которая подготовит заголовок к работе.

Функция waveOutPrepareHeader имеет следующий вид.

```
function waveOutPrepareHeader(
  hWaveOut: HWAVEOUT;
  lpWaveOutHdr: PWaveHdr;
  uSize: UINT
 ): MMRESULT; stdcall;
```

Определены следующие аргументы этой функции:

- hWaveOut — идентификатор устройства воспроизведения. Он уже получен после вызова функции waveOutOpen;
- lpWaveOutHdr — указатель на структуру wavehdr_tag;
- uSize — размер структуры wavehdr_tag.

Структура заголовка имеет следующий вид.

```
wavehdr_tag = record
  lpData: PChar;
  dwBufferLength: DWORD;
  dwBytesRecorded: DWORD;
  dwUser: DWORD;
```

```
    dwFlags: DWORD;
    dwLoops: DWORD;
    lpNext: PWaveHdr;
    reserved: DWORD;
  end;
```

Значения полей заголовка:

`lpData` — указатель на звуковые данные;

`dwBufferLength` — размер буфера в байтах;

`dwBytesRecorded` — размер записанной в буфер информации в байтах;

`dwFlags` — описание заголовка.

Для флагов (`dwFlags`), содержащих информацию о буфере, допустимы следующие значения:

`WHDR_BEGINLOOP` — звуковые данные являются первыми в цикле;

`WHDR_ENDLOOP` — звуковые данные являются последними в цикле;

`WHDR_DONE` — окончание воспроизведения. Этот флаг может задать только драйвер;

`WHDR_QUEUE` — данные помещены в очередь. Этот флаг может задать только драйвер;

`WHDR_PREPARED` — заголовок инициализирован. Этот флаг может задать только драйвер;

`dwLoops` — количество воспроизведений звуковых данных.

Перед вызовом функции `waveOutPrepareHeader` надо заполнить структуру `wavehdr_tag`. После окончания работы с этим заголовком, нужно освободить его с помощью функции `waveOutUnprepareHeader`.

Воспроизведение может осуществляться по следующей схеме. Выделяем два участка памяти (в нашем примере размер каждого из участков равен 64 Кбайт). Для каждого участка готовим соответствующие заголовки. Считываем данные из звукового файла в первый блок и направлем их в очередь воспроизведения. Подготовленный заголовок отправляет на воспроизведение функция `waveOutWrite`.

```
function waveOutWrite(
 hWaveOut: HWAVEOUT;
 lpWaveOutHdr: PWaveHdr;
 uSize: UINT
): MMRESULT; stdcall;
```

Определены следующие аргументы этой функции:

- `hWaveOut` — идентификатор устройства воспроизведения. Определяется после вызова функции `waveOutOpen`;

- `lpWaveOutHdr` — указатель на структуру, которая подготовлена с помощью `waveOutPrepareHeader`;
- `uSize` — размер структуры `wavehdr`.

Одновременно считываются данные во второй блок и отправляются в очередь воспроизведения, не дожидаясь, пока закончится воспроизведение первого блока. Как только закончится воспроизведение первого блока, начнется воспроизведение второго блока. В это же время начинается заполнение информацией освобожденного первого блока, после чего он вновь отправляется в очередь воспроизведения и воспроизводится по окончании воспроизведения второго блока. Такая методика воспроизведения звукового файла называется двойной буферизацией. Она требует задействования минимальных ресурсов системы и позволяет воспроизводить в непрерывном режиме звуковые файлы произвольной длины.

ГЛАВА 5

Компоненты диаграмм библиотеки *TeeChart*

5.1. Деловая графика

Очень часто в статистических документах, подготовленных на компьютере, используется деловая графика — диаграммы. В Delphi включена библиотека TeeChart, позволяющая создавать на экране и на принтере различные диаграммы. Она написана Дэвидом Бернеда и включает несколько компилируемых модулей компонентов диаграмм и несколько стандартных типов `Series`. Рисунок 5.1, на котором представлено диалоговое окно **TeeChart Gallery**, дает представление о разнообразии возможных видов диаграмм.

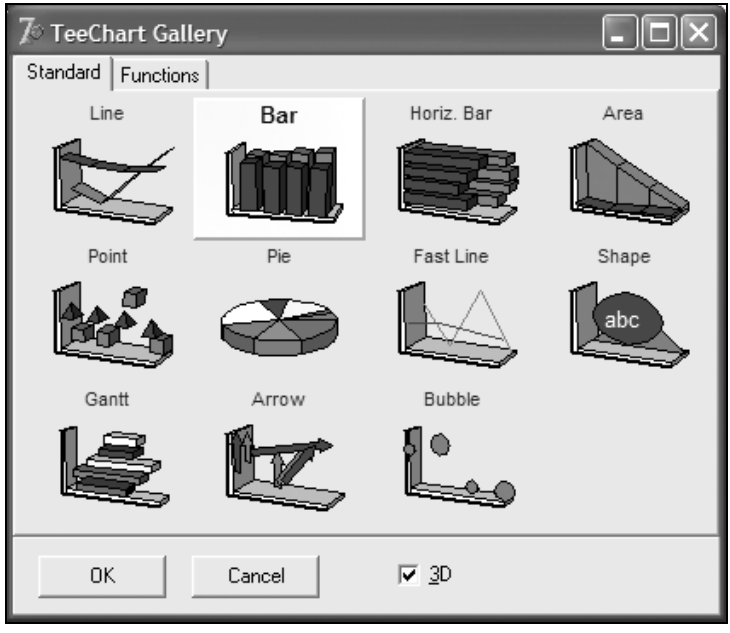


Рис. 5.1. Стандартные типы серий диаграмм

Имеется три компонента TeeChart на линейках Additional, DataControls и QReport палитры компонентов. Это компоненты TChart, TDBChart и TQRChart. Все они являются потомками класса TCustomChart. Основным компонентом, задающим массив данных для всех диаграмм, является компонент TChartSeries (серии).

Основными компонентами для построения всех диаграмм на экране являются TChart и TDBChart. При задании данных серии в тексте программы или вручную используют компонент TChart. Компонент TDBChart используется, если данные для диаграммы берутся из таблиц баз данных. Оба компонента происходят от класса TCustomChart, который является ответственным за большинство свойств диаграмм. Компонент TQRChart — потомок компонента TDBChart и предназначен для использования с генератором отчетов QuickReport. Компоненты TChart, TDBChart и TQRChart отвечают за внешний вид диаграмм, а их данные определяются сериями. На одной диаграмме может быть несколько серий и для каждого типа серии определены свои методы добавления и удаления точек.

TChart — наиболее важный компонент в библиотеке TeeChart. Этот компонент происходит от компонента Tpanel, наследует все его функциональные возможности и добавляет много новых. Компоненты диаграммы можно создавать как во время проектирования, так и во время выполнения.

Библиотека TeeChart включает наиболее характерные типы серий диаграмм. Такие типы серий диаграмм и их основные свойства представлены в табл. 5.1. Все серии Series порождены классом TChartSeries. Каждый тип серии имеет собственные свойства и обработчики событий для настройки диаграмм. Можно получать новые подкомпоненты из любого типа серии, создавая любые новые типы диаграммы.

Таблица 5.1. *Основные свойства типов серий диаграмм*

Тип серии	Название	Класс	Основные свойства
Line	Линия	TLineSeries	XValues, YValues, Xlabel
Fast Line	Быстрая линия	TFastLineSeries	XValues, YValues, Xlabel
Bar	Столбцы, пирамиды, цилиндры	TBarSeries	XValues, YValues (определяют Bar), Xlabel
HorizBar	Горизонтальные столбцы	ThorizBarSeries	Xvalues, YValues (определяют Bar), Xlabel, AddBar or AddXY
Area	Области	TAreaSeries	XValues, YValues, Xlabel
Point	Точки	TPointSeries	Xvalues, YValues, Xlabel
Pie	Секторы эллипса ("пирог")	TPieSeries	PieValues, Xlabel
Arrow	Стрелки	TArrowSeries	StartXValues, StartYValues, XLabel, EndXValues, EndYValues

Таблица 5.1 *(окончание)*

Тип серии	Название	Класс	Основные свойства
Bubble	Эллипсы ("пузыри")	TBubbleSeries	XValues, YValues, XLabel, RadiusValues
Gantt	Каждая точка отображается в виде горизонтального столбика	TGanttSeries	StartValues, EndValues, AY (Y axis level), AXLabel (Этикетка, дополнительно показанная на Y-оси или как маркер)
Shape	Фигуры	TShapeSeries	X0 (Top), Y0 (Bottom), X1 (Left), Y1 (Right)

Модуль TeeFunci содержит декларации для компонентов диаграммы и для типов, связанных с ними. В табл. 5.2 представлены классы функций, объявленные в этом модуле.

Таблица 5.2. *Классы функций модуля TeeFunci*

Класс функции	Назначение
TAddTeeFunction	Сложение серий
TAverageTeeFunction	Вычисление среднего
TSubractTeeFunction	Вычитание серий
TMultiplyTeeFunction	Умножение серий
TDivideTeeFunction	Деление серий
THighTeeFunction	Определение максимума
TLowTeeFunction	Определение минимума

5.2. Подготовка к работе

Прежде чем перейти к рассмотрению свойств компонентов диаграмм, рассмотрим их устройство (рис. 5.2) и разберемся с терминологией.

Компонент TChart — платформа для компонентов TChartSeries. Компонент TChart получен из компонента TPanel и наследует все его функциональные возможности.

Вся графическая информация располагается на панели (Panel). Сами диаграммы строятся в двумерной или трехмерной ортогональной системе координат (Axis).

На диаграмме может быть представлено одновременно несколько серий данных (Series). На компонент выносятся заголовки двух типов: верхний заголовок (Titles) и нижний заголовок (Foot). Кроме этого на компонент выносятся условные

обозначения (Legend). Данные могут снабжаться ярлыками (Marks). Диаграмма может выводиться с левой, нижней и задней плоскостями — стенками (Walls).

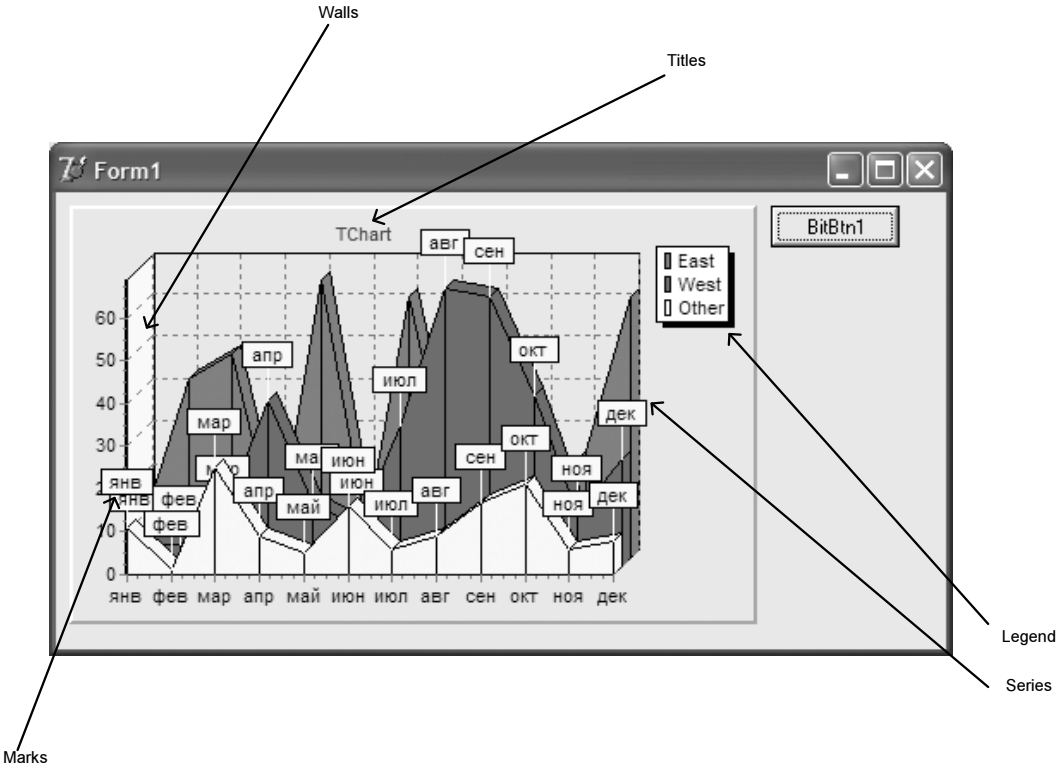


Рис. 5.2. Строение диаграммы

Данные могут отображаться в различных системах координат (Axis). В модуле TeEngine описан подкомпонент диаграммы — класс осей TchartAxis. Компоненты диаграммы обладают пятью типами осей TChartAxis: LeftAxis, RightAxis, TopAxis, BottomAxis и DepthAxis.

Верхний заголовок (Titles) и нижний заголовок (Foot) находятся соответственно в верхней и нижней областях диаграммы. Свойство Text определяет текст заголовков и параметры форматирования, с которыми должен выводиться этот текст. Свойство Text используется для набора текста. Установите Visible=true и измените свойства Font, Frame и Brush. Свойство Alignment можно использовать для выравнивания абзацев текста.

Условные обозначения (Legend) описаны в подкомпоненте TChartLegend модуля TeEngine. Подкомпонент рисует прямоугольник, заполненный названиями серий диаграммы или значениями серий. Можно переключать параметры условных обозначений, изменяя свойство LegendStyle. Свойство Alignment (выравнивание) опре-

деляет положение Legend в области диаграммы. Условные обозначения могут быть помещены в верхнюю, левую, правую или нижнюю область диаграммы. Левые и правые выравнивания определяют вертикальный вариант Legend с одной колонкой пунктов. Верхнее и нижнее выравнивание определяют горизонтальный вариант Legend с одной серией. Число показанных в условных обозначениях точек автоматически уменьшается в соответствии с доступным местом.

Свойство Legend.ResizeChart изменяется, если изменяются размеры или местоположение Legend. Обработчик события Legend.OnGetLegendRect обеспечивает механизм изменения размеров и местоположения Legend. Обработчик события Legend.OnGetLegendPos может использоваться для определения координат X и Y Legend.

Свойства Legend.HorizMargin и VertMargin управляют размерами полей между Legend и левым и правым краями диаграммы. Свойство Legend.TopPos может использоваться при левом или правом выравнивании Legend для того, чтобы управлять расстоянием по вертикали между Legend и верхним краем диаграммы.

Описанные методы позволяют обеспечить почти полный контроль над параметрами условных обозначений Legend.

Каждая серия диаграмм обладает свойством Marks (ярлыки или маркеры) типа TSeriesMarks. Свойства Marks аналогичны подсказкам (Hint) в Delphi и определены для каждой точки серии. Ярлыки Marks могут быть видимыми или не видимыми и обладают многими свойствами.

Компонент TChartWall управляет отображением стенок (Walls). Для этого используются свойства TChart.LeftWall и TChart.BottomWall.

Свойство Size определяет размер стенок. Свойства Color и Brush используется для заполнения стенок, а свойство Pen управляет видом сетки, рисуемой на стенках. Установка значения свойства Color равным clTeeColor означает, что стенки координатных плоскостей не будут заполнены и будут выглядеть прозрачными.

Основным инструментом при работе с диаграммами является **Редактор Диаграммы** (рис. 5.3). Если поместить на форму компонент TeeChart, то щелчком по нему правой кнопки мыши вызывается одноименное контекстное меню, в котором пользователь может выбрать пункт **Редактор Диаграммы**. Двойным щелчком левой кнопки мыши на компоненте, **Редактор Диаграммы** вызывается непосредственно. Окно редактора диаграмм состоит из двух вкладок — **Chart** и **Series**, которые предоставляют возможность выбора конфигурации диаграммы и доступ в галерею TeeChart, содержащую все типы серий данных. Заголовок окна по умолчанию принимает значение **Editing Charts 1**, которое впоследствии может быть изменено пользователем. Типы серий данных — подкомпоненты, к которым можно обращаться независимо от диаграммы.

В свою очередь, вкладка **Chart** окна редактора диаграмм содержит 9 вложенных вкладок — **Series**, **General**, **Axis**, **Titles**, **Legend**, **Panel**, **Paging**, **Walls**, **3D**.

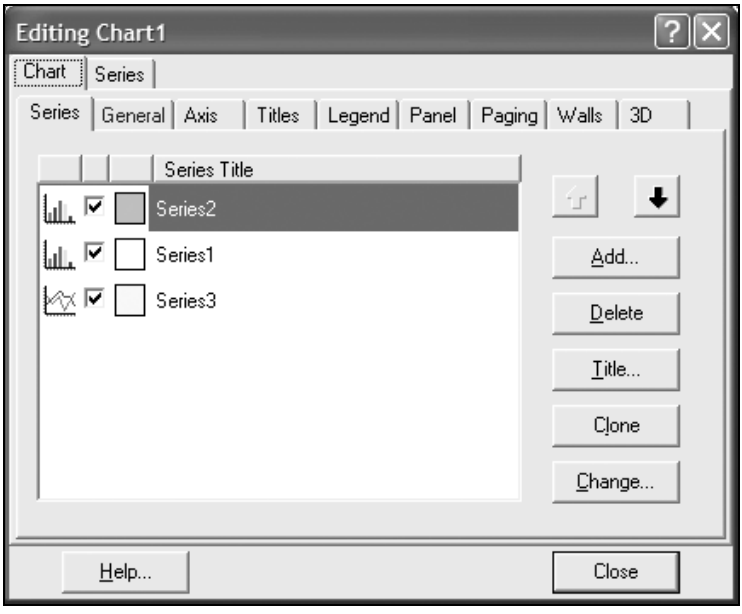


Рис. 5.3. Редактор диаграммы

Рассмотрим их назначение.

- На вкладке **Series** задается число и тип диаграмм на компоненте `Chart`.
- На вкладке **General** находятся кнопки для предварительного просмотра печати (**Print Preview**), экспорта (**Export**) в файлы форматов BMP, WMF и EMF, экспорта в буфер обмена, подключения анимации (**Animated Zoom**), поля отступа (**Margins**) и кнопки подключения линеек скроллинга.
- На вкладке **Axis** настраивается внешний вид осей координат.
- На вкладке **Titles** настраиваются параметры верхнего (`Titles`) и нижнего заголовков (`Foot`).
- На вкладке **Legend** задаются параметры условных обозначений.
- На вкладке **Panel**, кроме настройки границ, предоставляется возможность настраивать цвет панели, задавать градиентный переход цветов или выбирать в качестве фона графический файл.
- На вкладке **Paging** задается число точек серии, выводимых на компонент, и предоставляются средства навигации по точкам серии.
- На вкладке **Walls** содержатся инструменты для работы со стенками.
- На вкладке **3D** определяется внешний вид системы координат (2D, 3D или ортогональный), задаются увеличение, повороты и смещение системы координат.

Вкладка **Series** окна редактора диаграмм содержит 4 вложенные вкладки — **Format**, **General**, **Marks**, **Data Source**. Опишем их назначение.

- На вкладке **Format** задается стиль элементов диаграммы, стиль ребер и граней этих элементов, взаимное расположение элементов серий относительно друг друга.
- На вкладке **General** выводятся компоненты, управляющие расположением и форматом числовых значений данных серии.
- На вкладке **Marks** определяются параметры ярлыков, которыми могут снабжаться данные.
- На вкладке **Data Source** указывается источник данных. Этого источника может не быть (`No Data`), данные могут задаваться случайными значениями (`Random Values`), могут быть вычислены с помощью функций (`Function`) и, наконец, данные для серий могут поступать из таблиц баз данных.

5.3. Создание новой диаграммы с компонентом *TChart* или *TDBChart*

Определение свойств компонентов `TChart` или `TDBChart` во многом сходно. Далее описываются общие шаги, необходимые для создания диаграмм любого из этих двух типов.

1. Создайте новую форму и разместите на ней компонент `TChart` или `TDBChart`. Установите размеры этого нового компонента, потянув за его угол.
2. Отредактируйте новую диаграмму. Для этого поместите указатель мыши над созданным компонентом и нажмите правую кнопку мыши. Появится контекстное меню (рис. 5.4), содержащее пункт **Edit Chart**.

   Выберите пункт контекстного меню **Edit Chart** для редактирования диаграммы и для определения данных `Series`. На экране появится окно редактора диаграммы (рис. 5.3).

   Вкладка **Chart** редактора диаграммы была описана в *разд. 5.2*. Некоторые параметры не будут доступны, пока не заданы данные `Series`, используемые в диаграмме. Изменив, например, параметр `Title`, вы увидите, что это сразу отразится на диаграмме.
3. Для добавления серий `Series` нажмите кнопку **Add** на вложенной вкладке **Series** вкладки **Chart**. В появившейся галерее типов диаграмм (см. рис. 5.1) выберите тип серии, который вы решили добавить к диаграмме. Позже тип серии можно будет изменить. После выполнения указанных действий окно редактора диаграммы будет выглядеть так, как показано на рис. 5.5.

   После выбора типа серии, к диаграмме добавляется новая серия `Series`, заполненная случайными значениями. Эти случайные значения будут видны только во время проектирования, а во время выполнения проекта по умолчанию используются нулевые данные. Поэтому необходимо вернуться к редактору диаграммы и добавить источник данных или написать код, задающий значения данных.

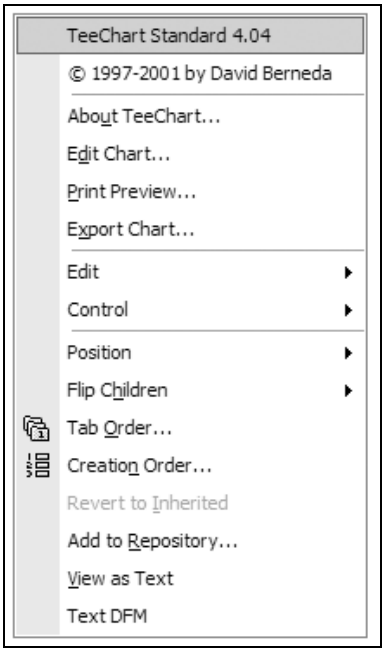


Рис. 5.4. Контекстное меню диаграммы

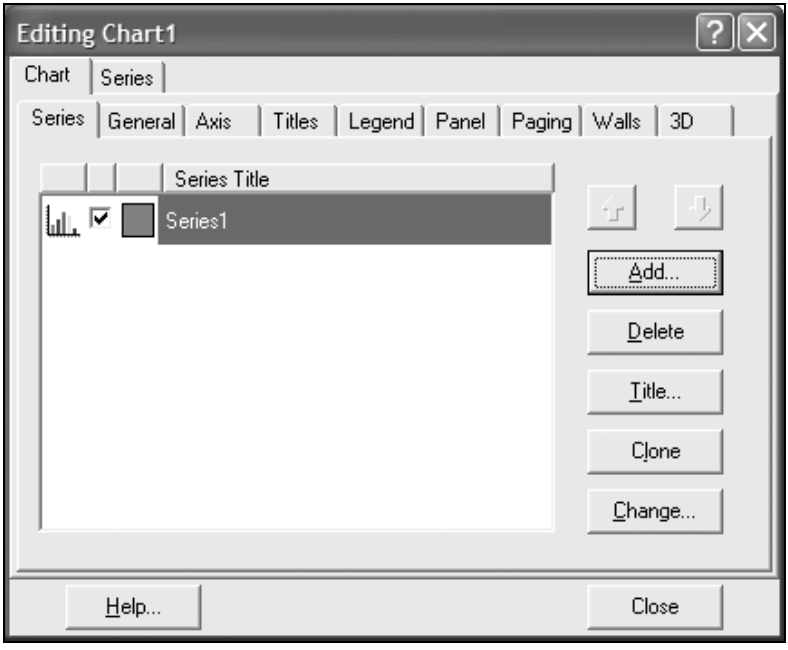


Рис. 5.5. Новая серия *Series*, добавленная к диаграмме

4. Следующий шаг — редактирование серии `Series`. На этом этапе серию заполняют данными. Действия, необходимые для включения данных в серию `Series` компонента `TChart` и для компонента `TDBChart`, немного отличаются.

Рассмотрим алгоритм добавления данных к компоненту `TChart`.

Напишем фрагмент программы на языке Pascal для того, чтобы добавить три значения в диаграмму. Поместим кнопку `TButton` на форму и, используя событие `OnClick`, вызовем три раза метод `Add` компонента `Series1` (листинг 5.1).

Листинг 5.1. Добавление значений в диаграмму

```
With Series1 do
Begin
       Add( 40, 'Pencil' , clRed ) ;
       Add( 60, 'Paper',  clBlue ) ;
       Add( 30, 'Ribbon', clGreen ) ;
end;
```

Описание метода `Add` и других методов и свойств доступно во время проектирования через справочную систему Delphi.

После запуска проекта на выполнение, нажмите кнопку **Button1** для того, чтобы увидеть диаграмму, создаваемую в результате работы приведенного в листинге 5.1 кода (рис. 5.6).

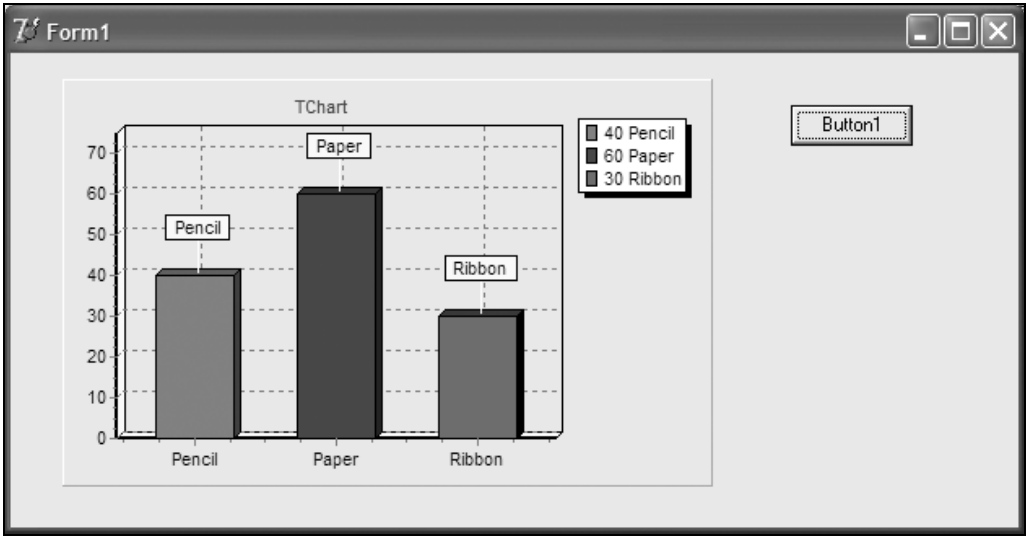


Рис. 5.6. Пример построения диаграммы с компонентом *TChart*

Закройте программу, чтобы возвратиться в среду Delphi. Во время проектирования диаграмма и ее свойство `Series` доступны через **Инспектор Объектов** Delphi, че-

рез **Редактор Диаграммы**, вызываемый с помощью пункта контекстного меню **Edit Chart**, или непосредственно из текста программы. Мы будем пользоваться для редактирования диаграммы и серии `Series` пунктом контекстного меню **Edit Chart**.

При изменении некоторых свойств `Series`, результат автоматически появляется на диаграмме. Для отмены изменений можно использовать кнопку **No**, хотя большинство параметров легко переключаются. Во время проектирования компонент `TChart` не управляет вашим кодом и поэтому показывает свой случайный набор данных. Только во время выполнения проекта будет видно, как новые параметры взаимодействуют с вашими данными.

Теперь рассмотрим более подробно добавление данных к компоненту `TDBChart`. Как уже говорилось в *разд. 5.1*, компонент `TDBChart` используется в случае, если данные для диаграммы берутся из таблиц баз данных.

До использования любых средств обслуживания базы данных, необходимо правильно установить BDE (Borland Database Engine) (или ее эквивалент) и компоненты VCL базы данных Delphi. Затем необходимо:

- разместить компонент `TTable` и направить его на базу данных, например, на используемые далее в примерах базу DBDEMOS и таблицу ANIMALS.DBF (эта база данных входит в стандартную поставку системы Delphi. Физически таблицы расположены в папке C: | Program Files | Common Files | Borland Shared);
- разместить компонент `TDatasource` и установить свойство `DataSet` на `Table1`;
- разместить компонент `TDBGrid` и установить свойство `DataSource` на `DataSource1`;
- установить значение свойства `Table1.Active` равным `True` для того, чтобы увидеть сетку, заполненную данными таблицы;
- разместить компонент `TDBChart` на форме;
- выбрать и установить серию `Series`, для этого:

  перейти к вкладке **Series** редактора диаграммы;

  выбрать нужную серию `Series` и перейти к списку источников данных;
- из списка выбрать тип источника данных.

Свойство `DataSource` может принимать следующие значения, соответствующие четырем возможным источникам данных:

- `No data`, если значения добавляются в исходном коде программы;
- `Random values`, если создаются диаграммы `Series` со случайными значениями;
- `A function`, если значения вычисляются с помощью математических функций. Значение `A function` может применяться к одной или нескольким сериям `Series` (например, `LineSeries1`, `BarSeries2` и т. д.), имеющимся на данной диаграмме, и использовать математические функции (`min`, `max`, `average` и другие);
- `Dataset`, значение которого может соответствовать компонентам `TTable`, `TQuery`, `TClientDataset` или любому другому компоненту `Tdataset`.

В рассматриваемом примере добавления набора данных выберем тип источника данных `Dataset`. В `TDBChart` будем включать данные из таблицы ANIMALS.DBF, уже добавленной к форме. После выбора типа источника данных `Dataset`, в окне редактора диаграммы откроется новая вкладка, предназначенная для выбора данных для `Series` (рис. 5.7).

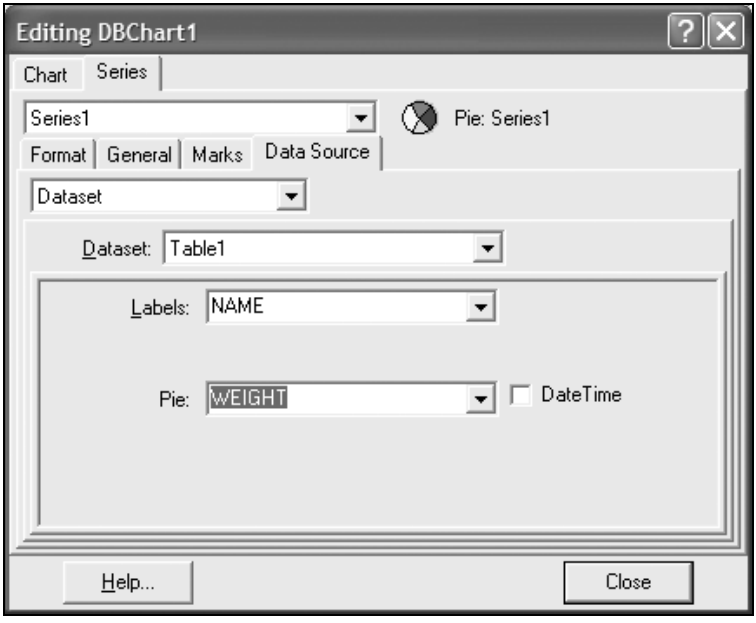


Рис. 5.7. Выбор источника данных для серии *Serie*s в окне редактора диаграммы

Данные для `Series` можно выбрать разными способами.

Откроем раскрывающийся список **Dataset**. Если на форме есть активизированные таблицы, то будет представлен их список. Выберем нужную таблицу и определим значения, которые будут использованы в диаграмме. Так как в данном примере выбран тип серии `Pie` (круговая диаграмма), то на экране будут представлены характерные для этого типа параметры конфигурации. Каждый тип `Series` имеет свои параметры, которые могут отличаться от примера, приведенного ниже.

Поле `Pie` выбирается из списка полей таблицы и должно содержать имя числового поля или строкового поля со значениями, которые могут быть преобразованы в числа. В нашем примере выберем поле `Weight` (таблица ANIMALS — список различных животных с информацией об их размере и весе).

Значения поля `Labels` можно выбирать произвольно для того, чтобы выводить соответствующую строку для каждой точки на диаграмме. В рассматриваемом примере выберем поле `Name`. Этот ярлык будет использоваться для создания условных обозначений `Legend` диаграммы.

Примечание

Почти все типы серии диаграммы требуют для представления точки два числа: величина `X` является специфической горизонтальной позицией для каждой величины `Y`. Если заменить серию `Pie` на серию другого типа, например `LineSeries`, то понадобится указать опции и для `X` и для `Y`. Для круговых диаграмм значений `X` нет.

Теперь нажмем на клавишу Enter, и компонент `TDBChart` покажет круговую диаграмму и условные обозначения `Legend` (рис. 5.8).

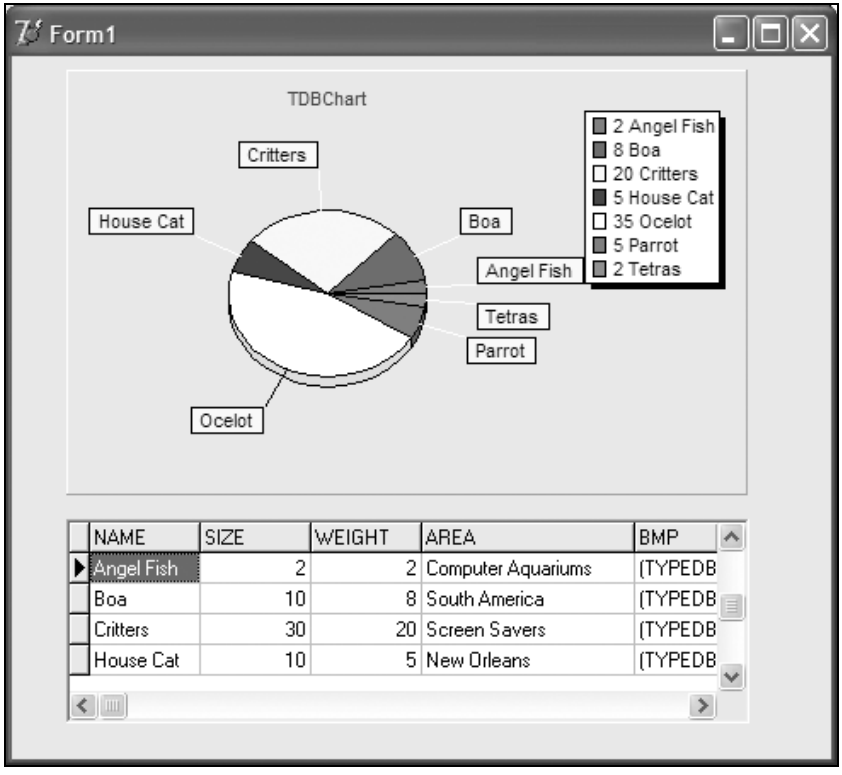


Рис. 5.8. Пример построения диаграммы с компонентом *TDBChart*

При изменении данных надо учитывать специфические свойства типа серии `Series`.

Для удаления точек используется метод `Delete`, имеющийся у каждого типа серий. Так, например, для серии типа `LineSeries` удаление пятой точки выглядит так: `LineSeries1.Delete (5)`.

Как показано в листинге 5.2, для изменения значений точек серии `Series` используются свойства `XValue` и `YValue`.

Листинг 5.2. Изменение значений в диаграмме

```
{в серии LineSeries }
LineSeries1.YValue[3]:= 27.1 ;

{в серии Bubble }
BubbleSeries1.RadiusValues.Value[8]:= 8.1 ;

{ в серии Pie }
PieSeries1.PieValues.Value[3]:= 111 ;
```

Для точек, данные которых представлены в формате "дата-время", значения DateTime задаются следующим образом:

во-первых, надо установить DateTime = True в списке значений X и (или) Y. Например, LineSeries1.XValues.DateTime := True;

во-вторых, надо использовать методы, описанные ранее, но задавать тип значений Date, Time или DateTime. Например,

```
LineSeries1.AddXY( EncodeDate(1999,1,23),25.4, 'Barcelona', clGreen ).
```

Наконец, необходимо установить свойства осей координат. Каждый компонент редактора диаграмм имеет четыре оси: LeftAxis, TopAxis, RightAxis и BottomAxis. Каждая ось имеет логическое свойство Automatic, установленное по умолчанию равным true. Это означает, что TeeChart будет всегда автоматически вычислять минимальные и максимальные значения данных на каждой оси.

Свойства Minimum и Maximum имеют тип double (числа с плавающей точкой), или представление DateTime в форматах Date, Time, Now, или любом другом допустимом формате "дата-время".

5.4. Соединение диаграммы с разными типами данных

Вернемся еще раз к проблеме подключения диаграммы TDBChart к таблицам с данными. Диаграммы TeeChart могут соединяться с тремя различными типами наборов данных: TTable, TQuery, TClientDataset.

Минимум свойств, который должен быть установлен в каждом случае:

- название базы данных;
- для TTable — имя таблицы TableName, в случае запроса — допустимая строка SQL, в случае ClientDataset — необходимо определить источник (с помощью контекстного меню объекта TClientDataset в форме).

Не забудьте активизировать набор данных, назначив значение свойства Active = true.

Когда в редакторе диаграммы выбирается новая серия, используется раскрывающийся список для выбора источника данных на вкладке **Series**. Если нужно связать данные с новым набором данных, нужно выбрать набор данных (`Dataset`) из раскрывающегося списка `ComboBox`. Вид вкладки **Data Source** будет зависеть от выбора типа набора данных и от выбранного типа `Series`. В табл. 5.3 представлены возможные значения свойства `DATASOURCE` для различных типов серий `Series`.

Таблица 5.3. *Значения свойства DATASOURCE для различных типов серий*

Тип серии *SERIES*	**Значения свойства *DATASOURCE***
Line	`XValues, YValues, Xlabel`
Fast Line	`XValues, YValues, Xlabel`
Bar	`XValues, YValues (called Bar), Xlabel`
Area	`XValues, YValues, Xlabel`
Point	`Xvalues, YValues, Xlabel`
Pie	`PieValues, Xlabel`
Arrow	`StartXValues, StartYValues, XLabel, EndXValues, EndYValues`
Bubble	`Xvalues, YValues, Xlabel, RadiusValues`
Gantt	`StartValues, EndValues, AY (Y axis level), AXLabel (Метка, дополнительно показанная на Y-оси, или маркер)`
Shape	`X0 (Top), Y0 (Bottom), X1 (Left), Y1 (Right)`
Bezier	`Xvalues, YValues, Xlabels`
Candle	`OpenValues, CloseValues, HighValues, LowValues, DateValues`
Contour	`Xvalues, YValues, Zvalues, Labels`
Error Bar	`Xvalues, YValues, Xlabel, ErrorValues`
Polar	`Xvalues, Yvalues`
3D Surface	`XValues, YValues, Zvalues`
Volume	`XValues, YValues (VolumeValues), Xlabel`

Диаграмму можно заполнять во время выполнения программы, написав код для добавления `Series` к диаграмме и определения полей этой серии. Такой код приведен в листинге 5.3 (в рассматриваемом примере пользователь расположил таблицу `Table1` на форме с полями `Name` и `Amount`).

Листинг 5.3. Изменение значений в диаграмме во время выполнения программы

```
Var MySeries, Ave: TLineSeries;
begin
```

```
        MySeries:=TLineSeries.Create( Self );
        With MySeries do
        begin
              ParentChart:=DBChart1;
              DataSource:=Table1;
              XLabelsSource:='Name';
              YValues.ValueSource:= 'Amount';
              CheckDatasource;
        end;
end;
```

5.5. Свойства компонента *TChart*

Компонент `TChart` является основным стандартным блоком для диаграмм, не использующим данные из баз данных.

Для определения характеристик диаграммы, добавления новой серии данных и изменения их свойств можно использовать **Редактор Диаграммы**. Серия данных для компонента `TChart` должна заполняться в коде программы.

Рассмотрим свойства компонента `TChart`.

- Свойство `AllowPanning` управляет прокруткой во время выполнения.

  ```
  property AllowPanning : TPanningMode
  ```

  Возможные значения этого свойства:

  `pmNone` — отвергать прокрутку;

  `pmHorizontal` — допускать только горизонтальную прокрутку;

  `pmVertical` — допускать только вертикальную прокрутку;

  `pmBoth` — допускать горизонтальную и вертикальную прокрутку.

- Логическое свойство `AllowZoom` позволяет переключать масштаб во время выполнения.

  ```
  property AllowZoom : Boolean
  ```

  Установка значения `AllowZoom = True` позволит изменять с помощью указателя мыши размеры окна во время выполнения проекта.

- Свойство `AnimatedZoom` определяет, будет ли изменение масштаба выполнено сразу или оно будет выполнено за несколько шагов.

  ```
  property AnimatedZoom : Boolean
  ```

  Если изменение масштаба будет выполнено за несколько шагов, то значение `AnimatedZoom = true`. По умолчанию значение свойства равно `false`. Свойство `AnimatedZoomSteps` управляет числом шагов изменения масштаба и может быть полезным, когда диаграмма содержит много точек.

5.6. Типы *Series*

Компонент TChartSeries — предок всех типов серий Series. Рассмотрим более детально различные типы Series и предоставляемые ими возможности при создании пользовательских приложений.

5.6.1. Серии *Line* и *Fast Line*

Создавая диаграммы серий Line и Fast Line, компонент TLineSeries выводит все точки диаграммы, рисуя линию между ними. Методы AddBar или AddXY используются для заполнения диаграммы вручную.

Компонент TLineSeries происходит от класса TCustomSeries, которому он добавляет поддержку для подкомпонента Pointer, маркеры и обработку событий.

Свойство LinePen определяет тип пера, используемого для рисования линии между точками.

Используйте LineBrush в случае использования серии 3D для того, чтобы изменить шаблон кисти линии.

Логическое свойство Stairs управляет видом линий, соединяющих точки — позволяет выводить эти линии в виде ступенек. Свойство InvertedStairs можно использовать для изменения порядка следования ступенек.

Вы можете менять форму шаблона точек, используя свойства подкомпонента Pointer.

Свойство Pointer.Style определяет тип формы. Свойство Pointer.Brush используется, чтобы изменить шаблон заполнения точки, а свойство Pointer.Pen определяет тип используемого пера. Размеры точки могут быть изменены с помощью свойств Pointer.HorizSize и Pointer.VertSize.

Для использования доступны два типа серий линейных диаграмм: Line и Fast Line. Как и указывает имя, диаграмма Fast Line ("быстрая линия") рисуется быстрее, но диаграммы Line поддерживают большее количество возможностей.

В листинге 5.4 приведен пример кода, который использует для изображения серии Line свойство Stairs.

Листинг 5.4. Пример использования свойства *Stairs* для серии *Line*

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var t:integer;
begin
        Randomize;
        with Series1 do
        for t:=1 to 12 do
                AddXY(t, Random(70), IntToStr(t),clTeeColor);
```

```
        with Series2 do
        begin
                Stairs:=true;
                InvertedStairs:=true;
                for t:=1 to 12 do
                        AddXY(t, Random(70), IntToStr(t),clTeeColor);
        end;
end;
```

Результат выполнения кода листинга 5.4 приведен на рис. 5.9.

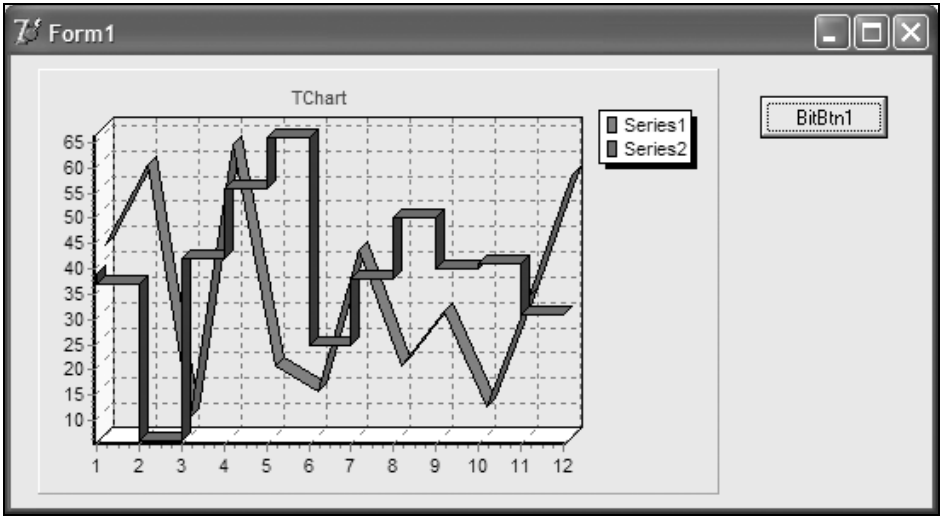


Рис. 5.9. Серия *Line 1.A 3D* со свойством ступенек

Серию `Fast Line` можно использовать только на плоской диаграмме, и она действительно выводится очень быстро. Если при использовании `Fast Line` приблизительно известно количество добавляемых к серии точек данных, то можно использовать глобальную переменную `TeeDefaultCapacity` для подготовки памяти. Например, обрабатывая серию данных размером 10000 точек, необходимо устанавливать `TeeDefaultCapacity = 10000`. Это позволит за один проход распределить память для заполнения всей серии. Иначе распределение памяти будет происходить пошагово и займет больше времени.

5.6.2. Серия *Bar*

Серия столбчатых диаграмм `Bar` использует разнообразные стили (рис. 5.10), может быть плоской или трехмерной.

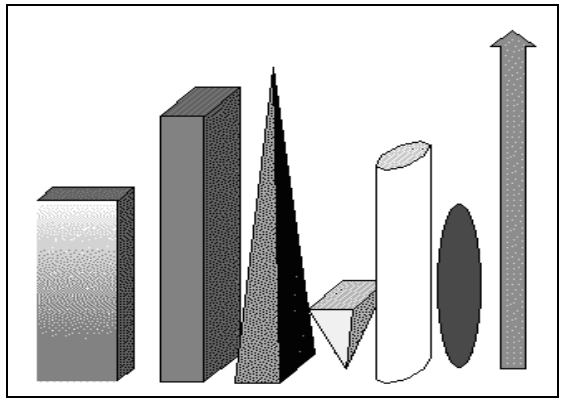


Рис. 5.10. Различные стили *Bar*

Предположим, что нам требуется визуально отобразить результаты работы компании за 12 месяцев, причем у нее есть подразделения East, West и Other. Для реализации такой задачи можно использовать построение столбчатой диаграммы. Для построения столбчатой диаграммы `Bar`, в которой используются различные типы стилей, применим следующий алгоритм:

1. Перетащим компонент диаграммы на форму.
2. Щелчком правой кнопки мыши над диаграммой вызовем контекстное меню, в котором выберем пункт **Редактор Диаграммы**.
3. Добавим три серии `Bar` и, используя кнопку **Title** на вкладке **Series** редактора диаграммы, дадим им имена `East`, `West` и `Other`.
4. Выберем на вложенной вкладке **Format** вкладки **Series** группу `Multiple Bar` и установим в ней значение `stacked` (это изменение относится сразу ко всем трем сериям и задает расположение столбцов друг над другом).
5. На вкладке **Titles** добавим название диаграммы (TChart).
6. Перейдем на вкладку **Axis** (оси). Выберем ось `Left`. Мы выбираем тип оси с автоматическим размером и на вложенной вкладке **Minor** вкладки **Axis** задаем ее параметры для промежуточных меток — размер меток `Minor Ticks` и частоту их размещения на оси `Minor Grid`. Аналогично поступаем применительно к нижней (`Bottom`) оси.

Каждая серия имеет свою форму столбцов (рис. 5.11): East — обычная пирамида, West — перевернутая пирамида, Other — параллелепипед. Форму столбцов можно задать на вкладке **Format**, имеющей опцию для задания стиля **Style**. То же задание стиля можно выполнить в тексте программы.

```
East.BarStyle:= bsPyramid ;
West.BarStyle:= bsInvPyramid ;
Other.BarStyle:= bsRectangle ;
```

Порядком следования серий можно управлять или из редактора диаграммы во время разработки, или в тексте программы через свойство `Serieslist`:

```
Chart1.SeriesList[ 0 ] := East;
Chart1.SeriesList[ 1 ] := West;
Chart1.SeriesList[ 2 ] := Other;
```

Данные серий этой демонстрационной диаграммы (листинг 5.5) заполнены случайными величинами.

Листинг 5.5. Заполнение значений серий *East*, *West*, *Other* случайными величинами

```
procedure Form1.Button1Click(Sender: TObject);
var t:integer;
begin
        Randomize;
        with East do
for t:=1 to 12 do
        AddY( Random(70), ShortMonthNames[t],clTeeColor);
        with West do
for t:=1 to 12 do
        AddY( Random(70), ShortMonthNames[t],clTeeColor);
        with Other do
for t:=1 to 12 do
        AddY( Random(30), ShortMonthNames[t],clTeeColor);
end;
```

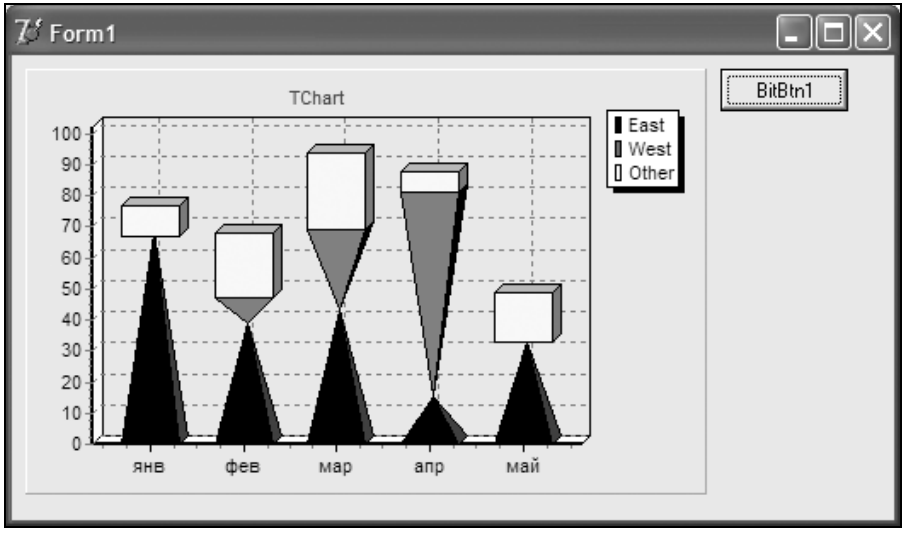


Рис. 5.11. Диаграмма, состоящая из трех столбчатых серий

Данные заполнены за 12 месяцев, но выводятся только за 5 месяцев (рис. 5.11). Вложенная вкладка **Paging** вкладки **Chart** редактора диаграмм имеет опцию для задания числа точек, отображаемых на диаграмме.

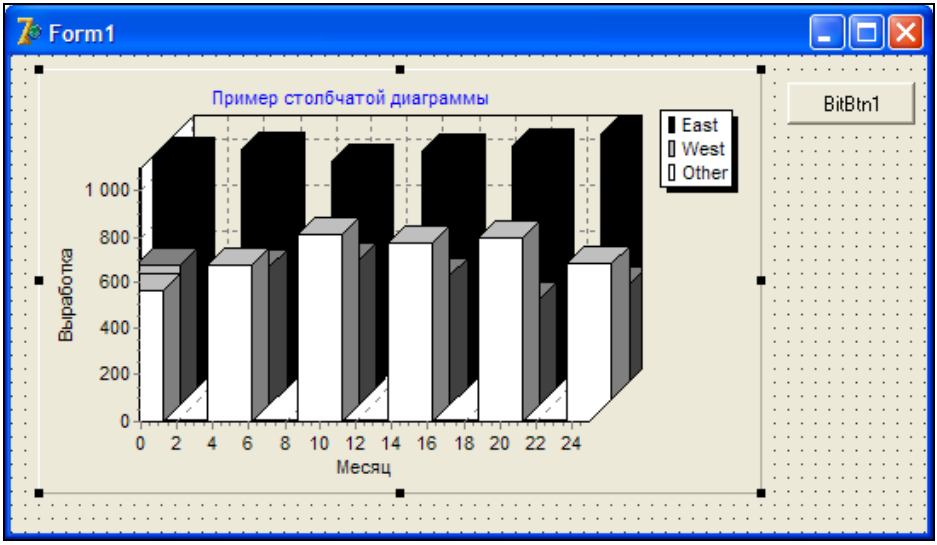


Рис. 5.12. Диаграмма серии *Bar* при *Multiple Bar=None*

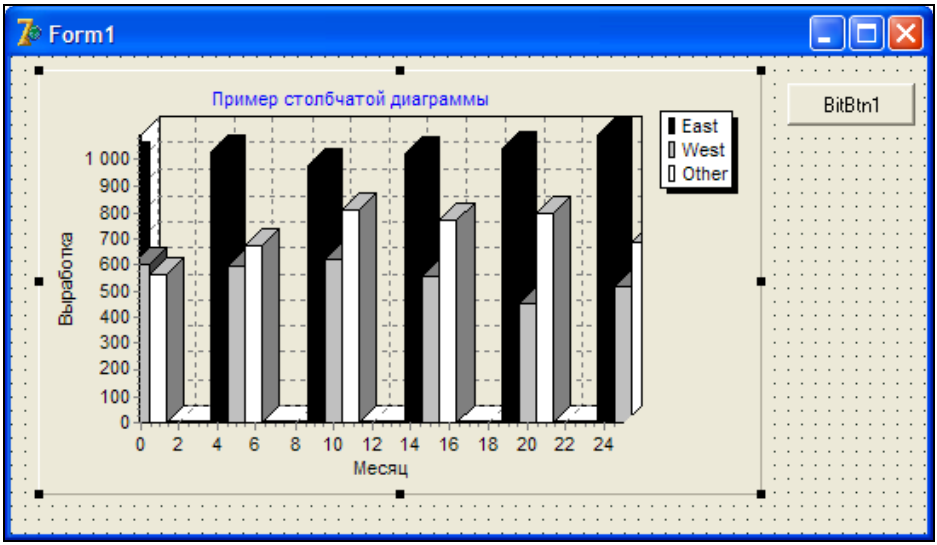


Рис. 5.13. Диаграмма серии *Bar* при *Multiple Bar=Side*

Параметр `Multiple Bar` для всех серий устанавливается на вложенной вкладке **Format** вкладки **Series**. При этом выравнивание устанавливается для всех серий

диаграммы, и нет необходимости заходить в каждую для изменения параметра `Multiple Bar`.

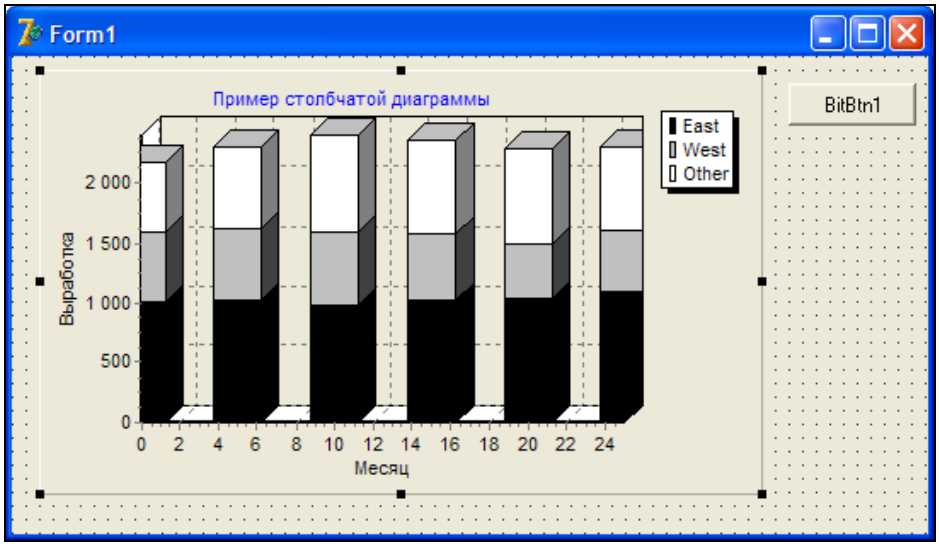


Рис. 5.14. Диаграмма серии *Bar* при *Multiple Bar=Stacked*

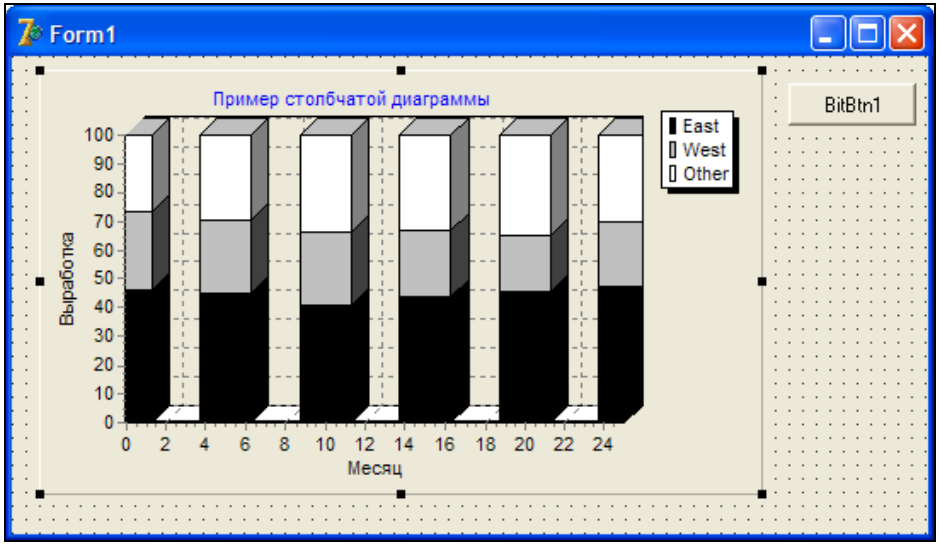


Рис. 5.15. Диаграмма серии *Bar* при *Multiple Bar=Stacked 100%*

На рис. 5.12—5.15 показаны различные варианты стилей, используемые для отображения столбчатых диаграмм. Все диаграммы содержат одну и ту же исходную информацию, но используют различные значения параметра `Multiple Bar`, опреде-

ляющего взаимное расположение столбцов диаграммы: рядом (`Side`), друг над другом (`Stacked`), друг над другом с нормированием по высоте (`Stacked 100%`).

5.6.3. Серия *Horizontal Bar*

Серия горизонтальных полос `Horizontal Bar` обладает такими же свойствами, как и рассмотренная в *разд. 5.6.2* серия `Bar`. Эту серию удобно использовать в случае длинных подписей у точек. Компонент `THorizBarSeries` заполняет все точки как горизонтальные полосы. Различные типы `ThorizBarSeries`, в зависимости от значения свойства `MultiBar`, могут быть отображены как горизонтальные полосы, ряды полос один за другим, сложенные или сложенные и пронормированные к 100%.

Методы `AddBar` или `AddXY` используются для заполнения диаграммы вручную.

Замечание

Обратите внимание на то, что для `THorizBarSeries` координаты `X` и `Y` инвертированы. Метод `AddBar` делает это преобразование автоматически и рекомендуется для добавления точек.

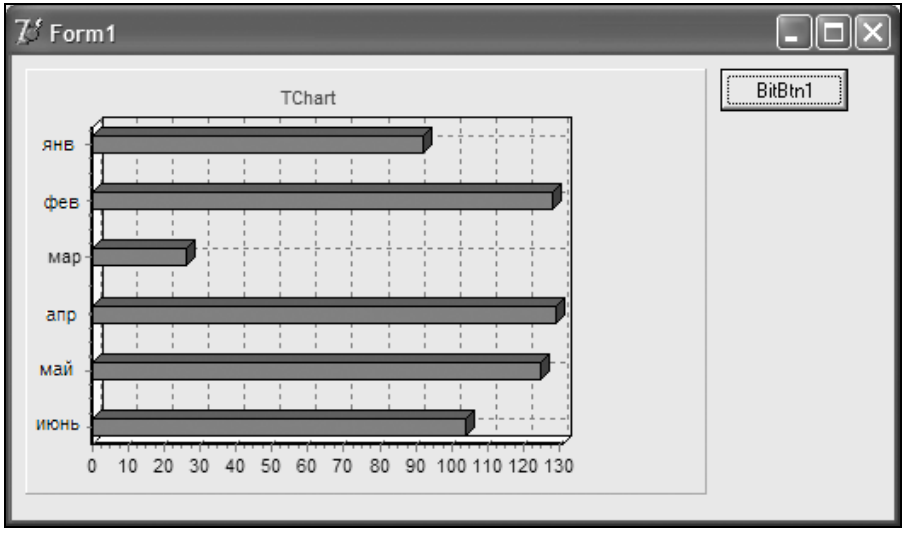


Рис. 5.16. Диаграмма серии *Horizontal Bar*

В следующем примере приведен участок кода, который добавляет к диаграмме 6 горизонтальных полос.

```
for t:= 1 to 6 do
        Series1.AddXY(MyValues[t],t,MyLabels[t],MyColor[t]);
```

Результат работы этого кода приведен на рис. 5.16.

Свойство `BarBrush` определяет шаблон для заполнения полосы, а свойство `BarPen` используется для задания границы полосы. Для изменения стиля полосы (цилиндры, пирамиды, и т. п.) используется свойство `BarStyle`. Для изменения относительного расстояния между полосами можно использовать свойство `BarWidthPercent`. Точная высота полосы устанавливается свойством `CustomBarHeight`. Свойство `Dark3D` изменяется, если боковые стороны полосы заполняются более темным цветом, чем передняя стенка полосы. Свойство `OffsetPercent` определяет вертикальное смещение полос.

По умолчанию, полоса использует нулевую горизонтальную координату. Установите свойство `YOrigin` в желаемом значении или установите `UseYOrigin = False` для того, чтобы сделать ширину полосы минимальной.

5.6.4. Серия *Area*

Создавая диаграммы серии `Area` (рис. 5.17), компонент `TAreaSeries` соединяет все точки, рисуя ломаную линию между ними, и заполняет область, определенную этой линией и нижней плоскостью диаграммы.

Методы `AddBar` или `AddXY` используются для заполнения диаграммы вручную. Свойство `AreaBrush` используется для выбора шаблона заполнения области. Свойство `AreaColor` определяет цвет фона области. Стиль линии определяется свойством `AreaLinesPen`.

Свойство `Stairs` управляет соединением линий между точками в виде ступенек. Свойство `InvertedStairs` можно использовать для изменения порядка следования ступенек.

Стиль линий может изменяться при изменении свойства `LinePen`.

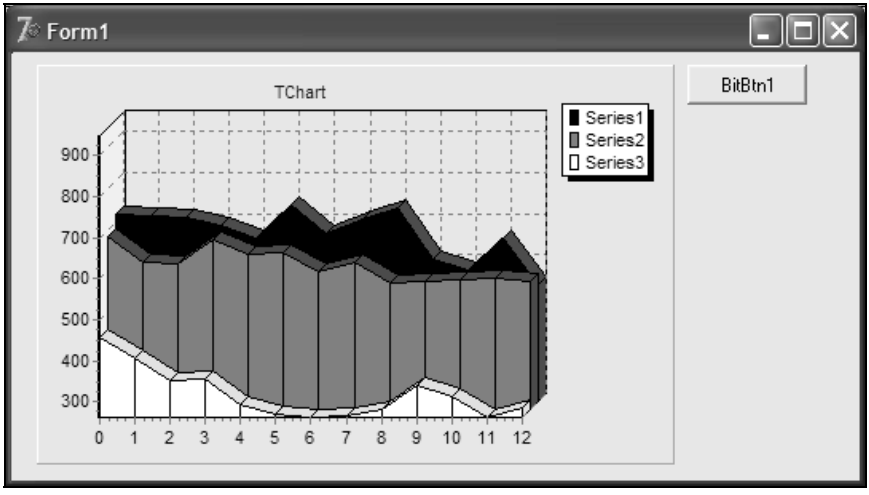


Рис. 5.17. Диаграмма серии *Area*

5.6.5. Серия *Point*

Создавая диаграммы серии `Point` (рис. 5.18), компонент `TPointSeries` заполняет все точки, используя свойства подкомпонента `Pointer`.

Компонент `TPointSeries` также происходит от класса `TCustomSeries`, которому он добавляет поддержку для подкомпонента `Pointer`, маркеры и обработку событий нажатия. Методы `AddBar` или `AddXY` используются для заполнения диаграммы вручную.

Свойство `Pointer.Style` определяет тип шаблона. Свойство `Pointer.Brush` используется для того, чтобы изменить шаблон заполнения точки, а свойство `Pointer.Pen` определяет тип используемого пера. Размеры точки могут быть изменены с помощью свойств `Pointer.HorizSize` и `Pointer.VertSize`.

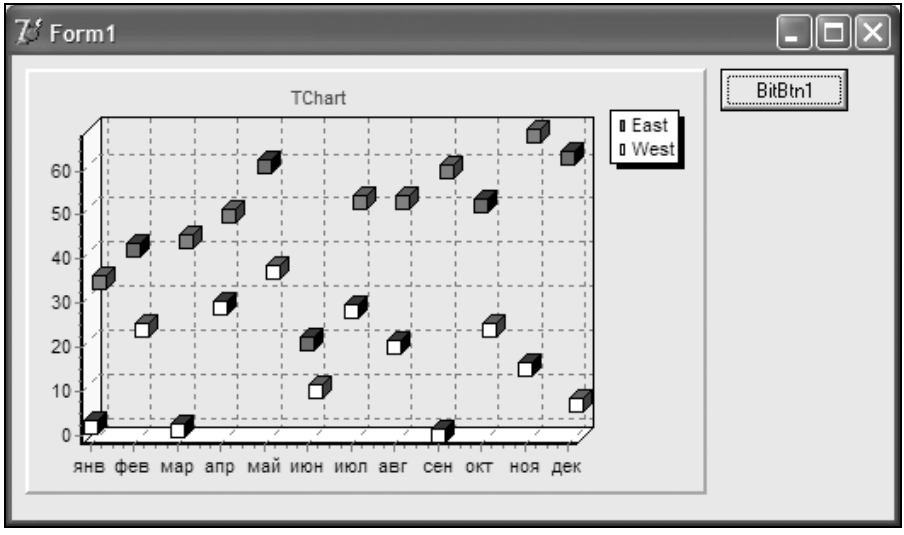


Рис. 5.18. Диаграмма серии *Point*

5.6.6. Серия *Pie*

Создавая диаграмму серии `Pie` (рис. 5.19), компонент `TPieSeries` придает ей форму эллипса ("пирога").

Для заполнения секторов круговой диаграммы вручную можно использовать метод `AddPie`. Свойство `PiePen` определяет тип пера. Свойства `TChart.View3D` и `TChart.Chart3DPercent` управляют трехмерным изображением круговой диаграммы. Установите свойство `UsePatterns = true` для заполнения секторов круговой диаграммы шаблоном кисти. Свойство `Color3D` управляет окрашиванием трехмерной круговой диаграммы. Свойства `ShadowColor` и `Shadowed3D` создают теневой эффект

для трехмерной круговой диаграммы. Свойство `PieValues` служит для получения доступа к значениям круговой диаграммы.

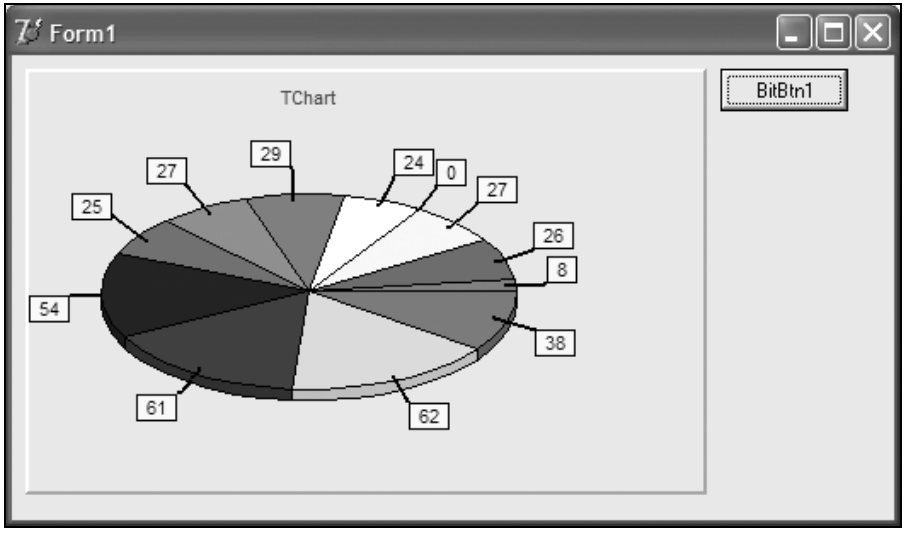


Рис. 5.19. Диаграмма серии *Pie*

5.6.7. Серия *Arrow*

Создавая диаграммы серии `Arrow`, компонент `TArrowSeries` изображает все точки в виде стрелок.

Для заполнения точек диаграммы вручную можно применять метод `AddArrow`. Параметры вершины стрелки могут быть изменены с помощью свойств `ArrowHeight` и `ArrowWidth`. Стрелки всегда рисуются только на плоскости, и каждая стрелка определяется своими четырьмя координатами: свойствами `StartXValues`, `StartYValues`, `EndXValues` и `EndYValues`.

Пример кода, заполняющего случайными значениями 12 точек диаграммы серии `Arrow`, приведен в листинге 5.6, а результат работы приложения с таким кодом представлен на рис. 5.20.

Листинг 5.6. Создание диаграммы серии *Arrow*

```
with Series1 do
begin
        StartXValues.DateTime:=False;
        Clear;
        for t:=1 to 12 do
```

```
        begin
                x0 :=t;
                y0 :=Random( 1000 );
                x1 :=x0 + Random( 10 )/5 ;
                y1 :=y0 + Random( 1000 ) - 500 ;
                AddArrow( x0, y0, x1, y1, '', clBlue);
        end;
end;
```

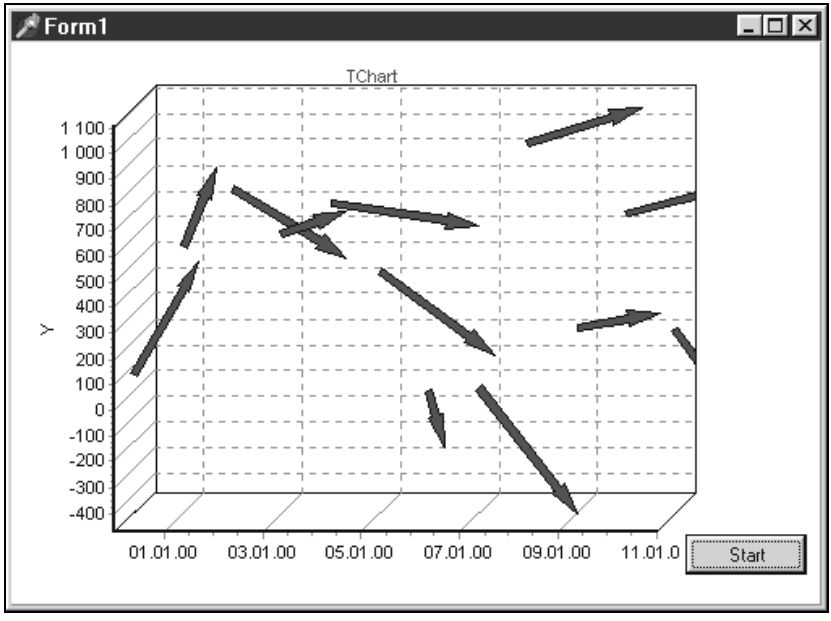


Рис. 5.20. Диаграмма серии *Arrow*

5.6.8. Серия *Bubble*

Создавая диаграммы серии `Bubble` (рис. 5.21), компонент `TBubbleSeries` изображает все точки в виде эллипсов ("пузырей"). Компонент имеет три настраиваемых параметра, которые определяют размеры "пузырей": `XValues`, `YValues` и `RadiusValues`. Эта серия полезна для показа веса значения. Обратите внимание на то, что у каждого "пузыря" свой радиус и все значения радиусов хранятся в свойстве `RadiusValues`. Можно использовать метод `AddBubble` для того, чтобы вручную заполнить точки диаграммы.

Свойство `Pointer` определяет все атрибуты форматирования. По умолчанию, величина радиуса (`RadiusValues`) выражается в единицах, установленных для координат вертикальной оси.

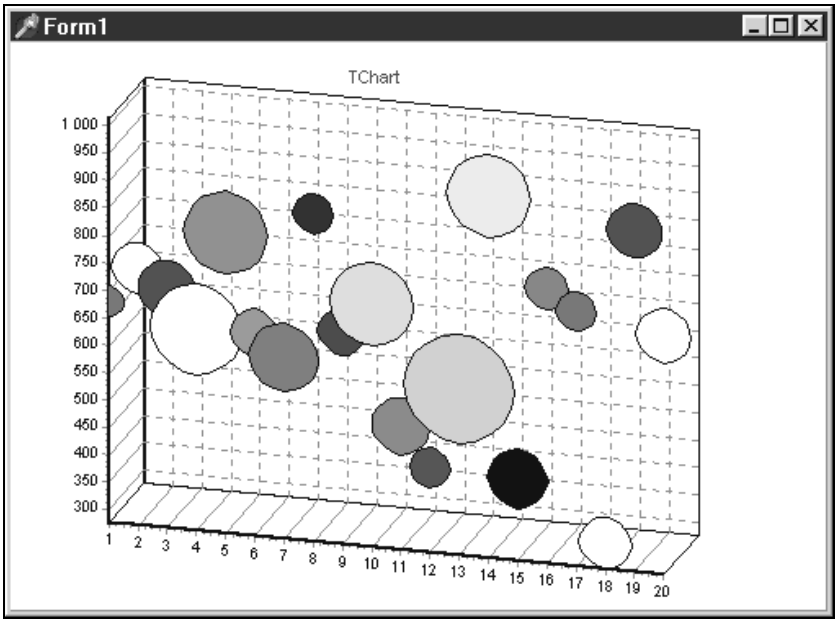


Рис. 5.21. Диаграмма серии *Bubble*

5.6.9. Серия *Gantt*

Создавая диаграммы серии `Gantt`, компонент `TGanttSeries` изображает все точки как горизонтальные полосы на разных высотах. Этот компонент может использоваться для того, чтобы отобразить последовательность временных заданий, отсортированных по времени вдоль горизонтальной оси.

Используйте методы `AddGantt` или `AddGanttColor` для того, чтобы вручную заполнить точки полосы.

Каждая полоса имеет начальное и конечное значение. Данные диаграмм серии `Gantt` могут иметь тип `TDateTime` или `Double`.

Свойство `StartValues` хранит начальные значения для всех полос `Gantt`. Свойство `EndValues` хранит конечные значения для всех полос `Gantt`.

Далее приводится пример (листинг 5.7) в котором изменяются значения в серии `Gantt`.

Листинг 5.7. Создание диаграммы серии *Gantt*

```
Var tmp1,tmp2 : Longint;
begin
        { Очистка всей серии Gantt }
        GanttSeries1.Clear;
```

```
        { Добавить первый Gantt }
        tmp1:=GanttSeries1.AddGantt(EncodeDate(1998,1,1),
                EncodeDate(1998,1,31),
                0,
                'Programming' );
        { Добавить второй Gantt }
        tmp2:=GanttSeries1.AddGantt(EncodeDate(1998,3,1),
                EncodeDate(1998,3,31),
                1,
                'Testing' );
        { Соединить первый и второй Gantt }
        GanttSeries1.NextTask[ tmp1 ]:= tmp2 ;
end;
```

Свойство `ConnectingLinePen` определяет параметры пера, используемого для соединения полос. Свойство `Pointer.VertSize` определяет высоту полос.

Результаты работы кода приведены на рис. 5.22.

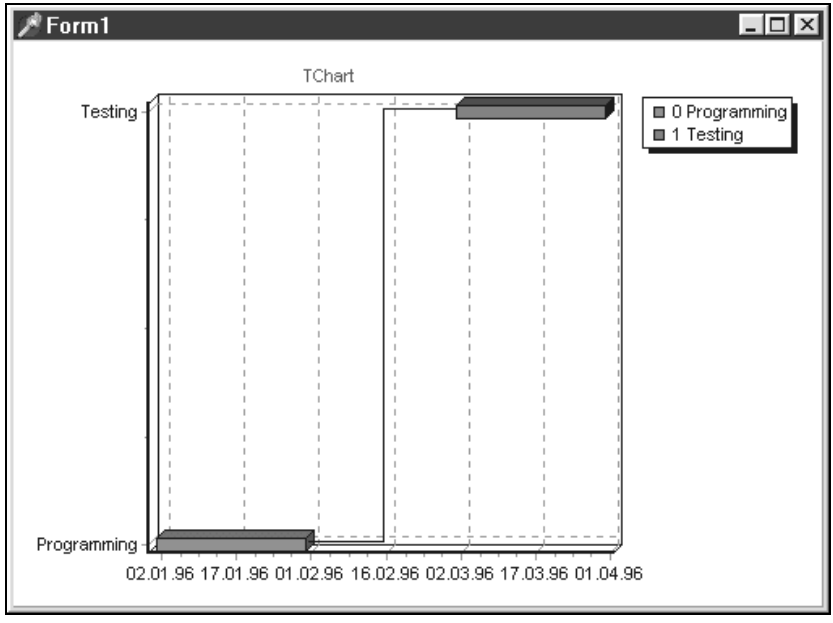


Рис. 5.22. Диаграмма серии *Gantt*

5.6.10. Серия *Shape*

Компонент `TChartShape` является специальной серией. Он позволяет устанавливать в диаграмме стандартные фигуры, на которые можно выводить текст. Этот компонент работает подобно стандартному компоненту `TShape`.

Размеры фигуры задаются координатами ее левого верхнего и правого нижнего углов `X0`, `Y0`, `X1` и `Y1`. Стиль фигуры определяется свойством `Style`. Возможные значения этого свойства и соответствующие им формы фигур приведены в табл. 5.4. По умолчанию используется значение `chasHorizLine`.

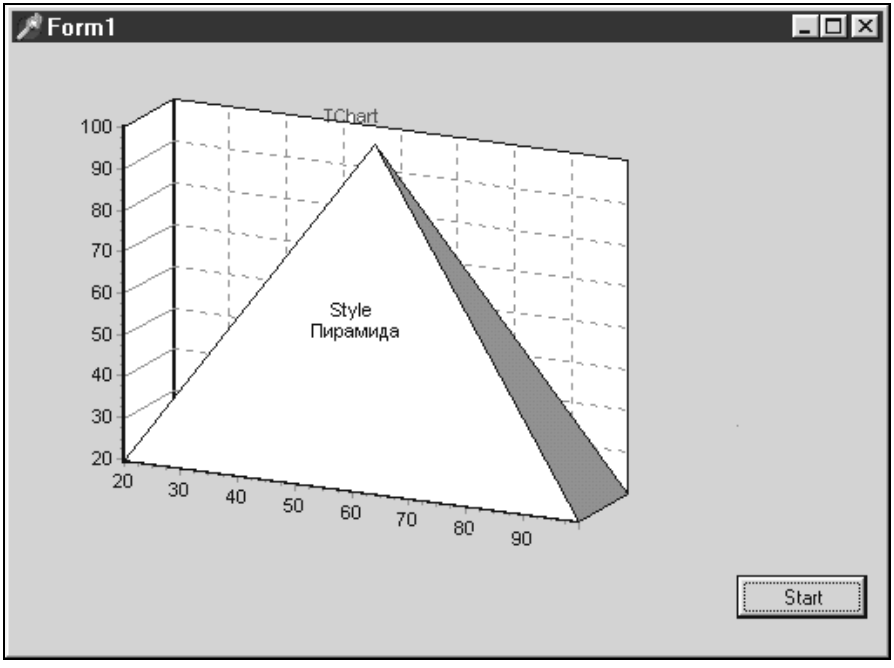


Рис. 5.23. Диаграмма серии *ChartShape*

Таблица 5.4. *Возможные значения свойства Style компонента TShape*

Значение	**Форма**
`chasRectangle`	Прямоугольник
`chasCircle`	Круг
`chasVertLine`	Вертикальная линия
`chasHorizLine`	Горизонтальная линия
`chasTriangle`	Треугольник
`chasInvertTriangle`	Инвертированный треугольник
`chasLine`	Линия
`chasDiamond`	Алмаз
`ChasCube`	Куб

Таблица 5.4 *(окончание)*

Значение	Форма
ChasCross	Крест
ChasDiagCross	Диагональный крест
ChasStar	Звезда
ChasPyramid	Пирамида
ChasInvPyramid	Перевернутая пирамида

Желаемая форма фигуры (прямоугольник, эллипс и т. п.) выбирается с помощью свойства Style. Свойства кисти (Brush) и пера (Pen) определяют цвет и атрибуты фигуры. Метод Clicked может использоваться для того, чтобы проверить, есть ли указатель мыши над компонентом Shape. Компонент Shape можно масштабировать и перемещать, как и любой другой тип серии.

Пример использования серии Shape приведен на рис. 5.23 (значение свойства Style= ChasPyramid).

5.6.11. Комбинированные серии

На диаграмме можно совмещать любое количество разнотипных серий. Почти все типы совместимы друг с другом. Если совместимости нет, то в галерее типов такая серия выделяется серым цветом и ее невозможно добавить на диаграмму.

С помощью встроенных функций можно создавать серию с информацией, обобщающей данные других серий. Например, Series3 может быть добавлена к нашему проекту (рис. 5.24) с помощью функции сложения TAddTeeFunction, используя **Редактор Диаграммы** во время разработки (рис. 5.26) или в результате выполнения приложением следующего кода (листинг 5.8).

Листинг 5.8. Создание комбинированной серии с помощью функции *TAddTeeFunction*

```
procedure TForm1.BitBtn1Click(Sender: TObject);
var t:integer;
        x0,y0,x1,y1 : Double;
                tmp1,tmp2 : Longint;
begin
        Series1.Clear;
        Series2.Clear;
        with Series1 do
                for t:=1 to 8 do
                        AddXY(t, Random(70), IntToStr(t),clTeeColor);
        with Series2 do
```

```
                for t:=1 to 8 do
                        AddXY(t, Random(70), IntToStr(t),clTeeColor);
        { установить функцию, используя метод SetFunction }
        Series3.SetFunction(TAddTeeFunction.Create(Self));
end;
```

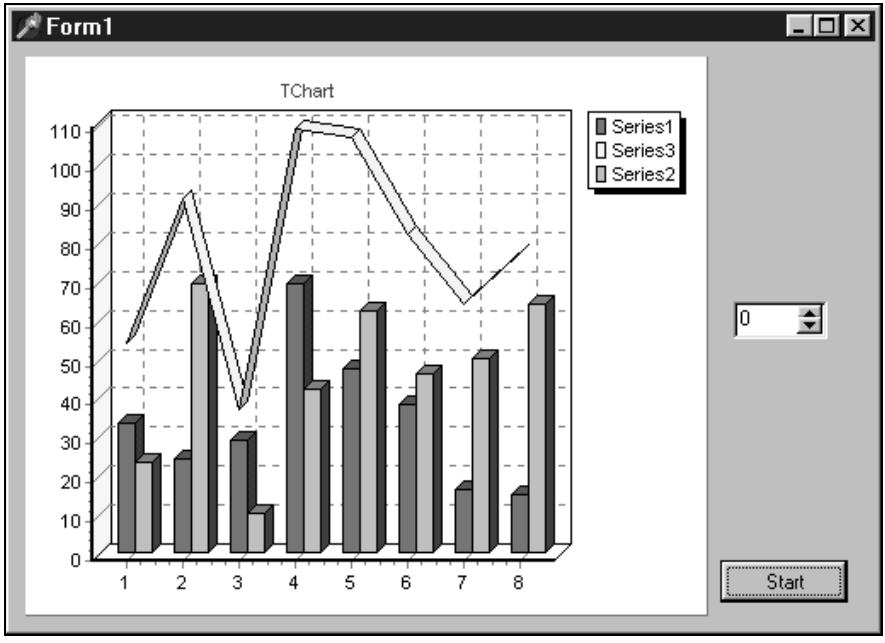


Рис. 5.24. Комбинированные серии

5.7. Функции для вычисляемых серий

Чтобы определять данные вычисляемых серий по данным других серий, можно использовать функции. Функция может создаваться с использованием других функций и ведет себя, как и любая серия. Она отличается от серии только в определении источника данных. Во время разработки функции могут быть добавлены с помощью редактора диаграммы. Чтобы выбрать тип функции, можно использовать галерею TeeChart. После добавления функции необходимо открыть вложенную вкладку **Data Source** на вкладке **Series** редактора диаграммы, где можно выбрать входную серию (или несколько серий) для функции.

На рис. 5.25 показана вкладка **Series** окна редактора диаграммы, на которой вычисляемая серия `Series3` определяется с помощью функции `Add` (сложение) как сумма серий `Series1` и `Series2`.

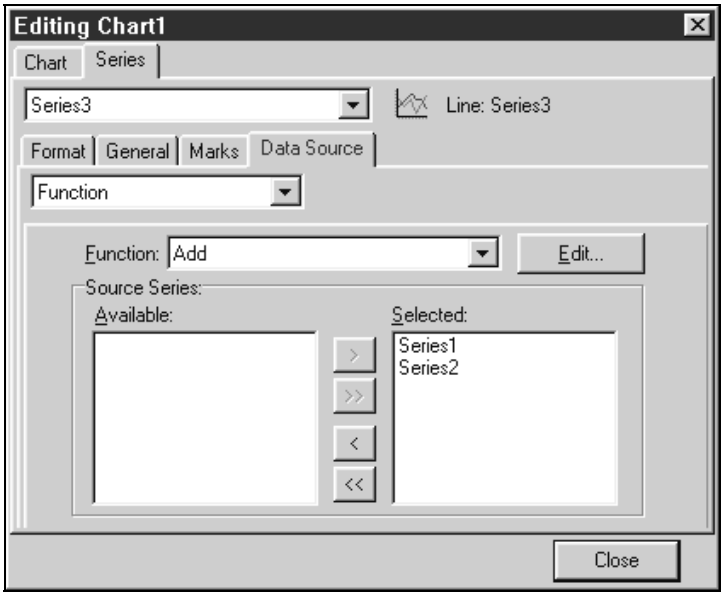


Рис. 5.25. Вкладка **Series**,
на которой устанавливается связь вычисляемой серии

Можно удалить серию, добавленную как носитель функции. Для этого нужно удалить серию в окне редактора диаграммы. Можно переопределить серию как имеющую другой источник данных на вложенной вкладке **Data Source** вкладки **Series**.

Для изменения типа функции с помощью редактора диаграммы нужно выбрать вложенную вкладку **Data Source** вкладки **Series**. Раскрывающийся список **Function** содержит список всех типов функций.

Для дополнения функции кодом необходимо добавить новую функцию и установить тип функции:

```
Series1.SetFunction(TAddTeeFunction.Create(Self));
```

Для удаления функции из серии удалите серию или используйте метод `SetFunction` для отключения соединения между серией и функцией:

```
Series1.SetFunction(nil);
```

Свойство `Period` чрезвычайно полезно для работы с функциями. Оно предназначено определять частоту, с которой вычисляется функция.

Пусть есть 6 точек данных (например, полосы серии `Bar`) с величинами: 3, 8, 6, 2, 9 и 12. Определяем функциональную серию при `Period=0` (по умолчанию, только одна серия является вводом в функцию). Получим среднее значение этих чисел: 6,667. При `Period=2` функцией будут возвращены 3 величины средних значений чисел: 5,5, 4 и 10,5. Эти величины соответствуют среднему значению их области пе-

риода, т. е. первая величина определяется между 1 и 2 точкой входной серии, вторая величина — между 3 и 4 точкой и т. д.

Можно модифицировать свойство `Period`, выбирая функцию в инспекторе объектов или используя следующий код: `FunctionType:Series2.FunctionType.Period:=2`.

5.7.1. Функция *TAddTeeFunction*

Функция `TAddTeeFunction` (сложение) складывает данные из одной или нескольких серий. Если создать линейную серию `Series1`, определить серию `Function Add` и установить свойство `Period=0`, то получим диаграмму с `Series1` и `Function Add` в виде горизонтальной линии, расположенной в самом верху диаграммы (рис. 5.26), которая является суммой всех значений `Series1`.

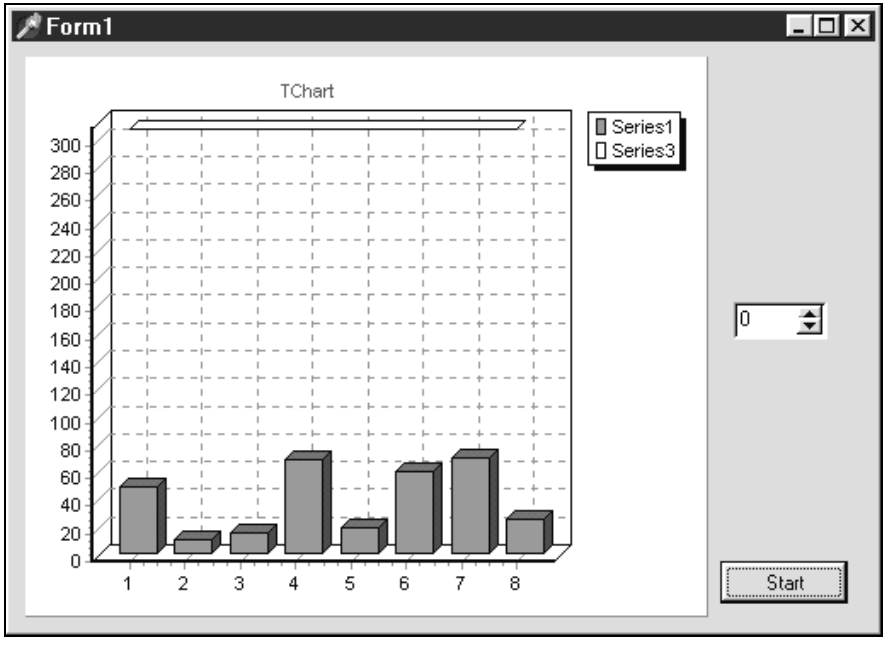


Рис. 5.26. Использование функции *Add* (*period*=0)

Мы можем изменить серию `Function Add`, чтобы представить серию `Series1` группировками точек по две (1+2), (3+4), (5+6), (7+8). Для этого установим значение свойства `Period=2` (вкладка **Data source** окна редактора диаграммы). В коде программы это будет выглядеть так: `Series2.FunctionType.Period: = 2`.

Значение свойства `period`, равное `2`, заставляет функцию `TAddTeeFunction` складывать каждые две точки (см. рис. 5.27).

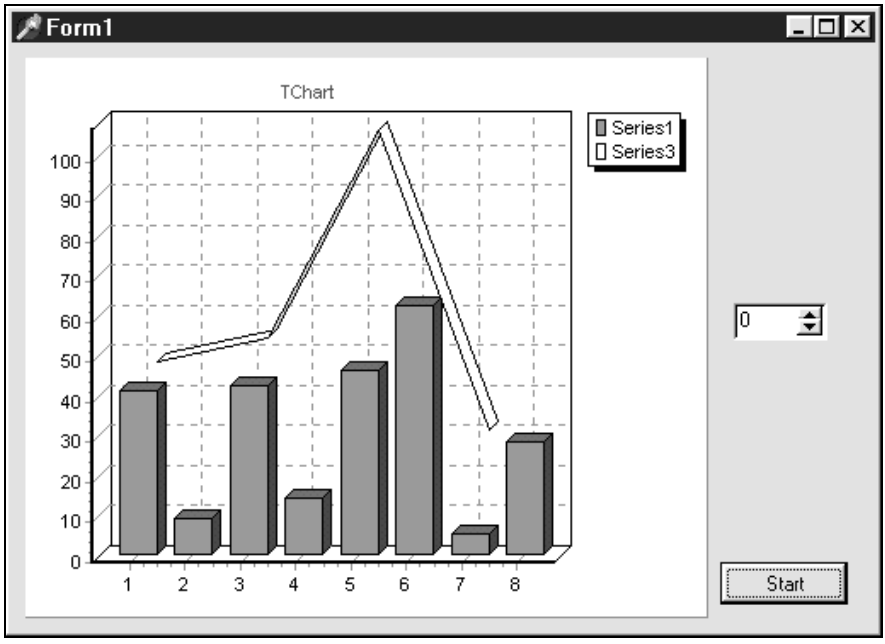


Рис. 5.27. Использование функции *Add* (*period*=2)

5.7.2. Функция *TSubtractTeeFunction*

Функция `TSubtractTeeFunction` (вычитание) может быть добавлена к проекту редактором диаграммы во время разработки или в коде программы. Функция производит вычитание одной серии из другой в каждой точке оси. Из первой серии в списке редактора диаграммы вычитается вторая серия. Значение свойства `period` для этой функции по умолчанию равно `1`.

Следующая диаграмма (рис. 5.28) построена со значением свойства `ApplyZOrder := false`, что приводит к изображению всех серий в одном и том же трехмерном пространстве (на рис. 5.28 `Series2 - Series1=Series3`).

5.7.3. Функция *TMultiplyTeeFunction*

Функция `TMultiplyTeeFunction` (умножение) может быть добавлена к проекту редактором диаграммы во время разработки или в коде программы. Функция умножает значения двух входных серий в каждой точке оси. По умолчанию, значение свойства `period` для этой функции равно `1`. Функция умножения может работать с любым числом входных серий (рис. 5.29). На этом рисунке `Series2*Series1=Series3`.

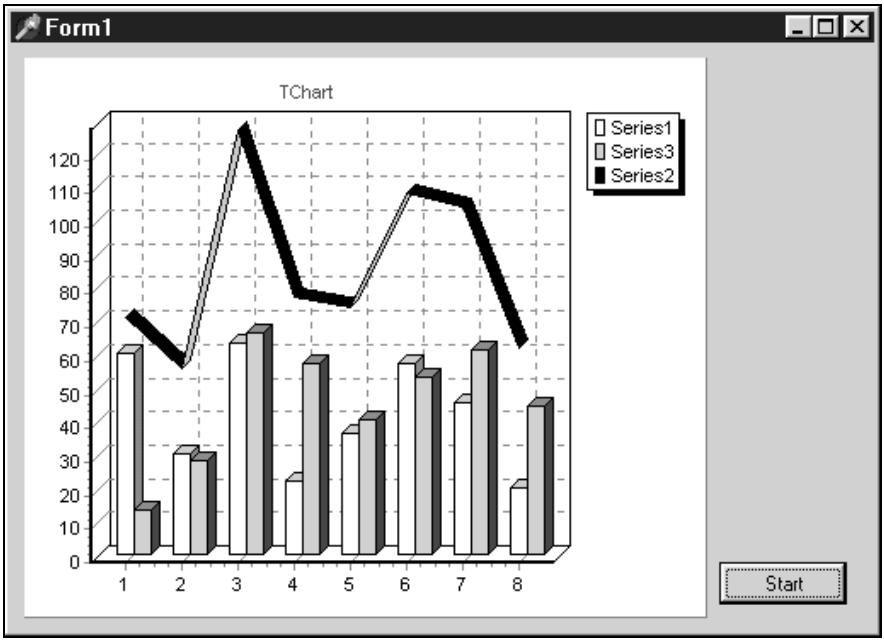


Рис. 5.28. Использование функции *Subtract*

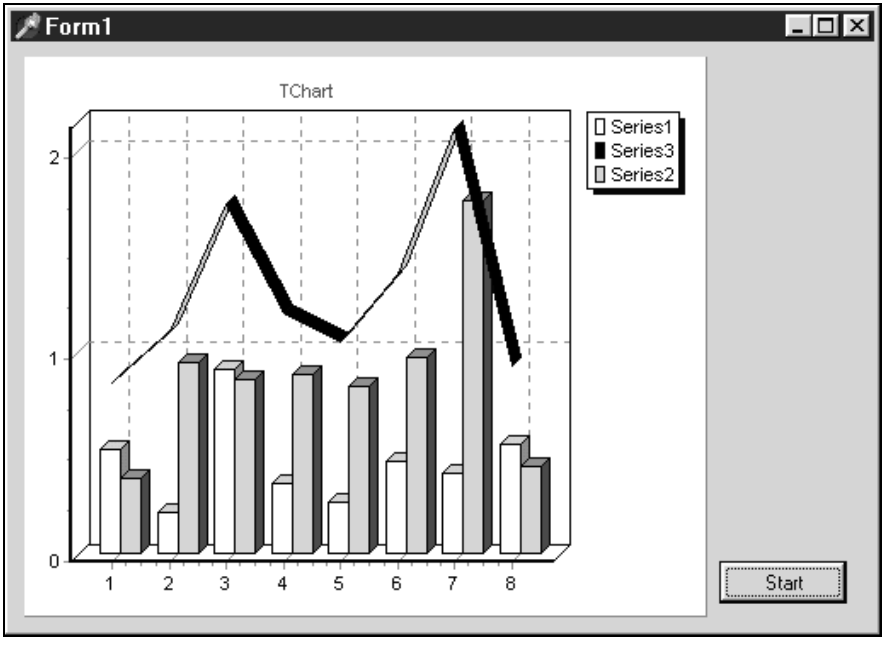


Рис. 5.29. Использование функции *Multiply*

5.7.4. Функция *TDivideTeeFunction*

Функция `TDivideTeeFunction` (деление) (рис. 5.30) требует наличия, по крайней мере, двух входных серий, причем вторая серия в списке является делителем. Если используется более двух серий, то значения данных первой будут разделены на значения данных второй серии, затем результаты будут разделены на данные третьей серии и так далее. Значение свойства `period` для этой функции равно `1`. На рис. 5.30 `Series1 / Series2=Series3`.

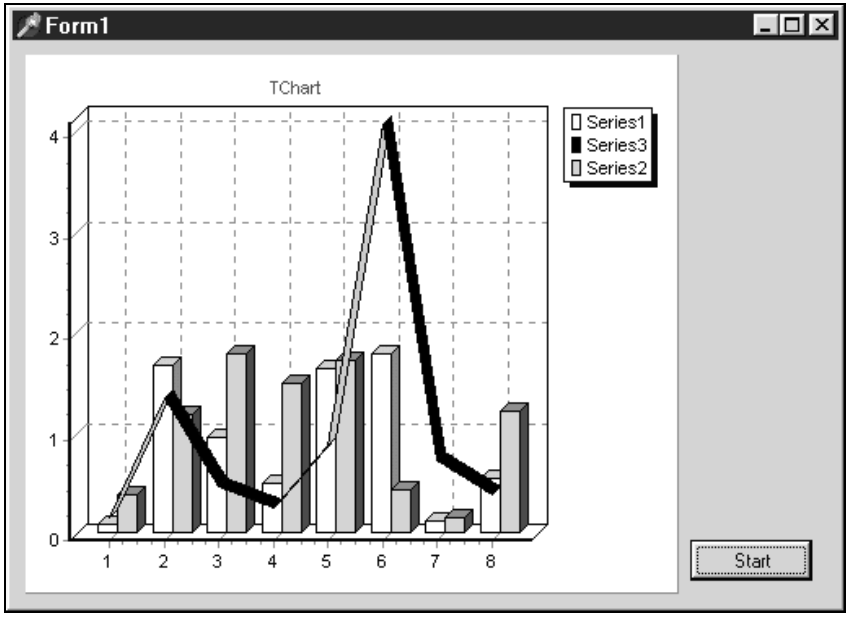


Рис. 5.30. Использование функции *Divide*

5.7.5. Функция *THighTeeFunction*

Функция `THighTeeFunction` (максимум) требует наличия нескольких входных серий и будет всегда показывать самую большую точку этих серий в каждом периоде (рис. 5.31). Если добавлено более одной серии данных в качестве источника данных для `THighTeeFunction`, тогда, по умолчанию, значение свойства `period` равно `1`. На рис. 5.31 функцию максимума содержит серия `Series3`.

5.7.6. Функция *TLowTeeFunction*

Функция `TLowTeeFunction` (минимум) требует наличия нескольких входных серий и будет всегда показывать минимальную точку этих серий в каждом периоде (рис. 5.32).

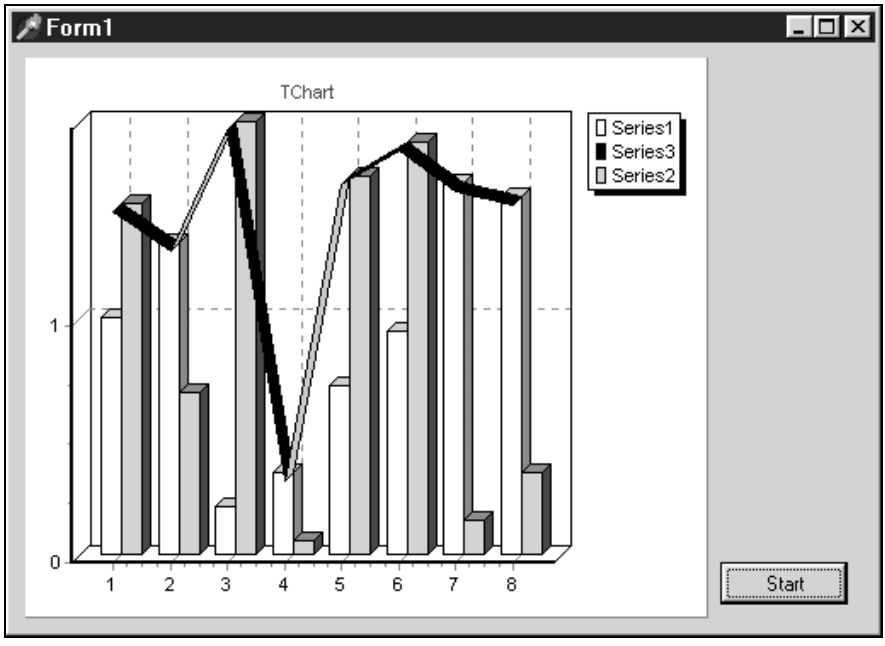


Рис. 5.31. Использование функции *High*

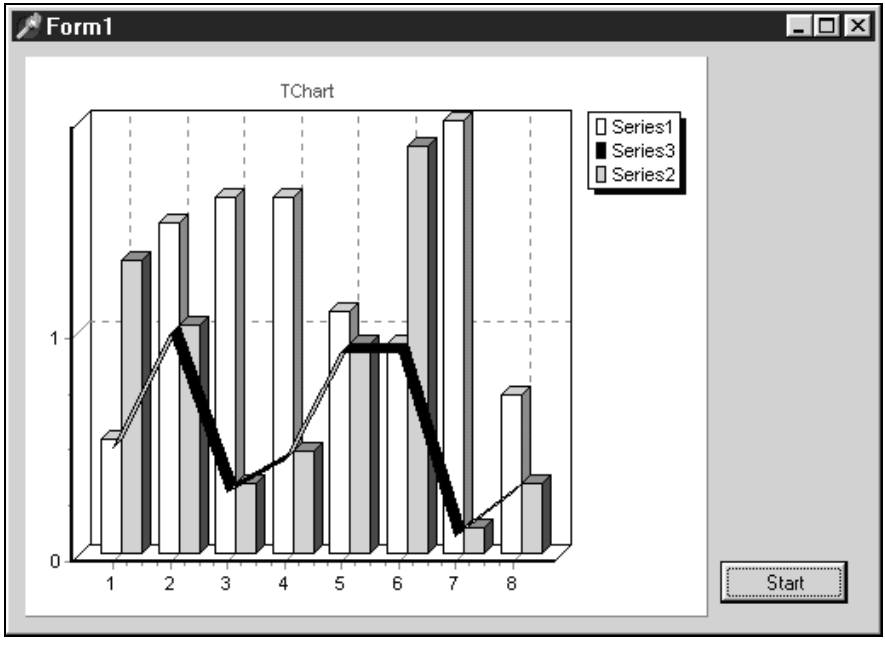


Рис. 5.32. Использование функции *Low*

Если добавлено более одной серии данных в качестве источника данных для TLowTeeFunction, тогда, по умолчанию, значение свойства period равно 1. На рис. 5.32 функцию минимума содержит Series3.

5.7.7. Функция TAverageTeeFunction

Функция TAverageTeeFunction (рис. 5.33) выполняет простое или средне взвешенное по периоду усреднение величин источника данных. Значение свойства period функции TAverageTeeFunction соответствует количеству значений, группируемых для вычисления среднего. Так, если серия имеет 1000 точек и период равен 200, то AverageTeeFunction возвратит 5 точек (так как 1000:200=5). При этом каждая величина Y будет являться простым средним значением для группы из 200 точек. На рис. 5.33 функцию усреднения содержит Series3.

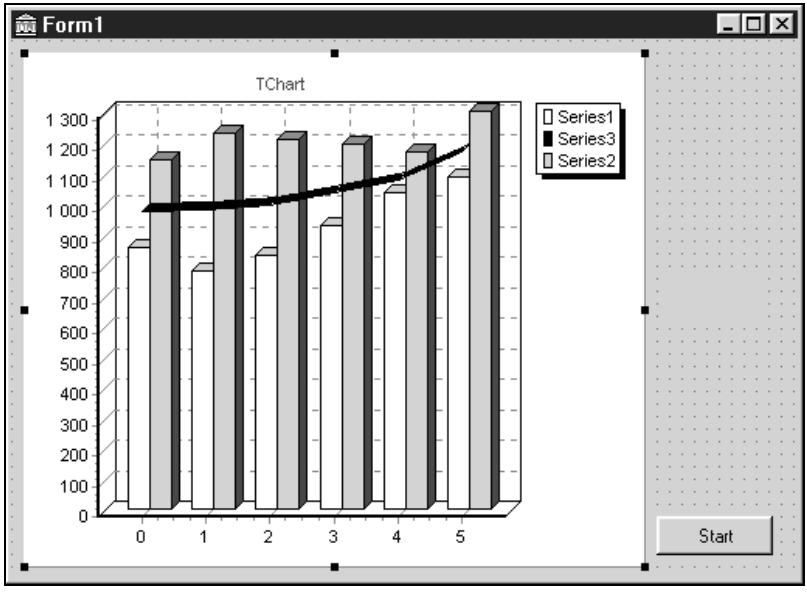


Рис. 5.33. Использование функции *Average*

5.8. Особенности разработки приложений, использующих диаграммы

5.8.1. Обработка событий нажатия кнопок

Обработчик события OnClickSeries разрешает доступ к любой активной серии на диаграмме. Для того чтобы получить доступ к обработчику события и добавить

определение процедуры, добавьте на форму компонент `Chart` и щелчком указателя мыши по нему откройте **Инспектор Объектов (Object Inspector)**. Выберите в нем вкладку **Events** и дважды щелкните на строке **OnClickSeries**. В обработчике событий поместите следующий код (листинг 5.9).

Листинг 5.9. Код обработчика события *OnClickSeries*

```
procedure TForm1.DBChart1ClickSeries(
        Sender: TCustomChart;
        Series: TChartSeries; ValueIndex: Longint; Button: TMouseButton;
        Shift: TShiftState; X, Y: Integer);
begin
        ShowMessage(' Щелчок на серии: '+Series.Name+
                ' в точке: '+IntToStr(valueindex));
end;
```

Параметр `valueindex` указывает на номер точки данных. Его можно использовать для получения доступа к величинам `X` и `Y` (см. листинг 5.10).

Листинг 5.10. Получение информации о координатах точки на диаграмме

```
procedure TForm1.DBChart1ClickSeries(
        Sender: TCustomChart;
        Series: TChartSeries; ValueIndex: Longint; Button: TMouseButton;
        Shift: TShiftState; X, Y: Integer);
begin
        ShowMessage(' Щелчок на серии: '+Series.Name+
                ' в точке: '+
                Floattostr(Series.XValue[valueindex]) + ',' +
                Floattostr(Series.YValue[valueindex]));
end;
```

Для получения той же информации можно использовать обработчик события диаграммы `OnClick` (листинг 5.11).

Листинг 5.11. Обработчик события *OnClick*

```
procedure TForm1.DBChart1Click(Sender: TObject);
var t,tmp:Longint;
        x,y:Double;
begin
        Series1.GetCursorValues(x,y);
        for t:=0 to DBChart1.SeriesCount-1 do
        begin
                tmp:=DBChart1.Series[t].GetCursorValueIndex;
```

```
                if tmp<>-1 then
                        ShowMessage(' Щелчок на серии: '
                                +DBChart1.Series[t].Name
                                +' в точке: '+IntToStr(tmp));
        end;
end;
```

Обработчик события серии `OnClick` отслеживает события нажатия на уровне серии. Таким образом, можно ограничить доступ к данным конкретной серии для нескольких диаграмм. Чтобы получить доступ к событию серии `OnClick`, надо выбрать серию в инспекторе объектов, перейти на вкладку **Events** и дважды щелкнуть на строке **OnClick**. В обработчике событий поместите следующий код (листинг 5.12).

Листинг 5.12. Доступ к событию серии *OnClick*

```
procedure TForm1.Series1Click(Sender: TChartSeries;
        ValueIndex: Longint;
        Button: TMouseButton; Shift: TShiftState;
        X, Y: Integer);
begin
        ShowMessage(' Щелчек на серии: '+Sender.Name
                +' at point: '+inttostr(valueindex));
end;
```

5.8.2. Рисование на диаграмме

Для добавления фрагмента текста или стандартных фигур на диаграмму лучше всего подходит серия `Shape`. Кроме того, можно создавать свои надписи и фигуры на диаграмме.

Координаты точек `Point` выражаются в единицах измерения, определенных пользователем. Оси (`Axis`) диаграммы ответственны за преобразование координат точки в экранные координаты `X` и `Y`. Как отдельные элементы компонентов (например, `Axis`), так и компоненты `Series` в целом предоставляют много различных функций для таких преобразований.

> ***Примечание***
>
> Использование функций преобразования координат возможно только после того, как компонент диаграммы изображен на экране или на своей скрытой канве `Canvas`.

С помощью функции `CalcPosValue` можно вычислить экранную позицию для точки любой оси:

```
Var MyPos : Longint;
MyPos := Chart1.LeftAxis.CalcPosValue(100.0);
```

После выполнения кода переменная `MyPos` будет содержать экранную позицию для значения `100, 0` на левой вертикальной оси `Chart1`.

Если необходимо выполнить обратное преобразование экранных пиксельных координат в величины оси, можно использовать обратную функцию `CalcPosPoint`:

```
Var MyValue : Double ;
MyValue := Chart1.LeftAxis.CalcPosPoint( 100 ); .
```

После выполнения кода, переменная `MyValue` будет содержать величину координаты для значения `100, 0` на левой вертикальной оси `Chart1`.

> ***ПРИМЕЧАНИЕ***
>
> Отсчет экранных позиций пикселов начинается с точки начала диаграммы `0,0`. Однако, рисуя в метафайлах, на пользовательских холстах или при выводе на печать, надо быть готовым к тому, что точка начала диаграммы может отличаться от `0,0`.

Свойство `Chart1.ChartBounds` возвращает начало и окончание координат для диаграммы в соответствующий прямоугольник.

Элемент компонента `Axis` предлагает функцию `CalcSizeValue` для вычисления экранного пространства данной оси:

```
Var  Space : Longint ;
Space := Chart1.LeftAxis.CalcSizeValue( 1000 );
```

Элемент компонента `Axis` может иметь тип `DateTime`, и можно, например, преобразовать период `Date Range` в пикселы:

```
Var  Space : Longint ;
Space := Chart1.BottomAxis.CalcSizeValue(
        EncodeDate( 1998,12,31) - EncodeDate( 1998,1,1) );
```

При рисовании на диаграмме также используются функции `CalcYPos` и `CalcXPos`, которые возвращают соответственно вертикальную и горизонтальную координаты аргумента в пикселах. Пользуясь при рисовании координатами `XPos` и `YPos`, помните, что координата `0,0` соответствует левой верхней точке прямоугольника диаграммы `ChartRect`.

В следующем примере (листинг 5.13) рисуется линия из средней точки интервала оси `Y` через диаграмму.

Листинг 5.13. Рисование линии на диаграмме

```
procedure TForm1.Button1Click(Sender: TObject);
var        MyHalfwayPoint,YPosition: longint;
Begin
       With Series1.YValues do
              MyHalfwayPoint:=round(((MaxValue-MinValue)* 0.5) +
                     MinValue);
```

```
        { затем вычисляет экранные пиксельные координаты }
        YPosition:=
                DBChart1.LeftAxis.CalcYPosValue(MyHalfwayPoint);
        With DBChart1.Canvas do
        begin
                Pen.Width:=3;
                Pen.Style:=psSolid;
                Pen.Color:=clBlack;
                with DBChart1 do
                begin
                        MoveTo(ChartRect.Left,YPosition);
                        LineTo(ChartRect.Left+Width3D,
                        YPosition-Height3D);
                        LineTo(ChartRect.Right+Width3D,
                                YPosition-Height3D);
                end;
        end;
end;
```

Серия имеет аналогичные методы для преобразования координат.

Следующий код вычисляет позицию на экране, в которой находится точка `Series1` со значением `1000`:

```
Var MyPos : Longint ;
MyPos := Series1.CalcPosValue( 1000 );
```

То же вычисление для первой точки `Series1`:

```
MyYPos := Series1.CalcPosValue( Series1.YValue[ 0 ] );
```

Можно вычислить как координату `X`, так и координату `Y` точки, принадлежащей `Series1`:

```
MyXPos := Series1.CalcXPos(EncodeDate(1997,12,31) );
```

То же вычисление для последней точки `Series1`:

```
MyXPos := Series1.CalcXPos( Series1.XValues.Last );
```

Для преобразования из экранных пикселов можно использовать функции `XScreenToValue` и `YscreenToValue`.

```
Var MyValue : Double ;
MyValue := Series1.YScreenToValue( Y ) ;
```

С помощью функции `Series.Clicked` можно запросить значение точки, соответствующей паре экранных координат `X` и `Y`.

Свойство `Canvas` диаграммы соответствует стандартному свойству `Canvas`, которое присутствует в других компонентах VCL. Используя его, можно управлять видом диаграммы.

В следующем примере (листинг 5.14) область фона прямоугольника диаграммы делится на 5 равных сегментов, которые окрашиваются согласно заданному массиву цветов.

Листинг 5.14. Окраска фона диаграммы

```
procedure TDrawForm.LineSeries1BeforeDrawValues(Sender:
        TObject);
Const MyColors:array[1..5] of TColor=
        ( clNavy,
                clGreen,
                clYellow,
                clRed,
                $00000080 { красный }
        );
var t,partial:Longint; tmpRect:TRect;
begin
        With Chart1 do
        Begin
                { разделим общую ширину диаграммы на 5 }
                tmpRect:=ChartRect;
                tmpRect.Right:=tmpRect.Left;
                partial:=ChartWidth div 5;
                Canvas.Brush.Style:=bsDiagCross;
                Canvas.Pen.Style:=psClear;
                { для каждого раздела, заполните своим цветом }
                for t:=1 to 5 do
                Begin
                        { назначить измерение прямоугольника }
                        tmpRect.Right:=tmpRect.Right+partial+1 ;
                        Canvas.Brush.Color:=MyColors[t];
                        { красить !!! }
                        With tmpRect do
                             Canvas.Rectangle( Left+Width3D,Top-Height3D,
                                       Right+Width3D,Bottom-Height3D );
                        tmpRect.Left:=tmpRect.Right;
                end;
        end;
end;
```

Компоненты `TChart` имеют внутренний объект битового изображения, который используется при рисовании как скрытый "буфер". Когда рисование завершается, этот "буфер" копируется в экранную видеопамять.

Примечание

Свойство `BufferedDisplay` изменяется, если использовано внутреннее битовое изображение. При использовании битового изображения не происходит мерцания при перерисовке диаграммы в реальном времени. Для некоторых диаграмм с большим количеством точек можно воспользоваться преимуществом большой скорости рисования в битовых изображениях вместо прямого рисования в медленных чипах видеопамяти.

Рисование на Canvas битового изображения с точки зрения результатов эквивалентно рисованию на "реальной" Canvas.

> ***Примечание***
>
> Битовое изображение нельзя использовать при рисовании в метафайле или при печати.

Для вставки пользовательских элементов в диаграмму лучше всего использовать обработчик события Chart1.OnAfterDraw, вызываемый перед перерисовкой компонента диаграммы.

Можно вставить пользовательские элементы выше сетки диаграммы и ниже серии диаграммы, помещая свой код в обработчике события Series.OnBeforeDrawValues.

Свойство ChartBounds соответствует наибольшему прямоугольнику, который может быть описан вокруг диаграммы. Свойства ChartWidth, ChartHeight, ChartXCenter и ChartYCenter согласованы со свойством ChartBounds. Оси рисуются в этом пространстве. Свойство ChartRect возвращает соответствующий прямоугольник для осей (в случае плоскостных и 3D-диаграмм).

Условные обозначения Legend диаграммы обладают свойством RectLegend, которое определяет границы прямоугольника Legend.

Аналогично, верхний и нижний заголовки диаграммы Title и Foot обладают свойством TitleRect.

На трехмерных диаграммах для каждой серии назначен специальный параметр (ось Z), характеризующий глубину диаграммы (ее третье измерение). Свойства Width3D и Height3D — измерения "глубины" трехмерной (3D) диаграммы в пикселах. Свойства SeriesWidth3D и SeriesHeight3D — измерения "глубины" для каждой серии на диаграмме.

В следующем примере (листинг 5.15) рисуется горизонтальная линия в середине диаграммы Chart1.

Листинг 5.15. Рисование линии на диаграмме

```
procedure TForm1.Chart1AfterDraw(Sender: TObject);
begin
        With Chart1 do
        begin
                Canvas.Pen.Color:=clYellow;
                Canvas.MoveTo( ChartBounds.Left,ChartYCenter );
                Canvas.LineTo( ChartBounds.Right,ChartYCenter);
        end;
end;
```

В примере листинга 5.16 реализовано рисование линий в окрестности каждой точки Series1.

Листинг 5.16. Рисование линий в окрестности каждой точки серии

```
procedure TForm1.Series1AfterDrawValues(Sender: TObject);
var t,x,y : Longint ;
begin
        for t := 0 to Series1.Count - 1 do
        begin
                x:=Series1.CalcXPos( t );
                y:=Series1.CalcYPos( t );
                Chart1.Canvas.MoveTo( x-8, y-8 );
                Chart1.Canvas.LineTo( x+8, y+8 );
        end;
end;
```

ЗАМЕЧАНИЕ

Для установки одинаковых шрифтовых размеров на экране и на принтере или в метафайлах, вместо использования `Font.Size` устанавливайте значение свойства `Chart1.Canvas.Font.Height` равным отрицательной величине.

5.8.3. Работа с осями

Компоненты диаграммы имеют пять осей: `LeftAxis`, `RightAxis`, `TopAxis`, `BottomAxis` и `DepthAxis`. Каждая ось является наследником класса `TChartAxis`.

Ось отвечает за перемещение области рисования и изменение масштаба. При включении новой серии или при добавлении новых точек к серии, ось пересчитывает минимальные и максимальные величины.

Можно отключить автоматический пересчет, устанавливая свойство `Automatic` равным `false`:

```
Chart1.LeftAxis.Automatic := False; .
```

Как для минимальных, так и максимальных величин оси может быть установлено или отключено свойство автоматического пересчета.

```
Chart1.LeftAxis.AutomaticMaximum := False ;
Chart1.LeftAxis.AutomaticMinimum := True ;
```

Можно изменить свойства `Minimum` и `Maximum` непосредственно в коде программы.

```
With Chart1.LeftAxis do
begin
        Automatic := False ;
        Minimum   := 0 ;
        Maximum   := 10000 ;
end;
```

Ту же операцию можно выполнить, используя метод `SetMinMax`.

```
Chart1.LeftAxis.SetMinMax( 0, 10000 ); .
```

Примечание

Ось содержит значения типа `DateTime`, если для соответствующих компонентов серии свойства `XValues.DateTime` или `YValues.DateTime` равны `true`. Для оси свойство `DateTime` не доступно.

Изменение свойств `Minimum` и `Maximum` оси, содержащей значения типа `DateTime`, выполняется так, как показано в следующем примере.

```
With Chart1.LeftAxis do
begin
        Automatic := False ;
        Minimum:=  EncodeDate( 1990, 3,16 );
        Maximum:=  EncodeDate( 1996, 5, 24 );
end;
```

Ось может быть логарифмической, если минимум и максимум диапазона изменения переменной имеют значения больше нуля. Это единственное различие между линейными и логарифмическими осями.

Примечание

Метки (`Labels`) оси не отображаются на логарифмических приращениях.

Ось может быть инвертирована. Чтобы инвертировать ось, нужно установить значение свойства `Inverted:=true`.

Ось может отображаться различными способами, с метками или без них, с сеткой или без нее. Можно устанавливать все параметры форматирования, например, цвет, вид шрифта и стиль пера. Свойство `Increment` (приращения) оси управляет числом линий сетки и меток и расстоянием между ними. По умолчанию свойство `Increment` равно нулю, и ось автоматически вычисляет приращение для меток. Значение свойства `Increment` возрастает автоматически, пока все метки не смогут быть отображены. Если это нежелательно, установите свойство оси `LabelsSeparation` равным нулю:

```
Chart1.LeftAxis.LabelsSeparation :=  0;
```

Примечание

При `LabelsSeparation=0` не выполняется проверка размера меток, и они могут перекрывать друг друга.

Следующий код устанавливает вертикальное приращение оси на `30`:

```
Series1.Clear;
Series1.AddArray( [ 20, 50, 120 ] );
Chart1.LeftAxis.Increment:= 30;
```

По умолчанию, первая метка выводится при нулевом приращении. Используйте встроенный массив констант `DateTimeStep` для того, чтобы определить приращение оси, содержащей значения типа `DateTime`:

```
Chart1.BottomAxis.Increment:=DateTimeStep[dtOneMonth];
```

Установите свойство `ExactDateTime` равным `true`, если необходимо, чтобы метки располагались на точных временных границах, например, на первом дне каждого месяца.

Линии сетки оси отображаются в каждой позиции приращения или в каждой позиции метки оси. Свойство `TickOnLabelsOnly` управляет этой характеристикой:

```
Chart1.BottomAxis.TickOnLabelsOnly := False ;
```

Существуют различные способы нанесения меток на оси, которые определяются свойством `LabelStyle`. Возможные значения этого свойства приведены в табл. 5.5.

Таблица 5.5. *Возможные значения свойства LabelStyle*

Значение	Комментарий
`talValue`	этикетки не отображают единицы измерения оси
`talMark`	маркеры
`talText`	серия меток `XLabels`
`talNone`	нет меток
`talAuto`	стиль вычисляется автоматически

Когда значение свойства `LabelStyle = talText`, компонент `TeeChart` установит автоматически значение `talValue`, если серия не имеет `XLabels`.

Для стилей `talMark` и `talText` метки оси отображаются в последовательных позициях точки, не используя свойство приращения оси.

Можно программировать текст меток, используя обработчик события `OnGetAxisLabel` (листинг 5.17).

Листинг 5.17. Программное изменение текста меток

```
procedure TForm1.Chart1GetAxisLabel(Sender: TChartAxis;
        Series: TChartSeries; ValueIndex: Integer;
        var LabelText: String);
begin
        if Sender=Chart1.LeftAxis then
        if ValueIndex < 2 then LabelText :='' ;
end;
```

Метод `CustomDraw` добавляет на диаграмму новую ось, как копию существующей оси в новой позиции.

В следующем примере (листинг 5.18) серия заполняется линиями с произвольными данными, и создаются две дополнительные оси. Три настраиваемых позиции `PosLabels`, `PosTitle` и `PosAxis` используются для того, чтобы установить положение, соответственно, для меток, названий и осей. Логический параметр `GridVisible` определяет видимость сетки оси.

Листинг 5.18. Добавление осей

```
procedure TForm1.FormCreate(Sender: TObject);
var t:integer;
begin
        Series1.XValues.DateTime := False;
        Chart1.BottomAxis.Increment:=0;
        For t:= -10 to 10 do
                Series1.addXY(t,Random(100),'', clTeeColor);
end;

procedure TForm1.Series1AfterDrawValues(Sender:
        TObject);
var posaxis,percent:longint;
begin
        percent:=50;
        With DBChart1 do
        begin
                PosAxis:=ChartRect.Left+
                        Trunc(ChartWidth*Percent/100.0);
                LeftAxis.CustomDraw(posaxis-10,posaxis-40,
                        posaxis,True);
                PosAxis:=ChartRect.Top+
                        Trunc(ChartHeight*Percent/100.0);
                BottomAxis.CustomDraw(posaxis+10,posaxis+40,
                        posaxis,True);
        end;
end;
```

5.8.4. Действия с сериями

Подобно любому другому компоненту в Delphi, серия может быть создана в коде программы. Для этого необходимо выполнить следующие действия:

- сначала определим переменную серии: `Var MySeries : TbarSeries`;
- затем создадим компонент: `MySeries := TBarSeries.Create( Self )`;
- привяжем его к компоненту диаграммы: `MySeries.ParentChart := Chart1`.

После выполнения этих действий можно, указывая на `MySeries`, выполнять все то, что можно делать во время проектирования.

Если доступ к `MySeries` не нужен, то код сводится к одной строке текста программы: `Chart1.AddSeries( TBarSeries.Create( Self)).`

Перед созданием серии можно объявить свой класс серии:

- объявим переменную класса серии: `Var MyClass : TchartSeriesClass;`
- установим желаемый тип серии: `MyClass := TbarSeries;`
- создадим компонент серии: `Chart1.AddSeries( MyClass.Create( Self ) ).`

Компоненты диаграммы хранят все серии в свойстве `SeriesList: TList`. К этому свойству разрешен доступ только для чтения тремя эквивалентными способами:

- использование свойства `SeriesList: MySeries := Chart1.SeriesList [ 0 ];`
- использование свойства серии: `MySeries := Chart1.Series [0];`
- использование встроенного свойства `Chart1: MySeries := Chart1 [ 0 ].`

Свойство `Chart1.SeriesCount` возвращает номер серии в `SeriesList`. Можно использовать `SeriesCount` для того, чтобы просмотреть все серии диаграммы (листинг 5.19).

Листинг 5.19. Доступ ко всем сериям диаграммы

```
for t := 0 to Chart1.SeriesCount - 1 do
With Chart1.Series [ t ] do
begin
        SeriesColor := clBlue ;
end;
```

Серия может удаляться тремя способами:

- установка свойства `Active` для серии: `Series1.Active := False;`
- удаление серии с родительской диаграммы: `Series1.ParentChart := nil;`
- удаление серии полностью: `Series1.Free.`

В трехмерном случае (`Chart1.View3D=true`), всем сериям назначен порядок вывода по оси `Z`. Можно управлять порядком создания серий, используя методы:

- `Chart1.ExchangeSeries( 0, 1 );`
- `Chart1.SeriesUp( Series1 );`
- `Chart1.SeriesDown( Series1 ).`

Свойство целого типа `Series.ZOrder` возвращает соответствующую позицию `Z` для серии.

Каждый тип серии имеет, по крайней мере, две величины для каждой точки: координаты `X` и `Y`. Эти величины определены как переменные с плавающей запятой типа `Double`. Для расширенных серий, типа `BubbleSeries`, каждая точка описывается свойствами `X`, `Y` и радиусом.

Каждый тип серии обладает своим методом для добавления точек, хотя наиболее общие серии, типа Line, Bar, Point и Area, имеют общий метод AddY для добавления точек.

В следующем примере (листинг 5.20) к пустой TPieSeries добавляются две точки.

Листинг 5.20. Добавление двух точек к серии *TPieSeries*

```
Series1.Clear ;
Series1.Add(  1234, 'Name 1', clBlue );
Series1.Add(  2001, 'Name 2', clRed );
```

Метод серии AddArray может использоваться для добавления нескольких значений:

```
Series3.Clear;
Series3.AddArray([ 1234, 2001, 4567.12 ] );
```

Метод AddArray не очищает серию перед добавлением точек.

Можно задать точки, как константы или переменные:

```
Series3.AddArray([Table1Sales.AsFloat,123,MyTotal]);
```

При некоторых обстоятельствах не известны значения величин для добавляемых точек. Можно добавлять эти точки со значениями zero или как величины null. Недействительные величины null не отобразятся, в то время как величины zero будут отображены обычным способом.

Следующий код (листинг 5.21) добавляет различные точки, в том числе и недействительную точку.

Листинг 5.21. Добавление недействительной точки

```
With Series1 do
begin
        Clear ;
        AddY( 10, 'John', clGreen );
        AddY( 20, 'Anne', clYellow );
        AddY( 15, 'Thomas', clRed );
        AddNull( 'Peter' );
        AddY( 25, 'Tony', clBlue );
        AddY( 20, 'Mike', clLime );
end;
```

Каждый тип серии оперирует с недействительными величинами по разному: Bar, HorizBar, Line, Area и Point, например, не отображают недействительные точки, а серия PieSeries отображает недействительные величины как zero.

Точки могут быть отсортированы по координатам X или Y. Свойства Series.XValues.Order и Series.YValues.Order управляют сортировкой точек:

```
Series1.XValues.Order := loAscending;
```

Возможные значения свойств сортировки: `loNone`, `loAscending` (возрастание) или `loDescending` (убывание). По умолчанию, свойство `XValues.Order` устанавливается равным `loAscending`, `YValues.Order=loNone`, что означает упорядочивание новых точек по координатам `X`.

Порядок сортировки должен быть установлен прежде, чем точки будут добавлены к серии. Впрочем, можно изменить свойство `Order`, вызвав затем метод `Sort`.

Рассмотрим пример такого изменения свойства `Order`:

- поставим компонент `TChart` на форму, добавим серию `Line`;
- назначим код для события `Button1Click: Series1.AddArray([5,2,3,9]);`
- поставим еще одну кнопку `TButton` и добавим этот код для `Button2Click`;

```
With Series1 do
begin
        YValues.Order:=loAscending;
        YValues.Sort;
        Repaint;
end;
```

- теперь запустим приложение на выполнение, нажмем кнопку `Button1` для заполнения `Series1` и нажмем кнопку `Button2` для сортировки по оси `Y`.

Если координаты `X` не используются, то потребуется еще одна строка текста программы:

- поставим `Button3` и добавим код в `Button3Click`:

```
Series1.XValues.FillSequence;
Series1.Repaint;
```

> ***Примечание***
>
> У диаграммы `Bar` могут возникать сложности при интерпретации координат `X`. Серия `Pie` не использует координат `X`.

Для добавления точек с указанием координат `X` и `Y` можно использовать метод `AddXY`.

```
With Series1 do
begin
Clear ;
        AddXY(  10, 10, 'Barcelona', clBlue );
        AddXY(  1, 10, 'San Francisco', clRed );
end;
```

> ***Примечание***
>
> При использовании серии `Bubble` добавление точек реализуется методом `AddBubble`.

Для удаления точки из серии надо вызвать метод `Series.Delete`, указав индекс точки как аргумент. Индексы точек начинаются с нуля:

```
Series1.Delete(0);          { удаление первой точки }
Series1.Delete(Series1.Count-1); {удаление последней}
```

При попытке удалить несуществующую точку, возникает исключительная ситуация `List out of bounds` , так что всегда проверяйте до удаления, есть ли точка в серии: `if Series1.Count > MyIndex then Series1.Delete( MyIndex )`.

Для извлечения или изменения координат существующих точек могут быть использованы свойства `XValues` и `YValues`.

```
Var MyValue: Double;
MyValue:=Series1.YValues[0];
```

Расширенные серии имеют дополнительные свойства массива, например, `BubbleSeries.RadiusValues`. Можно получить доступ к этим свойствам так же, как к `XValues` или к `YValues`:

```
if BubbleSeries1.RadiusValues[ Index ] > 100 then ...
```

Модификация величин точки может быть выполнена, используя вышеуказанные свойства:

```
Series1.YValues[ 0 ] := Series1.YValues[ 0 ] + 1 ;
Series1.RefreshSeries ;
```

Свойства `XValues` и `YValues` используют функцию `Locate`, которая ищет заданную величину в серии, и если обнаруживает ее, то возвращает индекс величины (листинг 5.22).

Листинг 5.22. Поиск заданной величины в серии

```
Var MyIndex : Integer ;
MyIndex := Series1.YValues.Locate( 123 );
if MyIndex = -1 then
        ShowMessage(' 123 нет в Series1 !' )
else
        ShowMessage(' 123 есть и ее номер = '+
                IntToStr( MyIndex+1 ) );
```

Свойства `Xvalues`, `YValues` поддерживают следующие статистические свойства:

- `Total` — сумма всех величин в списке;
- `TotalABS` — сумма всех абсолютных величин;
- `MaxValue` — максимальная величина в списке;
- `MinValue` — минимальная величина в списке.

При вызове метода `RecalcMinMax` свойства `MinValue` и `MaxValue` изменяются автоматически. Свойства `Total` и `TotalABS` также обновляются автоматически. Эти вели-

чины используются для ускорения различных масштабирующих вычислений и процентных вычислений.

Все серии поддерживает внутренний список цветов для каждой точки в серии. Доступ к этому списку реализуется через свойство массива `ValueColor`:

```
Var MyColor : TColor ;
MyColor := Series1.ValueColor[ 0 ] ;
Series1.ValueColor[ 1 ] := clBlue ; .
```

Компонент `TeeChart` определяет общую константу с именем `clTeeColor`. Точки с цветом `clTeeColor` будут отображены, используя цвет `SeriesColor`.

Каждая точка имеет связанный текст, называемый `XLabel`, который объявлен как тип `string`. Метки точки используются на метках оси, условных обозначениях диаграммы и маркерах точки.

5.8.5. Изменение масштаба изображения

При перемещении диаграммы и изменении ее масштаба устанавливаются новые диапазоны осей. После изменения масштаба или прокрутки диаграммы все серии перерисуют свои точки в новых позициях.

> ***Примечание***
>
> Серия `Pie` не будет перемещаться или выводиться с измененным масштабом. Можно управлять измерениями `Pie`, используя поля диаграммы или свойство радиуса.

Можно изменять масштаб диаграммы программными способами или трансформируя ее указателем мыши. Свойство диаграммы `AllowZoom` изменяется, если пользователю разрешено изменение масштаба: `Chart1.AllowZoom := true`.

Пользователи могут изменить масштаб изображения, выделяя указателем мыши прямоугольник вокруг области диаграммы, которую они хотят увидеть более подробно.

Можно задать кнопку мыши или "горячие" клавиши для задания прямоугольной области диаграммы с измененным масштабом. Следующий код использует глобальные переменные `TeeZoomMouseButton, TeeZoomKeyShift`.

```
TeeZoomMouseButton   := mbLeft ;
{для изменения масштаба используется левая кнопка мыши}
TeeZoomKeyShift      := [ ssShift ] ;
{должен нажиматься SHIFT для начала изменение масштаба}
```

Как только пользователи отпустят кнопку мыши, компонент `TeeChart` перерисовывается с измененным масштабом.

Можно изменять масштаб единовременно или выполнять изменение пошагово с помощью свойства `steps`. Использование этого свойства делает эффект изменения масштаба "анимационным", что помогает лучше идентифицировать область с измененным масштабом. Следующий код активизирует "оживленное" изменение масштаба:

```
Chart1.AnimatedZoom:= true.
```

Значение свойства `AnimatedZoomSteps` устанавливается равным желаемому числу промежуточных изменений масштаба.

```
Chart1.AnimatedZoomSteps:= 5 ;
```

Можно управлять числом шагов изменения масштаба, используя глобальную переменную `AnimatedZoomFactor`.

```
AnimatedZoomFactor:=2.0;
```

Открыть или закрыть диаграмму можно, используя любой из следующих методов:

- `ZoomRect` — устанавливает диапазоны осей для показа области параметра `TRect`. Прямоугольник выражается в экранных пикселах.

  ```
  Chart1.ZoomRect( Rect( 100, 100, 200, 200 ) );
  ```

- `ZoomPercent` — процентное изменение масштаба диаграммы. Например, для изменения масштаба на 110 % служит код:

  ```
  Chart1.ZoomPercent( 110 );
  ```

Метод `UndoZoom` восстанавливает диапазоны осей к их автоматическим минимальным и максимальным величинам:

```
Chart1.UndoZoom;
```

Обработчик события `OnZoom` запускается всякий раз, когда происходит изменение масштаба на диаграмме (листинг 5.23).

Листинг 5.23. Обработчик события изменения масштаба

```
procedure TForm1.Chart1Zoom(Sender: TObject);
begin
        Button1.Visible:=True ;
        { делает видимой кнопку }
end;
```

Обработчик события `OnUndoZoom` вызывается при восстановлении изменения масштаба кодом или поворотом колесика мыши.

Прокрутка диаграммы аналогична изменению масштаба. Диапазоны осей увеличиваются или уменьшаются и компонент диаграммы перерисовывается для того, чтобы показать точки в новых позициях.

Свойство `AllowPanning` изменяется, если пользователи перемещают содержимое диаграммы указателем мыши. Возможные значения этого свойства представлены в таблице 5.6.

Таблица 5.6. *Возможные значения свойства AllowPanning*

Значение	Допустимые перемещения
pmNone	Перемещение не допускается
pmHorizontal, pmVertical	Допускать перемещение только в направлениях Horizontal, Vertical
pmBoth	Допускать перемещение по всем направлениями

Пример кода программы с использованием свойства AllowPanning:

```
Chart1.AllowPanning:=pmNone;
{Перемещение не допускается} .
```

Можно назначить кнопку мыши и "горячую" клавишу клавиатуры для начала прокрутки.

```
TeeScrollMouseButton := mbRight;
{ кнопка для перемещения }
TeeScrollKeyShift    : [ ssCtrl ];
{ CONTROL должна нажиматься для начала перемещения }
```

Можно переместить диаграмму программным способом, используя метод оси Scroll.

```
Procedure Scroll(Const Offset: Double; CheckLimits: Boolean);
```

Код Chart1.BottomAxis.Scroll(1000, True) увеличивает диапазон BottomAxis на 1000, что аналогично изменению горизонтального диапазона на 1000:

```
With Chart1.BottomAxis do
   SetMinMax( Minimum+1000, Maximum+1000 );
```

и установке свойства BottomAxis.Automatic=false. Диаграмма изменится — горизонтальная нижняя ось будет "сдвинута" влево на 1000.

Логический параметр CheckLimits разрешает оси перемещаться только в том случае, если есть точки серии в направлении, указанном прокруткой.

Обработчик события диаграммы OnScroll вызывается при перемещении диаграммы вручную (листинг 5.24).

Листинг 5.24. Обработчик события прокрутки

```
procedure TForm1.Chart1Scroll(Sender: TObject);
begin
        Label1.Caption := 'This Chart has scrolled ! ' ;
end;
```

Обработчик события OnAllowScroll может использоваться для того, чтобы программно принять запланированное перемещение или отказаться от него (листинг 5.25).

Листинг 5.25. Обработчик события отказа от прокрутки

```
procedure TForm1.Chart1AllowScroll(Sender: TChartAxis; var AMin, AMax: Double;
var AllowScroll: Boolean);
begin
        if Sender = Chart1.BottomAxis then
                if AMax > 1000 then AllowScroll := False ;
end;
```

В следующем примере устанавливается запрет на перемещение при попытке установить величину максимума оси более чем `1000`. Та же проверка может выполняться для оси `DateTime`:

```
if Sender = Chart1.BottomAxis then
        if AMax > EncodeDate( 1997, 12, 31 ) then
                AllowScroll := False ;
```

Для организации перемещения диаграммы при нажатии клавиш можно использовать обработчик события формы `OnKeyDown`. Сначала установим свойство формы `KeyPreview` равным `true`. В обработчике события `KeyDown`, в зависимости от нажатой клавиши перемещения курсора, используется метод оси `Scroll`.

5.8.6. Особенности разработки проектов, работающих в реальном масштабе времени

Для ускорения работы с диаграммами в реальном масштабе времени надо выполнять два правила:

- по возможности, лучше рисовать диаграммы с небольшим количеством точек;
- необходимо использовать самые быстрые аппаратные средства.

Вместе эти два правила действительно сильно ускоряют создание диаграммы при многократном рисовании.

Некоторые другие предложения:

- использование плоских диаграмм предпочтительнее. Трехмерные диаграммы рисуются медленнее, чем плоские;
- по возможности, лучше создавать диаграммы небольших размеров. Диаграммы больших размеров используют большее количество пикселов;
- если это возможно, удалите условные обозначения и названия;
- предпочтительнее использование шрифтов и размеров шрифта, установленных по умолчанию;
- используйте свойство `FastLineSeries` для вычерчивания большого числа точек;
- используйте стиль `solid` (твердость) для пера и кисти;
- избегайте использования круглых фигур или круглых стилей `Bar` ;

- не используйте битовые карты или градиентные заливки для фона;
- установите диаграмму `BevelInner` и свойства `BevelOuter` в значение `bvNone`;
- если возможно, устанавливайте значение свойства `Chart1.AxisVisible` равным `false`, чтобы удалить оси;
- установите оптимальные величины видеоразрешения, согласно имеющейся видеокарте. На средних видеокартах в режиме 800×600×256 рисование может выполняться быстрее, чем в режиме 1024×768×32k;
- используйте Windows 95 или Windows NT с ускоренными драйверами для вашей видеокарты.

Глобальное свойство `BufferedDisplay` непосредственно управляет рисованием диаграммы. Если `BufferedDisplay=true`, то диаграммы рисуются на внутренней битовой карте. После завершения рисования, эта внутренняя битовая карта передается на экран за один прием. В некоторых случаях, устанавливая `BufferedDisplay=false`, можно ускорить перерисовку диаграммы. Скорость зависит от типа процессора, от скорости видеокарты и от количества точек диаграммы.

5.9. Проект с использованием диаграмм

Рассмотрим комплексный проект "Компонент *TChart*", который позволяет создавать различные типы диаграмм (в том числе, трехмерные), вносить в них данные, изменять свойства диаграмм.

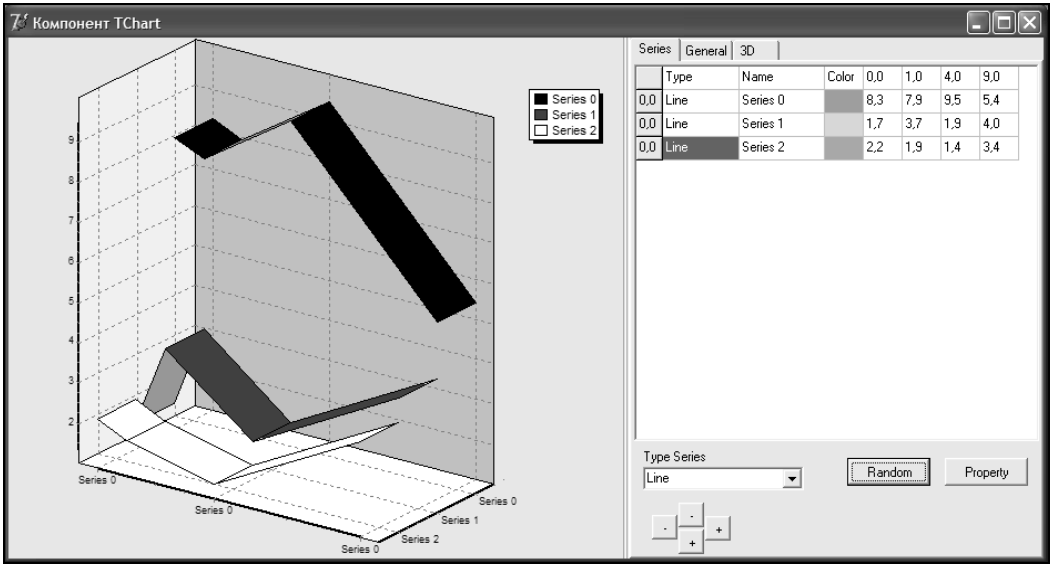


Рис. 5.34. Главная форма проекта "Компонент *TChart*"

Проект приведен на компакт-диске в папке **Примеры | Глава 5 | Chart**. Данный проект состоит из главной формы `FormMain` (с внешним именем **Компонент TChart**) (рис. 5.34), формы свойств серии `FormPropertySeries` и модуля `Lib`.

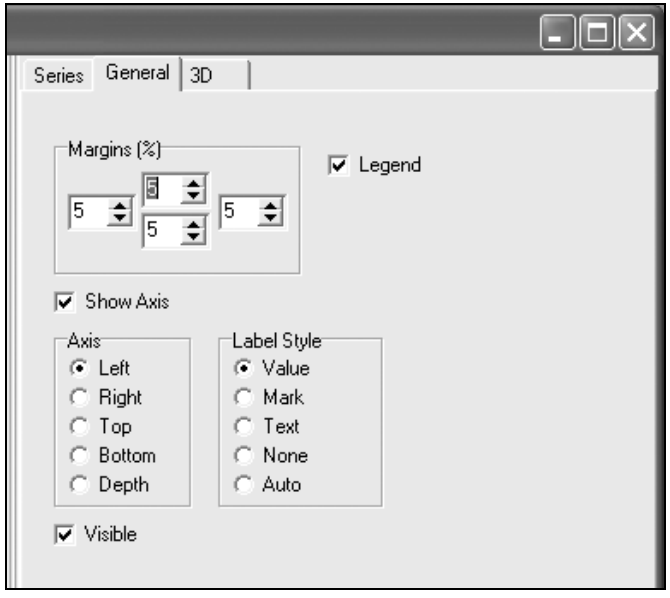


Рис. 5.35. Вкладка **General** главной формы проекта "Компонент *TChart*"

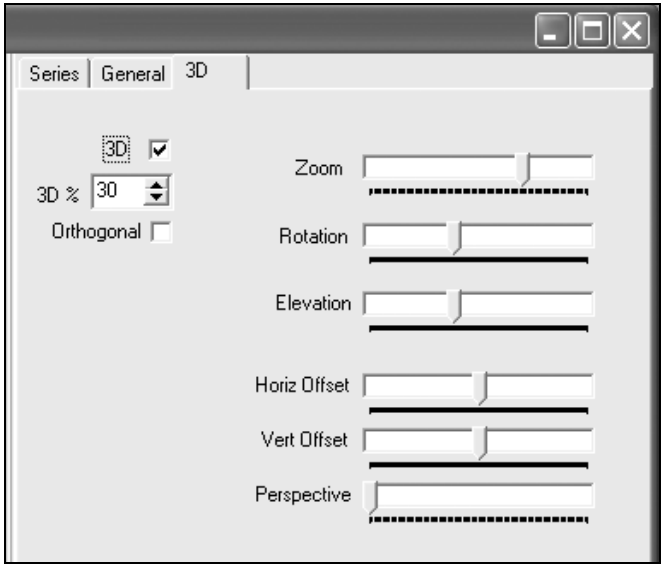


Рис. 5.36. Вкладка **3D** главной формы проекта "Компонент *TChart*"

На главной форме проекта FormMain расположены три вкладки: **General** (рис. 5.35), **3D** (рис. 5.36) и **Series** (рис. 5.37). Описание элементов вкладки **General** приведено в *разд. 5.9.3*, элементов вкладки **3D** — в *разд. 5.9.4*, элементов вкладки **Series** — в *разд. 5.9.2*.

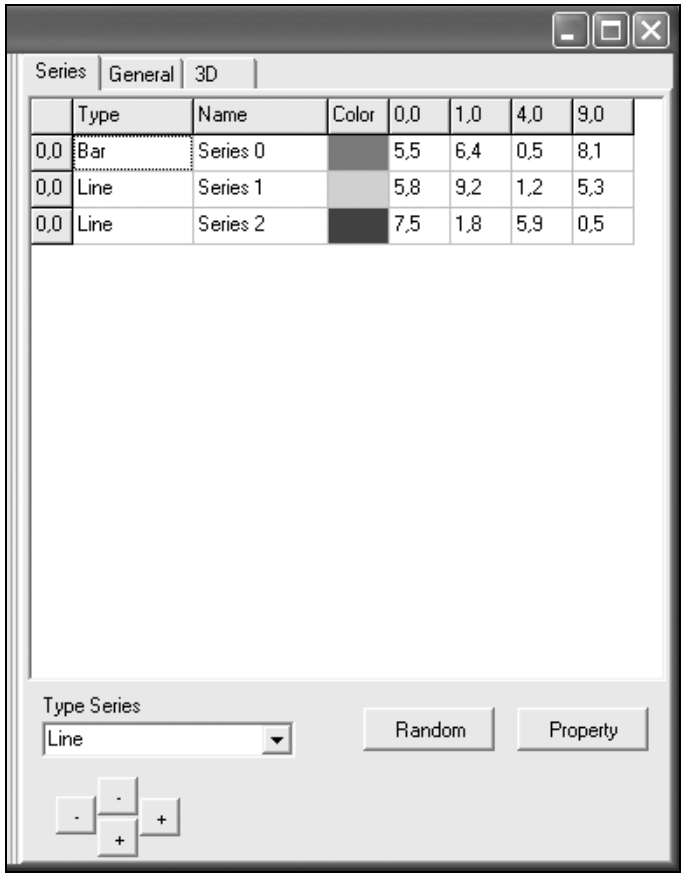


Рис. 5.37. Вкладка **Series**
главной формы проекта "Компонент *TChart*"

5.9.1. Генерация данных и добавление серий

Для хранения данных предназначена структура, представленная в листинге 5.26.

Листинг 5.26. Структура данных проекта "Компонент *TChart*"

```
TDate=record
        x: array of real;
```

```
        y: array of real;
        z: array of array of real;
        ASeries: array of TSeries;
    end;
```

В этой структуре поля X, Y, Z хранят данные по осям OX, OY, OZ, соответственно, а поле ASeries: array of TSeries содержит информацию о сериях (листинг 5.27).

Листинг 5.27. Структура данных серии

```
    TSeries=record
        Name: string;
        typeSeries: byte;
        Color     : TColor;
        BarStyle  : byte;
    end;
```

Поля в структуре листинга 5.27 имеют следующий смысл:

- Name — название серии;
- typeSeries — тип серии;
- Color — цвет серии;
- BarStyle — стиль серии для серий типа Bar.

При нажатии кнопки **Random** на вкладке **Series** происходит заполнение данных (листинг 5.28) и создание серий (листинг 5.29).

Листинг 5.28. Заполнение данных

```
procedure TFormMain.BtnDateClick(Sender: TObject);
var
    i,j,L1,L2: integer;
begin
    Randomize;
    with StringGrid1,Date do begin
    L1:=ColCount-1; L2:=RowCount-1;
    SetLength(x,L2); SetLength(y,L1);
    SetLength(ASeries,L1);
    SetLength(z,L1,L2);
        for i:=0 to L2-1 do x[i]:=i*i;
        for i:=0 to L1-1 do begin
            ASeries[i].Name:='Series '+IntToStr(i);
            ASeries[i].typeSeries:=typ_Series;
            ASeries[i].BarStyle:=0;
            ASeries[i].Color:=Random($FFFFFF);
        for j:=0 to L2-1 do
```

```
                        z[i,j]:=10*Random;
                end;
                end;
        SetChartSeries(Chart1);
end;
```

Листинг 5.29. Создание серий

```
procedure TFormMain.SetChartSeries(var Chart1: TChart);
var
        i,j,L1,L2: integer;
begin
        with Date do begin
                Chart1.SeriesList.Clear;
                L1:=Length(y); L2:=Length(x);
                SetLength(MySeries,L1);
        for i:=0 to L1-1 do begin
                case ASeries[i].typeSeries of
                        0: MySeries[i]:=TLineSeries.Create(self);
                        1: MySeries[i]:=TAreaSeries.Create(self);
                        2: MySeries[i]:=TPointSeries.Create(self);
                        3: begin
                                MySeries[i]:=TBarSeries.Create(self);
                                (MySeries[i] as TBarSeries).BarStyle:=
                                SetBarStyle(ASeries[i].BarStyle);
                           end;
                        4: begin
                               MySeries[i]:=THorizBarSeries.Create(self);
                              (MySeries[i] as THorizBarSeries).BarStyle:=
                                SetBarStyle(ASeries[i].BarStyle);
                           end;
                        5: MySeries[i]:=TChartShape.Create(self);
                        6: MySeries[i]:=TFastLineSeries.Create(self);
                        7: MySeries[i]:=TArrowSeries.Create(self);
                        8: MySeries[i]:=TGanttSeries.Create(self);
                        9: MySeries[i]:=TBubbleSeries.Create(self);
                            end;
                Chart1.AddSeries(MySeries[i]);
        for j:=0 to L2-1 do begin
                case ASeries[i].typeSeries of
                        7: (MySeries[i] as TArrowSeries).AddArrow(y[j],
                        0,x[j],z[i,j],ASeries[i].Name,ASeries[i].Color);
                        8: (MySeries[i] as TGanttSeries).AddGanttColor(0,
                        x[j],z[i,j],ASeries[i].Name, ASeries[i].Color);
```

```
                        9: (MySeries[i] as TBubbleSeries).AddBubble(x[j],
                        z[i,j],0.5,ASeries[i].Name, ASeries[i].Color);
        else
                Chart1.Series[i].AddXY(x[j],z[i,j],
                ASeries[i].Name, ASeries[i].Color)
                        end;
        end;
        end;
        end;
end;
```

Каждая из десяти серий создается своим методом `Create`. Добавление данных реализуется методом `AddXY` для всех типов серий, кроме серий типа `TArrowSeries`, `TganttSeries` и `TBubbleSeries`, для которых используются свои методы добавления.

5.9.2. Изменение свойств серии

На вкладке **Series** (рис. 5.37) расположены управляющие элементы, которые позволяют пользователю управлять типом серий (раскрывающийся список **Type Series**), заполнять серию набором случайных значений (кнопка **Random**), добавлять или уменьшать число серий на диаграмме (вертикально расположенная пара кнопок "+" и "-"), добавлять или уменьшать число значений в серии (горизонтально расположенная пара кнопок "+" и "-").

Нажатием кнопки **Property** на вкладке **Series** главной формы проекта "Компонент *TChart*" (рис. 5.34) или двойным щелчком указателя мыши по таблице на той же вкладке можно вызвать форму свойств серии FormPropertySeries. На этой форме можно изменить название, цвет и стиль серии Date.ASeries[NumSeries]. Изменение заканчивается вызовом метода `SetChartSeries`, который обновляет список серий компонента `Chart1` (листинг 5.30).

Листинг 5.30. Вызов формы свойств серии

```
procedure TFormMain.BtnPropertyClick(Sender: TObject);
begin
        with StringGrid2,Date do
        if Row<>0 then begin
                FormPropertySeries.NumSeries:=Row-1;
                if FormPropertySeries.ShowModal=mrOk then begin
                SetChartSeries(Chart1);
                        end;
end;
end;
```

5.9.3. Изменение общих свойств диаграммы

На вкладке **General** (рис. 5.35) находятся компоненты, позволяющие менять ширину полей, видимость условных обозначений (флажок **Legend**) и осей (флажок **Show Axis**). Комбинация группы радиокнопок **Axis** и флажка **Visible** управляет видимостью меток на отмеченной оси.

При изменении ширины полей (листинг 5.31) (группа текстовых полей **Margins %**) меняются свойства MarginTop, MarginBottom, MarginLeft, MarginRight, задающие ширину полей в процентах.

Листинг 5.31. Изменение ширины полей

```
Chart1.MarginTop:=SpinEdit2.Value;
Chart1.MarginBottom:=SpinEdit3.Value;
Chart1.MarginLeft:=SpinEdit4.Value;
Chart1.MarginRight:=SpinEdit5.Value;
```

Видимость условных обозначений определяется свойством `Legend.Visible`:

```
Chart1.Legend.Visible:=CheckBox5.Checked;
```

Видимость каждой из пяти осей (листинг 5.32) определяется свойствами:
`LeftAxis.Visible, RightAxis.Visible, TopAxis.Visible, BottomAxis.Visible, DepthAxis.Visible`.

Листинг 5.32. Изменение видимости осей

```
procedure TFormMain.CbAxisVisibleClick(Sender: TObject);
begin
        case RgAxis.ItemIndex of
                0: Chart1.LeftAxis.Visible:=CbAxisVisible.Checked;
                1: Chart1.RightAxis.Visible:=CbAxisVisible.Checked;
                2: Chart1.TopAxis.Visible:=CbAxisVisible.Checked;
                3: Chart1.BottomAxis.Visible:=CbAxisVisible.Checked;
                4: Chart1.DepthAxis.Visible:=CbAxisVisible.Checked;
        end;
end;
```

5.9.4. Изменение 3D-свойств диаграммы

На вкладке **3D** (рис. 5.36) находятся компоненты, позволяющие менять масштаб (регулятор **Zoom**), вид диаграммы (плоская или трехмерная) (флажок **3D**), вращать в вертикальной и горизонтальной плоскостях (регуляторы **Rotation** и **Elevation**) и перемещать диаграмму вверх и вниз по форме (регуляторы **Horiz Offset** и **Vert Offset**).

При изменении масштаба меняется свойство `View3dOptions.Zoom`:

```
Chart1.View3dOptions.Zoom:=TrackBar1.Position;
```

За вид диаграммы отвечает свойство `View3d`:

```
Chart1.View3d:=Cb3D.Checked;
```

Поворот диаграммы определяется значениями свойств `View3dOptions.Rotation` и `View3dOptions.Elevation`:

```
Chart1.View3dOptions.Rotation:=TrackBar2.Position;
Chart1.View3dOptions.Elevation:=TrackBar3.Position;
```

При перемещении меняются свойства `View3dOptions.HorizOffset` и `View3dOptions.VertOffset`:

```
Chart1.View3dOptions.HorizOffset:=TrackBar4.Position;
Chart1.View3dOptions.VertOffset:=TrackBar5.Position;
```

Глава 6

Алгоритмы компьютерной графики

6.1. Задачи компьютерной графики

Компьютерная обработка графической информации сводится к решению 3-х классов задач.

1. Задачи распознавания образа: на основе имеющегося оцифрованного изображения осуществить идентификацию объекта, т. е. получить его описание. Здесь выполняется преобразование вида "изображение → описание объекта".
2. Задача обработки изображения: осуществить преобразование имеющегося оцифрованного изображения с целью изменения тех или иных его характеристик. Здесь выполняется преобразование вида "изображение → изображение".
3. Задача построения изображения: по описанию объекта построить его изображение на графическом устройстве. Здесь выполняется преобразование вида "описание объекта → изображение".

Последняя задача и является предметом нашего рассмотрения. При этом описание объекта должно иметь вид математической модели. Можно говорить о двух основных подзадачах задачи построения изображения:

1. Построение математической модели изображаемого объекта;
2. Визуализация объекта в соответствии с этой моделью.

Согласно классификации Ф. Мартинеса [83] информацию, содержащуюся в изображении, можно классифицировать следующим образом:

- определительная информация — *идентификация* и *структура*;
- топологическая информация — *морфология* и *геометрия*;
- визуальная информация — *внешний вид* и *освещение*.

Идентификация основана на именовании объектов или множеств объектов.

Структура отражает различные виды отношений объектов между собой:

- логические отношения (принадлежность, включение и т. д.),
- топологические отношения (близость, касание и т. д.),

- функциональные отношения (зависимость характеристик одних объектов от других).

Морфология отражает форму объекта независимо от его положения и точки наблюдения. Все многообразие геометрических объектов является комбинацией различных примитивов — простейших фигур, которые, в свою очередь, состоят из графических элементов — точек, линий, поверхностей.

Геометрия отражает информацию о проекциях, видимости и т. д. Геометрия дополняется информацией о съемке (точке съемки, направлении, типе объектива, фокусном расстоянии, формате снимка) и отображении. Отображением называется преобразование двумерного изображения плоскости проецирования (плана) в двумерное изображение кадра.

Внешний вид определяется информацией, относящейся к материалу, из которого состоит объект, в частности, о цвете, текстуре, яркости и прозрачности материала.

Освещение определяется природой, числом и расположением источников света, а также условиями видимости: наличием тумана, дыма и т. д.

Не останавливаясь на всей проблеме синтеза изображения, перечислим основные математические задачи, встречающиеся при создании изображения на экране компьютера:

- преобразование системы координат и проецирование;
- удаление невидимых линий;
- освещение и тень;
- моделирование цвета;
- моделирование текстуры;
- логические операции над объектами.

Часть этих задач рассмотрена в данной главе, а другая часть — в *главе 9*.

6.2. Классификация алгоритмов

Алгоритмы компьютерной графики можно разделить на два уровня: нижний и верхний. Группа алгоритмов нижнего уровня предназначена для реализации графических примитивов (линий, окружностей, заполнений и т. п.). Эти алгоритмы или подобные им воспроизведены в графических библиотеках языков высокого уровня или реализованы аппаратно в графических процессорах рабочих станций.

Среди алгоритмов нижнего уровня можно выделить следующие группы:

- простейшие, в смысле используемых математических методов, и отличающиеся простотой реализации. Как правило, такие алгоритмы не являются оптимальными по объему выполняемых вычислений или требуемым ресурсам памяти;

- алгоритмы, использующие более сложные математические предпосылки, а также некоторые эвристические алгоритмы, отличающиеся большей эффективностью;
- аппаратно-реализуемые, к которым могут быть отнесены алгоритмы, допускающие распараллеливание, рекурсивные алгоритмы и алгоритмы, реализуемые в командах процессора. В эту группу могут попасть и алгоритмы, представленные в двух первых группах;
- алгоритмы со специальным назначением (например, для устранения лестничного эффекта).

К алгоритмам верхнего уровня относятся в первую очередь алгоритмы удаления невидимых линий и поверхностей. Задача удаления невидимых линий и поверхностей продолжает оставаться центральной в машинной графике. От эффективности алгоритмов, позволяющих решить эту задачу, зависят качество и скорость построения трехмерного изображения.

К задаче удаления невидимых линий и поверхностей примыкает задача построения полутоновых реалистических изображений. Решение этой задачи требует учета явлений, связанных с количеством и характером источников света, учета свойств поверхности тела (прозрачность, преломление, отражение света).

Однако при этом не следует забывать, что построение объектов в алгоритмах верхнего уровня обеспечивается примитивами, реализующими алгоритмы нижнего уровня, поэтому нельзя игнорировать проблему выбора и разработки эффективных алгоритмов нижнего уровня.

Для разных областей применения машинной графики на первый план могут выдвигаться разные свойства алгоритмов. Для научной графики большое значение имеет универсальность алгоритма, быстродействие может отходить на второй план. Для систем моделирования, воспроизводящих движущиеся объекты, быстродействие становится главным критерием, поскольку требуется генерировать изображение практически в реальном масштабе времени.

6.3. Построение растровых изображений

Обычные изображения, с которыми сталкивается человек в своей деятельности (чертежи, графики, карты, художественные картины и т. п.), реализованы на плоскости, которая является бесконечным и непрерывным множеством точек. Большинство же графических устройств являются растровыми, т. е. представляют изображение в виде *растра* — прямоугольной матрицы элементов изображения (пикселов). Именно таким устройством является и видеомонитор (дисплей). Пикселы на экране дисплея имеют конечные физические размеры, и их число ограничено. Растр, таким образом, представляет собой конечное дискретное множество точек. Поэтому воспроизведение изображения на экране дисплея (или на другом

растровом устройстве) требует выполнения определенных аппроксимационных процедур и неизбежно связано с частичной потерей информации. Отображение любого объекта на целочисленную решетку называется разложением его в растр или просто растровым представлением.

Рассмотрим для примера задачу растрового представления прямой линии. Можно попытаться воспользоваться для этого уравнением $y = ax + b$, задавая аргументу x последовательность целых значений. При этом результаты вычисления значений переменной y надо округлять до целых. В результате, вместо плавной прямой линии мы получим ступенчатую линию или даже набор изолированных точек (рис. 6.1).

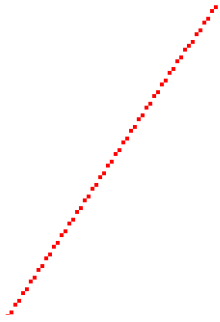


Рис. 6.1. Растровое представление прямой линии

Аналогичная ситуация имеет место для окружности или эллипса. Уравнение окружности выглядит следующим образом:

$$\begin{aligned} x &= x_c + r \times \cos(\varphi), \\ y &= y_c + r \times \sin(\varphi), \end{aligned} \qquad (6.1)$$

где (x_c, y_c) — координаты центра окружности, r — радиус, φ — угол для текущей точки (x, y).

Можно строить окружность прямо по этому уравнению, задав определенный шаг по углу ($\varphi = 0,..,360$). Однако если шаг будет слишком мал, окружность за счет округления координат станет неровной, и некоторые точки высветятся несколько раз. Обычно шаг по углу выбирается равным $1/r$ радиан (чаще всего шаг изменения угла должен быть переменным для того, чтобы избежать разрывов или отсутствия изменения координат).

При построении окружностей следует учитывать, что размеры пикселов по вертикали и горизонтали не совпадают (кроме режима VGA 640×480) и окружности будут вытягиваться в эллипсы. Процедура `GetAspectRatio (var Xasp, Yasp: word)` возвращает значения двух переменных, характеризующие физическую форму пиксела. Эти значения определяют выравнивающий коэффициент x_{asp} / y_{asp}, который следует использовать при построении изображений. Можно осуществить простой

алгоритм аппроксимации окружности отрезками прямых. Например, координаты 60 отрезков получаются с шагом 6 градусов, затем они соединяются прямыми линиями.

Для ускорения построения окружности используется ее симметрия (вычисляются координаты точек только 1/8 части окружности — для сегмента от 0 до 45 градусов). Кроме того, чтобы избежать выполнения сложных операций вычисления синуса и косинуса, можно воспользоваться рекуррентными формулами, выражающими координаты следующей точки окружности через координаты предыдущей точки:

$$\begin{aligned} x_2 &= x_c + (x_1 - x_c) \times cs + (y_1 - y_c) \times sn, \\ y_2 &= y_c + (y_1 - y_c) \times cs - (x_1 - x_c) \times sn, \end{aligned} \qquad (6.2)$$

где $sn = \sin(\Delta\varphi)$, $cs = \cos(\Delta\varphi)$ и $\Delta\varphi$ — шаг по углу.

Начальная точка для рекуррентной формулы: $x_1 = x_c$, $y_1 = y_c + r$.

Растровое представление верхней полуокружности показано на рис. 6.2.

Рис. 6.2. Растровое представление верхней полуокружности

6.3.1. Алгоритм Брезенхейма для отрезка прямой

Рис. 6.3. Дж. Брезенхейм

Проблема корректной реализации растрового представления отрезка прямой линии была решена Дж. Брезенхеймом (Jack E. Bresenham). В 1962 году им был разработан соответствующий алгоритм, первоначально предназначавшийся для цифровых графопостроителей, который впоследствии стал использоваться и для растровых дисплеев.

Алгоритм Брезенхейма в его первоначальном виде позволял получить непрерывное множество пикселов, представляющих отрезок прямой, проведенной между двумя точками растра (x_1, y_1) и (x_2, y_2). Идея алгоритма заключается в том, что одна координата изменяется на единицу, а другая — либо не изменяется, либо изменяет-

ся на единицу в зависимости от расстояния от соответствующей точки до ближайшего узла координатной сетки. Расстояние от точки отрезка до ближайшего узла по соответствующей ортогональной координате называется ошибкой. Алгоритм организован таким образом, что для вычисления второй координаты требуется знать только знак этой ошибки.

Выбор независимой координаты определяется угловым коэффициентом прямой:

$$dy/dx = (y_2 - y_1)/(x_2 - x_1).$$

Для линий, у которых абсолютная величина углового коэффициента не превосходит единицу, в качестве независимой следует выбирать координату x, а для линий с наклоном большим, чем единица — координату y. Будем считать для определенности, что линия имеет малый наклон и независимо изменяемой является координата x.

Будем использовать математическое описание отрезка в виде:

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1), \quad x \in [x_1, x_2]. \tag{6.3}$$

Предположим далее, что во время построения отрезка в качестве очередного выбран пиксел $P(x_p, y_p)$, как это показано на рис. 6.4. На рисунке показаны также пикселы Q_1 и Q_2, один из которых должен быть выбран на следующем шаге построения отрезка прямой. Еще три пиксела, представленные на рисунке, являются "кандидатами" для третьего шага.

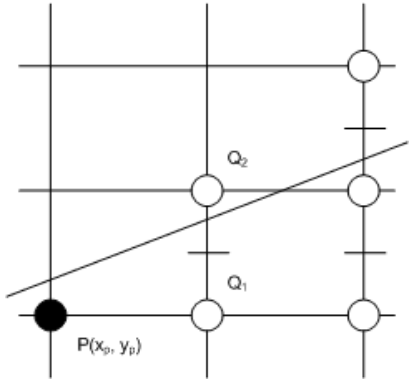


Рис. 6.4. Выбор следующей точки растра

Следующим может быть либо пиксел Q_1, либо Q_2. Определить, какой из двух вариантов наиболее точно будет продолжать линию, можно с помощью срединной точки. Если отрезок проходит выше срединной точки, то выбирается пиксел Q_2, иначе — пиксел Q_1.

Чтобы выразить это условие математически, введем функцию:

$$F(x, y) = (x - X_1) \times dY - (y - Y_1) \times dX. \quad (6.4)$$

Знак этой функции определяет расположение точки с координатами (x, y) относительно прямой, в соответствии со следующим правилом:

- если $F(x, y) = 0$, то точка (x, y) лежит на отрезке;
- если $F(x, y) > 0$, то точка (x, y) лежит ниже отрезка;
- если $F(x, y) < 0$, то точка (x, y) лежит выше отрезка.

Координаты срединной точки, определяющей выбор между пикселами Q_1 и Q_2, будут равны $x_p + 1, y_p + 1/2$, а значение функции F в этой точке:

$$d = F(x_p + 1, y_p + 1/2). \quad (6.5)$$

Если в качестве следующего будет выбран пиксел Q_1, то значение функции в новой срединной точке будет равно:

$$d' = F(x_p + 2, y_p + 1/2) \quad (6.6)$$

и приращение функции

$$\Delta d = d' - d = dy. \quad (6.7)$$

В случае выбора пиксела Q_2, значение функции в новой срединной точке будет равно:

$$d' = F(x_p + 2, y_p + 3/2), \quad (6.8)$$

а приращение функции:

$$\Delta d = d' - d = dy - dx. \quad (6.9)$$

Важно, что значения приращений (6.7) и (6.9) не зависят от текущих координат и определяются только разницами координат начальной и конечной точек отрезка. Если теперь выбрать в качестве точки P начальную точку (x_1, y_1), то, согласно (6.5), в качестве исходного значения величины d, получим:

$$d_0 = F(x_1 + 1, y_1 + 1/2) = dy - dx/2. \quad (6.10)$$

Знак d_0 определит выбор следующего пиксела, а также величину приращения Δd, добавляя которое к d_0 получим следующее значение d и т. д.

Поскольку координаты пикселов являются целочисленными, желательно исключить из алгоритма возможность появления дробных значений. Это можно сделать, введя множитель 2 для величины d, т. е. переопределив ее следующим образом:

$$d = 2 \times F(x_p + 1, y_p + 1/2). \quad (6.11)$$

Тогда алгоритм Брезенхейма для отрезка прямой можно записать в следующем виде (листинг 6.1).

Листинг 6.1. Алгоритм Брезенхейма для отрезка прямой

```
X := X1; Y: = Y1;
dX := X2 - X1;
dY := Y2 - Y1;
d : =2*dY - dX;
while X =< X2 do begin
  PutPixel (X, Y);
  X := X + 1;
  d := d + 2*dY
  if d >= 0 then begin
    Y := Y + 1;
    d := d - 2*dX
  end
end
```

Этот алгоритм имеет очевидное обобщение, позволяющее учесть возможность различных соотношений между координатами начальной и конечной точек отрезка, а также большого ($|dy| > |dx|$) и малого ($|dy| < |dx|$) наклона отрезка к оси OX (листинг 6.2).

Листинг 6.2. Обобщенный алгоритм Брезенхейма для отрезка прямой

```
// упорядочение концевых точек (X1<X2)
if X2 < X1 then begin
  swap(X1,X2);
  swap(Y1,Y2);
end
dX := X2 - X1;              // dX>=0
dY := Y2 - Y1;
// определение направления отрезка и шага по Y
if dY>0 then S := 1     // снизу-вверх
else
  if dY<0 then S := -1  // сверху-вниз
  else S := 0;         // горизонтально
d := 2*dY - dX;
Y := Y1; X := X1;
if abs(dY)<=dX then
  while X <= X2 do begin    // малый угол наклона
    PutPixel (X,Y);
    X := X + 1;
    d := d + 2*dY;        // без смещения по вертикали
```

```
    if d>0 then begin
      d := d - 2*dX;   // со смещением по вертикали
      Y := Y + S
    end;
  end
else
  while (Y2-Y)*S>=0 do  // большой угол наклона
    PutPixel (X,Y);
    Y := Y + S;
    d :=d + 2*dX;    // без смещения по горизонтали
    if d>0 then begin
      d := d - 2*dY; // со смещением по горизонтали
      X := X + 1
    end;
 end;
```

Оценивая достоинства полученного алгоритма, можно отметить, что он выполняет оптимальную аппроксимацию отрезка, используя при этом целочисленную арифметику и минимальное количество операций сложения и вычитания.

6.3.2. Алгоритм Брезенхейма для окружности

Рассмотрим применение алгоритма Брезенхейма для построения окружности. Учитывая высокую степень симметрии этой фигуры, ограничимся построением 1/8 ее части, поскольку остальные участки окружности могут быть получены путем простых преобразований симметрии.

Если в качестве текущего на данном шаге построения дуги выбран пиксел $P(x_p, y_p)$ (рис. 6.5), то следующим может быть пиксел Q_1 либо Q_2. Выбор между двумя возможностями будем осуществлять по той же схеме, что и в случае отрезка прямой. Введем функцию:

$$F(x, y) = x^2 + y^2 - R^2 . \tag{6.12}$$

Правило расположения точки (x, y) относительно окружности выглядит следующим образом:

- если $F(x, y) = 0$, то точка (x, y) принадлежит окружности;
- если $F(x, y) < 0$, то точка (x, y) лежит внутри окружности;
- если $F(x, y) > 0$, то точка (x, y) лежит вне окружности.

Значение функции F в срединной точке для пиксела P равно:

$$d = F(x_p + 1, y_p - 1/2). \tag{6.13}$$

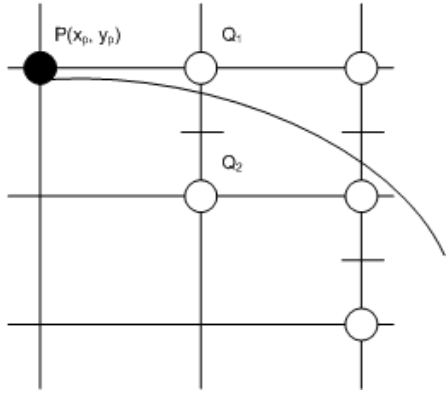


Рис. 6.5. Выбор следующей точки растра для дуги окружности

Если в качестве следующего будет выбран пиксел Q_1, то значение функции в новой срединной точке будет:

$$d' = F(x_p + 2, y_p - 1/2) \tag{6.14}$$

и приращение функции

$$\Delta d = d' - d = 2 \times x_p + 3. \tag{6.15}$$

В случае выбора пиксела Q_2, значение функции в новой срединной точке будет равно:

$$d' = F(x_p + 2, y_p - 3/2), \tag{6.16}$$

а приращение функции:

$$\Delta d = d' - d = 2 \times (x_p - y_p) + 5. \tag{6.17}$$

Выбрав самый верхний пиксел дуги в качестве начального, получим следующее стартовое значение для функции F в срединной точке:

$$d_0 = F(1, R - 1/2) = 5/4 - R. \tag{6.18}$$

Полученное значение является вещественным, однако ее приращения (6.15) и (6.17) — целые. Поэтому можно избежать использования вещественных операций, переопределив величину d следующим образом:

$$d = F(x_p + 1, y_p - 1/2) - 1/4. \tag{6.19}$$

Теперь значение функции F в срединной точке нужно было бы сравнивать с –1/4, однако, в силу целочисленности приращений Δd, сравнение можно по-прежнему производить с нулем.

Алгоритм построения дуги окружности в первом квадранте представлен в листинге 6.3.

Листинг 6.3. Алгоритм Брезенхейма для дуги окружности

```
  d1:=r2*(1-r);
  j:=0; i:=0;
  while i<=imin do begin
    if d1<0
    then d2:=d1+(2*i+3)*r2 // без смещения по вертикали
    else begin
      d2:=d1+ r2*(2*i-2*(r-j)+5); // со смещением по вертикали
      j:=j+1
    end;
    d1:=d2; i:=i+1;
  end;
  d1:=r*(1-r);
  j:=r; i:=r;
  while i>imin  do begin
    if d1<0
    then d2:=d1+(2*(r-j)+3)*r2   // без смещения по горизонтали
    else begin
      d2:=d1+r2*(2*(r-j)-2*i+5); // со смещением по горизонтали
      i:=i-1
    end;
    d1:=d2; j:=j-1;
  end;
end;
```

6.3.3. Экранная система координат

При реализации алгоритмов компьютерной графики следует иметь в виду особенности выбора координатной системы на растровых графических устройствах, в частности, на видеомониторе. Принято помещать начало связанной с экраном системы координат в левый верхний угол экрана, направляя ось абсцисс слева направо, а ось ординат — сверху вниз, как показано на рис. 6.6. Обозначения осей подчеркивают то обстоятельство, что экранные координаты являются целочисленными.

При построении изображений геометрических элементов на листе бумаги принято направлять ось ординат снизу вверх, а начало координат размещать произвольным образом (рис. 6.7).

Формулы преобразования экранных координат в бумажные координаты, а также формулы обратных преобразований легко получить на основе простых геометрических соотношений подобия:

$$x = x_1 + (i - i_1) \times (x_2 - x_1)/(i_2 - i_1), \qquad (6.20а)$$

$$y = y_1 + (j_2 - j) \times (y_2 - y_1)/(j_2 - j_1), \qquad (6.20б)$$

$$jj_2 - Round\left(\left(y - y_1\right) \times \left(j_2 - j_1\right) / \left(y_2 - y_1\right)\right), \quad (6.20в)$$

$$i = i_1 + Round\left(\left(x - x_1\right) \times \left(i_2 - i_1\right) / \left(x_2 - x_1\right)\right). \quad (6.20г)$$

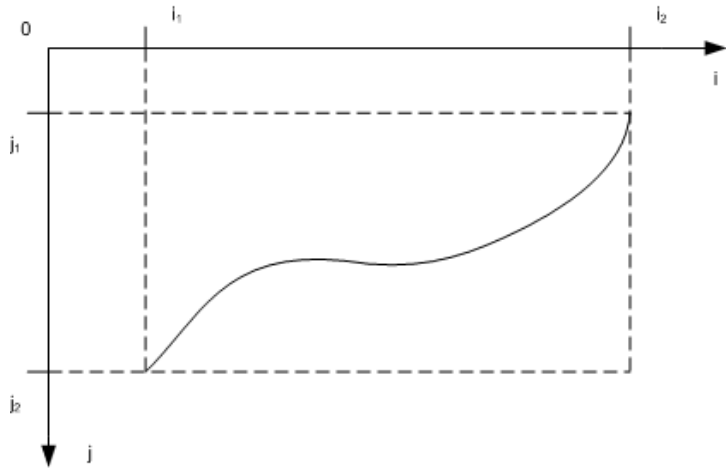


Рис. 6.6. Экранная система координат

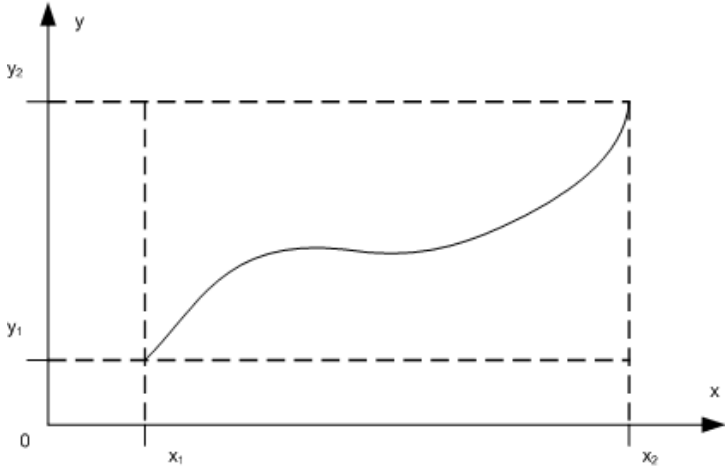


Рис. 6.7. Бумажная система координат

6.3.4. Проект "Алгоритмы Брезенхейма"

Проект приведен на компакт-диске в папке **Примеры** | **Глава 6** | **Алгоритмы Брезенхейма**. В предлагаемом проекте реализованы варианты алгоритма Брезен-

хейма для построения отрезка прямой, эллипса и дуги эллипса. Кроме того, имеется возможность заливки замкнутых областей произвольным цветом.

Работающее приложение имеет вид, представленный на рис. 6.8.

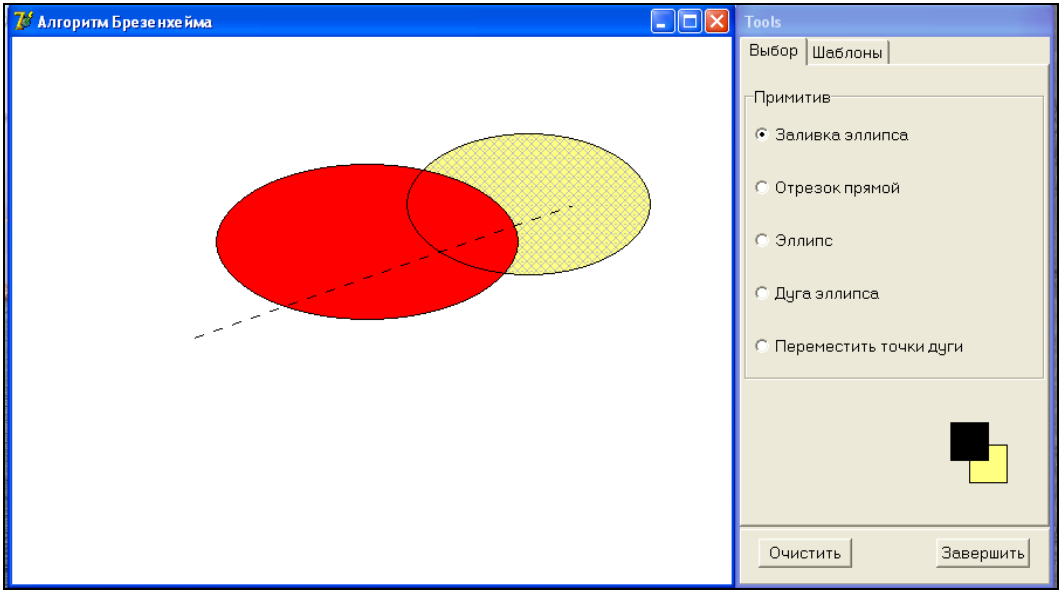


Рис. 6.8. Окно приложения "Алгоритм Брезенхейма"

Одна из форм проекта — `FormMain` (с внешним именем **Алгоритм Брезенхейма**) используется для вывода изображений, вторая — `FormTools` (с внешним именем **Tools**) представляет собой инструментальную панель с двумя вкладками. Вкладка **Выбор** содержит набор радиокнопок для задания функции (действия), а также экран текущих цветов для установки цвета линии и цвета заливки. На вкладке **Шаблоны** пользователь может задать шаблоны линии и заливки (рис. 6.9).

Модуль главной формы содержит обработчики событий `OnMouseDown`, `OnMouseMove` и `OnMouseUp`.

В процедуре `FormMain.MouseDown` (листинг 6.4) выполняется установка флага рисования `drawing`, инициализируются значения переменных `direct`, `x1`, `y1`, `x2` и `y2`, используемых при построении фокусного прямоугольника. В зависимости от выбранной пользователем функции (индекс функции задается значением переменной `fl_tools`, которое, в свою очередь, определяется активной радиокнопкой на вкладке **Выбор**) выполняются следующие действия:

- вызов процедуры `FillElps` для функции "Закрасить эллипс (сектор эллипса)";
- инициализация построения отрезка прямой для функции "Нарисовать отрезок прямой";

- вызов функции `FindPoint` для нахождения нового положения одной из концевых точек дуги эллипса.

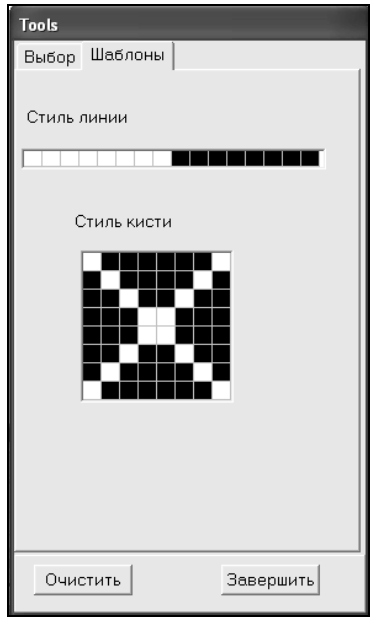


Рис. 6.9. Вкладка **Шаблоны** формы **Tools**

Листинг 6.4. Обработчик события *OnMouseDown* для главной формы

```
procedure TFormMain.FormMouseDown(Sender: TObject; Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
begin
  drawing:=true;
  direct:=0;
  x1:=X; y1:=Y; x2:=X; y2:=Y;
  with Canvas do
    case fl_tools of
      0: FillElps(X,Y,clWhite,Canvas);
      1: begin
           Pen.Mode:=pmNotXor;
           MoveTo(x1,y1); LineTo(x2,y2);
         end;
      4: if Length(Points)=2 then
           IndexPoint:=FindPoint(X,Y);
    end;
end;
```

Процедура `FormMain.MouseMove` (листинг 6.5) содержит операторы, обеспечивающие прорисовку фокусных прямоугольников при построении эллипса и дуги. Процедура `Rect`, задающая этот прямоугольник, требует указания координат левого верхнего и правого нижнего его углов, которые, в зависимости от направления движения мыши, выражаются разными комбинациями координат начального `(x1, y1)` и конечного `(x2, y2)` положений указателя мыши. Это приводит к необходимости изменять порядок перечисления указанных координат в вызовах процедуры `DrawFocusRect` при разных направлениях движения мыши. Последние фиксируются путем задания значения глобальной переменной `direct` и локальной переменной `direct1`. Использование двух переменных обеспечивает корректную прорисовку фокусного прямоугольника при изменении направления движения мыши.

При построении отрезка прямой в этой процедуре выполняются обычные действия, обеспечивающие прорисовку на канве сплошной линии. Что же касается случая перемещения концевой точки дуги, то для него обработка данного события сводится к изменению текущей координаты перемещаемой точки с последующим вызовом процедуры `DrawArc` для перерисовки дуги.

Листинг 6.5. Обработчик события *OnMouseMove* для главной формы

```
procedure TFormMain.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X,Y: Integer);
var
  direct1: Byte;
begin
  if drawing then begin
    if  (x1<>X) and (y1<>Y) then begin
      if x1<X then
        if y1>Y then direct1:=1
        else direct1:=2
      else
        if y1>Y then direct1:=3
        else direct1:=4;
    end
    else direct1:=0;
    with Canvas do
      case fl_tools of
        1: begin
             MoveTo(x1,y1); LineTo(x2,y2);
             x2:=x; y2:=y;
             MoveTo(x1,y1); LineTo(x2,y2);
           end;
      2,3: begin
             case direct of
               1: DrawFocusRect(Rect(x1,y2,x2,y1));
               2: DrawFocusRect(Rect(x1,y1,x2,y2));
```

```
              3: DrawFocusRect(Rect(x2,y2,x1,y1));
              4: DrawFocusRect(Rect(x2,y1,x1,y2));
            end;
            direct:=direct1;
            x2:=x; y2:=y;
            case direct1 of
              1: DrawFocusRect(Rect(x1,y2,x2,y1));
              2: DrawFocusRect(Rect(x1,y1,x2,y2));
              3: DrawFocusRect(Rect(x2,y2,x1,y1));
              4: DrawFocusRect(Rect(x2,y1,x1,y2));
            end;
          end;
        4: if Length(Points)=2 then begin
            Points[IndexPoint].x:=x;
            Points[IndexPoint].y:=y;
            ClearCanvas;
            DrawArc(Canvas);
          end;
      end;
  end;
end;
```

Обработка события `OnMouseUp` для главной формы включает:

- вызов процедуры `DrawLine`, реализующей алгоритм Брезенхейма для отрезка прямой;
- обращение к процедуре `DrawElps`, реализующей тот же алгоритм для эллипса;
- вызов процедуры `DrawArc` для прорисовки дуги эллипса.

В процедуре обработки этого события, представленной в листинге 6.6, снимается флаг `drawing`, а также завершается прорисовка фокусных прямоугольников для эллипса и его дуги. Кроме того, для дуги запоминаются значения координат `direct`, `x1`, `y1`, `x2` и `y2`, что позволяет продолжить ее обработку после построения других фигур. Прорисовка эллипса и его дуги выполняется только тогда, когда их размеры превосходят некоторый минимум, задаваемый константой `Min`, что обеспечивает устойчивость работы алгоритма.

Листинг 6.6. Обработчик события *OnMouseUp* для главной формы

```
procedure TFormMain.FormMouseUp(Sender: TObject;
Button: TMouseButton; Shift: TShiftState;
X, Y: Integer);
begin
  drawing:=false;
  with Canvas do
  case fl_tools of
    1: begin
```

```
        MoveTo(x1,y1); LineTo(x2,y2);
        Pen.Mode:=pmCopy;
        DrawLine(x1,y1,x2,y2,Canvas);
      end;
    2: begin
        case direct of
             1: DrawFocusRect(Rect(x1,y2,x2,y1));
             2: DrawFocusRect(Rect(x1,y1,x2,y2));
             3: DrawFocusRect(Rect(x2,y2,x1,y1));
             4: DrawFocusRect(Rect(x2,y1,x1,y2));
        end;
        if (abs(x2-x1)>Min) and (abs(y2-y1)>Min)
           then DrawElps(x1,y1,x2,y2,0,0,0,0,Canvas);
      end;
    3: begin
        SetLength(Points,2);
        x1a:=x1; y1a:=y1; x2a:=x2; y2a:=y2;
        directa:=direct;
        case directa of
             1: DrawFocusRect(Rect(x1a,y2a,x2a,y1a));
             2: DrawFocusRect(Rect(x1a,y1a,x2a,y2a));
             3: DrawFocusRect(Rect(x2a,y2a,x1a,y1a));
             4: DrawFocusRect(Rect(x2a,y1a,x1a,y2a));
        end;
        xa:=(x1a+x2a) div 2; ya:=(y1a+y2a) div 2;
        Points[0].x:=x2a;
        Points[0].y:=ya;
        Points[1].x:=xa;
        if y2a>y1a then Points[1].y:=y1a
        else Points[1].y:=y2a;
        ClearCanvas;
        if (abs(x2a-x1a)>Min) and (abs(y2a-y1a)>Min)
          then DrawArc(Canvas);
      end;
    4: if Length(Points)=2 then begin
        Points[IndexPoint].x:=x;
        Points[IndexPoint].y:=y;
        ClearCanvas;
        DrawArc(Canvas);
      end;
  end;
end;
```

Наконец, процедура `ClearCanvas` производит очистку холста, прорисовывая на нем прямоугольник, залитый белым цветом.

```
procedure TFormMain.ClearCanvas;
begin
```

```
  with Canvas do
    FillRect(Rect(0,0,Width,Height));
end;
```

Процедуры, реализующие основные алгоритмы, размещены в библиотечном модуле `Lib`. Рассмотрим каждую из этих процедур.

Процедура `DrawLine` для построения отрезка прямой линии по алгоритму Брезенхейма представлена в листинге 6.7.

Листинг 6.7. Построение отрезка прямой линии по алгоритму Брезенхейма

```
procedure DrawLine(x1,y1,x2,y2: Integer;
Canvas: TCanvas);
// построение отрезка прямой по Брезенхейму
var
  i,j,dj,dx,dy,k: Integer;
  d1,d2: Integer;
begin
  k:=-1;
  if x2<x1 then begin
    i:=x1; x1:=x2; x2:=i;
    i:=y1; y1:=y2; y2:=i;
  end;
  dx:=x2-x1;       // x1<x2
  dy:=y2-y1;
  if dy>0 then dj:=1
  else
    if dy<0 then dj:=-1
    else dj:=0;
  dy:=abs(dy);
  d1:=2*y2-y1;
  j:=y1; i:=x1;
  if (dx<>0) and (dy<=dx) or (dy=0) then
  while i<=x2 do begin
    if d1<0
    then d2:=d1+2*dy      // без смещения по вертикали
    else begin
      d2:=d1+2*(dy-dx);   // со смещением по вертикали
      if (y2-j)*dj>0 then j:=j+dj
    end;
    Inc(k);
    if w and (1 shl (k mod 16))<>0 then
      Canvas.Pixels[i,j]:=PenColor;
    d1:=d2;
    i:=i+1;
  end
```

```
  else
    while (y2-j)*dj>=0 do begin
      if d1<0
      then d2:=d1+2*dx   // без смещения по горизонтали
      else begin
        d2:=d1+2*(dx-dy);// со смещением по горизонтали
        if i<x2 then i:=i+1
      end;
      Inc(k);
      if w and (1 shl (k mod 16))<>0 then
        Canvas.Pixels[i,j]:=PenColor;
      d1:=d2;
      j:=j+dj;
   end;
end;
```

Текст этой процедуры, в основном, совпадает с обсуждавшимся ранее текстом алгоритма листинга 6.2 за одним исключением. В процедуре `DrawLine` предусмотрена возможность использования шаблона при построении линии. 16-битный шаблон задается значением беззнаковой целой переменной `w`. Для `k`-го пиксела отрезка выполняется проверка:

```
        if w and (1 shl (k mod 16))<>0 then
          Canvas.Pixels[i,j]:=PenColor; ,
```

определяющая необходимость закрашивания его цветом линии.

Построение дуги эллипса по алгоритму Брезенхейма осуществляется процедурой `DrawElps` (листинг 6.8).

Листинг 6.8. Построение дуги эллипса по алгоритму Брезенхейма

```
procedure DrawElps(x1,y1,x2,y2,x3,y3,x4,y4: Integer; Canvas: TCanvas);
// постоение дуги эллипса
var
  i,imin,j,d1,d2,n: Integer;
  rx,ry,r2x,r2y: Integer;
  e,ca,cb: Real;

  function Varnt(xa,ya,xb,yb: Integer): Integer;
  var
    n1,n2: Integer;

    function Quadrant(x,y: Integer): Integer;
    begin
      if (x-x0>0) and (y-y0<=0)
      then Result:=0
      else
```

```
      if (x-x0<=0) and (y-y0<0)
      then Result:=1
      else
        if (x-x0<0) and (y-y0>=0)
        then Result:=2
        else Result:=3;
  end;

begin
  n1:=Quadrant(xa,ya);
  n2:=Quadrant(xb,yb);
  if n1>n2 then n1:=n1-4;
  Result:=n2+4*(n2-n1);
end;

procedure Symmetry(x,y: Integer);
var f: Real;
begin
  f:=x/Sqrt(x*x+Sqr(ry-y));
  with Canvas do
  case n of
    0: if (f<ca) and (f>cb)    // 0 0
       then Pixels[x+x0,y+y1]:=PenColor;
    1: if (f>ca) and (f<cb)    // 1 1
       then Pixels[x0-x,y+y1]:=PenColor;
    2: if (f<ca) and (f>cb)    // 2 2
       then Pixels[-x+x0,2*ry-y+y1]:=PenColor;
    3: if (f>ca) and (f<cb)    // 3 3
       then Pixels[x+x0,2*ry-y+y1]:=PenColor;
    4: begin                   // -1 0
         if f>ca then Pixels[x+x0,
            2*ry-y+y1]:=PenColor;
         if f>cb then Pixels[x+x0,y+y1]:=PenColor;
       end;
    5: begin                   // 0 1
         if f<ca
           then Pixels[x0+x,y+y1]:=PenColor;
         if f<cb
           then Pixels[x0-x,y+y1]:=PenColor;
       end;
    6: begin                   // 1 2
         if f>ca then Pixels[-x+x0,y+y1]:=PenColor;
         if f>cb then Pixels[-x+x0,
            2*ry-y+y1]:=PenColor;
       end;
```

```
 7: begin                    // 2 3
      if f<ca then Pixels[-x+x0,
         2*ry-y+y1]:=PenColor;
      if f<cb then Pixels[x+x0,
         2*ry-y+y1]:=PenColor;
    end;
 8: begin                    // -2 -1 0
      if f<ca then Pixels[-x+x0,
         2*ry-y+y1]:=PenColor;
      if f>cb then Pixels[x+x0,y+y1]:=PenColor;
      Pixels[x+x0,2*ry-y+y1]:=PenColor;
    end;
 9: begin                    // -1 0 1
      if f>ca then Pixels[x+x0,
         2*ry-y+y1]:=PenColor;
      if f<cb then Pixels[-x+x0,y+y1]:=PenColor;
      Pixels[x+x0,y+y1]:=PenColor;
    end;
10: begin                    // 0 1 2
      if f<ca then Pixels[x+x0,y+y1]:=PenColor;
      if f>cb then Pixels[-x+x0,
         2*ry-y+y1]:=PenColor;
      Pixels[-x+x0,y+y1]:=PenColor;
    end;
11: begin                    // 1 2 3
      if f>ca then Pixels[-x+x0,y+y1]:=PenColor;
      if f<cb then Pixels[x+x0,
         2*ry-y+y1]:=PenColor;
      Pixels[-x+x0,2*ry-y+y1]:=PenColor;
    end;
12: begin                    // -3 -2 -1 0
      if f>ca then Pixels[-x+x0,y+y1]:=PenColor;
      if f>cb then Pixels[x+x0,y+y1]:=PenColor;
      Pixels[-x+x0,2*ry-y+y1]:=PenColor;
      Pixels[x+x0,2*ry-y+y1]:=PenColor;
    end;
13: begin                    // -2 -1 0 1
      if f<ca then Pixels[-x+x0,
         2*ry-y+y1]:=PenColor;
      if f<cb then Pixels[-x+x0,y+y1]:=PenColor;
      Pixels[x+x0,2*ry-y+y1]:=PenColor;
      Pixels[x+x0,y+y1]:=PenColor;
    end;
14: begin                    // -1 0 1 2
      if f>ca then Pixels[x+x0,
```

```
            2*ry-y+y1]:=PenColor;
          if f>cb then Pixels[-x+x0,
            2*ry-y+y1]:=PenColor;
          Pixels[x+x0,y+y1]:=PenColor;
          Pixels[-x+x0,y+y1]:=PenColor;
        end;
    15: begin                   // 0 1 2 3
          if f<ca then Pixels[x+x0,y+y1]:=PenColor;
          if f<cb then Pixels[x+x0,
            2*ry-y+y1]:=PenColor;
          Pixels[-x+x0,y+y1]:=PenColor;
          Pixels[-x+x0,2*ry-y+y1]:=PenColor;
        end;
    16: begin                   // -4 -3 -2 -1 0
          if f<=ca then Pixels[x+x0,y+y1]:=PenColor;
          if f>=cb then Pixels[x+x0,y+y1]:=PenColor;
          Pixels[-x+x0,y+y1]:=PenColor;
          Pixels[-x+x0,2*ry-y+y1]:=PenColor;
          Pixels[x+x0,2*ry-y+y1]:=PenColor;
        end;
    end; // case
  end;

begin
  PenColor:=PenColor;
  if x2<x1 then begin
    i:=x1; x1:=x2; x2:=i;
  end;
  if y2<y1 then begin
    i:=y1; y1:=y2; y2:=i;
  end;

  rx:=abs(x2-x1) div 2;
  ry:=abs(y2-y1) div 2;
  x0:=(x1+x2) div 2;
  y0:=(y1+y2) div 2;

  if (x3=x4) and (y3=y4) then n:=16
  else n:=Varnt(x3,y3,x4,y4);

  r2x:=Sqr(rx); r2y:=Sqr(ry);
  e:=ry/rx;     // эксцентриситет эллипса
  ca:=Abs(x3-x0)/Sqrt(sqr(x3-x0)+sqr(y3-y0));
  cb:=Abs(x4-x0)/Sqrt(sqr(x4-x0)+sqr(y4-y0));
  imin:=Trunc(rx/Sqrt(1+Sqr(e)));
```

```
  d1:=ry*(ry-r2x);
  j:=0; i:=0;
  while i<=imin do begin
    if d1<0
    then d2:=d1+(2*i+3)*r2y // без смещения по вертикали
    else begin
      d2:=d1+(2*i+3)*r2y-2*(ry-j-1)*r2x;// со смещением
                                        // по вертикали
      j:=j+1
    end;
    Symmetry(i,j);
    d1:=d2; i:=i+1;
  end;
  d1:=rx*(rx-r2y);
  j:=ry; i:=rx;
  while i>imin  do begin
    if d1<0
    then d2:=d1+(2*(ry-j)+3)*r2x // без смещения по горизонтали
    else begin
      d2:=d1+(2*(ry-j)+3)*r2x-2*(i-1)*r2y; // со
                            // смещением по горизонтали
      i:=i-1
    end;
    Symmetry(i,j);
    d1:=d2; j:=j-1;
  end;
end;
```

Эта процедура содержит ряд обобщений по сравнению с алгоритмом, представленным в листинге 6.2.

Во-первых, построение дуги окружности обобщается до построения дуги эллипса с полуосями `rx` и `ry`. Окружность рассматривается как частный случай эллипса.

Во-вторых, для задания дуги используются 4 пары целых чисел, задающих координаты двух концов диагонали охватывающего прямоугольника и координаты концов двух отрезков, исходящих из центра этого прямоугольника и ограничивающих дугу. Это позволяет произвольным образом выбирать концевые точки дуги. В частном случае совпадения концевых точек строится полный эллипс:

```
DrawElps(x1,y1,x2,y2,0,0,0,0,Canvas);
```

При построении дуги используется симметрия эллипса, а именно построение ведется в одном октанте, и для каждого очередного пиксела находятся соответствующие пикселы в других октантах. Такое отображение осуществляется вложенной процедурой `Simmetry`, в которой реализуется один из 16 возможных вариантов отображения в соответствии с 16 комбинациями пооктантного расположения концов дуги. Анализ вариантов производится во вложенной функции `Varnt`, которой в качестве аргументов передаются координаты точек, определяющих границы дуги.

Для вывода на полотно дуги эллипса обращение к процедуре `DrawElps` осуществляется из процедуры `DrawArc`, осуществляющей визуализацию дополнительных элементов — фокусного прямоугольника и ограничивающих дугу отрезков. Текст этой процедуры представлен в листинге 6.9.

Листинг 6.9. Обобщенная процедура построения дуги эллипса

```
procedure DrawArc(Canvas: TCanvas);
var i: integer;
begin
  with Canvas do begin
    case directa of
      1: DrawFocusRect(Rect(x1a,y2a,x2a,y1a));
      2: DrawFocusRect(Rect(x1a,y1a,x2a,y2a));
      3: DrawFocusRect(Rect(x2a,y2a,x1a,y1a));
      4: DrawFocusRect(Rect(x2a,y1a,x1a,y2a));
    end;
    for i:=0 to 1 do
    with Points[i] do begin
      Pen.Color:=clSilver;
      MoveTo(xa,ya); LineTo(Points[i].x,Points[i].y);
      Pen.Color:=clBlack;
      RectAngle(x-2,y-2,x+2,y+2);
    end;
  end;
DrawElps(x1a,y1a,x2a,y2a,Points[0].x,Points[0].y,
Points[1].x,Points[1].y,Canvas);
end;
```

В проекте реализован также алгоритм заливки цветом ограниченной области на канве (листинг 6.10).

Листинг 6.10. Процедура заливки цветом замкнутой области

```
procedure FillElps(X,Y: integer; Color: TColor; Canvas: TCanvas);
var
  P: TPoint;
  XL,XR,i,x1,x2: integer;

  procedure ShiftLeft(var x: integer; y: integer);
  // сдвиг левой границы до последней белой
  begin
    with Canvas do
    if Pixels[x,y]<>Color then // если x черная
      while (Pixels[x,y]<>Color) and (x<1200) do Inc(x)
    else begin                 // если x белая
```

```
      while (Pixels[x,y]=Color) and (x>0) do Dec(x);
      Inc(x);
    end;
  end;

  procedure ShiftRight(var x: integer; y: integer);
  // сдвиг правой границы до первой черной
  begin
    with Canvas do
    if Pixels[x,y]=Color then // если x белая
      while (Pixels[x,y]=Color) and (x<1200) do Inc(x)
    else begin                // если x черная
      while (Pixels[x,y]<>Color) and (x>0) do Dec(x);
      Inc(x);
    end;
  end;

begin
  InitStack(Stack);
  PushStack(Stack,Point(X,Y));
  with Canvas do
  while not StackIsEmpty(Stack) do begin
    P:=PopStack(Stack);

    XL:=P.X;
    while Pixels[XL,P.Y]=Color do begin
      if wb[P.Y mod 8] and (1 shl (XL mod 8))<>0 then
        Pixels[XL,P.Y]:=BrushColor
      else Pixels[XL,P.Y]:=clSilver;
      Dec(XL);
    end;           // XL - Black

    XR:=P.X+1;
    while Pixels[XR,P.Y]=Color do begin
      if wb[P.Y mod 8] and (1 shl (XR mod 8))<>0 then
        Pixels[XR,P.Y]:=BrushColor
      else Pixels[XR,P.Y]:=clSilver;
      Inc(XR);
    end;           // XR - Black

    x1:=XL+1; x2:=XR-1; // нижняя строка
    ShiftLeft(x1,P.Y+1);
    ShiftRight(x2,P.Y+1);
    for i:=x1 to x2-1 do
    if (Pixels[i,P.Y+1]=Color) and
```

```
       (Pixels[i+1,P.Y+1]<>Color) then
      PushStack(Stack,Point(i,P.Y+1));
    x1:=XL+1; x2:=XR-1; // верхняя строка
    ShiftLeft(x1,P.Y-1);
    ShiftRight(x2,P.Y-1);
    for i:=x1 to x2-1 do
    if (Pixels[i,P.Y-1]=Color) and (Pixels[i+1,
        P.Y-1]<>Color) then
      PushStack(Stack,Point(i,P.Y-1));
  end;
end;
```

В качестве аргументов в эту процедуру передаются координаты `(X,Y)` некоторой точки на канве и текущий цвет заливки `Color`. Идея алгоритма заключается в построчном сканировании области, включающей эту точку. Сканирование исходной строки осуществляется последовательным смещением на один пиксел влево от исходного до достижения пиксела с цветом, отличным от `Color`. Тем самым фиксируется левая граничная точка `XL` в этой строке. В процессе сканирования пикселы, окрашенные текущим цветом заливки, получают цвет, определяемый цветом кисти с учетом шаблона заливки:

```
while Pixels[XL,P.Y]=Color do begin
      if wb[P.Y mod 8] and (1 shl (XL mod 8))<>0 then
        Pixels[XL,P.Y]:=BrushColor
      else Pixels[XL,P.Y]:=clSilver;
```

Аналогичным образом производится сканирование исходной строки вправо от исходной точки и фиксируется правая граничная точка `XR`.

Далее производится смещение на одну строку вниз. В этой строке с помощью процедур `ShiftLeft` и `ShiftRight` ведется поиск новых граничных точек `XL` и `XR`. Затем просматривается диапазон пикселов строки в новых границах. В случае обнаружения внутри этого диапазона точки с цветом, отличным от исходного цвета, координаты такой точки заносятся в стек, в противном случае в стек заносится правая граница строки. Такая же процедура проделывается со строкой, находящейся над текущей.

Использование стека позволяет корректно выполнять заливку цветом при наличии внутри области любых границ.

6.4. Геометрические основы компьютерной графики

Для описания геометрических объектов, а также их взаимного расположения в пространстве, вводится система координат и каждой точке пространства сопоставляется набор вещественных чисел — координат этой точки. Число координат в та-

ком наборе определяет размерность пространства. Обычно рассматривают двумерные (2D) пространства на различных поверхностях и трехмерные (3D) пространства. В 2D-пространствах графическими элементами являются точки и линии, в 3D-пространствах к ним добавляются поверхности.

6.4.1. Графические элементы на плоскости

Простейшей формой поверхности является плоскость. Для описания геометрических объектов на плоскости используют *декартову* и *полярную* системы координат. Координаты (x, y) и (r, φ) в этих системах связаны соотношениями:

$$x = r\cos(\varphi), \quad y = r\sin(\varphi), \tag{6.21}$$

$$r = \sqrt{x^2 + y^2}, \quad tg(\varphi) = \frac{y}{x}. \tag{6.22}$$

Введем обозначение для точки на плоскости:

$$p = (x, y) \equiv (r, \varphi). \tag{6.23}$$

Взаимосвязь между координатами точек линии может быть задана в виде неявного уравнения $f(p) = 0$ или в виде параметрической функции $p(t)$. Эти соотношения могут быть записаны в координатной или в векторной форме. Векторная форма записи более компактна, однако в алгоритмах, как правило, заменяется более удобной для вычислений координатной формой.

Точка на плоскости имеет две степени свободы. Зависимость расстояния d между двумя точками p_1 и p_2 от их декартовых координат имеет вид:

$$d = |p_1 - p_2| = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}, \tag{6.24а}$$

а от полярных координат:

$$d = |p_1 - p_2| = \sqrt{r_1^2 + r_2^2 - 2r_1 \times r_2 \times \cos(\varphi_1 - \varphi_2)}. \tag{6.24б}$$

Линия на плоскости имеет одну степень свободы. Уравнения линии в неявной форме имеют вид:

$$f(x, y) = 0, \tag{6.25а}$$

а параметрическая функция —

$$p(t) = [x(t), y(t)]. \tag{6.25б}$$

Для прямой линии *неявное уравнение* имеет вид:

$$A \times x + B \times y + D = 0, \tag{6.26}$$

где хотя бы одно из чисел A или B должно быть отлично от нуля. В частности, если прямая задана координатами одной из своих точек p_0 и *вектором нормали*

$N = (N_x, N_y)$, то в этом случае неявное уравнение прямой записывается в *нормальной форме:*

$$N_x \times (x - x_0) + N_y \times (y - y_0) = 0. \tag{6.27}$$

Для задания прямой вместо вектора нормали можно использовать *направляющий вектор* $V = (V_x, V_y)$, направленный вдоль прямой. В этом случае для описания прямой удобно использовать *параметрическую функцию,* которая имеет вид:

$$x(t) = x_0 + V_x \times t, \quad y(t) = y_0 + V_y \times t. \tag{6.28}$$

Направляющий вектор $V = (V_x, V_y)$ начинается в точке p_0 и направлен в сторону увеличения t.

Из условия ортогональности векторов N и V следует, что

$$N_x \times V_x + N_y \times V_y = 0. \tag{6.29}$$

Сравнивая (6.26), (6.27) и (6.29), получаем для вектора нормали к прямой $N = (A, B)$, а для направляющего вектора $V = (-B, A)$.

Параметрическая функция удобна для построения частей прямой линии — отрезков и лучей. Для этого необходимо указать пределы изменения параметра:

- $-\infty < t < +\infty$ — протяженность прямой линии не ограничена;
- $t \ge 0$ — луч, выходящий из точки в направлении вектора V;
- $t_1 \le t \le t_2$ — отрезок прямой между точками $p_0 + V \times t_1$ и $p_0 + V \times t_2$.

Для произвольной линии на плоскости в любой регулярной (гладкой и некратной) точке $p_0 = (x_0, y_0) = p(t_0)$ возможна *линеаризация*, т. е. построение *касательной прямой.* Уравнение касательной удобно записать в нормальной форме (6.27), где компоненты вектора нормали вычисляются как частные производные от функции в левой части неявного уравнения (6.25а) этой линии:

$$N_x = \frac{df(x,y)}{dx}\Big|_{x_0,y_0}, \quad N_y = \frac{df(x,y)}{dy}\Big|_{x_0,y_0}. \tag{6.30а}$$

Вектор нормали $N = (N_x, N_y)$ ортогонален касательной и направлен в ту сторону, где $f(x, y) > 0$.

Если линия задана своей параметрической функцией, то аналогичная функция для касательной имеет вид (6.19) с компонентами направляющего вектора:

$$V_x = \frac{dx(t)}{dt}\Big|_{x_0,y_0}, \quad V_y = \frac{dy(t)}{dt}\Big|_{x_0,y_0}. \tag{6.30б}$$

Выбор между описанием линии с помощью уравнения или с помощью параметрических функций зависит от решаемой задачи. При построении линий удобнее ис-

пользовать их параметрическое представление либо, если это возможно, явную форму уравнения вида $y = f(x)$. С другой стороны, анализ параметров кривых, а также вычисление координат точек их пересечения удобно проводить с использованием явных и неявных уравнений. В целом параметрическое описание является более универсальным, и существует большой класс кривых, называемых параметрическими, для которых оно является единственно возможным.

В *главе 9* описан ряд проектов построения линий на плоскости с использованием явных и неявных уравнений, а также параметрического представления.

6.4.2. Графические элементы в пространстве

Наиболее часто геометрические объекты в пространстве описываются с использованием декартовой, цилиндрической и сферической систем координат. Соответственно, положение точки в пространстве может определяться любой тройкой вещественных чисел:

- x, y, z — в декартовой системе координат;
- r, φ, z — в цилиндрической системе координат;
- r, θ, φ — в сферической системе координат.

Расстояние d между двумя точками p_1 и p_2 вычисляется по формуле:

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2} \qquad (6.31а)$$

в декартовой системе координат,

$$d = \sqrt{r_1^2 + r_2^2 - 2r_1 \times r_2 \times \cos(\varphi_1 - \varphi_2) + (z_1 - z_2)^2} \quad — \qquad (6.31б)$$

в цилиндрической системе координат и

$$d = \sqrt{r_1^2 + r_2^2 - 2r_1 r_2 (\sin(\theta_1)\sin(\theta_2) + \cos(\theta_1)\cos(\theta_2)\cos(\varphi_1 - \varphi_2))} \qquad (6.31в)$$

в сферической СК.

Поверхность в пространстве имеет две степени свободы и ее можно описать с помощью неявного уравнения

$$f(x, y, z) = 0 \qquad (6.32)$$

или с помощью параметрической функции:

$$p(t, \tau) = (x(t, \tau), y(t, \tau), z(t, \tau)). \qquad (6.33)$$

Простейшей поверхностью является плоскость, для которой несложно записать неявное уравнение и параметрическую функцию.

В любой регулярной точке поверхности p_0 можно построить вектор нормали N с координатами:

$$N_x = \frac{df}{dx}\Big|_{x_0,y_0,z_0}, N_y = \frac{df}{dy}\Big|_{x_0,y_0,z_0}, N_y = \frac{df}{dz}\Big|_{x_0,y_0,z_0} \quad (6.34)$$

Неявное уравнение плоскости

Неявное уравнение плоскости задается четырьмя коэффициентами A, B, C, D:

$$A \times x + B \times y + C \times z + D = 0\,. \quad (6.35)$$

Хотя бы одно из чисел A, B или C должно быть отлично от нуля. Вектор нормали для плоскости равен $N = (A, B, C)$.

Нормальное уравнение плоскости, заданной точкой p_0 и вектором нормали N имеет вид:

$$N_x \times (x - x_0) + N_y \times (y - y_0) + N_z \times (z - z_0) = 0. \quad (6.36)$$

Параметрическая функция плоскости

Параметрическая функция плоскости, заданной точкой p_0 и линейно независимыми направляющимим векторами V и W, имеет вид:

$$\begin{aligned} x(t,\tau) &= x_0 + V_x \times t + W_x \times \tau, \\ y(t,\tau) &= y_0 + V_y \times t + W_y \times \tau, \\ z(t,\tau) &= z_0 + V_z \times t + W_z \times \tau. \end{aligned} \quad (6.37)$$

Такая форма удобна для задания, как всей плоскости, так и ее частей. Для этого достаточно указать пределы изменения параметров t и τ:

- $-\infty < t < +\infty$, $-\infty < \tau < +\infty$ — протяженность плоскости не ограничена;
- $-\infty < t < +\infty$, $\tau \geq 0$ — полуплоскость, находящаяся по одну сторону с вектором W от прямой $p_0 + V \times t$;
- $-\infty < t < +\infty$, $0 \leq \tau \leq 1$ — полоса шириной $|W|$ в направлении вектора V;
- $0 \leq t \leq 1$, $0 \leq \tau \leq 1$ — параллелограмм с вершинами в точках p_0, $p_0 + V$, $p_0 + W$, $p_0 + V + W$.

Вектор нормали к плоскости можно представить как векторное произведение векторов V и W:

$$N = [V \times W]. \quad (6.38)$$

Линии в пространстве

Линия в пространстве, как и на плоскости, имеет одну степень свободы и может быть описана либо как результат пересечения двух поверхностей, либо как траектория движения точки. В первом случае потребуется решение системы двух нелинейных уравнений $f_1(x, y, z) = 0$ и $f_2(x, y, z) = 0,$ что практически крайне неудобно. Во втором случае мы имеем дело с параметрическим представлением пространственных кривых:

$$p(t) = (x(t), y(t), z(t)). \tag{6.39}$$

Как уже отмечалось, параметрическое представление является более универсальным и удобным для проведения вычислений. При этом снимается ряд проблем, связанных с использованием многозначных функций в уравнениях для замкнутых кривых, а также с бесконечно большими значениями частных производных этих функций в некоторых точках кривых. В параметрическом представлении легко описываются замкнутые кривые и, кроме того, вместо тангенсов углов наклона кривых к координатным осям используются касательные векторы, которые никогда не бывают бесконечными (рис. 6.10).

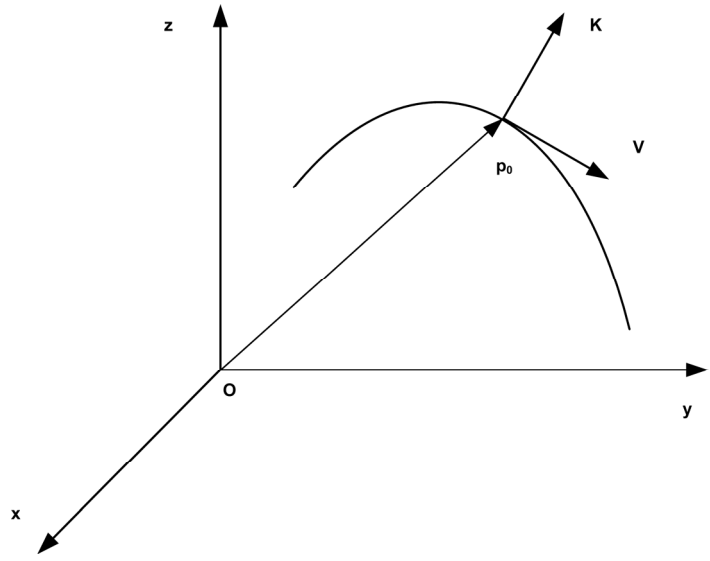


Рис. 6.10. Параметрическое описание пространственной кривой

В каждой регулярной точке $p_0 = p(t_0)$ (рис. 6.10) кривой L (первая производная $p'(t_0) <> 0$) можно вычислить единичный вектор касательной V и вектор кривизны K:

$$\begin{aligned} V &= p'(t)/|p'(t)|, \\ K &- [[p'(t) \times p''(t)] \times p'(t)]/|p'(t)|. \end{aligned} \tag{6.40}$$

Модуль вектора кривизны характеризует степень отклонения кривой L от прямой линии.

6.5. Задачи интерполяции, сглаживания и аппроксимации

При работе с линиями на плоскости или в трехмерном пространстве часто приходится решать одну из следующих трех задач:

- задачу интерполяции, состоящую в выборе линии, проходящей через заданное множество точек;
- задачу сглаживания — выбор линии, проходящей вблизи заданного множества точек;
- задачу аппроксимации — выбор линии, достаточно близкой к данной линии.

Для более строгой формулировки этих задач введем несколько понятий и обозначений.

Пусть F — функциональное пространство однозначных функций, в котором определена полунорма $\varphi, V \subset F$ — векторное функциональное пространство с базисом R^n. Обозначим через $\{\delta_i\}\, i \in I = 0, 1, ..., m$ множество функционалов, каждый из которых ставит в соответствие вектору $v \in V$ действительное число $\delta_i(v)$. Кроме того, пусть $\{z_i\}\ \ i \in I$ — множество точек, заданных на плоскости или в трехмерном пространстве. Введенные понятия позволяют дать следующие формулировки перечисленным задачам.

- Задача интерполяции: найти $v \in V$, такой, что $\delta_i(v) = z_i$ для любого $i \in I$.
- Задача сглаживания: найти $v \in V$, такой, что v близко к множеству $\{z_i\}$, т. е. что полунорма $\varphi(\{\delta_i(v)\} - \{z_i\}, i \in \mathrm{I})$ получает минимальное значение.
- Задача аппроксимации: найти $v \in V$, такой, что полунорма $\varphi(f - v)$ имеет минимальное значение. Здесь $f \in F$ — некоторая заданная функция.

Отметим, что сглаживание часто называют точечной аппроксимацией, подчеркивая близость двух последних задач.

6.5.1. Интерполяция полиномами

Рассмотрим функции $v(t) \in V = P^m$ над одномерным множеством $[a, b] \subset \Re$. Пусть $\delta_i(v) = v(t_i) = z_i$, $i \in I$, $z_i \in \Re$, $a <= t_0 <= t_1 <= ... <= t_m <= b$.

Если в качестве базиса R^n пространства V выбраны степенные функции $1, t, t^2,..., t^m$, то задача интерполяции сводится к решению линейной системы уравнений:

$$\sum_{j=0}^{m} v_j \times t_i^j = z_i \tag{6.41}$$

с симметричной обратимой матрицей, называемой матрицей Вандермонда.

Решение задачи интерполяции значительно упрощается, если в качестве базиса пространства V выбраны полиномы Лагранжа:

$$L_i(t) = \prod_{j=0, j\neq i} \frac{t - t_j}{t_i - t_j}, \quad i \in I. \tag{6.42}$$

В этом базисе решением является функция

$$v(t) = \sum_{i=0}^{m} z_i \times L_i(t). \tag{6.43}$$

В *главе 9* приводится пример программы, использующей интерполяционные многочлены Лагранжа.

Чаще всего в качестве базиса R^n выбираются полиномы Ньютона

$$N_0(t) = 1, \quad N_i(t) = \prod_{j=0}^{i} (t - t_j), \quad i \in I. \tag{6.44}$$

Решение системы уравнений

$$\sum_{j=0}^{m} v_j \times N_j(t_i) = z_i \tag{6.45}$$

записывается в рекуррентном виде

$$v_0 = z_0, \quad v_i = (z_i - \sum_{j=0}^{i-1} v_j N_j(t_i)) / N_i(t_i), \quad i \in I. \tag{6.46}$$

Величины v_i называются разделенными разностями порядка i на множестве $\{z_i\}$ $i \in I$.

Необходимо отметить, что и полиномы Лагранжа, и полиномы Ньютона могут давать большие отклонения от ожидаемых значений функции.

В случае если в точках $\{t_i\}$, $i \in I$ заданы не только значения функции $f(t_i)$, но и производные $f'(t_i)$, то число условий на интерполяционный многочлен удваивается, и он должен принадлежать пространству $V = P^{2m+1}$ полиномов степени не меньшей или равной $2m+1$.

Пусть $\{z_i\} \subset \Re$, $i = 0,1,...,2m+1$ и пусть $z_i = f(t_i)$, $z_{i+m+1} = f'(t_i)$, $i = 0,1,...,m$. Тогда решение задачи приводит к полиному Эрмита, выраженному через полиномы Лагранжа

$$v(t) = \sum_{i=0}^{m} z_i (1 - 2(t - t_i) L_i^{'}(t_i)) L_i^2(t) + \sum_{i=0}^{m} z_{i+m+1} (t - t_i) L_i^2(t) . \qquad (6.47)$$

Интерполяция полиномами может быть использована при небольшом числе точек (не более 15). Для большего числа точек растет степень полинома и неустойчивость линии, проявляющаяся в больших межузловых осцилляциях.

6.5.2. Интерполяция кубическими сплайнами

Проблема неустойчивости глобальных интерполирующих кривых решается путем перехода к кусочной (локальной) интерполяции полиномами невысоких степеней, называемых сплайнами. Коэффициенты сплайнов подбираются из условий достаточной гладкости результирующей кривой в точках сшивания. Минимальную интегральную степень кривизны интерполирующей кривой обеспечивают сплайновые полиномы третьей степени — кубические сплайны.

По-прежнему будем рассматривать функции $v(t) \in V = P^m$ над одномерным множеством $[a,b] \subset \Re$, $\delta_i(v) = z_i$, $i = 0,1,...,m$, $z_i \in \Re$, $a <= t_0 <= t_1 <= ... <= t_m <= b$. Интерполяционным кубическим сплайном называется функция $v(t)$, такая, что $v(t_i) = z_i, i = 0,1,...,m$; на любом интервале $[t_i, t_{i+1}]$, являющаяся многочленом третьей степени:

$$v_i(t) = \sum_{j=0}^{3} a_j^i (t - t_i)^j \qquad (6.48)$$

и дважды непрерывно дифференцируемая.

Построение кубического сплайна требует нахождения $4m$ коэффициентов a_{ij} $(i = 0,1,...,m;\ j = 0, 1, 2, 3)$ полиномов 3-й степени. Эти коэффициенты определяются системой линейных уравнений, которые получаются из условий непрерывности функции, а также ее первой и второй производных в $m-1$ промежуточных узлах, условия прохождения кривой через $m+1$ узел и двух условий в граничных узлах. Такая система линейных уравнений с трехдиагональной матрицей приводится в *главе 9* в рамках описания проекта приложения, реализующего построение графика сплайн-функции по заданному набору точек.

Существенным недостатком кубических сплайнов является то, что изменение координат хотя бы одной узловой точки приводит к необходимости полного пересчета всех коэффициентов a_{ij}. Этим недостатком не обладают некоторые методы сглаживания.

6.5.3. Сглаживание и аппроксимация

Одним из методов сглаживания является метод наименьших квадратов. Функция $v(t) \in V$ рассматривается над одномерным множеством $[a,b] \subset \Re$. Для решения задачи требуется минимизировать сумму квадратов отклонений функции $v(t)$ от заданных значений z_i, $i = 0,1,...,m$.

Кривые Безье

Другим способом сглаживания является использование кривой Безье, построенной на многочленах Бернштейна. Рассмотрим упорядоченный набор точек, определяемых множеством векторов $\{v_i\}$ $i \in I$. Ломаная линия $v_0 v_1 ... v_m$ называется контрольной ломаной, проходящей через эти точки.

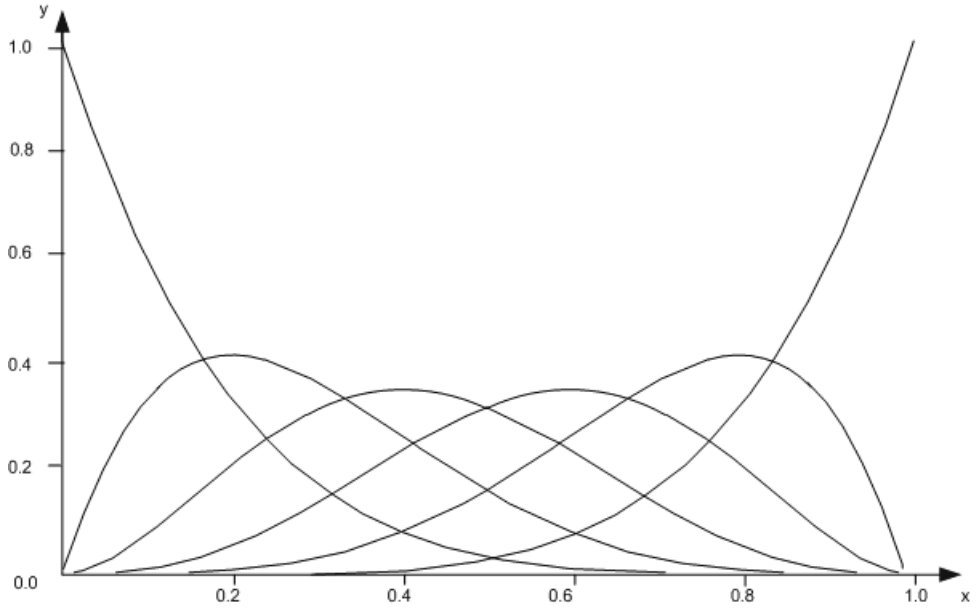


Рис. 6.11. Многочлены Бернштейна

Кривая Безье, порожденная множеством векторов $\{v_i\}$ — это линия, задаваемая векторной параметрической функцией вида:

$$r(t) = \sum_{i=0}^{m} C_m^i \times t^i \times (1-t)^{m-i} \times v_i, \quad 0 \le t \le 1, \tag{6.49}$$

где $C_m^i = \frac{m!}{i! \times (m-i)!}$ — биномиальные коэффициенты. Коэффициенты при вершинах v_i являются многочленами Бернштейна

$$C_m^i \times t^i \times (1-t)^{m-i}. \tag{6.50}$$

Многочлены Бернштейна обладают следующими свойствами:

- они определены на интервале [0, 1] и неотрицательны во всех его точках и при всех значениях параметров i и m ($i \le m$);
- их сумма равна 1

$$\sum_{i=0}^{m} C_m^i \times t^i \times (1-t)^{m-i} = (t + (1-t))^m = 1. \tag{6.51}$$

Для $m = 5$ многочлены Бернштейна представлены на рис. 6.11.

Кривая Безье является гладкой кривой, которая проходит через концевые точки v_0 и v_m контрольной ломаной и касается ее начальных отрезков (рис. 6.12). Таким образом, кривая Безье осуществляет сглаживание контрольной ломаной, заданной последовательностью векторов $\{v_i\}$.

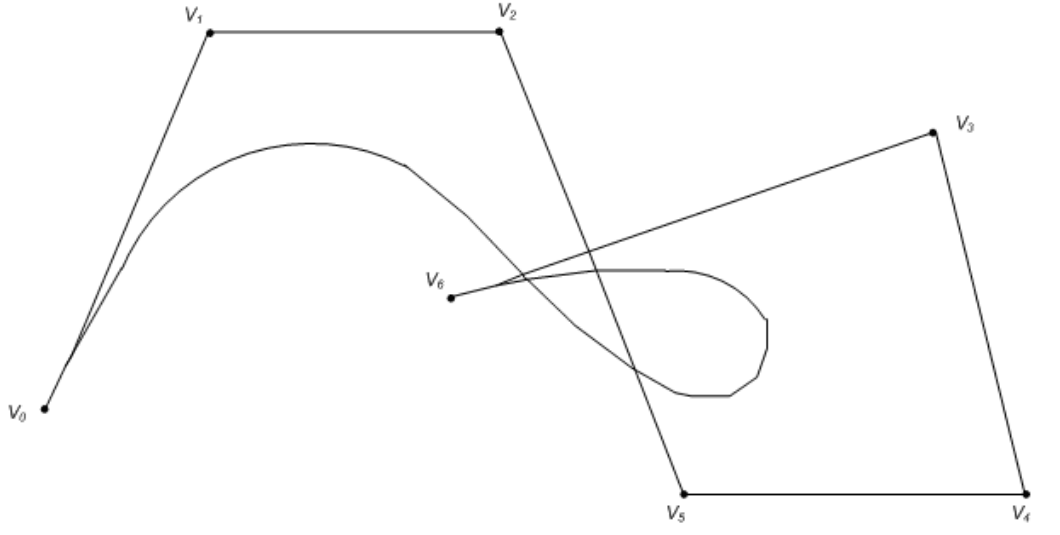


Рис. 6.12. Кривая Безье

Отметим, что при изменении координат любой точки v_i, необходимо заново строить параметрическую функцию (6.49). Это затрудняет внесение изменений с целью коррекции отдельных участков результирующей кривой.

B-сплайны

При построении сплайновых кривых в качестве базисных функций можно выбрать B-сплайны, по отношению к которым базис Бернштейна является частным случаем. Кривая, построенная на основе базиса B-сплайнов, определяются следующей векторной параметрической функцией:

$$r_q(t) = \sum_{i=0}^{m} N_{iq}(t) \times v_i, \tag{6.52}$$

где $N_{iq}(t)$ — нормализованные B-сплайны порядка q (степени $q-1$), которые определяются рекуррентным соотношением Кокса-де Бура (Cox-de Boor):

$$N_{i,q}(t) = \frac{t - t_i}{t_{i+q} - t_i} N_{i,q-1}(t) + \frac{t_{i+q+1} - t}{t_{i+q+1} - t_{i+1}} N_{i+1,q-1}(t),$$
$$N_{i,1}(t) = 1,\ t \in [t_i, t_{i+1}] \quad N_{i,1}(t) = 0,\ t \notin [t_i, t_{i+1}]. \tag{6.53}$$

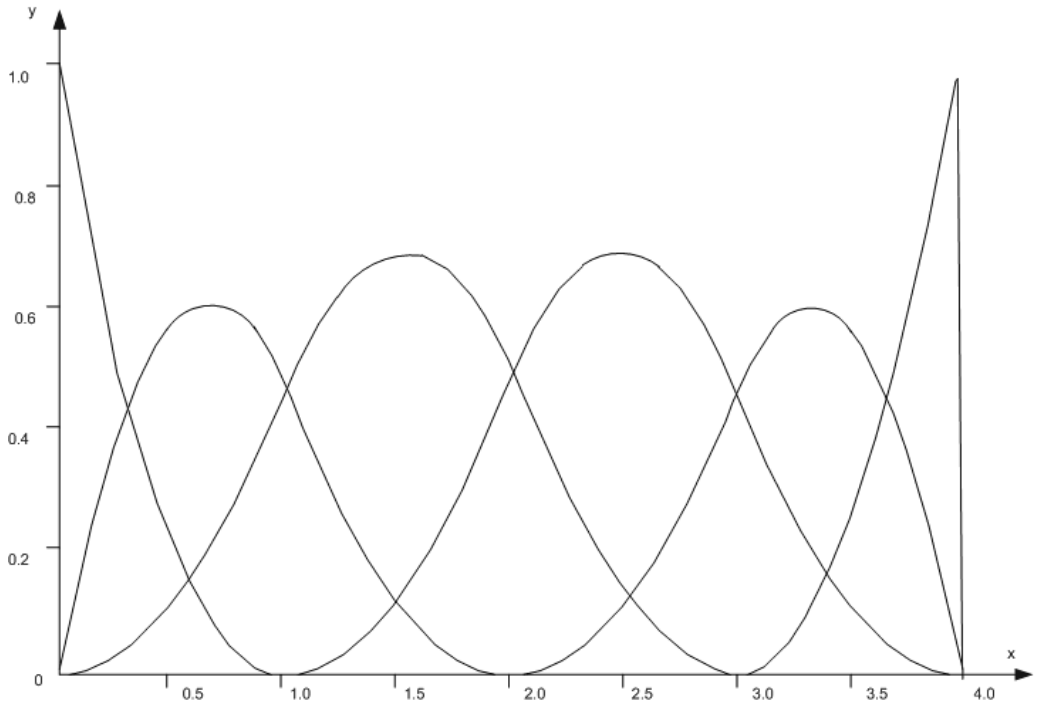


Рис. 6.13. B-сплайны четвертого порядка

Параметр t изменяется в интервале $[0, m-q+2]$, в котором размещается $[m+q+1]$ параметрический узел t_i. Как правило, для кривой порядка q в начале и в конце интервала изменения t ставятся узлы кратности q, так как при этом концы

кривой будут совпадать с крайними точками контрольной ломаной. Например, для кривой порядка $q = 3$ при числе контрольных точек $m + 1 = 6$ максимальное значение параметра t будет равно четырем, а число параметрических узлов — девяти. При этом $t_0 = t_1 = t_2 = 0$ и $t_6 = t_7 = t_8 = 4$. Остальные параметрические узлы могут быть размещены внутри интервала (0, 4) произвольным образом, однако удобно выбрать для них целочисленные значения.

В отличие от полиномов Бернштейна, которые существенно отличны от нуля всюду внутри интервала изменения параметра t, B-сплайны локализуются в окрестности одной из узловых точек, имея колоколообразную форму (bell — колокол). На рис. 6.13 представлены графики семи B-сплайнов четвертого порядка, построенных в интервале [0, 3] изменения параметра t.

Имеется еще одно существенное отличие B-сплайнов базисных кривых от кривых Безье. Оно заключается в том, что порядок полиномов Бернштейна всегда на единицу меньше числа контрольных точек, тогда как порядок B-сплайнов может быть любым в интервале [0, $m + 1$]. Кривые Безье — это частный случай B-сплайнов базисных кривых при $q = m$.

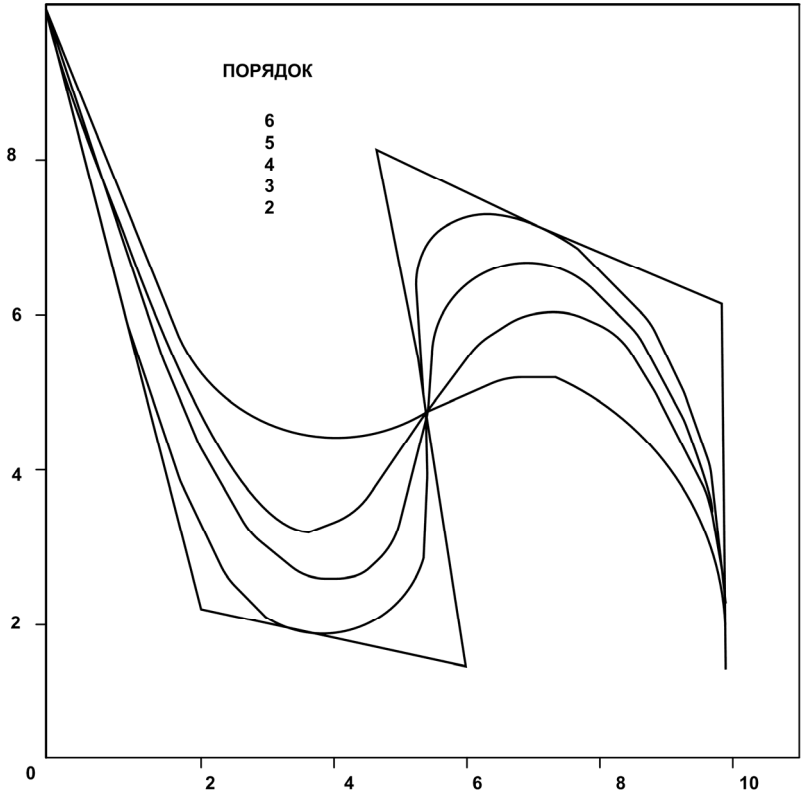


Рис. 6.14. B-сплайновые кривые различных порядков

На рис. 6.14 представлено несколько B-сплайнов — кривых различного порядка для контрольной ломаной с шестью вершинами. При этом кривая пятого порядка является кривой Безье для заданной ломаной, кривая четвертого порядка — кубическим сплайном, а кривая второго порядка — последовательностью связанных отрезков, совпадающих с исходной ломаной.

Таким образом, функции $N_{iq}(t)$ и определенная через них сплайновая кривая обладают следующими свойствами:

- являются полиномами степени $q-1$ ($N_{iq}(t) \in P_{q-1}$);
- для $q >= 3$ имеют непрерывные производные до порядка $q-2$ включительно ($N_{iq}(t) \in C^{q-2}[0,1]$);
- $\sum N_{iq}(t) = 1$, то есть кривая лежит в выпуклой оболочке $\{ri\}, i \in I$;
- сплайновая кривая касается начальных отрезков.

Сглаживание B-сплайнами обладает следующими дополнительными преимуществами по сравнению с кривыми Безье:

- обеспечивается возможность внесения локальных изменений без радикального изменения вида результирующей кривой;
- отсутствует жесткая связь между степенью сплайнового полинома и числом выбранных контрольных точек.

6.6. Аффинные преобразования координат

В компьютерной графике используются три системы координат:

- неподвижная мировая система координат (МСК);
- объектная система координат (ОСК), связанная с конкретным объектом и совершающая с ним все движения в МСК;
- экранная система координат (ЭСК), связанная с тем или иным графическим устройством.

Связь экранной системы координат с декартовой системой для двумерного плоского многообразия реального пространства рассматривалась ранее в *разд. 6.3.3*. Рассмотрим теперь связь между мировой и объектной системами, используя понятие аффинного преобразования координат.

6.6.1. Аффинные преобразования на плоскости

Наиболее просто такая связь описывается в простейшем случае двумерного пространства. На рис. 6.15 для случая плоского двумерного пространства представле-

ны мировая система координат XOY и связанная с некоторым объектом система координат X′O′Y′.

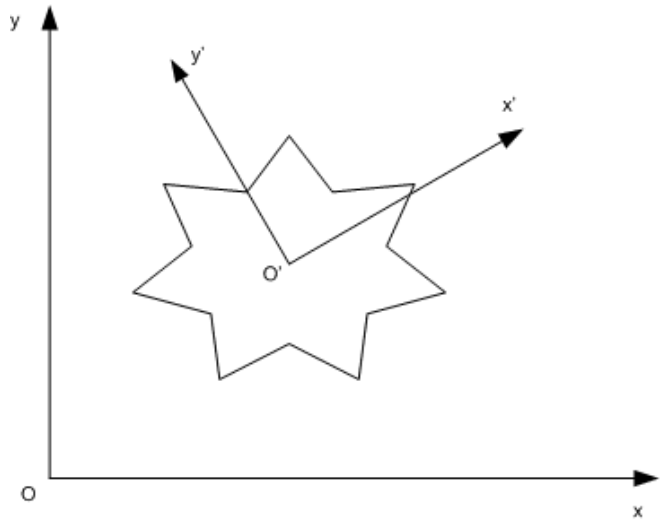


Рис. 6.15. Мировая и объектная системы координат на плоскости

Введем понятие *сцены* как системы объектов, изображение которой должно быть воспроизведено средствами компьютерной графики. Пусть некоторой точке P сцены в МСК соответствуют координаты (x, y), а в ОСК — координаты $(\bar{x}, \bar{y})$. Если угол поворота ОСК относительно МСК равен φ, а начало ОСК расположено в точке (x_0, y_0), то

$$\begin{aligned} \bar{x} &= (x - x_0) \times \cos\varphi + (y - y_0) \times \sin\varphi, \\ \bar{y} &= -(x - x_0) \times \sin\varphi + (y - y_0) \times \cos\varphi. \end{aligned} \tag{6.54}$$

Обратное преобразование имеет вид:

$$\begin{aligned} x &= \bar{x} \times \cos\varphi - \bar{y} \times \sin\varphi + x_0, \\ y &= \bar{x} \times \sin\varphi + \bar{y} \times \cos\varphi + y_0. \end{aligned} \tag{6.55}$$

В общем случае, переход от МСК к ОСК включает в себя два действия — поворот на угол φ и сдвиг в направлении вектора (x_0, y_0).

Преобразования (6.54)–(6.55) можно интерпретировать двояко:

- как изменение координат некоторой фиксированной точки сцены при изменении системы координат;
- как изменение точки сцены, находящейся в данной точке пространства, при использовании фиксированной системы координат.

В последнем случае можно говорить об отображении, переводящем точку сцены $(\bar{x}, \bar{y})$ в точку (x, y). Это отображение является линейным и может быть обобщено следующим образом:

$$\begin{aligned} x &= \alpha \times \bar{x} + \beta \times \bar{y} + \lambda, \\ y &= \gamma \times \bar{x} + \delta \times \bar{y} + \mu \end{aligned} \tag{6.56}$$

Коэффициенты преобразования должны быть связаны соотношением

$$\begin{vmatrix} \alpha & \beta \\ \gamma & \delta \end{vmatrix} \neq 0, \tag{6.57}$$

которое обеспечивает обратимость этого преобразования. Преобразование координат (6.56) называется *аффинным* (от латинского affinis — подобный), поскольку оно обеспечивает сохранение отношения подобия для геометрических элементов.

Аффинные преобразования удобно представлять в матричной форме. Для этого введем векторное представление для точек пространства:

$$p = (x, y). \tag{6.58}$$

Тогда произвольное аффинное преобразование можно представить в виде:

$$p = \bar{p}M + V, \tag{6.59}$$

где

$$M = \begin{pmatrix} \alpha & \gamma \\ \beta & \delta \end{pmatrix}, \quad V = (\lambda, \mu). \tag{6.60}$$

Можно строго доказать, что любое аффинное преобразование представимо в виде суперпозиции элементарных преобразований — поворота, растяжения, отражения и переноса.

Преобразование переноса

$$\begin{aligned} x &= \bar{x} + \lambda, \\ y &= \bar{y} + \mu. \end{aligned} \tag{6.61а}$$

В векторной форме:

$$p = \bar{p} + V. \tag{6.61б}$$

Преобразование растяжения

$$\begin{aligned} x &= \alpha \times \bar{x}, \\ y &= \delta \times \bar{y}. \\ (a &> 0, \delta > 0). \end{aligned} \tag{6.62а}$$

В матричной форме:

$$p = \bar{p}\begin{pmatrix} \alpha & 0 \\ 0 & \delta \end{pmatrix}. \tag{6.62б}$$

Преобразование отражения относительно оси абсцисс

$$\begin{aligned} x &= \bar{x}, \\ y &= -\bar{y}. \end{aligned} \tag{6.63а}$$

В матричной форме:

$$p = \bar{p}\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{6.63б}$$

Преобразование поворота

$$\begin{aligned} x &= \bar{x} \times \cos\varphi - \bar{y} \times \sin\varphi, \\ y &= \bar{x} \times \sin\varphi + \bar{y} \times \cos\varphi. \end{aligned} \tag{6.64а}$$

В матричной форме:

$$p = \bar{p}\begin{pmatrix} \cos\varphi & \sin\varphi \\ -\sin\varphi & \cos\varphi \end{pmatrix}. \tag{6.64б}$$

Нетрудно видеть, что преобразованию переноса соответствует аддитивный член в формуле (6.59), тогда как остальные преобразования выполняются мультипликативно.

Чтобы сделать аффинное преобразование однородным, включить в него преобразование переноса мультипликативным образом, в проективной геометрии вводятся понятия *расширенного пространства* и *однородных координат* точки. В основе этих понятий лежит представление о том, что каждая точка в n-мерном пространстве может рассматриваться как проекция точки из $(n + 1)$-мерного пространства при фиксированном значении $(n + 1)$-й координаты. Для достижения однородности аффинных преобразований это значение следует выбрать равным 1.

Однородными координатами точки p двумерного пространства, имеющей декартовы координаты (x, y), называется такая тройка чисел x_1, x_2, u, что

$$x = \frac{x_1}{u},\ y = \frac{x_2}{u} \tag{6.65}$$

и $u \neq 0$. При $u = 1$ вектор точки p в однородных координатах имеет вид:

$$p(x, y, 1). \tag{6.66}$$

В общем случае $u \neq 1$ точке p в двумерном пространстве соответствует *эквивалентный вектор* (рис. 6.16)

$$p' = (ux, uy, u). \tag{6.67}$$

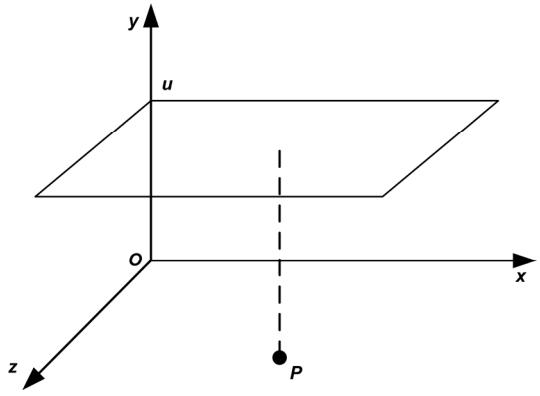


Рис. 6.16. Однородные координаты
в двумерном пространстве как проекция из трехмерного пространства

В однородных координатах любое аффинное преобразование записывается в виде:

$$p = \overline{p}M, \tag{6.68}$$

где M — комбинация матриц элементарных преобразований одного из следующих типов:

- поворота (rotation) на угол φ вокруг начала координат с матрицей

$$M \equiv R = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}; \tag{6.69}$$

- растяжения (dilatation) вдоль осей OX и OY с матрицей

$$M \equiv D = \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & 1 \end{pmatrix}; \tag{6.70}$$

- отражения (reflection), например, относительно оси OX, с матрицей

$$M \equiv F = \begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}; \tag{6.71}$$

- переноса (translation) с матрицей переноса

$$M \equiv T = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \lambda & \mu & 1 \end{pmatrix}. \tag{6.72}$$

Приведем примеры использования однородных координат для выполнения аффинных преобразований.

Пример 1: построить матрицу поворота вокруг точки $P(a, b)$ на угол φ. Рассмотрим шаги выполнения данного примера:

1-й шаг: перенос на вектор $P = (-a, -b)$ для совмещения центра поворота с началом координат.

$$T_{-P} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 1 \end{pmatrix}; \tag{6.73а}$$

2-й шаг: поворот на угол φ.

$$R_{\varphi} = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{pmatrix}; \tag{6.73б}$$

3-й шаг: перенос на вектор $P = (a, b)$ для возвращения центра поворота в прежнее положение.

$$T_{P} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{pmatrix}. \tag{6.73в}$$

Перемножая матрицы, получим:

$$M = T_{-P} \times R_{\varphi} \times T_{P} = \begin{pmatrix} \cos\varphi & \sin\varphi & 0 \\ -\sin\varphi & \cos\varphi & 0 \\ -a\times\cos\varphi + b\times\sin\varphi + a & -a\times\sin\varphi - b\times\cos\varphi + b & 1 \end{pmatrix}. \tag{6.74}$$

Пример 2: построить матрицу растяжения с коэффициентами растяжения α вдоль оси абсцисс, δ вдоль оси ординат и с центром в точке $P = (a, b)$.

1-й шаг: перенос на вектор $P = (-a, -b)$ для совмещения центра поворота с началом координат.

$$T_{-P} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -a & -b & 1 \end{pmatrix}; \tag{6.75а}$$

2-й шаг: растяжение вдоль координатных осей с коэффициентами α и δ соответственно.

$$D = \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ 0 & 0 & 1 \end{pmatrix}; \qquad (6.75б)$$

3-й шаг: перенос на вектор $P = [a, b]$ для возвращения центра поворота в прежнее положение.

$$T_P = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ a & b & 1 \end{pmatrix}. \qquad (6.75в)$$

Перемножая матрицы, получим

$$M = T_{-P} \times D \times T_P = \begin{pmatrix} \alpha & 0 & 0 \\ 0 & \delta & 0 \\ (1-\alpha)\times a & (1-\delta)\times b & 1 \end{pmatrix}. \qquad (6.76)$$

6.6.2. Аффинные преобразования в пространстве

В трехмерном пространстве (3D) положение точки может быть задано в однородных координатах $p(x, y, z, 1)$ или с помощью эквивалентного вектора $p'(ux, uy, uz, u)$.

Любое аффинное преобразование в 3D-пространстве, также как и в пространстве двумерном, можно представить в виде суперпозиции операций поворота, растяжения, отражения и переноса. Далее выписаны матрицы элементарных преобразований с использованием однородных координат.

Матрицы вращения

Вращение вокруг оси абсцисс на угол φ:

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (6.77а)$$

Вращение вокруг оси ординат на угол ψ:

$$R_y = \begin{pmatrix} \cos\psi & 0 & -\sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (6.77б)$$

Вращение вокруг оси аппликат на угол χ:

$$R_z = \begin{pmatrix} \cos\chi & \sin\chi & 0 & 0 \\ -\sin\chi & \cos\chi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (6.77в)$$

Матрица растяжения (сжатия)

$$D = \begin{pmatrix} \alpha & 0 & 0 & 0 \\ 0 & \beta & 0 & 0 \\ 0 & 0 & \gamma & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad (6.78)$$

$\alpha > 0$ — коэффициент растяжения (сжатия) вдоль оси абсцисс,

$\delta > 0$ — коэффициент растяжения (сжатия) вдоль оси ординат,

$\gamma > 0$ — коэффициент растяжения (сжатия) вдоль оси аппликат.

Матрицы отражения

Отражение относительно плоскости XOY:

$$M_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (6.79а)$$

Отражение относительно плоскости YOZ:

$$M_x = \begin{pmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (6.79б)$$

Отражение относительно плоскости XOZ:

$$M_y = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (6.79в)$$

Матрица переноса

$$T = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \lambda & \mu & \nu & 1 \end{pmatrix}, \qquad (6.80)$$

где $V = (\lambda, \mu, \nu)$ — вектор переноса.

Примеры выполнения аффинных преобразований

Рассмотрим несколько примеров выполнения аффинных преобразований в 3D-пространстве.

Пример 1: построить матрицу поворота на угол α вокруг прямой L, проходящей через точку $P(a, b, c)$ и имеющую единичный направляющий вектор (l, m, n).

1-й шаг: перенос на вектор $-P = (-a, -b, -c)$.

$$T_{-P} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -a & -b & -c & 1 \end{pmatrix}. \qquad (6.81а)$$

После этого прямая L будет проходить через начало координат.

2-й шаг: совмещение оси аппликат с прямой L. Достигается двумя последовательными поворотами:

Первый поворот выполняется вокруг оси абсцисс до совмещения оси аппликат с проекцией прямой на плоскость YOZ. Проекция прямой L на эту плоскость имеет направляющий вектор $(0, m, n)$, поэтому для угла поворота φ справедливы следующие соотношения:

$$\cos(\varphi) = \frac{n}{d}, \quad \sin(\varphi) = \frac{m}{d}, \qquad (6.81б)$$

где $d = \sqrt{m^2 + n^2}$.

Тогда для матрицы первого поворота получим:

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{n}{d} & \frac{m}{d} & 0 \\ 0 & \frac{-m}{d} & \frac{n}{d} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (6.81в)$$

Под действием этого преобразования изменятся координаты направляющего вектора прямой. Новый вектор будет равен

$$(l, \quad m, \quad n, \quad 1) \times R_x = (l, \quad 0, \quad d, \quad 1). \qquad (6.81г)$$

Второй поворот выполняется вокруг оси ординат до совмещения прямой с осью аппликат. Для этого угла поворота ψ справедливы следующие соотношения:

$$\cos(\psi) = d, \quad \sin(\psi) = -l. \qquad (6.81д)$$

Соответствующая матрица вращения запишется в виде:

$$R_y = \begin{pmatrix} d & 0 & l & 0 \\ 0 & 1 & 0 & 0 \\ -l & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (6.81е)$$

Теперь направляющий вектор прямой будет иметь координаты

$$(l, \quad 0, \quad d, \quad 1) \times R_y = (0, \quad 0, \quad 1, \quad 1). \qquad (6.81ж)$$

3-й шаг: вращение вокруг прямой L на заданный угол α. Так как теперь прямая совпадает с осью аппликат, то соответствующая матрица имеет вид:

$$R_z = \begin{pmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (6.81з)$$

4-й шаг: поворот вокруг оси ординат на угол $-\psi$.

5-й шаг: поворот вокруг оси абсцисс на угол $-\varphi$. Порядок вращения имеет значение, так как преобразование вращения не коммутативно.

6-й шаг: перенос на вектор $P(a, b, c)$.

Результирующая матрица получается в результате перемножения семи матриц:

$$A = T_{-P} \times R_x(\varphi) \times R_y(\psi) \times R_z(\alpha) \times R_y(-\psi) \times R_x(-\varphi) \times T_P. \qquad (6.82)$$

Пример 2: требуется подвергнуть заданному аффинному преобразованию выпуклый многогранник.

Выпуклый многогранник однозначно задается набором своих вершин $V_i = (x_i, y_i, z_i), \quad i = 1, 2, ..., n.$ Построим матрицу

$$V = \begin{pmatrix} x_1 & y_1 & z_1 & 1 \\ x_2 & y_2 & z_2 & 1 \\ . & . & . & . \\ x_n & y_n & z_n & 1 \end{pmatrix}. \tag{6.83}$$

Умножая ее на матрицу A заданного аффинного преобразования, получим набор вершин преобразованного многогранника.

$$V' = V \times A. \tag{6.84}$$

6.7. Проецирование

Проецирование — это отображение трехмерного объекта на двумерную плоскость, называемую *картинной плоскостью* (КП). Изображение, которое получается на картинной плоскости, называется *проекцией объекта* на эту плоскость.

Построение проекции осуществляется методом трассировки лучей. Из некоторой точки, называемой *центром проецирования (проектором)*, проводятся лучи через точки объекта. Точка пересечения луча или его продолжения с картинной плоскостью является проекцией (действительной или мнимой) соответствующей точки объекта. Очевидно, что, в общем случае, луч проходит через некоторое множество точек объекта, отображая все это множество в одну-единственную точку на картинной плоскости. Поэтому преобразование проецирования связано с потерей части информации о геометрических свойствах объекта и, следовательно, является необратимым. Однако при удачном выборе способа выполнения проективного преобразования полученное изображение субъективно может восприниматься как вполне реалистичное.

Из сказанного следует, что при решении задачи проецирования, кроме законов оптики в общем случае необходимо учитывать субъективное восприятие человеческим мозгом изображения, попадающего на сетчатку глаза. Отклонение математических проекций от изображений, воспринимаемых человеческим глазом, подмечено художниками давно, но лишь в последнее время предпринимаются попытки получить эмпирические формулы, описывающие эти отклонения.

Для упрощения дальнейших рассуждений будем считать, что проецируемый объект является многогранником и что некоторые из его граней параллельны координатным плоскостям ОСК.

На рис. 6.17 представлены наиболее часто используемые в геометрии, черчении и машинной графике способы проецирования (виды проекций).

Проекции можно разделить на два основных класса: центральные и параллельные. Различия между ними определяются способом выбора положения проектора. Если проектор размещается на конечном расстоянии от картинной плоскости, то имеет место центральное проецирование. При бесконечно большом удалении проектора от картинной плоскости выполняется параллельное проецирование.

Рис. 6.17. Виды проецирования

Центральная проекция позволяет достичь определенной степени реальности (эффект перспективного укорачивания), но непригодна для представления точной формы и размеров объекта: из нее нельзя получить информацию о расстояниях и углах.

Параллельная проекция дает менее реалистичное изображение, при этом истинные размеры и углы сохраняются только для тех граней объекта, которые параллельны картинной плоскости. Параллельные проекции разделяют на два типа в зависимости от соотношения между направлением проецирующих лучей и нормалью к картинной плоскости. В ортографических и аксонометрических проекциях лучи направляются вдоль нормали к картинной плоскости, а в косоугольных проекциях под углом к ней. В последнем случае угол между направлением на проектор и нормалью к картинной плоскости называется углом проецирования.

6.7.1. Ортографическое проецирование

Различие между ортографическим и аксонометрическим проецированием заключается в выборе способа ориентации объектной системы координат (ОСК) относительно МСК. При ортографическом проецировании оси ОСК направлены параллельно соответствующим осям МСК. Аксонометрическое проецирование выполняется при произвольной ориентации ОСК относительно МСК.

Математически проецирование реализуется путем умножения радиус-векторов точек объекта на *матрицу проецирования.* Такая матрица является вырожденной, что соответствует необратимому характеру проективного преобразования.

Обычно картинную плоскость выбирают параллельной одной из координатных плоскостей МСК (либо совпадающей с ней). По этому признаку проекции делятся на следующие классы:

профильные (вид сбоку), когда КП параллельна плоскости YOZ мировой системы координат либо совпадает с ней;

горизонтальные (вид сверху), когда КП параллельна плоскости ZOX мировой системы координат либо совпадает с ней;

фронтальные (вид спереди), когда КП параллельна плоскости XOY мировой системы координат либо совпадает с ней.

Матрицы профильного, горизонтального и фронтального ортографического проецирования имеют вид (6.85а–6.85в), где p, q, r — смещения картинных плоскостей относительно соответствующих координатных плоскостей МСК (рис. 6.18).

$$[P_x] = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ p & 0 & 0 & 1 \end{pmatrix}, \qquad (6.85а)$$

$$[P_y] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & q & 0 & 1 \end{pmatrix}, \qquad (6.85б)$$

$$[P_z] = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & r & 1 \end{pmatrix}. \qquad (6.85в)$$

Достоинство ортографического проецирования заключается в возможности точной передачи линейных и угловых размеров объектов, благодаря чему оно находит широкое применение в таких областях, как машиностроительное черчение и архитек-

тура. В то же время при ортографическом проецировании полностью теряется информация об одном из пространственных измерений объекта. Восстановление трехмерного образа объекта возможно лишь при совместном использовании трех видов.

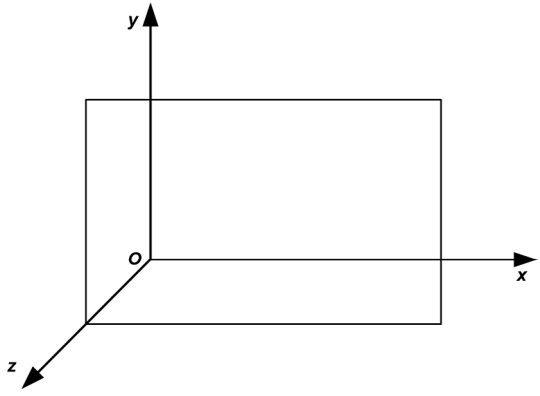


Рис. 6.18. Ортографическая проекция

6.7.2. Аксонометрическое проецирование

Аксонометрическое проецирование в отличие от ортографического позволяет частично воспроизвести трехмерную форму объекта. Для этого объект должен быть повернут так, чтобы его грани не были параллельны картинной плоскости. Однако обычно наиболее простое математическое описание объекта получается в предположении, что оси его ОСК параллельны соответствующим осям МСК. Поэтому аксонометрическая проекция должна строиться в картинной плоскости, которая некоторым произвольным образом ориентирована относительно осей МСК.

На рис. 6.19 показаны три угла — α, β и γ, определяющие ориентацию картинной плоскости относительно МСК.

Различают три вида аксонометрических проекций:

- триметрическая проекция — нормальный вектор картинной плоскости образует с ортами координатных осей попарно различные углы;
- диметрическая проекция — два из трех указанных углов равны;
- изометрическая проекция — все углы равны.

Задача аксонометрического проецирования объекта на картинную плоскость решается в два этапа.

- Выполняется поворот ОСК до совмещения направления нормали N к картинной плоскости с одной из координатных осей МСК.

- Выполняется ортографическое проецирование на картинную плоскость, параллельную (совпадающую) с соответствующей координатной плоскостью МСК.

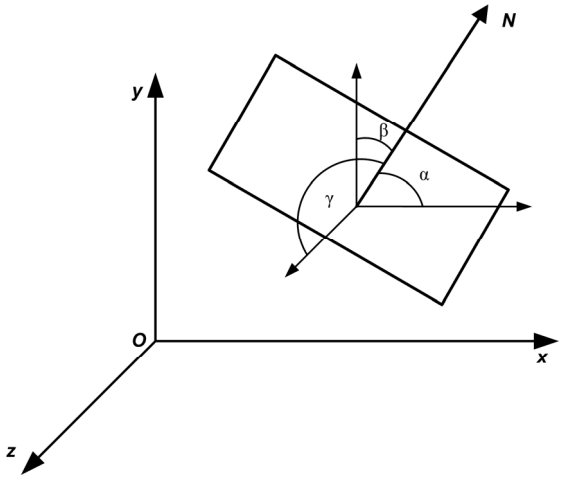


Рис. 6.19. Аксонометрическая проекция

В дальнейшем, если не оговорено иное, мы будем рассматривать фронтальные проекции объектов, а в качестве картинной плоскости выбирать плоскость XOY мировой системы координат. В этом случае для выполнения 1-го этапа проецирования необходимо произвести поворот ОСК на угол ψ вокруг оси OY мировой системы координат. Это обеспечит совмещение вектора N с плоскостью YOZ. Затем производится поворот ОСК на угол φ вокруг оси OX, после которого вектор N окажется направленным вдоль оси OZ МСК. С учетом последующего ортографического проецирования полная матрица преобразования имеет вид:

$$M = \begin{pmatrix} \cos\psi & 0 & -\sin\psi & 0 \\ 0 & 1 & 0 & 0 \\ \sin\psi & 0 & \cos\psi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & \sin\varphi & 0 \\ 0 & -\sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

После перемножения матриц получим:

$$M = \begin{pmatrix} \cos\psi & \sin\varphi \times \sin\psi & 0 & 0 \\ 0 & \cos\varphi & 0 & 0 \\ \sin\psi & -\sin\varphi \times \cos\psi & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (6.86)$$

Элемент M_{ij} этой матрицы равен проекции *i*-го орта ОСК на *j*-ю ось МСК. В общем случае триметрического проецирования углы поворотов ψ и φ выбираются произвольным образом.

Для дальнейшего рассмотрения удобно ввести *коэффициенты осевых искажений*:

$$\begin{aligned} m_x &= \sqrt{(M_{11})^2 + (M_{12})^2}, \\ m_y &= \sqrt{(M_{21})^2 + (M_{22})^2}, \\ m_z &= \sqrt{(M_{31})^2 + (M_{32})^2}, \end{aligned} \tag{6.87}$$

которые являются проекциями ортов ОСК на картинную плоскость (плоскость XOY МСК). В соответствии с приведенной ранее классификацией аксонометрических проекций при триметрическом проецировании эти коэффициенты должны быть попарно различны.

В диметрических проекциях два из трех коэффициентов осевых искажений равны друг другу. Это приводит к взаимной зависимости углов ψ и φ. В зависимости от способа выбора пары равных коэффициентов, получим разные соотношения между углами, а именно:

$$\sin^2 \psi = \mathrm{tg}^2 \varphi, \tag{6.88а}$$

при $m_x = m_y \neq m_z$,

$$\cos^2 \psi = \mathrm{tg}^2 \varphi, \tag{6.88б}$$

при $m_y = m_z \neq m_x$, и, наконец,

$$\sin^2 \psi = \cos^2 \psi, \tag{6.88в}$$

при $m_x = m_z \neq m_y$.

В стандартной диметрической проекции соотношение коэффициентов искажения составляет

$$m_x : m_y : m_z = 2 : 2 : 1. \tag{6.89}$$

Выражая коэффициенты осевых искажений через углы поворота и учитывая равенство $m_y{}^2 = 4m_z{}^2$, получим дополнительное условие:

$$\cos^2 \varphi = 4(\sin^2 \psi + \sin^2 \varphi \cos^2 \psi), \tag{6.90}$$

которое совместно с (6.88) образует систему двух уравнений для углов ψ и φ. В случае, когда единичный вектор нормали к картинной плоскости лежит в первом октанте МСК, имеем $\varphi > 0$ и $\psi < 0$.

Тогда решение системы уравнений дает следующие значения для углов поворота:

$$\varphi = \arccos(\sqrt{\frac{8}{9}}) \approx 19{,}5^\circ, \psi = -\arccos(\sqrt{\frac{7}{8}}) \approx -20{,}7^\circ. \quad (6.91)$$

Матрица стандартного диметрического преобразования для этого случая имеет вид:

$$M = \begin{pmatrix} 0{,}935 & -0{,}118 & 0 & 0 \\ 0 & 0{,}943 & 0 & 0 \\ -0{,}354 & -0{,}312 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (6.92)$$

Изометрические проекции получаются при условии равенства коэффициентов осевых искажений $m_x = m_y = m_z$, что приводит к равенствам:

$$\begin{aligned} &\cos^2\psi + \sin^2\varphi \times \sin^2\psi = \cos^2\varphi, \\ &\sin^2\psi + \sin^2\varphi \times \cos^2\psi = \cos^2\psi + \sin^2\varphi \times \sin^2\psi. \end{aligned} \quad (6.93)$$

Откуда следует, что

$$\sin^2\varphi = \frac{1}{3}, \sin^2\psi = \frac{1}{2}. \quad (6.94)$$

Стандартная изометрическая проекция получается в предположении, что, как и в предыдущем случае, единичный вектор нормали к картинной плоскости лежит в 1-м октанте МСК. Тогда

$$\varphi = \arccos(\sqrt{\frac{2}{3}}) \approx 35{,}3^\circ, \psi = -\arccos(\sqrt{\frac{1}{2}}) \approx -45^\circ, \quad (6.95)$$

и матрица преобразования имеет вид:

$$M = \begin{pmatrix} 0{,}707 & -0{,}408 & 0 & 0 \\ 0 & 0{,}816 & 0 & 0 \\ -0{,}707 & -0{,}408 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (6.96)$$

6.7.3. Косоугольное проецирование

При косоугольном проецировании пучок проецирующих лучей направлен под произвольным углом к картинной плоскости, а сама эта плоскость, как и при фронтальном ортографическом проецировании, параллельна фронтальной плоскости МСК. В этом случае удобно ввести косоугольную ОСК, у которой оси абсцисс и ординат параллельны соответствующим осям МСК, а ось аппликат имеет направ-

ление, противоположное направлению пучка проецирующих лучей. Тогда матрица косоугольного фронтального проецирования будет аналогична соответствующей матрице ортографического проецирования, отличаясь от нее ненулевыми значениями проекций 3-го орта ОСК на оси OX и OY мировой СК:

$$[K]=\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -c_x & -c_y & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{6.97}$$

где $c_x = e_{3x}/e_{3z}$, $c_y = e_{3y}/e_{3z}$. Коэффициенты осевых искажений (6.39) при косоугольном проецировании оказываются равными:

$$m_x = m_y = 1, m_z = \sqrt{c^2{}_x + c^2{}_y} = tg\alpha, \tag{6.98}$$

где α — угол между направлением проецирующего пучка и нормалью к картинной плоскости, т. е. между осями аппликат объектной и мировой СК.

Таким образом, косоугольные проекции сочетают в себе свойства ортографических проекций (видов сбоку, сверху и спереди) со свойствами аксонометрии. В частности, сторона объекта, параллельная картинной плоскости, проецируется без искажения пропорций для углов и расстояний. Косоугольное проецирование применяется при построении теней, а также отражений в плоском зеркале.

Различают две стандартных косоугольных проекции: *свободную* (называемую также *военной* или *кавальерной*) и *кабинетную.* Первая получается при условии $m_x = m_y = m_z = 1$. Это соответствует углу наклона проецирующих прямых к картинной плоскости, равному $\pi/4$. Кабинетной проекции соответствуют значения коэффициентов осевых искажений, равные $m_x = m_y = 1$ и $m_z = 0{,}5$. При этом угол наклона проецирующих прямых равен

$\alpha = arctg(0{,}5) \approx 26{,}6^\circ$.

6.7.4. Центральное проецирование

Обсуждение центральных проекций начнем с построения фронтальной проекции отдельной точки. Рассмотрим произвольную точку P с координатами (x, y, z). Поместим центр проецирования в точку $S(0, 0, s)$ и проведем из нее луч через точку P. Обозначим через P^* точку пересечения луча с плоскостью XOY (рис. 6.20). Из простых геометрических соображений следует, что координаты этой точки равны: $P^*(x/(1-z/s),\quad y/(1-z/s), 0)$.

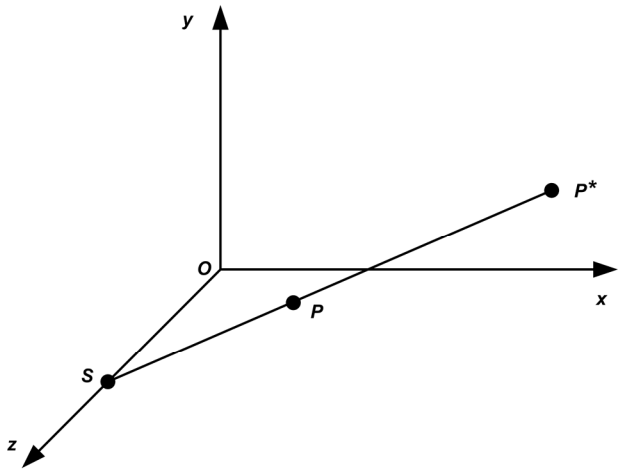


Рис. 6.20. Центральное проецирование

Этот же результат можно получить, выполнив следующее преобразование однородного вектора точки $P(x, y, z, 1)$ на матрицу центрального проецирования:

$$C_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1/s \\ 0 & 0 & 0 & 1 \end{pmatrix}. \qquad (6.99)$$

Результатом этого преобразования является эквивалентный вектор

$$(x, y, 0, 1-\frac{z}{s}),$$

которому соответствует однородный вектор

$$(x/(1-\frac{z}{s}), y/(1-\frac{z}{s}), 0, 1). \qquad (6.100)$$

Таким образом, задача центрального проецирования точки решается в два этапа:

- однородный вектор координат точки умножается на матрицу центрального проецирования;
- полученный эквивалентный вектор координат преобразуется в соответствующий однородный вектор.

Центральное проецирование обладает важным свойством, благодаря которому оно получило свое второе название — перспективное. Это свойство состоит в следующем: пучок параллельных прямых при центральном проецировании на картинную плоскость переходит в сходящийся пучок. Точка, в которой пересекаются проекции параллельных прямых, называется *точкой схода.*

Примечание

Пучок прямых не должен быть параллелен картинной плоскости.

Продемонстрируем это свойство на примере прямых, параллельных оси OZ. Параметрическое представление такой прямой в однородных координатах имеет вид:

$$p(t) = (x_0, y_0, z_0 + vt, 1). \tag{6.101}$$

Выполнив преобразование центрального проецирования с матрицей (6.99), получим:

$$p^*(t) = (\frac{x_0}{1 - \frac{z_0 + vt}{s}}, \frac{y_0}{1 - \frac{z_0 + vt}{s}}, 0, 1). \tag{6.102}$$

Устремляя значение параметра t к бесконечности, получим для точки схода:

$$p_\infty(0, 0, 0, 1). \tag{6.103}$$

Таким образом, центральные проекции пучка прямых, параллельных оси OZ, при размещении проектора на этой оси имеют точку схода в начале координат (рис. 6.21).

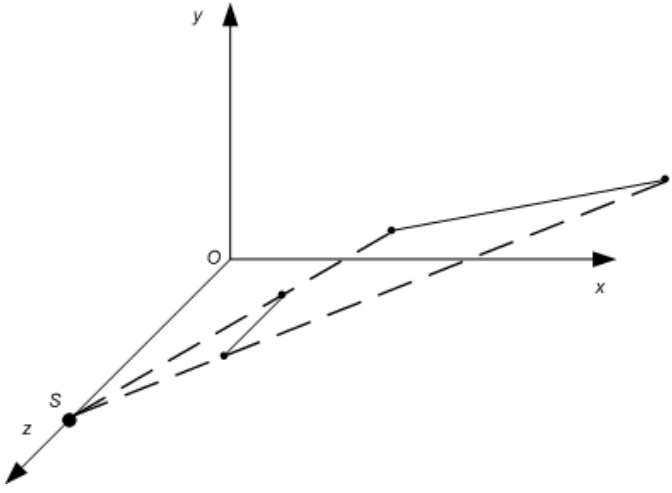


Рис. 6.21. Центральная проекция отрезка, параллельного оси OZ

Можно отделить перспективное преобразование от проецирования, введя матрицу вида:

$$Q = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1/s \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{6.104}$$

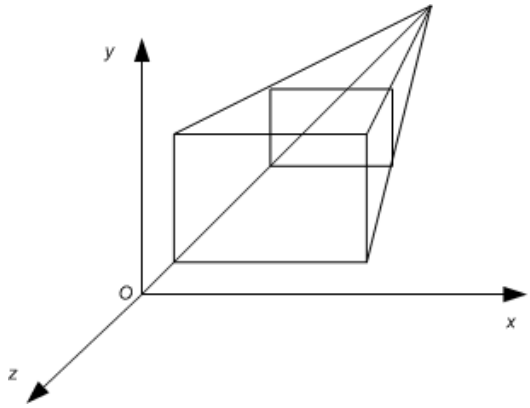


Рис. 6.22. Положение точки схода на оси OZ

Эта матрица преобразует вектор (6.101) к виду:

$$p^*(t) = \left(\frac{x_0}{1-\frac{z_0+vt}{s}}, \frac{y_0}{1-\frac{z_0+vt}{s}}, \frac{z_0+vt}{1-\frac{z_0+vt}{s}}, 1\right),$$

что в пределе $t \to \infty$ дает для вектора координат точки схода значение $(0, 0, -s, 1)$ (рис. 6.22).

В общем случае произвольного положения проектора S и произвольного направления пучка параллельных прямых, задаваемого вектором V, образующим ненулевой угол с фронтальной картинной плоскостью, координаты точки схода определяются вектором

$$p_\infty = S - \frac{s_z}{V_z} V. \tag{6.105}$$

Матрица фронтального центрального проецирования в общем случае имеет вид:

$$C_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -s_x/s_z & -s_y/s_z & 0 & -1/s_z \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{6.106}$$

Аналогичным образом могут быть определены матрицы для горизонтального центрального проецирования

$$C_h = \begin{pmatrix} 1 & 0 & 0 & 0 \\ -s_x/s_y & 0 & -s_z/s_y & -1/s_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{6.107}$$

и профильного центрального проецирования

$$C_z = \begin{pmatrix} 0 & -s_y/s_x & -s_z/s_x & -1/s_x \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \quad (6.108)$$

6.7.5. Проект "Проекции"

Проект "Проекции" приведен на компакт-диске в папке **Примеры | Глава 6 | Проекции**. Для иллюстрации различных способов проецирования в предлагаемом проекте рассматривается создание простого приложения, в котором строятся фронтальные проекции тетраэдра и куба. В этом приложении реализована возможность выбора способа проецирования.

В приложении используются две формы (рис. 6.23): `FormMain` (с внешним именем **`Проекции`**), на которой строится изображение, и `FormTools` (с внешним именем **`Tools`**) — панель инструментов, используемая для задания опций. Форма `FormTools` содержит две вкладки — **Виды проекций** и **Действия.**

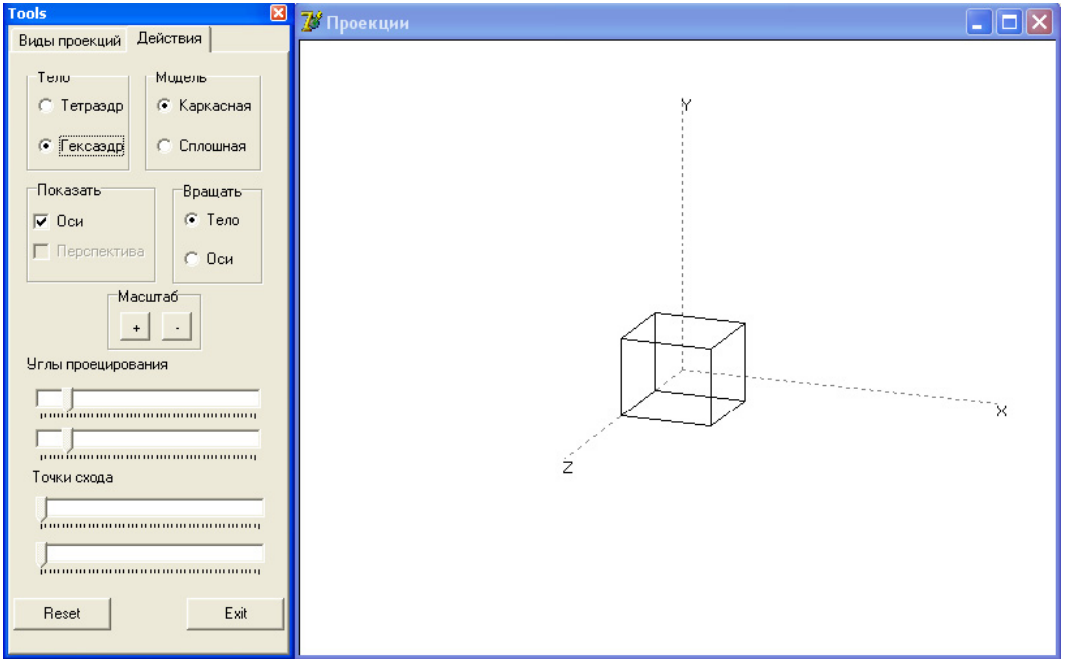


Рис. 6.23. Приложение для построения фронтальных проекций

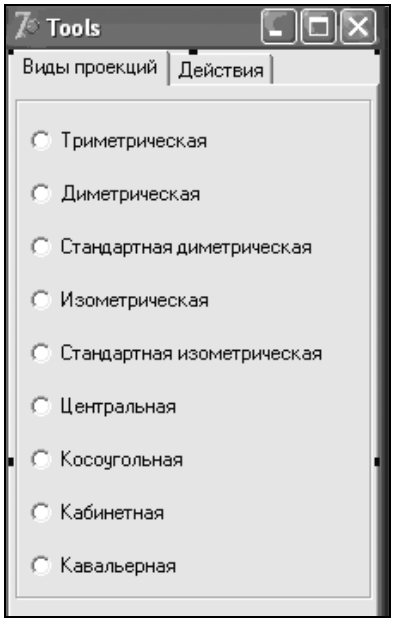


Рис. 6.24. Вкладка **Виды проекций** формы **Tools**

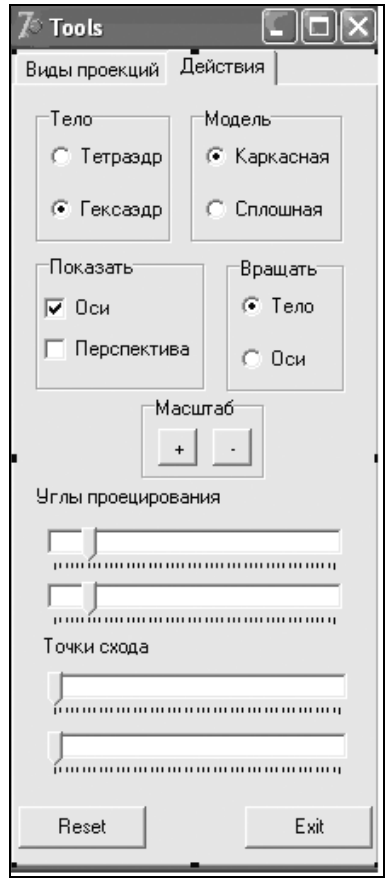


Рис. 6.25. Вкладка **Действия** формы **Tools**

На вкладке **Виды проекций** (рис. 6.24) размещен набор радиокнопок, предназначенных для выбора одного из девяти способов проецирования. Элементы управления, размещенные на вкладке **Действия** (рис. 6.25), позволяют пользователю произвести выбор:

- изображаемого тела — тетраэдра или гексаэдра (куба);
- модели трехмерного объекта — каркасной или сплошной;
- способа вращения — вращение тела или координатных осей.

Кроме того, имеется возможность:

- производить изменение масштаба изображения (группа кнопок **Масштаб**);
- по желанию пользователя показывать или скрывать изображение координатных осей и/или точек схода (группа флажков **Показать**);
- задавать с помощью ползунков положение точек схода при центральном проецировании;
- задавать с помощью ползунков углы проецирования для косоугольных проекций.

Кнопка Reset предназначена для восстановления опций, заданных по умолчанию, а кнопка Exit — для завершения работы приложения.

Основные структуры данных, используемые в проекте, описаны в библиотечном модуле Lib.pas. Определены следующие типы данных:

для задания однородных координат радиус-вектора точки

```
TVector = array[1..4] of Real ;
```

для хранения матрицы преобразования

```
TMatrix = array[1..4] of Tvector
```

Информации о многогранниках сохраняется в структурах следующих типов данных:

координаты вершин

```
TVertex=record
  x,y,z: real;
end;
```

грани

```
TSide =record
  p: array of word; // номера вершин
  A,B,C,D: single;  // коэффициенты уравнения плоскости
  N: TVector;       // вектор нормали к плоскости
end;
```

ребра

```
TEdge=record
  p1,p2: word;                  // номера вершин
end;
```

Многогранники задаются записями вида:

```
TBody=record
  Vertexs  : array of TVector;  // массив начальных
                                // положений вершин тела
  VertexsT : array of TVector;  // массив текущих
                                // положений вершин тела
  Edges    : array of TEdge;    // массив ребер тела
  Sides    : array of TSide;    // массив граней тел
end;
```

Модуль `UnitMain` включает процедуры построения изображения, обработки событий, связанных с манипулятором мышь, а также ряд вспомогательных процедур и функций. Обработчики событий `OnMouseDown` и `OnMouseUp` выполняют, соответственно, установку и снятие флага рисования (переменная `drawing`). Обработчик `FormMouseMove` содержит код, имитирующий вращение тела при перемещении пользователем указателя мыши. Код процедуры показан в листинге 6.11.

Листинг 6.11. Обработчик события перемещения указателя мыши

```
procedure TFormMain.FormMouseMove(Sender: TObject; Shift: TShiftState; X,Y:
Integer);
var a, b: real;
begin
  if drawing then begin
  a:=X-Width div 2; b:=Y-Height div 2;
  case FormTools.RgRotate.ItemIndex of
    1: case FormTools.RgPro.ItemIndex of
       0,5: begin
              Alf:=ArcTan2(b,a);
              Bet:=Sqrt(Sqr(a/10)+Sqr(b/10));
              end;
         1: begin
              Bet:=Sqrt(Sqr(a/10)+Sqr(b/10));
              Alf:=ArcTan(Sin(Bet));
            end;
         3: begin
              if fl_xy then begin
                Bet:=Bet+Pi/2;
                if Bet>1.5*Pi then begin
                  Bet:=Bet-2*Pi;
                  fl_xy:=false;
                end;
              end
              else begin
                case Ind of
                  1,3: Alf:=-Alf;
                  2: Alf:=Pi+Alf;
                end;
                Inc(Ind);
                if Ind=4 then begin
                  Alf:=Alf+Pi;
                  Ind:=1;
                end;
                fl_xy:=true;
              end; // else
            end; // 3:
       end; // case
    0: begin          // вращать тело
         Alf1:=ArcTan2(b,a);
         Bet1:=Sqrt(Sqr(a/10)+Sqr(b/10));
       end;
  end; // case
```

```
  Draw;
  end; // if
end;
```

В зависимости от выбора опции вращения на панели инструментов имитируется либо вращение тела при неподвижных координатных осях (`RgRotate.ItemIndex=0`), либо совместное вращение тела и координатных осей (`RgRotate.ItemIndex=1`). В первом случае координатные оси являются осями МСК, во втором — объектной СК. Углы поворота осей ОСК `Alf` и `Bet`, а также углы `Alf1` и `Bet1` поворота тела относительно осей МСК вычисляются через текущие координаты X и Y указателя мыши. Способ вычисления углов `Alf` и `Bet` зависит от выбранной проекции.

Стандартные проекции предполагают выбор определенных ориентаций тела относительно координатных осей, поэтому для них вращение не производится.

Основной процедурой этого модуля является процедура построения изображения `Draw`, текст которой приведен в листинге 6.12.

Листинг 6.12. Процедура построения изображения

```
procedure TFormMain.Draw;
const
  ax=7; // длина полуоси
var
  i,j,L,L0,L1,u,v: integer;
  Vt,NN,V1,V2: TVector;
  Vn,Wn: array[0..2] of TVector;
  P0,P: TPoint;
  W: array of TPoint;
  M: TMatrix;
begin
  with Bitmap.Canvas, Body do begin
    Brush.Color:=clWhite;
    Pen.Color:=clBlack;
    Pen.Style:=psSolid;
    Rectangle(0,0,Width,Height);
    Pen.Mode:=pmCopy;
    L:=Length(Vertexs);
     // повороты и проецирование для всех вершин
    for i:=0 to L-1 do begin
      VertexsT[i]:=Vertexs[i];
      case FormTools.RgPro.ItemIndex of
        0..5:begin
               VertexsT[i]:=Rotate(VertexsT[i], 2,
                 Bet1, 0, 0);// поворот тела вокруг OY
               VertexsT[i]:=Rotate(VertexsT[i], 1,
                 Alf1, 0, 0);// поворот тела вокруг OX
               VertexsT[i]:=Rotate(VertexsT[i], 4,
```

```
                0, Xs, Zs);
              VertexsT[i]:=Rotate(VertexsT[i], 2,
                 Bet, 0, 0); // поворот осей вокруг OY
              VertexsT[i]:=Rotate(VertexsT[i], 1,
                 Alf, 0, 0); // поворот осей вокруг OX
            end;
        6..8:begin
              VertexsT[i]:=Rotate(VertexsT[i], 2,
                 Bet1, 0, 0);// поворот тела вокруг OY
              VertexsT[i]:=Rotate(VertexsT[i], 1,
                 Alf1, 0, 0);// поворот тела вокруг OX
              VertexsT[i]:=Rotate(VertexsT[i], 5, 0,
                 q, r);
            end;
      end;
    end;
// прорисовка осей
    Pen.Color:=clGray;
    Pen.Style:=psDot;
    Brush.Style:=bsClear;
    if FormTools.ChkBoxAx.Checked then begin
      P0:=IJ(ToVector(0,0,0));
      P:=IJ(ToVector(ax,0,0));
      MoveTo(P0.X,P0.Y); LineTo(P.X,P.Y);
      TextOut(P.X,P.Y,'X');
      P:=IJ(ToVector(0,ax,0));
      MoveTo(P0.X,P0.Y); LineTo(P.X,P.Y);
      TextOut(P.X,P.Y,'Y');
      P:=IJ(ToVector(0,0,ax));
      MoveTo(P0.X,P0.Y); LineTo(P.X,P.Y);
      TextOut(P.X,P.Y,'Z');
    end;
    Brush.Style:=bsSolid;
    // изображение куба перспективы
    if FormTools.ChkBoxPersp.Checked then begin
      with Body0 do begin
        L:=Length(Vertexs);
        for i:=0 to L-1 do begin
          VertexsT[i]:=Vertexs[i];
          VertexsT[i]:=Rotate(VertexsT[i], 2, Bet1, 0, 0);
          VertexsT[i]:=Rotate(VertexsT[i], 1, Alf1, 0, 0);
          VertexsT[i]:=Rotate(VertexsT[i], 4, 0, Xs, Zs);
          VertexsT[i]:=Rotate(VertexsT[i], 2, Bet, 0, 0);
          VertexsT[i]:=Rotate(VertexsT[i], 1, Alf,
             0, 0);
        end;
```

```
      L0:=Length(Edges);
      for i:=0 to L0-1 do begin
        MoveTo(II(VertexsT[Edges[i].p1][1]),
          JJ(VertexsT[Edges[i].p1][2]));
        LineTo(II(VertexsT[Edges[i].p2][1]),
          JJ(VertexsT[Edges[i].p2][2]));
      end;
    end;
    // изображение точки схода Xs
    if Xs<>0 then begin
      V1[2]:=1/Xs; V1[1]:=0; V1[3]:=0; V1[4]:=1;
      V1:=Rotate(V1, 2, Bet, 0, 0);
      V1:=Rotate(V1, 1, Alf, 0, 0);
      u:=II(V1[1]); v:=JJ(V1[2]);
      for i:=0 to 5 do
      if i in [0,1,4,5] then begin
        MoveTo(u,v);
        LineTo(II(Body0.VertexsT[i][1]),
          JJ(Body0.VertexsT[i][2]));
      end;
      Pen.Style:=psSolid;
      Pen.Color:=clBlack;
      RectAngle(u-2,v-2,u+2,v+2);
    end;
    // изображение точки схода Zs
    if Zs<>0 then begin
      Pen.Color:=clGray;
      Pen.Style:=psDot;
      V2[1]:=0; V2[2]:=0; V2[3]:=1/Zs; V2[4]:=1;
      V2:=Rotate(V2, 2, Bet, 0, 0);
      V2:=Rotate(V2, 1, Alf, 0, 0);
      u:=II(V2[1]); v:=JJ(V2[2]);
      for i:=0 to 3 do begin
        MoveTo(u,v);
        LineTo(II(Body0.VertexsT[i][1]),
          JJ(Body0.VertexsT[i][2]));
      end;
      // изображение тела
      Pen.Style:=psSolid;
      Pen.Color:=clBlack;
      RectAngle(u-2,v-2,u+2,v+2);
    end;
  end;
  Pen.Color:=clBlack;
  Pen.Style:=psSolid;
```

```
    case FormTools.RgModel.ItemIndex of
      0: begin     // каркасная модель
           L:=Length(Edges);
           for i:=0 to L-1 do begin
           MoveTo(II(Body.VertexsT[Body.Edges[i].p1]
             [1]),JJ(VertexsT[Edges[i].p1][2]));
           LineTo(II(VertexsT[Edges[i].p2][1]),
             JJ(VertexsT[Edges[i].p2][2]));
           end;
         end;
      1: begin   // сплошная модель
           Brush.Color:=clSilver;
           Pen.Color:=clSilver;
           L1:=Length(Sides);
           L0:=Length(Sides[0].p);
           SetLength(w,L0);
           // перебор граней
           for i:=0 to L1-1 do begin
             for j:=0 to L0-1 do begin
               Vt:=Vertexs[Sides[i].p[j]];
               Vt:=Rotate(Vt, 1, Alf1,0,0);
               Vt:=Rotate(Vt, 2, Bet1,0,0);
               if j<=2 then Vn[j]:=Vt;
               Vt:=Rotate(Vt, 4,   0,Xs,Zs);
               Vt:=Rotate(Vt, 1, Alf,0,0);
               Vt:=Rotate(Vt, 2, Bet,0,0);
               w[j].X:=II(Vt[1]);
               w[j].Y:=JJ(Vt[2]);
               if j<=2 then Wn[j]:=Vt;
             end;
             Sides[i].N:= Norm(Vn[0],Vn[1],Vn[2]);
             if Sides[i].N[3]>=0 then
               Brush.Color:=Trunc(255*Sides[i].N[3])*$010000
             else
               Brush.Color:=Trunc(-255*Sides[i].N[3]/3)*$010000;
             Pen.Color:=Brush.Color;
             NN:= Norm(Wn[0],Wn[1],Wn[2]);
             if Norm(Wn[0],Wn[1],Wn[2])[3]<0 then
               Polygon(w);
           end; // перебор граней
         end;
    end; // case: каркасная/сплошная
  end;  // with Bitmap.Canvas, Body
  Canvas.Draw(0,0,Bitmap);
end;
```

Построение изображения тела в этой процедуре разбивается на ряд этапов.

Для всех вершин тела выполняется операция поворота и проецирования. Исходные значения координат копируются в массив `VertexsT` и преобразуются с помощью функции `Rotate`. В качестве аргументов этой функции передаются следующие величины: координаты точки, индекс преобразования `k`, параметры преобразования `fi`, `p`, `r`. Функция содержит внутреннюю процедуру `Matr`, формирующую матрицу преобразования в соответствии с заданными параметрами. После вызова этой процедуры производится перемножение вектора, задающего положение вершины, и матрицы преобразования. В листинге 6.13 показано преобразование координат вершин с использованием процедуры `Rotate`.

Листинг 6.13. Функция преобразования координат вершин

```
function Rotate(var V: TVector; k: Integer; fi,p,r: Real): TVector;
  procedure Matr;
  begin
    M:=I;
    case k of
      1: begin  // матрица поворота вокруг оси OX
           M[2,2]:= cos(fi); M[2,3]:=sin(fi);
           M[3,2]:=-sin(fi); M[3,3]:=cos(fi);
         end;
      2: begin  // матрица поворота вокруг оси OY
           M[1,1]:=cos(fi); M[1,3]:=-sin(fi);
           M[3,1]:=sin(fi); M[3,3]:=cos(fi);
         end;
      3: begin // матрица поворота вокруг оси OZ
           M[1,1]:= cos(fi); M[1,2]:=sin(fi);
           M[2,1]:=-sin(fi); M[2,2]:=cos(fi);
         end;
      4: begin // перспективное преобразование
           M[2,4]:= p; M[3,4]:=r;
         end;
      5: begin // косоугольное проецирование
           M[3,1]:=p; M[3,2]:=r;
         end;
    end;
  end; // Matr

begin
  Matr; // формирование матрицы преобразования
  Result:=VM_Mult(V,M);//перемножение вектора и матрицы
end;
```

Следующим этапом построения изображения является прорисовка координатных осей. В этой части процедуры `TFormMain.Draw` пространственные координаты начальной и конечных точек отрезков, изображающих координатные оси, отображаются в экранные координаты. Преобразование осуществляется в два этапа. Сначала функция `ToVector` преобразует пространственные координаты в однородные. Затем функция `IJ` (листинг 6.14) выполняет проецирование на картинную плоскость XOY и преобразование "бумажных" координат в экранные.

Листинг 6.14. Вычисление экранных координат точки

```
function IJ(Vt: TVector): TPoint;
begin
  case FormTools.RgPro.ItemIndex of
    0..5:begin  // Аксонометрические и центральная проекции
           Vt:=Rotate(Vt, 2, Bet, 0, 0);
           Vt:=Rotate(Vt, 1, Alf, 0, 0);
         end;
    6..8:Vt:=Rotate(Vt, 5, 0, q, r);  // Косоугольные проекции
  end;
  Result.x:=II(Vt[1]);
  Result.Y:=JJ(Vt[2]);
end;
```

Если пользователь выбрал опцию изображения перспективы, производится построение точек схода. Кроме того, для наглядности штриховыми линиями строится изображение куба с учетом перспективы. Разумеется, этот куб невидим, если в качестве многогранника выбран гексаэдр.

В рассматриваемом проекте строится изображение тела в одной из двух моделей — каркасной или сплошной. В первом случае построение сводится к проведению отрезков, соответствующих ребрам многогранника. Во втором случае для каждой грани строится вектор нормали и производится заливка грани тем или иным цветом в зависимости от ее ориентации относительно оси OZ.

6.8. Моделирование трехмерных тел

В компьютерной графике используется несколько типов моделей пространственных объектов, классификация которых представлена на рис. 6.26.

Выбор той или иной модели предопределяет следующие способы решения таких проблем:

- ❒ выделение видимых частей объекта;
- ❒ назначение цвета каждому элементу объекта;
- ❒ воспроизведение различных оптических эффектов.

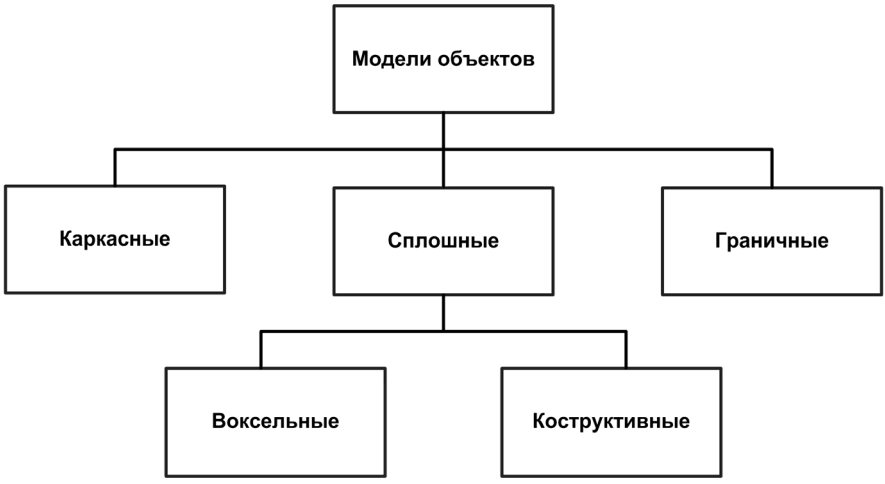


Рис. 6.26. Классификация моделей пространственных объектов

6.8.1. Каркасные модели

В простейшем варианте каркасной модели тело изображается в виде каркасной сетки с целью передачи формы охватывающих его поверхностей — граней. В случае многогранников достаточно построения ребер. В общем случае для построения каркасной модели тела используются различные способы кусочно-линейной интерполяции. В *главе 10* описан проект, реализующий примеры построения каркасных моделей поверхностей второго порядка.

В более сложных вариантах каркасной модели передается эффект разноудаленности от наблюдателя отдельных граней объекта, путем выделения особым стилем (например, пунктиром) или удаления невидимых ребер. Еще большая реалистичность изображения достигается при использовании заливки граней, когда оттенок цвета зависит от пространственной ориентации грани.

Для построения каркасной модели тела необходимо:

- ❒ пронумеровать его вершины, ребра и грани;
- ❒ задать координаты всех вершин относительно начала координат или некоторой базовой точки;
- ❒ описать топологию каркаса путем формирования для каждой из граней списка номеров инцидентных ей вершин, перечисленных в порядке их обхода в определенном едином для всех граней направлении.

На рис. 6.27, 6.28 и 6.29 представлены основные составляющие каркасной модели: поверхность тела (рис. 6.27), грани (рис. 6.28), ребра (рис. 6.29).

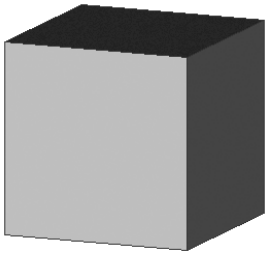


Рис. 6.27. Поверхность тела, образованная гранями

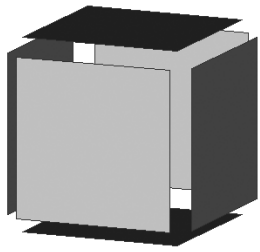


Рис. 6.28. Грани, ограниченные ребрами

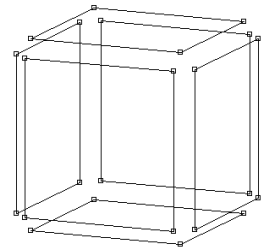


Рис. 6.29. Ребра, соединяющие вершины

6.8.2. Граничные модели

В граничной (поверхностной) модели (boundary representation) трехмерный объект представляется как система поверхностей, создающих его границы. Каждая из таких поверхностей описывается неявным уравнением или параметрической функцией с указанием границ изменения координат или параметров, что позволяет определить линии пересечения поверхностей. Граничная модель рассматривает только точки на поверхности тела, не затрагивая его объема.

В отличие от каркасной модели с плоскими гранями граничное описание позволяет изображать нелинейную поверхность каркасом с криволинейными ячейками.

6.8.3. Сплошные модели

Сплошные или твердотельные модели описывают точки как внутри, так и на поверхности объекта. Трехмерные тела рассматриваются как объединение (композиция) некоторых базовых блоков. Тип базовых объектов определяет различные методы моделирования этим способом.

Воксельное представление

В качестве базового блока выбирается кубическая ячейка пространства — воксел. Рассматриваемая часть пространства (сцена) представляется трехмерным битовым массивом c_{ijk}. Элементы массива равны 1, если куб c_{ijk} содержится в объекте, и равны 0 в противном случае. Иногда считают, что элементы массива могут меняться в диапазоне [0,1], задавая плотность в данной точке пространства (рис. 6.30).

Воксельное представление позволяет достаточно легко вычислить объем или центр масс и позволяет эффективно выполнять булевы операции над телами (пересечение, объединение), но дает только приближение реального объекта, качество кото-

рого зависит от размера вокселей. Хорошее качество требует больших размеров памяти, объем которой растет как N^3.

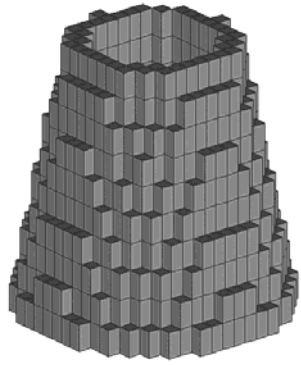


Рис. 6.30. Воксельное представление усеченного конуса с цилиндрическим отверстием

Конструктивное (твердотельное) моделирование

В случае конструктивного (твердотельного) моделирования объект задается набором плоских или объемных примитивов и операций над ними. Примитивы являются "строительными блоками" объекта и обычно описываются граничной моделью. Под операциями понимаются булевы операции над примитивами, а также геометрические преобразования, такие как перемещение, поворот, изменение размеров.

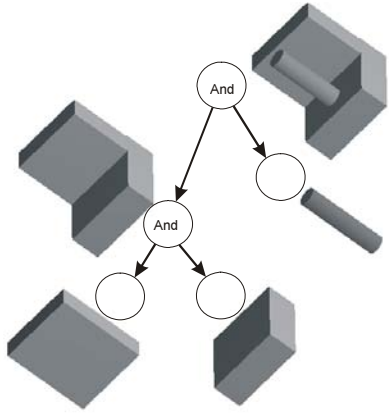


Рис. 6.31. Дерево конструктивной модели

Полная конструктивная модель объекта включает:

- данные о типе каждого примитива и его граничной модели, включая оптические свойства поверхностей;
- последовательность логических операций над примитивами при конструировании объекта;
- последовательность аффинных преобразований на каждом шаге конструирования.

Процесс конструирования можно представить в виде бинарного дерева с правилом обхода "снизу-вверх". В этом дереве листьями являются геометрические примитивы, а каждому узлу сопоставляется операция. Вершиной дерева является геометрический объект (рис. 6.31).

6.9. Освещение

Для придания изображению большей реалистичности, модели трехмерных объектов включают методы передачи различных оптических эффектов. Остановимся на рассмотрении некоторых из них.

Считается, что освещенность в точке, в основном, зависит от диффузионного отражения, определяемого глубинными слоями материала, и зеркального отражения от поверхностных слоев материала [83].

Прежде всего, отметим, что нет однозначного и более или менее точного алгоритма вычисления коэффициентов диффузионного (D) и зеркального (S) отражения. Далее пойдет речь о некоторых часто используемых моделях.

Рассмотрим поверхность с единичной нормалью N. Пусть L — единичный вектор в направлении источника света и наблюдение ведется со стороны оси OZ с единичным вектором k. Коэффициенты диффузионного D и зеркального S отражения зависят от следующих параметров (рис. 6.32):

- от угла l между направлением к источнику света и нормалью к поверхности;
- от угла m между вектором $M = L + k$ и нормалью к поверхности;
- от максимального коэффициента диффузионного отражения C_d;
- от шероховатости материала K_d;
- от максимального коэффициента отражения поверхностного слоя C_s;
- от шероховатости поверхностного слоя K_s.

Для определения коэффициентов отражения, в основном, используются модели диффузного и зеркального отражения. Значения перечисленных параметров для этих моделей приведены в табл. 6.1.

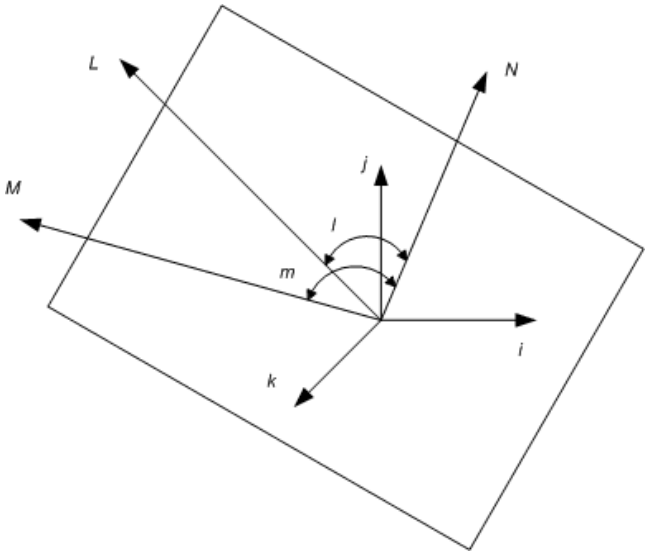


Рис. 6.32. Параметры коэффициента диффузионного освещения

Таблица 6.1. *Модели освещения*

Модель диффузного отражения	**Модель зеркального отражения**
$D = C_d \times \cos l$	$S = C_s \times \cos l \times \cos m, \quad 0 < m < \pi/2$
$D = C_d \times \cos l$	$S = 0$
$D = \cos^2(l) / D_s$	$S = 0$
$D = \cos^{K_d}(l), \quad 0 < l < \pi/2$	$S = 0$
$D = \cos(l) / D_s$	$S = C_s \cos^{Ks}(l) / D_s$
$D = \lvert\cos(l)\rvert / D_s$	$S = \cos^{Ks}(l) / D_s$
$D = C_d \times \cos(l)$	$S = C_s \cos^{Ks}(m)$
$D = C_d \times Exp(-(l / K_d))^2$	$S = C_s \times Exp(-(m / K_s))^2$

Модель освещения для любой точки определяется следующими выражениями:

$$\begin{aligned} ColorR &= \max(\max(1,\ A + D) \times C_m R + s,\ 1) \times C_s R; \\ ColorG &= \max(\max(1,\ A + D) \times C_m G + s,\ 1) \times C_s G; \\ ColorB &= \max(\max(1,\ A + D) \times C_m B + s,\ 1) \times C_s B, \end{aligned} \tag{6.109}$$

где A — интенсивность рассеянного света, B — интенсивность белого цвета, C_sR, C_sG, C_sB — красная, зеленая, синяя составляющие цвета источника света, C_mR, C_mG, C_mB — красная, зеленая, синяя составляющие цвета материала. Все параметры изменяются от 0 до 1.

Модели, рассмотренные ранее, применимы, если графический адаптер позволяет использовать одновременно большое количество цветов. Даже на 16-цветном мониторе можно использовать эти соотношения, если установить 7—8 оттенков одного цвета. В *главе 9* рассматривается пример построения тора, в котором коэффициент отражения пропорционален $|\cos l|$.

При моделировании освещения в условиях бедной палитры, когда необходимо построить изображение искривленной поверхности тремя цветами $(c1, c2, c3)$, возникает задача сглаживания границ между областями разного цвета. Эта задача относится к большой группе проблем, объединенных термином "явление алиайзинга". Чаще всего алиайзинг проявляется в появлении цветовых ступенек при изображении гладких поверхностей и пропадании малоразмерных объектов.

6.10. Моделирование цвета

Основным программным инструментом конструирования цвета графического модуля `Graphics` в Delphi является свойство `Color`, которое позволяет назначить любой из 256×256×256=16777216 цветов.

При этом интенсивность цвета $m(Color)$ равна сумме интенсивностей красного (R), зеленого (G) и синего (B) цветов: $m(Color) = r(R) + g(G) + b(B)$.

Получение любого цвета из трех основных (R, G, B) продиктовано технологией изготовления электронно-лучевых трубок и является достаточно искусственным приемом. Для оценки цветового восприятия удобнее пользоваться другими параметрами: цветовым тоном (H), яркостью (L) и насыщенностью (S) [98].

Цветовой тон определяется длиной волны, преобладающей в потоке излучения (усредненная длина волн).

Яркость соответствует световой энергии, достигающей наблюдателя. Можно сказать, что яркость — это интенсивность света, искаженная сетчаткой глаза. Человеческий глаз может различать порядка тысячи уровней яркости. Насыщенность (чистота цвета) характеризует долю белого цвета в чистых спектральных цветах. Вместо параметров (R, G, B) можно использовать параметры (H, L, S) в трехмерном пространстве с цилиндрическими координатами [98] (рис. 6.33). На центральной оси цилиндра находятся значения яркости, изменяющиеся от нулевой на нижнем основании цилиндра до максимальной на его верхнем основании. Цветовой тон задается углом поворота вектора, показанного в основании цилиндра, а насыщенность — величиной этого вектора.

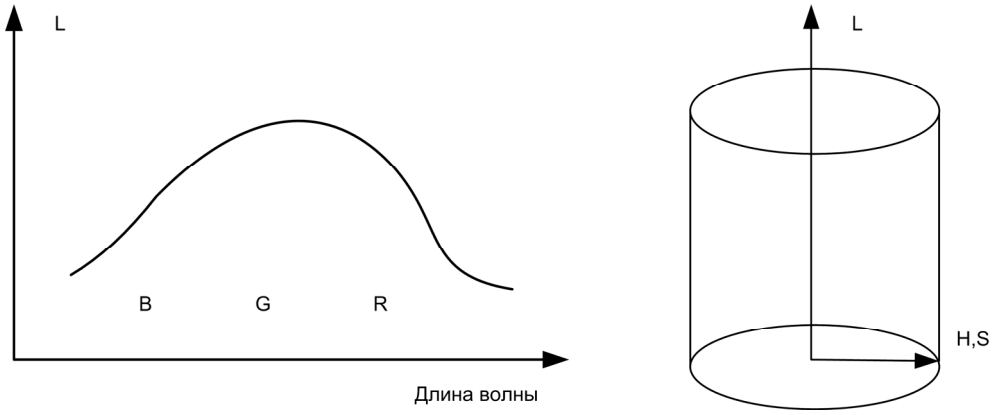


Рис. 6.33. Интенсивность света и параметры *H*, *L*, *S*

Свойства этого пространства хорошо изучены, переход от (H, L, S) к (R, G, B) и обратно определяется формулами:

$$H = \arcsin(\sqrt{(3/2)} \times (G - R)/S); \quad L = (R + G + B)/3;$$
$$S = \sqrt{R \times R + G \times G + B \times B - R \times G - R \times B - G \times B}; \qquad (6.110)$$
$$R = L - S \times \cos(H)/3 - S \times \sin(H)/\sqrt{3};$$
$$G = L + 2 \times S \times \cos(H)/3;$$
$$B = L - S \times \cos(H)/3 + S \times \sin(H)/\sqrt{3}.$$

Параметры H, L, S, во-первых, удобно использовать для интерактивного выбора цвета на экране монитора. Для этого необходимо представить на экране сокращенную палитру при фиксированной яркости L, а затем увеличить или уменьшить яркость.

Второй привлекательный момент параметров H, L, S заключается в удобстве их использования при моделировании затененности объектов или сумерек (линейное изменение яркости от начального цвета до черного) и при моделировании дымки (линейное изменение яркости от начального цвета до серого).

6.11. Удаление невидимых ребер и граней

Алгоритм Робертса представляет собой первое известное решение задачи об удалении невидимых линий [98]. Это математически элегантный метод, работающий в трехмерном пространстве. Алгоритм, прежде всего, удаляет из каждого тела невидимые ребра или грани, то есть те ребра или грани, которые экранируются самим

телом. Затем каждое из видимых ребер каждого тела сравнивается с каждым из оставшихся тел для определения того, какая его часть или части, если таковые есть, экранируются этими телами. Поэтому вычислительная трудоемкость алгоритма Робертса растет, как квадрат общего числа объектов. Это обстоятельство в сочетании с ростом интереса к растровым дисплеям, работающим в пространстве изображения, привело к снижению интереса к алгоритму Робертса. Однако математические методы, используемые в этом алгоритме, просты, мощны и точны. Кроме того, этот алгоритм можно использовать для иллюстрации некоторых важных концепций. Наконец, более поздние реализации алгоритма, использующие предварительную приоритетную сортировку вдоль оси z и простые габаритные или минимаксные тесты, демонстрируют почти линейную зависимость от числа объектов.

Работа Алгоритм Робертса проходит в два этапа:

- определение нелицевых граней отдельно для каждого тела;
- определение и удаление невидимых ребер.

Пусть F — некоторая грань многогранника. Плоскость, несущая эту грань, разделяет пространство на два подпространства. Назовем положительным то подпространство, в сторону которого смотрит внешняя нормаль к грани. Если точка наблюдения находится в положительном подпространстве, то грань — *лицевая*, в противном случае — *нелицевая*. Если многогранник выпуклый, то удаление всех нелицевых граней полностью решает задачу визуализации с удалением невидимых граней.

Для определения, лежит ли точка в положительном подпространстве, используют проверку знака скалярного произведения (l, n), где l — вектор, направленный к наблюдателю, фактически определяет точку наблюдения; n — вектор внешней нормали грани. Если $(l, n) > 0$, т. е. угол между векторами острый, то грань является лицевой. Если $(l, n) < 0$, т. е. угол между векторами тупой, то грань является нелицевой.

При использовании алгоритма Робертса необходимо, чтобы все изображаемые тела или объекты были выпуклыми. Невыпуклые тела должны быть разбиты на выпуклые части. В этом алгоритме выпуклое многогранное тело с плоскими гранями должно представляться набором пересекающихся плоскостей. Уравнение произвольной плоскости в трехмерном пространстве имеет вид (6.26). То же уравнение в матричной форме имеет вид:

$$(x, y, z, 1) \times P^T = 0, \qquad (6.111)$$

где $P = (a, b, c, d)$ — вектор, составленный из коэффициентов уравнения плоскости. Поэтому любой выпуклый многогранник можно выразить матрицей, состоящей из коэффициентов уравнений плоскостей.

Напомним, что любая точка S пространства представима в однородных координатах вектором $S = (x, y, z, 1)$. Более того, если эта точка лежит на плоскости, то

$S \times P^T = 0$. Если же точка не лежит на плоскости, то знак этого скалярного произведения показывает, по какую сторону от плоскости расположена точка. В алгоритме Робертса предполагается, что точки, лежащие внутри тела, дают отрицательное скалярное произведение, т. е. нормали направлены наружу. Чтобы проиллюстрировать эти положения, рассмотрим следующий пример.

Рассмотрим единичный куб с центром в начале координат. Шесть плоскостей, описывающих данный куб, таковы:

$x_1 = 1/2, \quad x_2 = -1/2, \quad y_3 = 1/2, \quad y_4 = -1/2, \quad z_5 = 1/2, \quad z_6 = -1/2.$

Более подробно уравнение правой плоскости можно записать как

$x_1 + 0 \times y_1 + 0 \times z_1 - 1/2 = 0$ или $2x_1 - 1 = 0.$

Полная матрица тела такова:

$$V = \begin{pmatrix} 2 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 2 \\ -1 & 1 & -1 & 1 & -1 & 1 \end{pmatrix}. \qquad (6.112а)$$

Экспериментально проверим матрицу тела с помощью точки, о которой точно известно, что она лежит внутри тела. Если знак скалярного произведения для какой-нибудь плоскости больше нуля, то соответствующее уравнение плоскости следует умножить на –1. Для проверки возьмем точку внутри куба с координатами $x = 1/4, \quad y = 1/4, \quad z = 1/4.$ В однородных координатах эта точка представляется в виде вектора

$$S = (1,\ 1,\ 1,\ 4).$$

Произведение этого вектора на матрицу объема равно

$$S \times V = (-2,\ 6, -2,\ 6, -2,\ 6).$$

Здесь результаты для второго, четвертого и шестого уравнения плоскостей (столбцов) положительны и, следовательно, составлены некорректно. Умножая эти уравнения (столбцы) на –1, получаем корректную матрицу тела для куба:

$$V = \begin{pmatrix} 2 & -2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & -2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & -2 \\ -1 & -1 & -1 & -1 & -1 & -1 \end{pmatrix}. \qquad (6.112б)$$

Глава 7

Простые графические проекты

Материал предыдущих глав содержит достаточно много технических деталей и, возможно, утомил читателя. В *главе 7* представлены иллюстрации к изложенному в предыдущих главах материалу в виде реализованных проектов, демонстрирующих некоторые свойства и методы описанных классов. При подборе этих демонстрационных задач мы, с одной стороны, постарались охватить вопросы, представленные в справочном материале первых глав, с другой стороны, представить наиболее интересные с практической точки зрения задачи.

В *разд. 7.1* описан проект, обеспечивающий просмотр файлов BMP, ICO, WMF, EMF и JPG. В *разд. 7.2* рассматриваются четыре проекта с использованием мультипликации: сортировка элементов массива, превращение одного предмета в другой, падение резинового мяча и движение велосипеда. В *разд. 7.3* описывается проект, рассматривающий особенности работы с принтером при выводе графики на печать. В *разд. 7.4* описывается проект, посвященный решению задачи отображения нестандартных векторных стилей линий.

Проект, описанный в *разд. 7.5*, показывает, как можно деформировать BMP-изображение: прямоугольная картинка преобразуется в произвольный четырехугольник, вершины которого можно задавать в интерактивном режиме.

В проекте, предложенном в *разд. 7.6*, реализуется графический редактор с широкими возможностями и даже некоторыми спецэффектами.

В проектах, описанных в *разд. 7.7—7.9*, рассматривается проектирование плоских схем. К этому кругу задач относятся задачи построения сложных диаграмм и электрических схем, структурных схем соединения компьютеров в сети и структурных схем программ, проектирования реляционных баз данных, задачи макетирования карт и множество других проблем. Предлагаемая программа позволяет включать в проект условные обозначения объектов; устанавливать связи между объектами; перемещать объекты и связи; удалять объекты и связи.

7.1. Просмотр файлов BMP, ICO, WMF, EMF и JPG

Проект "Просмотр файлов BMP, ICO, WMF, EMF и JPG" предназначен для просмотра графических файлов в соответствующих форматах. Просмотр файлов в формате JPG имеет свою специфику и допускает управление качеством изображения, скоростью вывода, размером изображения на экране и возможностью переключения режимов монохромного и цветного отображения.

Для организации интерфейса на главную форму проекта **FormMain** вынесен компонент `ScrollBox1`, на котором установлен компонент `Image1` (рис. 7.1).

Для настройки вывода изображений на форме **FormTools** (рис. 7.1) используется четыре компонента класса `TComboBox`:

- `Scale` (с возможными значениями `jsFullSize`, `jsHalf`, `jsQuarter`, `jsEighth`) — для задания следующих масштабов: реальный размер изображения, половина, четверть и одна восьмая размера изображения;

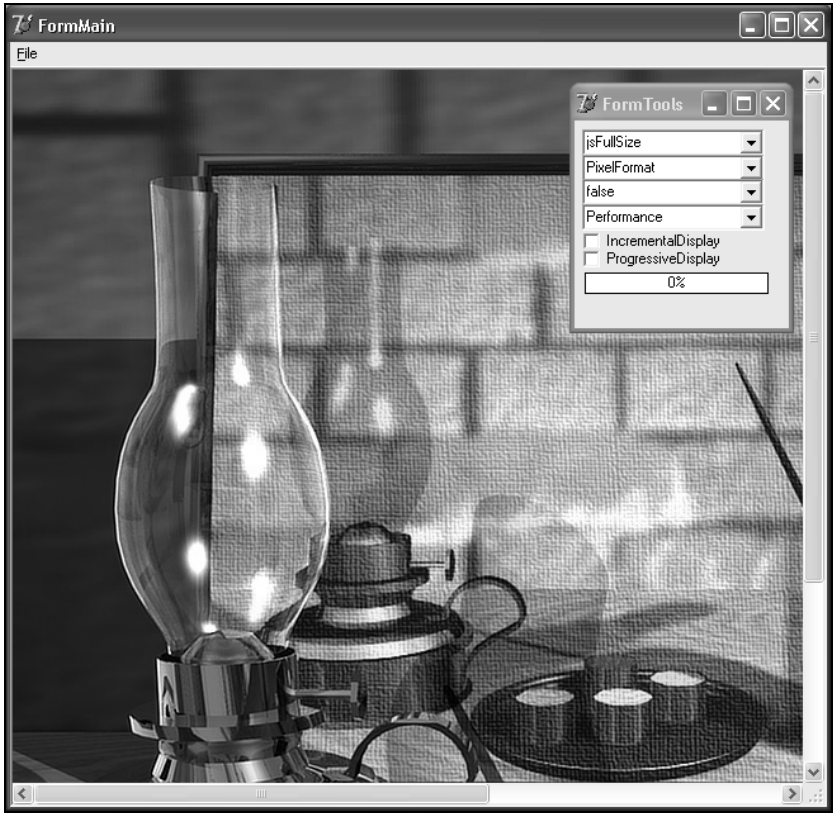


Рис. 7.1. Формы проекта "Просмотр файлов BMP, ICO, WMF, EMF и JPG"

- `PixelFormat` (с возможными значениями `jf24Bit` и `jf8Bit`) — для задания формата пикселов;
- `Colorspace` (с возможными значениями `false` и `true`) — для назначения монохромного вывода;
- `Performance` (с возможными значениями `jpBestQuality` и `jpBestSpeed`) — для управления отношением между качеством цвета и скоростью декомпрессии.

Для настройки вывода изображений также используется два компонента типа `TCheckBox`:

- `ProgressiveDisplay` — разрешает или запрещает показ изображения при чтении из файла;
- `IncrementalDisplay` — определяет, должны ли выводиться промежуточные изменения изображения в течение медленных действий.

При инициализации проекта для `Image1` назначается процедура индикации медленных процессов: `Image1.OnProgress := ProgressUpdate`, которая выводит процент выполнения задания на компонент `Gauge1` (листинг 7.1).

Листинг 7.1. Процедура отображения медленных процессов

```
procedure TFormTools.ProgressUpdate(Sender: TObject; Stage:
  TProgressStage; PercentDone: Byte; RedrawNow:
  Boolean; const R: TRect; const Msg: string);
begin
  if Stage = psRunning
    then Gauge1.Progress:=PercentDone
    else Gauge1.Progress:=0
end;
```

Выбор файла осуществляется с помощью диалогового окна для открытия файлов (рис. 7.2), вызываемого стандартным методом `OpenPictureDialog1`, в котором запоминается имя файла и вызывается процедура рисования (листинги 7.2, 7.3).

Листинг 7.2. Открытие файла

```
procedure TFormMain.Open1Click(Sender: TObject);
begin
  if OpenPictureDialog1.Execute then begin
    FileName:=OpenPictureDialog1.FileName;
    Caption:=FileName;
    DrawJPG;
  end;
end;
```

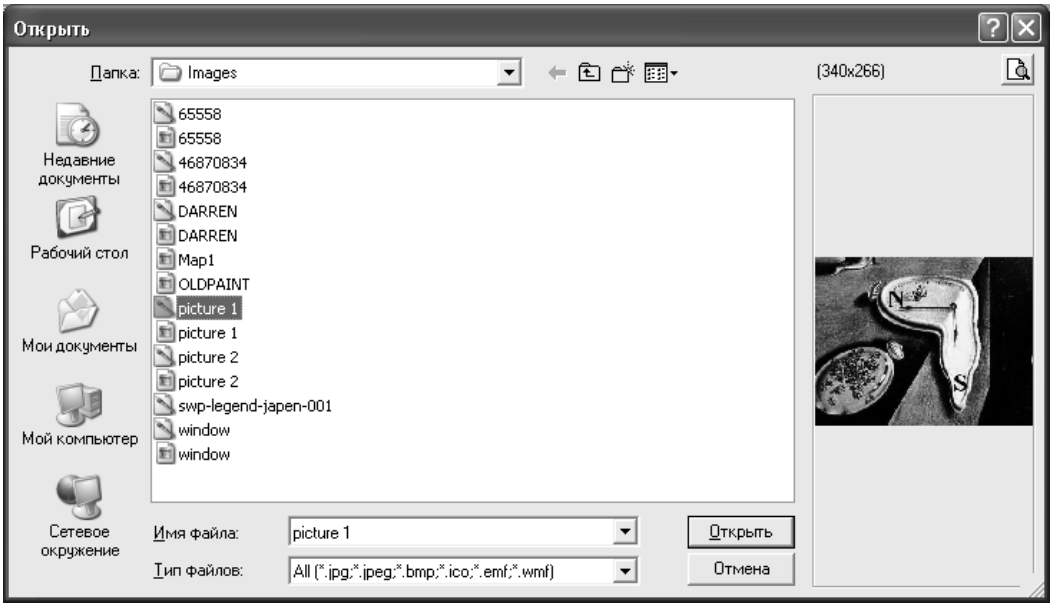


Рис. 7.2. Диалоговое окно для открытия файла

Листинг 7.3. Процедура рисования

```
procedure TFormMain.DrawJPG;
var s: string;
begin
  s:=UpperCase(ExtractFileExt(FileName));
  if s<>'.JPG' then
    Bitmap.LoadFromFile(FileName)
  else begin
    JPEGImage.LoadFromFile(FileName);
    SetJPEGOptions(self);
    Bitmap.Assign(jp);
  end;

  with Image1,Canvas do begin
    if (Bitmap.Width>Width) or (Bitmap.Height>Height) then begin
      Width:=Bitmap.Width; Height:=Bitmap.Height;
    end
    else begin
      Width:=640; Height:=580;
    end;
    FillRect(Rect(0,0,Width,Height));
```

```
    Draw((Width-Bitmap.Width) div 2,
         (Height-Bitmap.Height) div 2,Bitmap);
  end;
end;
```

Выбранный файл должен загрузиться в `Bitmap`. К сожалению, метод `LoadFromFile`, которым обладает класс `TBitmap`, позволяет работать только с файлами BMP, ICO, WMF и EMF. Поэтому для работы с файлами `JPG` используется переменная `JPEGImage: TJPEGImage`.

После загрузки файла в `JPEGImage` вызывается процедура `SetJPEGOptions(self)`, в которой устанавливаются параметры рисования (листинг 7.4).

Листинг 7.4. Процедура установки параметров рисования

```
procedure TFormMain.SetJPEGOptions(Sender: TObject);
begin
  with JPEGImage do begin
    PixelFormat :=
      TJPEGPixelFormat(FormTools.PixelFormat.ItemIndex);
    Scale := TJPEGScale(FormTools.Scale.ItemIndex);
    Grayscale := Boolean(FormTools.Colorspace.ItemIndex);
    Performance :=
      TJPEGPerformance(FormTools.Performance.ItemIndex);
    ProgressiveDisplay := FormTools.ProgressiveDisplay.Checked;
  end;
  Image1.IncrementalDisplay :=
    FormTools.IncrementalDisplay.Checked;
end;
```

При щелчке указателем мыши на компонентах управления параметрами вывода (`Scale`, `PixelFormat`, `Colorspace`, `Performance`, `IncrementalDisplay`) вызывается метод перерисовки `DrawJPG` (листинг 7.3).

Мы получили приложение для просмотра файлов. Фрагменты этого проекта могут быть включены, например, в редакторы графических файлов. Проект приведен на компакт-диске в папке **Примеры | Глава 7 | View JPG**.

7.2. Мультипликация

Изменение изображений во времени возможно двумя способами: во-первых, можно воспользоваться процедурой временной задержки `Sleep(milliseconds: Cardinal)` из модуля `SysUtils`; во-вторых, воспользоваться компонентом `Timer`, имеющим один обработчик события `onTimer`.

Демонстрацию процедуры `Sleep` покажем на примере сортировки элементов массива, а использование компонента `Timer` — на примере морфинга изображения одного предмета (стула) в другой (стол) и в проекте, показывающем движение велосипеда.

7.2.1. Сортировка элементов массива

Алгоритм *сортировки методом пузырька* в полной мере воплощает принцип *обменной* сортировки [65]: сравниваются и обмениваются местами два соседних элемента. Алгоритм очень прост и приведен в листинге 7.5.

Листинг 7.5. Сортировка методом пузырька

```
procedure SortBubble (var a: TData);
var i,j: integer;
    tmp: integer;
begin
  for i:=2 to N do begin
  for j:=N downto i do
    if a[j-1]>a[j] then begin // сравнение элементов
      tmp:=a[j]; a[j]:=a[j-1]; a[j-1]:=tmp
      //обмен элементов местами
    end
  end;
end;
```

В предлагаемом анимированном проекте этот алгоритм представлен в наглядном графическом виде (рис. 7.3), в котором миганием элементов обозначается их сравнение, а обмен происходит в динамике: верхний элемент уходит в сторону, а нижний становится на его место.

Для описания элементов массива используется структура, представленная в листинге 7.6.

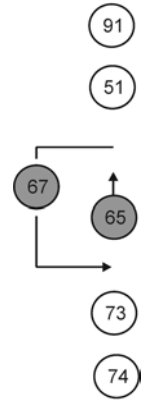


Рис. 7.3. Сортировка методом пузырька

Листингг 7.6. Структура элементов массива

```
  TElement = record
    x,y  : integer;
    Inf  : integer;
    Color: TColor;
  end;
```

В этой структуре поля `x,y: integer` отвечают за положение элемента на экране, поле `Inf: integer` содержит значение элемента массива, поле `Color: Tcolor` используются для задания цвета, с которым изображается элемент.

При нажатии кнопки с помощью генератора случайных чисел заполняются элементы массива, назначаются их начальные координаты на экране (листинг 7.7) и вызывается процедура сортировки.

Листинг 7.7. Начальное заполнение массива

```
procedure TForm1.Button1Click(Sender: TObject);
var i,L: integer;
begin
  Randomize;
  L:=12; SetLength(Element,L);
  for i:=0 to L-1 do begin
    Element[i].x:=100;
    Element[i].y:=20+i*40;
    Element[i].Color:=clWhite;
    Element[i].Inf:=Random(100);
  end;
  Sort;
end;
```

В листинге 7.8 процедура сортировки, представленная в листинге 7.5, слегка изменена: добавлена процедура мигания сравниваемых элементов `Delay(j,j-1)` и процедура динамического обмена местами двух элементов `Change(j,j-1)`.

Листинг 7.8. Процедура сортировки

```
procedure TForm1.Sort;
var
  i,j,L: integer;
  tmp: TElement;
begin
  L:=Length(Element);
  for i:=2-1 to L-1 do
  for j:=L-1 downto i do begin
    Delay(j,j-1);
    if Element[j-1].Inf>Element[j].Inf then begin
      Change(j,j-1);
      tmp:=Element[j]; Element[j]:=Element[j-1]; Element[j-1]:=tmp
    end
  end;
end;
```

Процедура мигания `Delay` (листинг 7.9) использует процедуру временной задержки `Sleep` для шестикратной смены синего цвета на белый.

Листинг 7.9. Процедура мигания сравниваемых элементов

```
procedure TForm1.Delay(n1,n2: integer);
var i: integer;
begin
  for i:=0 to 5 do begin
    Element[n1].Color:=clBlue;
    Element[n2].Color:=clBlue;
    Drawing;
    Sleep(100);
    Element[n1].Color:=clWhite;
    Element[n2].Color:=clWhite;
    Drawing;
    Sleep(200);
  end;
end;
```

Наиболее интересной является процедура динамического обмена местами двух элементов `Change` (листинг 7.10). Она выполняется за 15 тактов. За эти 15 тактов координата `y` первого элемента меняется от `y1` до `y2`, а координата `x` не меняется. Сложнее поведение второго элемента: первые 4 такта элемент движется налево, затем за 7 тактов он опускается от `y2` до `y1`, за последние 4 такта движется направо.

Листинг 7.10. Процедура динамического обмена местами двух элементов

```
procedure TForm1.Change(n1,n2: integer);
var
  t,x1,y1,x2,y2: integer;
begin
  Element[n1].Color:=clRed;
  Element[n2].Color:=clRed;
  x1:=Element[n1].x; y1:=Element[n1].y;
  x2:=Element[n2].x; y2:=Element[n2].y;
  for t:=1 to 15 do begin
    Element[n1].y:=y1+Round((y2-y1)*t/15);
    case t of
      1..4:
        Element[n2].x:=x1-Round(40*t/4);
      5..11:
        Element[n2].y:=y2+Round((y1-y2)*(t-4)/7);
      12..15:
        Element[n2].x:=x1-40+Round(40*(t-11)/4);
    end;
```

```
    Drawing;
    Sleep(100);
  end;
  Element[n1].Color:=clWhite;
  Element[n2].Color:=clWhite;
  Drawing;
end;
```

Для ускорения работы программы, рисование эллипсов, соответствующих всем элементам массива, происходит на канве `Bitmap: Tbitmap` (листинг 7.11).

Листинг 7.11. Рисование элементов

```
procedure TForm1.Drawing;
const
  d=15;
var
  i,L: integer;
  s: string;
begin
  L:=Length(Element);
  with Bitmap.Canvas do begin
    Brush.Color:=clWhite;
    FillRect(Rect(0,0,Width,Height));
    Font.Style:=[fsBold];
    for i:=0 to L-1 do
    with Element[i] do begin
      Brush.Color:=Color;
      Ellipse(x-d,y-d,x+d,y+d);
      s:=IntToStr(Inf);
      TextOut(x-TextWidth(s) div 2,y-TextHeight(s) div 2,s);
    end;
  end;
  Canvas.Draw(0,0,Bitmap);
end;
```

Полный текст проекта приведен на компакт-диске в папке **Примеры | Глава 7 | Сортировка**.

7.2.2. Морфинг

Морфинг — это анимация формы, при которой происходит "превращение одного предмета в другой". В качестве примера рассмотрим "превращение" стула (рис. 7.4) в стол (рис. 7.6). На рис. 7.5 представлена одна из промежуточных стадий превращения.

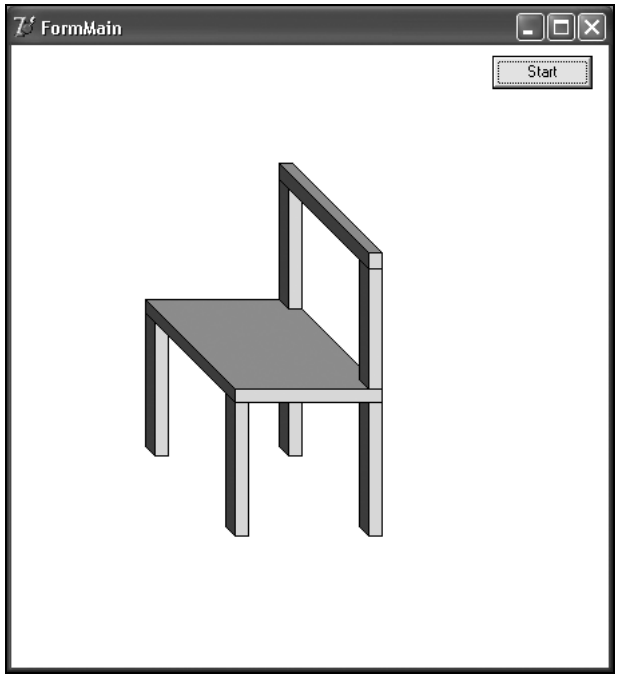


Рис. 7.4. Начальная стадия морфинга

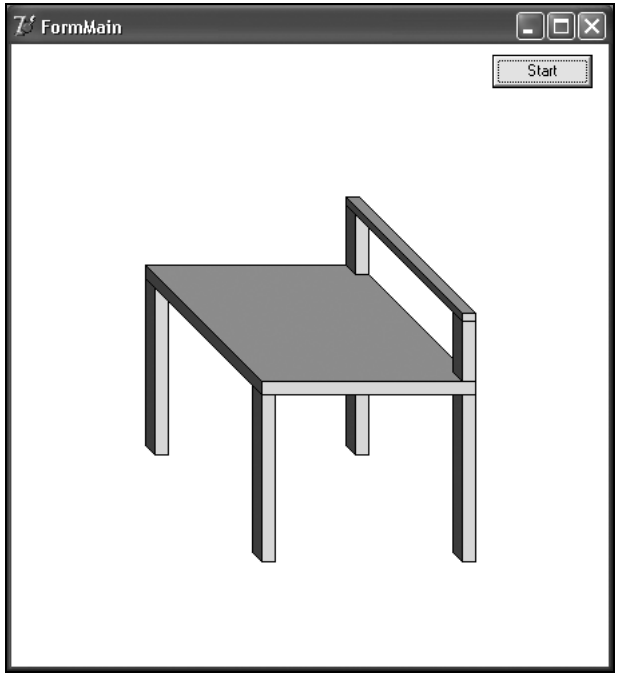


Рис. 7.5. Промежуточная стадия морфинга

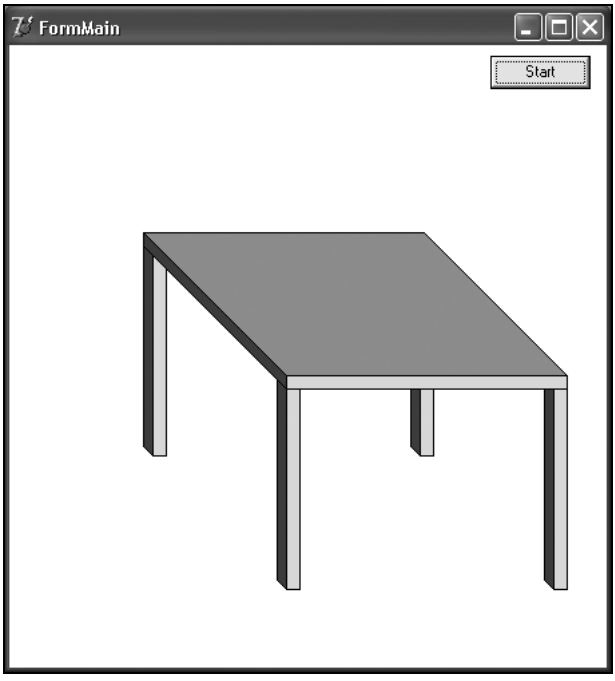


Рис. 7.6. Конечная стадия морфинга

Для реализации морфинга воспользуемся компонентом `Timer1`, который запускается (становится активным) при нажатии кнопки `Button1` (листинг 7.12).

Листинг 7.12. Активизация таймера

```
procedure TFormMain.Button1Click(Sender: TObject);
begin
  t:=-1;
  Timer1.Enabled:=true;
end;
```

Переменная целого типа `t` увеличивается на `1` при каждом вызове обработчика события `onTimer` и является для нас счетчиком секунд. На самом деле частота вызова настраивается свойством `Timer1.Interval`. В нашем случае значение этого свойства равно 100 "тикам", что приводит к вызову обработчика через 0,1 секунды.

Итак, обработчик увеличивает `t` на `1`, проверяет, не пора ли заканчивать процесс, и вызывает процедуру рисования `Draw(t)` (листинги 7.13 и 7.14).

Листинг 7.13. Обработчик события таймера

```
procedure TFormMain.Timer1Timer(Sender: TObject);
begin
  t:=t+1;
```

```
  Timer1.Enabled:=t<t10;
  Draw(t);
end;
```

Листинг 7.14. Процедура рисования

```
procedure TFormMain.Draw(t: integer);
const
  x100: integer=100;
  y300: integer=300;
  Vx7: integer=7;
  Vy7: integer=7;
  W10: integer=10;
  H10: integer=10;
var
  H90,H12,Vx80,Vy80,W100,H100: integer;
begin
  H90:=LineAB(90,0,t);
  H12:=LineAB(12,0,t);
  Vx80:=LineAB(60,100,t);
  Vy80:=LineAB(60,100,t);
  W100:=LineAB(100,200,t);
  H100:=LineAB(100,150,t);
  with Bitmap.Canvas do begin
    FillRect(Rect(0,0,Width,Height));
    // 1
    x0:=x100; y0:=y300; Vx:=Vx7; Vy:=Vy7; W:=W10; H:=H100;
    DrawV(x0,y0,Vx,Vy,W,H,Bitmap.Canvas);
    // 2
    x0:=x100+Vx80; y0:=y300+Vy80; Vx:=Vx7; Vy:=Vy7;
    W:=W10; H:=H100;
    DrawV(x0,y0,Vx,Vy,W,H,Bitmap.Canvas);
    // 3
    x0:=x100+Vx80+W100; y0:=y300+Vy80; Vx:=Vx7; Vy:=Vy7;
    W:=W10; H:=H100;
    DrawV(x0,y0,Vx,Vy,W,H,Bitmap.Canvas);
    // 4
    x0:=x100+W100; y0:=y300; Vx:=Vx7; Vy:=Vy7; W:=W10; H:=H100;
    DrawV(x0,y0,Vx,Vy,W,H,Bitmap.Canvas);
    // 5
    x0:=x100; y0:=y300-H100; Vx:=Vx80+Vx7; Vy:=Vy80+Vy7;
    W:=W100+W10; H:=H10;
    DrawV(x0,y0,Vx,Vy,W,H,Bitmap.Canvas);
    // 6
    if H90<>0 then begin
```

```
      x0:=x100+W100; y0:=y300-H100-H10; Vx:=Vx7; Vy:=Vy7;
      W:=W10; H:=H90;
      DrawV(x0,y0,Vx,Vy,W,H,Bitmap.Canvas);
    end;
    // 7
    if H90<>0 then begin
      x0:=x100+W100+Vx80; y0:=y300-H100-H10+Vy80; Vx:=Vx7;
      Vy:=Vy7; W:=W10; H:=H90;
      DrawV(x0,y0,Vx,Vy,W,H,Bitmap.Canvas);
    end;
    // 8
    if H12<>0 then begin
      x0:=x100+W100; y0:=y300-H100-H10-H90; Vx:=Vx7+Vx80;
      Vy:=Vy7+Vy80; W:=W10; H:=H12;
      DrawV(x0,y0,Vx,Vy,W,H,Bitmap.Canvas);
    end;
  end;
  Image1.Canvas.Draw(0,0,Bitmap);
end;
```

Стул изображается восемью параллелепипедами, два из которых на конечной стадии морфинга достигают нулевых размеров, а третий сливается с крышкой стола. Каждый из этих параллелепипедов рисуется тремя гранями в процедуре `DrawV`. Все восемь параллелепипедов зависят от параметров `H90`, `H12`, `Vx80`, `Vy80`, `W100`, `H100`, меняющихся во времени по линейному закону от начального к конечному значению. Для рисования любого параллелепипеда достаточно трех граней, зависящих от шести параметров `x0`, `y0`, `Vx`, `Vy`, `W`, `H` (листинг 7.15).

Листинг 7.15. Рисование параллелепипеда

```
procedure TFormMain.DrawV(x0,y0,Vx,Vy,W,H: integer;
  Canvas: TCanvas);
var p: array[0..3] of TPoint;
begin
  with Canvas do begin
    p[0].X:=x0; p[0].y:=y0;
    p[1].X:=x0; p[1].y:=y0-H;
    p[2].X:=x0+Vx; p[2].y:=y0-H+Vy;
    p[3].X:=x0+Vx; p[3].y:=y0+Vy;
    Polygon(p);
    p[0].X:=x0; p[0].y:=y0-H;
    p[1].X:=x0+W; p[1].y:=y0-H;
    p[2].X:=x0+W+Vx; p[2].y:=y0-H+Vy;
    p[3].X:=x0+Vx; p[3].y:=y0-H+Vy;
    Polygon(p);
    p[0].X:=x0+Vx; p[0].y:=y0+Vy;
```

```
    p[1].X:=x0+Vx; p[1].y:=y0+Vy-H;
    p[2].X:=x0+Vx+W; p[2].y:=y0-H+Vy;
    p[3].X:=x0+Vx+W; p[3].y:=y0+Vy;
    Polygon(p);
  end;
end;
```

Некоторые параметры изображения (`H90`, `H12`, `Vx80`, `Vy80`, `W100`, `H100`) меняются в зависимости от времени `t` от начального значения `a` до конечного значения `b`. Это изменение происходит по линейному закону и осуществляется функцией `LineAB(a,b,t: integer)` (листинг 7.16).

Листинг 7.16. Линейное изменение размера

```
function LineAB(a,b,t: integer): integer;
begin
  Result:=a+Trunc(t*(b-a)/t10);
end;
```

Проект "Морфинг" приведен на компакт-диске в папке **Примеры | Глава 7 | Морфинг**.

7.2.3. Падение мяча

В анимированном проекте "Падение мяча" с высоты H_0 падает резиновый мяч, который можно представить как круг радиуса R. В момент времени t_1 соприкосновения с поверхностью круг начинает превращаться в эллипс с полуосями A и B. Эти параметры достигают своих максимальных значений в момент времени $T_0/2$ и возвращаются к значению R в момент времени t_2. Затем начинается обратный процесс отскока мяча от поверхности. Временная диаграмма изменения параметров (высоты центра мяча H и полуосей A и B) представлена на рис. 7.7.

Высота центра мяча $h(t)$ на интервале времени $[0,t_1]$ меняется по параболе:

$$h = -k \times t^2 + H_0. \tag{7.1}$$

Из условия $h\big|_{t=t_1} = R$ можно определить t_1:

$$t_1 = \sqrt{(H_0 - R)/k}, \tag{7.2}$$

что позволяет определить t_2:

$$t_2 = T_0/2 + dt, \tag{7.3}$$

где $dt = T_0/2 - t_1$.

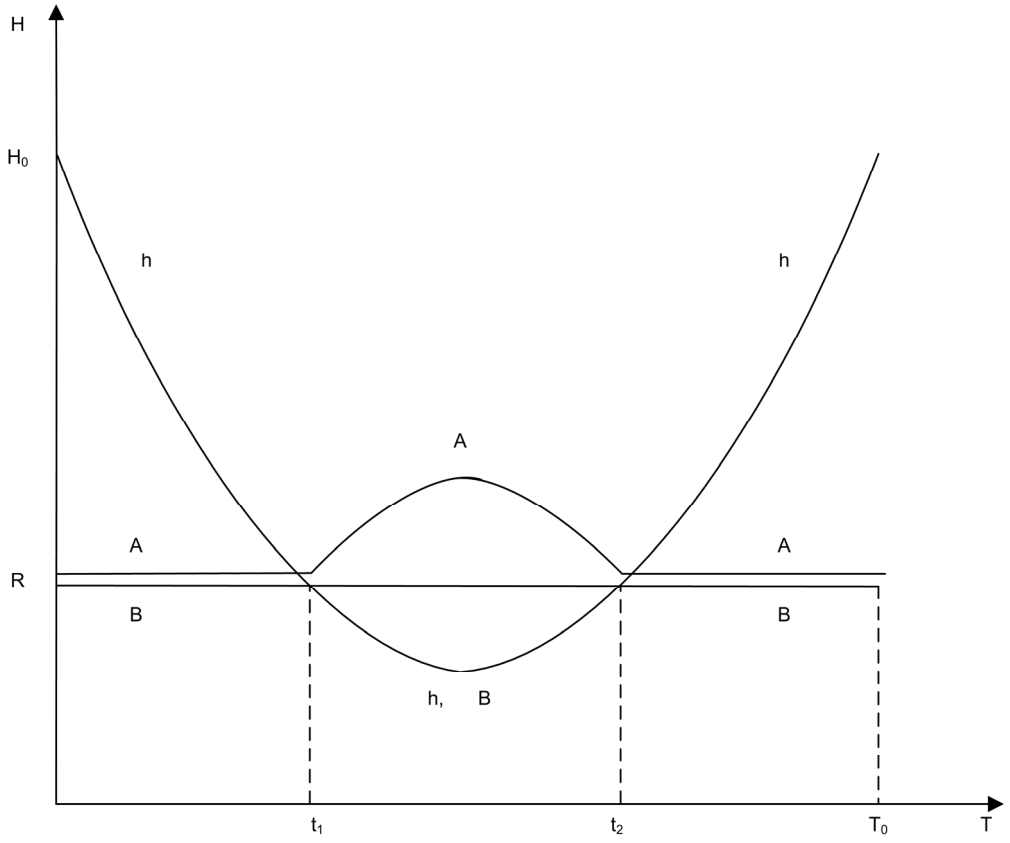


Рис. 7.7. Временная диаграмма изменения параметров мяча

Будем считать, что на интервале $[t_1, t_2]$ высота центра мяча $h(t)$ также меняется по параболе:

$$h = R - dR + k_1 \times (t - T_0 / 2)^2, \quad t \in [0, t_1]. \tag{7.4}$$

Из условия $h\big|_{t=t_1} = R$ определим k_1:

$$k_1 = dR / (t - T_0 / 2)^2. \tag{7.5}$$

Для полуоси эллипса $A(t)$ уравнение на интервале $[t_1, t_2]$ симметрично уравнению (7.4) относительно $h = R$:

$$A = R + dR - k_1 \times (t - T_0 / 2)^2, \quad t \in [t_1, t_2]. \tag{7.6}$$

Для полуоси эллипса $B(t)$ уравнение совпадает с уравнением (7.4).

Для интервала $[t_2, T_0]$ уравнение изменения высоты симметрично относительно $t = T_0 / 2$ и имеет вид:

$$h = -k \times (t - T_0)^2 + H_0, \quad t \in [2, T_0]. \tag{7.7}$$

На интервалах $[0,t_1]$ и $[t_2,T_0/2]$ параметры A и B не меняются и равны R.

$$A=R,\quad B=R,\quad t\in[0,t_1]\cup[t_2,T_0]. \tag{7.8}$$

Формулы (7.1)–(7.8) позволяют реализовать проект, представленный на компакт-диске в папке **Примеры | Глава 7 | Падение мяча**, следующим образом. Пусть заданы константы:

```
H0=550;
R=50;
T0=40;
dR=R/5;
k=5.0;
```

В момент запуска проекта вычислим параметры модели, описываемой уравнениями (7.1)–(7.8) (листинг 7.17).

Листинг 7.17. Вычисление параметров модели

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
  Bitmap1:=TBitmap.Create;
  Bitmap1.LoadFromFile('Земля.bmp');
  t1:=Round(Sqrt((H0-R)/k));
  dt:=T0 div 2-t1;
  t2:=T0 div 2+dt;
  k1:=dR/Sqr(t1-T0/2);
end;
```

Создадим обработчик события для компонента `Timer1` (листинг 7.18).

Листинг 7.18. Обработчик события для компонента *Timer*

```
procedure TFormMain.Timer1Timer(Sender: TObject);
begin
  t:=(t+1) mod T0;
  MyDraw(t);
end;
```

Процедура рисования `MyDraw` с помощью процедуры `GetHAB` вычисляет параметры `h`, `A`, `B` и копирует `Bitmap1` в прямоугольник `(100-A, y-B, 100+A, y+B)` (листинг 7.19).

Листинг 7.19. Процедура рисования

```
procedure TFormMain.MyDraw(t: integer);
var y: integer;
begin
  with Bitmap.Canvas do begin
```

```
    Brush.Color:=clWhite;      // очищаем канву
    FillRect(Rect(0,0,Width,Height));
    GetHAB(t,h,A,B);           // вычисляем h,A,B
    y:=H0-Round(h*(H0-R)/H0); // переход в экранные координаты
    //Brush.Color:=clBlue;       рисуем эллипс
    StretchDraw(Rect(100-A,y-B,100+A,y+B),Bitmap1);
    //Ellipse(100-A,y-B,100+A,y+B);
  end;
  Canvas.Draw(0,0,Bitmap);
end;
```

Процедура `GetHAB` вычисляет параметры `h`, `A`, `B` по разным формулам на интервалах `[0,t1]`, `[t1,t2]` и `[t2,T0]` (листинг 7.20).

Листинг 7.20. Вычисление параметров *h*, *A*, *B*

```
procedure GetHAB(t: integer; var h,A,B: integer);
begin
  A:=R; B:=R;
  if t<t1 then             // [0,t1]
    h:=-Round(k*t*t)+H0
  else
    if t<t2+1 then begin  // [t1,t2]
      h:=R+Round(-dR+k1*Sqr(t-T0/2));
      A:=R+Round(dR-k1*Sqr(t-T0/2));
      B:=h;
    end
    else                   // [t2,T0]
      h:=-Round(k*(t-T0)*(t-T0))+H0;
end;
```

Внешний вид проекта "Падение мяча" представлен на рис. 7.8.

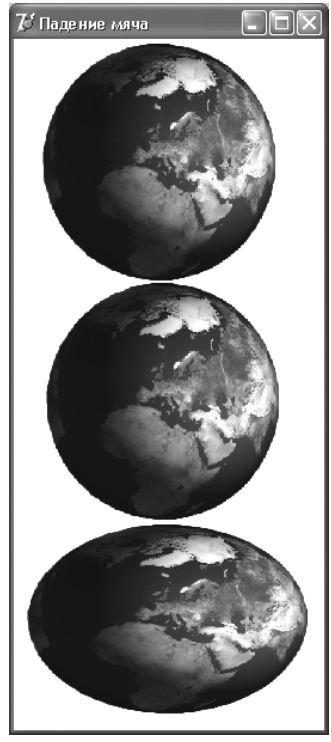


Рис. 7.8. Работа проекта "Падение мяча"

7.2.4. Велосипед

Рассмотрим еще один анимированный проект, использующий компонент `Timer1`, в котором двигаются ноги велосипедиста и спицы колес (рис. 7.9).

В проекте используются следующие константы и переменные:

```
  x0=282;              // центр педали
  y0=411;
```

```
  x1=218;              // сидение
  y1=229;
  R0=65; R1=23;        // радиусы колеса
  R=30;                // радиус вращения педали
  xR1=177; yR1=403;    // центр 1 и 2 колеса
  xR2=448; yR2=400;
var
  Angle1  : real=0;    // угол поворота первой педали
  Angle2  : real=Pi;   // угол поворота второй педали
  Angle3  : real=0;    // угол поворота колес
  StepAngl: real=0.1; // шаг угла
  L       : real;      // длина голени и бедра
```

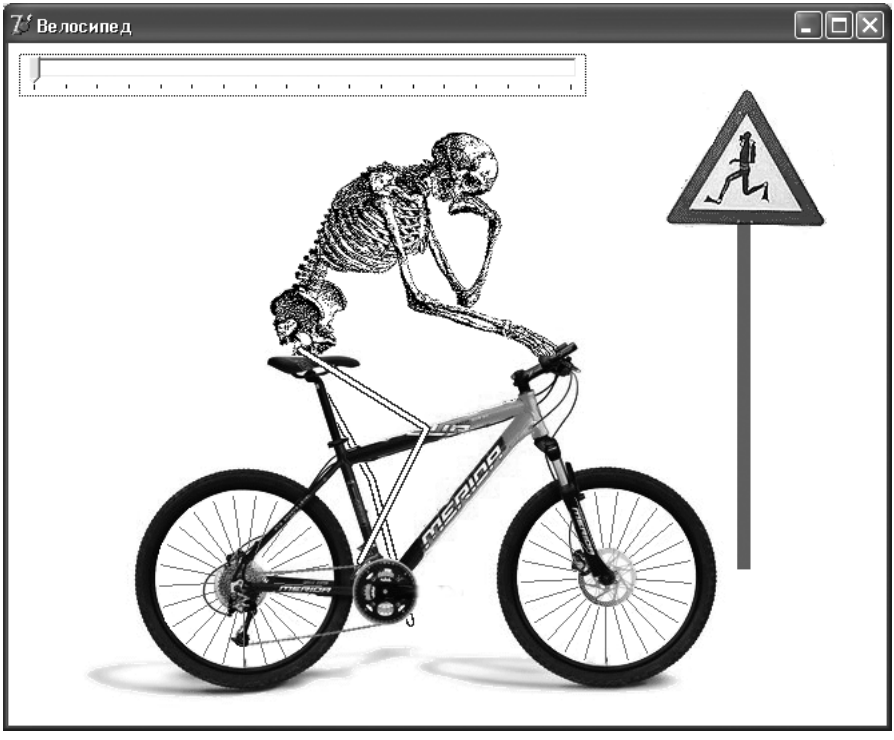


Рис. 7.9. Проект "Велосипед"

В проекте используется два рисунка — Velo1.BMP и Velo2.BMP. На первом изображен велосипед полностью, на втором (прозрачном по белому цвету) — только нижняя часть велосипеда. В момент запуска проекта рисунки загружаются на `Bitmap1` и на `Bitmap2` (листинг 7.21).

Листинг 7.21. Загрузка рисунков

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
  Bitmap:=TBitmap.Create;
  Bitmap.Width:=Width; Bitmap.Height:=Height;
  Bitmap1:=TBitmap.Create;
  Bitmap1.LoadFromFile('Velo1.bmp');
  Bitmap2:=TBitmap.Create;
  Bitmap2.LoadFromFile('Velo2.bmp');
  Bitmap2.TransparentColor:=clWhite;
  Bitmap2.Transparent:=true;
  Bitmap2.TransparentMode:=tmAuto;
  L:=(Sqrt(Sqr(x0-x1)+Sqr(y0-y1))+R)/2;
end;
```

Рисование на канве происходит в такой последовательности:

- выводится полный рисунок;
- рисуется дальняя нога;
- выводится нижняя часть велосипеда;
- рисуется ближняя нога;
- рисуются спицы.

Процесс рисования представлен в листинге 7.22.

Листинг 7.22. Рисование велосипеда

```
procedure TFormMain.DrawBicycle;
begin
  with Bitmap.Canvas do begin
    Brush.Color:=clWhite;
    FillRect(Rect(0,0,Width,Height));
    Draw(0,0,Bitmap1);     // выводится полный рисунок
    DrawFoot(Angle1);      // рисуется дальняя нога
    Draw(0,234,Bitmap2);   // выводится нижняя часть велосипеда
    DrawFoot(Angle2);      // рисуется ближняя нога
    DrawWheel;             // рисуются спицы
  end;
  Canvas.Draw(0,0,Bitmap);
end;
```

Графическое описание движения ноги представлено на рис. 7.10.

Педаль движется по окружности вокруг точки x_0, y_0 и в какой-то момент времени находится в точке x_2, y_2. Нога описывается ломаной из двух звеньев длиной L,

которая начинается в точке x_1, y_1, а заканчивается в точке x_2, y_2. Координаты точки x_2, y_2 в тексте программы соответствуют следующим операторам:

```
x2:=x0+R*cos(Angle); y2:=y0+R*sin(Angle);
```

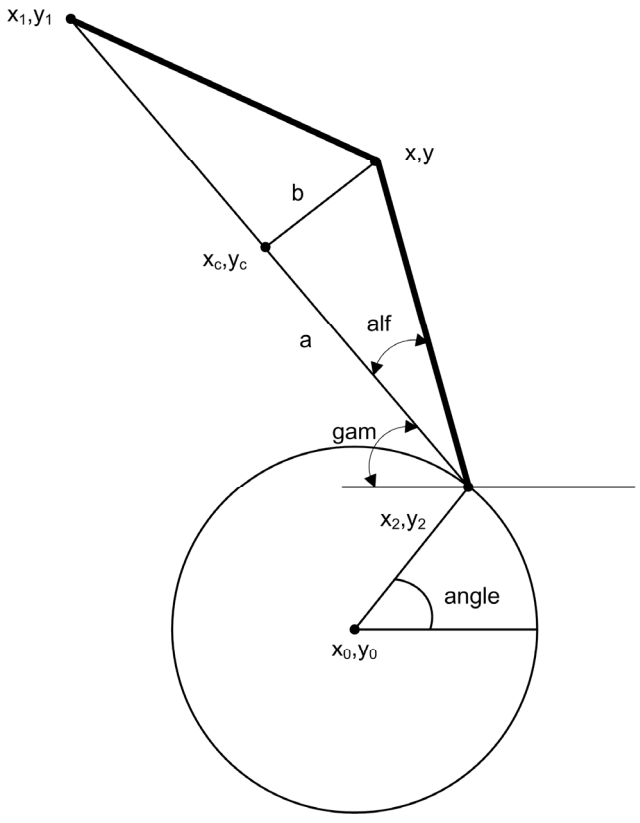


Рис. 7.10. Графическое описание алгоритма движения ноги

Затем вычисляются координаты точки x_c, y_c:

```
xc:=(x1+x2)/2; yc:=(y1+y2)/2;
```

и длины катетов прямоугольного треугольника:

```
b:=Sqrt(Sqr(x2-xc)+Sqr(y2-yc));
a:=Sqrt(L*L-b*b);
```

Это позволяет найти углы :

```
alf:=Arctan2(a,b); gam:=Arctan2(y1-y2,x1-x2);
```

а затем координаты точки x, y:

```
x:=x2+L*cos(gam+alf); y:=y2+L*sin(gam+alf);
```

Для рисования одной ноги предназначена процедура `DrawFoot(angle)` (листинг 7.23).

Листинг 7.23. Рисование одной ноги

```
procedure DrawFoot(angle: real); // нога
var
  x,y,x2,y2,xc,yc,a,b: real;
  alf,gam: real;
begin
  with Bitmap.Canvas do begin
    Pen.Width:=6;
    Pen.Color:=$2f2f2f; //clGray;
    x2:=x0+R*cos(Angle); y2:=y0+R*sin(Angle);
    xc:=(x1+x2)/2; yc:=(y1+y2)/2;
    b:=Sqrt(Sqr(x2-xc)+Sqr(y2-yc));
    a:=Sqrt(L*L-b*b);
    alf:=Arctan2(a,b); gam:=Arctan2(y1-y2,x1-x2);
    x:=x2+L*cos(gam+alf); y:=y2+L*sin(gam+alf);
    MoveTo(Round(x2),Round(y2));
    LineTo(Round(x),Round(y));
    LineTo(Round(x1),Round(y1));
  end;
end;
```

Рисование спиц проще и реализуется процедурой `DrawWheel` (листинг 7.24).

Листинг 7.24. Рисование спиц

```
procedure DrawWheel; // спицы
const n=25;
var
  x1,y1,x2,y2,a: real;
  i: integer;
begin
  with Bitmap.Canvas do begin
    Pen.Width:=1;
    Pen.Color:=clGray;
    for i:=0 to n-1 do begin
      a:=Angle3+2*Pi/n*i;
      x1:=xR1+R0*cos(a); y1:=yR1-R0*sin(a);
      x2:=xR1+R1*cos(a); y2:=yR1-R1*sin(a);
      MoveTo(Trunc(x2),Trunc(y2)); LineTo(Trunc(x1),Trunc(y1));
      x1:=xR2+R0*cos(a); y1:=yR2-R0*sin(a);
      x2:=xR2+R1*cos(a); y2:=yR2-R1*sin(a);
      MoveTo(Trunc(x2),Trunc(y2)); LineTo(Trunc(x1),Trunc(y1));
    end;
  end;
end;
```

Полностью проект представлен на компакт-диске в папке **Примеры | Глава 7 | Velo**.

7.3. Рисование на канве принтера

Для работы с принтером должен быть подключен модуль `Printers`. С канвой принтера можно работать как с текстовым файлом или как с полотном.

При работе с принтером как с текстовым файлом последовательность действий должна быть такой (листинг 7.25).

Листинг 7.25. Работа с принтером как с текстовым файлом

```
AssignPrn(f); Rewrite(f);
with Printer.Canvas.Font do begin
  Name:='Courier New Cyr';
  Style:=[fsBold];
end;
Writeln(f,s);
CloseFile(f);
```

Можно добраться к канве принтера с помощью методов `BeginDoc` и `EndDoc`. В этом случае последовательность действий должна быть такой (листинг 7.26).

Листинг 7.26. Работа с канвой принтера

```
Printer.BeginDoc;
Printer.Canvas…
Printer.EndDoc;
```

Методы `BeginDoc` и `EndDoc` в каком-то смысле эквивалентны процедурам `Rewrite` и `CloseFile`, используемым при работе с файлами. Отметим, что печать не начнется, пока не будет выполнена команда `EndDoc`.

Ширина и высота полотна принтера доступны через свойства `Printer.PageWidth` и `Printer.PageHeight`. Закончить печать на одной странице и начать на новой можно с помощью метода `Printer.NewPage`, который должен вызываться до метода `EndDoc`.

Далее приводится текст процедуры, печатающей `Image1.Picture.Graphic` с максимально допустимой шириной или высотой (листинг 7.27).

Листинг 7.27. Пример работы с канвой принтера

```
procedure TForm1.Print1Click(Sender: TObject);
var
  AspectRatio: Single;
  W, H: Single;
```

```
begin
  if PrintDialog1.Execute then begin
    Printer.BeginDoc;
    try
      W := Image1.Picture.Width;
      H := Image1.Picture.Height;
      AspectRatio := W / H;
      if (W<Printer.PageWidth) and (H<Printer.PageHeight) then
      begin
        if W < H then begin
          H := Printer.PageHeight;
          W := H * AspectRatio;
        end
        else begin
          W := Printer.PageWidth;
          H := W / AspectRatio;
        end
      end;
      if W > Printer.PageWidth then begin
        W := Printer.PageWidth;
        H := W / AspectRatio;
      end;
      if H > Printer.PageHeight then begin
        H := Printer.PageHeight;
        W := H * AspectRatio;
      end;
      Printer.Canvas.StretchDraw(Rect(0,0,Trunc(W),Trunc(H)),
        Image1.Picture.Graphic);
    finally
      Printer.EndDoc;
    end;
  end;
end;
```

7.4. Векторный стиль линии

Объектно-ориентированный язык Pascal, используемый в Delphi, обладает не слишком богатыми возможностями при задании стиля пера (рис. 2.10). Свойство `Style` определяет стиль линии, рисуемой пером. Возможные значения этого свойства представлены в табл. 2.6.

Столь бедный набор стилей не устраивает, например, геологов. Некоторые условные обозначения, принятые у геологов для стилей линий геграфических карт, приведены на рис. 7.11. Из этого рисунка видно, что, например, стили линий условных

обозначений № 83, 84, 86–93 не могут быть реализованы стандартными средствами Delphi.

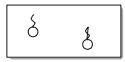
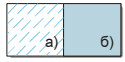
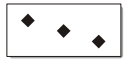
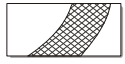
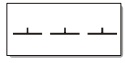
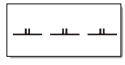
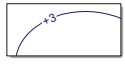
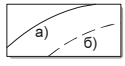
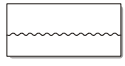
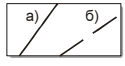
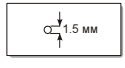
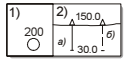
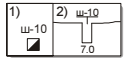
81		Выделения газа (газоносность)
82		Водоносность: а) слабая; б) обильная
83		Жидко-капельная нефть (следы по трещинам)
84		Пропитанные нефтью породы-коллекторы (в виде пленок и пятен)
85		Сильно пропитанные нефтью породы
86		Подошва многолетнемерзлых пород
87		Подошва криолитозоны
88		Геоизотермы
89		Геологические границы: а) установленные; б) предполагаемые
90		Стратиграфическое несогласное залегание пород
91		Разрывные нарушения: а) установленные; б) предполагаемые
92		Зоны дробления
93		Зоны интенсивной трещиноватости и развития зеркал скольжения
94		Кора выветривания и ее возраст
95		Точки геологических наблюдений
96		Скважина, ее номер, абс.отметка и глубина: 1) на плане; 2) на разрезе: - а) лежащая в плоскости разреза - б) спроецированная на плоскость разреза
97		Шурф, его номер и глубина (м): 1) на плане; 2) на разрезе

Рис. 7.11. Геологические условные обозначения линий

Для решения задачи отображения нестандартных стилей линий в данном разделе предлагается механизм рисования линий, использующий API-функции.

7.4.1. Рисование линии стандартными способами

Изображение линий, например, кривых Безье, стандартным (предусмотренным в Delphi) стилем `Pen.Style` не составляет особого труда, если координаты узловых точек заданы массивом `Bezier`.

```
Pen.Style:=psSolid;
PolyBezier(Bezier);
```

7.4.2. Применение векторного стиля линии

Прежде всего введем два понятия, используемых в Windows: *регион* и *траектория*. Понятия регион и траектория поверхностно описаны в документации Delphi. Но Юань Фень провел подробное исследование и в книге [125] достаточно подробно описывает эти понятия.

Регион (region) — совокупность точек на поверхности графического устройства, получающаяся в результате комбинации прямоугольников, многоугольников или эллипсов.

Для регионов определены операции заливки, инвертирования, обводки, отсечения и проверки принадлежности. Из этого списка, может быть, только операция отсечения требует дополнительного пояснения: в этом случае на поверхности графического устройства рисуются только те пикселы, которые попадают в регион. Регион описывается структурой данных переменного размера `REGION` и для работы с этой структурой предназначен целый набор API-функций. Важнейшими данными в структуре `REGION` является массив структур `SCAN` — скан-линий. Структура `SCAN` содержит информацию об одной "строке развертки" региона, высота которой обычно равна одному пикселу.

Траектория (path) — замкнутая совокупность геометрических фигур, к которой применяются операции заливки и (или) обводки. Траектории — это набор прямых и кривых линий, хранящийся внутри GDI (Graphics Device Interface — интерфейс графических устройств). Траектории были введены в Windows в версии Windows NT, они также поддерживаются и в Windows 95. Структура `PATH`, описывающая траектории, в отличие от структуры `REGION` имеет фиксированный размер. Эта структура содержит список структур `PATHDT`, каждая из которых описывает одну часть фигуры, образующей траекторию. В свою очередь структура `PATHDT` представляет группу точек на кривой, обладающих общими атрибутами. Структура `PATH` позволяет представить произвольные комбинации из отрезков прямых линий, кривых Безье и включать в построение траекторий вызовы метода `TextOut`. Отметим, что при построении траекторий для символов можно использовать только шрифты

TrueType. В этом случае в траектории записываются только контуры символов, представляющие собой кривые Безье. В траектории также можно включать эллиптические кривые. Например, при построении траектории можно использовать функции `AngleArc`, `Arc`, `ArcTo`, `Ellipse`, `Pie` и т. д. При этом эллиптические кривые разбиваются на последовательности кривых Безье. На первый взгляд траектории и регионы похожи, и, действительно, можно конвертировать траекторию в регион и использовать траекторию для отсечения.

Для создания нестандартного стиля линия, о котором пойдет речь в этом параграфе, важным понятием является понятие траектории.

Для того чтобы начать определение траектории, необходимо вызвать функцию:

```
BeginPath(hdc);
```

После этого вызова любая рисуемая линия (прямая, дуга или кривая Безье) будет запоминаться внутри GDI как часть траектории и не будет воспроизводиться в контексте устройства. Часто траектории состоят из связанных друг с другом линий. Для создания связанных линий используются функции `LineTo`, `PolylineTo` и `BezierTo`, каждая из которых рисует линию, начинающуюся из текущего положения пера. Если изменить текущее положение пера, используя функцию `MoveToEx`, или вызвать какую-нибудь другую функцию рисования линии, или вызвать одну из функций окна или области вывода, влияющих на изменение текущего положения пера, то создается новая подтраектория в рамках траектории. Таким образом, траектория состоит из одной или нескольких подтраекторий, причем каждая подтраектория — это серия связанных линий.

Каждая подтраектория в рамках траектории может быть открыта или закрыта. Во-первых, подтраектория закрывается, если в ней первая точка первой связанной линии и последняя точка последней связанной линии совпадают. Во-вторых, подтраектория закрывается вызовом функции `CloseFigure`. Эта функция закрывает подтраекторию, добавляя, если необходимо, прямую линию к первой точке первой связанной линии. Любой последующий вызов функции рисования начинает новую подтраекторию. Определение траектории завершается вызовом функции:

```
EndPath(hdc);
```

Теперь можно вызвать одну из следующих пяти функций:

1. `StrokePath(hdc);`
2. `FillPath(hdc);`
3. `StrokeAndFillPath(hdc);`
4. `hRgn = PathToRegion(hdc);`
5. `SelectClipPath(hdc, iCombine);`

Любая из этих функций уничтожает определенную траектории после завершения.

Функция `StrokePath` рисует траекторию, используя текущее перо.

Другие четыре функции закрывают все открытые траектории прямыми линиями. Функция `FillPath` закрашивает траекторию, используя текущую кисть, в соответ-

ствии с текущим режимом закрашивания многоугольников. Функция `StrokeAndFillPath` выполняет оба указанных действия. Можно также преобразовать траекторию в регион или использовать траекторию как область отсечения. Параметр `iCombine` — одна из констант региона (RGN-констант), используемых в функции `CombineRgn`, показывает, как траектория должна комбинироваться с текущим регионом отсечения.

Траектории являются более гибкими структурами по сравнению с регионами для закрашивания и отсечения, потому что регион может быть определен только как комбинация прямоугольников, эллипсов и многоугольников. Траектории же могут состоять из кривых Безье и дуг (в Windows NT). В GDI траектории и регионы хранятся совершенно по-разному. Траектории — это определение набора прямых и кривых, а регион — набор скан-линий.

Для применения стиля, заданного пользователем, необходимо использовать API-функцию `LineDDA`, определенную в модуле Windows. Эта функция использует массив точек и указатель на пользовательскую функцию `LineDrawFunc`.

Процедура `LineDDA` в процессе рисования линии вызывает пользовательскую процедуру `LineDrawFunc`, которая обеспечивает рисование любого пользовательского стиля линии. Кроме этой функции необходимо использовать процедуры `BeginPath`, `EndPath`, `FlattenPath`, `GetPath`, назначение которых будет описано далее.

API-функция `LineDDA` определяет, как должны быть нарисованы пикселы линии, определенной начальной и конечной точками (листинг 7.28).

Листинг 7.28. Заголовок API-функции *LineDDA*

```
function LineDDA (
  nXStart,  // x-координата отправной точки линии
  nYStart,  // y-координата отправной точки линии
  nXEnd,    // x-координата точки окончания линии
  nYEnd:    // y-координата точки окончания линии
        integer;
  lpLineFunc: LINEDDAPROC;//адрес пользовательской функции
  lpData    : LPARAM      //адрес канвы для рисования
): boolean;
```

В этой функции параметры имеют следующий смысл:

- `nXStart` — определяет `x`-координату начальной точки линии;
- `nYStart` — определяет `y`-координату начальной точки линии;
- `nXEnd` — определяет `x`-координату конечной точки линии;
- `nYEnd` — определяет `y`-координату конечной точки линии;
- `lpLineFunc` — определяет адрес определенной приложением функции;
- `lpData` — адрес канвы рисования.

Если функция выполнена успешно, то возвращаемое значение `true`, иначе — возвращается значение `false`.

Функция `LineDDA` передает координаты для каждой точки линии (если это не конечная точка линии), определенной приложением функции `LineDrawFunc`.

Пользовательская функция `LineDrawFunc` вызывается в любой точке линии, но на рисование можно накладывать условия в зависимости от значения счетчика точек `CountPoint`. Необходимо обратить внимание на то, что глобальная переменная — счетчик `CountPoint` должна увеличиваться на `1` при каждом обращении к функции.

Вернемся к осуждению процедур `BeginPath`, `EndPath`, `FlattenPath`, `GetPath`.

- Функция `BeginPath` открывает траекторию (path bracket) на указанном контексте устройства:

  ```
  function BeginPath (hdc: HDC): boolean;
  ```

  Параметр `Hdc` идентифицирует контекст устройства. Если функция выполнена успешно, то возвращаемое значение равно `true`, иначе возвращается значение `false`.

> ***ЗАМЕЧАНИЕ***
>
> После того как траектория открыта, приложение может начать вызывать функции рисования, для того чтобы определить точки, которые лежат на линии. Приложение может закрыть открытую траекторию, вызвав функцию `EndPath`.

В табл. 7.1 представлен список функций рисования (для Windows NT), которые определяют точки при рисовании.

Таблица 7.1. *Список функций рисования*

`AngleArc`	`LineTo`	`Polyline`
`Arc`	`MoveToEx`	`PolylineTo`
`ArcTo`	`Pie`	`PolyPolygon`
`Chord`	`PolyBezier`	`PolyPolyline`
`CloseFigure`	`PolyBezierTo`	`Rectangle`
`Ellipse`	`PolyDraw`	`RoundRect`
`ExtTextOut`	`Polygon`	`TextOut`

- Функция `EndPath` закрывает траекторию на указанном контексте устройства:

  ```
  function EndPath (hdc: HDC): boolean;
  ```

  Параметр `Hdc` идентифицирует контекст устройства, в который выбрана траектория. Если функция выполнена успешно, то возвращаемое значение `true`, иначе возвращается значение `false`.

- Функция `FlattenPath` преобразовывает кривые в траектории, которая выбрана в текущем контексте устройства, превращая каждую кривую в последовательность линий.

  ```
  function FlattenPath (hdc: HDC): boolean;
  ```

  Параметр `Hdc` идентифицирует контекст устройства, который содержит траекторию. Если функция выполнена успешно, то возвращаемое значение `true`, иначе возвращается значение `false`.

- Функция `GetPath` находит координаты, определяющие конечные точки линий и контрольные точки кривых, найденных на траектории, которая выбрана в указанном контексте устройства (листинг 7.29).

Листинг 7.29. Заголовок функции *GetPath*

```
function GetPath (
    hdc: HDC,          // дескриптор контекста устройства
    lpPoints: LPPOINT, // адрес массива, получающего вершины траектории
    lpTypes: LPBYTE,   // адрес массива типов вершин траектории
    nSize: integer     // число точек, определяющих траекторию
  ): integer;
```

Параметры функции `GetPath` имеют следующий смысл:

`Hdc` — идентифицирует контекст устройства, который содержит траекторию;

`LpPoints` — массив типа `TPoint`, который содержит конечные и промежуточные точки кривой;

`lpTypes` — массив байтов, в который помещаются типы вершины. Каждый элемент массива может принимать одно из значений, представленных в табл. 7.2.

Таблица 7.2. *Значения параметра lpTypes*

Тип	Описание
`PT_MOVETO=6`	Определяет, что соответствующая точка в параметре `lpPoints` начинает непересекающуюся линию
`PT_LINETO=2`	Определяет, что предыдущая точка и соответствующая точка в `lpPoints` — конечные точки линии
`PT_BEZIERTO=4`	Определяет, что соответствующая точка в `lpPoints` — промежуточная точка или конечная точка для кривой Безье. Значения `PT_BEZIERTO` всегда добавляются по три. Предыдущая точка на их траектории определяет начальную точку для кривой Безье. Первые две точки `PT_BEZIERTO` — промежуточные точки, а третья точка `PT_BEZIERTO` — конечная. Значение `PT_LINETO` или `PT_BEZIERTO` могут быть объединены операцией ИЛИ со следующей точкой траектории для того, чтобы указать, что соответствующая точка — последняя точка в линии и линия должна быть закрыта

***Таблица 7.2** (окончание)*

Тип	Описание
PT_CLOSEFIGURE=1	Определяет, что рисунок должен быть автоматически закрыт после рисования кривой

nSize — определяет общее число элементов массива типа TPoint, которые могут быть помещены в массив, на который указывает lpPoints. Это значение должно быть такое же, как число байтов, которые могут быть помещены в массив, на который указывает lpTypes.

Если параметр nSize отличен от нуля, то возвращаемое значение — число перечисленных точек. Если nSize=0, то возвращаемое значение равно количеству точек на траектории и GetPath ничего не изменяет в буфере. Если nSize отлично от нуля и меньше, чем число точек на траектории, то возвращаемое значение отрицательно и равно -1.

Замечания

Контекст устройства, идентифицированный параметром hdc, должен содержать закрытую траекторию.

Функцию FlattenPath необходимо вызвать раньше, чем функцию GetPath, для того, чтобы преобразовать все кривые в траектории.

7.4.3. Проект "Рисование линии произвольным стилем"

Бедем рисовать кривую Безье с двухбайтовым шаблоном $AA00, проводя через каждые 16 точек перпендикуляр к линии длиной 10 пикселов.

Тип данных TBezier позволяет ввести семь узловых точек для двух фрагментов кривой Безье.

```
type
  TBezier=array[0..6] of TPoint;
var
  Bezier  : TBezier;
```

В момент запуска проекта элементам массива Bezier присваиваются некоторые значения (листинг 7.30).

Листинг 7.30. Инициализация данных проекта

```
procedure TFormMain.FormCreate(Sender:TObject);
begin
  drawing:=false;
```

```
  DragPoint:=-1;
  Bitmap:=TBitmap.Create;
  Bitmap.Canvas.Brush.Color:=Color;
  Bitmap.Width:=Width;
  Bitmap.Height:=Height;
  Bezier[0].X:=100; Bezier[0].Y:=100;
  Bezier[3].X:=300; Bezier[3].Y:=300;
  Bezier[1].X:=Bezier[0].X+Round((Bezier[3].X-Bezier[0].X)/3);
  Bezier[1].Y:=Bezier[0].Y+Round((Bezier[3].Y-Bezier[0].Y)/3);
  Bezier[2].X:=Bezier[0].X+Round(2*(Bezier[3].X-Bezier[0].X)/3);
  Bezier[2].Y:=Bezier[0].Y+Round(2*(Bezier[3].Y-Bezier[0].Y)/3);
  Bezier[6].X:=500; Bezier[6].Y:=400;
  Bezier[4].X:=Bezier[3].X+Round((Bezier[6].X-Bezier[3].X)/3);
  Bezier[4].Y:=Bezier[3].Y+Round((Bezier[6].Y-Bezier[3].Y)/3);
  Bezier[5].X:=Bezier[3].X+Round(2*(Bezier[6].X-Bezier[3].X)/3);
  Bezier[5].Y:=Bezier[3].Y+Round(2*(Bezier[6].Y-Bezier[3].Y)/3);
  ShowLine;
end;
```

Рисование осуществляется процедурой `ShowLine` (листинг 7.31), которая очищает канву, рисует касательные векторы к началу и концу фрагмента кривой Безье, рисует кривую Безье и узловые точки (рис. 7.12).

Листинг 7.31. Рисование кривой Безье произвольным стилем

```
procedure TFormMain.ShowLine;
var i: integer;
begin
  with Bitmap.Canvas do begin // очистка канвы
    Brush.Style:=bsSolid;
    FillRect(Rect(0,0,Width,Height));
    Pen.Color:=clBlue;
  end;
  with Bitmap.Canvas do begin
     // касательные векторы (усы)
     Pen.Style:=psDot;
     Pen.Width:=1;
     Pen.Color:=clDkGray;
     MoveTo(Bezier[0].X,Bezier[0].Y);
     LineTo(Bezier[1].X,Bezier[1].Y);
     MoveTo(Bezier[3].X,Bezier[3].Y);
     LineTo(Bezier[2].X,Bezier[2].Y);
     // кривая Безье
     DrawBezier(clBlack,Bitmap.Canvas);
     // узловые точки (эллипсы)
     Pen.Style:=psSolid;
```

```
      Pen.Color:=clRed;
      Brush.Style:=bsClear;
      for I:=0 to Length(Bezier)-1 do
      with Bezier[I] do
        Ellipse(X-3,Y-3,X+4, Y+4)
  end;
  Canvas.Draw(0,0,Bitmap);
end;
```

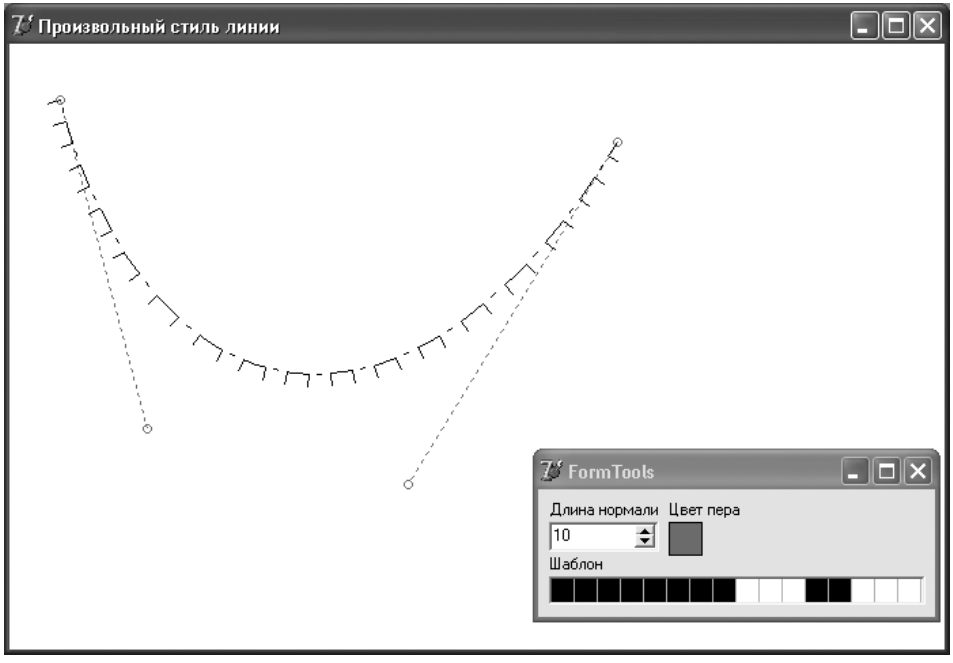


Рис. 7.12. Кривая Безье с произвольным стилем

Процедура `ShowLine` вызывает процедуру `DrawBezier` (листинг 7.32), которая выполняет следующие действия:

- заполняет значения переменных `Size`, `PointBuf` и `TypeBuf`;
- проходит циклом по всем элементам массива `PointBuf`, вычисляя проекции `dx` и `dy` вектора нормали длиной `LengthNorm` к линии;
- вызывает API-функцию `LineDDA`.

Листинг 7.32. Настройка стиля линии

```
procedure TFormMain.DrawBezier(Color: TColor; Canvas: TCanvas);
var i,Size   : integer;
    PointBuf : array of TPoint;
```

```
    TypeBuf  : array of byte;
    D        : real;
begin
  Canvas.Pen.Color:=Color;
  BeginPath(Canvas.Handle);    // открыть траекторию
  Canvas.PolyBezier(Bezier);  // перечислить рисующие методы
  EndPath(Canvas.Handle);      // закрыть траекторию
  FlattenPath(Canvas.Handle); // преобразовывает кривые в траекторию
  Size:=GetPath(Canvas.Handle,i,i,0);
  SetLength(PointBuf,Size);
  SetLength(TypeBuf,Size);
  GetPath(Canvas.Handle,PointBuf[0],TypeBuf[0],Size);
  CountPoint:=0;
  for i:=1 to Size-1 do begin
    D:=Sqrt(Sqr(PointBuf[I-1].X-PointBuf[I].X)+
            Sqr(PointBuf[I-1].Y-PointBuf[I].Y));
    // dx= LengthNorm*cos(a)
    dx:=Round(LengthNorm*(PointBuf[i-1].Y-PointBuf[i].Y)/D);
    // dy= LengthNorm*sin(a)
    dy:=Round(LengthNorm*(PointBuf[i].X-PointBuf[i-1].X)/D);
    LineDDA(PointBuf[i-1].X,PointBuf[i-1].Y,
            PointBuf[i].X,PointBuf[i].Y,
            @LineDrawFunc,Integer(Canvas))
  end
end;
```

В нашем случае для двух фрагментов кривой Безье, опирающейся на семь точек, значение параметра `Size` равно 5. В массиве `PointBuf` будут содержаться координаты пяти точек: трех узловых и двух промежуточных (по одной на каждый сегмент кривой Безье). В элементах массива `TypeBuf` будут содержаться следующие значения: `6`, `2`, `2`, `2`, `2`.

Функция `LineDDA` вызывает пользовательскую процедуру `LineDrawFunc` (листинг 7.33).

Листинг 7.33. Пользовательская процедура *LineDrawFunc*

```
procedure LineDrawFunc(X,Y: Integer; Canvas: TCanvas); stdcall;
begin
  with Canvas do begin
    if CountPoint mod 16=0 then begin
      Pen.Style:=psSolid;
      Pen.Width:=1;
      MoveTo(X,Y); LineTo(X+dx,Y+dy)
    end;
    if $AA00FF11 and (1 shl (CountPoint div 2))<>0 then
      Pixels[x,y]:=Pen.Color;
```

```
  end;
  Inc(CountPoint); CountPoint:=CountPoint mod 32;
end;
```

Процедура `LineDrawFunc` в каждой 16-ой точке рисует нормаль (`dx`, `dy`) и, в соответствии с ненулевыми битами шаблона `w`, после проверки условия

```
if w and (1 shl (CountPoint div 2))<>0 then
```

рисует или не рисует точку на линии.

Полностью проект приведен на компакт-диске в папке **Примеры | Глава 7 | Произвольный стиль линии**.

7.5. Деформация изображений

В многочисленных графических редакторах используются различные фильтры, изменяющие первоначальные изображения. Некоторые из этих фильтров меняют геометрию изображения. В модуле OpenGL, например, такая технология называется наложением текстуры на грани [64, 72, 80, 90, 105].

В данном разделе описывается проект, решающий задачу деформации изображения из прямоугольной области $(0,0)$, (I_2, J_2) в произвольный четырехугольник (u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3). Вершины этого четырехугольника (рис. 7.13) можно перетаскивать указателем мыши.

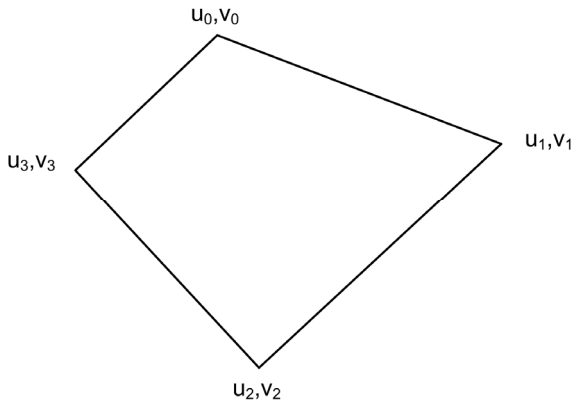


Рис. 7.13. Деформированный четырехугольник

На форму проекта (рис. 7.14) вынесены два компонента для рисования: `Image1` и `Image2`. Пункт основного меню **File** содержит три подпункта — `Open1Click`, `N1Click` и `N2Click`, предназначенные для выбора файла формата BMP, прямой и обратной деформации четырехугольника.

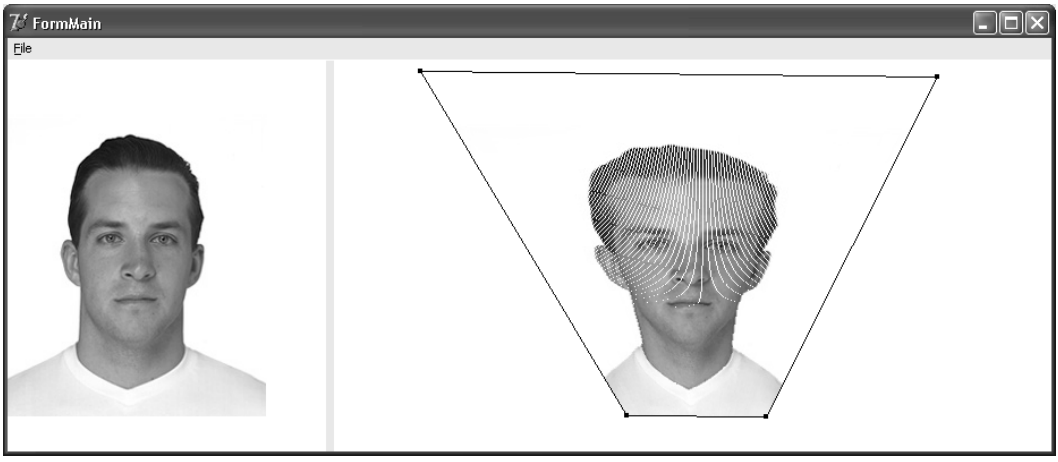


Рис. 7.14. Прямая деформация четырехугольников в проекте "Деформация изображений"

После выбора графического файла с помощью стандартного метода открытия графических файлов `OpenPictureDialog1`, изображение попадает в `Bitmap` и выводится на `Image1` (листинг 7.34).

Листинг 7.34. Выбор графического файла

```
if OpenPictureDialog1.Execute then
begin
  BitMap.LoadFromFile(OpenPictureDialog1.FileName);
  Image1.Canvas.Draw(0,0, Bitmap);
end;
```

Закон преобразования прямоугольной области (0,0), (I_2, J_2) в четырехугольник (u_0, v_0), (u_1, v_1), (u_2, v_2), (u_3, v_3) выразим в виде квазилинейного соотношения:

$$u = A_1 \times x + B_1 \times x \times y + C_1 \times y + D_1;$$
$$v = A_2 \times x + B_2 \times x \times y + C_2 \times y + D_2. \qquad (7.9)$$

После подстановки всех четырех вершин и решения системы уравнений, для коэффициентов этого линейного преобразования получаем:

$$D_1 = u_0;$$
$$D_2 = v_0;$$
$$A_1 = (u_1 - u_0) / I_2;$$
$$A_2 = (v_1 - v_0) / I_2;$$
$$C_1 = (u_3 - u_0) / J_2;$$

$$C_2 = (v_3 - v_0)/J_2;$$

$$B_1 = (u_2 - u_3 - u_1 + u_0)/I_2/J_2;$$

$$B_1 = (v_2 - v_3 - v_1 + v_0)/I_2/J_2.$$

Затем, в процедуре `Texture1` (листинг 7.35) все точки `Bitmap.Canvas.Pixels[x,y]` преобразуются в точки `Bitmap2.Canvas.Pixels[i,j]`. Обратите внимание на то, что преобразуются не пикселы канвы компонента `Image2`, а пикселы битовой карты `Bitmap2` в памяти, и только после полной подготовки `Bitmap2` выводится на канву `Image2`. Это значительно ускоряет работу программы.

Листинг 7.35. Преобразование пикселов канвы

```
procedure Texture1;
var
  i,j,x,y: integer;
begin
  I2:=Bitmap.Width; J2:=Bitmap.Height;
  D1:=u[0]; D2:=v[0];
  A1:=(u[1]-u[0])/I2;  A2:=(v[1]-v[0])/I2;
  C1:=(u[3]-u[0])/J2;  C2:=(v[3]-v[0])/J2;
  B1:=(u[2]-u[3]-u[1]+u[0])/I2/J2;
  B2:=(v[2]-v[3]-v[1]+v[0])/I2/J2;
  with Bitmap2,Canvas do
  FillRect(Rect(0,0,Width,Height));
  for x:=0 to I2-1 do
  for y:=0 to J2-1 do begin
    i:=Round(A1*x+B1*x*y+C1*y+D1);
    j:=Round(A2*x+B2*x*y+C2*y+D2);
    Bitmap2.Canvas.Pixels[i,j] := Bitmap.Canvas.Pixels[x,y];
  end;
end;
```

У этого простого метода есть два недостатка:

- для больших `Bitmap` он будет работать долго;
- для маленьких `Bitmap`, как это видно на рис. 7.14, он не будет плотно заполнять все точки `Image2`.

Немного лучше будет работать проект, в котором используется обратное преобразование от четырехугольника (u_0,v_0), (u_1,v_1), (u_2,v_2), (u_3,v_3) к прямоугольнику $(0,0)$, (I_2,J_2).

Из уравнения (7.9) исключим x. Для переменной y получаем квадратное уравнение:

$$A \times y^2 + B \times y + C = 0, \tag{7.10}$$

где:

$$A = C_2 \times B_1 - C_1 \times B_2;$$

$$B = B_2 \times (u - D_1) - B_1 \times (v - D_2) - A_2 \times C_1 + C_2 \times A_1;$$

$$C = A_2 \times (u - D_1) - A_1 \times (v - D_2).$$

Из уравнения (7.9) для переменных x и y получаем следующие соотношения:

$$y = (-B + \sqrt{D})/(2 \times A);$$

$$x = (u - D_1 - C_1 \times y)/(A_1 + B_1 \times y).$$

В этом случае процедура `Texture2` заполнения четырехугольника выглядит следующим образом (листинг 7.36).

Листинг 7.36. Обратное преобразование пикселов канвы

```
procedure Texture2;
var
  x,y,A,B,C,D: real;
  i,j,H1,H2,ut,vt: integer;
begin
  I2:=Bitmap.Width; J2:=Bitmap.Height;
  H1:=Bitmap2.Width; H2:=Bitmap2.Height;
  D1:=u[0]; D2:=v[0];
  A1:=(u[1]-u[0])/I2;  A2:=(v[1]-v[0])/I2;
  C1:=(u[3]-u[0])/J2;  C2:=(v[3]-v[0])/J2;
  B1:=(u[2]-u[3]-u[1]+u[0])/I2/J2;
  B2:=(v[2]-v[3]-v[1]+v[0])/I2/J2;
  with Bitmap2,Canvas do
  FillRect(Rect(0,0,Width,Height));
  for ut:=0 to H1-1 do
  for vt:=0 to H2-1 do
  if PointInPolygon(ut,vt) then begin
    A:=C2*B1-C1*B2;
    B:=(B2*(ut-D1)-A2*C1+C2*A1+(D2-vt)*B1);
    C:=A2*(ut-D1)+A1*(D2-vt);
    D:=B*B-4*A*C;
    if D>=0 then begin
      y:=(-B+Sqrt(D))/(2*A); x:=(ut-D1-C1*y)/(A1+B1*y);
      i:=Round(x); j:=Round(y);
      Bitmap2.Canvas.Pixels[ut,vt] := Bitmap.Canvas.Pixels[i,j];
    end;
  end;
end;
```

В процедуре обратного преобразования пикселов канвы появилось два ограничения: во-первых, функция `PointInPolygon` проверяет, принадлежит ли точка `ut,vt` четырехугольнику, во-вторых, из двух решений квадратного уравнения берется только одно. Обсуждение функции `PointInPolygon`, проверяющей принадлежность точки четырехугольнику, оставим до *главы 11*, в которой показано использование этой функции в более общем случае.

Обратное преобразование, как и следовало ожидать, заполняет четырехугольник плотно (рис. 7.15).

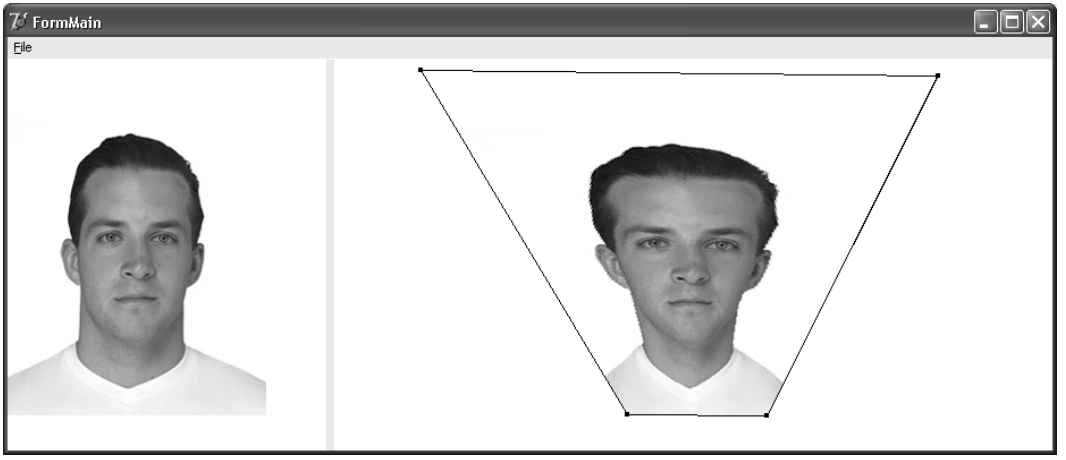


Рис. 7.15. Обратная деформация четырехугольников в проекте "Деформация изображений"

У обратного механизма деформации также есть свои недостатки. Если плотность точек во втором четырехугольнике меньше, чем в прямоугольнике, то у нескольких точек может быть одна и та же точка-прообраз в прямоугольнике. Особенно хорошо это видно для маленьких изображений. В модуле OpenGL для решения этой проблемы предлагается целых шесть методов. Первый, самый простой, использован у нас: для цвета точки в прямоугольнике используется цвет ближайшей точки. Второй метод использует четыре ближайших точки и осредняет их цвета. Остальные методы используют, если это необходимо, более мелкие копии прямоугольников. Более подробно мы обсудим эту проблему в *главе13*.

Обсудим теперь реализацию механизма перетаскивания вершин четырехугольника. На компоненте `Image2` рисуются четыре точки с координатами `u,v: array[0..3] of integer`, в которых находятся вершины деформированного четырехугольника. Эти точки можно перетаскивать указателем мыши. Начало перетаскивания обрабатывается процедурой `Image2MouseDown` (листинг 7.37).

Листинг 7.37. Начало перетаскивания вершины четырехугольника

```
procedure TForm1.Image2MouseDown(Sender: TObject;
       Button: TMouseButton;  Shift: TShiftState;
       X, Y: Integer);
begin
  //определить захваченную точку
  IndexPoint:=0;
  drawing:=(u[0]-3<=x) and (x<=u[0]+3) and (v[0]-3<=y)
    and (y<=v[0]+3);
  while not drawing and (IndexPoint<3) do  begin
    Inc(IndexPoint);
    drawing:=(u[IndexPoint]-3<=x) and (x<=u[IndexPoint]+3) and
      (v[IndexPoint]-3<=y) and (y<=v[IndexPoint]+3);
  end;
  //если захвачена точка с номером IndexPoint, то начать
  //перетаскивание
  if drawing then begin
    xt:=x; yt:=y;  Image2.Canvas.Pen.Mode:=pmNotXor;
  end;
end;
```

В этой процедуре сначала определяем номер точки, в окрестности которой произошел щелчок указателя мыши. Затем, если захвачена точка с номером `IndexPoint`, начинаем перетаскивание, запоминая начальную точку (операторы `xt:=x; yt:=y`) и меняя режим рисования `Image2.Canvas.Pen.Mode :=pmNotXor`.

Перемещение указателем мыши вершины четырехугольника обрабатывается процедурой `Image2MouseMove` (листинг 7.38), в которой от точки `IndexPoint` рисуется два отрезка: до предыдущей точки с номером `(IndexPoint+3) mod 4` и до следующей точки с номером `(IndexPoint+1) mod 4`. Рисование реализовано дважды — в старых координатах точки `IndexPoint` и в новых координатах этой точки.

Листинг 7.38. Процедура обработки перемещения указателем мыши вершины четырехугольника

```
procedure TForm1.Image2MouseMove(Sender: TObject; Shift:
  TShiftState; X, Y: Integer);
begin
  if drawing then
  with Image2.Canvas do begin
    MoveTo(u[(IndexPoint+3) mod 4],v[(IndexPoint+3) mod 4]);
    LineTo(xt,yt);
    LineTo(u[(IndexPoint+1) mod 4],v[(IndexPoint+1) mod 4]);
    xt:=x; yt:=y;
    MoveTo(u[(IndexPoint+3) mod 4],v[(IndexPoint+3) mod 4]);
    LineTo(xt,yt);
```

```
    LineTo(u[(IndexPoint+1) mod 4],v[(IndexPoint+1) mod 4]);
  end;
end;
```

Перемещение вершины четырехугольника завершается вызовом процедуры `Image2MouseUp`, в которой запоминаются новые координаты точки `u[IndexPoint]:=x; v[IndexPoint]:=y`, опускается флаг перетаскивания `drawing:=false`, устанавливается нормальный режим рисования `Image2.Canvas.Pen.Mode :=pmCopy` и выводится изображение `ShowRect(0)` (листинг 7.39).

Листинг 7.39. Процедура обработки отпускания кнопки мыши

```
procedure TForm1.Image2MouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if drawing then begin
    u[IndexPoint]:=x; v[IndexPoint]:=y;  drawing:=false;
    Image2.Canvas.Pen.Mode:=pmCopy;
    ShowRect(0);
  end;
end;
```

Полностью проект приводится на компакт-диске в папке **Примеры | Глава 7 | Деформация BMP**. Заметим, что для реальных проектов механизм, реализованный в предложенном виде, применять сложно — все-таки он работает достаточно медленно.

7.6. Растровый редактор

В проекте "Растровый редактор" описывается создание многооконного графического растрового редактора типа редактора `PaintBrush`. Внешний вид такого графического редактора представлен на рис. 7.16.

Проект состоит из трех форм: главная форма `FormMain` (с внешним именем **My Paint**), дочерняя форма `FormEd` (с внешним именем **Form Edit**) и инструментальная форма `FormTools` (с внешним именем **Tools**) (рис. 7.17).

На главной форме находится компонент основного меню `MainMenu`, содержащий пункт **File (New, Open, Save)**. Так как стиль дочерней формы `FormEd.FormStyle=fsMDIChild`, то ее создание (листинг 7.40) происходит не при запуске проекта на выполнение, а в момент выполнения команды **File/New**.

Листинг 7.40. Создание дочерней формы

```
procedure TFormMain.New1Click(Sender: TObject);
begin
  Application.CreateForm(TFormEd, FormEd);
```

```
  FormEd.Show;
  FormTools.Show;
end;
```

Рис. 7.16. Многооконный проект "Растровый редактор"

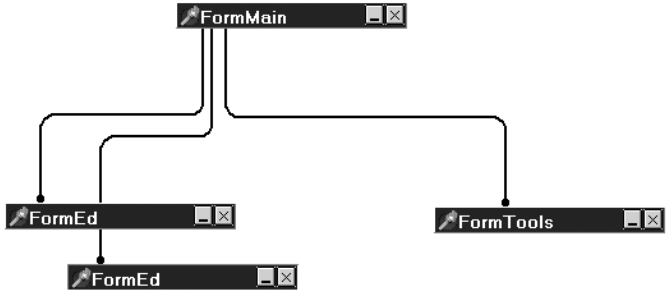


Рис. 7.17. Структура проекта "Растровый редактор"

Такой порядок обеспечивает, с одной стороны, многооконность приложения, однако, с другой стороны, приводит к некоторым проблемам, например, при открытии графического файла из главной формы (листинг 7.41). Сначала необходимо убедиться в том, что у главной формы есть дочерняя, затем в том, что активна форма `TFormEd` и, наконец, использовать стандартный метод `OpenPictureDialog1` для открытия графического файла.

Листинг 7.41. Обращение к дочерней форме при открытии файла

```
procedure TFormMain.Open1Click(Sender: TObject);
begin
  if (ActiveMDIChild<>nil) and (ActiveMDIChild is TFormEd) and
     OpenPictureDialog1.Execute then
  with (ActiveMDIChild as TFormEd),OpenPictureDialog1 do
    Image1.Picture.LoadFromFile(FileName);
end;
```

Аналогичные действия необходимо выполнить при записи изображения в файл (листинг 7.42).

Листинг 7.42. Обращение к дочерней форме при записи

```
procedure TFormMain.Save1Click(Sender: TObject);
begin
  if (ActiveMDIChild<>nil) and (ActiveMDIChild is TFormEd) and
     SavePictureDialog1.Execute then
  with (ActiveMDIChild as TFormEd),SavePictureDialog1 do
    Image1.Picture.SaveToFile(FileName);
end;
```

Инструментальная форма `FormTools` (с внешним именем **Tools**) (рис. 7.18) играет вспомогательную роль.

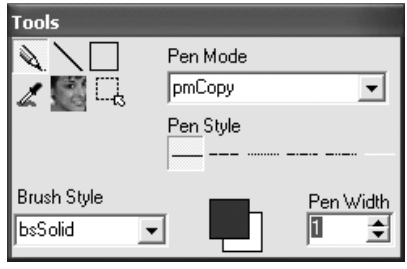


Рис. 7.18. Инструментальная форма **Tools**

Эта форма не должна теряться под другими, поэтому ей назначен стиль `FormStyle=fsStayOnTop`. На инструментальной форме помещен ряд элементов. Раскрывающиеся списки **Pen Mode** и **Bruch Style** в ее верхней и нижней частях содержат перечень режимов пера и стилей кисти соответственно, группа кнопок **Pen Style** с фиксацией предназначена для задания стиля линии, текстовое поле **Pen Width** позволяет выбрать ширину линии, а экран текущих цветов в нижней части формы используется для указания цвета линии и цвета заливки. В левом верхнем углу формы расположена группа кнопок типа `TSpeedButton`, у которых значение свойства `GroupIndex равно 1`. Назначение этих кнопок приведено далее в

табл. 7.3. Все кнопки обладают различными значениями свойства `Tag` и одним обработчиком событий, меняющим при щелчке левой кнопкой мыши значение переменной `fl_tools` (листинг 7.43).

Листинг 7.43. Общий обработчик для группы кнопок

```
procedure TFormTools.ButtonPenClick(Sender: TObject);
begin
  fl_tools:=(Sender as TSpeedButton).tag;
end;
```

Переменная `fl_tools` используется при рисовании различных примитивов, и ее возможные значения заданы в разделе описания констант:

```
const tl_Pen=0;
      tl_Line=1;
      tl_Rect=2;
      tl_Picture=3;
      tl_SetColor=4;
      tl_Move=5; .
```

В табл. 7.3 приведено назначение кнопок инструментальной формы `FormTools`.

Таблица 7.3. *Инструментальные кнопки*

Имя	Значение	Назначение
ButtonPen	tl_Pen=0	Рисовать линию
ButtonLine	tl_Line=1	Рисовать отрезок прямой
ButtonRect	tl_Rect=2	Рисовать прямоугольник
ButtonMove	tl_Move=5	Перемещать прямоугольник
ButtonSetColor	tl_SetColor=4	Задавать цвет пипеткой
ButtonPicture	tl_Picture=3	Вставлять изображение

На инструментальную форму помещен также компонент `CbBrushStyle:TComboBox`, позволяющий изменять стиль заливки области. С этим компонентом связан обработчик `onChange`, изменяющий значение переменной стиля кисти `BrushStyle` (листинг 7.44).

Листинг 7.44. Изменение стиля кисти

```
procedure TFormTools.CbBrushStyleChange(Sender: TObject);
begin
  BrushStyle:=TBrushStyle(CbBrushStyle.ItemIndex);
end;
```

Компонент PenWidth:TSpinEdit задает толщину линии, а два компонента ShPen и ShBrush типа TShape позволяют задавать цвет карандаша и цвет кисти. Оба этих компонента используют стандартный метод ColorDialog (листинг 7.45).

Листинг 7.45. Изменение цвета пера и кисти

```
procedure TFormTools.ShPenMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  if ColorDialog1.Execute then
    ShPen.Brush.Color:=ColorDialog1.Color;
end;
```

Перейдем, наконец, к описанию самой важной дочерней формы FormEd, на которой размещены компоненты ScrollBox1 и Image1. Рисование происходит на канве компонента Image1, и в этом процессе участвуют, как обычно, три обработчика событий: onMouseDown, onMouseMove и onMouseUp. Напомним общую схему рисования. В обработчике события onMouseDown (листинг 7.46), который вызывается при нажатии левой кнопки мыши, переменной drawing присваивается значение true. Затем выполняются действия в зависимости от значения переменной fl_tools.

Листинг 7.46. Начало перемещения указателя мыши

```
procedure TFormEd.Image1MouseDown(Sender: TObject;
  Button: TMouseButton;  Shift: TShiftState; X, Y: Integer);
begin
  with Image1.Canvas do
  case fl_tools of
    …
  end;
end;
```

Если drawing=true, то работает обработчик события перемещения указателя мыши onMouseMove (листинг 7.47).

Листинг 7.47. Перемещение указателя мыши

```
procedure TFormEd.Image1MouseMove(Sender: TObject;
  Shift: TShiftState; X,Y: Integer);
begin
  if drawing then
  with Image1.Canvas do
  case fl_tools of
    …
  end;
end;
```

При отпускании кнопки мыши вызывается обработчик события `onMouseUp`, в нем переменной `drawing` присваивается значение `false`, а также выполняются действия в зависимости от значения переменной `fl_tools` (листинг 7.48).

Листинг 7.48. Завершение перемещения указателя мыши

```
procedure TFormEd.Image1MouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState;
  X,Y: Integer);
begin
  drawing:=false;
  with Image1.Canvas do
  case fl_tools of
    …
  end;
end;
```

Теперь можно приступить к описанию рисования всех примитивов. Начнем с самого простого — рисования произвольной линии. Этот примитив рисуется при `fl_tools=tl_Pen`. При нажатии кнопки мыши назначаем толщину и цвет линии с помощью соответствующих компонентов инструментальной формы и перемещаем указатель мыши в начальную точку `x1,y1` (листинг 7.49).

Листинг 7.49. Начало рисования линии

```
    tl_Pen:
       begin
         drawing:=true;
         Pen.Width:=FormTools.PenWidth.Value;
         Pen.Color:=FormTools.ShPen.Brush.Color;
         MoveTo(x1,y1);
       end;
```

Во время перемещения указателя мыши рисуем линию до точки `x2,y2`:

```
    tl_Pen: LineTo(x2,y2);
```

При отпускании кнопки мыши никаких дополнительных действий, кроме установки значения `drawing:=false`, делать не надо.

Рисование отрезка линии немного сложнее: необходимо запомнить координаты начальной точки отрезка `x0,y0`, запомнить координаты конечной точки отрезка `xt,yt`, установить цвет и толщину линии и, самое главное, установить режим рисования `pmNotXor`. Этот режим при повторном рисовании позволяет восстановить цвета точек фона (листинг 7.50).

Листинг 7.50. Начало рисования отрезка прямой линии

```
    tl_Line:
       begin
```

```
      drawing:=true;
      x0:=x; y0:=y; xt:=x; yt:=y;
      Pen.Width:=FormTools.PenWidth.Value;
      Pen.Color:=FormTools.ShPen.Brush.Color;
      Pen.Mode:=pmNotXor;
    end;
```

Во время перемещения указателя мыши мы должны повторным рисованием `MoveTo-LineTo` "стереть" линию в старом положении, изменить координаты конца отрезка `xt,yt` и нарисовать линию в новом положении (листинг 7.51).

Листинг 7.51. Рисование отрезка прямой линии

```
tl_Line:
   begin
     MoveTo(x0,y0); LineTo(xt,yt);
     xt:=x; yt:=y;
     MoveTo(x0,y0); LineTo(xt,yt);
   end;
```

На заключительном этапе рисования при отпускании кнопки мыши процедура `MouseUp` возвращает режим рисования в стандартный режим `Pen.Mode:=pmCopy` и окончательно рисует линию (листинг 7.52).

Листинг 7.52. Завершение рисования отрезка прямой линии

```
tl_Line:
   begin
     Pen.Mode:=pmCopy;
     MoveTo(x0,y0); LineTo(xt,yt);
   end;
```

Рисование прямоугольников при `fl_tools=tl_Rect`, вставка изображений при `fl_tools=tl_Picture` и перемещение изображений при `fl_tools=tl_Move` начинается одинаково:

1. Присваиваем переменным `x0`, `y0` значения, соответствующие координатам левого верхнего угла прямоугольника.
2. Присваиваем переменным `xt`, `yt` значения, соответствующие координатам правого нижнего угла прямоугольника.
3. Устанавливаем черный цвет линии и назначаем значение толщины линии, равное `1`.
4. Назначаем белый цвет фона, так как во время перемещения мы будем пользоваться примитивом `DrawFocusRect`, а он достаточно капризен к значениям свойств `Pen` и `Brush` (листинг 7.53).

Листинг 7.53. Начало рисования прямоугольника

```
tl_Rect,tl_Picture,tl_Move:
   begin
     drawing:=true;
     x0:=x; y0:=y; xt:=x; yt:=y;
     Pen.Width:=1;
     Pen.Color:=clBlack;
     Brush.Style:=bsSolid;
     Brush.Color:=clWhite;
     DrawFocusRect(Rect(x0,y0,xt,yt));
   end;
```

Во время перемещения указателя мыши также происходят одинаковые события: сначала необходимо стереть старый фокусный прямоугольник и изменить координаты `xt`, `yt`, а затем нарисовать прямоугольник в новом месте (листинг 7.54).

Листинг 7.54. Рисование прямоугольника во время перемещения указателя мыши

```
tl_Rect,tl_Picture,tl_Move:
   begin
     DrawFocusRect(Rect(x0,y0,xt,yt));
     xt:=x; yt:=y;
     DrawFocusRect(Rect(x0,y0,xt,yt));
   end;
```

При отпускании кнопки мыши, при рисовании прямоугольников, при вставке изображения и при перемещении прямоугольника сначала последний раз рисуем `DrawFocusRect`, а затем уже происходят различные события, например, рисование прямоугольника установленными свойствами пера и кисти или вставка прямоугольного изображения. При рисовании прямоугольников устанавливаем свойства карандаша и кисти и рисуем прямоугольник (листинг 7.55).

Листинг 7.55. Окончательное рисование прямоугольника

```
tl_Rect:
   begin
     DrawFocusRect(Rect(x0,y0,xt,yt));
     Pen.Width:=FormTools.PenWidth.Value;
     Pen.Color:=FormTools.ShPen.Brush.Color;
     Brush.Color:=FormTools.ShBrush.Brush.Color;
     Brush.Style:=BrushStyle;
     RectAngle(x0,y0,xt,yt);
   end;
```

Вставка изображения заканчивается выбором изображения с помощью стандартного метода открытия графических файлов `OpenPictureDialog1`, загрузкой изображения в `Bitmap` и копированием `Bitmap` в выбранный прямоугольник (листинг 7.56).

Листинг 7.56. Вставка изображения

```
tl_Picture:
   begin
     DrawFocusRect(Rect(x0,y0,xt,yt));
     With FormMain.OpenPictureDialog1 do
     If Execute then begin
       Bitmap:=TBitmap.Create;
       Bitmap.LoadFromFile(FileName);
       StretchDraw(Rect(x0,y0,xt,yt),Bitmap);
       Bitmap.Free;
     end;
   end;
```

Для перемещения прямоугольника нам придется воспользоваться еще одним компонентом `Im:TImage`: установим компонент `Im` в выбранный прямоугольник `x0,y0,xt,yt`, скопируем в него изображение из выбранного прямоугольника, выбранный прямоугольник зальем белым цветом (листинг 7.57).

Листинг 7.57. Перемещение прямоугольника

```
tl_Move:
   begin
     DrawFocusRect(Rect(x0,y0,xt,yt));
     L:=x0; T:=y0; W:=xt-x0; H:=yt-y0;
     Im.Left:=L; Im.Top:=T;
     Im.Width:=W; Im.Height:=H;
     Im.Canvas.CopyRect(Rect(0,0,W,H),Image1.Canvas,
       Rect(L,T,L+W,T+H));
     Im.Canvas.DrawFocusRect(Rect(0,0,W-1,H-1));
     Im.Visible:=true;
     Brush.Color:=clWhite;
     FillRect(Rect(x0,y0,xt,yt));
   end;
```

Само перемещение определяется в обработчиках события `onMouseDown`, `onMouseMove` и `onMouseUp` компонента `Im`. В начале перемещения поднимаем флаг перемещения `drawing:=true`, запоминаем координаты левого верхнего угла и отклонение точки `x`, `y` от этого угла (листинг 7.58).

Листинг 7.58. Начало перемещения компонента *Im*

```
procedure TFormEd.ImMouseDown(Sender: TObject;
  Button:TMouseButton; Shift:TShiftState; X,Y: Integer);
begin
  drawing:=true; xt:=Im.Left+x; yt:=Im.Top+y;
  dx:=x; dy:=y;
end;
```

Во время перемещения меняем координаты левого верхнего угла компонента `Im` (листинг 7.59).

Листинг 7.59. Перемещение компонента *Im*

```
procedure TFormEd.ImMouseMove(Sender: TObject; Shift:
  TShiftState; X,Y: Integer);
begin
  if drawing then begin
    Im.Left:=xt+x-dx; Im.Top:=yt+y-dy;
    xt:=xt+x-dx; yt:=yt+y-dy;
  end;
end;
```

При завершении перемещения опускаем флаг перемещения `drawing:=false`, копируем с компонента `Im` изображение на канву `Image1` в новом месте и прячем компонент-контейнер `Im` (листинг 7.60).

Листинг 7.60. Завершение перемещения компонента *Im*

```
procedure TFormEd.ImMouseUp(Sender: TObject;
  Button:TMouseButton; Shift:TShiftState; X,Y: Integer);
begin
  drawing:=false;
  Im.Canvas.DrawFocusRect(Rect(0,0,W-1,H-1));
  Image1.Canvas.CopyRect(Rect(xt,yt,xt+W,yt+H),
    Im.Canvas,Rect(0,0,W,H));
  Im.Visible:=false;
end;
```

Рисование других примитивов типа `Ellipse`, `RounRect` реализуется так же просто. Полный текст проекта приведен на компакт-диске в папке **Примеры | Глава 7 | Paint**.

7.7. Проектирование плоских схем

К задачам проектирования плоских схем относятся задачи построения сложных диаграмм и электрических схем, структурных схем соединения компьютеров в сети и структурных схем программ, проектирования реляционных баз данных, задачи макетирования карт, а также множество других проектов, в которых требуется установить связи между объектами, изображенными на экране в виде условных обозначений.

Существует большое количество программ, предназначенных для проектирования таких плоских схем. В качестве примера можно назвать пакет PCAD, позволяющий проектировать и исследовать различные электрические схемы, пакет Visio Pro,

предназначенный для проектирования любых плоских схем, и ErWin, назначение которого — проектирование реляционных баз данных. Программы, предназначенные для проектирования плоских схем, должны, прежде всего, выполнять следующие функции:

- вставлять в проект условные обозначения объектов;
- устанавливать связи между объектами;
- перемещать объекты и связи;
- удалять объекты и связи;
- менять свойства объектов.

В проекте "Проектирование плоских схем", рассматриваемом в данном разделе, продемонстрирован один из вариантов решения такой задачи.

7.7.1. Структура данных

Проект состоит из трех форм **FormMain**, **FormIco** и **FormTools** (рис. 7.19). На форме **FormIco** размещен набор изображений (объектов) из файлов BMP размера 32×32 пиксела, на главную форму **FormMain** выводится плоская схема, состоящая из этих объектов и бинарных связей между ними. Форма **FormTools** представляет собой инструментальную панель с набором кнопок, которые будут подробно описаны в *разд. 7.7.2*.

Список имен файлов с изображениями объектов, для упрощения проекта, хранится в текстовом файле BMP.TXT, фрагмент которого приведен далее (листинг 7.61).

Листинг 7.61. Файл с именами изображений объектов

```
CALC.BMP Object 1
CALCOLD.BMP Object 2
CALENDAR.BMP Object 3
CHECKMRK.BMP Object 4
CLDTABL.BMP Object 5
CLIP01.BMP Object 6
```

В каждой строке этого файла приведено имя файла и через пробел имя объекта. Для хранения этой информации во время выполнения проекта введен тип `TObmp`:

```
type TObmp=record
       namefile: string[12];
       name: string[12];
     end;
```

и массив `Obmp: array[0.. cBmp] of TObmp`, число элементов которого определяется во время чтения файла `BMP.txt` при активизации проекта (листинг 7.62) и хранится в переменной `Num_Ico`.

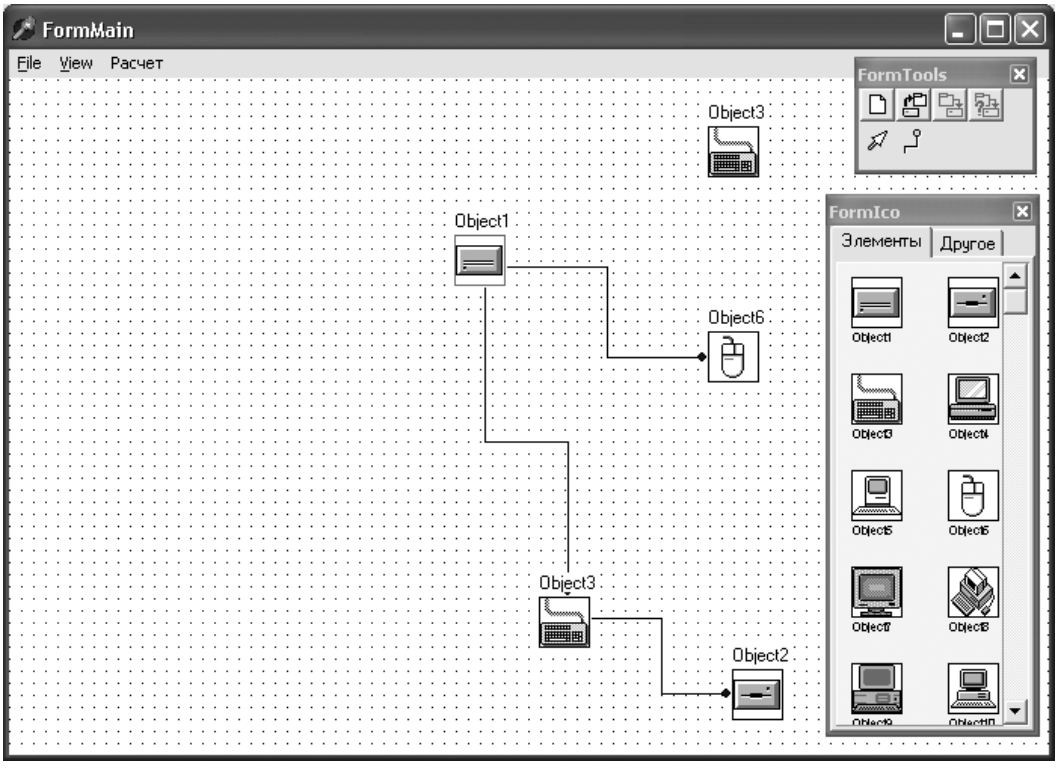


Рис. 7.19. Формы проекта
"Проектирование плоских схем"

Листинг 7.62. Процедура активизации проекта

```
procedure TFormMain.FormCreate(Sender: TObject);
var q: word;
begin
  Bitmap:= TBitmap.Create;
  Num_Ico:=-1;
  AssignFile(ft,'t.txt'); Reset(ft);
  while not EOF(ft) do begin
    Readln(ft,s); Inc(Num_Ico); SetLength(oBmp,Num_Ico+1);
    q:=Pos(' ',s);
    Obmp[Num_Ico].NameFile:=Copy(s,1,q-1);
    Obmp[Num_Ico].Name:=Copy(s,q+1,Length(s)-q);
  end;
  CloseFile(ft);
end;
```

Каждый эскиз хранится в нетипизированном файле.

Структура данных проекта очень проста. Узлы описываются структурой `Tnode`.

```
TNode=record
  u,v: integer;
  NameFile: string[12];
  Name: string[12];
end;
```

Эта структура содержит следующие поля: `Name` — имя объекта; `u` и `v` — две экранных координаты. Сами данные хранятся в массиве `Nodes: array of TNode`, число элементов которого определяется переменной `NumNode`.

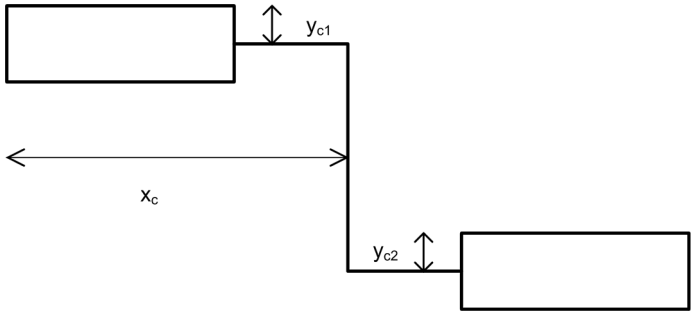


Рис. 7.20. Вертикальная бинарная связь между объектами

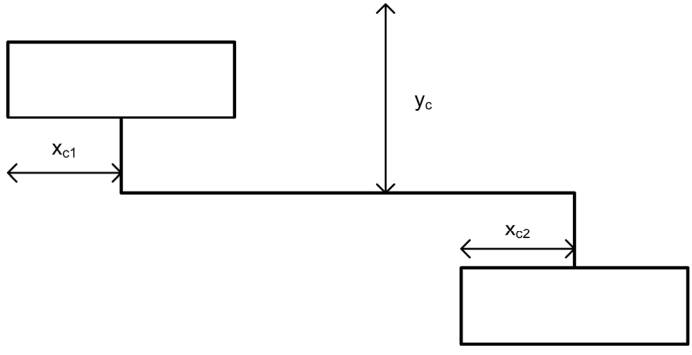


Рис. 7.21. Горизонтальная бинарная связь между объектами

Формат типизированного файла бинарных связей чуть сложнее:

```
TEdge=record
  xc,y1c,y2c: integer;
  n1,n2: byte;
  fl_Edge: byte;
end;
```

Поля `n1` и `n2` определяют номера объектов, участвующих в бинарном отношении, а остальные поля содержат информацию о геометрии этой связи. Бинарное отношение изображается в виде трехзвенной ломаной линии (рис. 7.20 и 7.21).

У этой ломаной линии средняя часть может быть вертикальной (`fl_Edge=1`) или горизонтальной (`fl_Edge=2`). В первом случае x_c — это абсцисса вертикальной средней части, а y_{c1} и y_{c2} — отклонения по вертикали первого отрезка ломаной от левого верхнего угла первого объекта и третьего отрезка ломаной от левого верхнего угла второго объекта. При `fl_Edge=2` значение поля y_c — это ордината горизонтальной средней части, а x_{c1} и x_{c2} — отклонения по горизонтали первого отрезка ломаной от левого верхнего угла первого объекта и третьего отрезка ломаной от левого верхнего угла второго объекта.

Сами данные о бинарных связях хранятся в массиве `Edges: array of TEdge`, число элементов которого содержится в переменной `Num_Edge`.

7.7.2. Структура проекта

На инструментальную форму **`FormTools`** вынесено 6 кнопок:

- `NewButton` — создать новый эскиз;
- `OpenButton` — открыть эскиз;
- `SaveButton` — сохранить эскиз;
- `SaveAsButton` — сохранить эскиз под новым именем;
- `MoveButton` — кнопка режима перемещения объектов и бинарных отношений;
- `AddButton` — кнопка режима добавления бинарных отношений.

Две последние кнопки образуют группу: нажатие кнопки `MoveButton` меняет значение параметра `fl_tools` на `0`, а нажатие кнопки `AddButton` меняет значение параметра `fl_tools` на `1`.

Кнопка `NewButton` предназначена для создания нового эскиза и обладает единственным обработчиком события `NewButtonClick` (листинг 7.63).

Листинг 7.63. Создание нового эскиза

```
procedure TFormMain.New1Click(Sender: TObject);
begin
  FormTools.SaveButton.Enabled:=False;
  Num_Node:=0; Num_Edge:=0;
  SetLength(Nodes,Num_Node); SetLength(Edges,Num_Edge);
  ShowNode;
end;
```

В рассмотренной процедуре обнуляется число объектов `Num_Node:=0` и число бинарных отношений `Num_Edge:=0`, а затем вызывается процедура рисования эскиза `ShowNode` (листинг 7.64).

Листинг 7.64. Процедура рисования эскиза

```
procedure TFormMain.ShowNode;
var i,j: integer;
begin
  with Canvas do begin
    Brush.Color:=clWhite;
    RectAngle(-1,-1,Width,Height);

    if N4.Checked then  // рисование сетки
    for i:=0 to Width div 8 do
    for j:=0 to Height div 8 do
      Pixels[i*8,j*8]:=0;

    for i:=0 to Num_Edge-1 do  // рисование линий связи
    with Edges[i] do
      Line_2(i,fl_Edge,
             Nodes[n1].u,Nodes[n1].v,
             Nodes[n2].u,Nodes[n2].v,
             xc,y1c,y2c);

    Brush.Color:=clWhite; // рисование объектов
    for i:=0 to Num_Node-1 do
    with Nodes[i] do begin
      Bitmap.LoadFromFile(NameFile);
      if i=NumSel then Pen.Color:=clRed else Pen.Color:=clBlack;
      RectAngle(u-1-16,v-1-16,u-16+33,v-16+33);
      TextOut(u-1-16,v-1-16-16,Name);
      Draw(u-16,v-16,Bitmap);
    end;
  end;
end;
```

В процедуре `ShowNode` после очистки окна двойным циклом рисуется сетка из точек. Затем процедурой `Line_2` (листинг 7.65) рисуются ломаные линии бинарных отношений и, наконец, после загрузки в `Bitmap` файлов с именами `Nodes[i].Name`, выводятся изображения объектов.

Процедура `Line_2`, в зависимости от значения параметра `fl`, который может принимать значение `1` или `2`, рисует трехзвенную ломаную линию с вертикальным или горизонтальным средним звеном.

Листинг 7.65. Процедура рисования линий бинарных отношений

```
procedure TFormMain.Line_2(k: integer; fl: byte;
   x1,y1,x2,y2,xc,yc1,yc2: integer);
const
  c4=3;
var
  t: integer;
begin
  with Canvas do
  case fl of
    1:  begin // xc,yc1,yc2
          Pen.Color:=clBlack;
          if x1<xc
            then MoveTo(x1+hx div 2+2,y1+yc1)
            else MoveTo(x1-hx div 2-2,y1+yc1);
          LineTo(xc,y1+yc1);
          LineTo(xc,y2+yc2);
          if xc<x2
            then t:=x2-hx div 2-5
            else t:=x2+hx div 2+5;
          LineTo(t,y2+yc2);
          Brush.Color:=clNavy;
          Ellipse(t-c4,y2+yc2-c4,t+c4,y2+yc2+c4);
        end;
    2:  begin // yc,xc1,xc2
          Pen.Color:=clBlack;
          if y1<xc
            then MoveTo(x1+yc1,y1+hy div 2+2)
            else MoveTo(x1+yc1,y1-hy div 2-2);
          LineTo(x1+yc1,xc);
          LineTo(x2+yc2,xc);
          if xc<y2
            then t:=y2-hy div 2-5
            else t:=y2+hy div 2+5;
          LineTo(x2+yc2,t);
          Brush.Color:=clNavy;
          Ellipse(x2+yc2-c4,t-c4,x2+yc2+c4,t+c4);
        end;
  end;
end;
```

Нажатие кнопки `OpenButton` вызывает стандартный метод `OpenDialog2` (листинг 7.66), после успешного завершения которого читается нетипизированный файл с расширением `GRP` и заполняется массив объектов `Nodes[0..NumNode-1]` и массив бинарных отношений `Edges[0..Num_Edge-1]`.

Листинг 7.66. Процедура чтения существующего эскиза

```
procedure TFormMain.Open1Click(Sender: TObject);
var i: integer;
begin
  if OpenDialog2.Execute then begin
    FileName:=OpenDialog2.FileName;
    AssignFile(f,OpenDialog2.FileName); Reset(f,1);
    BlockRead(f,Num_Node,SizeOf(Num_Node));
    SetLength(Nodes,Num_Node);
    for i:=0 to Num_Node-1 do
      BlockRead(f,Nodes[i],SizeOf(Nodes[i]));
    BlockRead(f,Num_Edge,SizeOf(Num_Edge));
    SetLength(Edges,Num_Edge);
    for i:=0 to Num_Edge-1 do
      BlockRead(f,Edges[i],SizeOf(Edges[i]));
    CloseFile(f);
    ShowNode;
  end;
end;
```

Нажатие кнопки `SaveButton` приводит к вызову процедуры записи эскиза (листинг 7.67).

Листинг 7.67. Процедура записи эскиза

```
procedure TFormMain.SaveGrp;
var i: integer;
begin
  if FileName<>'' then begin
    AssignFile(f,FileName); Rewrite(f,1);
    BlockWrite(f,Num_Node,SizeOf(Num_Node));
    for i:=0 to Num_Node-1 do
      BlockWrite(f,Nodes[i],SizeOf(Nodes[i]));
    BlockWrite(f,Num_Edge,SizeOf(Num_Edge));
    for i:=0 to Num_Edge-1 do
      BlockWrite(f,Edges[i],SizeOf(Edges[i]));
    CloseFile(f);
  end;
end;
```

Нажатие кнопки `SaveAsButton` вызывает стандартный метод `SaveDialog2`, после удачного завершения которого процедура `SaveGrp` записывает эскиз под новым именем (листинг 7.68).

Листинг 7.68. Процедура записи эскиза под новым именем

```
procedure TFormMain.Save1Click(Sender: TObject);
var i: integer;
begin
  if SaveDialog2.Execute then begin
    FileName:=SaveDialog2.FileName;
    SaveGrp;
  end;
end;
```

7.7.3. Добавление нового объекта в эскиз

На форме **FormIco** справа на компоненте `Image1` методом `ShowIco` выводятся изображения объектов. Предполагается, что все изображения объектов представляют собой рисунки размером 32×32 пиксела. На форме в два столбца выводятся только 14 изображений объектов (листинг 7.69).

Листинг 7.69. Процедура вывода библиотеки объектов

```
procedure TFormIco.ShowIco;
var i: integer;
begin
  with Image1.Canvas do begin
    Font.Name:='Sans Serif';
    Font.Size:=6;
    Brush.Color:=$00C4F7FF;
    RectAngle(-1,-1,Width,Height);
    Brush.Color:=clWhite;
    for i:=fl_ofs to c7+fl_ofs do begin
      Bitmap.LoadFromFile(Obmp[i].namefile);
      Brush.Color:=clWhite;
      RectAngle(10+((i-fl_ofs) mod 2)*ch-1,
           10+((i-fl_ofs) div 2)*ch-1,
           10+((i-fl_ofs) mod 2)*ch+33,
           10+((i-fl_ofs) div 2)*ch+33);
      Draw(10+((i-fl_ofs) mod 2)*ch,
           10+((i-fl_ofs) div 2)*ch,Bitmap);
      Brush.Color:=$00C4F7FF;
      TextOut(10+((i-fl_ofs) mod 2)*ch,
           10+((i-fl_ofs) div 2)*ch+34,Obmp[i].name);
    end;
  end;
end;
```

Параметр `fl_ofs` отвечает за прокрутку и меняется при изменении положения ползунка компонента `ScrollBar1` (листинг 7.70).

Листинг 7.70. Процедура изменения параметра смещения *fl_ofs*

```
procedure TFormIco.ScrollBar1Change(Sender: TObject);
var t,h: byte;
begin
  h:=100 div (Num_Ico-c7);
  t:=ScrollBar1.Position div h;
  if Abs(t-fl_ofs)>1 then begin
    fl_ofs:=t; ShowIco;
  end;
end;
```

Новый объект можно перетащить в эскиз при нажатой кнопке `MoveButton`. Для того чтобы перетаскивание могло начаться у компонента `Image1`, свойство `DragMode` должно иметь значение `dmAutomatic`, а начало перетаскивания должно вызывать обработчик события `Image1DragOver` (листинг 7.71).

Листинг 7.71. Обработчик события начала перетаскивания

```
procedure TFormIco.Image1DragOver(Sender, Source: TObject;
  X, Y: Integer; State: TDragState; var Accept: Boolean);
begin
  if State=dsDragEnter then
    NumDrag:=2*(y div ch) + (x div ch)+fl_ofs;
end;
```

Параметр `State: TDragState = (dsDragEnter, dsDragLeave, dsDragMove)` в методе `Image1DragOver` отвечает за стадии перетаскивания над компонентом источника (в нашем случае над компонентом `Image1`). Он может принимать одно из следующих значений:

- `DsDragEnter` — указатель мыши начал перемещение над компонентом;
- `dsDragMove` — указатель мыши перемещается над компонентом;
- `DsDragLeave` — указатель мыши закончил перемещение над компонентом.

Номер перетаскиваемого объекта `NumDrag` до начала перетаскивания должен быть равен `-1`. Затем этот номер вычисляется и при перемещении указателя мыши над компонентом `Image1` уже не меняется.

Перетаскивание завершается над формой `FormMain` и должно привести к увеличению числа объектов в эскизе (листинг 7.72).

Листинг 7.72. Обработчик события завершения перетаскивания

```
procedure TFormMain.FormDragDrop(Sender, Source: TObject; X, Y: Integer);
begin
```

```
  x:=8*Round(x/8); y:=8*Round(y/8);
  // добавление бинарного отношения
  Inc(Num_Node); SetLength(Nodes,Num_Node);
  Nodes[Num_Node-1].u:=x; Nodes[Num_Node-1].v:=y;
  Nodes[Num_Node-1].NameFile:=Obmp[NumDrag].namefile;
  Nodes[Num_Node-1].Name:=Obmp[NumDrag].name;
  NumDrag:=-1;
  // активизация кнопки записи
  FormTools.SaveButton.Enabled:=True;
  ShowNode;
end;
```

7.7.4. Перемещение объектов и линий связи на эскизе

Перемещение объектов и линий возможно при нажатой кнопке `MoveButton`. В момент нажатия кнопки мыши вызывается обработчик события (листинг 7.73).

Листинг 7.73. Обработчик события начала перемещения объекта

```
procedure TFormMain.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  // определение номера объекта
  Drawing:=false; NumSel:=-1; fl_Node:=0;
  while (NumSel<Num_Node-1) and not Drawing do begin
    Inc(NumSel);
    with Nodes[NumSel] do
    Drawing:=(u-16<=x) and (x<=u+16) and
        (v-16<=y) and (y<=v+16);
  end;
  if Drawing then begin
    with Canvas do
    case fl_tools of
      0: begin // начало перемещения объекта
           xt:=Nodes[NumSel].u;
           yt:=Nodes[NumSel].v;
           DrawFocusRect(Rect(xt-16,yt-16,xt+16,yt+16));
         end;
      1: begin // начало добавления линии
           Pen.Mode:=pmNotXOR;
           xt:=x; yt:=y;
           xk:=x; yk:=y;
           Line2(Nodes[NumSel].u,Nodes[NumSel].v,xt,yt);
```

```
        end;
    end;
  end
  else begin // попытка найти близкую линию
    Drawing:=false; NumSel:=-1;
    while (NumSel<Num_Edge-1) and not Drawing do begin
      Inc(NumSel);
      Drawing:=SetFlagNode(x,y,NumSel);
    end;

    if Drawing then // начало перемещения линии
    case fl_tools of
      0:  with Canvas do begin
            Pen.Mode:=pmNotXor;
            Pen.Color:=clBlack;
          end;
    end;
  end;
  PopupMenu1.AutoPopup:=Drawing;

  Drawing:=Drawing and (Button=mbLeft);
  if not Drawing then
    Canvas.Pen.Mode:=pmCopy;
end;
```

Сначала в цикле `while (NumSel<NumNode) and not Ok do` делается попытка определить номер объекта, над которым произошло событие нажатия кнопки мыши. Если событие произошло над объектом и нажата кнопка `MoveButton`, то начинается перемещение объекта:

```
xt:=Nodes[NumSel].u;
yt:=Nodes[NumSel].v;
DrawFocusRect(Rect(xt-16,yt-16,xt+16,yt+16));
```

Если событие произошло над объектом и нажата кнопка `AddButton`, то начинается добавление линии бинарного отношения:

```
Pen.Mode:=pmNotXOR;
xt:=x; yt:=y; //текущая точка мыши
xk:=x; yk:=y; // начальная точка линии
Line2(Nodes[NumSel].u,Nodes[NumSel].v,xt,yt);
```

Если событие произошло не над объектом, то в цикле `while (NumSel<NumEdge) and not Ok do` делается попытка определить номер ближайшей линии. При этом используется функция определения близости линии `SetFlagUz`, в которой для горизонтальной или вертикальной ломаной линии проверяется условие близости (линия близка, если один из ее сегментов находится от точки нажатия мыши на расстоянии не более `del`) (листинг 7.74).

Листинг 7.74. Функция определения близости линии

```
function TFormMain.SetFlagNode(x,y: integer; n: byte): boolean;
var x1,y1,x2,y2: integer;
begin
  fl_Node:=0;
  with Edges[n] do begin
    x1:=Nodes[n1].u; y1:=Nodes[n1].v;
    x2:=Nodes[n2].u; y2:=Nodes[n2].v;
    case fl_Edge of
      1: if (Abs(y1+y1c-y)<=del) and InInt(x1,xc,x)
         then fl_Node:=1
         else
           if (Abs(xc-x)<=del) and InInt(y1+y1c,y2+y2c,y)
           then fl_Node:=2
           else
             if (Abs(y2+y2c-y)<=del) and InInt(x2,xc,x)
             then fl_Node:=3;
      2: if (Abs(x1+y1c-x)<=del) and InInt(y1,xc,y)
         then fl_Node:=1
         else
           if (Abs(xc-y)<=del) and InInt(x1+y1c,x2+y2c,x)
           then fl_Node:=2
           else
             if (Abs(x2+y2c-x)<=del) and InInt(y2,xc,y)
             then fl_Node:=3;
    end;
  end;
  Result:=fl_Node<>0;
end;
```

Параметр `fl_Node` принимает значение `1`, `2` или `3` в зависимости от того, какой из трех сегментов ломаной линии находится вблизи от точки нажатия мыши. Функция `InInt(x1,x2,x)` определяет принадлежность точки `x` интервалу `[x1,x2]`.

Если найдена близкая ломаная линия связи, то параметр `Ok` принимает значение `true` и меняется режим пера `Pen.Mode:=pmNotXor`.

Во время перемещения указателя мыши работает обработчик события `onMouseMove` (листинг 7.75).

Листинг 7.75. Обработчик события перемещения указателя мыши

```
procedure TFormMain.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X,Y: Integer);
begin
  if Drawing then
```

```
with Canvas do
case fl_tools of
  0:  if fl_Node=0 then begin // перемещение объекта
        DrawFocusRect(Rect(xt-16,yt-16,xt+16,yt+16));
        xt:=x; yt:=y;
        DrawFocusRect(Rect(xt-16,yt-16,xt+16,yt+16));
      end
      else // перемещение линии связи обектов
      with Edges[NumSel] do begin
        Line_2(NumSel,fl_Edge,
               Nodes[n1].u+x0,Nodes[n1].v+y0,
               Nodes[n2].u+x0,Nodes[n2].v+y0,
               xc,y1c,y2c);
        if fl_Edge=1 then // вертикальная линия
        case fl_Node of
          1: if InInt(Nodes[n1].v-hy div 2,
                      Nodes[n1].v+hy div 2,y-y0)
             then y1c:=y-y0-Nodes[n1].v
             else begin
               xc:=y-y0; fl_Node:=2;
               y1c:=0; y2c:=0;
               fl_Edge:=2;
             end;
          2: xc:=x-x0;
          3: if InInt(Nodes[n2].v-hy div 2,
                      Nodes[n2].v+hy div 2,y-y0)
             then y2c:=y-y0-Nodes[n2].v
             else begin
               xc:=y-y0; fl_Node:=2;
               y1c:=0; y2c:=0;
               fl_Edge:=2;
             end;
        end
        else // горизонтальная линия
        case fl_Node of
          1: if InInt(Nodes[n1].u-hx div 2,
                      Nodes[n1].u+hx div 2,x-x0)
             then y1c:=x-x0-Nodes[n1].u
             else begin
               xc:=x-x0; fl_Node:=2;
               y1c:=0; y2c:=0;
               fl_Edge:=1;
             end;
          2: xc:=y-y0;
          3: if InInt(Nodes[n2].u-hx div 2,
```

```
                    Nodes[n2].u+hx div 2,x-x0)
              then y2c:=x-x0-Nodes[n2].u
              else begin
                xc:=x-x0; fl_Node:=2;
                y1c:=0; y2c:=0;
                fl_Edge:=1;
              end;
          end;
          Line_2(NumSel,fl_Edge,
                 Nodes[n1].u+x0,Nodes[n1].v+y0,Nodes[n2].u+x0,
                 Nodes[n2].v+y0,
                 xc,y1c,y2c);
        end;
    1: begin // добавление линии
         Line2(Nodes[NumSel].u,Nodes[NumSel].v,xt,yt);
         xt:=x; yt:=y;
         Line2(Nodes[NumSel].u,Nodes[NumSel].v,xt,yt);
       end;
  end;
end;
```

В этой процедуре в зависимости от значения параметра `fl_tools` (нажата кнопка `MoveButton` или `AddButton`) перемещаются существующие элементы (объекты или ломаные линии) или методом `Line2` строится новая ломаная линия. При перемещении существующих элементов учитывается тип ломаной линии `Edges[NumSel].fl_net` и значение параметра `Fl_Node`. При `Fl_Node=0` перемещается объект с номером `NumSel`, а при `Fl_Node` равном `1`, `2` или `3` — одно из звеньев ломаной линии.

Завершается перемещение элементов вызовом процедуры `onMouseUp` (листинг 7.76).

Листинг 7.76. Завершение перемещения указателя мыши

```
procedure TFormMain.FormMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var i: integer;
begin
  if Drawing then
  with Canvas do
  begin
    case fl_tools of
      0: if fl_Node=0 then begin //завершение перемещения объекта
           DrawFocusRect(Rect(xt-16,yt-16,xt+16,yt+16));
           Nodes[NumSel].u:=8*Round(x/8);
           Nodes[NumSel].v:=8*Round(y/8);
         end
```

```
        else begin // завершение перемещения линии
          with Edges[NumSel] do
          Line_2(NumSel,fl_Edge,
                 Nodes[n1].u+x0,Nodes[n1].v+y0,
                 Nodes[n2].u+x0,Nodes[n2].v+y0,
                 xc,y1c,y2c);
          Canvas.Pen.Mode:=pmCopy;
        end;
      1: begin // завершение добавления линии
          Line2(Nodes[NumSel].u,Nodes[NumSel].v,xt,yt);
          Canvas.Pen.Mode:=pmCopy;
          // проверка принадлежности точки объекту
          Drawing:=false; i:=-1;
          while (i<Num_Node) and not Drawing do begin
            Inc(i);
            with Nodes[i] do
            Drawing:=(u-16<=x) and (x<=u+16) and
                (v-16<=y) and (y<=v+16);
          end;
          if Drawing then begin // если принадлежит, то добавляем
            Inc(Num_Edge); SetLength(Edges,Num_Edge);
            with Edges[Num_Edge-1] do begin
              n1:=NumSel; n2:=i;
              if Abs(Nodes[i].u-Nodes[NumSel].u) >
                 Abs(Nodes[i].v-Nodes[NumSel].v)
              then begin
                fl_Edge:=1;
                xc:=(Nodes[i].u+Nodes[NumSel].u) div 2;
                y1c:=-Nodes[NumSel].v+yk;
                y2c:=-Nodes[i].v+y;
              end
              else begin
                fl_Edge:=2;
                xc:=(Nodes[i].v+Nodes[NumSel].v) div 2;
                y1c:=-Nodes[NumSel].u+xk;
                y2c:=-Nodes[i].u+x;
              end;
            end;
          end;
        end;
    end;
    FormTools.SaveButton.Enabled:=True;
    Drawing:=false;
    ShowNode;
  end;
end;
```

Естественно, завершающие действия также зависят от значения параметра `fl_tools` (нажата кнопка `MoveButton` или `AddButton`) и от значения параметра `Fl_Node`. При добавлении новой линии связи циклом `while (i<NumNode) and not Ok do` определяется номер объекта, на котором закончено перемещение указателя мыши, и если перемещение действительно закончено на объекте, то добавляется новая ломаная линия `Edges[NumEdge]`.

7.7.5. Удаление объектов и линий связи на эскизе

Пункт **Удаление** включен в контекстное меню `PopupMenu1`, привязанное к форме **`FormMain`** свойством `PopupMenu=PopupMenu1`. Для того чтобы это меню появлялось только на элементах схемы, в процедуру `onMouseDown` необходимо включить оператор `PopupMenu1.AutoPopup := Ok`. Параметр `Ok` принимает значение `true`, если щелчок правой кнопкой мыши произошел на объекте или вблизи линии. Тогда при нажатии правой кнопки мыши над элементом схемы появится контекстное меню и можно будет вызвать процедуру `Delete1Click` (листинг 7.77).

Листинг 7.77. Удаление элементов схемы

```
procedure TFormMain.Delete1Click(Sender: TObject);
var i: integer;
    Done: boolean;
begin
  if Application.MessageBox('Удаляем?','',MB_OKCANCEL) = IDOK then begin
    FormTools.SaveButton.Enabled:=True;
    if fl_Node=0 then begin
     Done:=false; i:=-1;
     while (i<Num_Edge-1) and not done do begin
       Inc(i); Done:=(Edges[i].n1=NumSel) or (Edges[i].n2=NumSel);
     end;
     if not Done then begin
       for i:=NumSel+1 to Num_Node-1 do
         Nodes[i-1]:=Nodes[i];
       Num_Node:=Num_Node-1;
       for i:=0 to Num_Edge-1 do begin
         if Edges[i].n1>NumSel then Edges[i].n1:=Edges[i].n1-1;
         if Edges[i].n2>NumSel then Edges[i].n2:=Edges[i].n2-1;
       end;
     end
     else ShowMessage('Есть линии. Удалять нельзя!');
   end
   else begin
     for i:=NumSel+1 to Num_Edge-1 do Edges[i-1]:=Edges[i];
     Num_Edge:=Num_Edge-1;
```

```
      SetLength(Edges,Num_Edge);
    end;
    ShowNode;
  end;
end;
```

Удаление линий связи при `Fl_Node<>0` не вызывает проблем: надо сдвинуть элементы массива `Edges[i]` и уменьшить число линий связи `NumEdge:=NumEdge-1`. Немного сложнее удалять объекты при `Fl_Node=0`, так как сначала надо убедиться, что у этого объекта нет связей. Если связей действительно нет, то сдвигаем элементы массива `Nodes[i]`, уменьшаем число объектов `NumNode:= NumNode-1`, а затем изменяем номера объектов у бинарных связей.

Мы не ставили перед собой задачу создать законченный продукт для проектирования, например, локальных вычислительных сетей. Понятно, что эта конкретная задача должна включать в себя методы аналитических расчетов и имитационного моделирования, а это выходит далеко за рамки данной книги. Полностью проект приведен на компакт-диске в папке **Примеры | Глава 7 | LVS**.

7.8. Редактирование графа

Понятие графов вводится во многих книгах, в частности, в книге [109] приведен достаточно обширный список литературы, посвященной этому вопросу. В этом разделе рассматривается только часть проблем, связанных с изображением графа, добавление узлов и связей между узлами. Предлагаемый проект "Редактирование графа" состоит из главной формы `FormGraph`, вспомогательных форм `FormPropertyGraph`, `FormListBox`, `FormMatr`, `FormTools`, `FormViewGraph` и модуля `LibGraph`, в котором собраны структуры данных и все реализованные алгоритмы для графов (рис. 7.22).

Рисование графа реализовано на канве формы `FormGraph` и позволяет изображать граф в трех видах: в виде прямоугольников, соединенных ломаными линиями (рис. 7.23); в виде окружностей, соединенных прямыми линиями (рис. 7.24); в виде элементов "строка заголовка" (рис. 7.25).

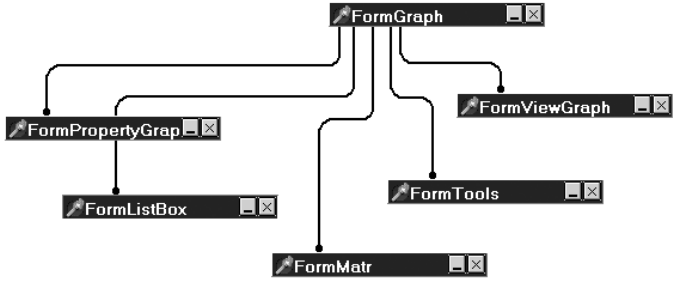


Рис. 7.22. Структура проекта "Редактирование графа"

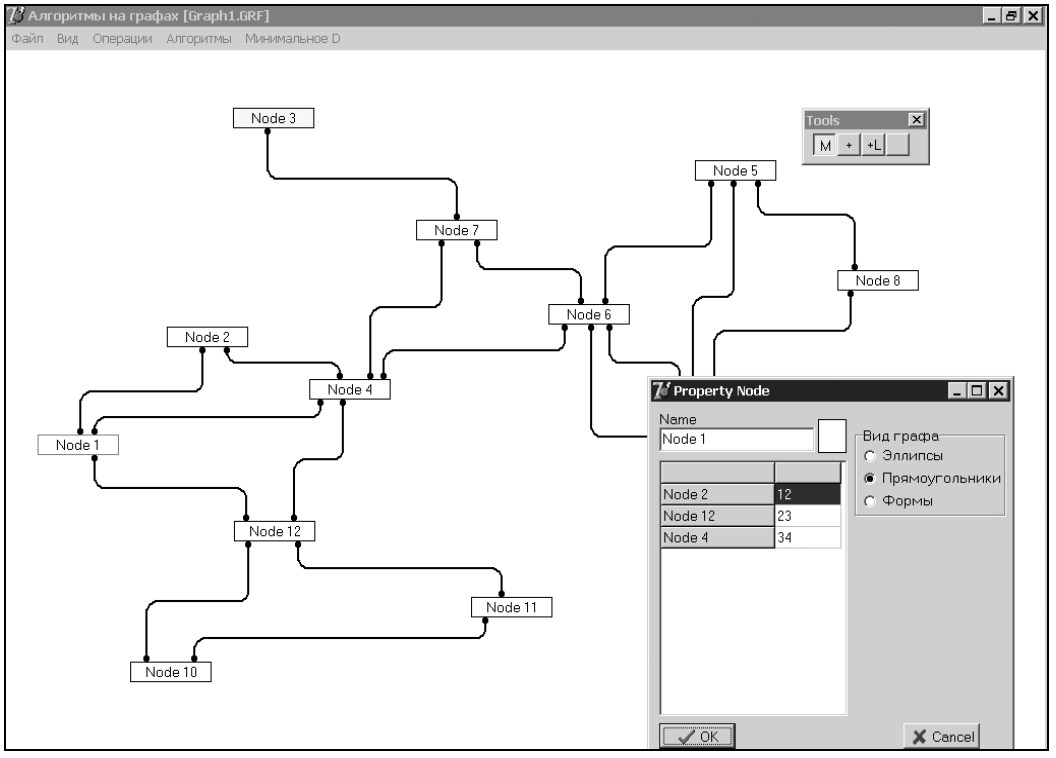


Рис. 7.23. Проект "Редактирование графа", узлы — прямоугольники

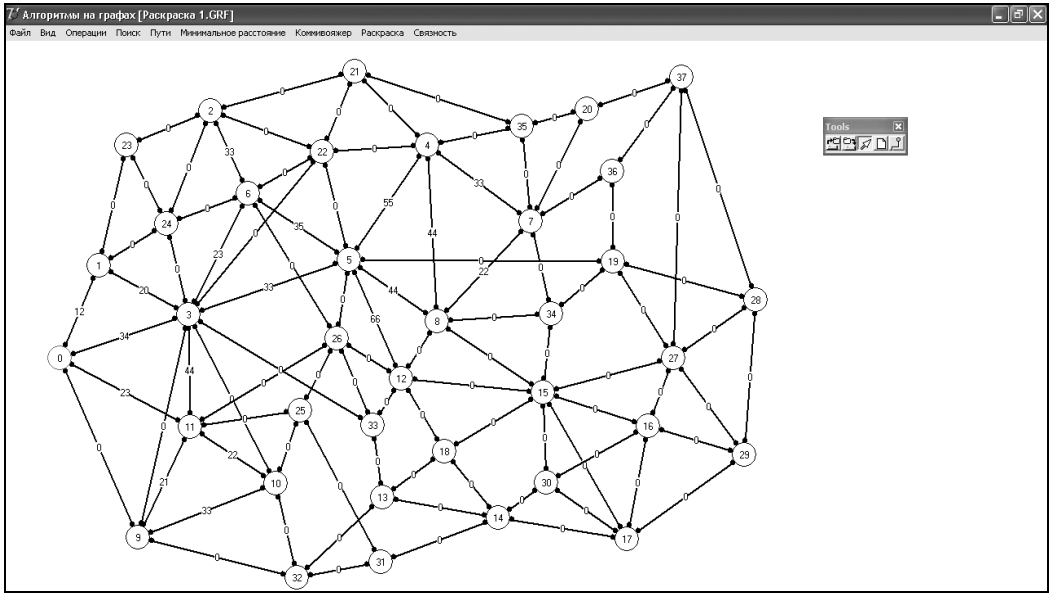


Рис. 7.24. Проект "Редактирование графа", узлы — окружности

Переключение видов графа происходит на форме `FormViewGraph`. На форме `FormTools` находятся три инструментальные кнопки, позволяющие добавлять и перемещать узлы, создавать дуги. Форма `FormMatr` показывает матрицу смежности, а с помощью формы `FormPropertyNode` можно задавать свойства любого узла.

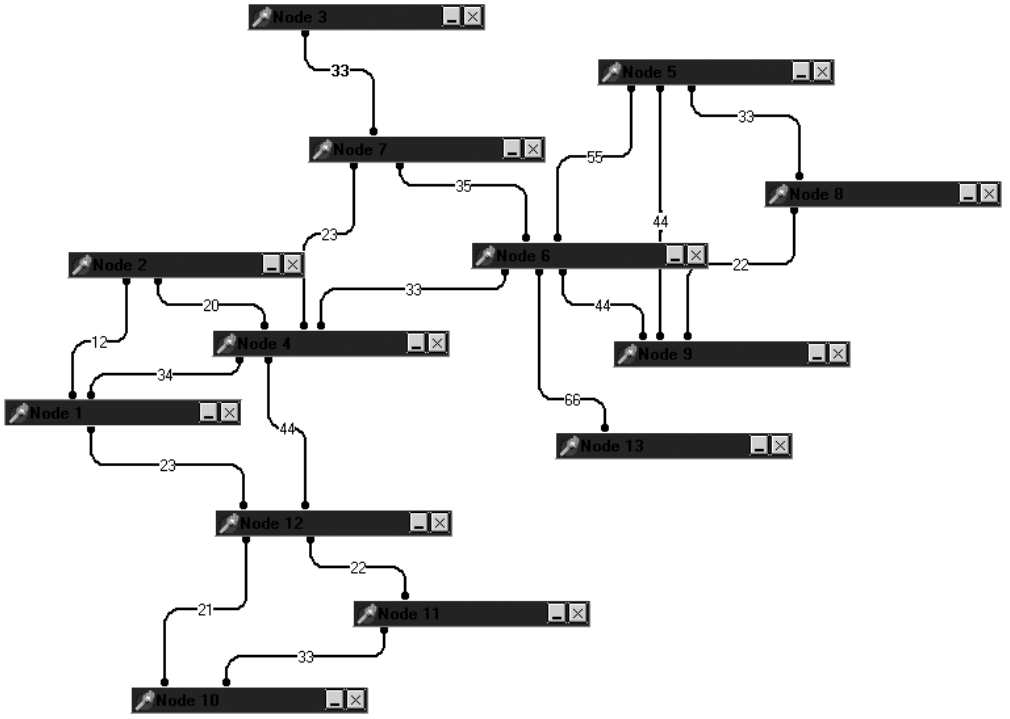


Рис. 7.25. Проект "Редактирование графа", узлы — элементы "строка заголовка"

7.8.1. Структура данных

Выбор структуры данных оказывает решающее влияние на эффективность алгоритмов. Будем использовать для описания узлов графа динамический массив `Node: array of TNode`, каждый элемент которого определяется структурой, представленной в листинге 7.78.

Листинг 7.78. Структура узлов графа

```
TNode = record
  Name    : string;          // имя узла
  Edge    : array of TEdge;  // массив дуг (ребер)
  Visit   : Boolean;         // признак "узел посещен"
```

```
  x0, y0  : Integer;          // координаты центра узла
  NumVisit: Integer;          // № посещения
  Color   : TColor;           // цвет узла
  Dist    : Integer;  // минимальное расстояние до этого узла
end;
```

В этой структуре поле `Name` предназначено для хранения имени узла, поле `Edge` описывает список ребер, выходящих из вершины, поля `x0` и `y0` задают координаты центра вершины. Поле `Visit` будет играть важную роль при реализации многих алгоритмов, в нем мы будет отмечать посещение вершины. Поле `Color` также играет вспомогательную роль при решении задач раскраски графов, а поле `Dist` будет использоваться при решении задач определения кратчайших путей на графе.

В листинге 7.79 представлена структура, предназначенная для описания ребер. Самым важным в этой структуре является поле `NumNode`, содержащее номер вершины, на которую показывает ребро. Не менее важную информацию содержит поле `A`, содержащее "вес ребра", т. е связанное с этим ребром значение (например, длину пути для графа, представляющего дорожную сеть). Для веса ребра мы ограничились целым типом, хотя в реальных задачах это поле может быть вещественным.

Листинг 7.79. Структура ребер

```
TEdge = record
  A           : Integer;  // вес ребра
  NumNode     : Integer;  // номер узла
  x1c, x2c, yc: Integer;  // относительные координаты узла
  Color       : TColor;   // цвет дуги
end;
```

Поле `Color` (цвет дуги) играет вспомогательную роль: при реализации некоторых алгоритмов, связанных с поиском путей между узлами, мы будем менять цвет ребра при включении его в искомый путь. Поля x_{c1}, x_{c2} и y_c содержат геометрические параметры ребра, смысл которых показан на рис. 7.21. Поле y_c указывает на расстояние горизонтальной части ребра от верхнего края формы.

7.8.2. Изображение графов

При рисовании на канве `Bitmap` мы сначала выводим дуги методом `Line_2()`, а затем все узлы (листинг 7.80). Как и раньше, для перетаскивания узлов и создания новых дуг реализованы обработчики событий `onMouseDown`, `onMouseMove` и `onMouseUp`.

Листинг 7.80. Рисование графа

```
procedure TFormGraph.ShowGraph;
var i, j: Integer;
begin
  Bitmap.Width := Width;  Bitmap.Height := Height;
```

```
  with Bitmap.Canvas do begin
    Brush.Color := clWhite;
    FillRect(Rect(0, 0, Width, Height));  // очистка канвы
    L := Length(Node);
    for i := 0 to L - 1 do                // рисование дуг
    with Node[i] do begin
      LL := Length(Edge);
      for j := 0 to LL - 1 do
      with Edge[j] do begin
        Pen.Color := Color;
        if Color = clBlack then Pen.Width := 2
        else Pen.Width := 1;
        Line_2(x0 + x1c, y0, Node[NumNode].x0 + x2c,
                             Node[NumNode].y0, yc, A);
      end;
    end;
    Brush.Color := clWhite;  Pen.Width := 1;
    for i := 0 to L - 1 do  // рисование узлов
    with Node[i] do begin
      if i = SelectNode then Pen.Color := clRed
                        else Pen.Color := clBlack;
      if Visit then Brush.Color := clSilver
               else Brush.Color := clWhite;
      RectAngle(x0 - hx, y0 - hy, x0 + hx, y0 + hy);
      s := Name;
      if N11.Checked then s:=s+ (' + IntToStr(Numbe) + ')';
      TextOut(x0-TextWidth(s) div 2,y0-TextHeight(s) div 2,s);
    end;
  end;
  Canvas.Draw(0, 0, Bitmap);
end;
```

7.8.3. Чтение и запись графов

Более интересным является механизм записи и чтения двухуровневой структуры узлы/ребра в один файл. Для реализации этого механизма предлагается использовать нетипизированный файл с файловым указателем `f: file`. При записи данных выполняются следующие действия (листинг 7.81).

Во-первых, записываем число узлов `L`.

Во-вторых, в цикле записываем информацию о каждом узле:

- координаты центра `x0`, `y0`;
- строку с именем узла;
- число ребер этого узла;

- информацию обо всех ребрах (для каждого ребра записываются: вес ребра; номер узла, на которое направлено это ребро; геометрические параметры ребра `x1c`, `x2c`, `yc`).

Для строки сначала записывается длина строки, а затем символы строки.

Листинг 7.81. Сохранение данных о графе

```
procedure SaveGraph(FileName: string);
var i, j: Integer;
  procedure WriteStr(s: string);
  var L: Word;
  begin
    L := Length(s);
    BlockWrite(f, L, SizeOf(L));  // Длина строки
    BlockWrite(f, s[1], L);       // Запись строки
  end;
begin
  AssignFile(f, FileName);  Rewrite(f, 1);
  L := Length(Node);
  BlockWrite(f, L, SizeOf(L));
  for i := 0 to L - 1 do
  with Node[i] do begin
    BlockWrite(f, x0, SizeOf(x0));
    BlockWrite(f, y0, SizeOf(y0));
    WriteStr(Name);
    LL := Length(Edge);
    BlockWrite(f, LL, SizeOf(LL));
    for j := 0 to LL - 1 do
    with Edge[j] do begin
      BlockWrite(f, A, SizeOf(A));
      BlockWrite(f, NumNode, SizeOf(NumNode));
      BlockWrite(f, x1c, SizeOf(x1c));
      BlockWrite(f, x2c, SizeOf(x2c));
      BlockWrite(f, yc, SizeOf(yc));
    end;
  end;
  CloseFile(f);
end;
```

Чтение файла происходит в том же порядке (листинг 7.82).

Листинг 7.82. Чтение данных о графе

```
procedure OpenGraph(FileName: string);
var i, j: Integer;
  function ReadStr: string;      // чтение строки
```

```
  var L: Word;
  begin
    BlockRead(f, L, SizeOf(L));  // длина строки
    SetLength(Result, L);
    BlockRead(f, Result[1], L);  // чтение строки
  end;
begin
  AssignFile(f, FileName);  Reset(f, 1);
  BlockRead(f, L, SizeOf(L));    // число узлов
  SetLength(Node, L);
  if L <> 0 then
  for i := 0 to L - 1 do         // чтение узлов
  with Node[i] do begin
    BlockRead(f, x0, SizeOf(x0));
    BlockRead(f, y0, SizeOf(y0));
    Name := ReadStr;               // чтение имени узла
    BlockRead(f, LL, SizeOf(LL));  // число дуг
    SetLength(Edge, LL);
    if LL <> 0 then
    for j := 0 to LL - 1 do        // чтение дуг узла
    with Edge[j] do begin
      BlockRead(f, A, SizeOf(A));
      BlockRead(f, NumNode, SizeOf(NumNode));
      BlockRead(f, x1c, SizeOf(x1c));
      BlockRead(f, x2c, SizeOf(x2c));
      BlockRead(f, yc, SizeOf(yc));
    end;
  end;
  CloseFile(f);
end;
```

Полный текст проекта представлен на компакт-диске в папке **Примеры | Глава 7 | Графы**.

7.9. Проект газификации домов

Разработка проекта "Газификация домов" также относится к разряду плоских схем. Предполагается, что бумажный проект дома готов и отсканирован в формате BMP или JPG.

Перед выполнением проекта небходимо:

- подготовить эскизы газификации дома, трассы трубы от газопровода до дома;
- расставить газовое оборудование на эскизе газификации дома. Должна быть возможность выбора газового оборудования из списка имеющегося;

- на эскизе газификации дома нарисовать трубы;
- иметь возможность редактирования (перемещение, удаление) оборудования и труб;
- при рисовании эскиза трассы трубы от газопровода до дома необходимо учитывать профиль поверхности земли, расставить бетонные трубы и показать глубину их прокладки;
- распечатать подготовленные эскизы и список необходимого оборудования.

Главная форма этого проекта имеет название **Газификация всей страны** и содержит четыре вкладки. На рис. 7.26. показан вид этой формы с открытой вкладкой **Эскизы**, на которой отображен эскиз газификации дома. Отдельным объектам (общим числом 16), из которых строится эскиз, соответствуют кнопки на инструментальной панели **Tools**, показанной в правой верхней части рис. 7.26.

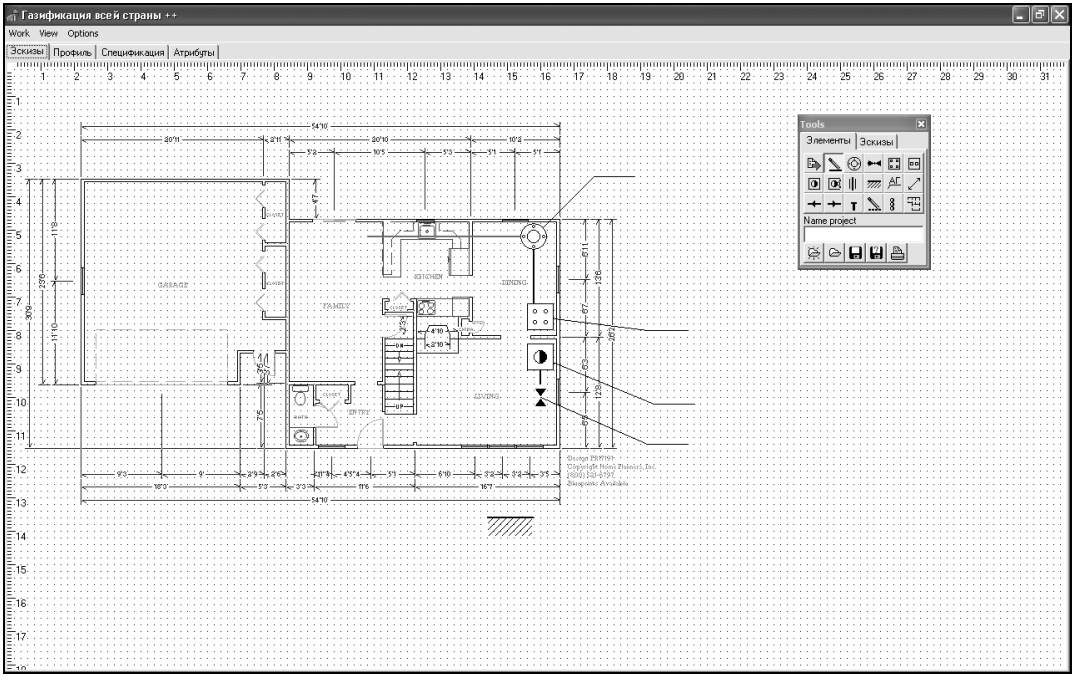


Рис. 7.26. Эскиз газификации дома

На рис. 7.27 показан вид главной формы **Газификация всей страны** с открытой вкладкой **Профиль**, на которой изображен эскиз трассы трубы от газопровода до дома.

В проекте имеется справочник оборудования. Редактирование этого справочника возможно на форме **Оборудование**, которая представлена на рис. 7.28.

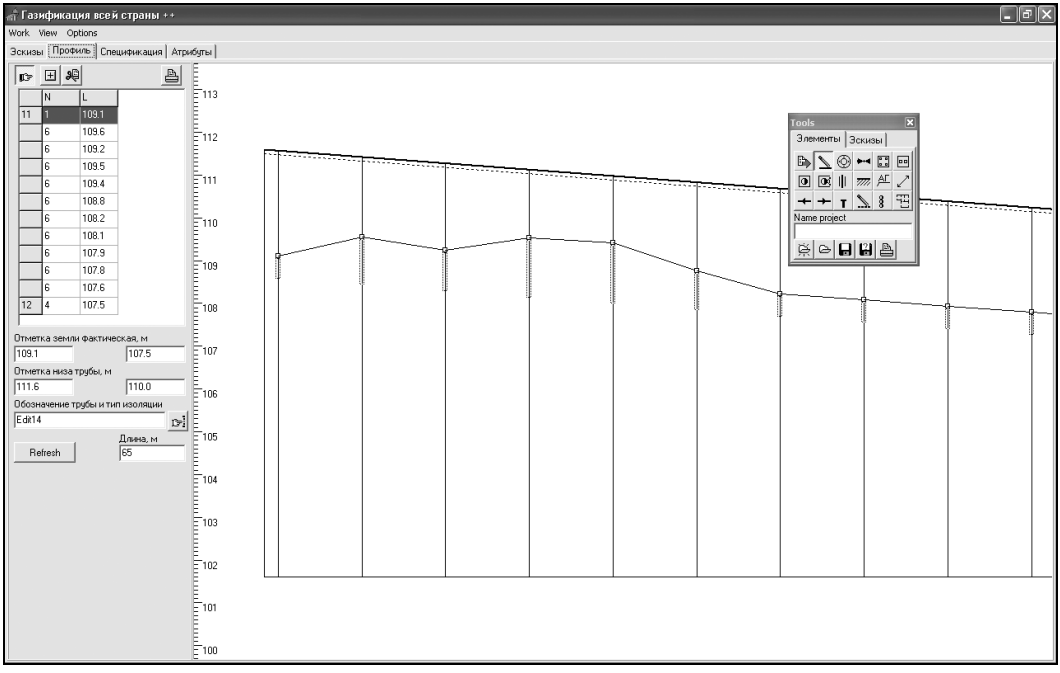
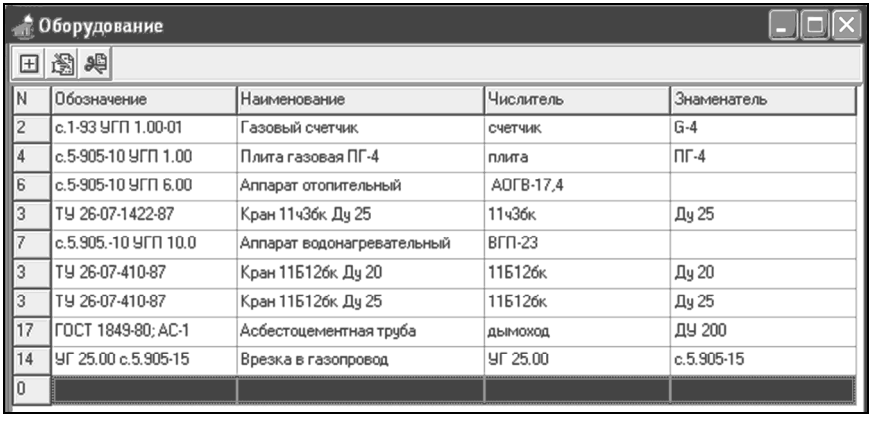


Рис. 7.27. Эскиз трассы трубы от газопровода до дома



N	Обозначение	Наименование	Числитель	Знаменатель
2	с.1-93 УГП 1.00-01	Газовый счетчик	счетчик	G-4
4	с.5-905-10 УГП 1.00	Плита газовая ПГ-4	плита	ПГ-4
6	с.5-905-10 УГП 6.00	Аппарат отопительный	АОГВ-17,4	
3	ТУ 26-07-1422-87	Кран 11ч3бк Ду 25	11ч3бк	Ду 25
7	с.5.905.-10 УГП 10.0	Аппарат водонагревательный	ВГП-23	
3	ТУ 26-07-410-87	Кран 11Б12бк Ду 20	11Б12бк	Ду 20
3	ТУ 26-07-410-87	Кран 11Б12бк Ду 25	11Б12бк	Ду 25
17	ГОСТ 1849-80; АС-1	Асбестоцементная труба	дымоход	ДУ 200
14	УГ 25.00 с.5.905-15	Врезка в газопровод	УГ 25.00	с.5.905-15
0				

Рис. 7.28. Справочник оборудования

7.9.1. Структура проекта

Проект состоит из следующих форм и модулей:

- `UnitMain` — главная форма;
- `UnitSpr` — форма справочника;

- `Dbl` — модуль общих типов и переменных;
- `UnitEditEquipment` — форма редактирования оборудования;
- `UnitEditListEquipment` — форма редактирования списка оборудования;
- `UnitScript` — форма редактирования подписей проекта;
- `UnitField` — форма редактирования выделенного объекта;
- `UnitTools` — инструментальная форма.

7.9.2. Структура данных

Любое оборудование, используемое в проекте, описывается структурой `TObj` (листинг 7.83).

Листинг 7.83. Структура для описания оборудования

```
PObj=^TObj;
TObj=record
  Id    : byte;   // Id: 0 - Find 1 - Line
  x1    : array[1..2] of TPoint;
  Select,
  g     : byte;
  Next  : PObj;
  adr   : integer;
  Name1,Name2: string[20];
end;
```

На основе этой структуры строятся три динамических списка, начало каждого из которых определяется массивом `HeadObj: array[0..2] of PObj` для трех эскизов проекта.

Во время проектирования оборудование попадает в эти списки из справочника оборудования, элементы которого имеют следующую структуру (листинг 7.84).

Листинг 7.84. Структура справочника оборудования

```
TEquipment=record // оборудование
  Id  : byte;
  Name: array[1..4] of string[20];
end;
```

Поля `Name: array[1..4] of string` этой структуры имеют следующий смысл: обозначение, наименование, числитель и знаменатель для выноски.

На рис. 7.29 представлена вкладка **Спецификация** главной формы проекта, на которой представлен справочник оборудования с кнопками добавления, редактирования и удаления оборудования.

Список включенного в проект оборудования может быть шире, чем список оборудования на эскизах, и должен редактироваться независимо. Для этого списка введена еще одна структура — `TListEquipment` (листинг 7.85).

Листинг 7.85. Структура включенного в проект оборудования

```
TListEquipment=record
  Id    : byte;
  typeEquipment: byte;
  Name1 : string[20];
  Name2 : string[20];
  Count : integer;
  Ed    : string[10];
  Rem   : string[10];
end;
```

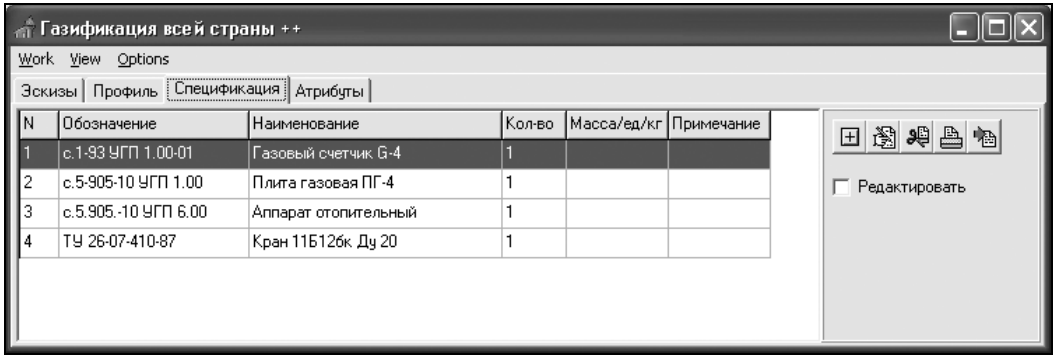


Рис. 7.29. Редактирование списка оборудования проекта

7.9.3. Рисование эскиза газификации дома

Рисование эскиза газификации дома происходит на канве `Image1` (листинг 7.86).

Листинг 7.86. Рисование эскиза газификации дома

```
procedure TFormMain.DrawProject;
var i,j: integer;
    hx,hy: integer;
begin
  hx:=4; hy:=4;
  with Image1.Canvas do begin
    Brush.Color:=clWhite; Pen.Color:=clWhite;
    RectAngle(0,0,I2,J2);
```

```
    if N2.Checked and (nameBMP<>'') then
      Draw(0,0,BitMap1);

    Pen.Color:=clBlack; Pen.Width := 1; // линейки
    if N6.Checked then begin
      for i:=3 to J2 div hy do begin
        MoveTo(0,i*hy);
        if i mod 10<>0 then LineTo(5,i*hy)
        else begin
          LineTo(10,i*hy);
          TextOut(10,i*hy,IntToStr(i div 10));
        end;
      end;
      for i:=3 to I2 div hx do begin
        MoveTo(i*hx,0);
        if i mod 10<>0 then LineTo(i*hy,5)
        else begin
          LineTo(i*hy,10);
          TextOut(i*hy,10,IntToStr(i div 10));
        end;
      end;
    end;

    if N1.Checked then
    for i:=1 to I2 div h do
      for j:=1 to J2 div h do
        Pixels[i*h,j*h]:=clBlack;

    if N3.Checked then begin
      if HeadObj[FormTools.Rg2.ItemIndex]<>Nil then begin
        Obj:=HeadObj[FormTools.Rg2.ItemIndex];
        repeat
          Pen.Color:=clBlack;
          if Obj.Select<>0 then Pen.Color:=clRed;
          DrawObj(pmCopy,2,Obj.fl,Obj);
          Obj:=Obj.Next;
        until Obj=Nil;
      end;
    end;
  end;
end;
```

Отличие данного примера от приведенных ранее — это рассмотрение работы со списком оборудования.

Рисование оборудования процедурой `DrawObj` имеет свои особенности: во-первых, каждый из 16-ти объектов рисуется по своему сценарию линиями, прямоугольни-

ками и эллипсами; во-вторых, некоторые объекты снабжаются выносками с надписями в числителе и знаменателе. В листинге 7.87 показано рисование некоторых объектов.

Листинг 7.87. Рисование оборудования

```
procedure TFormMain.DrawObj(AMode: TPenMode; PenWidth: byte;
  Obj: PObj);
const c50=50; c1=1; c2=2; c4=4; c5=5; c6=6; c9=9;
      c10=10; c12=12; c15=15; c13=13; c17=17;
      c18=18; c20=20; c22=22;
      alpha0=Pi/8; b20=12;
var Color: TColor; i: integer;
    alpha: real;
    T1,T2: TPoint;
begin
  T1:=Obj.x1[1]; T2:=Obj.x1[2];
  with Image1.Canvas do begin
    Pen.Color:=clBlack; Pen.Mode := AMode; Pen.Width := th;
    if Obj.fl in [2..8,10,14,17] then begin
      Pen.Width := 1;
      MoveTo(T1.X,T1.Y); LineTo(T2.X, T2.Y);
      LineTo(T2.X+c50, T2.Y);
      if (fl_tools=0) then begin
        TextOut(T2.X,T2.Y-c15,Obj.Name1);
        TextOut(T2.X,T2.Y+c2,Obj.Name2);
      end;
    end;
    case Obj.fl of
      0: if HeadObj[FormTools.Rg2.ItemIndex]<>Nil then
         with ObjSelect^ do
         case fl_move of
           0: begin
                MoveTo(x1[1].x+T2.x-T1.x,x1[1].y+T2.y-T1.y);
                LineTo(x1[2].x+T2.x-T1.x,x1[2].y+T2.y-T1.y);
              end;
           1: begin
                MoveTo(x1[1].x+T2.x-T1.x,x1[1].y+T2.y-T1.y);
                LineTo(x1[2].x,x1[2].y);
              end;
           2: begin
                MoveTo(x1[2].x+T2.x-T1.x,x1[2].y+T2.y-T1.y);
                LineTo(x1[1].x,x1[1].y);
              end;
         end;
```

```
      1: begin
           MoveTo(T1.X, T1.Y); LineTo(T2.X, T2.Y);
         end;
      2: begin
           Ellipse(T1.X-2*h, T1.Y-2*h,T1.X+2*h, T1.Y+2*h);
           Ellipse(T1.X-1*h, T1.Y-1*h,T1.X+1*h, T1.Y+1*h);

           Ellipse(T1.X-c2,T1.Y-1*h-c6,T1.X+c2,T1.Y-1*h-c2);
           Ellipse(T1.X-c2,T1.Y+1*h+c6,T1.X+c2,T1.Y+1*h+c2);
           Ellipse(T1.X-1*h-c6,T1.Y-c2,T1.X-1*h-c2,T1.Y+c2);
           Ellipse(T1.X+1*h+c6,T1.Y-c2,T1.X+1*h+c2,T1.Y+c2);
         end;
    …
    14: begin
           Brush.Style:=bsSolid;
           Brush.Color:=clBlack;
           Pen.Width := 1;
           RectAngle(T1.X-7,T1.Y-5,T1.X+5,T1.Y);
           Brush.Color:=clWhite;
         end;
     …
   end;
 end;
end;
```

К сожалению, подробное описание этого проекта в рамках книги невозможно. Полностью проект приведен на компакт-диске в папке **Примеры | Глава 7 | Газификация**.

ГЛАВА 8

Векторный редактор

В данной главе рассматривается проект создания многооконного векторного графического редактора. Векторный графический редактор, наподобие редактора Corel DRAW, значительно сложнее растрового, реализация которого была рассмотрена в *главе 7*. Даже существенно "облегченная" версия проекта, представленная далее, занимает на диске в 5 раз больше места, чем растровый редактор. Внешний вид окна многооконного векторного редактора представлен на рис. 8.1.

Проект состоит из трех форм: главной формы `FormMain` (с внешним именем **My Corel**), дочерней формы `FormEd` и формы `FormEdText`, предназначенной для редактирования текста. Структура проекта векторного редактора показана на рис. 8.2.

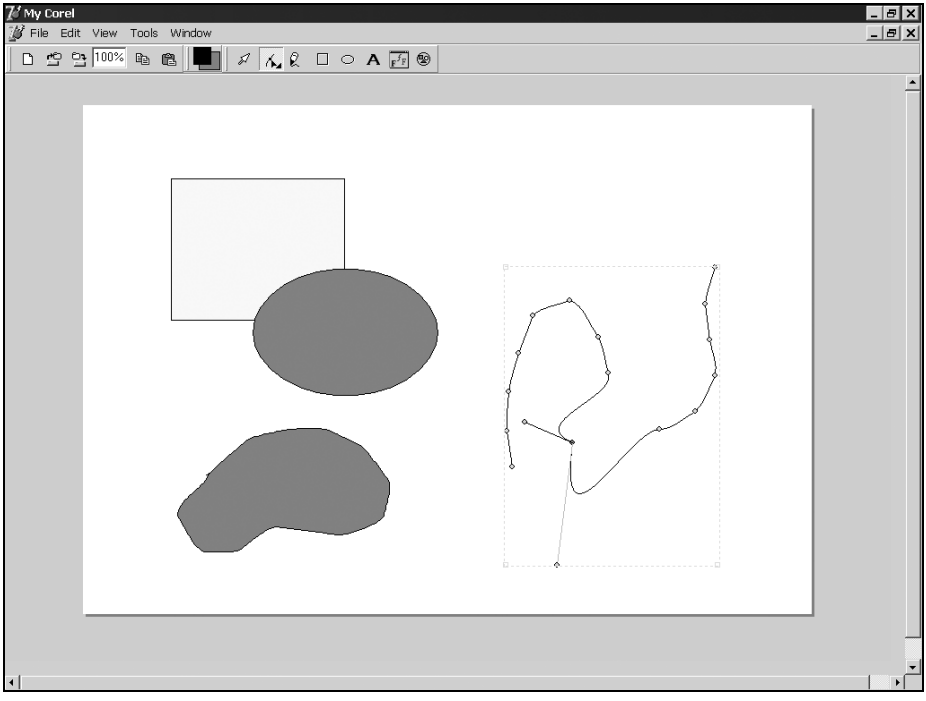


Рис. 8.1. Многооконный векторный редактор

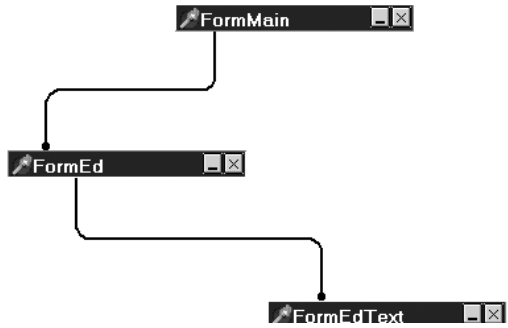


Рис. 8.2. Структура проекта векторного редактора

Большая часть функций, необходимых для работы проекта, сосредоточена в модуле `LibCorel`. В проекте представлены следующие функциональные возможности: рисование прямоугольника, эллипса, кривых Безье и фрагмента текста; перетаскивание, масштабирование и поворот этих объектов. Режимы, реализующие эти возможности, назначают кнопки типа `TSpeedButton`, изменяющие значение переменной `fl_tools` (табл. 8.1).

Таблица 8.1. *Инструментальные кнопки*

Имя	Значение	Назначение
`ButtonMove`	`tl_Move=0`	Перемещать объект
`ButtonPen`	`tl_AddLineBz=1`	Рисовать кривую Безье
`ButtonRotate`	`tl_Rotate=4`	Вращать объект
`ButtonMovePoint`	`tl_MovePoint=8`	Перемещать точку
`ButtonText`	`tl_Text=10`	Вставить текст
`ButtonRect`	`tl_Rect=17`	Рисовать прямоугольник
`ButtonEllipse`	`tl_Ellipse=18`	Рисовать эллипс

8.1. Структура данных

Одна из основных проблем, возникающих при создании векторного редактора — это проблема масштабирования. Поэтому в основной структуре проекта `TPage` (листинг 8.1) представлены поля `xMin`, `yMin`, `xMax`, `yMax`, задающие окно "на бумаге", ширину листа бумаги `PageWidth` и высоту листа бумаги `PageHeight`.

Листинг 8.1. Основная структура векторного редактора

```
TPage=record
  NumPointNode,NumPoint,NumObj,NumShape: integer;
```

```
    xMin,yMin,xMax,yMax : single;
    PageWidth,PageHeight: single;
    Obj         : TAObj;
  end;
```

Информация об объектах содержится в динамическом массиве `Obj: TAObj`, каждый элемент которого описывается структурой `TObj` (листинг 8.2).

Листинг 8.2. Структура векторных объектов

```
  PObj=^TObj;
  TObj=record
    Select      : boolean; // признак выделения
    PColor      : TColor;  // цвет линии
    BColor      : TColor;  // цвет заливки
    PWidth      : byte;    // толщина линии
    PStyle      : byte;    // стиль линии
    BStyle      : byte;    // стиль заливки
    Pc          : TXY;     // центр поворота
    Shape       : TA4XY;   // 4 компонента для углов четырехугольника
    typeObj     : byte;    // 0,1-Line 10-Text
    MyFont      : TFont;   // шрифт для фрагмента текста
    Text        : string;  // фрагмент текста
    Points      : array of TXYalf; // массив точек
  end;
  PAObj=^TAObj;
  TAObj=array of TObj;
```

Каждый объект может быть выделен. За выделение отвечает поле `Select` (рис. 8.3).

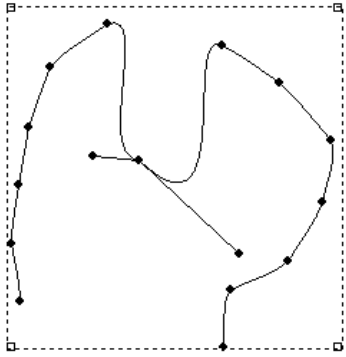


Рис. 8.3. Выделенный векторный объект

Каждый объект имеет следующие свойства: цвет линии `PColor`; стиль линии `PStyle`; толщина линии `PWidth`; цвет заливки `BColor`; стиль заливки `BStyle`. Объект можно

поворачивать, точка Pc:TXY задает точку поворота. Вокруг выделенного объекта рисуется пунктирный четырехугольник, вершины которого определяются массивом из четырех маркеров выделения Shape: TA4XY=array[0..3] of TXY. В дальнейшем мы будем называть эти элементы "шейпами" с номерами 0,..,3. За тип объекта отвечает поле typeObj, принимающее значения: to_BezierPen=0 для незамкнутой кривой Безье; to_BezierBrush=1 для замкнутой кривой Безье; to_Text=10 для фрагмента текста; to_Rect=17 для прямоугольника; to_Ellipse=18 для эллипса. Если объект является фрагментом текста, то поле MyFont задает свойства шрифта, а поле Text содержит текст. Самым важным в структуре TObj является поле Points, представляющее собой динамический массив координат точек типа TXYalf (листинг 8.3).

Листинг 8.3. Структура координат точек

```
TXYalf=record
  x,y,alf: single;
end;
```

В данной структуре, помимо обычных координат x,y, добавлено поле alf, отвечающее за угол касательной к линии в точке. Это поле необходимо для сглаживания кривых в точках сопряжения фрагментов кривых Безье. Более подробно мы будем обсуждать кривые Безье в *разд.* 8.3.

Вернемся к обсуждению структуры TPage. В этой структуре поле NumObj отвечает за номер выделенного объекта, поле NumShape указывает на номер "шейпа", т. е. на номер углового маркера прямоугольника выделения, поле NumPoint содержит номер выделенной точки линии, а поле NumPointNode — номер точки сопряжения.

8.2. Масштабирование

Проект оперирует со следующими объектами:

- страница. Весь проект — это массив страниц Pages: array of TPage (в нашем упрощенном случае одна страница Page). У каждой страницы есть ширина PageWidth и высота PageHeight, измеряемые в миллиметрах;
- физический прямоугольник — реальный двумерный участок, задаваемый параметрами xMin, yMin, xMax, yMax:real. Эти параметры измеряются в любых единицах длины, например, километрах, метрах или сантиметрах;
- экран монитора или принтер, на который проецируются страницы с окнами. Мы имеем дело с пикселами формы или канвы принтера, положение которых по горизонтали и вертикали задается целыми числами. Ширина формы задается параметром I2, а высота — параметром J2.

Необходимо спроецировать физический прямоугольник на страницу, а страницу спроецировать на экран монитора или на канву принтера.

При работе с экраном монитора необходимо обеспечить две функциональные возможности: *масштабирование* и *скроллинг*.

За масштабирование отвечают два параметра страницы: горизонтальный коэффициент масштабирования `Kx` и вертикальный коэффициент масштабирования `Ky`, которые при значении равном `1` должны давать на мониторе изображение, соответствующее изображению, отпечатанному на принтере или плоттере. Конечно, коэффициенты `Kx` и `Ky` связаны между собой и зависят от высоты экрана, ширины экрана, ширины и высоты страницы: `ky:=kx*PageHeight*I2/PageWidth/J2`.

За скроллинг отвечают два компонента: компонент горизонтального скроллинга `SH` и компонент вертикального скроллинга `SV` типа `TScrollBar`.

Основным свойством компонентов скроллинга является свойство `Position`, которое может меняться от минимального значения `Min=0` до максимального значения `Max=100`. Если ползунки компонентов скроллинга находятся в середине своих возможных значений, то центр страницы должен совпадать с центром формы.

Необходимо описать еще одну константу — коэффициент для полей страницы λ. Будем считать, что значение этой константы равно 5/60.

Выведем функциональную зависимость между координатами экрана и координатами страницы.

Найдем зависимость координат точки x, y на странице от координат точки на экране I, J.

На экран необходимо вывести окно — часть страницы с левым верхним углом u_1, v_1 и нижним правым углом u_2, v_2. Очевидно, что соответствующие уравнения преобразования будут линейны:

$$\frac{x-u_1}{u_2-u_1}=\frac{I}{I_2}, \tag{8.1}$$

$$\frac{y-v_1}{v_2-v_1}=\frac{J}{J_2}. \tag{8.2}$$

Ширина и высота окна на странице зависят от коэффициентов масштабирования k_x и k_y:

$$u_2-u_1=k_xL_w; \tag{8.3}$$

$$v_2-v_1=k_yL_h. \tag{8.4}$$

Рассмотрим сначала задачу преобразования горизонтальных координат. Ясно, что u_1 и u_2 зависят от коэффициента масштабирования k_x и линейно зависят от положения компонента горизонтального скроллинга S_w:

$$u_1=f_1(k_x,S_w)=A(k_x)\times S_w+B_1(k_x); \tag{8.5}$$

$$u_2=f_2(k_x,S_w)=A(k_x)\times S_w+B_2(k_x). \tag{8.6}$$

При минимальном положении компонента горизонтального скроллинга $S_w = 0$ значение параметра u_1 должно быть немного левее левой границы страницы. Используя коэффициент для полей страницы λ, запишем условие в виде:

$$u_1(S_w = 0) = -\frac{L_w}{\lambda} = B_1. \tag{8.7}$$

При максимальном положении компонента горизонтального скроллинга $S_w = S_{\max}$ значение параметра u_2 должно быть немного правее правой границы страницы. Используя коэффициент для полей страницы λ, запишем это условие в виде:

$$u_2(S_w = S_{\max}) = L_w + \frac{L_w}{\lambda} = A \times S_{\max} + B_2. \tag{8.8}$$

Из уравнения (8.3) следует, что

$$u_2 - u_1 = B_2 - B_1 = k_x L_w. \tag{8.9}$$

Подставив в уравнение (8.9) значение B_1 из уравнения (8.7), получаем для B_2:

$$B_2 = L_w \times (k_x - \frac{1}{\lambda}). \tag{8.10}$$

Тогда из уравнения (8.8) следует, что

$$A = \frac{L_w}{S_{\max}} \times (1 + \frac{2}{\lambda} - k_x). \tag{8.11}$$

Подставив A, B_1 и B_2 в уравнение (8.3), найдем экранную координату I через координату страницы x:

$$I = \frac{I_2}{k_x}(\frac{x}{L_w} - \frac{S_w}{S_{\max}} \times (1 + \frac{2}{\lambda} - k_x) + \frac{1}{k_x}). \tag{8.12}$$

Уравнение (8.12) реализовано в виде функции масштабирования `II(x)`, текст которой приведен в листинге 8.4.

Листинг 8.4. Прямая функция масштабирования горизонтальных координат

```
function II(x: real): integer;
begin
  with FormEd,Page do
    Result:=Round(I2*kx*(x/PageWidth-(SH.Max-
  SH.Position)/SH.Max*(-1-2*ka+1/kx)-1-ka+1/kx));
end;
```

Решив уравнение (8.12) относительно x, найдем зависимость горизонтальной страничной координаты от экранной координаты I. В листинге 8.5 приведена функция обратного масштабирования горизонтальных координат.

Листинг 8.5. Обратная функция масштабирования горизонтальных координат

```
function XX(I: integer): real;
begin
  with FormMain,PrGIS.Pages[NumPage] do
    Result:=Lw*(I/I2/kx - (SH.Max-SH.Position)/SH.Max*(1+2*ka-
    1/kx)+1+ka-1/kx);
end;
```

Точно так же можно найти закон преобразования для вертикальных координат. При этом необходимо учитывать, что нумерация строк пикселов на экране идет сверху вниз.

$$J = \frac{J_2}{k_y}\left(\frac{y}{L_h} - \frac{S_{\max} - S_h}{S_{\max}} \times \left(1 + \frac{2}{\lambda} - k_y\right) - 1 - \lambda + \frac{1}{k_y}\right). \qquad (8.13)$$

Уравнение (8.13) реализовано в виде функции масштабирования `JJ(y)`, текст которой приведен в листинге 8.6.

Листинг 8.6. Прямая функция масштабирования вертикальных координат

```
function JJ(y: real): integer;
begin
  with FormMain,PrGIS.Pages[NumPage] do
    Result:=Round(J2*ky*(y/Lh-(SV.Max-SV.Position)/SV.Max*(-1-2*ka+1/ky)-1-
ka+1/ky));
end;
```

Решив уравнение (8.13) относительно y, найдем зависимость вертикальной страничной координаты от экранной координаты J. В листинге 8.7 приведена функция обратного масштабирования вертикальных координат.

Листинг 8.7. Функция обратного масштабирования вертикальных координат

```
function YY(J: integer): real;
begin
  with FormMain,PrGIS.Pages[NumPage] do
    Result:=Lh*(J/J2/ky - (SV.Max-SV.Position)/SV.Max*(1+2*ka-1/ky)+1+ka-
1/ky);
end;
```

8.3. Кривые Безье

В векторной параметрической форме уравнения, описывающие кривые Безье (рис. 8.4) по четырем точкам V_0, V_1, V_2, V_3, выглядят так [97, 98]:

$$X = V_0 \times t^3 + 3 \times V_1 \times t^2 \times (1-t) + 3 \times V_2 \times t \times (1-t)^2 + V_3 \times (1-t)^3. \qquad (8.14)$$

В этом уравнении параметр $t \in [0,1]$.

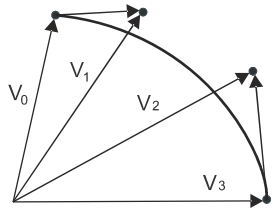


Рис. 8.4. Фрагмент кривой Безье

8.4. Создание объектов

Начнем с самых простых объектов: прямоугольника и фрагмента текста. При нажатии кнопки мыши мы, как обычно, поднимаем флаг перемещения `drawing:=true`, запоминаем координаты левого верхнего и нижнего правого углов и рисуем фокусный четырехугольник (листинг 8.8).

Листинг 8.8. Начало рисования прямоугольника и текста

```
tl_Rect,tl_Text: // Rect,Text
   begin
     drawing:=true;
     x0:=x; y0:=y; xt:=x; yt:=y;
     Canvas.DrawFocusRect(Rect(x0,y0,xt,yt));
   end;
```

При перемещении указателя мыши мы стираем фокусный четырехугольник, меняем координаты нижнего правого угла и рисуем фокусный четырехугольник в новом месте (листинг 8.9).

Листинг 8.9. Перемещение указателя мыши при рисовании прямоугольника и текста

```
tl_Rect,tl_Text: // Rect,Text
   begin
     Canvas.DrawFocusRect(Rect(x0,y0,xt,yt));
     xt:=x; yt:=y;
     Canvas.DrawFocusRect(Rect(x0,y0,xt,yt));
   end;
```

Когда создаются прямоугольник или фрагмент текста, событие отпускания кнопки мыши обрабатывается следующим образом: увеличиваем на `1` длину динамического массива, процедурой `SetNewLine` задаем параметры объекта `(PStyle, PColor,`

PWidth, BColor, BStyle, typeObj), создаем динамический массив Points из 4 элементов под угловые точки прямоугольника и заполняем эти элементы массива координатами точек (листинг 8.10).

Листинг 8.10. Завершение рисования прямоугольника и текста

```
tl_Rect,tl_Text: // Add RectAngle, Text
   begin
     Canvas.DrawFocusRect(Rect(x0,y0,xt,yt));
     L0:=Length(Obj);
     L0:=L0+1; NumObj:=L0-1; SetLength(Obj,L0);
     Obj[L0-1].Select:=true;
     SetLength(Obj[NumObj].Points,4);
     SetNewLine(@Obj[NumObj],fl_tools);
     with Obj[NumObj] do begin
       Points[0].x:=XX(x0); Points[0].y:=YY(y0);
       Points[1].x:=XX(x0); Points[1].y:=YY(yt);
       Points[2].x:=XX(xt); Points[2].y:=YY(yt);
       Points[3].x:=XX(xt); Points[3].y:=YY(y0);
     end;
     if fl_tools=tl_Text then begin // для текста
       Obj[L0-1].MyFont:=TFont.Create;
       Obj[L0-1].MyFont.Assign(Font);
       with Obj[NumObj] do begin
         PStyle:=5; PColor:=clBlack; BStyle:=1;
         If FormEdText.ShowModal=mrOk then
           Text:=FormEdText.Memo1.Text;
       end;
     end;
   end;
```

При создании фрагмента текста необходимо выполнить некоторые дополнительные действия: во-первых, методом TFont.Create создать поле MyFont; во-вторых, методом MyFont.Assign(Font) передать этому полю значение свойства Font; в третьих, вызвать форму FormEdText для редактирования текста.
Создание эллипса заканчивается несколько иначе (листинг 8.11).

Листинг 8.11. Завершение рисования эллипса

```
tl_Ellipse:
   begin
     Canvas.Pen.Mode:=pmCopy;
     L0:=Length(Obj);
     L0:=L0+1; NumObj:=L0-1; SetLength(Obj,L0);
     Obj[L0-1].Select:=true;
     x_0:=(XX(x0)+XX(xt))/2; y_0:=(YY(y0)+YY(yt))/2;
     Ea:=Abs((XX(x0)-XX(xt))/2);
```

```
        Eb:=Abs((YY(y0)-YY(yt))/2);
        L:=NUM_POINT_ELLIPSE;
        SetLength(Obj[NumObj].Points,L);
        SetNewLine(@Obj[NumObj],fl_tools);
        with Obj[NumObj] do
        for i:=0 to L-1 do
        with Points[i] do begin
          Points[i].x:=x_0+Ea*cos(2*i*Pi/L);
          Points[i].y:=y_0+Eb*sin(2*i*Pi/L);
        end;
      end;
```

Вместо четырех угловых точек прямоугольника мы создаем `NUM_POINT_ELLIPSE=60` точек на эллипсе, координаты которых вычисляем по параметрическому уравнению эллипса.

Наиболее сложным процессом является рисование кривой Безье. На начальном этапе добавляется только одна точка в массив `Points` (листинг 8.12).

Листинг 8.12. Начало рисования кривой Безье

```
tl_AddLineBz:
  begin         // Add Line Bezier (typObj=0,1)
    UnSelect;
    drawing:=true;
    L0:=Length(Obj);
    L0:=L0+1; NumObj:=L0-1; SetLength(Obj,L0);
    Obj[L0-1].Select:=true;
    SetLength(Obj[NumObj].Points,1); // первая точка линии
    x0:=x; y0:=y; xt:=x; yt:=y;
    SetNewLine(@Obj[NumObj],0);   // назначение свойств линии
    with Obj[NumObj] do begin
      Points[L-1].x:=XX(x0); Points[L-1].y:=YY(y0);
    end;
    FormMain.ButtonSave.Enabled:=true;
  end;
```

Напомним, что кривая Безье проходит через `3N+1` точку, поэтому при перемещении указателя мыши мы должны добавлять по 3 точки, если отклонение положения указателя от последней точки кривой больше `DELTA_BZ=40` пикселов (листинг 8.13).

Листинг 8.13. Добавление трех точек при перемещении указателя мыши

```
  tl_AddLineBz:
    with Obj[NumObj] do       // Add Line Points
    if (Abs(x-x0)>=DELTA_BZ) or (Abs(y-y0)>=DELTA_BZ) then
    begin
      L:=Length(Points); L:=L+3;
      SetLength(Points,L); //добавить точки
      Points[L-1].x:=XX(x); Points[L-1].y:=YY(y);
```

```
    dx:=Points[L-1].x-Points[L-4].x;
    dy:=Points[L-1].y-Points[L-4].y;
    if L=4 then Points[0].alf:=ArcTan2(dy,dx);
    Points[L-1].alf:=ArcTan2(dy,dx)+Pi;
    if L>=6 then          // поправка угла
      Points[L-4].alf:=
        ArcTan2(Points[L-1].y-Points[L-7].y,
                Points[L-1].x-Points[L-7].x)+Pi;
    SetBezier(L-1,@Obj[NumObj]);
    x0:=x; y0:=y;
  end
  else begin
    n:=FindPointSel(XX(x),YY(y),@Obj[NumObj],Eps);
    L:=Length(Points);
    if (L>1) and (n=0) then begin // замыкание
      L:=L+3; SetLength(Points,L);
      with Points[L-1] do begin
        typeObj:=1;
        Points[L-1].x:=Points[n].x;
        Points[L-1].y:=Points[n].y;
      end;
      dx:=Points[L-1].x-Points[L-4].x;
      dy:=Points[L-1].y-Points[L-4].y;
      if L=4 then Points[0].alf:=ArcTan2(dy,dx);
      Points[L-1].alf:=ArcTan2(dy,dx)+Pi;
      SetBz(L-1,@Obj[NumObj]);
    end;
  end;
```

При добавлении новых трех точек вычисляется угол в новой точке сопряжения `Points[L-1].alf`, корректируется угол в предыдущей точке сопряжения `Points[L-4].alf` и процедурой `SetBezier` вычисляются координаты в добавленных промежуточных точках с номерами `L-2` и `L-3` (рис. 8.5).

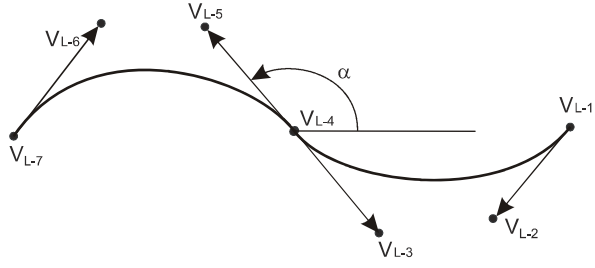


Рис. 8.5. Добавление фрагмента кривой Безье

Если во время перемещения курсор попадает в начальную точку кривой Безье, то меняется тип линии `typeObj:=1`, добавляется три новых точки и кривая замыкается.

8.5. Перемещение объектов

Любой объект может быть окружен прямоугольником выделения, угловые точки которого определяются координатами `Shape: array[0..3] of TXY`. В начале перемещения мы обращаемся к функции `FindObj`, которая должна по координатам точки `x, y` попытаться определить номер объекта `NumObj` и, возможно, если точка `x, y` попадает в окрестность угловой точки, определить номер этой точки `NumShape` (листинг 8.14).

Листинг 8.14. Функция поиска объекта

```
function FindObj(u,v: real; Eps: real): boolean;
var i,L0: integer; Ok: boolean;
begin
  with Page do begin
    NumObj:=-1; NumShape:=-1; NumPointNode:=-1;
    L0:=Length(Obj); i:=L0; Result:=false; // поиск объекта
    while (i>0) and not Result do begin
      Dec(i);
      with Obj[i] do
      Result:=
       (Shape[0].x-Eps<=u) and (u<=Shape[2].x+Eps) and
       (Shape[0].y-Eps<=v) and (v<=Shape[2].y+Eps);
    end;
    if Result then begin                    // найден Object
      NumObj:=i; Obj[i].Select:=true;
      Ok:=false; i:=-1;
      while (i<3) and not Ok do begin       // поиск Shape
        Inc(i);
        with Obj[NumObj] do
          Ok:=(Abs(Shape[i].x-u)<Eps) and
              (Abs(Shape[i].y-v)<Eps);
      end;
      if Ok then NumShape:=i;               // найден Shape
    end
  end;
end;
```

Действия, которые выполняет обработчик события `onMouseDown` при перемещении объектов, показаны в листинге 8.15.

Листинг 8.15. Начало перемещения объектов

```
tl_Move:
 begin                          // Move
   UnSelect;
   if (Length(Obj)<>0) and FindObj(XX(x),YY(y),Eps) then
   begin                        // Move Object
     if NumShape=-1 then begin  // Move RectAngle
       drawing:=true;
       x0:=x; y0:=y;
     end
     else                       // Move Shape
     if NumObj<>-1 then begin
       drawing:=true;
       with Obj[NumObj] do
       case NumShape of
         0: begin
             x0:=II(Shape[2].x); y0:=JJ(Shape[2].y);
            end;
         1: begin
             x0:=II(Shape[0].x); y0:=JJ(Shape[2].y);
            end;
         2: begin
             x0:=II(Shape[0].x); y0:=JJ(Shape[0].y);
            end;
         3: begin
             x0:=II(Shape[2].x); y0:=JJ(Shape[0].y);
            end;
       end;
     end;
     ShowBz(1);
   end
 end;
```

Функция `FindObj` устанавливает номер объекта `NumObj` и номер угловой точки прямоугольника выделения `NumShape`. Если `NumShape=-1`, то перемещается объект и запоминаются координаты текущей точки `x0`, `y0`. Иначе перемещается угловая точка прямоугольника выделения `Shape` с номером `NumShape` и координаты противоположного угла прямоугольника запоминаются как `x0`, `y0`.

При перемещении указателя мыши обрабатывается событие `onMouseMove` (листинг 8.16).

Листинг 8.16. Действия при перемещении указателя мыши

```
tl_Move:                                // Move Obj
   if NumObj<>-1 then begin
     if NumShape=-1 then begin          // Move Rect
```

```
        dx:=XX(x)-XX(x0); dy:=YY(y)-YY(y0);
        MoveObj(@Obj[NumObj],dx,dy);
        x0:=x; y0:=y;
      end
      else                                  // Move Shape
        SetMoveShape(@Obj[NumObj],XX(x),YY(y),NumShape);
      ShowBz(0);
      FormMain.ButtonSave.Enabled:=true;
    end;
```

При перемещении всего объекта вызывается процедура `MoveObj` (листинг 8.17), в которой на `dx,dy` увеличиваются координаты точек `Shape`, центра вращения и всех точек объекта `Points`.

Листинг 8.17. Процедура перемещения объекта

```
procedure MoveObj(Obj: PObj; dx,dy: single);
var i: word;
begin
  with Obj^ do begin
    for i:=0 to 3 do
    with Shape[i] do begin
      x:=x+dx; y:=y+dy;
    end;
    L:=Length(Points);
    if l>0 then
    for i:=0 to L-1 do
    with Points[i] do begin
      x:=x+dx; y:=y+dy;
    end;
    Pc.x:=Pc.x+dx; Pc.y:=Pc.y+dy;
  end;
end;
```

Если перемещается одна из угловых точек прямоугольника выделения ("шейп"), то вызывается процедура `SetMoveShape` (листинг 8.18), которая реализует масштабирование точек объекта с различными неподвижными угловыми точками.

Листинг 8.18. Перемещение "шейпа"

```
procedure SetMoveShape(Obj: PObj; xm,ym: real; NumShape: integer);
var L,i: word;
    kx,ky,dx,dy: real;
begin
   with Obj^ do begin
     L:=Length(Points);
     dx:=Shape[2].x-Shape[0].x; dy:=Shape[2].y-Shape[0].y;
     case NumShape of
```

```
      0: begin
           kx:=(Shape[2].x-xm)/dx; ky:=(Shape[2].y-ym)/dy;
           for i:=0 to L-1 do
           with Points[i] do begin
             x:=xm+kx*(x-Shape[0].x);
             y:=ym+ky*(y-Shape[0].y);
           end;
           Shape[0].x:=xm; Shape[0].y:=ym;
         end;
      1: begin
           kx:=(xm-Shape[0].x)/dx; ky:=(Shape[2].y-ym)/dy;
           for i:=0 to L-1 do
           with Points[i] do begin
             x:=Shape[0].x+kx*(x-Shape[0].x);
             y:=ym+ky*(y-Shape[0].y);
           end;
           Shape[2].x:=xm; Shape[0].y:=ym;
         end;
      2: begin
           kx:=(xm-Shape[0].x)/dx; ky:=(ym-Shape[0].y)/dy;
           for i:=0 to L-1 do
           with Points[i] do begin
             x:=Shape[0].x+kx*(x-Shape[0].x);
             y:=Shape[0].y+ky*(y-Shape[0].y);
           end;
           Shape[2].x:=xm; Shape[2].y:=ym;
         end;
      3: begin
           kx:=(Shape[2].x-xm)/dx; ky:=(ym-Shape[0].y)/dy;
           for i:=0 to L-1 do
           with Points[i] do begin
             x:=xm+kx*(x-Shape[0].x);
             y:=Shape[0].y+ky*(y-Shape[0].y);
           end;
           Shape[0].x:=xm; Shape[2].y:=ym;
         end;
    end;
    SetMinMax(Obj);
    Shape[1].x:=Shape[2].x; Shape[1].y:=Shape[0].y;
    Shape[3].x:=Shape[0].x; Shape[3].y:=Shape[2].y;
  end;
end;
```

8.6. Поворот объектов

Поворот объекта начинается в обработчике события `onMouseDown` с проверки выделенности объекта `NumObj<>-1` и определения номера выделенной угловой точки прямоугольника выделения `NumShape` (листинг 8.19).

Листинг 8.19. Начало поворота объекта

```
tl_Rotate:
   drawing:=(NumObj<>-1) and (Length(Obj)<>0) and
     FindObj(XX(x),YY(y),Eps) and (NumShape<>-1);
```

Если `drawing=true`, то в обработчике события `onMouseMove` определяется угол между точкой `x,y` и центром вращения (листинг 8.20).

Листинг 8.20. Вращение объекта

```
tl_Rotate:
   if NumObj<>-1 then
   with Obj[NumObj] do begin // Rotate Curve
     al:=ArcTan2(YY(y)-Pc.y,XX(x)-Pc.x);
     if Abs(al-a)>0.1 then begin
       RotateObj(@Obj[NumObj],Pc.x,Pc.y,al-a);
       ShowBz(0);
     end;
   end;
```

Затем вызывается процедура `RotateObj` для изменения координат точек объекта (листинг 8.21).

Листинг 8.21. Процедура изменения координат объекта при вращении

```
procedure RotateObj(Obj: PObj; x0,y0,al0: single);
var
  i: word;
  R,al: single;
begin
  with Obj^ do begin
    L:=Length(Points);
    for i:=0 to L-1 do
    with Points[i] do begin
      R:=Sqrt(Sqr(x-x0)+Sqr(y-y0));
      al:=ArcTan2(y-y0,x-x0);
      x:=x0+R*cos(al+al0);
      y:=y0+R*sin(al+al0); alf:=alf+al0;
```

```
    end;
  end;
  SetMinMax(Obj);
end;
```

8.7. Перемещение точек

Перемещение точек начинается с проверки выделенности объекта и определения номера перемещаемой точки функцией `FindPointBezier` (листинг 8.22).

Листинг 8.22. Начало перемещения точки

```
tl_MovePoint:
  drawing:=(Length(Obj)<>0) and (NumObj<>-1) and
    FindPointBezier(XX(x),YY(y),@Obj[NumObj],
       NumPoint,NumPointNode,Eps);
```

Функция `FindPointBezier` определяет два номера точек: на первом этапе определяется номер точки сопряжения `NumPointNode`, для которой выполняется условие `i mod 3=0`; на втором этапе проверяются промежуточные точки `(i mod 3<>0)` и определяется номер точки `NumPoint` (листинг 8.23).

Листинг 8.23. Функция поиска точки

```
function FindPointBezier(u,v: real; Obj: PObj;
  var NumPoint,NumPointNode:integer; Eps:real):boolean;
var i,L: integer;
begin
  NumPoint:=-1;
  L:=Length(Obj^.Points); i:=-3; Result:=false;
  while (i<L-1) and not Result do begin
    Inc(i,3);
    with Obj^.Points[i] do
      Result:=(Abs(x-u)<Eps) and (Abs(y-v)<Eps);
  end;
  if Result then begin
    NumPointNode:=i; NumPoint:=i;
  end
  else begin
    i:=-1;
    while (i<L-1) and not Result do begin
      Inc(i);
      if i mod 3<>0 then
      with Obj^.Points[i] do
        Result:=(Abs(x-u)<Eps) and (Abs(y-v)<Eps);
```

```
    end;
    if Result then NumPoint:=i else NumPointNode:=-1;
  end;
end;
```

8.8. Прорисовка объектов

Для ускорения работы приложения рисование объектов реализуется процедурой `DrawObj` на канве `Bitmap` с последующим копированием на канву формы (листинг 8.24).

Листинг 8.24. Процедура рисования объектов

```
procedure TFormEd.DrawObj(fl: byte);
var
  i,iLin: integer;
  CanvasT: TCanvas;
begin
  if fl=2 then // Printer
  with Printer do begin
    I2:=PageWidth; J2:=PageHeight; CanvasT:=Canvas;
    BeginDoc;
  end
  else begin
    I2:=ClientWidth; J2:=ClientHeight;
    CanvasT:=Bitmap.Canvas;
  end;
  Bitmap.Width:=I2; Bitmap.Height:=J2;
  Ky:=Kx*I2/J2;
  Caption:=ExtractFileName(FName);
  with CanvasT do begin
    Brush.Color:=$00E1E707;
    FillRect(Rect(0,0,I2,J2)); // очистка канвы
    Brush.Color:=clWhite;
    FillRect(Rect(II(0),JJ(0), // лист бумаги
      II(Page.PageWidth),JJ(Page.PageHeight)));
    with Page do begin
      Pen.Color:=clGray;
      Pen.Style:=psSolid;    // тень от бумаги
      for i:=0 to 2 do begin
        MoveTo(II(0)+3,JJ(PageHeight)+i);
        LineTo(II(PageWidth)+3,JJ(PageHeight)+i);
      end;
      for i:=0 to 2 do begin
```

```
        MoveTo(II(PageWidth)+i,JJ(PageHeight)+1);
        LineTo(II(PageWidth)+i,JJ(0)+3);
      end;
      L0:=Length(Obj);       // рисование линий
      for iLin:=0 to L0-1 do
      with Obj[iLin] do
      if Visible then
      case typeObj of
        to_Bezier,to_BezierFill: // кривые Безье
           if Length(Points)>=4 then begin
             DrawBezier(fl,II,JJ,@Obj[iLin],CanvasT);
             if (iLin=NumObj) then begin // Line & Ellipse
               DrawSelectBz(NumPointNode,CanvasT);
               if NumPointNode<>-1 then
                 DrawLine_Ellipse(NumPointNode,II,JJ,
                   @Obj[iLin], CanvasT);
               if fl_tools=tl_Rotate then
                 DrawRotate(II(Pc.x),JJ(Pc.y),CanvasT);
               DrawRect(Shape,II,JJ,CanvasT);
             end;
           end;
        to_Text,to_Rect,to_Ellipse:
           if Length(Points)>0 then begin // PolyLine
             DrawPolygon(fl,II,JJ,@Obj[iLin], CanvasT);
             if Select then
               DrawSelectPolygon(typeObj,CanvasT);
             if typeObj=tl_Text then           // Text
               MyOutText(@Obj[iLin],II,JJ,Kx, CanvasT);
             if iLin=NumObj then begin // Line & Ellipse
               if fl_tools=tl_Rotate then
                 DrawRotate(II(Pc.x),JJ(Pc.y),CanvasT);
               DrawRect(Shape,II,JJ, CanvasT);
             end;
           end;
      end; // case
    end;   // Page
  end;
  if fl=2 then // Printer
  with Printer do
    EndDoc
  else
    Canvas.Draw(0,0,Bitmap);
end;
```

8.9. Печать

Печать на принтере или на графопостроителе осуществляется той же процедурой DrawObj, которая используется при рисовании на канве формы. Но вызов ее происходит несколько иначе (листинг 8.25).

Листинг 8.25. Вызов процедуры рисования для печати

```
procedure TFormMain.ButtonPrintClick(Sender: TObject);
begin
  if PrintDialog1.Execute then
  if ActiveMDIChild<>nil then
  with ActiveMDIChild as TFormEd do
    DrawObj(2,IIP,JJP);
end;
```

Во-первых, при вызове процедуры DrawObj ей передаются указатели на функции масштабирования IIP и JJP, которые используют ширину и высоту канвы принтера.

```
function IIP(x: real): integer;
begin
  with Printer,Page do
  Result:=Trunc((x-xMin)*PageWidth/(xMax-xMin));
end;

function JJP(y: real): integer;
begin
  with Printer,Page do
  Result:=Trunc((y-yMin)*PageHeight/(yMax-yMin));
end;
```

Во-вторых, значением параметра fl=2 мы сообщаем процедуре DrawObj о том, что надо перед рисованием дать команду BeginDoc, а после рисования — команду EndDoc.

В третьих, рисование происходит не на канве Bitmap, на канве принтера.

8.10. Запись и чтение данных

При записи данных проекта возникают три проблемы:

1. Данные представляют собой трехуровневое дерево, т. к. число объектов на странице может быть произвольным и число точек объекта также может быть произвольным;
2. Некоторые объекты содержат строки произвольной длины;

3. Для объектов со строками необходимо сохранять параметры шрифта, а стиль шрифта — это свойство множественного типа.

Все эти проблемы позволяет решить запись в нетипизированный файл с указателем `f: file`.

Процедура `SaveVRT`, приведенная в листинге 8.26, осуществляет запись в такой файл. Обратите внимание на то, что перед записью объектов сначала записывается число объектов `L`, а перед записью точек записывается число точек `L1`.

Листинг 8.26. Запись проекта в нетипизированный файл

```
procedure SaveVRT(FileName: string);
var
  i,L,j,L1: integer;
  w: word;
begin
  AssignFile(f,FileName); Rewrite(f,1);
  with Page do begin
    BlockWrite(f,NumPointNode,SizeOf(NumPointNode));
    BlockWrite(f,NumPoint,SizeOf(NumPoint));
    BlockWrite(f,NumObj,SizeOf(NumObj));
    BlockWrite(f,NumShape,SizeOf(NumShape));
    BlockWrite(f,xMin,SizeOf(xMin));
    BlockWrite(f,yMin,SizeOf(yMin));
    BlockWrite(f,xMax,SizeOf(xMax));
    BlockWrite(f,yMax,SizeOf(yMax));
    BlockWrite(f,PageWidth,SizeOf(PageWidth));
    BlockWrite(f,PageHeight,SizeOf(PageHeight));
    L:=Length(Obj);
    BlockWrite(f,L,SizeOf(L)); // запись числа объектов
    if L>0 then
    for i:=0 to L-1 do
    with Obj[i] do begin
      BlockWrite(f,Select,SizeOf(Select));
      BlockWrite(f,PColor,SizeOf(PColor));
      BlockWrite(f,BColor,SizeOf(BColor));
      BlockWrite(f,PWidth,SizeOf(PWidth));
      BlockWrite(f,PStyle,SizeOf(PStyle));
      BlockWrite(f,BStyle,SizeOf(BStyle));
      BlockWrite(f,Pc,SizeOf(Pc));
      BlockWrite(f,Rt,SizeOf(Rt));
      BlockWrite(f,a,SizeOf(a));
      BlockWrite(f,Shape,SizeOf(Shape));
      BlockWrite(f,typeObj,SizeOf(typeObj));
      WriteStr(Text);
```

```
      L1:=Length(Points);
      BlockWrite(f,L1,SizeOf(L1)); // запись числа точек
      if L1>0 then
      for j:=0 to L1-1 do
        BlockWrite(f,Points[j],SizeOf(Points[j]));

      if MyFont<>nil then begin
        w:=FontStyleToByte(MyFont);
        BlockWrite(f,w,SizeOf(w));
        WriteStr(MyFont.Name);
        BlockWrite(f,MyFont.Color,SizeOf(MyFont.Color));
        w:=MyFont.Size;
        BlockWrite(f,w,SizeOf(w));
      end
      else begin
        w:=$FFFF;
        BlockWrite(f,w,SizeOf(w));
      end;
    end;
  end;
  CloseFile(f);
end;
```

Запись полей типа длинная строка (`string`) реализует процедура `WriteStr` (листинг 8.27), в которой сначала записывается длина строки `L`, а затем `L` символов строки.

Листинг 8.27. Процедура записи строки

```
  procedure WriteStr(s: string);
  var L: word;
  begin
    L:=Length(s);
    BlockWrite(f,L,SizeOf(L));              // Length Name
    BlockWrite(f,s[1],L);                   // Name
  end;
```

Для преобразования множественного свойства шрифта `Font.Style` в целочисленное значение типа `byte` используется функция `FontStyleToByte` (листинг 8.28).

Листинг 8.28. Преобразование стиля шрифта в целочисленное значение типа *byte*

```
function FontStyleToByte(Font: TFont): byte;
begin
  Result:=0;
  with Font do begin
    if fsBold in Style then Result:=Result or (1 shl 0);
```

```
    if fsItalic in Style then Result:=Result or (1 shl 1);
    if fsUnderline in Style then Result:=Result or (1 shl 2);
    if fsStrikeOut in Style then Result:=Result or (1 shl 3);
    if fpDefault=Pitch then Result:=Result or (1 shl 4);
    if fpVariable=Pitch then Result:=Result or (1 shl 5);
    if fpFixed=Pitch then Result:=Result or (1 shl 6);
  end;
end;
```

Чтение проекта из нетипизированного файла происходит строго в обратном порядке: перед чтением объектов или точек сначала считывается их число и назначается длина динамических массивов (листинг 8.29).

Листинг 8.29. Чтение проекта из нетипизированного файла

```
procedure OpenVRT(FileName: string);
var
  i,L,j,L1,Col: integer;
  w: word;
begin
  AssignFile(f,FileName); Reset(f,1);
  with Page do begin
    BlockRead(f,NumPointNode,SizeOf(NumPointNode));
    BlockRead(f,NumPoint,SizeOf(NumPoint));
    BlockRead(f,NumObj,SizeOf(NumObj));
    BlockRead(f,NumShape,SizeOf(NumShape));
    BlockRead(f,xMin,SizeOf(xMin));
    BlockRead(f,yMin,SizeOf(yMin));
    BlockRead(f,xMax,SizeOf(xMax));
    BlockRead(f,yMax,SizeOf(yMax));
    BlockRead(f,PageWidth,SizeOf(PageWidth));
    BlockRead(f,PageHeight,SizeOf(PageHeight));
    BlockRead(f,L,SizeOf(L)); // чтение числа объектов
    SetLength(Obj,L);         // назначение длины массива
    if L>0 then
    for i:=0 to L-1 do
    with Obj[i] do begin
      BlockRead(f,Select,SizeOf(Select));
      BlockRead(f,PColor,SizeOf(PColor));
      BlockRead(f,BColor,SizeOf(BColor));
      BlockRead(f,PWidth,SizeOf(PWidth));
      BlockRead(f,PStyle,SizeOf(PStyle));
      BlockRead(f,BStyle,SizeOf(BStyle));
      BlockRead(f,Pc,SizeOf(Pc));
      BlockRead(f,Rt,SizeOf(Rt));
      BlockRead(f,a,SizeOf(a));
```

```
      BlockRead(f,Shape,SizeOf(Shape));
      BlockRead(f,typeObj,SizeOf(typeObj));
      Text:=ReadStr;
      BlockRead(f,L1,SizeOf(L1));
      SetLength(Points,L1);
      if L1>0 then
      for j:=0 to L1-1 do
        BlockRead(f,Points[j],SizeOf(Points[j]));

      BlockRead(f,w,SizeOf(w));
      if w<>$FFFF then begin
        MyFont:=TFont.Create;
        ByteToFontStyle(MyFont, w);
        MyFont.Name:=ReadStr;
        BlockRead(f,Col,SizeOf(Col));
        MyFont.Color:=Col;
        BlockRead(f,w,SizeOf(w));
        MyFont.Size:=w;
      end;
    end;
  end;
  CloseFile(f);
end;
```

Для чтения строк предназначена функция `ReadStr` (листинг 8.30), которая сначала считывает длину строки, назначает длину строки, а затем считывает символы строки.

Листинг 8.30. Чтение строки

```
  function ReadStr: string;
  var L: word;
  begin
    BlockRead(f,L,SizeOf(L));              // Length Name
    SetLength(Result,L);
    BlockRead(f,Result[1],L);              // Name
  end;
```

Процедура `ByteToFontStyle` преобразует значение типа `byte` в множественное значение стиля шрифта (листинг 8.31).

Листинг 8.31. Преобразование целочисленного значения в стиль шрифта

```
procedure ByteToFontStyle(var Font: TFont; FStyle: byte);
begin
  with Font do begin
    Style:=[];
```

```
    if FStyle and (1 shl 0)<>0 then Style:=Style + [fsBold];
    if FStyle and (1 shl 1)<>0 then Style:=Style + [fsItalic];
    if FStyle and (1 shl 2)<>0 then Style:=Style + [fsUnderline];
    if FStyle and (1 shl 3)<>0 then Style:=Style + [fsStrikeOut];
    if FStyle and (1 shl 4)<>0 then Pitch:=fpDefault;
    if FStyle and (1 shl 5)<>0 then Pitch:=fpVariable;
    if FStyle and (1 shl 6)<>0 then Pitch:=fpFixed;
  end;
end;
```

Хранение информации в нетипизированном файле имеет минимум один недостаток: при изменении структуры данных проекта нет преемственности по предыдущим версиям. Но небольшое изменение позволит этот недостаток устранить: необходимо в файл записывать и читать номер версии структуры, а этот номер использовать при чтении данных.

Мы не стремились в этом проекте реализовать все функции векторной графики. Более того, для ускорения рисования можно было бы использовать, например, API-функции или библиотеки DirectX/OpenGL. Задача была значительно скромнее: показать проблемы, возникающие при реализации векторного редактора.

Полностью проект приведен на прилагаемом к книге компакт-диске в папке **Примеры | Глава 8 | MyCorel**.

Глава 9

Графики функций

9.1. График функции одной переменной

В *главе 1* мы уже рассматривали задачу построения графика функции, заданной параметрически. Вернемся к задаче построения графика функции одной переменной для более подробного обсуждения.

Для построения графика функции $y = f(x)$ на бумаге выбирается прямоугольник с размерами $(X_1, X_2) \times (Y_1, Y_2)$. Основная проблема, возникающая при построении графиков функций на экране монитора, обусловлена необходимостью масштабировать прямоугольник на бумаге $(X_1, X_2) \times (Y_1, Y_2)$ в прямоугольник на экране $(I_1, I_2) \times (J_1, J_2)$ (рис. 9.1).

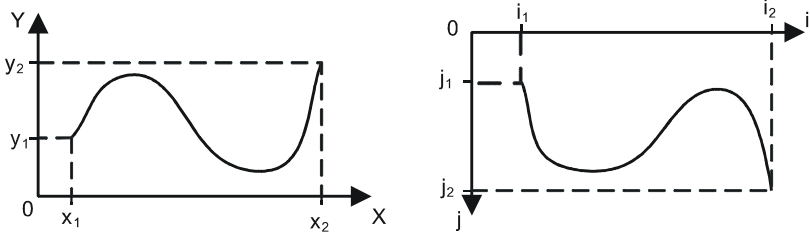


Рис. 9.1. Масштабирование графиков

Масштабирование по осям OX и OY реализуется при помощи линейных зависимостей (9.1), (9.2), графики которых представлены на рис. 9.2.

$$\frac{x - X_1}{X_2 - X_1} = \frac{i - I_1}{I_2 - I_1}, \tag{9.1}$$

$$\frac{y - Y_1}{Y_2 - Y_1} = \frac{j - J_2}{J_1 - J_2}. \tag{9.2}$$

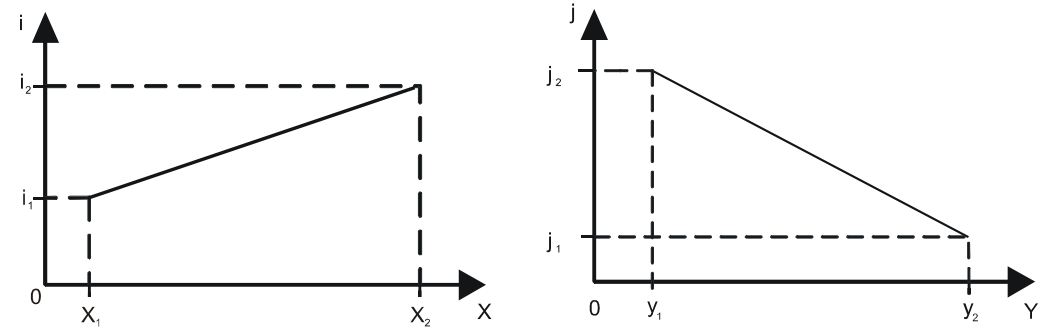


Рис. 9.2. Функции масштабирования по осям OX и OY

При вычислении вертикальных экранных координат необходимо учитывать с помощью изменения знака, что нумерация строк на экране идет сверху вниз. В листинге 9.1 приводятся две функции масштабирования, часто используемые в графических программах (листинг 9.1).

Листинг 9.1. Функции масштабирования

```
function II(x: real): integer;
// функция масштабирования по оси OX
begin
  II:=I1+Trunc((x-X1)*(I2-I1)/(X2-X1));
end;

function JJ(y: real): integer;
// функция масштабирования по оси OY
begin
  JJ:=J2+Trunc((y-Y1)*(J1-J2)/(Y2-Y1));
end;
```

Рассматриваемый в данном разделе проект "График функции одной переменной" состоит из двух форм: `FormMain` и `FormTools` (рис. 9.3) и модуля `Lbr`. Первая (`FormMain`) предназначена для прорисовки графика. На инструментальной форме (`FormTools`) помещены компоненты, позволяющие менять размеры окна "на бумаге" (текстовые поля для задания координат **x1**, **x2**, **y1**, **y2**), рисовать или не рисовать координатную сетку (группа **Сетка**, переключатели **Dot**, **Line**), строить график функции в обычной или логарифмической системе координат (флажок **LnY**).

При запуске проекта вызывается метод `FormCreate` (листинг 9.2), в котором определяются размеры окна на экране.

Листинг 9.2. Активизация проекта

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
```

```
  // Задание окна на экране
  I1:=0; J1:=0; I2:=ClientWidth; J2:=ClientHeight;
end;
```

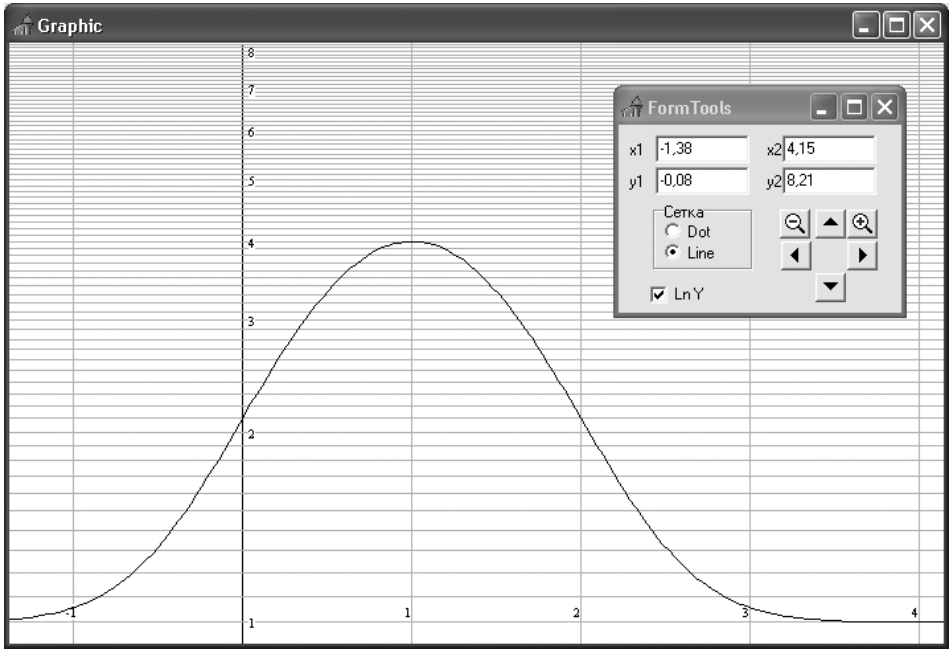


Рис. 9.3. Формы проекта "Построение графика функции одной переменной"

График функции строится с помощью отрезков прямых линий, соединяющих узловые точки. Для того чтобы график выглядел плавным, необходимо использовать около 60 точек по горизонтали. Для рисования вызывается метод `DrawGraphic`, выводящий график (листинг 9.3).

Листинг 9.3. Процедура построения графика

```
procedure TFormMain.DrawGraphic;
var i: integer;
    k: byte;
    h,x,y: real;
begin
  // вычисление шага по оси OX
  h:=(x2-x1)/n;
  with Canvas do begin
    RectAngle(-1,-1,Width,Height);
    // Построение осей координат
    k:=FormTools.RadioGroup1.ItemIndex;
```

```
    OX(k,x1,x2);
    if not FormTools.ChLnY.Checked
      then OY(k,y1,y2)
      else OYLn(k,y1,y2);
    // Построение графика функции отрезками
    x:=x1; y:=Func(x,FormTools.ChLnY.Checked);
    MoveTo(II(x),JJ(y));
    Pen.Style:=psSolid;
    Pen.Color:=clBlack;
    for i:=1 to n do begin
      x:=x+h; y:=Func(x,FormTools.ChLnY.Checked);
      LineTo(II(x),JJ(y));
    end;
  end;
end;
```

Ось OX строится достаточно просто (листинг 9.4).

Листинг 9.4. Построение оси OX

```
procedure TFormMain.OX(fl: byte; x1,x2: real);
// ось OX
var i,j,k1,k2: integer;
    Digits: byte;
    h1: double;
begin
  h1:=HH(x1,x2); // вычисление шага по оси OX
  k1:=Trunc(x1/h1)-1; k2:=Trunc(x2/h1);
  Digits:=GetDigits(x1,x2);

  with Canvas do begin
    Pen.Color:=clBlack;
    Pen.Width:=1;
    Pen.Style:=psSolid;
    MoveTo(II(x1),JJ(0)); LineTo(II(x2),JJ(0));
    Font.Name:='Times New Roman';
    Font.Size:=7;
    for i:=k1 to k2 do begin
      Pen.Color:=clBlack;
      MoveTo(II(i*h1),JJ(0)-7); LineTo(II(i*h1),JJ(0)+7);
      for j:=1 to 9 do begin // мелкие деления оси
        MoveTo(II(i*h1+j*h1/10),JJ(0)-3);
        LineTo(II(i*h1+j*h1/10),JJ(0)+3);
      end;
      if fl=1 then begin     // вертикальные линии сетки
        Pen.Color:=clSilver;
```

```
        MoveTo(II(i*h1),J1); LineTo(II(i*h1),J2);
      end;
      TextOut(II(i*h1)-5,JJ(0)-c2-10,
              FloatToStrF(h1*i,ffFixed,8,Digits));
    end;
  end;
end;
```

На осях системы координат выводятся деления. Проблема заключается в том, что при разных значениях `X1` и `X2` число делений, определяющее шаг, с которым выводятся деления, должно быть примерно одинаковым. Функция `HH(a1,a2)` (листинг 9.5) позволяет вычислить шаг делений любого интервала `(a1,a2)`, при котором число делений всегда будет в диапазоне 5..10.

Листинг 9.5. Вычисление шага

```
function HH(a1,a2: real): real;
begin
  Result:=1;
  while Abs(a2-a1)/Result<1 do Result:=Result/10.0;
  while Abs(a2-a1)/Result>=10 do Result:=Result*10.0;
  if Abs(a2-a1)/Result<2.0 then Result:=Result/5.0;
  if Abs(a2-a1)/Result<5.0 then Result:=Result/2.0;
end;
```

Другая полезная функция `GetDigits`, используемая при построении осей, позволяет определить число знаков мантиссы в зависимости от длины интервала `dx` (листинг 9.6).

Листинг 9.6. Установка формата

```
function GetDigits(dx: real): byte;
begin
  if dx>=5 then Result:=0
  else if dx>=0.5 then Result:=1
  else if dx>=0.05 then Result:=2
  else if dx>=0.005 then Result:=3
  else if dx>=0.0005 then Result:=4 else Result:=5;
end;
```

Вычисленное значение `Digits` используется при печати числа, определяющего деление:

```
TextOut(II(i*h1)-5,JJ(0)-c2-10,
  FloatToStrF(h1*i,ffFixed,8,Digits));
```

Построение оси `OY` в обычной системе координат лишь немного отличается от построения оси `OX`. В логарифмической системе координат отличий больше и они связаны с тем, что необходимо вычислять логарифм от значения `Y` (листинг 9.7).

Листинг 9.7. Построение оси OY в логарифмической системе координат

```
procedure TFormMain.OYLn(fl: byte; y1,y2: real);
// ось OY Ln
var i,j,k1,k2: integer;
    Digits: byte;
    h1: double;
begin
  h1:=HH(0,y2);
  k1:=Trunc(y1/h1)-1; k2:=Trunc(y2/h1);
  Digits:=GetDigits(Abs(y2-y1));

  with Canvas do begin
    Pen.Width:=1;
    Pen.Style:=psSolid;
    MoveTo(II(0),JJ(y1)); LineTo(II(0),JJ(Ln(y2)));
    Font.Name:='Times New Roman';
    Font.Size:=7;
    for i:=0 to k2 do begin
      Pen.Color:=clBlack;
      if i<>0 then begin
        MoveTo(II(0),JJ(Ln(i*h1))); LineTo(II(0)+7,JJ(Ln(i*h1)));
        TextOut(II(0)+5,JJ(Ln(i*h1))-5,
                FloatToStrF(h1*i,ffFixed,8,Digits));
      end;
      for j:=1 to 9 do begin
        MoveTo(II(0)-3,JJ(Ln(i*h1+j*h1/10)));
        LineTo(II(0)+3,JJ(Ln(i*h1+j*h1/10)));
      end;
      if fl=1 then begin
        Pen.Color:=clSilver;
        if i<>0 then begin
          MoveTo(I1,JJ(Ln(i*h1))); LineTo(I2,JJ(Ln(i*h1)));
        end;
      end;
    end;
  end;
end;
```

На инструментальной форме `FormTools` (рис. 9.3) размещены шесть компонентов типа `TSpeedButton`. Все они имеют разные значения свойства `Tag` (`4,5,6,7,8,9`) и вызывают один обработчик события `onSpeedButton6Click` (листинг 9.8). Два компонента (`Tag (4,5)`) отвечают за изменение масштаба изображения (увеличение или уменьшение), остальные — за смещение графика по осям координат.

Листинг 9.8. Изменение размеров окна

```
procedure TFormTools.SpeedButton6Click(Sender: TObject);
begin
  case (Sender as TSpeedButton).Tag of
    6: begin
         Y1:=Y1-0.1; Y2:=Y2-0.1;
       end;
    7: begin
         X1:=X1-0.1; X2:=X2-0.1;
       end;
    8: begin
         Y1:=Y1+0.1; Y2:=Y2+0.1;
       end;
    9: begin
         X1:=X1+0.1; X2:=X2+0.1;
       end;
    5: begin
         X1:=0.8*X1; X2:=0.8*X2;
         Y1:=0.8*Y1; Y2:=0.8*Y2;
       end;
    4: begin
         X1:=1.2*X1; X2:=1.2*X2;
         Y1:=1.2*Y1; Y2:=1.2*Y2;
       end;
  end;
  SetEdit;
  FormMain.DrawGraphic;
end;
```

В зависимости от значения свойства `Tag` происходят те или иные изменения размеров окна `X1`, `Y1`, `X2`, `Y2`, т. е. меняется масштаб рисования. Затем вызывается процедура перерисовки графика `DrawGraphic`.

При щелчке мыши на компоненте `ChLnY`, отвечающем за обычные или логарифмические координаты, вызывается процедура перерисовки графика (листинг 9.9).

Листинг 9.9. Процедура изменения вида графика

```
procedure TFormTools.ChLnYClick(Sender: TObject);
begin
  FormMain.DrawGraphic;
end;
```

Проект полностью приведен на компакт-диске в папке **Примеры | Глава 9 | 2D**.

9.2. График функции двух переменных

Рассмотрим задачу построения поверхности, описываемой функцией $z = f(x, y)$, в параллелепипеде $(a_1, a_2) \times (b_1, b_2) \times (c_1, c_2)$. Прежде всего, приведем формулы суперпозиции двух поворотов системы координат относительно точки (x_0, y_0, z_0). Сдвиг в точку (x_0, y_0, z_0), поворот относительно оси OZ на угол α с последующим поворотом вокруг оси OX на угол β описывается соотношениями (9.3) и (9.4) [63], соответственно:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix} \times \begin{pmatrix} x - x0 \\ y - y0 \\ z - z0 \end{pmatrix}, \tag{9.3}$$

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\beta & -\sin\beta \\ 0 & \sin\beta & \cos\beta \end{pmatrix} \times \begin{pmatrix} x \\ y \\ z \end{pmatrix}. \tag{9.4}$$

После перемножения матриц в соотношениях (9.3) и (9.4), получаем:

$$\begin{aligned} x &= (x - x_0) \times \cos\alpha - (y - y_0) \times \sin\alpha; \\ y &= ((x - x_0) \times \sin\alpha + (y - y_0) \times \cos\alpha) \times \cos\alpha - (z - z_0) \times \sin\alpha; \\ z &= ((x - x_0) \times \sin\alpha + (y - y_0) \times \cos\alpha) \times \sin\alpha + (z - z_0) \times \cos\alpha. \end{aligned} \tag{9.5}$$

На рис. 9.4 изображены две системы координат — исходная (координаты X, Y, Z) и преобразованная (координаты $\overline{X}, \overline{Y}, \overline{Z}$).

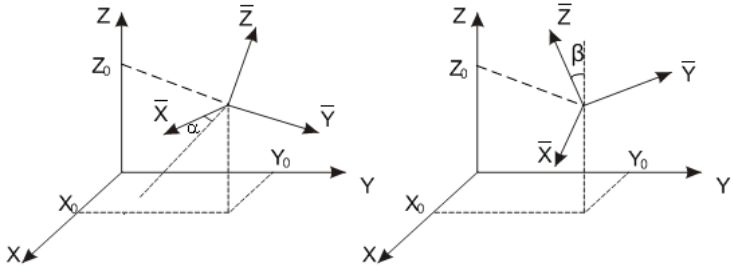


Рис. 9.4. Сдвиг и поворот системы координат

Будем считать, что оси OX и OY лежат в плоскости монитора, ось OZ перпендикулярна экрану. Будем сжимать координаты точки (x, y) к единственной точке схода (0, 0) по формулам (9.6) и (9.7):

$$x_n = \frac{x}{z/a + 1}; \tag{9.6}$$

$$y_n = \frac{y}{z/a+1}. \tag{9.7}$$

Параметр a подбирается экспериментально.

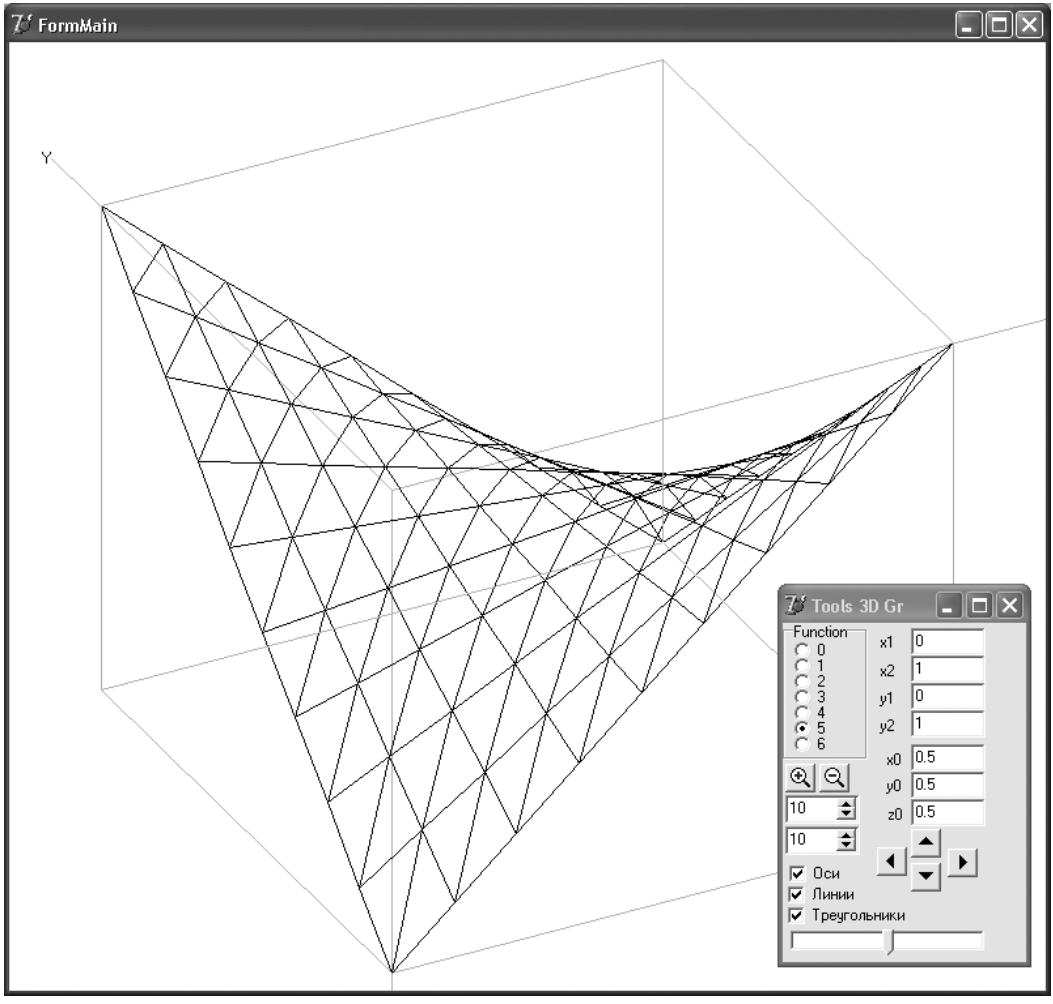


Рис. 9.5. Поверхность в единичном кубе

Для того чтобы не "утяжелять" листинги, мы в тексте сознательно не используем возможности объектно-ориентированного программирования, т. е. не создаем классы. Но на примере задачи построения графика функции двух переменных отступим от этого правила, и в модуле `Lib3dGraph.pas` создадим класс `TGraph3D`. Проект "Построение графика функции двух переменных" состоит из двух форм (рис. 9.5): **FormMain** — для визуального отображения построенного графика и инструмен-

тальной формы **Tools 3D Gr**, на которой содержится ряд управляющих инструментов. Это:

- группа переключателей **Function**, где пользователь может выбрать для построения одну из семи возможных поверхностей;
- кнопки, увеличивающие (+) и уменьшающие (-) масштаб изображения графика;
- флажки **Оси**, **Линии**, **Треугольники** устанавливающие или отменяющие режим видимости на графике соответственно для координатных осей, линий, ограничивающих единичный куб и треугольников, формирующих поверхность функции двух переменных;
- поля для задания числа разбиений по осям X и Y;
- поля для ручного задания размера окна.

Программа строит на экране монитора проекцию одной из семи поверхностей в единичном кубе (рис. 9.5), например, поверхности $z = 1 + 2 \times x \times y - x - y$ при $(x, y) \in [0,1] \times [0,1]$.

Класс `TGraph3D` наследуется от `TObject` и содержит следующие поля, методы и свойства (листинг 9.10).

Листинг 9.10. Класс *TGraph3D*

```
TGraph3D=class
  Xmin,Xmax,Ymin,Ymax: real; // размеры окна на бумаге
  FX1,FX2,FY1,FY2: real;     // диапазоны изменения X и Y
  Fxc,Fyc,Fzc: real;         // точка поворота СК
  n     : integer;           // число разбиений по оси X
  m     : integer;           // число разбиений по оси Y
  Alf,Bet: real;             // углы поворота
  FA    : real;              // коэффициент перспективы
  Bitmap: TBitmap;           // поле с канвой для рисования в памяти
  V     : array of array of TVector; // узловые точки
  FFunc : TF;                // указатель на функцию f(x,y)
  FVisibleAxes: boolean;     // видимость осей
  FtypeFace: boolean;        // true - Line, false - Face
  FTypeFaceTRIANGLES: boolean; // TRIANGLES, QUADS
  Sides: array of TSides;    // массив граней
  procedure SetVisibleAxes(V: boolean);
  procedure SetF(VF: TF);
  procedure SetX1(V: real);
  procedure SetX2(V: real);
  procedure SetY1(V: real);
  procedure SetY2(V: real);
  procedure SetN (V: integer);
  procedure SetM (V: integer);
```

```
    procedure SetXc(V: real);
    procedure SetYc(V: real);
    procedure SetZc(V: real);
    procedure SetA (V: real);
    function  GetNx: integer;
    function  GetNy: integer;
  public
    constructor Create(VW,VH: integer; VF: TF);
    destructor Destroy; override;
    function  IJ(P: TVector): TPoint;
    procedure SetV;
    procedure DrawCub;
    procedure DrawAxes;
    procedure Draw3D;
    property  VisibleAxes: boolean read FVisibleAxes
      write SetVisibleAxes;
    property  Func: TF read FFunc write SetF;
    property  X1: real read Fx1 write SetX1;
    property  X2: real read Fx2 write SetX2;
    property  Y1: real read Fy1 write SetY1;
    property  Y2: real read Fy2 write SetY2;
    property  Nx: integer read GetNx write SetN;
    property  Ny: integer read GetNy write SetM;
    property  Xc: real read Fxc write SetXc;
    property  Yc: real read Fyc write SetYc;
    property  Zc: real read Fzc write SetZc;
    property  A : real read FA write SetA;
    procedure SetWH(VW,VH: integer);
    procedure SetAngle(x,y: integer);
    function Rotate(P: TVector): TVector;
    procedure Sort;
    function VectorToPoint(P: TVector): TPoint;
  end;
```

Используемые в классе типы описаны в листинге 9.11.

Листинг 9.11. Вспомогательные типы

```
TF=function(x,y: real): real;
TVector=record
  x,y,z: real;
end;
TSides=record
  P: array[0..3] of TVector;
  N: real;
  Zmin: real;
end;
```

Задачи преобразования координат по формулам (9.5)–(9.7), одноточечного проецирования и масштабирования при переходе к экранным координатам были описаны ранее в *главе 6* и в рассматриваемом проекте реализованы в методе `IJ(x,y,z: real; var i,j: integer)` (листинг 9.12).

Листинг 9.12. Метод преобразования систем координат и масштабирования

```
function TGraph3D.IJ(P: TVector): TPoint;
var Xn,Yn,Zn: real;
begin
  Xn:=(P.x-Fxc)*cos(alf)-(P.y-Fyc)*sin(alf);
  Yn:=((P.x-Fxc)*sin(alf)+(P.y-Fyc)*cos(alf))*cos(Bet)-
    (P.z-Fzc)*sin(Bet);
  Zn:=((P.x-Fxc)*sin(alf)+(P.y-Fyc)*cos(alf))*sin(Bet)+
    (P.z-Fzc)*cos(Bet);
  Xn:=Xn/(Zn*A+1); Yn:=Yn/(Zn*A+1);
  Result.x:=Trunc(Bitmap.Width*(Xn-Xmin)/(Xmax-Xmin));
  Result.y:=Trunc(Bitmap.Height*(Yn-Ymax)/(Ymin-Ymax))
end;
```

Метод `SetV` использует указатель на пользовательскую функцию `FFunc` для задания массива узловых точек `V` (листинг 9.13).

Листинг 9.13. Метод вычисления элементов массива *V*

```
procedure TGraph3D.SetV;
var
  i,j: integer;
  hx,hy: real;
begin
  SetLength(V,n,m);
  hx:=(Fx2-Fx1)/(m-1); hy:=(Fy2-Fy1)/(n-1);
  for i:=0 to n-1 do
  for j:=0 to m-1 do begin
    V[i,j].x:=Fx1+j*hx;
    V[i,j].y:=Fy1+i*hy;
    V[i,j].z:=FFunc(Fx1+j*hx,Fy1+i*hy);
  end;
end;
```

Область определения функции $z = f(x, y)$ задана точками $x_i = x_1 + h_x \times i$, $y_j = y_1 + h_y \times j$, где $i = 0,1,...,n$, $j = 0,1,...,m$ и разбивается на $n \times m$ прямоугольников, над каждым из которых поверхность интерполируется ребрами четырехугольника [97, 98]. Построение спроецированных на экран четырехугольников реализуется с помощью методов канвы `LineTo` или `Polygon()` в процедуре `Draw3D` (листинг 9.14).

Листинг 9.14. Рисование поверхности в трехмерном пространстве

```
procedure TGraph3D.Draw3D;
var
  i,j,L,q: integer;
  w: array[0..3] of TPoint;
begin
  with Bitmap.Canvas do begin
    Brush.Style:=bsSolid;
    Brush.Color:=clWhite;
    FillRect(Rect(0,0,Bitmap.Width,Bitmap.Height));

    if FtypeFace then begin // рисование линиями
      if FVisibleAxes then DrawAxes;  // рисуем оси координат
      Pen.Color:=clBlack;
      Pen.Width:=1;
      Pen.Style:=psSolid;
      for i:=0 to n-2 do
      for j:=0 to m-2 do begin
        P0:=IJ(V[i,j]);
        P1:=IJ(V[i,j+1]);
        MoveTo(P0.X,P0.Y); LineTo(P1.X,P1.Y);
        P1:=IJ(V[i+1,j]);
        MoveTo(P0.X,P0.Y); LineTo(P1.X,P1.Y);
        if FTypeFaceTRIANGLES then begin // диагональ
          P1:=IJ(V[i+1,j+1]);
          MoveTo(P0.X,P0.Y); LineTo(P1.X,P1.Y);
        end;
      end;
      for j:=0 to m-2 do begin // последние ряды
        P0:=IJ(V[n-1,j]);
        P1:=IJ(V[n-1,j+1]);
        MoveTo(P0.X,P0.Y); LineTo(P1.X,P1.Y);
      end;
      for i:=0 to n-2 do begin
        P0:=IJ(V[i,m-1]);
        P1:=IJ(V[i+1,m-1]);
        MoveTo(P0.X,P0.Y); LineTo(P1.X,P1.Y);
      end;
    end
    else begin  // рисование четырехугольниками
      L:=n*m; q:=-1;
      SetLength(Sides,L); // собираем грани
      for j:=0 to m-2 do
      for i:=0 to n-2 do begin
        Inc(q);
```

```
        with Sides[q] do begin
          P[0]:=Rotate(V[i+0,j+0]);
          P[1]:=Rotate(V[i+1,j+0]);
          P[2]:=Rotate(V[i+1,j+1]);
          P[3]:=Rotate(V[i+0,j+1]);
          N:=Norm(P[0],P[1],P[2]);
          Zmin:=Z_Min(Sides[q]);
        end;
      end;
      Sort; // сортируем по Zmin
      for i:=0 to L-1 do // рисуем четырехугольники
      with Sides[i] do begin
        for j:=0 to 3 do w[j]:=VectorToPoint(P[j]);
        Brush.Color:=Trunc(255*Abs(N))*$010100;
        Pen.Color:= Brush.Color;
        Polygon(w)
      end;
    end;
    DrawCub; // рисование единичного куба
  end;
end;
```

В процедуре `Draw3D` после очистки канвы рисуются оси повернутой системы координат и методом `Polygon` выводятся фрагменты поверхности. Поверхность рисуется над единичным квадратом, стороны которого разбиты, соответственно, на `n` и `m` частей. Каждый фрагмент представляет собой четырехугольник, вершинами которого являются значения функции над узлами сетки в повернутой системе координат.

При создании класса вызывается конструктор `TGraph3D.Create` (листинг 9.15).

Листинг 9.15. Инициализация данных

```
constructor TGraph3D.Create(VW,VH: integer; VF: TF);
begin
  Xmin := -2; Xmax :=  2; Ymin := -2; Ymax :=  2;
  FX1  :=  0; FX2  :=  1; FY1  :=  0; FY2  :=  1;
  Fxc  := 0.5; Fyc  := 0.5; Fzc  := 0.5;
  n    := 10; m    := 10;
  Alf  := 4.31; Bet  := 4.92;
  FFunc:= VF;
  FA   := 0;
  FVisibleAxes:=true; FtypeFace:=true; FTypeFaceTRIANGLES:=true;
  Bitmap:=TBitmap.Create;
  Bitmap.Width:=VW; Bitmap.Height:=VH;
  SetV;
end;
```

Изменение многих свойств класса `TGraph3D` должно приводить к повторному вычислению элементов массива `V`. Поэтому в процедурах, отвечающих за изменение этих свойств, предусмотрен вызов метода `SetV`. Так, например, для свойства `FX1` метод изменения выглядит так (листинг 9.16).

Листинг 9.16. Метод изменения свойства *Graph3D.FX1*

```
procedure TGraph3D.SetX1(V: real);
begin
  if Fx1<>V then begin
    Fx1:=V; SetV;
  end;
end;
```

При создании главной формы проекта "Построение графика функции двух переменных" создадим экземпляр класса `Graph3D: TGraph3D`, передавая ширину, высоту формы и указатель на одну из пользовательских функций (листинг 9.17).

Листинг 9.17. Создание экземпляра класса *TGraph3D*

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
  drawing:=false;
  Graph3D:=TGraph3D.Create(Width,Height,F(fl_function));
end;
```

Рисование на форме проходит в два этапа: сначала рисуем на внутреннем `Bitmap` класса, а затем копирум этот `Bitmap` на канву формы или любого другого компонента (листинг 9.18).

Листинг 9.18. Рисование на форме

```
procedure TFormMain.MyDraw;
begin
  Graph3D.Draw3D;
  Canvas.Draw(0,0,Graph3D.Bitmap);
end;
```

Перемещение указателя мыши при нажатой левой кнопке по форме `TFormMain` приводит к изменению углов `Alf` и `Bet` (`Alf` — это угол между линией, соединяющей точку положения указателя мыши и центр формы, а угол `Bet` — расстояние между точкой указателя положения мыши и центром экрана). Поворот системы координат реализован в трех обработчиках событий: `FormMouseDown`, `FormMouseMove`, `FormMouseUp`. Нажатие кнопки мыши вызывает процедуру `FormMouseDown`, в которой поднимается флаг перемещения `drawing:=true` (листинг 9.19).

Листинг 9.19. Начало поворота графика

```
procedure TFormMain.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  drawing:=true;
end;
```

При перемещении указателя мыши по форме вызывается обработчик события `FormMouseMove` (листинг 9.20).

Листинг 9.20. Поворот графика при перемещении указателя мыши

```
procedure TFormMain.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X,Y: Integer);
begin
  if drawing then begin
    Graph3D.SetAngle(x-Width div 2,y-Height div 2);
    MyDraw;
  end;
end;
```

При перемещении указателя мыши углы `Alf` и `Bet` меняются по закону

```
a:=x-Width div 2; b:=y-Height div 2;
Alf:=arctan(b/a);
Bet:=Sqrt(Sqr(a/10)+Sqr(b/10));
```

Этот алгоритм реализован в методе `TGraph3D.SetAngle` (листинг 9.21).

Листинг 9.21. Изменение углов системы координат при перемещении указателя мыши

```
procedure TGraph3D.SetAngle(x,y: integer);
begin
  Alf:=ArcTan2(y,x);
  Bet:=Sqrt(Sqr(x/10)+Sqr(y/10));
end;
```

Завершается поворот графика опусканием флага перемещения `drawing:=false` (листинг 9.22).

Листинг 9.22. Завершение поворота графика

```
procedure TFormMain.FormMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  drawing:=false;
end;
```

При изменении свойств класса, например, свойства `Graph3D.nX`, с помощью компонентов инструментальной формы, во многих случаях предусмотрено повторное изменение элементов массива `V`. Поэтому достаточно просто вызвать процедуру `TFormTools3DGr.SpinEdit1Change` (листинг 9.23).

Листинг 9.23. Изменение свойств с помощью инструментальной формы

```
procedure TFormTools3DGr.SpinEdit1Change(Sender: TObject);
begin
  if SpinEdit1.Text<>'' then begin
    Graph3D.nX:=SpinEdit1.Value;
    FormMain.MyDraw;
  end;
end;
```

Остальные методы класса и проект "Построение графика функции двух переменных" полностью приведены на компакт-диске в папке **Примеры | Глава 9 | 3D графика**.

9.3. Интерполяция функций

На практике часто встречается следующая задача: по заданным на плоскости значениям (x_i, y_i), $i = 0,1,...,n$ построить функцию, либо точно проходящую через эти точки, либо проходящую как можно ближе к этим точкам (рис. 9.6). В данном разделе рассмотрены четыре способа решения этой задачи: интерполяционный многочлен Лагранжа, метод наименьших квадратов, интерполяция кубическими сплайнами и интерполяция кривыми Безье [109, 121].

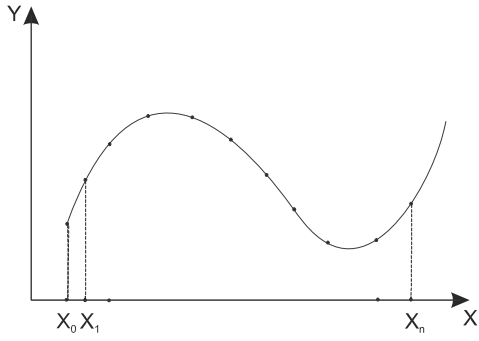


Рис. 9.6. Задача интерполяции

9.3.1. Проект "Построение интерполяционных кривых"

Проект "Построение интерполяционных кривых" предназначен для рисования линий по множеству заданных на плоскости точек с использованием четырех интерполяционных методов: интерполяционного многочлена Лагранжа (рис. 9.8), метода наименьших квадратов (рис. 9.9), интерполяции кубическими сплайнами (рис. 9.10) и интерполяции кривыми Безье (рис. 9.12).

В проекте используются две формы (рис. 9.7) — главная форма `FormMain` (с внешним имением **Интерполяция**) и форма для настройки параметров `FormTools` (с внешним имением **`Tools`**) и модуль `Lbr.pas`.

В проекте реализованы следующие функции:

- выбор одного из методов интерполяции;
- изменение масштаба (`Zoom`);
- перемещение точек;
- добавление узловых точек;
- удаление узловых точек.

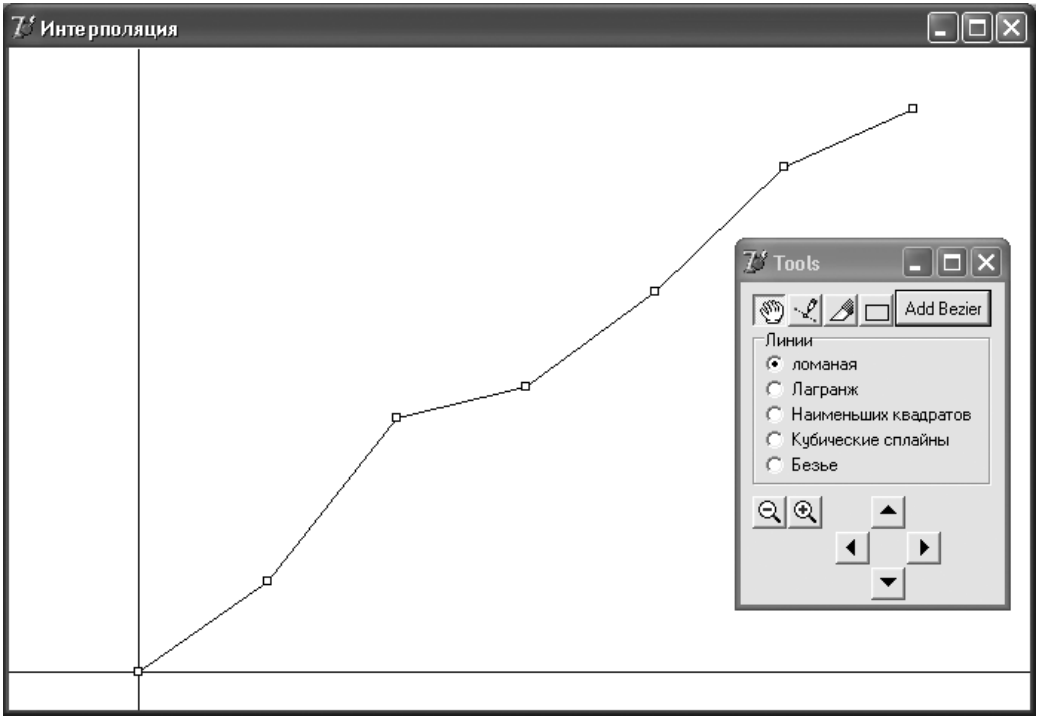


Рис. 9.7. Формы проекта "Построение интерполяционных кривых"

Для описания узловых точек в проекте используется следующая структура:

```
  TVector=record
    x,y: real;
  end;
  TAVector=array of TVector;
```

Сами узловые точки хранятся в динамическом массиве `Point: TAVector`.

При запуске проекта создается массив из 7 элементов. Элементы этого массива заполняются случайными данными (листинг 9.24).

Листинг 9.24. Заполнение элементов массива случайными данными

```
procedure TFormMain.FormCreate(Sender: TObject);
var i: byte;
begin
  I1:=0; J1:=0; I2:=Width; J2:=Height;
  SetLength(Point,7);
  for i:=0 to 6 do begin
    Point[i].x:=i;
    Point[i].y:=Random+i;
  end;
end;
```

Метод рисования определяется свойством `ItemIndex` компонента `RgMethod` на форме `FormTools`, которое может принимать следующие значения:

- `0` — построение ломаной линии
- `1` — использование интерполяционного многочлена Лагранжа
- `2` — метод наименьших квадратов
- `3` — интерполяция кубическими сплайнами
- `4` — интерполяция кривыми Безье

Рисование линий осуществляется процедурой `Draw` на канве формы `FormMain`. В листинге 9.25 приведен основной фрагмент кода процедуры `Draw` с реализацией рисования ломаной линии (рис. 9.7).

Листинг 9.25. Процедура рисования ломаной линии

```
procedure TFormMain.Draw;
var
  i,L,j,k: integer;
  h,u,v: real;
begin
  with Canvas do begin
    Brush.Color:=clWhite;
    FillRect(Rect(0,0,I2,J2));
```

```
    // Оси
    MoveTo(II(0),JJ(Y1)); LineTo(II(0),JJ(Y2));
    MoveTo(II(X1),JJ(0)); LineTo(II(x2),JJ(0));
    L:=Length(Point);
    if L>0 then begin
      Pen.Color:=clBlack;
      k:=FormTools.RgMethod.ItemIndex; // № метода
      case k of
        0: begin  // ломаные
             MoveTo(II(Point[0].x),JJ(Point[0].y));
             for i:=1 to L-1 do
               LineTo(II(Point[i].x),JJ(Point[i].y));
           end;
        …          // остальные методы
      end;
      // узловые точки
      for i:=0 to L-1 do
        RectAngle(II(Point[i].x)-3,JJ(Point[i].y)-3,
                  II(Point[i].x)+3,JJ(Point[i].y)+3);
    end;
  end;
end;
```

В проекте предусмотрена возможность перетаскивания любой точки указателем мыши. Поэтому, наряду с традиционными функциями масштабирования, которые обсуждались в *разд. 9.1* (листинг 9.26), используются функции обратного масштабирования (листинг 9.27).

Листинг 9.26. Функции прямого масштабирования

```
function II(x: real): integer;
begin
  II:=I1+Trunc((I2-I1)*(x-x1)/(x2-x1));
end;

function JJ(y: real): integer;
begin
  JJ:=J1+Trunc((J2-J1)*(y-y2)/(y1-y2));
end;
```

Листинг 9.27. Функции обратного масштабирования

```
function XX(i: integer): real;
begin
  XX:=x1+(i-I1)*(x2-x1)/(I2-I1);
end;
```

```
function YY(j: integer): real;
begin
  YY:=y1+(j-j2)*(y2-y1)/(J1-J2);
end;
```

В проекте созданы обработчики трех событий: `onMouseDown` (нажатие кнопки мыши), `onMouseMove` (перемещение указателя мыши), `onMouseUp` (завершение движения указателя мыши). На инструментальной форме `FormTools` имеются четыре кнопки, изменяющие значения переменной `fl_tools`. Эта переменная может принимать значения `0`, `1`, `2`, `3`, причем значению `0` соответствует перемещение точек, значению `1` — добавление точек, значению `2` — удаление точек, значению `3` — изменение окна просмотра на плоскости. При `fl_tools=0` с помощью функции `FindPoint` определяется номер `NumPoint` точки, ближайшей к точке (`X,Y`), и поднимается флаг, разрешающий перемещения — `drawing:= true` (листинг 9.28).

Листинг 9.28. Процедура обработки события нажатия кнопки мыши

```
procedure TFormMain.FormMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var L,i: integer;
begin
  case fl_tools of
    0: begin  // перемещение точек
         NumPoint:=FindPoint(XX(x),YY(y));
         drawing:=true;
       end;
    1: begin  // добавление точек
         L:=Length(Point); Inc(L); SetLength(Point,L);
         Point[L-1].x:=XX(x);
         Point[L-1].y:=YY(y);
         case FormTools.RgMethod.ItemIndex of
           2: Gauss(CountStep,Point,C2);
           3: SetSystem3D(a,b,c,d);
         end;
         Draw;
       end;
    2: begin  // удаление точек
         NumPoint:=FindPoint(XX(x),YY(y));
         L:=Length(Point);
         for i:=NumPoint+1 to L-1 do Point[i-1]:=Point[i];
         Dec(L); SetLength(Point,L);
         case FormTools.RgMethod.ItemIndex of
           2: Gauss(CountStep,Point,C2);
           3: SetSystem3D(a,b,c,d);
         end;
```

```
        Draw;
      end;
    3: begin  // изменение окна просмотра
        drawing:=true;
        x0:=x; y0:=y; xt:=x; yt:=y;
        with Canvas do
        DrawFocusRect(Rect(x0,y0,xt,yt));
      end;
  end;
end;
```

Для добавления точек при `fl_tools=1` увеличивается на `1` размер динамического массива точек `Point` и добавленный элемент заполняется преобразованными координатами точки, в области которой мы щелкнули указателем мыши.

Для удаления точек при `fl_tools=2` также с помощью функции `FindPoint` определяется номер `NumPoint` точки, ближайшей к точке (`X,Y`), и эта точка удаляется из массива `Point`. Отметим, что удаление точки требует пересчета коэффициентов для метода наименьших квадратов и кубических сплайнов.

Для изменения окна просмотра при `fl_tools=3` поднимается флаг, разрешающий перемещения — `drawing := true` и первый раз рисуется фокусный прямоугольник.

Во время перемещения указателя мыши (листинг 9.29) работает процедура `FormMouseMove`, в которой при перемещении точки (`fl_tools=0`) изменяются координаты выделенной точки, обновляются коээфициенты для метода наименьших квадратов и кубических сплайнов и методом `Draw` перерисовывается весь график. При изменении окна просмотра стирается фокусный прямоугольник, меняются координаты его подвижного угла и фокусный прямоугольник рисуется в новом месте.

Листинг 9.29. Процедура обработки перемещения указателя мыши

```
procedure TFormMain.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  if drawing then
  case fl_tools of
    0: begin
        Point[NumPoint].x:=XX(x);
        Point[NumPoint].y:=YY(y);
        case FormTools.RgMethod.ItemIndex of
          2: Gauss(CountStep,Point,C2);
          3: SetSystem3D(a,b,c,d);
        end;
        Draw;
      end;
```

```
    3: with Canvas do begin
         DrawFocusRect(Rect(x0,y0,xt,yt));
         xt:=x; yt:=y;
         DrawFocusRect(Rect(x0,y0,xt,yt));
       end;
  end;
end;
```

Заканчивается движение указателя мыши вызовом процедуры `FormMouseUp`, в которой опускается флаг перемещения `drawing := false`. Действие предусмотрено только при изменении окна просмотра: меняются параметры окна просмотра (`X1`, `Y1`, `X2`, `Y2`) (листинг 9.30).

Листинг 9.30. Процедура обработки события завершения движения указателя мыши

```
procedure TFormMain.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  drawing:=false;
  case fl_tools of
    3: with Canvas do begin
         DrawFocusRect(Rect(x0,y0,xt,yt));
         X1:=XX(x0); X2:=XX(xt);
         Y1:=YY(yt); Y2:=YY(y0);
         FormMain.Draw;
       end;
  end;
end;
```

9.3.2. Интерполяционный многочлен Лагранжа

Продолжая рассмотрение проекта "Построение интерполяционных кривых", остановимся на построении интерполяционного многочлена Лагранжа (рис. 9.8). График функции, определенной интерполяционным многочленом Лагранжа, проходит через все точки (x_i, y_i):

$$L(x) = \sum_{i=0}^{n} y_i \prod_{j=0, j<>i}^{n} \frac{x - x_j}{x_i - x_j}. \qquad (9.8)$$

Этот метод чрезвычайно прост в использовании, но обладает существенным недостатком: отклонение значений функции от ожидаемых может быть достаточно большим.

Для вычисления значений многочлена Лагранжа по уравнению (9.8) можно воспользоваться функцией `Lagr` (листинг 9.31).

Листинг 9.31. Функция Лагранжа

```
function Lagr(xt: Real): real;
var
  i,j,L: Byte;
  P: Real;
begin
  Result:=0; L:=Length(Point);
  for i:=0 to L-1 do
    begin
      P:=1;
      for j:=0 to L-1 do
        if i<>j then P:=P*(xt-Point[j].x)/(Point[i].x-Point[j].x);
      Result:=Result+Point[i].y*P;
    end;
end;
```

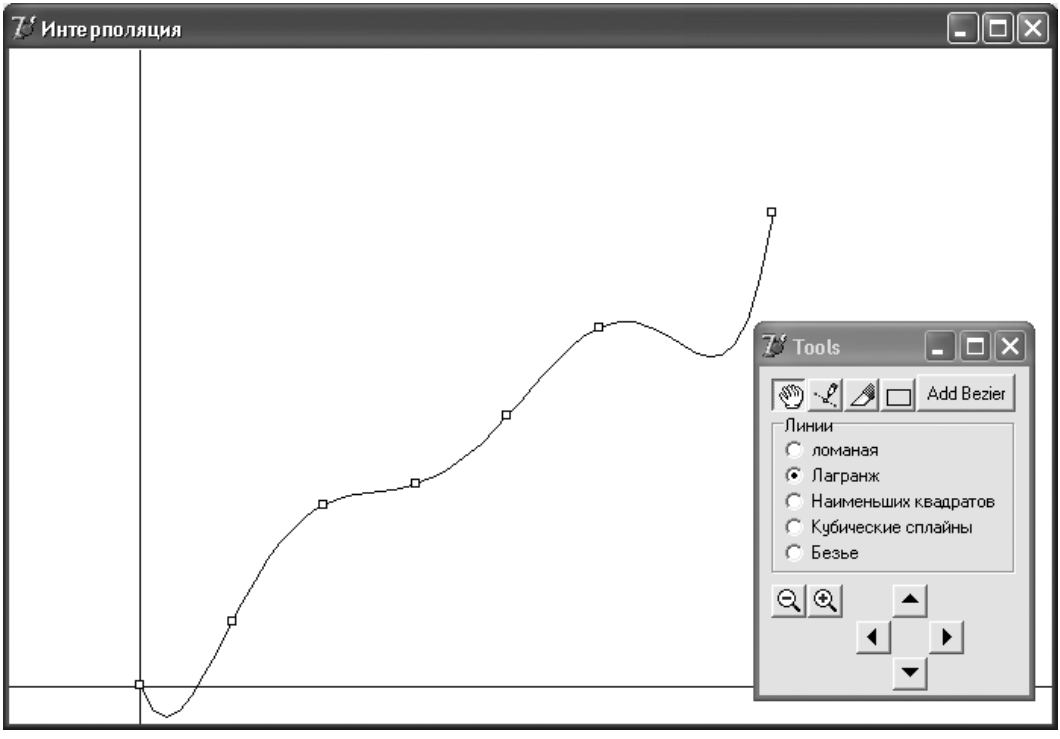


Рис. 9.8. Интерполяция с помощью многочлена Лагранжа

Приведем фрагмент текста программы, использующей функцию `Lagr` для построения графика по множеству точек (x_i, y_i), $i = 0,1,...,n$ (листинг 9.32).

Листинг 9.32. Построение графика многочлена Лагранжа

```
h:=SetH;
u:=Point[0].x; v:=Lagr(u);
MoveTo(II(u),JJ(v));
for i:=1 to m do begin
  u:=u+h; v:=Lagr(u);
  LineTo(II(u),JJ(v));
end;
```

Функция `SetH` опредяет шаг изменения переменной `x` при разбиении интервала на `m` частей (листинг 9.33).

Листинг 9.33. Определение шага изменения по *x*

```
function SetH: real;
var
  i,L: word;
  Xmin,Xmax: real;
begin
  L:=Length(Point);
  Xmin:=Point[0].x; Xmax:=Point[0].x;
  for i:=1 to L-1 do begin
    if Xmin>Point[i].x then Xmin:=Point[i].x;
    if Xmax<Point[i].x then Xmax:=Point[i].x;
  end;
  Result:=(Xmax-Xmin)/m;
end;
```

Существенно, что первая и последняя точка не обязательно удовлетворяют условиям $x_{первая} = \min(x_i)$, $x_{последняя} = \max(x_i)$. Поэтому формула $H = (x_{последняя} - x_{первая})/m$ неприемлема.

9.3.3. Метод наименьших квадратов

Методом наименьших квадратов можно искать интерполяционную функцию $F(x)$ среди многочленов степени m:

$$F(x) \in \{S(x) = \sum_{j=0}^{m} a_i x^j\} \tag{9.9}$$

так, чтобы сумма квадратов отклонений была минимальна:

$$D = \sum_{i=0}^{n} (\sum_{j=0}^{m} a_j x^j{}_i - y_i)^2 \to \min. \tag{9.10}$$

При этом функция $F(x)$, вообще говоря, не проходит через точки (x_i, y_i).

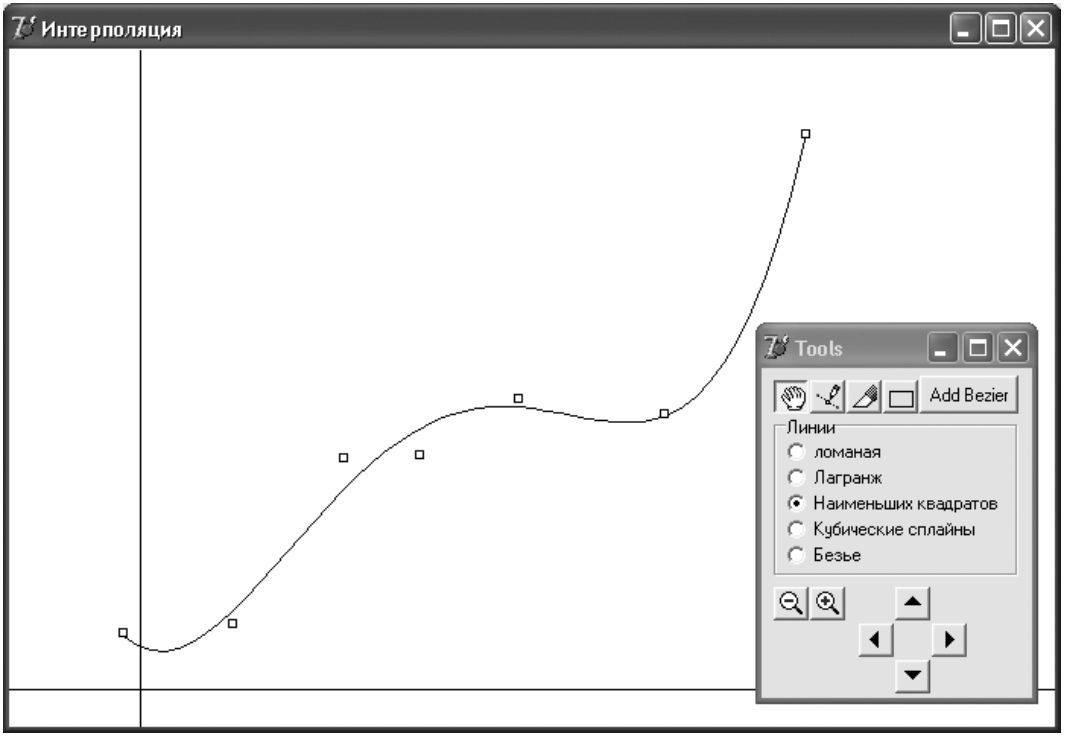


Рис. 9.9. Интерполяция методом наименьших квадратов

Для определения коэффициентов $a_j, \quad j = 0, 1, ..., m$ запишем необходимые условия экстремума:

$$\frac{\partial D}{\partial a_k} = 0, \quad k = 0, 1, ..., n \tag{9.11}$$

или

$$\sum_{i=0}^{n} x_i{}^k \left(\sum_{j=0}^{m} a_j x^j{}_i - y_i \right)^2 = 0, \quad k = 1, ..., m. \tag{9.12}$$

После преобразований получаем следующую систему линейных уравнений:

$$\sum_{j=0}^{m} C_{kj} a_j = B_k, \quad k = 0, ..., m, \tag{9.13}$$

где

$$C_{kj} = \sum_{i=0}^{n} x_i{}^{k+j}, \quad B_k = \sum_{i=0}^{n} x_i{}^k y_i. \tag{9.14}$$

Для вычисления элементов матрицы линейных уравнений по формулам (9.14) и решения этой системы методом Гаусса можно воспользоваться процедурой `Gauss`. Необходимо отметить, что система линейных уравнений для метода наименьших квадратов, как правило, хорошо обусловлена и не имеет нулевых элементов, что позволяет максимально упростить алгоритм решения системы.

В процедуре `Gauss` параметры имеют следующий смысл: `n+1` — число точек; `m` — степень многочлена; `x`, `y` — массивы узловых точек; `c` — массив коэффициентов многочлена (листинг 9.34).

Листинг 9.34. Процедура *Gauss*

```
procedure Gauss(m: Byte; Point: TAVector; var c: TArray);
var A: TMatr;
    b: TArray;
    i,j,k,l,n: word;
    P: Real;
begin
  SetLength(A,m+1);
  for i:=0 to m do SetLength(A[i],m+1);
  SetLength(C,m+1);
  SetLength(B,m+1);
  n:=Length(Point);
  // коэффициенты матрицы
  for j:=0 to m do
    for k:=j to m do begin
      A[j,k]:=0;
      for i:=0 to n-1 do begin
        p:=1; for l:=1 to j+k do p:=p*Point[i].x;
        A[j,k]:=A[j,k]+P;
      end;
      A[k,j]:=A[j,k];
    end;
  // свободные члены
  for k:=0 to m do begin
    B[k]:=0;
    for i:=0 to n-1 do begin
      P:=Point[i].y; for l:=1 to k do P:=P*Point[i].x;
      B[k]:=B[k]+P;
    end;
  end;
  // решение системы: прямой ход
  for i:=0 to m-1 do
    for j:=i+1 to m do begin
      for k:=i+1 to m do
```

```
        A[k,j]:=A[k,j]-A[i,j]*A[k,i]/A[i,i];
      B[j]:=B[j]-B[i]*A[i,j]/A[i,i];
    end;
  // решение системы: обратный ход
  for j:=m downto 0 do begin
    c[j]:=B[j];
    for k:=j+1 to m do c[j]:=c[j]-A[k,j]*c[k];
    c[j]:=c[j]/A[j,j];
  end;
end; // Gauss
```

Ниже приводится текст программы, в которой для массива (x_i, y_i), $i = 0,1,...,n$ методом наименьших квадратов определяются коэффициенты многочлена степени m и строится график этого многочлена (листинг 9.35).

Листинг 9.35. Фрагмент кода рисования линии методом наименьших квадратов

```
h:=SetH;
u:=Point[0].x; v:=F2(u);
MoveTo(II(u),JJ(v));
for i:=1 to m do begin
  u:=u+h; v:=F2(u);
  LineTo(II(u),JJ(v));
end;
```

Функция `F2` вычисляет значение полинома (9.12) по известным коээфициентам `C2` (листинг 9.36).

Листин 9.36. Вычисление значения полинома

```
function F2(x: Real): Real;
// Вычисление значения многочлена
var
  i,L: word;
begin
  L:=Length(C2);
  Result:=0;
  for i:=L-1 downto 0 do Result:=Result*x+C2[i];
end;
```

9.3.4. Кубические сплайны

Сплайн-функция интерполирует значения (x_i, y_i) набором функций, каждая из которых определена на интервале $[x_{i-1}, x_i]$:

- функция $S(x)$ проходит через все точки (x_i, y_i): $S(x_i) = y_i$, $i = 0,...,n$;

- на всем интервале $[x_0, x_n]$ функция $S(x)$ дважды непрерывно дифференцируема.

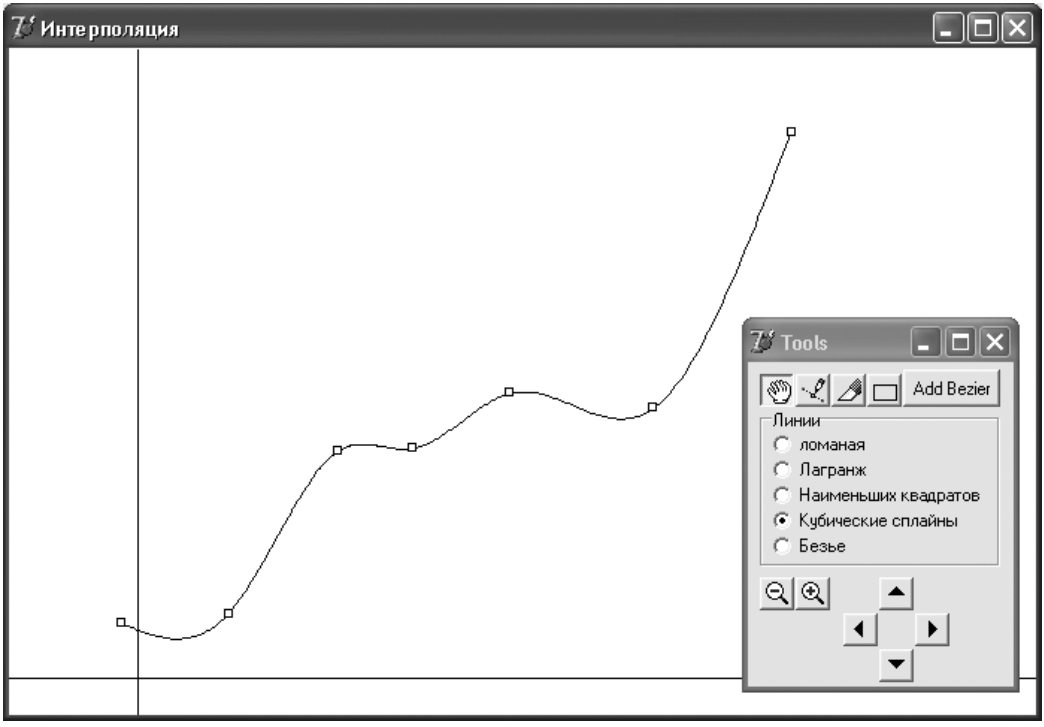


Рис. 9.10. Интерполяция кубическими сплайнами

Будем считать, что на i-ом интервале сплайн-функция определяется многочленом третьей степени

$$S_i(x) = a_i(x - x_i)^3 + b_i(x - x_i)^2 + c_i(x - x_i) + d_i, \quad i = 1,...n. \tag{9.15}$$

Необходимо определить $4 \times n$ коэффициентов a_i, b_i, c_i, d_i. Условия прохождения многочленов через узловые точки дают $2 \times n$ уравнений

$$S_i(x_i) = d_i = y_i, \tag{9.16}$$

$$S_i(x_{i-1}) = a_i D_i^3 + b_i D_i^2 + c_i D_i + y_i = y_{i-1}, \quad i = 1,...n, \quad D_i = x_{i-1} - x_i. \tag{9.17}$$

Условие гладкости первых производных во внутренних точках дает $n-1$ соотношений

$$S_i'(x_i) = S_{i+1}'(x_i), \tag{9.18}$$

$$c_i = 3a_{i+1}D_{i+1}^2 + 2b_{i+1}D_{i+1} + c_{i+1}, \quad i = 1,...,n-1. \tag{9.19}$$

Условие гладкости вторых производных дает еще $n-1$ соотношений

$$S_i''(x_i) = S_{i+1}''(x_i) \tag{9.20}$$

или

$$2b_i = 6a_{i+1}D_{i+1} + 2b_{i+1}, \quad i = 1,...n-1. \tag{9.21}$$

Два недостающих условия следуют из условия обращения в ноль вторых производных на границах интервала $S_1''(x_0) = 0$ и $S_n''(x_n) = 0$:

$$6a_1D_1 + 2b_1 = 0, \quad D_1 = x_0 - x_1, \tag{9.22}$$

$$b_n = 0. \tag{9.23}$$

Из соотношений (9.20) и (9.21) найдем a_i:

$$a_i = \begin{cases} (b_{i-1} - b_i)/3D_i, & i = 2,3,...,n \\ -b_1/3D_1, & i = 1. \end{cases} \tag{9.24}$$

Подставим a_i в соотношение (9.17)

$$(b_{i-1} - b_i)D_i^2/3 + b_iD_i^2 + c_iD_i + y_i = y_{i-1}, \quad i = 2,...,n, \tag{9.25}$$

$$2b_1D_1^2 + c_1D_1 = y_0 - y_{i-1}, \quad i = 1$$

и найдем c_i:

$$c_i = \begin{cases} (y_{i-1} - y_i)/D_i - D_ib_{i-1}/3 - 2D_ib_{i-1}/3, & i = 2,...,n \\ (y_0 - y_1)/D_1 - b_1D_1/3. \end{cases} \tag{9.26}$$

После исключения коэффициентов a_i и b_i из соотношения (9.18) и преобразования для коэффициентов b_i, получаем трехдиагональную систему линейных уравнений:

$$\frac{2\times(D_1 + D_2)\times b_1}{3} + \frac{b_2 \times D_2}{3} = \frac{-(y_1 - y_2)}{D_2} + \frac{y_0 - y_1}{D_1}; \tag{9.27}$$

$$\frac{b_{i-1}D_i}{3} + \frac{2(D_i + D_{i+1})b_i}{3} + \frac{b_{i+1}D_{i+1}}{3} = -\frac{y_i - y_{i+1}}{D_{i+1}} + \frac{y_{i-1} - y_i}{D_i}, \; i = 2,...,n-1;$$

$$b_n = 0.$$

Рассмотрим теперь решение трехдиагональной системы линейных уравнений:

$$\begin{pmatrix} b_1 & c_1 & 0 & ... & 0 \\ a_2 & b_2 & c_2 & ... & 0 \\ 0 & a_3 & b_3 & ... & 0 \\ ... & ... & ... & ... & ... \\ 0 & 0 & 0 & ... & b_n \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ ... \\ x_n \end{pmatrix} = \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ ... \\ d_n \end{pmatrix}.$$

Трехдиагональная система решается в два прохода: за первый проход элементы матрицы под диагональю преобразуются в 0, за второй, обратный ход, определяются x_i, $i = 0,1,...,n$. Для решения трехдиагональных систем можно воспользоваться процедурой `System_3D`, которая получает массивы `a,b,c,d: Vect=array[1..n] of Real` и возвращает решение через массив `d` (листинг 9.37).

Листинг 9.37. Процедура решения трехдиагональных систем

```
procedure System_3D(a,b,c: TArray; var d: TArray);
// решение трехдиагональных систем
var
  i,L: word;
begin
  L:=Length(a)-1;
  SetLength(d,L+1);
  // прямой ход
  for i:=2 to L-1 do begin
    b[i]:=b[i]-c[i-1]*a[i]/b[i-1];  d[i]:=d[i]-d[i-1]*a[i]/b[i-1];
  end;
  // обратный ход
  d[L-1]:=d[L-1]/b[L-1];
  for i:=L-2 downto 1 do d[i]:=(d[i]-d[i+1]*c[i])/b[i];
end;
```

Для определения коэффициентов многочленов (9.15) по формулам (9.27), (9.26), (9.24) и (9.18) можно воспользоваться процедурой `SetSystem3D`, которая использует массивы координат узловых точек `x,y:Vect` и возвращает коэффициенты многочленов третьей степени `a,b,c,d: Vect` (листинг 9.38).

Листинг 9.38. Процедура подготовки коэффициентов для системы

```
procedure SetSystem3D(var a,b,c,d: TArray);
var i,L: word;
    delt,                // x[i-1]-x[i]
    p,q,r,w: TArray;     // коэффициенты системы
begin
  L:=Length(Point);
  SetLength(delt,L);
  SetLength(w,L);
  SetLength(q,L);
  SetLength(p,L);
  SetLength(r,L);
  SetLength(a,L);
  SetLength(b,L);
  SetLength(c,L);
  SetLength(d,L);
```

```
  for i:=1 to L-1 do delt[i]:=Point[i-1].x-Point[i].x;
  // вычисление коэффициентов для системы уравнений
  for i:=1 to L-2 do begin
    w[i]:=-(Point[i].y-Point[i+1].y)/delt[i+1]+
           (Point[i-1].y-Point[i].y)/delt[i];
    p[i]:=delt[i]/3;
    q[i]:=2*(delt[i]+Delt[i+1])/3; r[i]:=delt[i+1]/3;
  end;
  // Решение трехдиагональной системы уравнений
  System_3D(p,q,r,w);
  for i:=1 to L-1 do b[i]:=w[i]; b[L-1]:=0;
  // Вычисление остальных коэффициентов многочлена
  c[1]:=(Point[0].y-Point[1].y)/delt[1]-2*b[1]*delt[1]/3;
  a[1]:=-b[1]/(3*delt[1]);  d[1]:=Point[1].y;
  for i:=2 to L-1 do begin
    d[i]:=Point[i].y;  a[i]:=(b[i-1]-b[i])/(3*delt[i]);
    c[i]:=(Point[i-1].y-Point[i].y)/delt[i]-
          delt[i]*b[i-1]/3-2*delt[i]*b[i]/3;
  end;
end;
```

В листинге 9.39 приводится фрагмент текста программы, в котором строится график функции, соответствующей многочлену (9.17).

Листинг 9.39. Процедура рисования графика функции

```
   for i:=1 to L-1 do begin
     h:=(Point[i].x-Point[i-1].x)/m;
     u:=Point[i-1].x; v:=F3(i,u);
     MoveTo(II(u),JJ(v));
     for j:=1 to m do begin
       u:=Point[i-1].x+j*h; v:=F3(i,u); LineTo(II(u),JJ(v));
     end;
   end;
```

Для вычисления значения многочлена на i-ом интервале используется функция `F3(i,u)` (листинг 9.40).

Листинг 9.40. Вычисление значения многочлена на *i*-ом интервале

```
function F3(i: Byte; u: Real): Real;
begin
  Result:=((a[i]*(u-Point[i].x)+b[i])*(u-Point[i].x)+
          c[i])*(u-Point[i].x)+d[i];
end;
```

9.3.5. Кривые Безье

Напомним, что в векторной параметрической форме уравнения, описывающие кривую Безье (рис. 9.11) по четырем точкам V_0, V_1, V_2, V_3, выглядят так:

$$X = V_0 \times t^3 + 3 \times V_1 \times t^2 \times (1-t) + 3 \times V_2 \times t \times (1-t)^2 + V_3 \times (1-t)^3. \quad (9.28)$$

В этом уравнении параметр $t \in [0,1]$. Векторы $V_1 - V_0$ и $V_2 - V_3$ касаются кривой в точках V_0 и V_3. Для кривой, состоящей из нескольких фрагментов, должно выполняться условие: количество точек равно $N \times 3 + 1$.

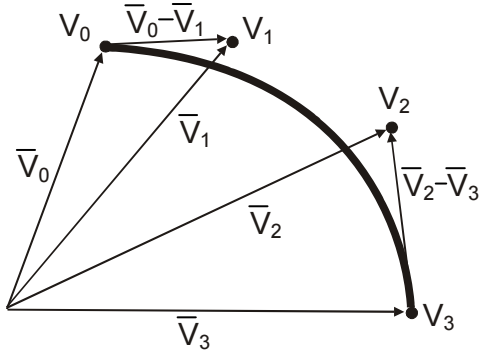


Рис. 9.11. Фрагмент кривой Безье

Для рисования кривой Безье можно воспользоваться уравнением (9.28), а можно использовать метод класса канвы `PolyBezier`, который требует введения массива координат точек типа `TPoint`. Мы будем пользоваться методом канвы, и в этом случае фрагмент кода, рисующий линию, выглядит так (листинг 9.41).

Листинг 9.41. Рисование кривой Безье

```
SetLength(w,L);
for i:=0 to L-1 do begin
  w[i].x:=II(Point[i].x);
  w[i].y:=jj(Point[i].y);
end;
PolyBezier(w);
// касательные линии
Pen.Color:=clSilver;
if L mod 3=1 then
for i:=0 to L div 3-1 do begin
  MoveTo(w[3*i].X,w[3*i].Y);
  LineTo(w[3*i+1].X,w[3*i+1].Y);
```

```
    MoveTo(w[3*(i+1)].X,w[3*(i+1)].Y);
    LineTo(w[3*(i+1)-1].X,w[3*(i+1)-1].Y);
  end;
```

На рис. 9.12 приведен пример кривой Безье, опирающейся на 7 точек.

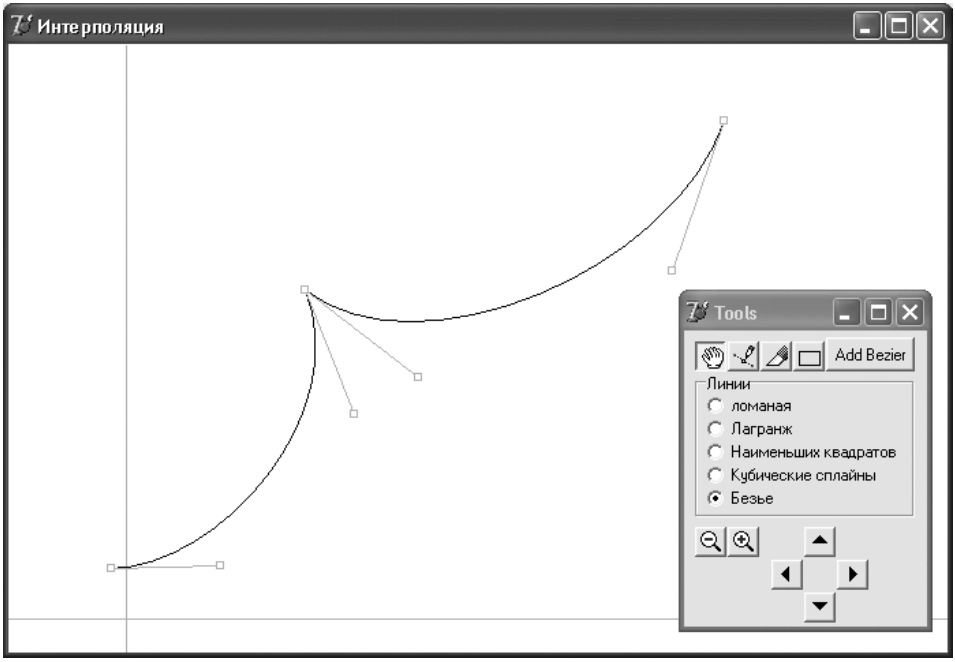


Рис. 9.12. Кривые Безье

Полностью проект "Построение интерполяционных кривых" приведен на компакт-диске в папке **Примеры | Глава 9 | Интерполяция**.

9.4. Параметрические кривые

В данном разделе мы рассмотрим простой проект "Рисование трансцендентных кривых" для рисования семи трансцендентных кривых, заданных параметрически, зависящих от шести параметров — `wx`, `wx0`, `wy`, `wy0`, `r0`, `r1`:

- фигуры Лиссажу

```
x:=cos(wx*t+wx0);
y:=sin(wy*t+wy0);
```

- спираль Архимеда

```
x:=(r0+r1*t)*cos(wx*t+wx0);
y:=(r0+r1*t)*sin(wy*t+wy0);
```

- спираль Бернулли

```
x:=r0*exp(r1*t)*cos(wx*t+wx0);
y:=r0*exp(r1*t)*sin(wy*t+wy0);
```

- параболическая спираль

```
x:=(r0+r1*sqrt(t))*cos(wx*t+wx0);
y:=(r0+r1*sqrt(t))*sin(wy*t+wy0);
```

- циклоида

```
x:=r0*exp(r1*t)*cos(wx*t+wx0);
y:=r0*exp(r1*t)*sin(wy*t+wy0);
```

- улитка Паскаля

```
x:=(r0*cos(t)+r1)*cos(wx*t+wx0);
y:=(r0*cos(t)+r1)*sin(wy*t+wy0);
```

- трисектрисса

```
x:=(r0*cos(t)-r1/cos(t))*cos(wx*t+wx0);
y:=(r0*cos(t)-r1/cos(t))*sin(wy*t+wy0);
```

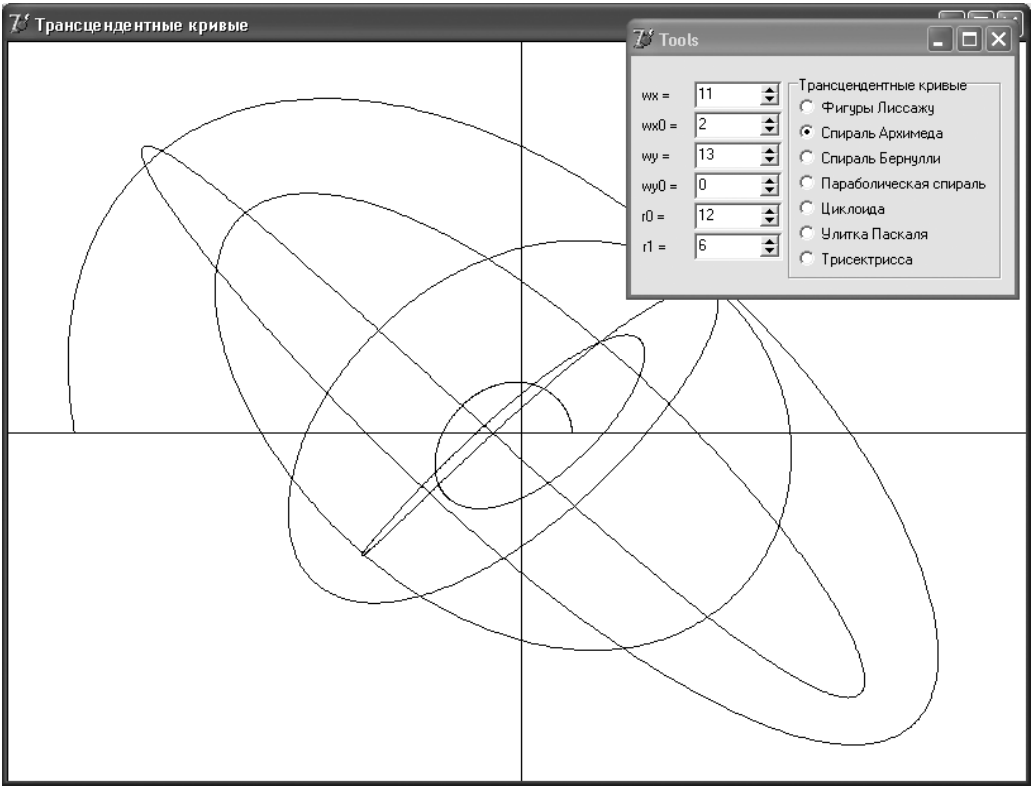


Рис. 9.13. Формы проекта "Рисование трансцендентных кривых"

Две формы этого проекта — FormMain (с внешним именем **Трансцендентные кривые**) и FormTools (с внешним именем **Tools**) представлены на рис. 9.13. Форма FormMain предназначена для прорисовки кривых, форма FormTools содержит группу радиокнопок для задания типа трансцендентных кривых (**Фигуры Лиссажу** и т. д.) и поля для задания параметров (**wx**, **wx0**, **wy** и т. д.) для каждой параметрически заданной кривой.

Данные определяются типом TPointReal

```
TPointReal=record
  X,Y: real;
end;
```

и переменными

```
x1,x2,y1,y2,h: Real;
i1,i2,j1,j2: Integer;
wx,wx0,wy,wy0: Real;
r0,r1: Real;
```

Перед рисованием кривых выбираются значения параметров из компонентов инструментальной формы SpinEdit (листинг 9.42).

Листинг 9.42. Рисование трансцендентных кривых

```
procedure TFormMain.DrawGraphic;
var
  i: Integer;
  t: Real;
  P: TPointReal;
  v: Byte;
begin
  // ввод значений параметров
  wx :=FormTools.SpinEdit1.Value/10;
  wx0:=FormTools.SpinEdit2.Value/10;
  wy :=FormTools.SpinEdit3.Value/10;
  wy0:=FormTools.SpinEdit4.Value/10;
  r0 :=Abs(FormTools.SpinEdit5.Value)/100;
  r1 :=Abs(FormTools.SpinEdit6.Value)/200;
  with Canvas do begin
    Brush.Color:=clWhite;
    Rectangle(0,0,Width,Height);
      // построение координатных осей
    Pen.Color:=clBlack;
    MoveTo(i1,j2 div 2);
    LineTo(i2,j2 div 2);
    MoveTo(i2 div 2,j1);
    LineTo(i2 div 2,j2);
      // построение графика функции
    Pen.Color:=clBlack;
```

```
    t:=0;
    v:=FormTools.RadioGroup1.ItemIndex;
    P:=F(v,t);
    MoveTo(II(P.x),JJ(P.y));
    for i:=1 to 5*n do begin
      t:=t+h; P:=F(v,t);
      LineTo(II(P.x),JJ(P.y));
    end;
  end;
end;
```

Полностью проект "Рисование трансцендентных кривых" приведен на компакт-диске в папке **Примеры | Глава 9 | Параметрические кривые**.

9.5. Построение графика функции с помощью интерпретатора

Рассмотрим следующую задачу: в виде строки задано выражение, т. е. конструкция из переменных, констант, бинарных операций и функций. Зная значения всех переменных, необходимо вычислить значение выражения. Бинарные операции обладают определенным приоритетом. Это означает, например, что операции умножения и деления должны выполняться раньше, чем операции сложения и вычитания. Для изменения порядка выполнения операций в выражении могут использоваться скобки. На практике такая задача возникает достаточно часто. В данном разделе рассмотрим проект "Интерпретатор". Цель этого проекта — создание приложения, которое будет рисовать график функции, введенной в виде строки (рис. 9.14).

9.5.1. Структура данных

Далее мы будем рассматривать даже несколько более сложную задачу: в виде строки задан оператор. Мы не будем реализовывать интерпретатор со всеми операторами типа `for`, `while`, `repeat`. Поэтому сразу оговорим ограничения.

1. В качестве оператора может быть или оператор присваивания

   ```
   Переменная := Выражение ,
   ```

   или условный оператор со структурой

   ```
   if Выражение then ОператорThen else ОператорElse .
   ```

   Для описания типа оператора введем перечисляемый тип:

   ```
   TOperatorType=(ot_expression, ot_ifthen);
   ```

   и рекурсивную структуру, определяющую операторы:

   ```
   TOperator = record  // оператор
     Node: PNode;      // указатель на выражение
   ```

```
    case TypeOperator: TOperatorType of
      ot_expression: (NumVar: Word);  // W := A
      ot_ifthen    : (OperThen, OperElse: POperator);
  end;
```

В этой структуре поле `Node` является указателем на выражение логического типа, поле `NumVar` — номер переменной, если структура описывает оператор присваивания, поля `OperThen` и `OperElse` — указатели на операторы, выполняющиеся при истинном или ложном значении выражения.

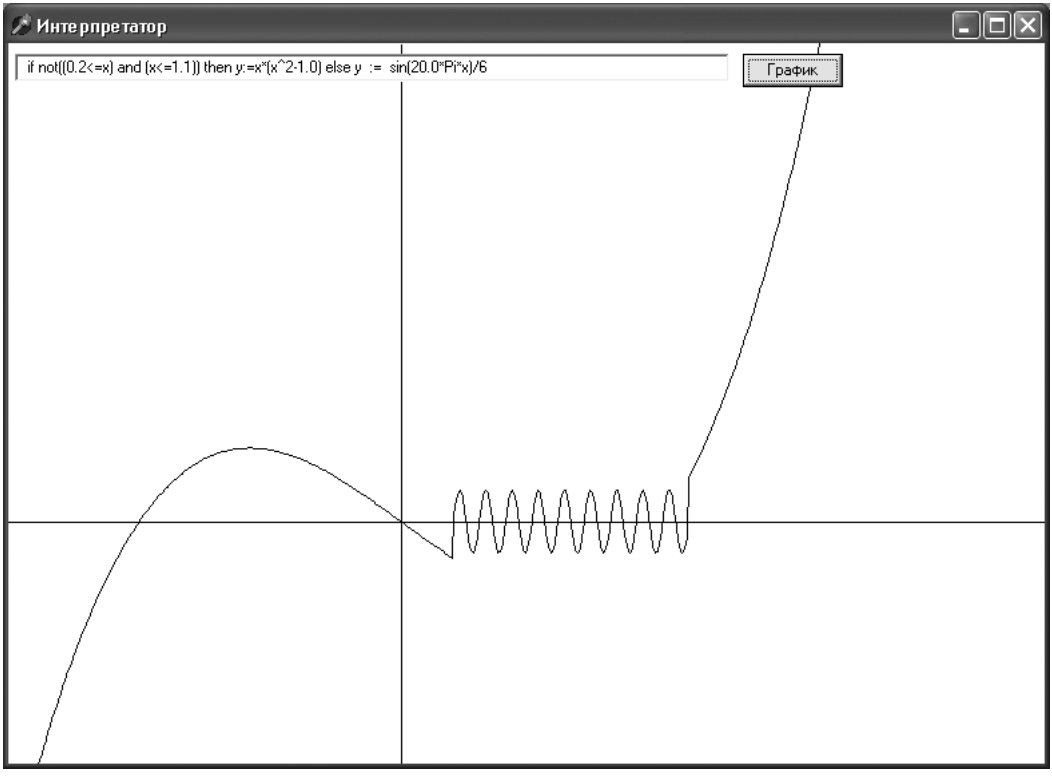


Рис. 9.14. График функции, заданной в виде строки в проекте "Интерпретатор"

2. Любая функция, входящая в выражение, может иметь только один параметр — значение.
3. Типы переменных, констант, функций и выражений могут быть или вещественными (`Double`), или байтовыми (`Byte`), или логическими (`Boolean`), или целыми (`Integer`). Для описания типов данных введем перечисляемый тип:

```
TVariableType = (vt_double, vt_byte, vt_boolean, vt_integer)
TVar = record  // значение
     case typ: TVariableType of
```

```
        vt_double : (d: Double);   // вещественное
        vt_byte   : (b: Byte);     // Byte
        vt_boolean: (l: Boolean);  // логическое
        vt_integer: (i: Integer);  // целое
end;
```

4. Любое выражение можно представить в виде дерева. Например, для выражения $x^2 + (A-1) \times \ln(B \times x - 3)$ бинарное дерево будет содержать 7 узлов. Будем считать, что каждый узел такого дерева может быть или бинарной (унарной) операцией, или функцией, или переменной, или константой. Для описания типа узлов введем перечисляемый тип:

   ```
   TNodeType=(nt_Oper, nt_Func, nt_Cons, nt_Variable);
   ```

   а для описания узлов дерева введем рекурсивную структуру с вариантной частью:

   ```
   PNode = ^TNode;
   TNode = record  // узел дерева
     V: TVar;      // значение узла
     case typNode : TNodeType of                // тип узла
       // операция
       nt_Oper    : (Label_Op: Char; Left, Right: PNode);
       nt_Func    : (NFunc: Byte; arg: PNode);  // функция
       nt_Cons    : (vConst: Real);             // константа
       nt_Variable: (Variable: TVariable);      // переменная
   end;
   ```

   В этой структуре поле `V: TVar` предназначено для хранения значения узла, которое мы будем вычислять.

   Если тип узла — операция `typNode = nt_Oper`, то поле `Label_Op: Char` указывает на бинарную операцию, а поля `Left, Right: PNode` указывают на левый и правый операнды.

   Если тип узла — функция `typNode = nt_Func`, то поле `NFunc: Byte` показывает на номер функции, а поле `arg: PNode` является указателем на вершину дерева, в котором сохраняется структура единственного параметра-значения функции.

   Если тип узла — константа `typNode = nt_Cons`, то поле `vConst: Real` содержит значение вещественной константы.

   Если тип узла — переменная `typNode = nt_Variable`, то поле `Variable: TVariable` содержит переменную.

5. Листьями дерева могут быть переменные, имена и значения которых мы будем хранить в динамическом массиве:

   ```
   AVariable : array of TVariable;
   ```

   Тип `TVariable` содержит два поля: имя переменной `Name` и значение переменной `V`:

   ```
   TVariable = record
     Name: string;  // имя переменной
   ```

```
    V   : TVar;    // значение
  end;
```

Именем переменной может быть строка, состоящая из символов множества `Chars`:

```
  Chars = ['a'..'z', 'A'..'Z', '_', '0'..'9'];
```

6. При анализе строки могут возникать ошибки, связанные с неверным введением выражения, например, отсутствие парной закрывающей скобки. В этом случае переменной `Error` будем присваивать ненулевой код ошибки и аварийно прерывать разбор строки вызовом оператора `Exit`. Список сообщений, выдаваемых в случае возникновения ошибки, собран в массиве `MsgEr`:

```
  MsgEr: array[1..12] of string = (
    'Не найден оператор присваивания :=',
    'Не найдено THEN',
    'Ожидается выражение',
    'Ожидается простое выражение',
    'Ожидается слагаемое',
    'Ожидается множитель',
    'Неизвестная переменная',
    'Нет скобки "("',
    'Нет скобки ")"',
    'Деление на 0',
    'Ln от отрицательного числа',
    'Корень отрицательного числа');
```

9.5.2. Анализ строки

Решение задачи проекта состоит из двух этапов: на первом этапе рекурсивной функцией `SetOperator()` создается структура данных `Operator`; на втором этапе при заданном значении `x` эта структура используется для вычисления переменной `y` с помощью процедуры `Run_Formula(Operator)`.

Создание структуры данных *Operator*

Первый этап состоит из нескольких шагов:

1. С помощью процедуры `SetOperator()` определить, является введенный оператор оператором присваивания или условным оператором `if`.
2. С помощью процедуры `PFormula()` проанализировать операции отношения <, >, =, <>, <=, >=.
3. С помощью процедуры `PEFormula()` проанализировать операции сложения и вычитания.

4. С помощью процедуры `PAdd()` проанализировать операции умножения и сложения.
5. С помощью процедуры `PMult()` проанализировать константы, скобки, функции и переменные.
6. С помощью процедуры `SetConst()` проанализировать константы.
7. Будем рассматривать строку, содержащую оператор, как очередь, из начала которой поочередно извлекаются символы. Пробелы, естественно, игнорируются, и для изъятия пробелов используется функция `PopSpace()` (листинг 9.43).

Листинг 9.43. Процедура пропускания пробелов

```
procedure PopSpace(var s: string);
begin
  while (s <> '') and (s[1] = ' ') do Delete(s, 1, 1);
end;
```

Для извлечения из строки `n` символов предназначена функция `Pop()` (листинг 9.44).

Листинг 9.44. Извлечение из строки *n* символов

```
function Pop(var s: string; n: Byte): string;
begin
  Result := Copy(s, 1, n);
  Delete(s, 1, n);  PopSpace(s);
end;
```

Анализ оператора

При анализе оператора мы будем придерживаться следующего порядка.

Шаг 1. При помощи рекурсивной функции `SetOperator()` (листинг 9.45), которая возвращает код ошибки, определим тип оператора и построим узел оператора. Рассмотрим работу этой функции более подробно.

1. В строку `st` собираем символы множества `Chars`. Если `st = 'if'`, то мы имеем дело с оператором `if`, иначе — с выражением.
2. Если `st = 'IF'`, то в строку `sExpres` собираются символы логического выражения между `if` и `then`.
3. Вызываем процедуру `PFormula(sExpres, Op.Node)` (листинг 13.46) для создания дерева логического выражения.
4. В строку `sThen` собираются символы оператора между `then` и `else` (если `else` в данном условном операторе присутствует). Этот оператор выполняется, если выражение — истинно.
5. Создается узел оператора `Op` типа `TypeOperator = ot_ifthen`.
6. Рекурсивно вызывается функция `SetOperator(sThen, Op.OperThen)`.

7. Если `sElse = ''`, то `Op.OperElse := nil`, иначе рекурсивно вызывается функция `SetOperator(sElse, Op.OperElse)`.
8. Если `st <> 'if'`, то определяем номер переменной, соответствующий этой строке.
9. Создается узел оператора `Op` типа `TypeOperator = ot_expression`.
10. Вызываем процедуру `PFormula(s, Op.Node)` для создания дерева, соответствующего выражению в правой части оператора присваивания.

Листинг 9.45 содержит полный текст функции `SetOperator()`.

Листинг 9.45. Процедура построения узла оператора

```
function SetOperator(var s: string; var Op: POperator): Integer;
var i: Integer;
    kThen, kElse: Byte;
    sExpres, sThen, sElse: string;
begin
  Error := 0;  Space(s); st := Push(s, 1);
  while (s <> '') and (s[1] in Chars) do begin
    st := st + s[1];  Delete(s, 1, 1);
  end;
  Space(s);
  if st = 'if' then begin
    kThen := Pos('then', s);  kElse := Pos('else', s);
    sExpres := Copy(s, 1, kThen - 1);
    if kThen = 0 then begin
      Error := 2;  Result := 2;  Exit;  // нет then!
    end;
    New(Op);  Op.TypeOperator := ot_ifthen;
    PFormula(sExpres, Op.Node);  Result := Error;
    if Error <> 0 then Exit;
    if kElse = 0 then begin  // нет else
      sThen := Copy(s, kThen + 4, Length(s) - kThen - 2);
      Error := SetOperator(sThen, Op.OperThen);
      Result := Error;  if Error <> 0 then Exit;
      Op.OperElse := nil;
    end
    else begin
      sThen := Copy(s, kThen + 4, kElse - kThen - 5);
      sElse := Copy(s, kElse + 4, Length(s) - kElse - 2);
      Error := SetOperator(sThen, Op.OperThen);
      Result := Error;  if Error <> 0 then Exit;
      Error := SetOperator(sElse, Op.OperElse);
      Result := Error;  if Error <> 0 then Exit;
    end;
  end
```

```
  else begin  // переменная
    Ok := False;  i := -1;
    while (i < LenD - 1) and not Ok do begin
      Inc(i);  Ok := st = UpperCase(AVariable[i].Name);
    end;
    if Ok then begin  // найдена переменная
      if (Length(s)<2) or (s[1]<>':')
          or (s[2]<>'=') then begin
        // нет присваивания!
        Result := Error;  Error := 1;  Exit;
      end;
      Push(s, 2);
      New(Op);  Op.TypeOperator := ot_expression;
      Op.NumVar := i;
      PFormula(s, Op.Node);
    end;
  end;
  Result := Error;
end;
```

Операции отношения

Шаг 2. При помощи рекурсивной процедуры `PFormula()` (листинг 9.46) построим узел левого операнда и попытаемся найти операции самого нижнего приоритета — операции отношения. Эта процедура работает по следующему алгоритму.

1. Обращаемся к процедуре `PEFormula(s, D)` (листинг 9.47) для построения левого операнда.
2. Пытаемся найти одну из операций отношения.
3. Если одна из них найдена, то с помощью процедуры `PEFormula(s, D)` строим узел правого операнда.

В листинге 9.46 приведена процедура `PFormula()`.

Листинг 9.46. Процедура построения узла операции отношения

```
procedure PFormula(var s: string; var D: PNode);
var D1, D2: PNode;
    chRelation: Byte;
// определение кода операции отношения
function SetChRelation: Byte;
begin
  if Length(s) < 2 then Result := 0
  else if (s[1] = '=') then Result := 1  // =
  else if (s[1]='<') and (s[2]<>'=') and (s[2] <> '>')  // <=
          then Result := 2
```

```
  else if (s[1]='>') and (s[2]<>'=') then Result := 3   // >=
  else if (s[1]='<') and (s[2]='>') then Result := 4    // <>
  else if (s[1]='<') and (s[2]='=') then Result := 5    // <=
  else if (s[1]='>') and (s[2]='=') then Result := 6    // >=
end;
begin  // PFormula
  PopSpace(s);
  if s <> '' then begin
    // левый операнд
    PEFormula(s, D);  if Error <> 0 then Exit;
    PopSpace(s);
    // определить операцию отношения
    chRelation := SetChRelation;
    if chRelation <> 0 then
    begin
      case chRelation of
        1..3: Pop(s, 1);
        4..6: Pop(s, 2);
      end;
      // правый операнд
      PEFormula(s, D2);  if Error <> 0 then Exit;
      D1 := D;  New(D);  D.TypNode := nt_Oper;
      D.Label_Op := Chr(chRelation + 48);
      D.Left := D1;  D.Right := D2;  D.V.typ := vt_boolean;
    end;
  end
  else begin
    D := nil;  Error := 3;  // Ожидается выражение
  end;
end;  // PFormula
```

Операции сложения и вычитания

Шаг 3. При помощи рекурсивной процедуры `PAdd` (листинг 9.47) построим узел операций сложения и вычитания. Эта процедура работает по следующему алгоритму.

1. Обращаемся к процедуре `PAdd()` для построения левого операнда.
2. Если первый символ в строке `'+'` или `'-'`, то строим узел унарной операции.
3. Пытаемся найти одну из операций сложения: `'+'`, `'-'`, `'or'`.
4. Если одна из них найдена, то с помощью процедуры `PAdd()` строим узел правого операнда.

Листинг 9.47. Процедура построения узлов операции '+', '–' или *'or'*

```
procedure PEFormula(var s: string; var D: PNode);
var sign: string; D1, D2: PNode;
```

```
begin
  PopSpace(s);
  if s <> '' then begin
    sign := '+';
    if (s[1] = '+') or (s[1] = '-') then sign := Pop(s, 1);
    // левый операнд
    PAdd(s, D);  if Error <> 0 then Exit; // !!!
    if sign = '-' then begin
      D1 := D;  New(D);  D.TypNode := nt_Func;  D.NFunc := 7;
      D.arg := D1;  D.V.typ := D1.V.typ;
    end;
    PopSpace(s);
    while ((Length(s) > 0) and ((s[1] = '+') or (s[1] = '-')))
      or ((Length(s)>1) and ((s[1]='o') or (s[2] = 'r'))) do
    begin
      sign := Pop(s,1);
      if s[1] = 'r' then Pop(s, 1);
      if s = '' then begin
        Error := 5;  Exit  // Ожидается слагаемое
      end;
      // добавляем узел слагаемого
      PAdd(s, D2);  if Error <> 0 then Exit;
      D1 := D;  New(D);
      D.TypNode := nt_Oper;  D.Label_Op := sign[1];
      D.Left := D1;  D.Right := D2;
      case sign[1] of
        '+', '-': D.V.typ := vt_double;
        'o'     : D.V.typ := vt_boolean;
      end;
    end;
  end
  else begin
    D := nil;  Error := 4;  // Ожидается простое выражение
  end;
end;  // PEFormula
```

Операции умножения и деления

Шаг 4. При помощи рекурсивной процедуры `PAdd()` (листинг 9.48) построим узел операций умножения и деления. Эта процедура работает по следующему алгоритму.

1. Обращаемся к процедуре `PMult()` для построения левого операнда.
2. Пытаемся найти одну из операций умножения: `'*'`, `'/'`, `'^'`, `'div'`, `'mod'`, `'and'`.
3. Если одна из операций найдена, то с помощью процедуры `PMult()` строим узел правого операнда.

Листинг 9.48. Процедура построения узлов операции '*', '/', '^', *'div'*, *'mod'*, *'and'*

```
procedure PAdd(var s: string; var D: PNode);
var sign: string;
    D1, D2: PNode;
begin
  PopSpace(s);
  if s <> '' then begin
    // левый множитель
    PMult(s, D);  if Error <> 0 then Exit;
    while
      ((Length(s) > 0) and
       ((s[1] = '*') or (s[1] = '/') or (s[1] = '^'))) or
      ((Length(s) > 3) and
       ((Copy(s,1,4)='div ') or (Copy(s,1,4)='mod ') or
        (Copy(s,1,4)='and ')) )) do
    begin
      sign := Pop(s, 1);
      if sign[1] in ['d', 'm', 'a'] then Delete(s, 1, 3);
      PopSpace(s);
      if s = '' then begin
        Error := 6;  Exit;  // Ожидается множитель
      end;
      // следующий множитель
      PMult(s, D2);  if Error <> 0 then Exit;
      D1 := D;  New(D);
      D.TypNode := nt_Oper;  D.Label_Op := sign[1];
      case sign[1] of
        '*', '/', '^': D.V.typ := vt_double;
        'd', 'm'     : D.V.typ := vt_integer;
        'a'          : D.V.typ := vt_boolean;
      end;
      D.Left := D1;  D.Right := D2;
    end;
  end
  else begin
    D := nil;  Error := 5;  // Ожидается слагаемое
  end;
end;  // PAdd
```

Анализ констант (скобки, функции, переменные)

Шаг 5. С помощью процедуры `PMult()` (листинг 9.49) анализируем константы, скобки, функции и переменные.

1. Если первый символ в строке — цифра, то с помощью процедуры `SetConst()` создаем узел константы.

2. Если первый символ в строке — скобка, то рекурсивно обращаемся к процедуре `PFormula()` для анализа формулы.
3. Если начало строки символы `'not'`, то строим узел унарной операции.
4. Собираем символы, принадлежащие множеству `Chars`, в строку `st` и пытаемся определить номер функции с именем `st`.
5. Если такая функция есть, то с помощью рекурсивного вызова процедуры `PFormula()` строим узел для единственного аргумента этой функции.
6. Иначе пытаемся найти переменную с именем `st` и создаем узел переменной.

Листинг 9.49. Процедура анализа констант, скобок, функций и переменных

```
procedure PMult(var s: string; var D: PNode);
var D1  : PNode;
    i, k: Integer;
    st  : string;
begin
  PopSpace(s);
  if s <> '' then begin
    if s[1] in ['0'..'9'] then begin  // Const
      New(D);  D.TypNode := nt_cons;
      SetConst(s, D.vConst, D.V.typ);
      if Error <> 0 then Exit;
    end
    else
    if s[1] = '(' then begin  // формула в скобках
      Pop(s, 1);
      PFormula(s, D);  if Error <> 0 then Exit;
      PopSpace(s);
      if (Length(s) > 0) and (s[1] <> ')') then begin
        Error := 9;  Exit;    // нет ')'
      end;
      Pop(s, 1);
    end
    else  // NOT стандартная функция или переменная
    if (Length(s) > 3) and (Copy(s,1,4)='not ')
    then begin
      Pop(s, 4);
      D1 := D;  New(D);
      D.TypNode := nt_Func;  D.V.typ := vt_boolean;
      D.NFunc := 8;  D.arg := D1;
    end
    else begin  // стандартная функция или переменная
      st := Pop(s, 1);
      while (s<>'') and (s[1] in Chars) do st:=st+Pop(s, 1);
      Ok := False;  i := 0;  // функция
```

```
      while (i < NumFunc) and not Ok do begin
        Inc(i);  Ok := st = UpperCase(NameFunc[i]);
      end;
      // функция найдена
      if Ok then begin
        if (Length(s) = 1) or (s[1] <> '(') then begin
          Error := 8;  Exit;  // Нет скобки '('
        end;
        PopSpace(s);
        New(D);
        D.TypNode:=nt_Func; D.NFunc:=i; D.V.typ:=vt_double;
        // считаем скобки
        st := '';  k := 1;  i := 1;
        while (i < Length(s)) and (k <> 0) do begin
          Inc(i);  st := st + s[i];
          case s[i] of
            '(': Inc(k);
            ')': Dec(k);
          end;
        end;
        st := Copy(st, 1, Length(st) - 1);
        s := Copy(s, i + 1, Length(s) - i);
        PFormula(st, D.arg);  if Error <> 0 then Exit;
        PopSpace(s);
        Exit;
      end
      else begin  // функция не найдена
        Ok := False;  i := -1;  // переменная
        while (i < LenVariabl - 1) and not Ok do begin
          Inc(i);  Ok := st = UpperCase(AVariable[i].Name);
        end;
        if Ok then begin  // переменная найдена
          New(D);
          D.TypNode:=nt_Variable;
          D.V.typ := AVariable[i].V.typ;
          D.Variable := AVariable[i];
          PopSpace(s);  Exit;
        end
        else Error := 7;  // неизвестная переменная
      end
    end  // формула в скобках
  end  // s <> ''
  else begin
    D := nil;  Error := 6;  // Ожидается множитель
  end;
end; // PMult
```

Анализ констант ('*e*', '*E*', '+', '-', '0'..'9', '.')

Шаг 6. С помощью процедуры `SetConst()` (листинг 9.50) анализуем константу, состоящую из символов `['e', 'E']`, `['+', '-', '0'..'9', '.']`.

Листинг 9.50. Процедура анализа констант ('*e*', '*E*', '+', '-', '0'..'9', '.')

```
procedure SetConst(var s: string; var x: Real;
                   var t: TVariableType);
label 1;
begin
  t := vt_integer;  st := Pop(s, 1);
  1:
  if (s <> '') and (s[1] in ['0'..'9', '.']) then begin
    if s[1] = '.' then t := vt_double;
    st := st + Pop(s, 1);
    goto 1;
  end
  else
  if (Length(s) > 1) and (s[1] in ['e', 'E']) and
     (s[2] in ['+', '-', '0'..'9', '.']) then begin
    st := st + s[1];  t := vt_double;
    st := st + Pop(s,1);
    goto 1;
  end;
  Val(st, x, Code);
  PopSpace(s);
end;
```

9.5.3. Вычисление переменной

Как уже говорилось в *разд. 9.5.2*, решение задачи создания интерпретатора состоит из двух этапов: на первом этапе создается структура данных `Operator`; на втором этапе при заданном значении `x` эта структура используется для вычисления переменной `y` с помощью процедуры `Run_Formula(Operator)`.

Перейдем к рассмотрению второго этапа. Если узел структуры `Operator` имеет тип "выражение", то для вычисления значения узла вызывается рекурсивная функция `Run_Tree()`. Для узла типа "условный оператор" логическое выражение также вычисляется с помощью функции `Run_Tree()`, а операторы после `'then'` и `'else'` выполняются рекурсивным вызовом процедуры `Run_Formula()`. Текст этой процедуры приведен в листинге 9.51.

Листинг 9.51. Процедура выполнения операторов присваивания и условных операторов

```
procedure Run_Formula(Operator: POperator);
begin
```

```
  case Operator.TypeOperator of
    ot_expression:
      AVariable[Operator.NumVar].V := Run_Tree(Operator.Node);
    ot_ifthen:
      if Run_Tree(Operator.Node).L
        then Run_Formula(Operator.OperThen)
        else
          if Operator.OperElse <> nil then
            Run_Formula(Operator.OperElse);
  end;
end;
```

Рекурсивная процедура `Run_Formula()` для обхода дерева использует восходящую стратегию, если тип узла — "операция". В таких узлах значение вычисляется арифметическими операциями по значениям левой и правой ветвей дерева. Если тип узла — "функция", то рекурсивным вызовом функции `Run_Tree()`, приведенной в листинге 9.52, вычисляется значение аргумента и, в зависимости от номера функции `NFunc()`, используется одна из арифметических функций. Для узла типа "константа" значение берется из поля `vConst`. Для узла типа "переменная" в массиве находится соответствующая переменная, и значение узла берется из этой переменной.

Листинг 9.52. Процедура вычисления выражений по бинарному дереву

```
function Run_Tree(D: PNode): TVar;
const Eps = 1.0e-6;
var p1, p2: TVar;
    n, i: Integer;
    sign: Shortint;
begin
  with D^ do
  case TypNode of
    nt_Oper:
      begin
        p1 := Run_Tree(Left);  if Error <> 0 then Exit;
        p2 := Run_Tree(Right);  if Error <> 0 then Exit;
        case Label_Op of
          '+': Result.d := p1.d + p2.d;
          '-': Result.d := p1.d - p2.d;
          '*': Result.d := p1.d * p2.d;
          '/': if p2.d = 0 then begin
                 Error := 10;  Exit;  // Деление на 0
               end
               else Result.d := p1.d/p2.d;
          'a': Result.L := p1.L and p2.L;
          'o': Result.L := p1.L or p2.L;
          'x': Result.L := p1.L xor p2.L;
```

```
        'n': Result.L := not p1.L;
        '1': Result.L := p1.d = p2.d;
        '2': Result.L := p1.d < p2.d;
        '3': Result.L := p1.d > p2.d;
        '4': Result.L := p1.d <> p2.d;
        '5': Result.L := p1.d <= p2.d;
        '6': Result.L := p1.d >= p2.d;
        '^': begin             //p1^p2
               n := Round(p2.d);
               if Abs(n - p2.d) < Eps then begin
                 if n < 0 then sign := -1 else sign := 1;
                 n := Abs(n);  Result.d := 1;
                 for i := 1 to n do
                   Result.d := Result.d * p1.d;
                 if (sign < 0) and (Result.d <> 0)
                   then Result.d := 1/Result.d;
               end
               else
                 if p1.d = 0 then Result.d := 0 else begin
                   Result.d := Exp(p2.d * ln(Abs(p1.d)))
                 end;
             end;
    end;
  end;
nt_Func:
  begin
    p1 := Run_Tree(arg);
    if Error <> 0 then Exit;
    case NFunc of
      1: Result.d := sin(p1.d);
      2: Result.d := cos(p1.d);
      3: if p1.d > 0 then Result.d := ln(p1.d)
         else begin
           Error := 11; Exit;  // Ln от отриц. числа
         end;
      4: Result.d := Exp(p1.d);
      5: Result.d := Abs(p1.d);
      6: if p1.d > 0 then Result.d := sqrt(p1.d)
         else begin
           Error:=12;  Exit;  //Корень отриц. числа
         end;
      7: Result.d := -p1.d;
      8: Result.L := not p1.L;
    end;
  end;
```

```
    nt_Cons: Result.d := vConst;
    nt_Variable:
      begin
        i := -1;  Ok := False;
        while (i < LenVariabl - 1) and not Ok do begin
          Inc(i);  Ok := Variable.Name = AVariable[i].Name;
        end;
        Result := AVariable[i].V;
      end;
  end;
end;
```

В листинге 9.53 приведен пример функции, используемой при построении графика.

Листинг 9.53. Пример использования интерпретатора

```
function F(x: Real; Operator : POperator): Real;
begin
  Error := 0;
  AVariable[1].V.d := x;
  Run_Formula(Operator);
  Result := AVariable[2].V.d;
end;
```

Полностью проект "Интерпретатор" приведен на компакт-диске в папке **Примеры | Глава 9 | Интерпретатор**.

Глава 10

Визуальный генератор отчетов

10.1. Постановка задачи

В этой главе рассматривается задача визуального создания сложных отчетов, которые могут включать строковые надписи (`Label`) и таблицы (`Grid`). Надписи могут быть статическими или вычисляться в процессе заполнения отчета другой программой. Таблицы могут иметь горизонтальную или вертикальную ориентацию. Они могут содержать статические заголовки колонок и поля, вычисляемые в процессе заполнения отчета. Результатом работы генератора отчетов является нетипизированный файл, в котором хранятся следующие данные:

- общие данные об отчете (название отчета, размеры полей при выводе на лист бумаги, шаг сетки); данные о строковых надписях (положение на листе бумаги, ширина, высота, вид шрифта, отметка о статичности или номер функции);
- данные о таблицах (положение на листе бумаги, ширина, высота, вид шрифта, номер таблицы);
- данные о строках таблиц (название, номер таблицы, отметка о статичности или номер функции).

Генератор отчетов должен выполнять следующие функции:

- создавать и удалять объекты (надписи и таблицы);
- настраивать свойства объектов;
- перемещать объекты;
- изменять общие параметры отчета.

В качестве примера приведем проект, предназначенный для обработки информации, полученной в процессе съемки профилограмм подземных камер с помощью звуколокационного профилографа. Для подготовки отчетов в этом проекте с помощью генератора отчетов было подготовлено несколько шаблонов, два из которых представлены на рис. 10.1 и 10.2. Они отличаются графиками, надписями и таблицами.

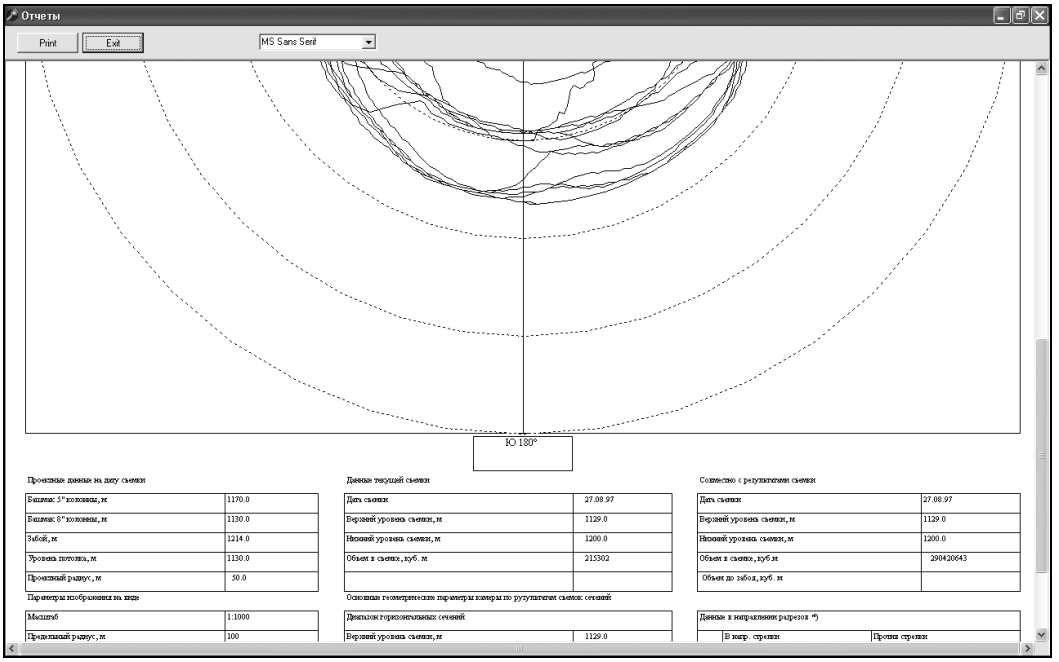


Рис. 10.1. Подготовка шаблона отчета для горизонтального разреза каверны

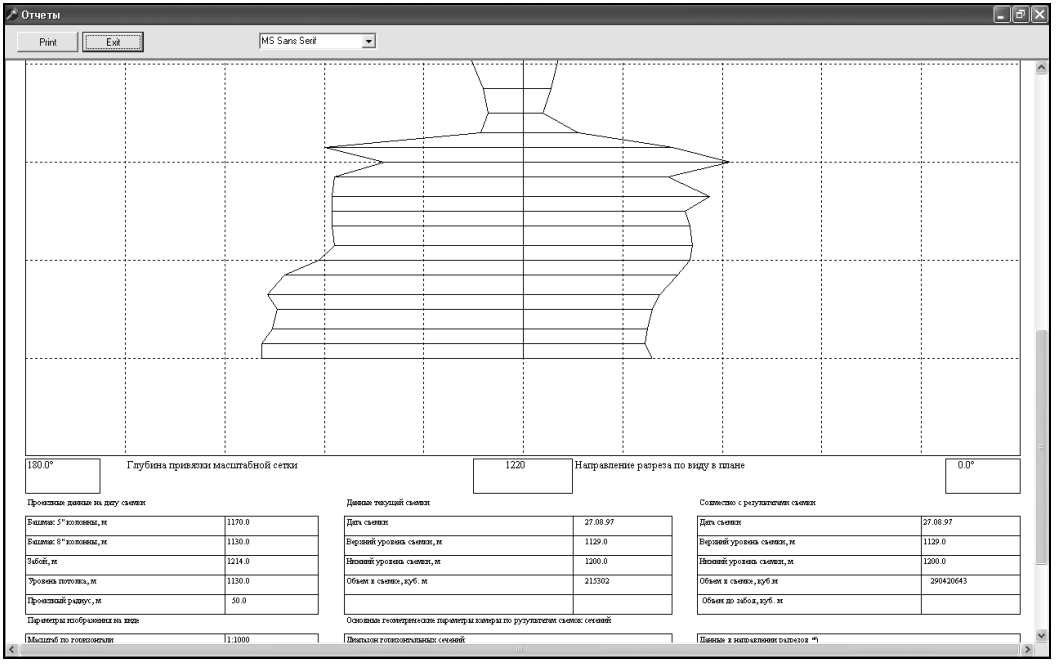


Рис. 10.2. Вывод на экран отчета для вертикального разреза каверны

Разрабатываемое приложение, используя подготовленный шаблон отчета, заполняет поля по номерам функций и выводит отчет на экран или на принтер.

Обращаем ваше внимание на то, что генератор отчетов является универсальным приложением и может быть использован для визуализации данных из разных предметных областей. Проект, предназначенный для обработки информации, полученной в процессе съемки профилограмм подземных камер с помощью звуколокационного профилографа и генератор отчетов почти независимы: общими у них являются только номера функций у объектов. Если развивать идею этого генератора отчетов на реляционные базы данных, то, естественно, придется подключать весь механизм доступа к данным.

10.2. Описание структуры данных

Все типы объектов (строки, таблицы, области графиков и т. д.) описываются одной структурой (листинг 10.1).

Листинг 10.1. Структура объектов

```
TObj=record
  Inf     : TInf;
  ObjRow  : array of TObjRow;
end;
```

В свою очередь, основное информационное поле объектов `Inf` имеет следующую структуру (листинг 10.2).

Листинг 10.2. Структура информационного поля объектов

```
  TInf=record
    Caption : string[80]; // название объекта
    type_Obj: byte;       // тип объекта
    Id      : byte;       // № функции,0-статичен
    fl_vert : byte;       // вертикальная таблица
    fl_all  : byte;       // заполнять все строки
    pr      : byte;       // процент ширины
    Left,Top,Width,Height: Single;  // геометрические параметры объекта
    FontName: string[15]; // имя шрифта
    FontSize: shortint;   // размер шрифта
    FontStyle,            // стиль шрифта
    rezerv  : byte;       // резервное поле
  end;
```

В приведенной структуре поля имеют следующий смысл:

- `Caption` — название объекта (содержимое статичной строки или название таблицы);
- `type_Obj` — тип объекта (1 — строка, 2 — таблица и т. д.);
- `Id` — номер функции строки или уникальный номер таблицы. При `Id=0` строка статична, т. е. выводится содержимое свойства `Text`;
- `fl_vert` — признак вертикальной или горизонтальной ориентации таблицы;
- `fl_all` — признак заполнения одной или всех строк таблицы;
- `pr` — поле, задающее процент ширины первого столбца таблицы от ширины горизонтальной таблицы;
- `Left`, `Top`, `Width`, `Height` — геометрические параметры объекта (координаты верхнего левого угла, ширина и высота);
- `FontName` — имя шрифта;
- `FontSize` — размер шрифта;
- `FontStyle` — стиль шрифта;
- `reserv` — резервное поле.

Второй важной структурой является столбец таблицы (листинг 10.3).

Листинг 10.3. Структура столбцов таблицы

```
TObjRow=record
  Text: string[80];
  Id,F: byte;
  f_1,f_2: byte;
end;
```

В приведенной структуре столбцов таблицы поля имеют следующий смысл:

- `Text` — название статичного столбца. Для таблиц вертикальной ориентации в названии статичного столбца можно использовать символ "\" для разбиения названия на строки;
- `Id` — номер таблицы;
- `F` — номер функции (при `F=0` функция не определена).

Данные о каждом отчете записываются в нетипизированный файл с именем `FileName`. Сначала записываются размеры полей на листе бумаги `MarginsLeft`, `MarginsTop`, `MarginsRight`, `MarginsBottom` в миллиметрах, название отчета `NameProject` и шаг сетки `dh` в миллиметрах. Затем записываются данные об объектах (поле `Inf`), и, наконец, записываются данные о столбцах таблиц `ObjRow`. Во время выполнения проекта данные об объектах и о столбцах хранятся в массивах `Obj: array of Tobj`. Процедуры записи данных приведены в листинге 10.4, а процедуры чтения данных — в листинге 10.5.

Листинг 10.4. Запись данных

```
procedure SaveObj(FileName: string);
var i,j: integer;
begin
  AssignFile(f,FileName); Rewrite(f,1);
  BlockWrite(f,MarginsLeft,SizeOf(MarginsLeft));
  BlockWrite(f,MarginsTop,SizeOf(MarginsTop));
  BlockWrite(f,MarginsRight,SizeOf(MarginsRight));
  BlockWrite(f,MarginsBottom,SizeOf(MarginsBottom));
  BlockWrite(f,NameProject,SizeOf(NameProject));
  BlockWrite(f,dh,SizeOf(dh));

  BlockWrite(f,CountObj,SizeOf(CountObj));
  for i:=0 to CountObj-1 do
  with Obj[i] do begin
    BlockWrite(f,Inf,SizeOf(Inf));
    CountRow:=Length(ObjRow);
    BlockWrite(f,CountRow,SizeOf(CountRow));
    for j:=0 to CountRow-1 do
      BlockWrite(f,ObjRow[j],SizeOf(ObjRow[j]));
  end;
  CloseFile(f);
end;
```

Листинг 10.5. Чтение данных

```
procedure Open_Obj(FileName: string);
var i,j: integer;
  Ok: boolean;
begin
  AssignFile(f,FileName); Reset(f,1);
  BlockRead(f,MarginsLeft,SizeOf(MarginsLeft));
  BlockRead(f, MarginsLeft,SizeOf(MarginsTop));
  BlockRead(f,MarginsRight,SizeOf(MarginsRight));
  BlockRead(f,MarginsBottom,SizeOf(MarginsBottom));
  BlockRead(f,NameProject,SizeOf(NameProject));
  BlockRead(f,dh,SizeOf(dh));

  BlockRead(f,CountObj,SizeOf(CountObj));
  SetLength(Obj,CountObj);
  for i:=0 to CountObj-1 do
  with Obj[i] do begin
    BlockRead(f,Inf,SizeOf(Inf));
    BlockRead(f,CountRow,SizeOf(CountRow));
```

```
    SetLength(Obj[i].ObjRow,CountRow);
    for j:=0 to CountRow-1 do
      BlockRead(f,ObjRow[j],SizeOf(ObjRow[j]));
  end;
  CloseFile(f);
end;
```

10.3. Структура проекта

На рис. 10.3 представлен внешний вид основной формы проекта "Визуальный генератор отчетов".

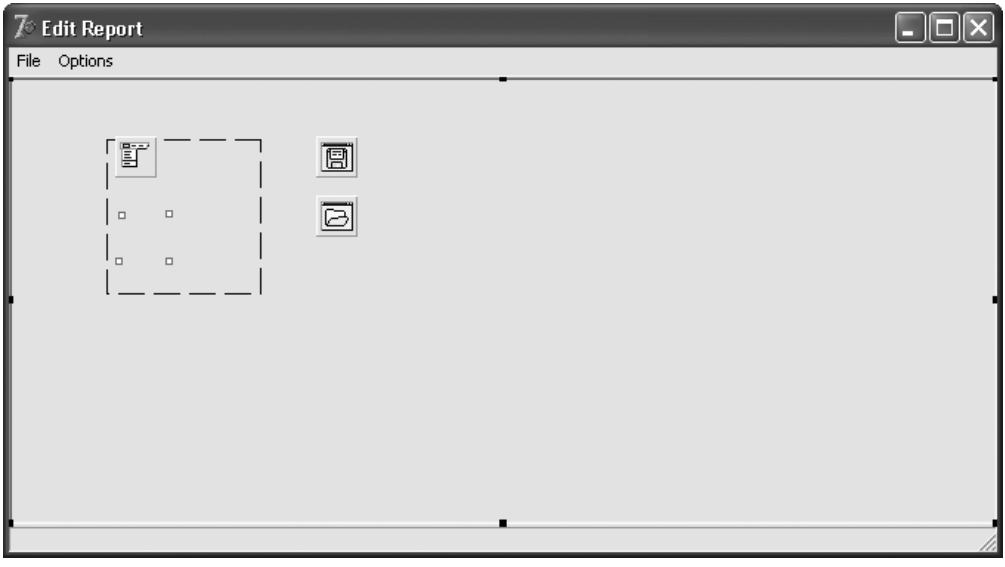


Рис. 10.3. Основная форма проекта "Визуальный генератор отчетов"

Основная форма проекта `FormMain` представлена модулем `UnitMain` и имеет внешнее имя **Edit Report**. На форме размещен компонент `Im: TImage` для рисования и перемещения надписей и таблиц, четыре компонента типа `TShape` для изменения положения надписей и таблиц, компонент меню и кнопки доступа к диалоговым окнам для открытия и сохранения файлов.

На инструментальной форме `FormTools` с внешним именем **Tools** (рис. 10.4) размещены четыре кнопки для работы с файлами (**NewReport**, **ReadReport**, **SaveReport**, **SaveAsReport**) и три инструментальные кнопки, образующие группу, для работы с объектами (**MoveObject** — перемещение объекта, **NewLabel** — создать новую надпись, **NewGrid** — создать новую таблицу).

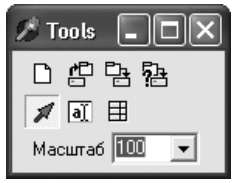


Рис. 10.4. Инструментальная форма проекта "Визуальный генератор отчетов"

Рабочим полем является компонент `Im: TImage`, размещенный на компоненте `Sb: TScrollBox`, обладающим линейками прокрутки. Для изменения масштаба просмотра используется компонент `FormTools.ComboBox1`, свойство `Items` которого содержит строки с возможными процентами масштаба: 50, 75, 100, 120, 150, 175, 200, 250.

При инициализации проекта задается масштаб просмотра `Ms:=100` (листинг 10.6).

Листинг 10.6. Инициализация проекта

```
procedure TFormMain.FormCreate(Sender: TObject);
begin
  drawing:=false;   // флаг перемещения объектов
  NameProject:='';  // имя отчета
  fl_tools:=0;      // флаг инструмента
  CountObj:=0;      // число объектов
  NumSelect:=0;     // № выделенного объекта
  Ms:=100;          // масштаб 100 %
  SetPage;          // установить параметры страницы
  ShowPage;         // нарисовать страницу
end;
```

При выборе нового масштаба в компоненте `ComboBox1` и нажатии клавиши `<Enter>` меняется масштаб, переустанавливаются параметры страницы, и перерисовывается вся страница эскиза (листинг 10.7).

Листинг 10.7. Смена масштаба

```
procedure TFormTools.ComboBox1KeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
var I: integer;
begin
  case Key of
    13: begin
          Val(ComboBox1.Text,i,Code);
          if Code=0 then begin
            Ms:=i; FormMain.SetPage; FormMain.ShowPage;
          end;
```

```
        end;
  end;
end;
```

Параметры страницы устанавливаются в зависимости от размеров компонента `Sb: TScrollBox`, размеров листа (`hx, hy`) и масштаба `Ms` (листинг 10.8).

Листинг 10.8. Установка параметров страницы

```
procedure TFormMain.SetPage;
const c1=40;
begin
  Im.Width:=Round(Ms/100*(Sb.Width-c1));
  Im.Height:=Round(HeightPage/WidthPage*Im.Width);

  if (Sb.width-c1-Im.Width) div 2>0
    then Im.Left:=(Sb.width-c1-Im.Width) div 2
    else Im.Left:=10;
  Im.Top:=10;
end;
```

10.4. Рисование страницы эскиза

Выполнение многих функций в данном проекте завершается рисованием всей страницы эскиза `ShowPage` (листинг 10.9).

Листинг 10.9. Рисование страницы эскиза

```
procedure TFormMain.ShowPage;
var i,j,n,m: integer;
begin
  with Im.Canvas do begin
    Pen.Mode:=pmCopy; Brush.Color:=clWhite;
    RectAngle(0,0,Im.Width,Im.Height);

    Pen.Color:=clSilver;
    RectAngle(ImX(MarginsLeft),ImY(MarginsTop),
       ImX(WidthPage-MarginsRight),ImY(HeightPage-MarginsBottom));

          //число узловых точек по вертикали
    n:=Trunc((HeightPage-MarginsTop-MarginsBottom)/dh);
          //число узловых точек по горизонтали
    m:=Trunc((WidthPage-MarginsLeft-MarginsRight)/dh);
    // точки сетки
    for i:=0 to m do
```

```
      for j:=0 to n do
        Pixels[ImX(MarginsLeft+i*dh),ImY(MarginsTop+j*dh)]:=
          clSilver;

    for i:=0 to CountObj-1 do
    with Obj[i].Inf do begin
      SetFont(i,Font); // установить параметры шрифта
      case type_Obj of
        1: begin       // элемент типа строка
             if FontStyle and (1 shl 4)<>0 then
               // окаймляющий прямоугольник
               RectAngle(
                     ImX(Left-1),
                     ImY(Top)-TextHeight(Caption)-1,
                     ImX(Left)+TextWidth(Caption)+1,
                     ImY(Top+1));
             // вывод строки
             TextOut(ImX(Left),
                     ImY(Top)-TextHeight(Caption),Caption);
           end;
        2: if fl_vert=0 then ShowGridV(i) // таблица
                        else ShowGridG(i);
        3..5: ShowGrid3(i);               // прямоугольники
      end;
    end;
  end;
  ShowShape;
end;
```

При рисовании страницы используются функции масштабирования `ImX`, `ImY` (листинг 10.10), процедура установки параметров шрифта `SetFont(i)` (листинг 10.11), процедура `ShowGridV(i)` рисования вертикальной таблицы (листинг 10.12) и процедура `ShowGridG(i)` рисования горизонтальной таблицы (листинг 10.13).

Функции масштабирования, использующие ширину `hx` и высоту `hy` листа бумаги в миллиметрах, неоднократно описывались ранее, например, в *разд. 9.1*.

Листинг 10.10. Функции масштабирования

```
function TFormMain.ImX(x: real48): integer;
begin
  ImX:=Round(x*Im.Width/WidthPage);
end;

function TFormMain.ImY(y: real48): integer;
begin
  ImY:=Round(y*Im.Height/HeightPage);
end;
```

Процедура установки параметров шрифта для i-го объекта (листинг 10.11) использует поля FontName, FontSize и биты 0–3 поля FontStyle. Обратите внимание на использование параметра масштаба при установке размера шрифта.

Листинг 10.11. Установка параметров шрифта

```
procedure SetFont(i: integer; MyFont: TFont);
begin
  with Obj[i].Inf do begin
    MyFont.Name:=FontName;
    MyFont.Size:=Round(Ms/100*FontSize);

    MyFont.Style:=[];
    if FontStyle and (1 shl 0)<>0
      then MyFont.Style:=MyFont.Style+[fsBold]
      else MyFont.Style:=MyFont.Style-[fsBold];
    if FontStyle and (1 shl 1)<>0
      then MyFont.Style:=MyFont.Style+[fsItalic]
      else MyFont.Style:=MyFont.Style-[fsItalic];
    if FontStyle and (1 shl 2)<>0
      then MyFont.Style:=MyFont.Style+[fsUnderline]
      else MyFont.Style:=MyFont.Style-[fsUnderline];
    if FontStyle and (1 shl 3)<>0
      then MyFont.Style:=MyFont.Style+[fsStrikeOut]
      else MyFont.Style:=MyFont.Style-[fsStrikeOut];
  end;
end;
```

На рис. 10.1 представлены только таблицы вертикальной ориентации, у которых заголовочные ячейки располагаются вертикально. Над таблицей выводится название таблицы Obj[m].Text с использованием полужирного стиля начертания символов. Сама таблица состоит из двух столбцов: в первом столбце выводятся названия строк Obj1[i].Text, а во втором — значения поля Obj1[i].F, в соответствии с которым в основном проекте будут выполняться какие-либо действия по конкретной тематике отчета.

Листинг 10.12. Рисование вертикальной таблицы

```
procedure TFormMain.ShowGridV(m: integer);
var i,j,n: integer;
begin
  with Obj[m],Inf,Im.Canvas do begin
    n:=Length(ObjRow); // вычисление числа строк
    Pen.Color:=clBlack;
```

```
    Font.Name:=FontName;
    Font.Size:=Round(Ms/100*FontSize);
    Font.Style:=[fsBold]; // название таблицы
    TextOut(ImX(Left),ImY(Top)-TextHeight(Caption),Caption);
    Font.Style:=[];

    RectAngle(ImX(Left),ImY(Top),ImX(Left+Width),ImY(Top+Height));
    // вертикальная линия
    MoveTo(ImX(Left+0.01*pr*Width),ImY(Top));
    LineTo(ImX(Left+0.01*pr*Width),ImY(Top+Height));
    // горизонтальные линии
    if n<>0 then
    for i:=1 to n-1 do begin
      MoveTo(ImX(Left),ImY(Top+i*Height/n));
      LineTo(ImX(Left+Width),ImY(Top+i*Height/n));
    end;
    CountRow:=n; // заполнение столбцов
    for j:=0 to CountRow-1 do begin
      // заполнение 1-го столбца
      TextRect(Rect(
        ImX(Left)+1,ImY(Top+(j-1)*Height/n)+1,
        ImX(Left+0.01*pr*Width)-1,ImY(Top+j*Height/n)-1),
        ImX(Left)+2,
        ImY(Top+(j-0.5)*Height/n)-
           Im.Canvas.TextHeight(ObjRow[j].Text) div 2,
        ObjRow[i].Text);
      // заполнение 2-го столбца
      s:=IntToStr(ObjRow[i].F);
      if s<>'0' then
      TextRect(Rect(
          ImX(Left+0.01*pr*Width)+1,
          ImY(Top+(j-1)*Height/n)+1,
          ImX(Left+Width)-1,
          ImY(Top+(j-0)*Height/n)-1),
          ImX(Left+0.01*pr*Width)+2,
          ImY(Top+(j-0.5)*Height/n)-
              Im.Canvas.TextHeight(ObjRow[j].Text) div 2,s);
    end;
  end;
end;
```

При рисовании таблицы вертикальной ориентации сначала по массиву `Obj1` вычисляется число строк этой таблицы `n`, затем методом `TextOut` выводится заголовок `Obj[m].Text`. При проведении вертикальной линии, разделяющей два столб-

ца, учитывается значение поля `Obj[m].pr`, в котором содержится информация о проценте ширины первого столбца от общей ширины таблицы. Заполнение ячеек таблицы реализуется методом `TextRect`, отсекающим информацию вне прямоугольника вывода.

У таблицы горизонтальной ориентации (рис. 10.5) заголовки столбцов могут состоять из нескольких строк.

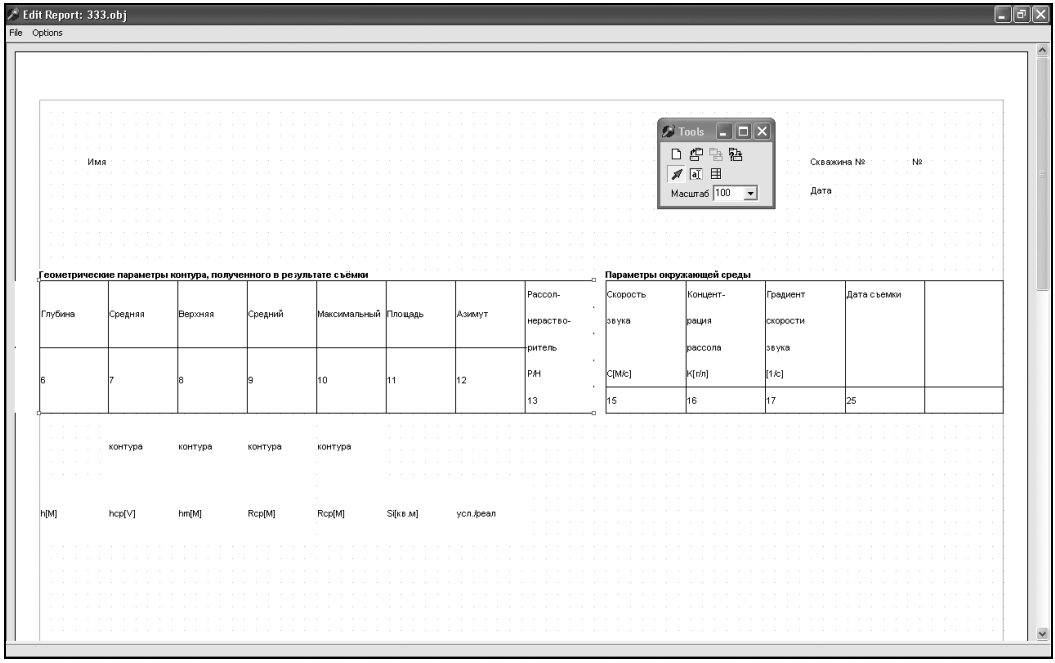
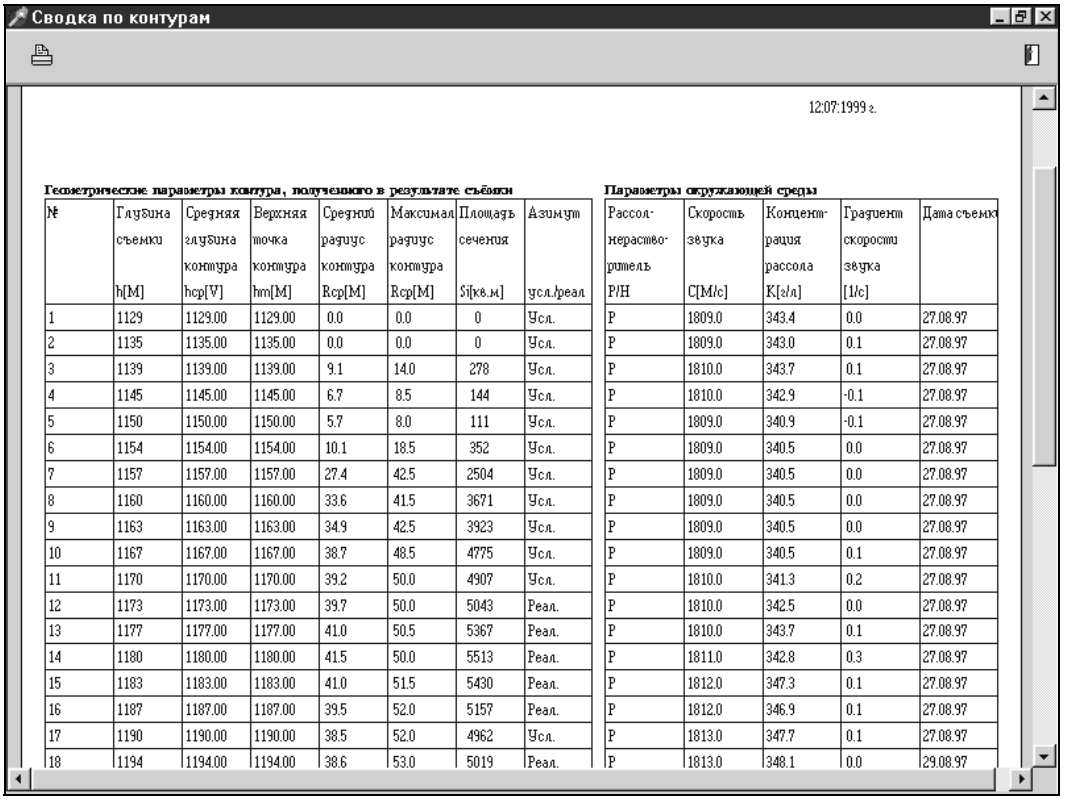


Рис. 10.5. Таблица горизонтальной ориентации

Над таблицей выводится название таблицы `Obj[m].Text`. Сама таблица состоит из двух строк: в первой строке выводятся названия столбцов `Obj1[i].Text`, а во второй — значения поля `Obj1[i].F`, в соответствии с которыми в основном проекте будут выполняться какие-либо действия по тематике отчета. Для разбиения заголовка столбца на несколько строк используется символ "\". Например, для таблицы "Параметры окружающей среды" заголовок второго столбца имеет вид `Obj1[2].Text :='Концент-\рация\рассола\К[г/л]'`.

Горизонтальные таблицы обычно используются для представления нескольких строк информации (рис. 10.6).

Рисование горизонтальной таблицы реализуется процедурой `ShowGridG` (листинг 10.13).



Геометрические параметры контура, полученного в результате съёмки

№	Глубина съемки h[M]	Средняя глубина контура hcp[V]	Верхняя точка контура hm[M]	Средний радиус контура Rcp[M]	Максимал радиус контура Rcp[M]	Площадь сечения Si[кв.м]	Азимут усл./реал
1	1129	1129.00	1129.00	0.0	0.0	0	Усл.
2	1135	1135.00	1135.00	0.0	0.0	0	Усл.
3	1139	1139.00	1139.00	9.1	14.0	278	Усл.
4	1145	1145.00	1145.00	6.7	8.5	144	Усл.
5	1150	1150.00	1150.00	5.7	8.0	111	Усл.
6	1154	1154.00	1154.00	10.1	18.5	352	Усл.
7	1157	1157.00	1157.00	27.4	42.5	2504	Усл.
8	1160	1160.00	1160.00	33.6	41.5	3671	Усл.
9	1163	1163.00	1163.00	34.9	42.5	3923	Усл.
10	1167	1167.00	1167.00	38.7	48.5	4775	Усл.
11	1170	1170.00	1170.00	39.2	50.0	4907	Усл.
12	1173	1173.00	1173.00	39.7	50.0	5043	Реал.
13	1177	1177.00	1177.00	41.0	50.5	5367	Реал.
14	1180	1180.00	1180.00	41.5	50.0	5513	Реал.
15	1183	1183.00	1183.00	41.0	51.5	5430	Реал.
16	1187	1187.00	1187.00	39.5	52.0	5157	Реал.
17	1190	1190.00	1190.00	38.5	52.0	4962	Усл.
18	1194	1194.00	1194.00	38.6	53.0	5019	Реал.

Параметры окружающей среды

Рассол-нерасство-ритель P/H	Скорость звука C[M/c]	Концент-рация рассола K[г/л]	Градиент скорости звука [1/c]	Дата съемки
P	1809.0	343.4	0.0	27.08.97
P	1809.0	343.0	0.1	27.08.97
P	1810.0	343.7	0.1	27.08.97
P	1810.0	342.9	-0.1	27.08.97
P	1809.0	340.9	-0.1	27.08.97
P	1809.0	340.5	0.0	27.08.97
P	1809.0	340.5	0.0	27.08.97
P	1809.0	340.5	0.0	27.08.97
P	1809.0	340.5	0.0	27.08.97
P	1809.0	340.5	0.1	27.08.97
P	1810.0	341.3	0.2	27.08.97
P	1810.0	342.5	0.0	27.08.97
P	1810.0	343.7	0.1	27.08.97
P	1811.0	342.8	0.3	27.08.97
P	1812.0	347.3	0.1	27.08.97
P	1812.0	346.9	0.1	27.08.97
P	1813.0	347.7	0.1	27.08.97
P	1813.0	348.1	0.0	29.08.97

Рис. 10.6. Заполнение горизонтальной таблицы в основном проекте

Листинг 10.13. Рисование горизонтальной таблицы

```
procedure TFormMain.ShowGridG(m: integer);
var i,j,n,k,q,NumS: integer;
    p,p1: Single;
    s1: array[1..6] of string[20];

procedure SetS(s: string);
// разбиение заголовка столбца на NumS строк
var i: byte;
begin
  NumS:=1; s1[NumS]:='';
  for i:=1 to Length(s) do
    if s[i]<>'\' then s1[NumS]:=s1[NumS]+s[i]
    else begin
      NumS:=NumS+1; s1[NumS]:='';
    end;
end;
```

```
begin
     // вычисление числа столбцов и максимального числа
     // строк в заголовках
  with Obj[m],Inf,Im.Canvas do begin
    q:=0;
    n:=Length(ObjRow);
    for i:=0 to CountRow-1 do begin
      k:=0;
      for j:=1 to Length(ObjRow[i].Text) do
        if ObjRow[i].Text[j]='\' then Inc(k);
      if k>q then q:=k;
    end;
    q:=q+1;
    if n<>0 then p:=1.0/n else p:=1;

    Pen.Color:=clBlack;
    Font.Name:=FontName;
    Font.Size:=Round(Ms/100*FontSize);
          // заголовок
    Font.Style:=[fsBold];
    TextOut(ImX(Left),ImY(Top)-TextHeight(Caption),Caption);
    Font.Style:=[];

    RectAngle(ImX(Left),ImY(Top),ImX(Left+Width),ImY(Top+Height));
    for i:=1 to n-1 do begin //vertical
      MoveTo(ImX(Left+p*i*Width),ImY(Top));
      LineTo(ImX(Left+p*i*Width),ImY(Top+Height));
    end;
    p1:=1.0/(q+1); // горизонтальная линия
    MoveTo(ImX(Left),ImY(Top+(q+0)*p1*Height));
    LineTo(ImX(Left+Width),ImY(Top+(q+0)*p1*Height));

    CountRow:=n;
    for j:=0 to CountRow-1 do begin
      SetS(ObjRow[j].Text);
      for k:=1 to NumS do
      if s1[k]<>'' then begin
        TextRect(Rect(
        ImX(Left+p*(j-1)*Width)+1,
        ImY(Top+(k-1)*p1*Height)+1,
        ImX(Left+p*j*Width)-1,
        ImY(Top+(k+0)*p1*Height)-1),
        ImX(Left+p*(j-1)*Width)+2,
        ImY(Top+(k-0.5)*p1*Height)-
```

```
          Im.Canvas.TextHeight(s1[k]) div 2,s1[k]);
      end;
      s:=IntToStr(ObjRow[j].F);
      if s<>'0' then // вывод номера функции
      TextRect(Rect(
          ImX(Left+p*(j-1)*Width)+1,
          ImY(Top+(q-0)*p1*Height)+1,
          ImX(Left+p*j*Width)-1,
          ImY(Top+(q+1)*p1*Height)-1 ),
          ImX(Left+p*(j-1)*Width)+2,
          ImY(Top+(q+0.5)*p1*Height)-
            Im.Canvas.TextHeight(s) div 2,s);
    end;
  end;
end;
```

При рисовании горизонтальной таблицы сначала вычисляется число столбцов `n` и максимальное число строк в заголовке `q`. Высота таблицы делится на `q+1` частей и на высоте `h*q/(q+1)` проводится горизонтальная линия. Процедура `SetS` разбивает заголовок столбца `Obj1[m].Text` на несколько строк, используя в качестве разделителя символ "\". Вывод заголовков столбцов реализован методом `TextRect`.

10.5. Добавление объектов

Рассматриваемое приложение позволяет добавлять строковые надписи (`Label`) при нажатой кнопке `NewLabel` или таблицы (`Grid`) при нажатой кнопке `NewGrid`. В первом случае флаг инструмента `fl_tools` принимает значение `1`, а во втором — значение `2` (листинг 10.14).

Листинг 10.14. Смена флага инструмента *fl_tools*

```
procedure TFormTools.NewLabelClick(Sender: TObject);
begin
  fl_tools:=1;
end;

procedure TFormTools.NewGridClick(Sender: TObject);
begin
  fl_tools:=2;
end;
```

При добавлении нового объекта строки обработчик события `ImMouseDown` вызывает форму `Form2` с внешним именем **Label** (рис. 10.7), с помощью которой задаются параметры этого объекта (листинг 10.15).

Листинг 10.15. Вызов формы для ввода новой строки

```
procedure TFormMain.ImMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  t:=Sb.VertScrollBar.Position;
  if Button=mbRight then begin
  end
  else
  case fl_tools of
    0: if FindObj(x,y) then begin
          if fl_dbl=0 then begin
            drawing:=true;
            dx:=x-Shape1.Left; dy:=y-Shape1.Top;
            x0:=x-d-2; y0:=y-d-2+t;
            h1:=Shape3.Left-Shape1.Left;
            h2:=Shape3.Top-Shape1.Top;
            Im.Canvas.DrawFocusRect(
              Rect(x0-dx,y0-dy,x0-dx+h1,y0-dy+h2));
          end
          else begin
            fl_new:=1;
            case Obj[NumSelect].Inf.type_Obj of
              1: FormEditLabel.ShowModal;
              2: FormEditTable.ShowModal;
            end;
            if fl_ok=1 then begin
              FormTools.SaveReport.Enabled:=True;
              ShowPage;
            end;
          end;
       end;
    1: begin
         fl_new:=0;
         du:=Round((ImI(x)-MarginsLeft)/dh)*dh+MarginsLeft;
         dv:=Round((ImJ(y)-MarginsLeft)/dh)*dh+MarginsLeft;
         FormEditLabel.ShowModal;
         if fl_ok=1 then ShowPage;
         FormTools.SaveReport.Enabled:=True;
         FormTools.MoveObject.Down:=true;
         fl_tools:=0;
       end;
    2..5:
       begin
         drawing:=true;
```

```
        x0:=x; y0:=y; dx:=x; dy:=y;
        Im.Canvas.DrawFocusRect(Rect(x0,y0,dx,dy));
      end;
  end;
  fl_dbl:=0;
end;
```

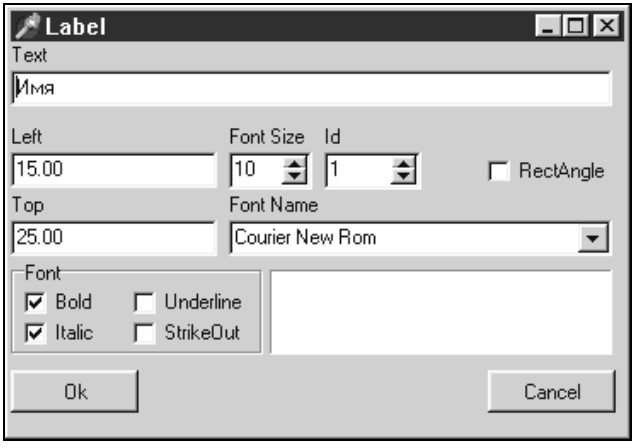


Рис. 10.7. Форма для редактирования параметров строки

На форме можно изменить текст надписи (поле **Текст**), изменить положение верхнего левого угла надписи (поля **Left** и **Top**), настроить параметры шрифта (группа флажков **Font**, поля **Font Size**, **Font Name**) и разрешить или запретить рисование прямоугольника вокруг надписи (флажок **RectAngle**).

При добавлении нового объекта таблицы необходимо указателем мыши отметить на форме прямоугольник, в котором будет отображаться таблица, и только после отпускания кнопки мыши обработчик события `ImMouseUp` вызовет форму `Form3` с внешним именем **Table** (рис. 10.8) для задания параметров таблицы (листинг 10.16).

Листинг 10.16. Вызов формы для ввода новой таблицы

```
procedure TFormMain.ImMouseDown(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  case fl_tools of
  2..5: begin
        drawing:=true;
        x0:=x; y0:=y; dx:=x; dy:=y;
        Im.Canvas.DrawFocusRect(Rect(x0,y0,dx,dy));
      end;
```

```
  end;
end;

procedure TFormMain.ImMouseMove(Sender: TObject;
  Shift: TShiftState; X,Y: Integer);
begin
  if drawing then
  case fl_tools of
  2..5: begin
         Im.Canvas.DrawFocusRect(Rect(x0,y0,dx,dy));
         dx:=x; dy:=y;
         Im.Canvas.DrawFocusRect(Rect(x0,y0,dx,dy));
       end;
  end;
end;

procedure TFormMain.ImMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  t:=Sb.VertScrollBar.Position;
  if drawing then
  case fl_tools of
  2: begin
         Im.Canvas.DrawFocusRect(Rect(x0,y0,dx,dy));
         drawing:=false; fl_new:=0;
         du:=Round((ImI(x0)-hl)/dh)*dh+hl;
         dv:=Round((ImJ(y0)-hl)/dh)*dh+hl;
         du1:=Round((ImI(dx)-hl)/dh)*dh+hl;
         dv1:=Round((ImJ(dy)-hl)/dh)*dh+hl;
         Form3.ShowModal;
         if fl_ok=1 then ShowPage;
         MoveObject.Down:=true; fl_tools:=0;
         SaveReport.Enabled:=True;
    end;
  end;
end;
```

Форма `Form3` позволяет редактировать название таблицы, определять параметры шрифта (поля **Font Name**, **Font Size**), добавлять, вставлять и удалять строки таблицы (три кнопки типа `TSpeedButton`), менять их названия в колонке таблицы с именем **Name** и номер вызываемой функции в колонке таблицы с названием **Value**, менять ориентацию таблицы (флажок "**вертикальная**").

Перед вызовом `Form3` масштабированные координаты левого верхнего и нижнего правого углов смещаются в ближайшую узловую точку. Например, для абсциссы левого верхнего угла: `du:=Round((ImI(x0)-hl)/dh)*dh+hl`.

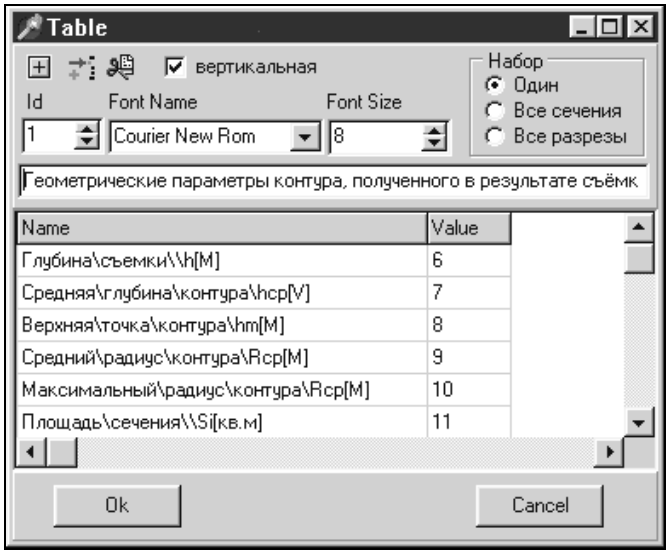


Рис. 10.8. Форма для редактирования свойств таблицы

10.6. Редактирование объектов

Вызов форм `Form2` или `Form3`, в которых можно изменять свойства объектов строки или таблицы, начинается обработчиком события двойного щелчка указателем мыши `ImDblClick` по компоненту `Im`, на котором рисуется объект (листинг 10.17).

Листинг 10.17. Начало вызова формы для редактирования свойств строки

```
procedure TFormMain.ImDblClick(Sender: TObject);
begin
  fl_dbl:=1;
end;
```

После обработчика события `ImDblClick` автоматически вызывается процедура `ImMouseDown`, в которой вызывается форма для редактирования параметров строки или таблицы (листинг 10.18).

Листинг 10.18. Вызов формы для редактирования параметров строки или таблицы

```
procedure TFormMain.ImMouseDown(Sender: TObject;
  Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var i: integer;
begin
  t:=Sb.VertScrollBar.Position;
```

```
  case fl_tools of
    0: if FindObj(x,y) then  // поиск объекта
       begin
          if fl_dbl=0 then   // перемещение объекта
          begin
            drawing:=true;
            dx:=x-Shape1.Left; dy:=y-Shape1.Top;
            x0:=x-d-2; y0:=y-d-2+t;
            h1:=Shape3.Left-Shape1.Left;
            h2:=Shape3.Top-Shape1.Top;
            Im.Canvas.DrawFocusRect(
              Rect(x0-dx,y0-dy,x0-dx+h1,y0-dy+h2));
          end
          else               // редактирование объекта
          begin
            fl_new:=1;
            case Obj[NumSel].typ of
              1: Form2.ShowModal;
              2: Form3.ShowModal;
            end;
            if fl_ok=1 then
            begin
              SaveReport.Enabled:=True;
              ShowPage;
            end;
          end;
       end;
  end;
  fl_dbl:=0;
end;
```

Обработчик события `ImMouseDown` при значении `fl_tools`, равном `0`, вызывает процедуру `FindObj`, которая определяет номер выделенного объекта `NumSel` (листинг 10.19).

Листинг 10.19. Поиск объекта

```
function TFormMain.FindObj(x,y: integer): boolean;
var i: integer; Ok: boolean;
begin
   Ok:=false; i:=-1;
   while (i<CountObj-1) and not Ok do begin
     Inc(i);
     with Obj[i].Inf do
     case type_Obj of
       1: Ok:=(ImX(Left)+2<=x) and
```

```
              (x<=ImX(Left)+Im.Canvas.TextWidth(Text)+2) and
              (ImX(Top)-Im.Canvas.TextHeight(Text)+2<=y) and
              (y<=ImX(Top)+2);
       2..5: Ok:=(ImX(Left)+2<=x) and
              (x<=ImX(Left+Width)+2) and
              (ImX(Top)+2<=y) and
              (y<=ImX(Top+Height)+2);
     end;
   end;
   FindObj:=Ok;
   if Ok then begin
     NumSelect:=i; ShowShape;
   end;
end;
```

В случае удачного завершения поиска меняется номер выделенного объекта `NumSelect` и вызывается процедура `ShowShape`, рисующая четыре прямоугольных маркера выделения типа `TShape` по углам выделенного объекта (листинг 10.20).

Листинг 10.20. Изменение координат компонентов *TShape*

```
procedure TFormMain.ShowShape;
begin
  if CountObj>0 then begin
    t:=Sb.VertScrollBar.Position;
    Shape1.Visible:=true;
    Shape2.Visible:=true;
    Shape3.Visible:=true;
    Shape4.Visible:=true;
    with Obj[NumSelect].Inf,Im.Canvas do
    case type_Obj of
      1: begin
           Im.Canvas.Font.Name:=FontName;
           Im.Canvas.Font.Size:=FontSize;
           Shape1.Left:=ImX(Left)+2+d;
           Shape1.Top:=ImX(Top)-TextHeight(Text)+2+d-t;
           Shape2.Left:=ImX(Left)+TextWidth(Text)+2+d;
           Shape2.Top:=ImX(Top)-TextHeight(Text)+2+d-t;
           Shape3.Left:=ImX(Left)+TextWidth(Text)+2+d;
           Shape3.Top:=ImX(Top)+2+d-t;
           Shape4.Left:=ImX(Left)+2+d;
           Shape4.Top:=ImX(Top)+2+d-t;
         end;
      2..5:
         begin
           Shape1.Left:=ImX(Left)+2+d;
```

```
          Shape1.Top:=ImX(Top)+2+d-t;
          Shape2.Left:=ImX(Left+Width)+2+d;
          Shape2.Top:=ImX(Top)+2+d-t;
          Shape3.Left:=ImX(Left+Width)+2+d;
          Shape3.Top:=ImX(Top+Height)+2+d-t;
          Shape4.Left:=ImX(Left)+2+d;
          Shape4.Top:=ImX(Top+Height)+2+d-t;
        end;
    end;
  end;
end;
```

При задании координат компонентов необходимо учитывать скроллинг компонента `Sb`, т. е. значение `Sb.VertScrollBar.Position`.

Вернемся к редактированию параметров строки или таблицы (листинг 10.17). Обработчик события `ImDblClick` поменял значение `fl_dbl` на `1`, поэтому в зависимости от типа объекта `Obj[NumSel].typ` мы вызовем `Form2` или `Form3`.

10.7. Перемещение объектов

Перемещение объектов начинается вызовом ранее описанной процедуры `ImMouseDown` (листинг 10.18) при `fl_dbl=0`. После определения номера выделенного объекта, поднимается флаг перемещения `drawing:=true`, запоминаются координаты верхнего левого угла, ширина и высота объекта и управление передается процедуре `ImMouseMove` (листинг 10.21).

Листинг 10.21. Обработка перемещения объекта

```
procedure TFormMain.ImMouseMove(Sender: TObject;
  Shift: TShiftState; X,Y: Integer);
begin
  t:=Sb.VertScrollBar.Position;
  if drawing then
  case fl_tools of
    0: begin
         Im.Canvas.DrawFocusRect(
           Rect(x0-dx,y0-dy,x0-dx+h1,y0-dy+h2));
         x0:=x-d-2; y0:=y-d-2+t;
         Im.Canvas.DrawFocusRect(
           Rect(x0-dx,y0-dy,x0-dx+h1,y0-dy+h2));
       end;
  end;
end;
```

При изменении координат снова учитывается значение вертикального скроллинга `Sb.VertScrollBar.Position`.

Завершается перемещение объекта вызовом процедуры `ImMouseUp` (листинг 10.22).

Листинг 10.22. Завершение перемещения объекта

```
procedure TFormMain.ImMouseUp(Sender: TObject;
  Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
Var L: integer;
begin
  t:=Sb.VertScrollBar.Position;
  if drawing then
  case fl_tools of
    0: begin
        Im.Canvas.DrawFocusRect(
          Rect(x0-dx,y0-dy,x0-dx+h1,y0-dy+h2));
        with Obj[NumSel] do
        case typ of
         1: begin
             L:=Im.Canvas.TextHeight(Obj[NumSel].Text);
             Left1:=Round((ImI(x0-dx)-hl)/dh)*dh+hl;
             Top1:=Round((ImJ(y0-dy+L)-hl)/dh)*dh+hl;
            end;
         2..5: begin
             Left1:=Round((ImI(x0-dx)-hl)/dh)*dh+hl;
             Top1:=Round((ImJ(y0-dy)-hl)/dh)*dh+hl;
            end;
        end;
        drawing:=false;
        SaveReport.Enabled:=True;
        ShowPage;
       end;
  end;
end;
```

В процедуре `ImMouseUp` меняются координаты левого верхнего угла (`Left1`, `Top1`) выделенного объекта.

10.8. Изменение размеров объектов

На углы выделенного объекта процедурой `ShowShape` устанавливаются четыре прямоугольных маркера выделения `TShape`. Поэтому перемещение углов начинается с использованием обработчиков событий типа `Shape3MouseDown` (листинг 10.23).

Листинг 10.23. Начало перемещения угла объекта

```
procedure TFormMain.Shape3MouseDown(Sender: TObject;
  Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  t:=Sb.VertScrollBar.Position;
  drawing:=true;
  x0:=Shape1.Left-2-d;
  y0:=Shape1.Top-2-d+t;
  dx:=Shape3.Left-2-d;
  dy:=Shape3.Top-2-d+t;
  fl_new:=1;
  Im.Canvas.DrawFocusRect(Rect(x0,y0,dx,dy));
  SetCapture(Handle);
end;
```

В данной процедуре запоминаются координаты углов выделенной фигуры, и управление передается форме `FormMain`. Перемещение указателя мыши отслеживается обработчиком события `FormMouseMove`, а номер захваченного угла определяется параметром `fl_new` (листинг 10.24).

Листинг 10.24. Перемещение угла объекта

```
procedure TFormMain.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X,Y: Integer);
begin
  t:=Sb.VertScrollBar.Position;
  if Drawing then
  case fl_new of
    1: begin  // нижний правый угол
        Im.Canvas.DrawFocusRect(Rect(x0,y0,dx,dy));
        dx:=x-2-d; dy:=y-29-2-d+t;
        Im.Canvas.DrawFocusRect(Rect(x0,y0,dx,dy));
       end;
    2: begin  // верхний левый угол
        Im.Canvas.DrawFocusRect(Rect(x0,y0,dx,dy));
        x0:=x-2-d; y0:=y-29-2-d+t;
        Im.Canvas.DrawFocusRect(Rect(x0,y0,dx,dy));
       end;
  end;
end;
```

Завершается перемещение угла обработчиком события `FormMouseUp`, в котором координаты углов изменяются до координат ближайшего узла сетки (листинг 10.25).

Листинг 10.25. Завершение перемещения угла объекта

```
procedure TFormMain.FormMouseUp(Sender: TObject;
  Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
begin
  t:=Sb.VertScrollBar.Position;
  if Drawing then
  begin
    case fl_new of
      1: begin  // нижний правый угол
           Im.Canvas.DrawFocusRect(Rect(x0,y0,dx,dy));
           with Obj[NumSel] do
           begin
             Width1:=Round((ImI(dx)-hl)/dh)*dh+hl-
               Left1;
             Height1:=Round((ImJ(dy)-hl)/dh)*dh+hl-
               Top1;
           end;
           ShowPage;
         end;
      2: begin  // верхний левый угол
           Im.Canvas.DrawFocusRect(Rect(x0,y0,dx,dy));
           with Obj[NumSel] do
           begin
             Left1:=Round((ImI(x0)-hl)/dh)*dh+hl;
             Top1:=Round((ImJ(y0)-hl)/dh)*dh+hl;
             Width1:=Round((ImI(dx)-hl)/dh)*dh+hl-
               Left1;
             Height1:=Round((ImJ(dy)-hl)/dh)*dh+hl-
               Top1;
           end;
           ShowPage;
         end;
    end;
    Drawing:=false;
    ReleaseCapture;
  end;
end;
```

10.9. Печать отчета

Печать отчета реализуется из основной программы и несколько отличается от рисования отчета на экране с помощью процедуры `ShowPage` (листинг 10.26).

Листинг 10.26. Печать отчета

```
procedure TForm8.PrintPage;
var i: integer;
begin
  if PrintDialog1.Execute then
  with Printer.Canvas do begin
    Pen.Mode:=pmCopy;
    Brush.Color:=clWhite;
    Pen.Color:=clBlack;
    Printer.BeginDoc;  // начало печати
    for i:=1 to NumObj do
    with Obj[i] do begin
      SetPrintFont(i);
      case typ of
        1: begin
             if Id=0 then s:=Obj[i].Text
                     else S:=SetField(Id);
             if f1 and (1 shl 4)<>0 then
               RectAngle(
                     PrX(Left1-1),
                     PrY(Top1)-TextHeight(s)-1,
                     PrX(Left1)+TextWidth(s)+1,
                     PrY(Top1+1) );
             TextOut(PrX(Left1),PrY(Top1)-
               TextHeight(s),s);
           end;
        2: if fl=0 then PrShowGridV(i)
                   else PrShowGridG(i);
        3: PrShowGrid3(i);
      end;
    end;
    Printer.EndDoc;  // завершение печати
  end;
end;
```

Перечислим отличия вывода отчета на принтер от вывода на экран:

1. Процедура печати использует методы канвы принтера.
2. Печать начинается методом `Printer.BeginDoc`, а заканчивается вызовом метода `Printer.EndDoc`.
3. Для масштабирования используются функции `PrX` и `PrY` (листинг 10.27).

Листинг 10.27. Функции масштабирования для принтера

```
function TForm8.PrX(x: real48): integer;
begin
```

```
  PrX:=Round(x*Printer.PageWidth/hx);
end;

function TForm8.PrY(y: real48): integer;
begin
  PrY:=Round(y*Printer.PageHeight/hy);
end;
```

4. Процедуры рисования таблиц `PrShowGridV`, `PrShowGridG` и `PrShowGrid3`, текст которых аналогичен тексту процедур `ShowGridV`, `ShowGridG` и `ShowGrid3`, также используют методы канвы принтера.
5. Если для объектов типа "строка" значение поля `Obj[i].Id` не равно `0`, то оно вычисляется процедурой `SetField` в соответствии со значением `Obj[i].Id`. Аналогично для объектов типа "таблица", если значение поля `Obj1[i].F` не равно `0`, то оно вычисляется процедурой `SetField` в соответствии со значением `Obj1[i].F`.

10.10. Заключение

На этом мы заканчиваем обсуждение проекта "Визуальный генератор отчетов", оставляя за рамками этой книги описание форм для задания свойств объектов, подключения к генератору инструментов баз данных, создание сложных древовидных заголовков столбцов, использование в отчетах графических примитивов и многие другие задачи, которые можно решать при подготовке макетов отчетов. Полный текст этого проекта находится на компакт-диске в папке **Примеры | Глава 10 | EditReport**.

Глава 11

Геометрия трехмерных тел

11.1. Платоновы тела

Простейшими трехмерными объектами являются платоновы тела. Платоновыми телами называются правильные многогранники [97], т. е. такие выпуклые многогранники, все грани которых — правильные многоугольники и все многогранные углы при вершинах равны между собой.

Существует всего пять правильных многогранников. Этот факт впервые был доказан Евклидом. Приведем его рассуждения. Пусть к каждой вершине правильного многогранника примыкает m граней и каждая из них является правильным n-угольником. Внутренний угол у каждой грани равен

$$\varphi_n = \frac{\pi(n-2)}{n},$$

а сумма внутренних углов, примыкающих к вершине, равна

$$m\varphi_n = m\frac{\pi(n-2)}{n}.$$

Поскольку телесный угол при вершине не является плоским, то из приведенных формул следует неравенство

$$m\varphi_n < 2\pi.$$

В результате получаем систему неравенств:

$$m(n-2) < 2n,$$
$$m \geq 3,$$
$$n \geq 3.$$

Эта система имеет пять целочисленных решений, соответствующих многогранникам, представленным на рис. 11.1–11.5. В подписях к этим рисункам после названия каждого многогранника приведены два числа. Первое показывает число вершин грани, второе — число ребер, сходящихся в каждую вершину.

Рис. 11.1. Тетраэдр (3,3)

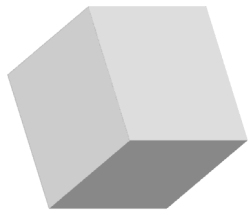


Рис. 11.2. Куб (4,3)

Рис. 11.3. Октаэдр (3,5)

Рис. 11.4. Икосаэдр (3,5)

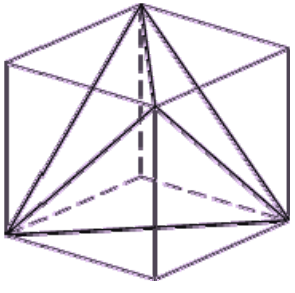
Рис. 11.5. Додекаэдр (5,3)

11.1.1. Построение платоновых тел

Рассмотрим задачи построения некоторых платоновых тел в компьютерной графике.

Тетраэдр

Хотя тетраэдр имеет всего четыре грани, каждая из которых представлена в виде правильного треугольника, вычерчивание его трехмерной проекции — непростая задача.



Простейший способ построения тетраэдра заключается в использовании куба в качестве вспомогательного тела, как показано на рис. 11.6. Сначала вычерчивается куб, выбираются нужные грани, проводятся диагонали, а затем лишние линии куба стираются. При желании куб можно поворачивать на требуемый угол.

Рис. 11.6. Построение тетраэдра

Октаэдр

На рис. 11.7 видно, что две вершины октаэдра 5 и 6 расположены по обе стороны от квадрата 1234. Предположим, что стороны квадрата 1234 имеют единичную длину. Точка 7 расположена в центре квадрата и является также центром октаэдра. Точка 8 находится в середине ребра 41. Поскольку точка 5 лежит на перпендикуляре к плоскости 1234, восстановленном в точке 7, то все, что нам надо знать, это расстояние h между этими двумя точками, которое можно вычислить из треугольника 578. Поскольку все вершины правильного многоугольника находятся на одинаковом расстоянии от центра, треугольники 157, 257, 357, 457 являются равнобедренными. Следовательно, весь октаэдр состоит из последовательности равнобедренных треугольников.

Додекаэдр

Додекаэдр (рис. 11.8) имеет 12 граней, 30 ребер, 20 вершин. Каждая из 12 граней является правильным пентагоном (пятиугольником). Додекаэдр вполне вписывается в куб, и это его свойство можно использовать для конструирования.

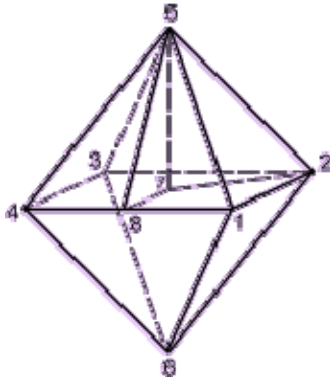


Рис. 11.7. Построение октаэдра

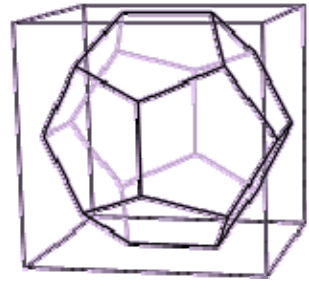


Рис. 11.8. Построение додекаэдра

11.1.2. Проект "Платоновы тела"

Проект "Платоновы тела" полностью приведен на компакт-диске в папке **Примеры | Глава 11 | Платоновы тела** и предназначен для рисования всех пяти платоновых тел: тетраэдра, гексаэдра, октаэдра, икосаэдра и додекаэдра. Проект состоит из двух форм (рис. 11.9): главной формы `FormMain` (с внешним именем **Платоновы тела**) и формы для настройки параметров `FormTools` (с внешним именем **Tools**).

В проекте реализованы следующие функции:

- выбор одного из тел;
- вращение системы координат или тела;
- рисование тела гранями или ребрами;
- изменение размера тела;
- изменение масштаба (`Zoom`);
- окрашивание граней полутонами;

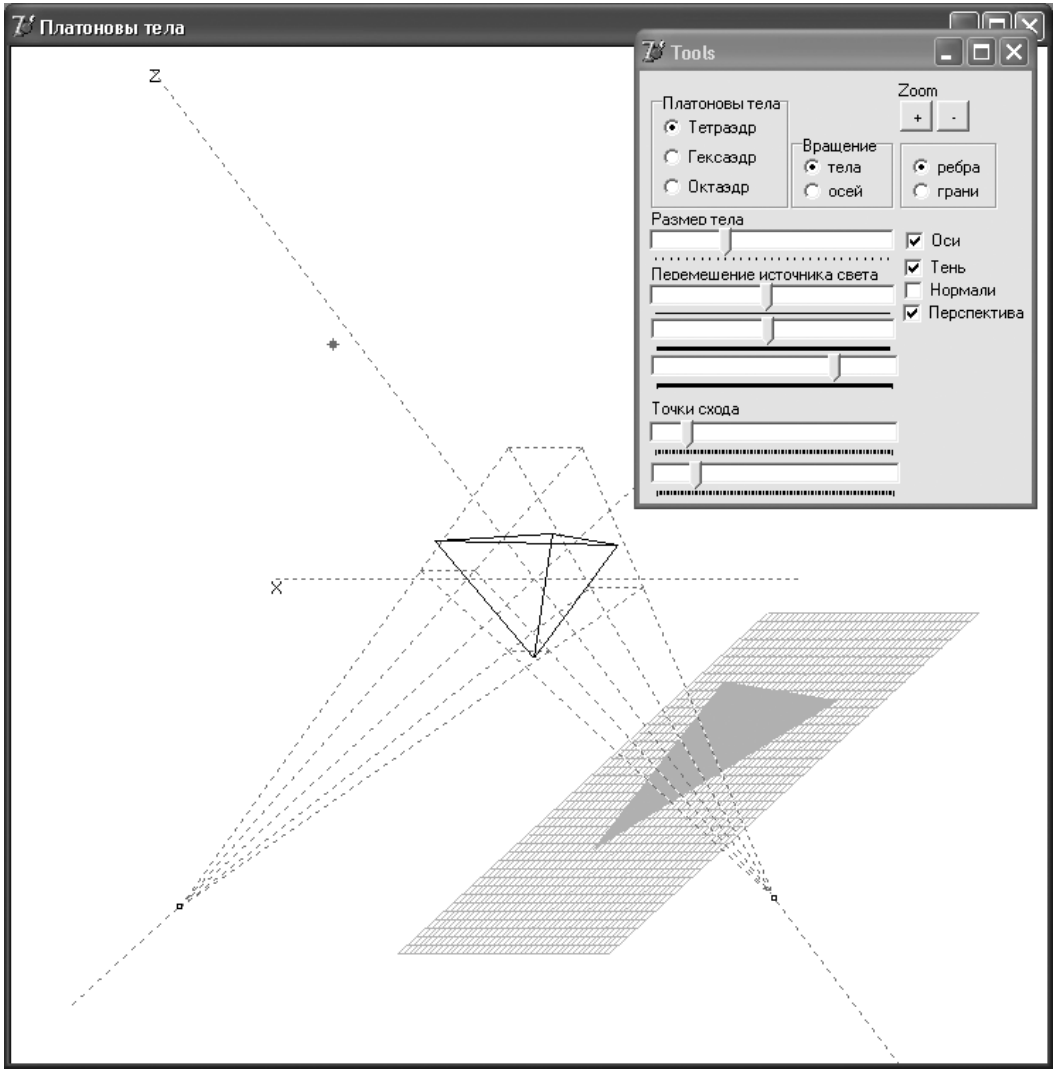


Рис. 11.9. Проект "Платоновы тела"

- рисование тени на плоскости z=const;
- изменение положения точек схода на осях OZ и OY.

Структура данных

В проекте используется следующая структура данных.

Тип TVector = array[1..4] of Real; описывает трехмерные точки.

Структура TBody, описывающая тело, содержит массив вершин Vertexs: array of Tvector, массив ребер Edges: array of TEdge и массив граней Sides: array of TSide.

```
TBody=record
  Vertexs  : array of TVector;  // массив вершин
  VertexsT : array of TVector;  // вспомогательный массив вершин
  Edges    : array of TEdge;    // массив ребер
  Sides    : array of TSide;    // массив граней
end;
```

Структура TSide описывает грани.

```
TSide =record
  p: array of word;          // номера вершин
  A,B,C,D: single;           // коэффициенты уравнения плоскости
  N: TVector;
end;
```

Поля массива p: array of word типа TSide показывают номера вершин грани, поля A, B, C, D, N используются для вычисления коэффициентов уравнения плоскости и нормали.

Следующий тип предназначен для описания ребер:

```
TEdge=record
  p1,p2: word;                  // номера вершин
end;
```

Два поля p1 и p2 переменных этого типа показывают номера вершин ребер.

> ***ЗАМЕЧАНИЕ***
>
> Такая структура данных, во-первых, избыточна (ребра, например, можно вычислить по грани), во-вторых, не является единственно возможным вариантом решения данной задачи. Но обсуждение этого вопроса мы отложим до *главы 12*.

В листинге 11.1 приведена функция, инициализирующая данные для тетраэдра, который опирается на 4 вершины, имеет 6 ребер и 4 грани. Существенным моментом при задании номеров вершин, на которые опирается грань, является порядок их обхода. Это сказывается при вычислении нормали к грани и, в зависимости от порядка обхода, нормаль может быть либо внешней к телу, либо внутренней.

Листинг 11.1. Задание данных для тетраэдра

```
function Tetrahedron(Size: real): TBody;
var
  i,j : Integer;
begin { Tetrahedron }
  with Result do begin
    SetLength(Vertexs,4);
    SetLength(VertexsT,4);
    for i:=0 to 3 do
    for j:=0 to 2 do Vertexs[i][j+1]:= Size*TetData[i,j];
    SetLength(Sides,4);
    for i:=0 to 3 do begin
      SetLength(Sides[i].p,3);
      for j:=0 to 2 do
        Sides[i].p[j]:=TetIndex[i,j];
    end;
    Edges:=GetEdges(Sides);
  end;
end; { Tetrahedron }
```

Массив `TetData` задает четыре вершины тетраэдра в единичном кубе.

```
  TetT = 1.73205080756887729;
  TetData :
    array[0..3] of TAReal =
      ( ( TetT,  TetT,  TetT),
        ( TetT, -TetT, -TetT),
        (-TetT,  TetT, -TetT),
        (-TetT, -TetT,  TetT) );
```

Массив `TetIndex` задает номера вершин четырех граней.

```
  TetIndex :
    array[0..3] of TAInteger =
      (  (0, 1, 3),
         (2, 1, 0),
         (3, 2, 0),
         (1, 2, 3)  );
```

Функция `GetEdges` формирует массив ребер по граням, перебирая для каждой грани все ребра и включая в массив только те из них, которые ранее в него не входили.

```
function GetEdges(var Faces: array of TFace): TAEdge;
var
  Ls,Le,L,i,j,k: integer;
  Ok: boolean;
begin
  Ls:=Length(Faces); Le:=0;
  for i:=0 to Ls-1 do
```

```
  with Faces[i] do begin
    L:=Length(p);
    for j:=0 to L-1 do begin
      Ok:=false; k:=-1; // проверяем вхождение ребра
      while (k<Le-1) and not Ok do begin
        Inc(k);
        Ok:=((Result[k].p1=p[j]) and
             (Result[k].p2=p[(j+1) mod L])) or
            ((Result[k].p2=p[j]) and
             (Result[k].p1=p[(j+1) mod L]))
      end;
      // если ребро не входило, то добавляем его
      if not Ok then begin
        Inc(Le); SetLength(Result,Le);
        Result[Le-1].p1:=p[j];
        Result[Le-1].p2:=p[(j+1) mod L];
      end;
    end;
  end;
end;
```

Функции инициализации данных для остальных платоновых тел приведены в модуле Lib.pas на компакт-диске в папке **Примеры | Глава 11 | Платоновы тела**.

Рисование тел

Рисование тела происходит на канве `Bitmap` в двух режимах: рисование ребер и рисование граней (листинг 11.2).

Листинг 11.2. Рисование платоновых тел

```
procedure TFormMain.Draw;
var
  i,j,L,L0,L1: integer;
  Vt,NN: TVector;
  Vn,Wn: array[0..2] of TVector;
  P0,P: TPoint;
  w: array of TPoint;
begin
  with Bitmap.Canvas, Body do begin
    Brush.Color:=clWhite;
    Pen.Color:=clBlack;
    Rectangle(0,0,Width,Height);
    Pen.Mode:=pmCopy;
```

```
// плоскость проекции тени
if FormTools.CheckBox1.Checked then begin
  …
end;

L:=Length(Vertexs);
for i:=0 to L-1 do begin
  VertexsT[i]:=Rotate(Vertexs[i], 1, Alf1);
  VertexsT[i]:=Rotate(VertexsT[i], 2, Bet1);
  VertexsT[i]:=Rotate(VertexsT[i], 1, Alf);
  VertexsT[i]:=Rotate(VertexsT[i], 2, Bet);
end;

// тень
if FormTools.CheckBox1.Checked then begin
   …
end;

// Оси
if FormTools.CheckBox2.Checked then begin
  …
end;

// Точки схода
if FormTools.CheckBox4.Checked then begin
  …
end;

// Тело
case FormTools.RgEdgesSides.ItemIndex of
  0: begin  // построение ребрами
       L:=Length(Edges);
       for i:=0 to L-1 do begin
         MoveTo(II(VertexsT[Edges[i].p1][1]),
                JJ(VertexsT[Edges[i].p1][2]));
         LineTo(II(VertexsT[Edges[i].p2][1]),
                JJ(VertexsT[Edges[i].p2][2]));
       end;
     end;
  1: begin  // построение гранями
       Brush.Color:=clSilver;
       Pen.Color:=clSilver;
       L1:=Length(Faces);
       L0:=Length(Faces[0].p);
       SetLength(w,L0);
```

```
          for i:=0 to L1-1 do begin
            for j:=0 to L0-1 do begin
              Vt:=Vertexs[Faces[i].p[j]];
              Vt:=Rotate(Vt, 1, Alf1);
              Vt:=Rotate(Vt, 2, Bet1);
              if j<=2 then Vn[j]:=Vt;
              Vt:=Rotate(Vt, 4,   0,Xs,Zs);
              Vt:=Rotate(Vt, 1, Alf);
              Vt:=Rotate(Vt, 2, Bet);
              w[j].X:=II(Vt[1]);
              w[j].Y:=JJ(Vt[2]);
              if j<=2 then Wn[j]:=Vt;
            end;
            Faces[i].N:=Lib.Norm(Vn[0],Vn[1],Vn[2]);
            if Faces[i].n[3]>=0 then
              Brush.Color:=Trunc(255* Faces[i].n[3])*$010000
            else
              Brush.Color:=Trunc(-255* Faces[i].n[3]/3)*$010000;
            Pen.Color:=Brush.Color;
            NN:=Lib.Norm(Wn[0],Wn[1],Wn[2]);
            if NN[3]<0 then
              Polygon(w);
            // нормаль
            if FormTools.CheckBox3.Checked then begin
              Pen.Color:=clRed;
              MoveTo(w[0].X,w[0].Y);
              LineTo(II(Wn[0][1]+NN[1]),JJ(Wn[0][2]+NN[2]));
            end;
          end;
        end;
    end; // case: ребра/грани

    // Источник света
    if FormTools.CheckBox1.Checked then begin
      …
    end;
  end;  // Bitmap.Canvas
  Canvas.Draw(0,0,Bitmap);
end;
```

Массив вершин `Vertexs` задает тело в неподвижной системе координат. В процессе вращения мы будем использовать вспомогательный массив `VertexsT`. При рисовании учитывается, что тело повернуто в неподвижной системе координат на углы $Alf1_x$, $Bet1_y$, а сама система координат повернута на углы Alf_x, Bet_y. Между

двумя этими поворотами осуществляется перспективное проецирование. Поворот реализуется процедурой `Rotate` следующим образом:

```
L:=Length(Vertexs);
for i:=0 to L-1 do begin
  VertexsT[i]:=Rotate(Vertexs[i],  1, Alf1,0,0);
  VertexsT[i]:=Rotate(VertexsT[i], 2, Bet1,0,0);
  VertexsT[i]:=Rotate(VertexsT[i], 4, 0,Xs,Zs);
  VertexsT[i]:=Rotate(VertexsT[i], 1, Alf,0,0);
  VertexsT[i]:=Rotate(VertexsT[i], 2, Bet,0,0);
end;
```

Код функции `Rotate` представлен в листинге 11.3.

Листинг 11.3. Поворот вокруг осей координат и проецирование

```
function Rotate(var V: TVector; k: Integer; fi: Real): TVector;
var
  i: Integer;
  M: TMatrix;
begin
  for i:=1 to 3 do begin
    M[4,i]:=0; M[i,4]:=0;
  end;
  M[4,4]:=1;
  case k of
    1:begin  // Матрица поворота вокруг оси OX
        M[1,1]:=1; M[1,2]:=0;        M[1,3]:=0;
        M[2,1]:=0; M[2,2]:= cos(fi); M[2,3]:=sin(fi);
        M[3,1]:=0; M[3,2]:=-sin(fi); M[3,3]:=cos(fi);
      end;
    2:begin  // Матрица поворота вокруг оси OY
        M[1,1]:=cos(fi); M[1,2]:=0; M[1,3]:=-sin(fi);
        M[2,1]:=0;       M[2,2]:=1; M[2,3]:=0;
        M[3,1]:=sin(fi); M[3,2]:=0; M[3,3]:=cos(fi);
      end;
    3:begin // Матрица поворота вокруг оси OZ
        M[1,1]:= cos(fi); M[1,2]:=sin(fi); M[1,3]:=0;
        M[2,1]:=-sin(fi); M[2,2]:=cos(fi); M[2,3]:=0;
        M[3,1]:= 0;       M[3,2]:=0;       M[3,3]:=1;
      end;
    4:begin // двухточечное перспективное проецирование
        M[1,1]:= 1; M[1,2]:=0; M[1,3]:=0;
        M[2,1]:= 0; M[2,2]:=1; M[2,3]:=0;
```

```
        M[3,1]:= 0; M[3,2]:=0; M[3,3]:=1;
        M[2,4]:= p; M[3,4]:=r;
      end;
  end;
  Result:=VM_Mult(V,M); // Выполняем поворот на угол fi
end;
```

Процедура умножения четырехмерного вектора на матрицу реализуется функцией `VM_Mult` (листинг 11.4).

Листинг 11.4. Умножение четырехмерного вектора на матрицу

```
function VM_Mult(var A: TVector; var B: TMatrix): TVector;
var     // Процедура умножения четырехмерного вектора на матрицу
  j,k: integer;
begin
  for j:=1 to 4 do begin
    Result[j]:=A[1]*B[1,j];
    for k:=2 to 4 do
      Result[j]:=Result[j]+A[k]*B[k,j];
  end;
  for j:=1 to 3 do Result[j]:=Result[j]/Result[4];
  Result[4]:=1;
end;
```

Если тело рисуется ребрами, то необходимо в цикле перебрать все ребра и нарисовать отрезки от первой точки ребра до второй, получая доступ к вершинам следующим образом: `VertexsT[Edges[i].p1]`.

При рисовании тела гранями, необходимо в цикле (`i=0,..,L-1`) перебрать все грани. Для каждой грани в цикле (`j=0,..,L0-1`) необходимо перебрать все вершины. Доступ к вершинам реализуется через номера вершин грани `VertexsT[Sides[i].p[j]]`.

Для каждой грани нам потребуются два единичных вектора нормали: вектор нормали `Sides[i].N` к грани повернутого тела в неподвижной системе координат — для вычисления освещенности и вектор нормали `NN` к грани повернутого тела в повернутой системе координат — для вычисления невидимых граней.

Для вычисления этих нормалей потребуются три вершины грани в неподвижной системе координат `Vn: array[0..2] of Tvector` и три вершины грани в повернутой системе координат `Wn: array[0..2] of TVector`.

Нормали вычисляются функцией `Norm`.

```
  Sides[i].N:=Norm(Vn[0],Vn[1],Vn[2]);
  NN:=Lib.Norm(Wn[0],Wn[1],Wn[2]);
```

Текст функции Norm приведен в листинге 11.5.

Листинг 11.5. Вычисление вектора нормали

```
function Norm(V1,V2,V3: TVector): TVector;
// Вычисление нормали к грани многогранника
var
  A,B: TVector;
  u,v,w,d: real;
begin
  A[1]:=V2[1]-V1[1]; A[2]:=V2[2]-V1[2]; A[3]:=V2[3]-V1[3];
  B[1]:=V3[1]-V1[1]; B[2]:=V3[2]-V1[2]; B[3]:=V3[3]-V1[3];
  u:=A[2]*B[3]-A[3]*B[2];
  v:=-A[1]*B[3]+A[3]*B[1];
  w:=A[1]*B[2]-A[2]*B[1];
  d:=Sqrt(u*u+v*v+w*w);
  if d<>0 then begin
    Result[1]:=u/d; Result[2]:=v/d; Result[3]:=w/d;
  end
  else begin
    Result[1]:=0; Result[2]:=0; Result[3]:=0;
  end;
end;
```

Использование вектора нормали NN для отсечения невидимых граней, у которых проекция этого вектора на ось Z отрицательна, сводится к проверке:

```
  if NN[3]<0 then Polygon(w).
```

Использование вектора нормали Sides[i].N требует дополнительных пояснений, которые будут даны далее (см. листинг 11.6).

Рисование полутонами

Напомним, что класс цвета точки TColor определен как Longint: TColor = - $80000000 ..$7FFFFFFF. Четыре байта переменных этого типа содержат информацию о долях синего (B), зеленого (G) и красного (R) цветов и устроены следующим образом: $00BBGGRR. Поэтому для рисования полутонами синего цвета, например, необходимо определить некую целочисленную функцию f(x) с диапазоном изменения [0..255] и цвет определять соотношением Color:=f(x)*$00010000.

Будем считать, что выбор цвета грани определяется длиной проекции на ось Z единичной нормали Sides[i].N в неподвижной системе координат и реализуется следующими операторами (листинг 11.6).

Листинг 11.6. Вычисление цветов

```
if Sides[i].n[3]>=0 then
  Brush.Color:=Trunc(255*Sides[i].n[3])*$010000
else
  Brush.Color:=Trunc(-255*Sides[i].n[3]/3)*$010000;
```

Введем для определенности следующие предположения:

1. Источник света находится в бесконечности на оси Z.
2. Грани, не видимые из источника света, освещаются в 3 раза слабее, чем видимые.

Второе предположение весьма условно, но позволяет получить объемное изображение.

Достаточно просто решить более общую задачу, когда источник находится в бесконечности на прямой, проходящей через любую точку и центр тела. В этом случае необходимо найти через скалярное произведение проекцию вектора нормали к грани на эту прямую.

Построение тени

Будем считать, что:

- источник света находится в точке с координатами `Sv: TVector=(-0.5, -0.5,5)`. Впрочем, эти координаты можно изменять на форме `FormToos`;
- тело отбрасывает тень на плоскость `z=const`. На рис. 11.10 изображена одна из граней тела.

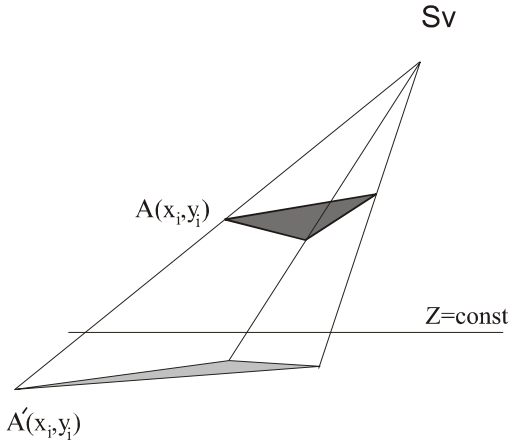


Рис. 11.10. Проекция грани на плоскость Z

Для нахождения координат точки A' на плоскости $z = z_h$ необходимо записать условие колинеарности двух векторов

$$\begin{vmatrix} i & j & k \\ x_i - x_0 & y_i - y_0 & z_i - z_0 \\ x - x_0 & y - y_0 & z_i - z_0 \end{vmatrix} = 0, \tag{11.1}$$

что приводит к системе уравнений для определения x и y:

$$\begin{aligned} &(y_i - y_0)(z_h - z_0) - (z_i - z_0)(y_i - y_0) = 0, \\ &(x_i - x_0)(z_h - z_0) - (x - z_0)(z_i - z_0) = 0, \\ &(x_i - x_0)(y - y_0) - (x - x_0)(y_i - y_0) = 0. \end{aligned} \tag{11.2}$$

Решая уравнения (11.2) относительно переменных x и y, получаем

$$\begin{aligned} y &= y_0 + \frac{(y_i - y_0)(z_h - z_0)}{z_i - z_0}, \\ x &= x_0 + \frac{(x_i - x_0)(z_h - z_0)}{z_i - z_0}. \end{aligned} \tag{11.3}$$

Соотношения (11.3) используются при построении тени от многогранника следующим образом: переберем все грани тела в цикле `i=0,...,L1-1`. Для каждой грани переберем все вершины в цикле `j=0,...,L0-1`. Вершину повернем вместе с телом на углы `Alf1` и `Bet1`. Затем по формулам (11.3) найдем координаты точки `Vt` на плоскости проекции. Повернем систему координат на углы `Alf` и `Bet`. Наконец, с помощью функций `II()` и `JJ()` найдем координаты образа точки на экране. После вычисления всех таких точек методом канвы `Polygon` строится тень от одной грани тела (листинг 11.7).

Листинг 11.7. Построение тени от многогранника

```
Brush.Color:=clSilver;
Pen.Color:=clSilver;
L1:=Length(Faces);
L0:=Length(Faces[0].p);
SetLength(w,L0);
for i:=0 to L1-1 do begin
  for j:=0 to L0-1 do begin
    Vt:=Vertexs[Faces[i].p[j]];
    Vt:=Rotate(Vt, 1, Alf1);
    Vt:=Rotate(Vt, 2, Bet1);
    Vt:=Rotate(Vt, 4,   0,Xs,Zs);
    Vt[1]:=Sv[1]+(Vt[1]-Sv[1])*(Zh-Sv[3])/(Vt[3]-Sv[3]);
    Vt[2]:=Sv[2]+(Vt[2]-Sv[2])*(Zh-Sv[3])/(Vt[3]-Sv[3]);
```

```
      Vt[3]:=Zh;
      Vt:=Rotate(Vt, 1, Alf);
      Vt:=Rotate(Vt, 2, Bet);
      w[j].X:=II(Vt[1]);
      w[j].Y:=JJ(Vt[2]);
    end;
    Polygon(w);
end;
```

Завершает построение тени рисование сетки на плоскости $z = z_h$ (листинг 11.8):

Листинг 11.8. Построение сетки на плоскости *z=Zh*

```
Pen.Color:=clSilver;
for i:=0 to 40 do begin
  P0:=IJ(ToVector(-4+i*0.2,-4,Zh));
  MoveTo(P0.X,P0.Y);
  P0:=IJ(ToVector(-4+i*0.2,4,Zh));
  LineTo(P0.X,P0.Y);
end;
for i:=0 to 40 do begin
  P0:=IJ(ToVector(-4,-4+i*0.2,Zh));
  MoveTo(P0.X,P0.Y);
  P0:=IJ(ToVector( 4,-4+i*0.2,Zh));
  LineTo(P0.X,P0.Y);
end;
```

и рисование источника света (листинг 11.9).

Листинг 11.9. Рисование источника света

```
P:=IJ(Sv);
Brush.Color:=clRed;
Pen.Color:=clRed;
RectAngle(P.X-2,P.Y-2,P.X+3,P.Y+3);
MoveTo(P.X-4,P.Y); LineTo(P.X+5,P.Y);
MoveTo(P.X,P.Y-4); LineTo(P.X,P.Y+5);
```

Напомним, что полный текст данного проекта приведен на компакт-диске в папке **Примеры | Глава 11 | Платоновы тела**.

Перспективное проецирование

Для реалистичного изображения трехмерного тела необходимо учитывать искажение его проекции в соответствии с законами перспективы.

В *главе 6* приведена общая матрица преобразования однородных координат, которая имеет вид:

$$\begin{vmatrix} a & x & x & p \\ x & d & x & q \\ x & x & e & r \\ m & n & l & s \end{vmatrix}. \qquad (11.4)$$

Элементы данной матрицы имеют следующее значение:

- a, d, e — масштабирование;
- m, n, l — смещение;
- p, q, r — перспектива;
- s — комплексное масштабирование;
- x — вращение.

Всего различают три типа перспективных преобразований:

- одноточечная перспектива $(r \neq 0)$;
- двухточечная перспектива $(q, r \neq 0)$;
- трехточечная перспектива $(p, q, r \neq 0)$.

Объединение перспективного преобразования с проецированием называется перспективным или центральным проецированием. В дальнейшем изложении подразумевается, что выполняется фронтальное проецирование.

На фотографиях, картинах, экране изображения кажутся нам естественными и правильными. Такие изображения называют перспективными. Одним из основных свойств таких изображений является то, что более удаленные предметы изображаются в меньших масштабах, параллельные прямые в общем случае непараллельны. В итоге геометрия изображения оказывается достаточно сложной, и по готовому изображению сложно определить размеры тех или иных частей объекта.

Обычная перспективная проекция — это центральная проекция на плоскость прямыми лучами, проходящими через точку, которая называется центром проецирования. Один из проецирующих лучей перпендикулярен к плоскости проецирования и называется главным. Точка пересечения этого луча и плоскости проекции называется главной точкой картины.

Матрица (11.5) соответствует одноточечному проецированию на линию (чтобы не терялись друг за другом точки линии, параллельные оси z). Глубина исчезла, и дальние предметы стали мелкими.

$$\begin{vmatrix} x & y & z & 1 \end{vmatrix} \times \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & r \\ 0 & 0 & 0 & 1 \end{vmatrix} = \begin{vmatrix} \dfrac{x}{rz+1} & \dfrac{y}{rz+1} & 0 & 1 \end{vmatrix}. \qquad (11.5)$$

Рассмотрим двухточечное проецирование (рис. 11.11). Параметры q, r определяют положение точек схода S_q и S_r на осях OY и OZ, соответственно. Напомним, что точка схода — это точка пересечения пучка прямых, полученных в результате перспективного преобразования пучка параллельных прямых.

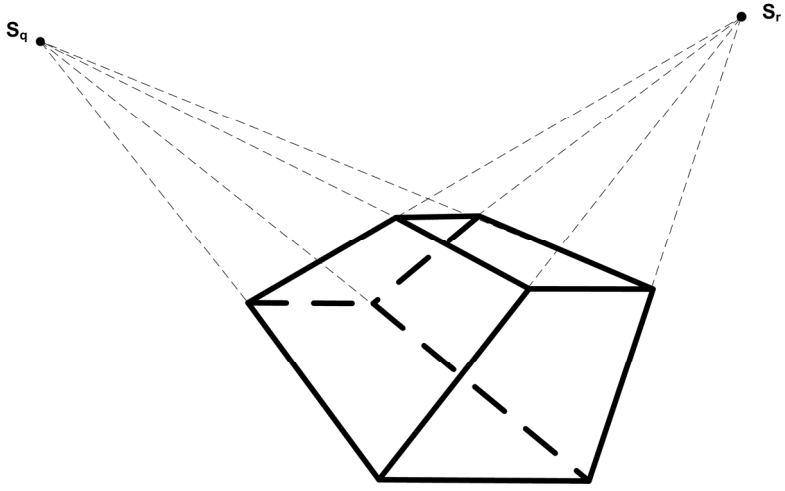


Рис. 11.11. Двухточечное проецирование

Перспективное двухточечное проецирование реализуется матрицей (листинг 11.10).

Листинг 11.10. Перспективное двухточечное проецирование

```
4:begin // перспективное проецирование
    M[1,1]:= 1; M[1,2]:=0; M[1,3]:=0; M[1,4]:=0;
    M[2,1]:= 0; M[2,2]:=1; M[2,3]:=0; M[2,4]:=p;
    M[3,1]:= 0; M[3,2]:=0; M[3,3]:=1; M[3,4]:=r;
    M[4,1]:= 0; M[4,2]:=0; M[4,3]:=0; M[4,4]:=1;
  end;
```

Эта матрица используется функцией `Rotate` следующим образом.

```
VertexsT[i]:=Rotate(Vertexs[i],  1, Alf1,0,0);
VertexsT[i]:=Rotate(VertexsT[i], 2, Bet1,0,0);
VertexsT[i]:=Rotate(VertexsT[i], 4, 0,Xs,Zs);
VertexsT[i]:=Rotate(VertexsT[i], 1, Alf,0,0);
VertexsT[i]:=Rotate(VertexsT[i], 2, Bet,0,0);
```

Стереоизображение

Напомним, что плоское изображение ребрами многогранника в повернутой системе координат реализуется циклом по всем ребрам (листинг 11.11).

Листинг 11.11. Плоское рисование ребрами

```
with Bitmap.Canvas, Body do begin
  for i:=0 to L-1 do
  with Edges[i] do begin
    MoveTo(II(VertexsT[p1][1]),JJ(VertexsT[p1][2]));
    LineTo(II(VertexsT[p2][1]),JJ(VertexsT[p2][2]));
  end
end
```

Для создания стереоизображения, на которое можно смотреть сквозь очки с красной и синей линзами, нарисуем два совмещенных изобращения, слегка повернутых вокруг оси `OY` в разные стороны на угол `AnglStereo`. Изображение красными линиями повернем на угол `AnglStereo`, а изображение синими линиями повернем на угол `-AnglStereo`. Простые расчеты показывают, что угол `AnglStereo` должен меняться в диапазоне от 0,01 до 0,03 радиан (листинг 11.12).

Листинг 11.12. Стереорисование ребрами

```
with Bitmap.Canvas, Body do begin
  Pen.Color:=clRed;  // изображение красными линиями
  Pen.Width:=WidthLine;
  for i:=0 to L-1 do
  with Edges[i] do begin
    V1:=Rotate(VertexsT[p1], 2, AnglStereo,0,0);
    MoveTo(II(V1[1]),JJ(V1[2]));
    V2:=Rotate(VertexsT[p2], 2, AnglStereo,0,0);
    LineTo(II(V2[1]),JJ(V2[2]));
  end;
  with BitmapBlue.Canvas do begin // изображение синими линиями
    Brush.Color:=clBlue;
    FillRect(Rect(0,0,Width,Height));
    Pen.Width:=WidthLine;
    Pen.Color:=clBlue;
    for i:=0 to L-1 do
    with Edges[i] do begin
      V1:=Rotate(VertexsT[p1], 2, -AnglStereo,0,0);
      MoveTo(II(V1[1]),JJ(V1[2]));
      V2:=Rotate(VertexsT[p2], 2, -AnglStereo,0,0);
      LineTo(II(V2[1]),JJ(V2[2]));
    end;
  end;
```

```
  CopyMode:=cmSrcPaint;
  Draw(0,0,BitmapBlue);  // совмещение изображений
end;
```

Красное изображение рисуется на канве `Bitmap`, синее — на канве `BitmapBlue`. Затем `BitmapBlue` копируется на `Bitmap` в режиме `cmSrcPaint`, в котором цвета точек `Bitmap` и `BitmapBlue` складываются операцией `OR`. В результате общие точки изображений имеют цвет `$FF00FF`.

На рис. 11.12 показан результат работы программы, рисующей стереоизображение многогранника.

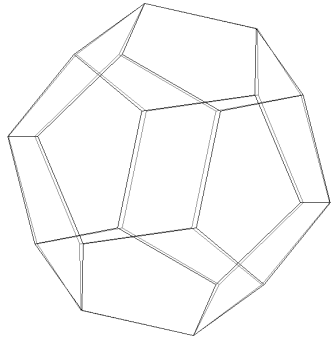


Рис. 11.12. Стереоизображение многогранника

11.2. Квадратичные поверхности

В общем случае уравнения поверхностей, заданные полиномами второй степени, имеют вид [97]:

$$Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz + k = 0.$$

После приведения этого уравнения к каноническому виду число различных квадратичных поверностей существенно сокращается. В *разделе 11.2.1* приведены все типы таких поверхностей.

11.2.1. Уравнения квадратичных поверхностей в явной форме

Приведем уравнения квадратичных поверхностей в явной форме, используя переменные x, y, z и параметры a, b, c, имеющие, например, для эллипсоида смысл полуосей.

1. Действительный эллипсоид — невырожденная центральная поверхность (рис. 11.13).

$$\frac{x^2}{a^2}+\frac{y^2}{b^2}+\frac{z^2}{c^2}-1=0.$$

2. Однополостный гиперболоид — невырожденная центральная поверхность (рис. 11.14).

$$\frac{x^2}{a^2}-\frac{y^2}{b^2}+\frac{z^2}{c^2}-1=0.$$

3. Двуполостный гиперболоид — невырожденная центральная поверхность (рис. 11.15).

$$\frac{x^2}{a^2}-\frac{y^2}{b^2}+\frac{z^2}{c^2}+1=0.$$

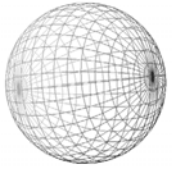


Рис. 11.13. Эллипсоид

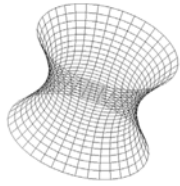


Рис. 11.14. Однополостный гиперболоид

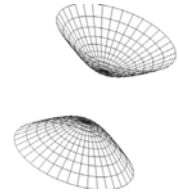


Рис. 11.15. Двуполостный гиперболоид

4. Эллиптический конус — вырожденная центральная поверхность (рис. 11.16).

$$\frac{x^2}{a^2}-\frac{y^2}{b^2}+\frac{z^2}{c^2}=0.$$

5. Гиперболический параболоид — невырожденная нецентральная поверхность (рис. 11.17).

$$\frac{x^2}{a^2}-\frac{y}{b}-\frac{z^2}{c^2}=0.$$

6. Эллиптический параболоид — невырожденная нецентральная поверхность (рис. 11.18).

$$\frac{x^2}{a^2}-\frac{y}{b}+\frac{z^2}{c^2}=0.$$

7. Эллиптический цилиндр — вырожденная нецентральная поверхность (рис. 11.19).

$$\frac{x^2}{a^2}+\frac{z^2}{c^2}-1=0.$$

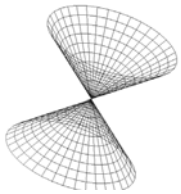


Рис. 11.16. Эллиптический конус

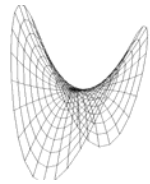


Рис. 11.17. Гиперболический параболоид

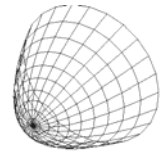


Рис. 11.18. Эллиптический параболоид

8. Параболический цилиндр — вырожденная нецентральная поверхность (рис. 11.20).

$$\frac{x^2}{a^2} - \frac{z}{c} = 0.$$

9. Гиперболический цилиндр — вырожденная нецентральная поверхность (рис. 11.21).

$$\frac{x^2}{a^2} - \frac{z^2}{c^2} - 1 = 0.$$

10. Тор — поверхность четвертого порядка (рис. 11.22).

$$(x^2 + y^2 + z^2 + R^2 - r^2)^2 - 4R^2(x^2 + y^2) = 0.$$

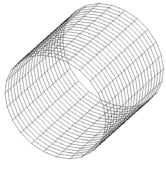


Рис. 11.19. Эллиптический цилиндр

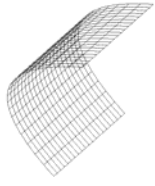


Рис. 11.20. Параболический цилиндр

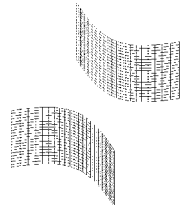


Рис. 11.21. Гиперболический цилиндр

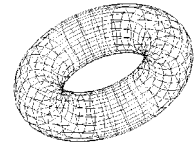


Рис. 11.22. Тор

11.2.2. Параметрическое представление квадратичных поверхностей

В данном разделе представлены уравнения в параметрической форме, как функции от параметров t и τ, для тех же поверностей, что и в *разд. 11.2.1*.

Для каждого параметра указан диапазон изменения.

1. Действительный эллипсоид — невырожденная центральная поверхность.

$$p(t,\tau)=[a\times\sin(t)\sin(\tau),\quad b\times\cos(t),\quad c\times\sin(t)\cos(\tau)],$$
$$t\in[0,\pi],\quad \tau\in[0,2\pi].$$

2. Однополостный гиперболоид — невырожденная центральная поверхность.

$$p(t,\tau)=[a\times ch(t)\sin(\tau),\quad b\times sh(t),\quad c\times ch(t)\cos(\tau)],$$
$$t\in(-\infty,\infty),\quad \tau\in[0,2\pi].$$

3. Двуполостный гиперболоид — невырожденная центральная поверхность.

$$p(t,\tau)=[a\times sh(t)\sin(\tau),\quad \pm b\times ch(t),\quad c\times sh(t)\cos(\tau)],$$
$$t\in[0,\infty),\quad \tau\in[0,2\pi].$$

4. Эллиптический конус — вырожденная центральная поверхность.

$$p(t,\tau)=[a\times t\times\sin(\tau),\quad b\times t,\quad c\times t\times\cos(\tau)],$$
$$t\in(-\infty,\infty),\quad \tau\in[0,2\pi].$$

5. Гиперболический параболоид — невырожденная нецентральная поверхность.

$$p(t,\tau)=[a\times t\times\sin(\tau),\quad -b\times t^2\times\cos(2\tau),\quad c\times t\times\cos(\tau)],$$
$$t\in[0,\infty),\quad \tau\in[0,2\pi].$$

6. Эллиптический параболоид — невырожденная нецентральная поверхность.

$$p(t,\tau)=[a\times t\times\sin(\tau),\quad b\times t^2,\quad c\times t\times\cos(\tau)],$$
$$t\in[0,\infty),\quad \tau\in[0,2\pi].$$

7. Эллиптический цилиндр — вырожденная нецентральная поверхность.

$$p(t,\tau)=[a\times\sin(\tau),\quad b\times t,\quad c\times\cos(\tau)],$$
$$t\in(-\infty,\infty),\quad \tau\in[0,2\pi].$$

8. Параболический цилиндр — вырожденная нецентральная поверхность.

$$p(t,\tau)=[a\times\tau,\quad b\times t,\quad c\times\tau^2],$$
$$t\in(-\infty,\infty),\quad \tau\in(-\infty,\infty).$$

9. Гиперболический цилиндр — вырожденная нецентральная поверхность.

$$p(t,\tau)=[a\times sh(\tau),\quad b\times t,\quad c\times ch(\tau)],$$
$$t\in(-\infty,\infty),\quad \tau\in(-\infty,\infty).$$

10. Тор — поверхность четвертого порядка.

$$p(t,\tau)=[(R+r\times\cos(\tau))\times\cos(t),\quad (R+r\times\cos(\tau))\times\sin(t),\quad r\times\sin(\tau)],$$
$$t\in[0,2\pi],\quad \tau\in[0,2\pi].$$

11.2.3. Проект "Квадратичные поверхности"

Проект "Квадратичные поверхности" предназначен для рисования десяти поверхностей: эллипсоида, однополостного гиперболоида, двуполостного гиперболоида, эллиптического конуса, гиперболического параболоида, эллиптического параболоида, эллиптического цилиндра, параболического цилиндра, гиперболического цилиндра и тора и состоит из двух форм (рис. 11.23): главной формы `FormMain` (с внешним именем **Квадратичные поверхности**) и формы для настройки параметров `FormTools` (с внешним именем **Tools**). В проекте реализованы следующие функции:

- выбор одной из поверхностей;
- вращение системы координат;
- рисование поверхности гранями или ребрами;
- изменение размера поверхности;
- изменение масштаба (`Zoom`);

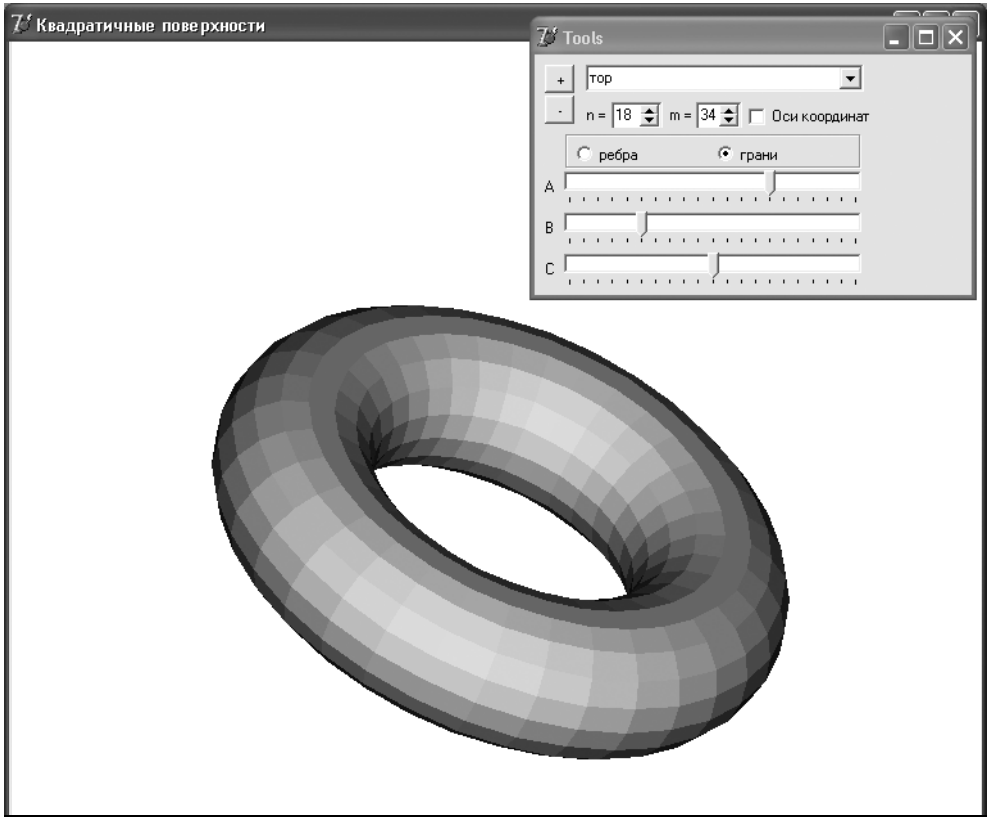


Рис. 11.23. Проект "Квадратичные поверхности"

- окрашивание граней полутонами;
- сортировка граней перед рисованием.

Структура данных

В проекте используется следующая структура данных.

Тип:

```
TVector=record
  x,y,z: real;
end;
```

описывает трехмерные точки.

Структура `TSide` описывает грани.

```
TSides=record
  P: array[0..3] of TVector;
  N: real;
  Zmin: real;
end;
```

Массив `p: array[0..3] of TVector` содержит координаты четырех вершин грани, поле `N` используется для хранения проекции нормали на ось `Z`, поле `Zmin` — для хранения минимальной координаты `z`. По полю `Zmin` идет сортировка всех граней перед рисованием.

> ***ЗАМЕЧАНИЕ***
>
> Такая структура данных избыточна: почти каждая вершина при повороте системы координат обрабатывается четырежды, т. к. входит в четыре соседние грани.

Для вычисления координат точек на поверхностях используется параметрическая функция `XYZ` (листинг 11.13).

Листинг 11.13. Функция вычисления координат точек на поверхностях

```
function XYZ(n: integer; t1,t2: real): TVector;
begin
  case n of
    0: begin   // эллипсоид
         Result.x:=a*Sin(t1)*Sin(t2);
         Result.y:=b*Cos(t1);
         Result.z:=c*Sin(t1)*Cos(t2);
       end;
    1: begin   // однополостной гиперболоид
         Result.x:=a*Ch(t1)*Sin(t2);
         Result.y:=b*Sh(t1);
```

```
        Result.z:=c*Ch(t1)*Cos(t2);
      end;
    …
  end;
end;
```

При выборе одной из поверхностей необходимо задать диапазоны изменения параметров (t, τ) и коэффициенты a, b, c. Это задание реализует процедура `InitData`, код которой представлен в листинге 11.14.

Листинг 11.14. Задание параметров поверхности

```
procedure InitData(k: integer);
begin
  sg:=0;
  a:=1; b:=1; c:=1;
  n:=20; m:=36;
  a1:=-1; a2:=1;
  b1:=0; b2:=2*Pi;
  case k of
    0: begin  // эллипсоид
         a1:=0; a2:=2*Pi;
         n:=36;
       end;
    1: ;      // однополостной гиперболоид
    2: begin  // двуполостной гиперболоид
         sg:=1;
         a1:=0; a2:=3;
       end;
    …
  end;
end;
```

В этой функции: `a`, `b`, `c` — параметры поверхности; `a1`, `a2` — диапазон изменения первого параметра `t`; `b1`, `b2` — диапазон изменения второго параметра τ; `n`, `m` — число разбиений интервалов; параметр `sg` может принимать значение `0` для однозначных поверхностей и значение `1` для двузначных поверхностей типа двуполостного гиперболоида. Эти параметры могут изменяться с помощью компонентов формы `FormTools`.

Рисование поверхностей

Рисование поверхностей происходит на канве `Bitmap` в двух режимах: с закраской и без закраски граней (листинг 11.15).

Листинг 11.15. Рисование поверхностей

```
procedure TFormMain.Show3D;
var i,j,k,q,L: integer;
    t1,t2,h1,h2: real;
    P: TVector;
    w: array[0..3] of TPoint;
begin
  with Bitmap.Canvas do begin
    Brush.Color:=clWhite;
    Brush.Style:=bsSolid;
    RectAngle(-1,-1,Width,Height);

    // Оси
    if FormTools.CheckBox1.Checked then begin
      …
    end;

    h1:=(a2-a1)/n; h2:=(b2-b1)/m;

    k:=FormTools.ComboBox1.ItemIndex;
    if sg=0 then L:=n*m else L:=2*n*m;
    SetLength(Sides,L);
    q:=-1;
    if k<>-1 then begin
      for j:=0 to n-1 do
        for i:=0 to m-1 do begin
          Inc(q);
          t1:=a1+h1*j; t2:=b1+h2*i;
          Sides[q].P[0]:=Rotate(0,XYZ(k,t1,t2));
          t1:=a1+h1*j; t2:=b1+h2*(i+1);
          Sides[q].P[1]:=Rotate(0,XYZ(k,t1,t2));
          t1:=a1+h1*(j+1); t2:=b1+h2*(i+1);
          Sides[q].P[2]:=Rotate(0,XYZ(k,t1,t2));
          t1:=a1+h1*(j+1); t2:=b1+h2*i;
          Sides[q].P[3]:=Rotate(0,XYZ(k,t1,t2));
          Polygon(w);

          if sg<>0 then begin
            t1:=a1+h1*j; t2:=b1+h2*i;
            Sides[q+n*m].P[0]:=Rotate(1,XYZ(k,t1,t2));
            t1:=a1+h1*j; t2:=b1+h2*(i+1);
            Sides[q+n*m].P[1]:=Rotate(1,XYZ(k,t1,t2));
            t1:=a1+h1*(j+1); t2:=b1+h2*(i+1);
            Sides[q+n*m].P[2]:=Rotate(1,XYZ(k,t1,t2));
```

```
            t1:=a1+h1*(j+1); t2:=b1+h2*i;
            Sides[q+n*m].P[3]:=Rotate(1,XYZ(k,t1,t2));
          end;
        end;

      case FormTools.RadioGroup1.ItemIndex of
        0: begin  // ребра
             Brush.Style:=bsClear;
             Pen.Color:=0;
             for i:=0 to L-1 do
             with Sides[i] do begin
               for j:=0 to 3 do w[j]:=IJ_(P[j]);
               Polygon(w)
             end;
           end;
        1: begin   // грани
              for i:=0 to L-1 do
              with Sides[i] do begin
                N:=Norm(P[0],P[1],P[2]);
                Zmin:=Z_Min(Sides[i]);
              end;
              SortShell;
              Brush.Style:=bsSolid;
              for i:=0 to L-1 do
              with Sides[i] do begin
                for j:=0 to 3 do w[j]:=IJ_(P[j]);
                Brush.Color:=Trunc(255*Abs(N))*$010100;
                Pen.Color := Brush.Color;
                Polygon(w)
              end;
           end;
      end; // case
    end;
  end;
  Canvas.Draw(0,0,Bitmap);
end;
```

На первом этапе в повернутой системе координат в динамическом массиве `Sides: array of TSides` накапливаются координаты четырех вершин граней:

`Sides[q].P[0]:=Rotate(0,XYZ(k,t1,t2))`.

Длина этого массива равна `n*m` для однозначных поверхностей и `2*n*m` для двузначных поверхностей.

Функция поворота системы координат `Rotate` может менять знак у координаты `Y`, что необходимо для двузначных поверхностей (листинг 11.16).

Листинг 11.16. Функция поворота системы координат

```
function Rotate(fl: byte; P: TVector): TVector;
begin
  if fl<>0 then P.y:=-P.y;
  Result.x:=(P.x-x0)*cos(alf)-(P.y-y0)*sin(alf);
  Result.y:=((P.x-x0)*sin(alf)+(P.y-y0)*cos(alf))*cos(Bet)-
    (P.z-z0)*sin(Bet);
  Result.z:=((P.x-x0)*sin(alf)+(P.y-y0)*cos(alf))*sin(Bet)+
    (P.z-z0)*cos(Bet);
end;
```

На втором этапе при рисовании без закраски граней после преобразования к экранным координатам выводим все четырехугольники граней в режиме прозрачности `Brush.Style:=bsClear`.

При рисовании с закраской граней необходимо сначала для всех граней вычислить проекцию нормали на ось `Z` и `Zmin`.

```
for i:=0 to L-1 do
with Sides[i] do begin
  N:=Norm(P[0],P[1],P[2]);
  Zmin:=Z_Min(Sides[i]);
end;
```

Затем необходимо отсортировать грани по `Zmin`. Для сортировки используется алгоритм Шелла (листинг 11.17).

Листинг 11.17. Сортировка Шелла

```
procedure SortShell;
var
  i,j,k,step,N: integer;
  Tmp: TSides;
begin
  N:=Length(Sides);
  step:=N div 2; // первый шаг
  while step>=1 do begin
    k:=step;
    for i:=k+0 to N-1 do begin
      tmp:=Sides[i]; j:=i-k;
      while (j>0) and (tmp.Zmin<Sides[j].Zmin) do begin
        Sides[j+k]:=Sides[j]; j:=j-k
      end;
      Sides[j+k]:=tmp
    end;
    step:=3*step div 5;  // определение следующего шага
  end;
end; // Shell
```

И, наконец, при рисовании всех отсортированных граней назначается цвет кисти в зависимости от проекции нормали N:

```
Brush.Color:=Trunc(255*Abs(N))*$010100.
```

Напомним, что полностью проект "Изображение квадратичных поверхностей" приведен на компакт-диске в папке **Примеры | Глава 11 | Квадратичные поверхности**.

11.3. Построение тела по трем проекциям

В начертательной геометрии и черчении одна из основных задач состоит в том, чтобы спроецировать трехмерное тело на координатные плоскости OX, OY, OZ. Обратная задача более сложна: по трем (или даже по двум проекциям) восстановить объемное изображение тела. Для решения обратной задачи предназначен рассматриваемый в данном разделе проект "Восстановление трехмерного тела по проекциям". В общем случае эта задача может не решаться однозначно. На рис. 11.24 приведен пример трех проекций, которые соответствуют двум различным изображениям.

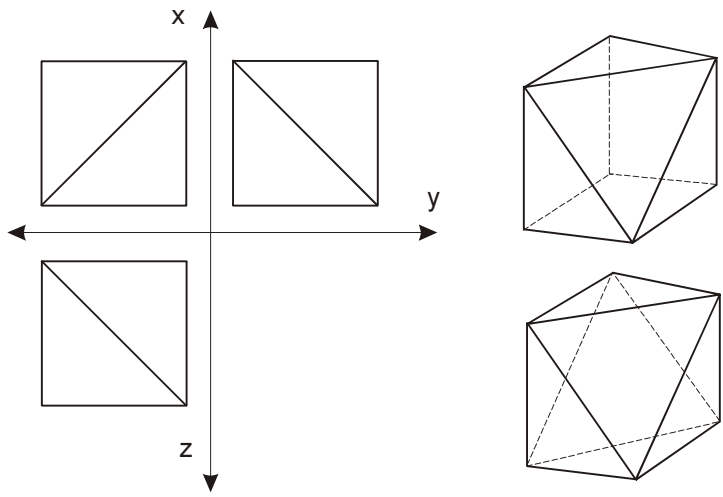


Рис. 11.24. Пример неоднозначного соответствия проекций и трехмерных изображений

На рис. 11.25 приведен пример проекций тела, трехмерное изображение которого восстанавливается однозначно.

Данные для проекций, изображенных на рис. 11.25, записаны в текстовый файл и содержат следующую информацию:

```
10 1 1 3 1 5 1 5 6 3 6 1 6 2 3 3 3 3 4 2 4
10 1 1 3 1 5 1 5 6 3 6 1 6 3 3 3 4 5 4 5 3
8 1 1 5 1 5 3 3 3 3 5 1 5 3 2 5 2
```

```
13 0 1 1 2 2 3 3 4 4 5 5 0 0 2 3 5 6 7 8 9 9 6 1 7 8 4
17 0 1 1 2 2 3 3 4 4 5 5 0 1 6 7 4 0 2 3 5 6 7 7 8 8 9 9 6 1 4 2 9 3 8
10 0 1 1 2 2 3 3 4 4 5 5 0 6 7 3 6 2 7 4 6
```

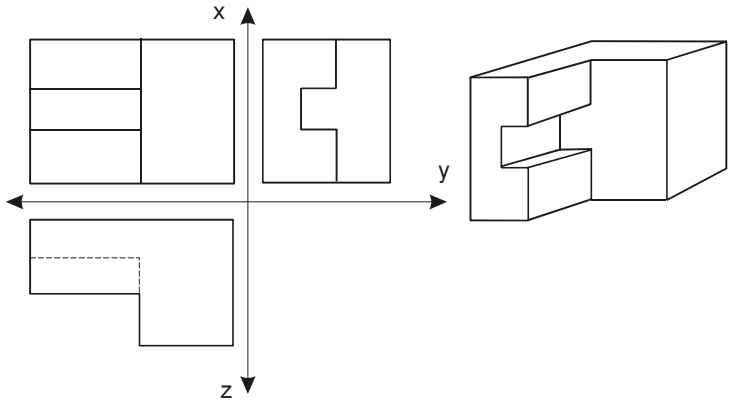


Рис. 11.25. Проекции тела, трехмерное изображение которого восстанавливается однозначно

В первых трех строках записаны координаты (`x0[i], y0[i]`) трех проекций. Первое число `n0[i]` в строке указывает на число точек каждой проекции. Три следующих строки этого файла указывают на отрезки, из которых составлены проекции. Первое число `m0[i]` в строке указывает на число отрезков, из которых составлена `i`-я проекция. Затем числа идут парами. В каждой паре первое число `l1[i]` указывает на номер точки начала отрезка, второе `l2[i]` — на номер точки конца отрезка. Необходимо указывать все возможные перекрытия отрезков. Так, например, на третьей проекции указаны отрезки (3,6), (3,4) и их перекрытие (4,6).

В проекте используется тип `TVector` для описания вершин (листинг 11.18).

Листинг 11.18. Тип для координат вершин

```
TVector = record
  x,y,z: real;
  NumPoint: array[1..3] of byte;
end;
```

В проекте используется также динамический массив `P: array of TVector`. Переменные `P[i].x`, `P[i].y` и `P[i].z` предназначены для хранения трехмерных координат вершин тела, поля `P[i].NumPoint[j]` используются для хранения номеров точек `j`-й проекции.

Для хранения ребер используется структура `TEdge`:

```
TEdge=record
  p1,p2: word; // номера вершин
end;
```

в которой поля `p1`, `p2` показывают номера вершин начала и конца ребра.

В проекте используются следующие переменные:

- двумерный массив ребер `Edges: array[0..3] of array of Tedge`, предназначенный для хранения информации в трехмерном пространстве и на проекциях `XY`, `XZ`, `YZ`;
- массивы `x0,y0: array[1..3] of array of real`, содержащие информацию о координатах на проекциях `XY,XZ,YZ`;
- элементы массива `CountPoint: array[0..3] of byte`, определяющие число вершин в трехмерном пространстве и на проекциях `XY`, `XZ`, `YZ`;
- элементы массива `CountEdge : array[0..3] of byte`, содержащие информацию о числе ребер в трехмерном пространстве и на проекциях `XY`, `XZ`, `YZ`.

Формирование массивов `P` и массива ребер `Edges` реализовано в процедуре `SetPointEdge` (листинг 11.17). Трехкратный цикл по индексам `i`, `j`, `k` формирует массив трехмерных точек `P[n]` с полями `P[n].x`, `P[n].y`, `P[n].z` и запоминает соответствующие им номера точек на проекциях в массиве `P[n].NumPoint[j]`, `j=1,2,3`. Включение точки в массив `P[n]` происходит по следующему алгоритму: на первой проекции берется `i`-я точка, на второй — `j`-я, на третьей — `k`-я. Если для координат точек на проекциях выполняется условие `(x0[2,j]=x0[3,k]) and (y0[3,k]=x0[1,i]) = true`, то точка с номерами `i`, `j`, `k` включается в массив `P[n]`.

Четырехкратный цикл по индексам `k`, `j`, `k1`, `k2` формирует неизбыточный массив трехмерных отрезков, точнее, номеров точек начала `Edges[m].p1` и конца `Edges[m].p2` отрезков. На каждой проекции (`k=1,2,3`) перебираются все отрезки (`j=0,..., CountEdge[k]-1`). Для всех `CountPoint[0]*CountPoint[0]` возможных комбинаций из трехмерных точек проверяется четыре условия:

- на `k`-й проекции началу `j`-го отрезка соответствует трехмерная точка (`Edges[k,j].p1=P[k1].NumPoint[k]`) и концу `j`-го отрезка соответствует трехмерная точка (`Edges[k,j].p2=P[k2].NumPoint[k]`);
- при помощи функции `Test(k,a,b: byte): boolean` проверяется, что отрезок `P[k1].NumPoint[u],P[k2].NumPoint[u]` принадлежит проекции с номером `u`;
- при помощи той же функции `Test` проверяется, что отрезок `P[k1].NumPoint[v],P[k2].NumPoint[v]` принадлежит проекции с номером `v`;
- при помощи функции `Test0(k1,k2: integer): boolean` проверяется, что отрезок с началом/концом `k1/k2` еще не входил в массив `Edges[m].p1/ Edges[m].p2`.

Если все четыре условия выполнены, то число отрезков `m` увеличивается на `1` и в массивы `Edges[m].p1`, `Edges[m].p2` вносятся номера точек начала и конца нового отрезка (листинг 11.19).

Листинг 11.19. Формирование массивов *P* и массива трехмерных отрезков

```
procedure TForm1.SetPointEdge;
var
  i,j,k,k1,k2,u,v: byte;
  done1,done3,done4,done5: boolean;
```

```
  function Test(k,a,b: byte): boolean;
  var i: integer;
  begin
    Result:=false; i:=-1;
    while (i<CountEdge[k]-1) and not Result do begin
      i:=i+1;
      Result:=(a=b) or
            ((a=Edges[k,i].p1) and (b=Edges[k,i].p2)) or
            ((b=Edges[k,i].p1) and (a=Edges[k,i].p2));
    end;
  end;

  function Test0(k1,k2: integer): boolean;
  var i: integer;
  begin
    Result:=true; i:=-1;
    while (i<CountEdge[0]-1) and not Result do begin
      i:=i+1;
      Result:=not (
            ((k1=Edges[0,i].p1) and (k2=Edges[0,i].p2)) or
            ((k1=Edges[0,i].p2) and (k2=Edges[0,i].p1)));
    end;
  end;

begin
  CountPoint[0]:=0;
  for i:=0 to CountPoint[1]-1 do
  for j:=0 to CountPoint[2]-1 do
  if abs(y0[1,i]-y0[2,j])<eps then
  for k:=0 to CountPoint[3]-1 do
  if (abs(x0[2,j]-x0[3,k])<eps) and
     (abs(y0[3,k]-x0[1,i])<eps) then begin
    CountPoint[0]:=CountPoint[0]+1;
    SetLength(P,CountPoint[0]);
    P[CountPoint[0]-1].x:=x0[2,j];
    P[CountPoint[0]-1].y:=x0[1,i];
    P[CountPoint[0]-1].z:=y0[1,i];
    P[CountPoint[0]-1].NumPoint[1]:=i;
    P[CountPoint[0]-1].NumPoint[2]:=j;
    P[CountPoint[0]-1].NumPoint[3]:=k;
  end;
  CountEdge[0]:=0;
  for k:=1 to 3 do begin
    u:=k mod 3 + 1;
    v:=(k+1) mod 3 + 1;
```

```
    for j:=0 to CountEdge[k]-1 do
    for k1:=0 to CountPoint[0]-1 do
    for k2:=0 to CountPoint[0]-1 do begin
      done1:=(Edges[k,j].p1=P[k1].NumPoint[k]) and
             (Edges[k,j].p2=P[k2].NumPoint[k]);
      done3:=Test(u,P[k1].NumPoint[u],P[k2].NumPoint[u]);
      done4:=Test(v,P[k1].NumPoint[v],P[k2].NumPoint[v]);
      done5:=Test0(k1,k2);
      if done1 and done3 and done4 and done5 then begin
        CountEdge[0]:=CountEdge[0]+1;
        SetLength(Edges[0],CountEdge[0]);
        Edges[0,CountEdge[0]-1].p1:=k1;
        Edges[0,CountEdge[0]-1].p2:=k2;
      end;
    end;
  end;
end;
```

При построении предполагается, что форма проекта "Восстановление трехмерного тела по проекциям" разбита на четыре окна с номерами 0, 1, 2, 3 (рис. 11.26).

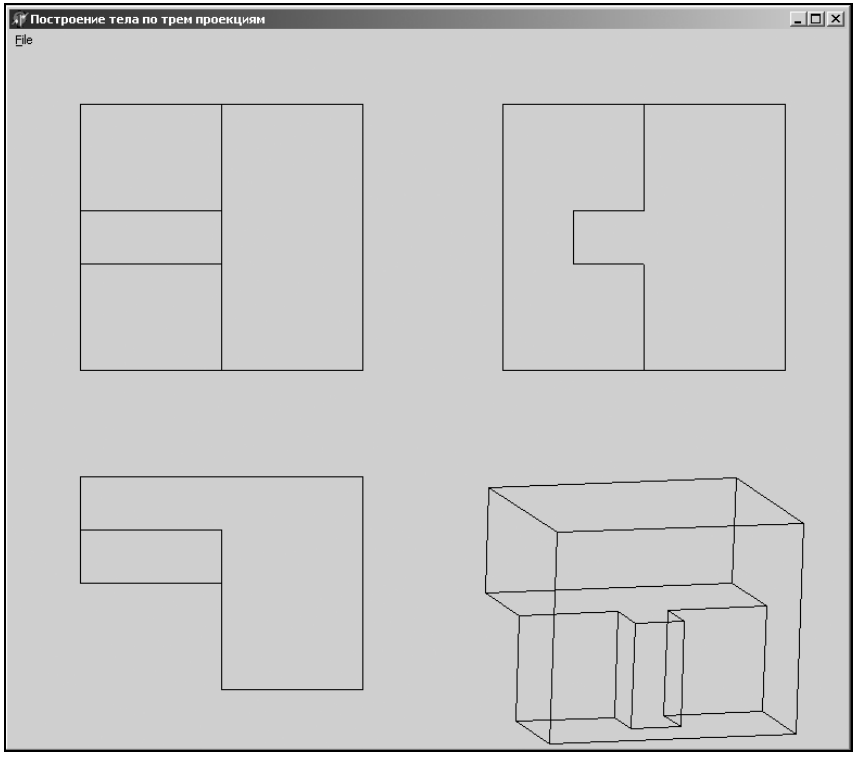


Рис. 11.26. Форма для проекта "Восстановление трехмерного тела по проекциям"

Размеры окон задают массивы:

```
a10: array[0..3] of real=(-3,0,0,0);
a20: array[0..3] of real=(9,6,6,6);
b10: array[0..3] of real=(-7,0,0,0);
b20: array[0..3] of real=(2,7,7,7);
```

Для масштабирования по всем четырем окнам используются функции `II(fl: byte; x: real)` и `JJ(fl: byte; y: real): integer`, где `fl` — номер окна (листинг 11.20).

Листинг 11.20. Функции масштабирования для всех окон

```
function TForm1.II(fl: byte; x: real): integer;
begin
  II:=I1[fl]+Trunc((x-a10[fl])*(I2[fl]-I1[fl])/(a20[fl]-a10[fl]));
end;

function TForm1.JJ(fl: byte; y: real): integer;
begin
  JJ:=J2[fl]+Trunc((y-b10[fl])*(J1[fl]-J2[fl])/(b20[fl]-b10[fl]));
end;
```

Перед выводом трехмерного изображения тела последовательно выполняются два поворота системы координат с помощью процедуры `Rotate(fl: byte; a: real; var x,y,z: real)`. Здесь `fl` — номер оси, вокруг которой реализуется поворот, параметр `a` — угол поворота (листинг 11.21).

Листинг 11.21. Процедура поворота системы координат

```
procedure TForm1.Rotate(fl: byte; a: real; var x,y,z: real);
var
  u:real;
begin
  case fl of
    1: begin
         u:=y;
         y:=u*cos(a)-z*sin(a);
         z:=u*sin(a)+z*cos(a);
       end;
    2: begin
         u:=x;
         x:=u*cos(a)-z*sin(a);
         z:=u*sin(a)+z*cos(a);
       end;
    3: begin
         u:=x;
```

```
        x:=u*cos(a)-y*sin(a);
        y:=u*sin(a)+y*cos(a);
      end;
  end;
end;
```

При пересчете координат формируются величины `MinX`, `MinY`, `MaxX`, `MaxY` для нулевого окна, в котором строится трехмерное изображение тела (листинг 11.22).

Листинг 11.22. Рисование окон

```
procedure TForm1.Draw(Alf,Bet: real);
var
  i,j: byte;
  MinX,MinZ,MaxX,MaxZ: real;
begin
  with Canvas do begin
    FillRect(Rect(0,0,Width,Height));
    //прорисовка 3-х проекций
    for i:=1 to 3 do
    for j:=0 to CountEdge[i]-1 do begin
      MoveTo(II(i,x0[i,Edges[i,j].p1]),JJ(i,y0[i,Edges[i,j].p1]));
      LineTo(II(i,x0[i,Edges[i,j].p2]),JJ(i,y0[i,Edges[i,j].p2]));
    end;
    //построение трехмерного изображения
    MinX:=100; MinZ:=100; MaxX:=-100; MaxZ:=-100;
    for i:=0 to CountPoint[0]-1 do begin
      Rotate(1,Pi/180*Alf,P[i].x,P[i].y,P[i].z);
      Rotate(2,Pi/180*Bet,P[i].x,P[i].y,P[i].z);
      if P[i].x<MinX then MinX:=P[i].x;
      if P[i].z<MinZ then MinZ:=P[i].z;
      if P[i].x>MaxX then MaxX:=P[i].x;
      if P[i].z>MaxZ then MaxZ:=P[i].z;
    end;
    a10[0]:=MinX-1; a20[0]:=MaxX+1;
    b10[0]:=MinZ-1; b20[0]:=MaxZ+1;
    for i:=0 to CountEdge[0]-1 do begin
      MoveTo(II(0,P[Edges[0,i].p1].x),JJ(0,P[Edges[0,i].p1].z));
      LineTo(II(0,P[Edges[0,i].p2].x),JJ(0,P[Edges[0,i].p2].z));
    end;
  end;
end;
```

Полный текст проекта "Восстановление трехмерного тела по проекциям" приведен на компакт-диске в папке **Примеры | Глава 11 | Проекции**.

11.4. Бинарные операции с многоугольниками

Операции объединения, вычитания, пересечения двумерных и трехмерных тел используются при проектировании в архитектуре и машиностроении. Эта задача является достаточно сложной. Однако бинарная нумерация операций над множествами позволяет сформулировать общий алгоритм для ЭВМ.

Из элементов двух множеств A и B пространства M, используя операции объединения "$\cup$", пересечения "$\cap$" и разности "/", можно составить новое множество C (рис. 11.27).

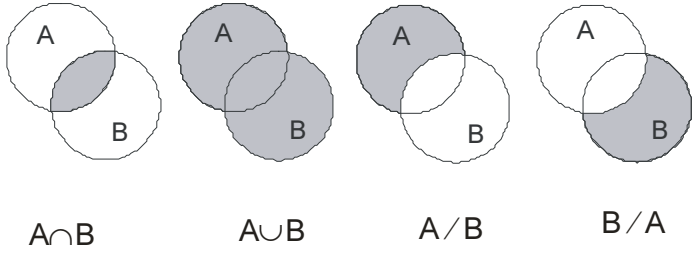


Рис. 11.27. Операции над множествами

В общем случае к двум множествам A и B можно применить шестнадцать различных операций. Для дальнейших рассуждений введем двоичную нумерацию этих операций. Без ограничения общности можно считать, что пересечение множеств не пустое — $A \cap B <> \varnothing$, то есть существуют x такие, что $(x \in A) \wedge (x \in B)$.

Каждому элементу $x \in M$ поставим в соответствие индекс (рис. 11.28):

0: $(x \in A) \wedge (x \in B)$;

1: $(x \in A) \wedge (x \notin B)$;

2: $(x \notin A) \wedge (x \in B)$;

3: $(x \notin A) \wedge (x \notin B)$.

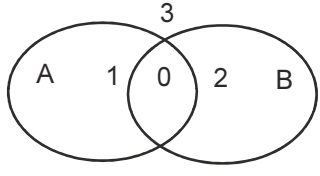


Рис. 11.28. Нумерация при разбиении подмножеств

Пусть индекс элемента $x \in M$ соответствует номеру бита в двоичном представлении номера операции (табл. 11.1). В той же таблице приведены логические операции, соответствующие операциям над множествами.

Таблица 11.1. *Операции над множествами*

N	Логическая операция	Операция над множествами
0000	FALSE	0
0001	$A \wedge B$	$A \cap B$
0010	$A \wedge \overline{B}$	A / B
0011	A	A
0100	$\overline{A} \wedge B$	B / A
0101	B	B
0110	A xor B	$(A / B) \cup (B / A)$
0111	$A \vee B$	$A \cup B$
1000	$\overline{A} \wedge \overline{B}$	$(M / A) \cap (M / B)$
1001	$(A \vee \overline{B}) \wedge (\overline{A} \vee B)$	$(A \cup (M / B)) \cap ((M / A) \cup B)$
1010	$\overline{B}$	M / B
1011	$A \vee \overline{B} = B \Rightarrow A$	$A \cup (M / B)$
1100	$\overline{A}$	M / A
1101	$\overline{A} \vee B = A \Rightarrow B$	$(M / A) \cup B$
1110	$\overline{A} \vee \overline{B}$	$(M / A) \cup (M / B)$
1111	TRUE	M

Так, например, 1 в нулевом бите номера операции пересечения $C = A \cap B$ указывает на то, что во множество C входят только элементы с номером 0. Ясно, что практический интерес, прежде всего, представляют операции с номерами 1, 2, 4, 6, 7: $A \cap B$, A / B, B / A, $(A / B) \cup (B / A)$, $A \cup B$.

Каждому граничному элементу множеств A и B также поставим в соответствие число, которое будем называть индексом границы. Получим индекс границы, складывая номера множеств, которые разделяет эта граница (рис. 11.29). Так, 1+0=1 — индекс границы между множествами 0 и 1; 2+0=2 — индекс границы между мно-

жествами 0 и 2; 3+0=1+2=3 — узловые элементы; 3+1=4 — индекс границы между множествами 1 и 3; 3+2=5 — индекс границы между множествами 3 и 2.

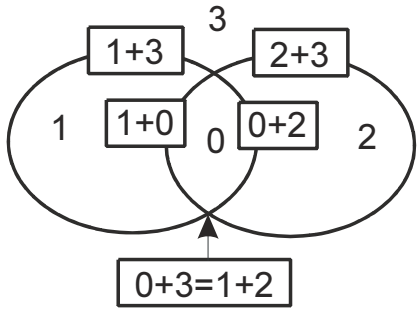


Рис. 11.29. Индексы границ между множествами

В любой узловой точке, точке пересечения границ, сходятся две или четыре (для операции с номером 6) границы нового множества C.

Между номерами операций и индексами граничных элементов есть связь: сумма номеров пар различных битов в двоичном представлении номера операции совпадает с индексами граничных элементов.

Так, например, для операции $C = (A / B) \cup (B / A)$ с номером 0110 граница состоит из элементов с номерами: 0+1, 2+3, 1+3, 2+3, 3+0=1+2. В двоичном числе 0110 различны следующие пары битов: (0,1), (0,2), (2,3), (1,3). Элементы пересечения границ (0+3=1+2) входят в границы любого нового множества. Для операции, например, пересечения $A \cap B$ с номером 0001 граница множества C состоит из элементов с индексами 0+1, 0+2, 0+3=1+2. Эти же пары различных битов есть в числе 0001: (0,1), (0,2), (0,3).

Рассмотрим теперь задачу взаимодействия двух многоугольников на плоскости (рис. 11.30).

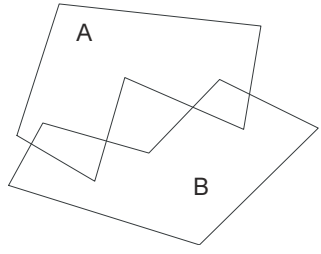


Рис. 11.30. Взаимодействие двух многоугольников на плоскости

Для решения этой задачи предлагается проект "Бинарные операции с многоугольниками". На рис. 11.31 представлен вид единственной формы проекта, на которой размещены радиокнопки (компонент `RadioGroup`) для выбора бинарной операции.

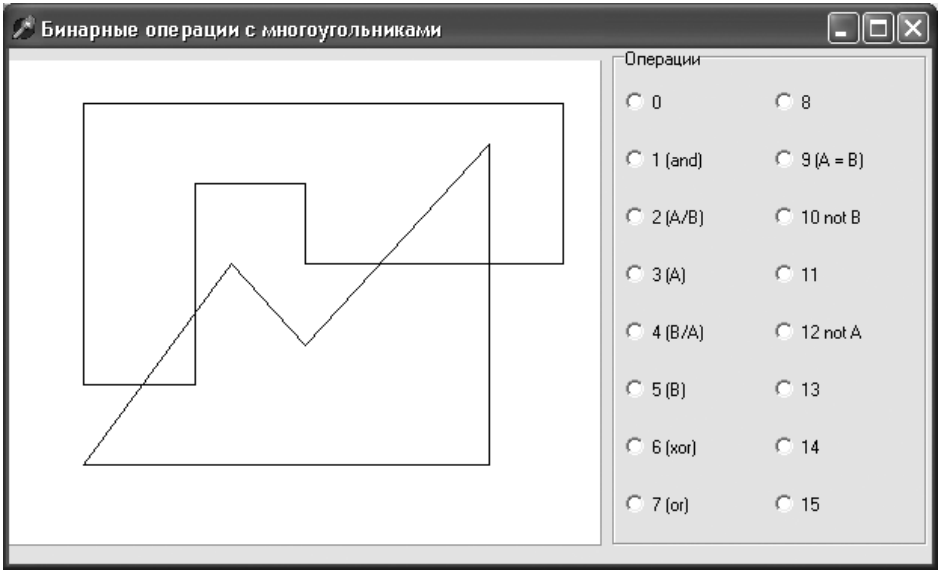


Рис. 11.31. Форма проекта "Бинарные операции с многоугольниками"

Для решения задачи взаимодействия двух многоугольников A и B на плоскости используется следующий алгоритм:

- найти точки пересечения границ и добавить их в массивы вершин многоугольников;
- расставить индексы вершин;
- собрать в новый многоугольник C отрезки с подходящими индексами.

Для описания координат вершин многоугольников введем тип:

```
TVector=record
  x,y: real;
  p  : byte;
end; ,
```

где `TD1.p` — индекс вершины, `TD1.x`, `TD1.y` — координаты вершины.

Для описания ребер многоугольника введем следующий тип:

```
TEdge=record
  n1,n2: byte;
end; ,
```

где `n1`, `n2` — номера вершин, в которых находятся начало и конец ребра.

Многоугольники A и B описываются следующими переменными:

```
CountPoint: array[1..2] of byte; // число точек 1/2 контура
CountEdge : array[1..2] of byte; // число отрезков 1/2 контура
Points    : array[1..2,1..20] of TVector; // точки 1/2 контура
Edges     : array[1..2,1..20] of TEdge;   // отрезки 1/2 контура
```

Задачу пересечения границ многоугольников решает процедура `AddPoint(n,m: Byte)`, где `n` — номер стороны первого контура, `m` — номер стороны второго контура.

Решение системы уравнений (листинг 11.23), описывающих две прямые

$$\frac{x-x_1}{x_2-x_1}=\frac{y-y_1}{y_2-y_1};\qquad \frac{x-x_3}{x_4-x_3}=\frac{y-y_3}{y_4-y_3}, \tag{11.6}$$

ищем в виде: `x=w1/w`, `y=-w2/w`, где

```
w:=(y2-y1)*(x4-x3)-(y4-y3)*(x2-x1);
w1:=(y3-y1)*(x4-x3)*(x2-x1)+x1*(y2-y1)*(x4-x3)-x3*(y4-y3)*(x2-x1);
w2:=(x3-x1)*(y4-y3)*(y2-y1)+y1*(x2-x1)*(y4-y3)-y3*(x4-x3)*(y2-y1).
```

Листинг 11.23. Пересечения границ многоугольников

```
procedure AddPoint(n,m: Byte);
var
  x,y,x1,y1,x2,y2,x3,y3,x4,y4: Real;
  w,w1,w2: Real;
begin
  x1:=Points[1,Edges[1,n].n1].x; y1:=Points[1,Edges[1,n].n1].y;
  x2:=Points[1,Edges[1,n].n2].x; y2:=Points[1,Edges[1,n].n2].y;
  x3:=Points[2,Edges[2,m].n1].x; y3:=Points[2,Edges[2,m].n1].y;
  x4:=Points[2,Edges[2,m].n2].x; y4:=Points[2,Edges[2,m].n2].y;
  w:=(y2-y1)*(x4-x3)-(y4-y3)*(x2-x1);
  w1:=(y3-y1)*(x4-x3)*(x2-x1)+x1*(y2-y1)*(x4-x3)-
      x3*(y4-y3)*(x2-x1);
  w2:=(x3-x1)*(y4-y3)*(y2-y1)+y1*(x2-x1)*(y4-y3)-
      y3*(x4-x3)*(y2-y1);
  if Eqv(w,0) then
    if Eqv(w1,0) and Eqv(w2,0) then begin{совпадают}
      TestAdd(x1,y1); TestAdd(x2,y2);
    end
    else
  else begin  {пересекаются}
    x:=w1/w; y:=-w2/w; TestAdd(x,y);
  end;
end; // AddPoint
```

При решении системы уравнений (11.6) возможны следующие случаи (рис. 11.32):

1. Отрезки параллельны.
2. Отрезки лежат на одной прямой.
3. Отрезки имеют общий конец.
4. Конец одного отрезка принадлежит другому отрезку.
5. Отрезки пересекаются.
6. Отрезки не пересекаются.

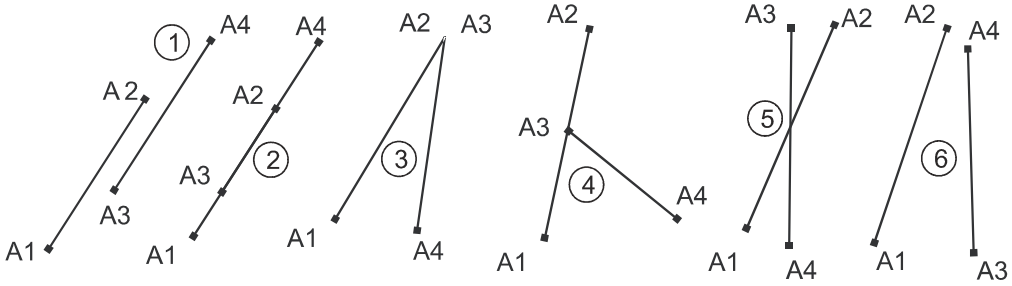


Рис. 11.32. Варианты пересечения отрезков

Новые вершины и стороны добавляются во втором, в четвертом и в пятом случаях. За добавление вершин и сторон в массивы `A[]` и `L[]` отвечает процедура `TestAdd`, которая получает возможную новую вершину с координатами (`x,y`) и анализирует ситуацию (листинг 11.24).

Листинг 11.24. Добавление вершин

```
procedure TestAdd(x,y: Real);
var
  b0,b1,b2,q0,q1,q2: boolean;
begin
  b0:=((Min(x1,x2)+Eps<x) and (x<Max(x1,x2)-Eps)) or
      ((Min(y1,y2)+Eps<y) and (y<Max(y1,y2)-Eps));
  b1:=Eqv(x,x1) and Eqv(y,y1);
  b2:=Eqv(x,x2) and Eqv(y,y2);
  q0:=((Min(x3,x4)+Eps<x) and (x<Max(x3,x4)-Eps)) or
      ((Min(y3,y4)+Eps<y) and (y<Max(y3,y4)-Eps));
  q1:=Eqv(x,x3) and Eqv(y,y3); {x=3}
  q2:=Eqv(x,x4) and Eqv(y,y4); {x=4}

  if (q1 and b1) or (q2 and b1) or
     (q1 and b2) or (q2 and b2) then begin
    Points[1,Edges[1,n].n1].p:=3;
```

```
  Points[2,Edges[2,m].n1].p:=3;
end;
if q0 and (b1 or b2) then begin  // конец первого на втором}
  CountPoint[2]:=CountPoint[2]+1;
  Points[2,CountPoint[2]].x:=x;
  Points[2,CountPoint[2]].y:=y;
  Points[2,CountPoint[2]].p:=3;
  CountEdge[2]:=CountEdge[2]+1;
  Edges[2,CountEdge[2]].n2:=Edges[2,m].n2;
  Edges[2,m].n2:=CountPoint[2];
  Edges[2,CountEdge[2]].n1:=CountPoint[2];
  if b1
    then Points[1,Edges[1,n].n1].p:=3
    else Points[1,Edges[1,n].n2].p:=3;
end;
if b0 and (q1 or q2) then begin {конец второго на первом}
  CountPoint[1]:=CountPoint[1]+1;
  Points[1,CountPoint[1]].x:=x;
  Points[1,CountPoint[1]].y:=y;
  Points[1,CountPoint[1]].p:=3;
  CountEdge[1]:=CountEdge[1]+1;
  Edges[1,CountEdge[1]].n2:=Edges[1,n].n2;
  Edges[1,n].n2:=CountPoint[1];
  Edges[1,CountEdge[1]].n1:=CountPoint[1];
  if q1
    then Points[2,Edges[2,m].n1].p:=3
    else Points[2,Edges[2,m].n2].p:=3;
end;
if q0 and b0 then begin {посередине обеих отрезков}
  CountPoint[2]:=CountPoint[2]+1;
  Points[2,CountPoint[2]].x:=x;
  Points[2,CountPoint[2]].y:=y;
  Points[2,CountPoint[2]].p:=3;
  CountEdge[2]:=CountEdge[2]+1;
  Edges[2,CountEdge[2]].n2:=Edges[2,m].n2;
  Edges[2,m].n2:=CountPoint[2];
  Edges[2,CountEdge[2]].n1:=CountPoint[2];
  CountPoint[1]:=CountPoint[1]+1;
  Points[1,CountPoint[1]].x:=x;
  Points[1,CountPoint[1]].y:=y;
  Points[1,CountPoint[1]].p:=3;
  CountEdge[1]:=CountEdge[1]+1;
  Edges[1,CountEdge[1]].n2:=Edges[1,n].n2;
  Edges[1,n].n2:=CountPoint[1];
  Edges[1,CountEdge[1]].n1:=CountPoint[1];
```

```
      with Form1,Image1.Canvas do
        RectAngle(II(x)-3,JJ(y)-3,II(x)+3,JJ(y)+3);
    end;
  end;  // TestAdd
```

Задача расстановки индексов вершин решается следующим образом: если вершина первого многоугольника принадлежит второму многоугольнику, то ее индекс равен 0 + 2 = 2, иначе индекс равен 1 + 3 = 4; если вершина второго многоугольника принадлежит первому многоугольнику, то ее индекс равен 0 + 1 = 1, иначе индекс равен 2 + 3 = 5. Принадлежность точки многоугольнику определяет функция `PointInPolygon(j,n: Byte): Boolean`, где `j` — номер контура, `n` — номер вершины контура. Если вершина с номером `n` контура `j` принадлежит другому контуру, то `PointInPolygon:= true`, иначе `PointInPolygon:= false`. Алгоритм определения принадлежности точки `P(x,y)` заключается в следующем: через точку `P(x,y)` и точку `Q(x0,y)`, не принадлежащую многоугольнику, проводится горизонтальный отрезок (рис. 11.33). Если число пересечений отрезка (`P,Q`) с контуром четно, то `P` не принадлежит многоугольнику, если нечетно, то принадлежит (листинг 11.25).

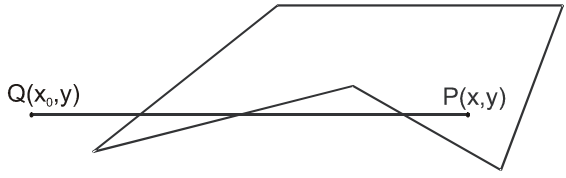


Рис. 11.33. К алгоритму определения принадлежности точки многоугольнику

При подсчете числа пересечений не учитываются стороны многоугольника, имеющие горизонтальное направление. При прохождении отрезка (`P,Q`) через вершину многоугольника, пересечение учитывается только для тех его сторон, для которых эта вершина является верхней.

Листинг 11.25. Принадлежность точки многоугольнику

```
function PointInPolygon(j,nn: Byte): Boolean;
{j=1: n-ая точка 2 контура принадлежит 1 контуру}
{j=2: n-ая точка 1 контура принадлежит 2 контуру}
var i,k: byte;
begin
  k:=0;
  for i:=1 to CountEdge[j] do
  if not Eqv(Points[j,Edges[j,i].n1].y,Points[j,Edges[j,i].n2].y)
     and PointInLine(
     Points[j,Edges[j,i].n1].x,Points[j,Edges[1,i].n1].y,
     Points[j,Edges[j,i].n2].x,Points[j,Edges[j,i].n2].y,
```

```
    Points[3-j,Edges[3-j,nn].n1].x,
              Points[3-j,Edges[3-j,nn].n1].y) then k:=k+1;
  Result:=k mod 2<>0;
end;  // PointInPolygon
```

В этой функции пересечение горизонтального отрезка (`P,Q`) с негоризонтальным отрезком (`x1,y1,x2,y2`) определяется функцией `PointInLine(x1, y1, x2, y2, x3, y3: Real): Boolean`, которая является частным случаем процедуры `Test` (листинг 11.26).

Листинг 11.26. Пересечение горизонтального отрезка (*P,Q*)

```
function PointInLine(x1,y1,x2,y2,x3,y3: Real): Boolean;
var x,y,w,w1,w2,x4: Real;
    b0,b1,b2,q0,q1,q2: boolean;
begin
  x4:=-100;
  w:=(y2-y1)*(x4-x3);
  w1:=(y3-y1)*(x4-x3)*(x2-x1)+x1*(y2-y1)*(x4-x3);
  w2:=-y3*(x4-x3)*(y2-y1);
  x:=w1/w; y:=-w2/w;
  b0:=((Min(x1,x2)+Eps<x) and (x<=Max(x1,x2)-Eps)) or
      ((Min(y1,y2)+Eps<y) and (y<Max(y1,y2)-Eps));
  b1:=Eqv(x,x1) and Eqv(y,y1);
  b2:=Eqv(x,x2) and Eqv(y,y2);
  q0:=((Min(x3,x4)+Eps<x) and (x<Max(x3,x4)-Eps));
  q1:=Eqv(x,x3) and Eqv(y,y3);
  q2:=Eqv(x,x4) and Eqv(y,y3);
  Result:=(b0 or (b1 and (y1<y2)) or (b2 and (y2<y1))) and
          (q0 or q1 or q2);
end; // PointInLine
```

Для решения задачи выбора сторон нового многоугольника, подходящих для операции с номером `k`, используется функция `Op_In_Set(k,n,m: Byte): Boolean`, где `n`, `m` — индексы концов стороны, `k` — номер операции. В функции `Op_In_Set` по номеру операции `k` формируется множество допустимых индексов `V`; отрезок считается подходящим (`Op_In_Set:= true`), если оба индекса входят в множество `V` (листинг 11.27).

Листинг 11.27. Выбор сторон нового многоугольника

```
function Op_In_Set(k,n,m: Byte): Boolean;
// отрезок n удовлетворяет операции k
const W: array[1..5] of set of byte=([1],[2],[3],[4],[5]);
var V: set of byte;   i,j: byte;
begin
  V:=[3];
  for i:=0 to 2 do
```

```
    for j:=i+1 to 3 do
    if ((k and (1 shl i))<>0) xor
       ((k and (1 shl j))<>0) then V:=V+W[i+j];
    Result:=(n in V) and (m in V);
  end;  // Op_In_Set
```

При выборе учитывается, что если оба конца отрезка имеют индекс 3, то для первого контура это ребро имеет индекс 0 + 2, а для второго — индекс 0 + 1 = 1 (рис. 11.29).

Все три задачи решает процедура `Set_Op(k: byte)`, которая получает номер операции `k` (листинг 11.28).

Листинг 11.28. Задача взаимодействия двух многоугольников

```
procedure TForm1.Set_Op(k: byte);
var i,j,nn,mm: Byte;
begin
  for i:=1 to CountEdge[1] do          // Узловые точки
    for j:=1 to CountEdge[2] do AddPoint(i,j);

  // Пройти точки 1 и 2 контура (A.p<>3):
  // Если точка 1 контура прин 2-му, то A.p=0+2 иначе A.p=1+3
  // Если точка 2 контура прин 1-му, то A.p=0+1 иначе A.p=2+3
  for j:=1 to 2 do
  for i:=1 to CountPoint[j] do
  if Points[j,i].p<>3 then
  case j of
    1: if PointInPolygon(2,i) then Points[j,i].p:=2
       else Points[j,i].p:=4;
    2: if PointInPolygon(1,i) then Points[2,i].p:=1
       else Points[2,i].p:=5;
  end;

  with Image1.Canvas do begin
    Pen.Color:=clRed;
    for j:=1 to 2 do
    for i:=1 to CountEdge[j] do
    with Edges[j,i] do begin
      nn:=Points[j,n1].p; mm:=Points[j,n2].p;
      if (nn=3) and (mm=3) then begin nn:=3-j; mm:=3-j; end;
      if Op_In_Set(k,nn,mm) then begin
        MoveTo(II(Points[j,n1].x),JJ(Points[j,n1].y));
        LineTo(II(Points[j,n2].x),JJ(Points[j,n2].y));
      end;
    end;
  end;
end;  // Set_Op
```

Аналогичным образом решается задача пересечения многоугольников в трехмерном пространстве, которая будет рассмотрена в *главе 12*:

- сначала необходимо найти линии пересечения двух многогранников, добавляя в массивы вершин, ребер и граней новые элементы;
- затем расставить индексы вершин многоугольников;
- и, наконец, сформировать новый многогранник, собирая те грани, у которых все вершины имеют подходящие индексы.

Полностью проект "Бинарные операции с многоугольниками" приведен на компакт-диске в папке **Примеры | Глава 11 | Бинарные операции**.

ГЛАВА 12

Графические редакторы трехмерных тел

12.1. Упрощенный проект "Редактор многогранников"

12.1.1. Описание проекта

Проект "Редактор многогранников" предназначен для визуальной работы с многогранниками. В проекте реализованы следующие функции: чтение и запись данных; перемещение окна просмотра и геометрических тел; масштабирование изображений на любой проекции; перемещение вершин на любой проекции; выделение граней и увеличение их числа путем добавления к ним новых вершин. Предполагается, что все грани представляют собой треугольники.

В проекте используется следующая структура данных.

Тип

```
TVector=record
  x,y,z: real;
  Select: boolean;
end;
```

описывает трехмерные точки. К трем обычным координатам точки добавлено поле `Select: boolean`, отвечающее за визуальное выделение точки. Выделение трех точек на одной грани позволяет выделить эту грань и разбить ее на три грани добавлением новой точки.

Для описания ребер предназначен следующий тип:

```
TEdge=record
  n1,n2: word;
end;
```

Два поля переменных этого типа `n1` и `n2` показывают на номера вершин ребра.

Поля `p[1..3]` переменных типа `TFace` показывают на номера вершин треугольной грани, поля `A`, `B`, `C`, `D`, `NormalZ` используются для вычисления коэффициентов уравнения плоскости и проекции нормали на ось `Z`:

```
TFace=record
  p     : array[1..3] of word;
  A,B,C,D,NormalZ: real;
  ZMid  : real;
end;
```

В проекте можно обрабатывать одновременно несколько многогранников. Для хранения информации о каждом многограннике предназначен тип `TBody`.

```
TBody=record
  x0,y0,z0,n1,n2,n3: real;
  Visible: boolean;
  Vertex : array of TVector;
  VertexT: array of TVector;
  Edge   : array of TEdge;
  Face   : array of TFace;
end;
```

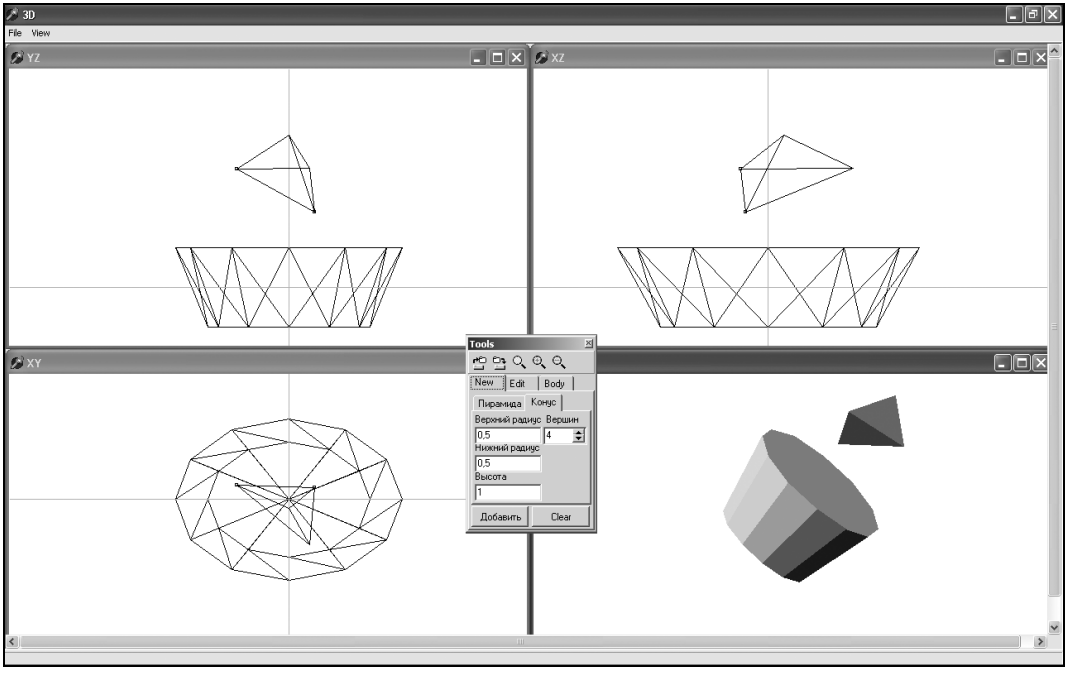


Рис. 12.1. Внешний вид проекта "Редактор многогранников"

Массивы

```
Vertex : array of TVector;
VertexT: array of TVector;
Edge   : array of TEdge;
Face   : array of TFace;
```

предназначены для хранения информации о вершинах, ребрах и гранях. Поля `x0`, `y0`, `z0`, `n1`, `n2`, `n3` описывают смещение и поворот многогранника.

Массив `Body: array of TBody` содержит данные обо всех многогранниках.

Проект (рис. 12.1) состоит из главной формы `FormMain` (с внешним именем **3D**) и инструментальной формы `FormTools` (с внешним именем **Tools**).

Главная форма вызывает четыре дочерних формы — **`FormYZ`**, **`FormXZ`**, **`FormXY`** и **`FormXYZ`** (с внешними именами **YZ**, **XZ**, **XY** и **XYZ**) для рисования трех проекций и общего вида фигуры в повернутой системе координат. У основной формы назначено свойство `FormStyle = fsMDIForm`, а у дочерних — `FormStyle = fsMDIChild`.

Инструментальная форма содержит три вкладки: **New** (рис. 12.2) — для добавления новых тел (пирамида или конус); **Edit** (рис. 12.3) — для редактирования положения тел или точек; **Body** (рис. 12.4) — перечень тел с возможностью изменения их видимости. На инструментальную форму вынесены 12 кнопок типа `TSpeedButton`:

`NewButton` (вкладка **New**) — новая фигура (одна грань);

`OpenButton` — открыть файлы с данными;

`SaveButton` — записать данные в файл;

`OffButton` (вкладка **Edit**) — отключить функции;

`MoveButton` (вкладка **Edit**) — переместить вершины;

`SelectButton` (вкладка **Edit**) — выделить вершины;

`RectButton` — растянуть рисунок до выделенного прямоугольника;

`Div2Button` — уменьшить рисунок вдвое;

`Mult2Button` — увеличить рисунок вдвое;

`AddButton` (вкладка **Edit**) — добавить точку к выделенной грани;

`MoveXYZButton` (вкладка **Edit**) — переместить систему координат;

`MovePictButton` (вкладка **Edit**) — переместить рисунок.

Кроме того, в проект включены:

два компонента `CheckBox1` и `CheckBox2` (вкладка **Edit** формы **Tools**) для рисования фигуры ребрами или гранями, с вершинами или без них;

компонент `MainMenu1` для организации меню;

компонент `OpenDialog1` для выбора файлов с данными;

компонент `SaveDialog1` для записи данных в файлы.

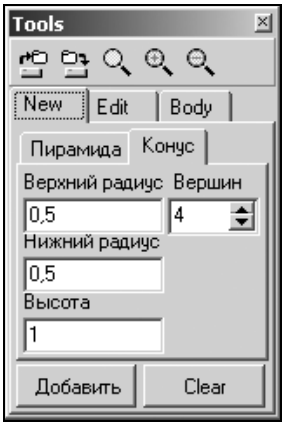


Рис. 12.2. Вкладка **New** формы **Tools**

Рис. 12.3. Вкладка **Edit** формы **Tools**

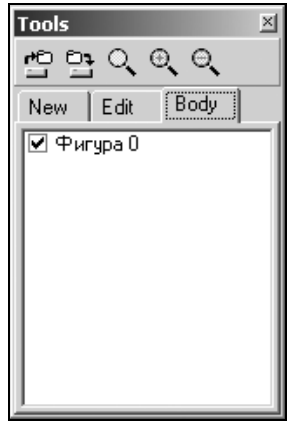


Рис. 12.4. Вкладка **Body** формы **Tools**

12.1.2. Чтение и запись данных

При инициализации проекта (листинг 12.1) задаются начальные размеры трехмерного параллелепипеда (`x1`, `y1`, `z1`), (`x2`, `y2`, `z2`), габариты окна в повернутой системе координат (`x10`, `y10`), (`x20`, `y20`), опускается флаг перемещения инструментальной панели (`drawing:=false`) и устанавливается инструмент с номером `0` (`fl_tools:=0`), позволяющий перетаскивать вершины на проекциях.

Листинг 12.1. Процедура инициализации данных

```
procedure TFormMain.FormCreate(Sender: TObject);
var k: byte;
```

```
begin
  drawing:=false; fl_tools:=0;
  x1:=-1; y1:=-5; z1:=-3;
  x2:=11; y2:=11; z2:=11;
  a1:=Pi/6; a2:=Pi/7;
  x10:=-6; x20:=6; y10:=-6; y20:=6;
  for k:=0 to 71 do begin
    si[k]:=sin(Pi/3+k*Pi/36); co[k]:=cos(Pi/3+k*Pi/36);
  end;
end;
```

При нажатии кнопки `OpenButton` вызывается стандартный метод `OpenDialog1`, позволяющий найти один из нетипизированных файлов проекта с расширением PRJ (листинг 12.2).

Листинг 12.2. Открытие файла

```
procedure TFormTools.SpeedButton3Click(Sender: TObject);
begin
  if FormMain.OpenDialog1.Execute then begin
    FileName:=FormMain.OpenDialog1.FileName;
    OpenPr;
    FormMain.Show3D;
  end;
end;
```

В начале файла записано число многогранников `CountBody`. В цикле до `CountBody` для каждого многогранника считывается число вершин и координаты этих вершин, затем — число треугольных граней и номера вершин, на которые эти грани опираются (листинг 12.3).

Листинг 12.3. Процедура чтения данных

```
procedure OpenPr;
var k,i,L: integer;
begin
  AssignFile(f,FileName); Reset(f,1);
  BlockRead(f,CountBody,SizeOf(CountBody));
  SetLength(Body,CountBody);
  for k:=0 to CountBody-1 do
  with Body[k] do begin
    Visible:=true;
    L:=Length(Body[k].Vertex);
    BlockRead(f,L,SizeOf(L));
    SetLength(Vertex,L);
```

```
    SetLength(VertexT,L);
    for i:=0 to L-1 do
      BlockRead(f,Vertex[i],SizeOf(Vertex[i]));
    BlockRead(f,L,SizeOf(L));
    SetLength(Face,L);
    for i:=0 to L-1 do
      BlockRead(f,Face[i],SizeOf(Face[i]));
  end;
  CloseFile(f);
  SetEdges;
end;
```

При нажатии кнопки `SaveButton` вызывается стандартный метод `SaveDialog1`.

```
procedure TFormTools.SpeedButton4Click(Sender: TObject);
begin
  FormMain.SaveDialog1.FileName:=FileName;
  if FormMain.SaveDialog1.Execute then begin
    FileName:=FormMain.SaveDialog1.FileName;
    SavePr;
  end;
end;
```

После задания имени файла, данные записываются в файл (листинг 12.4).

Листинг 12.4. Процедура записи проекта

```
procedure SavePr;
var k,i,L: integer;
begin
  AssignFile(f,FileName); Rewrite(f,1);
  BlockWrite(f,CountBody,SizeOf(CountBody));
  for k:=0 to CountBody-1 do
  with Body[k] do begin
    L:=Length(Vertex);
    BlockWrite(f,L,SizeOf(L));
    for i:=0 to L-1 do
      BlockWrite(f,Vertex[i],SizeOf(Vertex[i]));
    L:=Length(Face);
    BlockWrite(f,L,SizeOf(L));
    for i:=0 to L-1 do
      BlockWrite(f,Face[i],SizeOf(Face[i]));
  end;
  CloseFile(f);
end;
```

12.1.3. Анализ данных и рисование

После чтения данных из файла вызываются процедуры `SetEdges` (листинг 12.5) и `Show3D` (листинг 12.6). Процедура `SetEdges` анализирует информацию о гранях и создает массив `Edge[]` с информацией о ребрах. В элементах этого массива содержатся номера начальной и конечной точек для каждого ребра. В процедуре `SetEdges` организован цикл по всем граням многогранника. Если очередное ребро грани не входило в массив ребер, что проверяется функцией `EdgeInArray`, то процедура `AddEdge` увеличивает число ребер и фиксирует номер начальной и конечной точки ребра.

Листинг 12.5. Процедура заполнения массива ребер

```
procedure SetEdges;
var k,i,CountEdge: word;

  function EdgeInArray(m1,m2: integer): boolean;
  var j: integer;
  begin
    Result:=false; j:=-1;
    with Body[k] do
    while not Result and (j<CountEdge-1) do begin
      Inc(j);
      Result:=((Edge[j].n1=m1) and (Edge[j].n2=m2)) or
              ((Edge[j].n1=m2) and (Edge[j].n2=m1));
    end;
  end;

  procedure AddEdge(m1,m2: integer);
  begin
    with Body[k] do begin
      Inc(CountEdge); SetLength(Edge,CountEdge);
      Edge[CountEdge-1]:=SetEdge(m1,m2);
    end;
  end;

begin
  for k:=0 to CountBody-1 do
  with Body[k] do begin
    CountEdge:=0; SetLength(Edge,CountEdge);
    for i:=0 to Length(Face)-1 do
    with Face[i] do begin
      if not EdgeInArray(p[1],p[2]) then AddEdge(p[1],p[2]);
      if not EdgeInArray(p[2],p[3]) then AddEdge(p[2],p[3]);
```

```
      if not EdgeInArray(p[3],p[1]) then AddEdge(p[3],p[1]);
    end;
  end;
end;
```

Процедура `Show3D` вызывает процедуры рисования многогранника на всех четырех дочерних формах (листинг 12.6).

Листинг 12.6. Вызов процедур рисования на дочерних окнах

```
procedure TFormMain.Show3D;
begin
  FormXZ.Show3D;
  FormYZ.Show3D;
  FormXY.Show3D;
  FormXYZ.Show3D;
end;
```

Рисование на формах-проекциях реализуется примерно одинаково и, например, для формы `FormXZ`, на которой изображается проекция `XZ`, выглядит так (листинг 12.7).

Листинг 12.7. Процедура рисования многогранника на плоскости XZ

```
procedure TFormXZ.Show3D;
const c4=2;
var k,i: integer;
begin
  with Canvas do begin
    Brush.Color:=clWhite;
    RectAngle(-1,-1,Width,Height);
    for k:=0 to CountBody-1 do
    with Body[k] do
    if visible then begin
      Pen.Color:=clSilver;
      MoveTo(II(x1),JJ(0)); LineTo(II(x2),JJ(0));
      MoveTo(II(0),JJ(z1)); LineTo(II(0),JJ(z2));
      Pen.Color:=clBlack;
      for i:=0 to Length(Vertex)-1 do if Vertex[i].Select then
        RectAngle(II(Vertex[i].x)-c4,JJ(Vertex[i].z)-c4,
                  II(Vertex[i].x)+c4,JJ(Vertex[i].z)+c4);
      for i:=0 to Length(Edge)-1 do begin
        MoveTo(II(Vertex[Edge[i].n1].x),JJ(Vertex[Edge[i].n1].z));
        LineTo(II(Vertex[Edge[i].n2].x),JJ(Vertex[Edge[i].n2].z));
      end;
    end;
  end;
end;
```

В процедуре `FormXZ.Show3D` после очистки всей канвы рисуются оси координат. Затем организуется цикл по всем вершинам, и если вершина видима (`Vertex[i].v=true`), то она выводится. Далее организуется цикл по многогранникам, и для каждого многогранника рисуются его ребра.

При рисовании фигуры используются функции масштабирования — для перехода от бумажных координат к экранным и обратно (листинг 12.8).

Листинг 12.8. Процедуры прямого и обратного масштабирования

```
function II(x: real): integer;
begin
  with FormXZ do
  II:=Trunc((x-x1)*ClientWidth/(x2-x1));
end;

function TFormXZ.XX(i: integer): real;
begin
  XX:=x1+(x2-x1)*I/ClientWidth;
end;

function TFormXZ.YY(j: integer): real;
begin
  YY:=z1-(z2-z1)*(J-ClientHeight)/ClientHeight;
end;

function JJ(y: real): integer;
begin
  with FormXZ do
  JJ:=ClientHeight-Trunc((y-z1)*ClientHeight/(z2-z1));
end;
```

Процедура рисования многогранника на форме `FormXYZ` в повернутой системе координат несколько отличается от методов рисования на остальных дочерних формах. Поворот системы координат реализуется процедурой `TForm3.W_XYZ`, которая использует заранее вычисленные массивы синусов и косинусов (листинг 12.9).

Листинг 12.9. Поворот системы координат

```
procedure TFormXYZ.Rotate(fl: integer; q: word; var x,y,z: real);
var u: real;
begin
  case fl of
    1: begin
         u:=y; y:=u*co[q]-z*si[q]; z:=u*si[q]+z*co[q];
       end;
```

```
    2: begin
         u:=x; x:=u*co[q]-z*si[q]; z:=u*si[q]+z*co[q];
       end;
    3: begin
         u:=x; x:=u*co[q]-y*si[q]; y:=u*si[q]+y*co[q];
       end;
  end;
end;
```

Рисование многогранника (листинг 12.10) начинается с преобразования координат вершин при повороте системы координат на углы `Al` и `Be`. Затем для каждой грани вычисляются координаты нормали `Face[i].A`, `Face[i].B`, `Face[i].C` и проекция единичной нормали `Face[i].n` на ось Z. После сортировки списка выводимых граней, грани прорисовываются методом `Polygon(w)`. Перед рисованием каждой грани в зависимости от состояния свойства `CheckBox2.Checked` устанавливается либо белый цвет (`Brush.Color:=clWhite`; `Pen.Color:=clBlack`), либо, в зависимости от проекции единичной нормали `Face[i].n` на ось `Z`, полутона красного: `Brush.Color:=$00000001*Trunc(Face[p[i]].n*105+150);` `Pen.Color:=$00000001*Trunc(Face[p[i]].n*105+150)` (листинг 12.10).

Листинг 12.10. Процедура рисования многогранника в повернутой системе координат

```
procedure TFormXYZ.Show3D;
var
  i,j,L: integer;
begin
  CountBody:=Length(Body);
  if CountBody=0 then Exit;

  for j:=0 to CountBody-1 do
  with Body[j] do
  for i:=0 to Length(Vertex)-1 do begin
    VertexT[i]:=Vertex[i];
    Rotate(1,Al,VertexT[i].x,VertexT[i].y,VertexT[i].z);
    Rotate(2,Be,VertexT[i].x,VertexT[i].y,VertexT[i].z);
  end;

  with Bitmap.Canvas do begin
    Brush.Style:=bsSolid;
    Brush.Color:=clWhite;
    FillRect(Rect(0,0,ClientWidth,ClientHeight));

    if FormTools.CheckBox2.Checked then begin // грани
      CountFace:=0;
      for j:=0 to CountBody-1 do
      with Body[j] do
```

```
  if Visible then
  for i:=0 to Length(Face)-1 do
  with Face[i] do begin
    Norma(VertexT[p[1]],VertexT[p[2]],VertexT[p[3]],A,B,C,D);
    Zmid:=Z_Mid(j,i);
    if C>0 then begin
      Inc(CountFace);
      SetLength(Faces,CountFace);
      Faces[CountFace-1].IndexFace:=i;
      Faces[CountFace-1].IndexBody:=j;
    end;
  end;

  for i:=1 to CountFace-1 do
  for j:=i downto 1 do
  if Body[Faces[j].IndexBody].Face[Faces[j].IndexFace].ZMid<
    Body[Faces[j-1].IndexBody].Face[Faces[j-1].IndexFace].ZMid
  then begin
    FaceT:=Faces[j]; Faces[j]:=Faces[j-1]; Faces[j-1]:=FaceT;
  end;

  Brush.Style:=bsSolid;
  for i:=0 to CountFace-1 do
  with Faces[i],Body[IndexBody] do begin
    w[1].x:=II(VertexT[Face[IndexFace].p[1]].x);
    w[1].y:=JJ(VertexT[Face[IndexFace].p[1]].y);
    w[2].x:=II(VertexT[Face[IndexFace].p[2]].x);
    w[2].y:=JJ(VertexT[Face[IndexFace].p[2]].y);
    w[3].x:=II(VertexT[Face[IndexFace].p[3]].x);
    w[3].y:=JJ(VertexT[Face[IndexFace].p[3]].y);
    Brush.Color:=$000101*
      Trunc(Face[IndexFace].NormalZ*155+100);
    Pen.Color:=Brush.Color;
    Polygon(w);
    if FormTools.CheckBox3.Checked then begin
      w[1].x:=II(VertexT[Face[IndexFace].p[1]].x);
      w[1].y:=JJ(VertexT[Face[IndexFace].p[1]].y);
      w[2].x:=II(Face[IndexFace].A+
                 VertexT[Face[IndexFace].p[1]].x);
      w[2].y:=JJ(Face[IndexFace].B+
                 VertexT[Face[IndexFace].p[1]].y);
      MoveTo(w[1].x,w[1].y); LineTo(w[2].x,w[2].y);
    end;
  end;
end
```

```
    else begin // ребра
      for j:=0 to CountBody-1 do
      with Body[j] do
      if Visible then begin
        if IndexBody=j then Pen.Color:=clRed
        else Pen.Color:=clBlack;
        L:=Length(Edge);
        for i:=0 to L-1 do
        with Edge[i] do begin
          w[1].x:=II(VertexT[n1].x);
          w[1].y:=JJ(VertexT[n1].y);
          w[2].x:=II(VertexT[n2].x);
          w[2].y:=JJ(VertexT[n2].y);
          MoveTo(w[1].x,w[1].y); LineTo(w[2].x,w[2].y);
        end;
      end;
    end;
  end;
  Canvas.Draw(0,0,Bitmap);
end;
```

12.1.4. Новый многогранник

При нажатии кнопки **Добавить** на вкладке **New** инструментальной формы (рис. 12.2) вызывается процедура `BtnAddClick` и создается новый многогранник, представляющий собой либо пирамиду, либо усеченный конус (листинг 12.11).

Листинг 12.11. Процедура выбора добавления новых тел

```
procedure TFormTools.BtnAddClick(Sender: TObject);
begin
  Case PageControl2.TabIndex of
    0: AddPiramida(StrToFloat(LabeledEdit4.Text),
                   StrToFloat(LabeledEdit5.Text),
                   StrToFloat(LabeledEdit6.Text));
    1: AddCilyndr(StrToFloat(LabeledEdit1.Text),
                  StrToFloat(LabeledEdit2.Text),
                  StrToFloat(LabeledEdit3.Text),
                  SpinEdit1.Value );
  end;
  SetCheckListBox;
  FormMain.Show3D;
end;
```

При добавлении пирамиды необходимо добавить элемент в массив тел `Body` и в этом новом элементе задать координаты четырех вершин, определить 6 ребер и 4 грани (листинг 12.12).

Листинг 12.12. Процедура добавления пирамиды

```
procedure AddPiramida(a,b,c: real);
var Lb,Lp,LL,Ls: word;
begin
  Lb:=Length(Body); Lb:=Lb+1; SetLength(Body,Lb);
  CountBody:=Lb; IndexBody:=CountBody-1;
  with Body[Lb-1] do begin
    Body[Lb-1].Visible:=true;

    Lp:=Length(Vertex); Lp:=Lp+4;
    SetLength(Vertex,Lp); SetLength(VertexT,Lp);
    Vertex[Lp-4]:=SetVertex(0,0,0,false);
    Vertex[Lp-3]:=SetVertex(a,0,0,false);
    Vertex[Lp-2]:=SetVertex(0,b,0,false);
    Vertex[Lp-1]:=SetVertex(0,0,c,false);

    LL:=Length(Edge); LL:=LL+6; SetLength(Edge,LL);
    Edge[Ll-6]:=SetEdge(Lp-4,Lp-3);
    Edge[Ll-5]:=SetEdge(Lp-4,Lp-2);
    Edge[Ll-4]:=SetEdge(Lp-4,Lp-1);
    Edge[Ll-3]:=SetEdge(Lp-3,Lp-2);
    Edge[Ll-2]:=SetEdge(Lp-3,Lp-1);
    Edge[Ll-1]:=SetEdge(Lp-2,Lp-1);

    Ls:=Length(Face); Ls:=Ls+4; SetLength(Face,Ls);
    Face[Ls-4]:=SetSide(Lp-2,Lp-4,Lp-1);
    Face[Ls-3]:=SetSide(Lp-4,Lp-3,Lp-1);
    Face[Ls-2]:=SetSide(Lp-2,Lp-1,Lp-3);
    Face[Ls-1]:=SetSide(Lp-3,Lp-4,Lp-2);
  end;
end;
```

12.1.5. Добавление вершины

При нажатии кнопки `AddButton` вызывается процедура `AddPoint`, в которой сначала организуется цикл по граням с целью поиска грани с тремя отмеченными вершинами. Если такой грани нет, то компонент `ShowMessage` выдает сообщение "Нет выделенной плоскости". Если же такая грань с номером `ip` была найдена, то число вершин `CountVertex` увеличивается на `1`. В зависимости от значения флага `fl`, зада-

ются параметры новой вершины. Флаг `fl` принимает значение `1`, если активизировано дочернее окно с проекцией `XZ`, значение `2` — для окна с проекцией `YZ` и значение `3` — для окна с проекцией `XY`. Так, например, для окна с проекцией `XZ` проекции `X` и `Z` определяются из экранных координат указателя мыши с помощью функций обратного масштабирования `XX()` и `YY()`, а координата `Y` задается, для определенности, как среднее арифметическое координат выделенной грани. Затем число ребер увеличивается на три, и задаются номера начальной и конечной вершины для каждого ребра.

Наконец, на два увеличивается число граней, и выделенная грань разбивается на три части (листинг 12.13).

Листинг 12.13. Процедура добавления новой вершины

```
procedure AddPoint(fl: byte; u,v,w: real);
var Ok: boolean;
    ip,temp,CountVertex,CountEdge,CountSide: integer;
    t: real;
begin
  ip:=-1; Ok:=false;
  with Body[IndexBody] do begin
    while (ip<Length(Face)-1) and not Ok do begin
      Inc(ip);
      Ok:=Vertex[Face[ip].p[1]].Select and
          Vertex[Face[ip].p[2]].Select and
          Vertex[Face[ip].p[3]].Select;
    end;

    if not Ok then ShowMessage('Нет выделенной плоскости')
    else begin
      CountVertex:=Length(Vertex);
      CountVertex:=CountVertex+1;
      SetLength(Vertex,CountVertex);
      SetLength(VertexT,CountVertex);
      case fl of
        1: begin
             with Face[ip] do
             t:=(Vertex[p[1]].y+Vertex[p[2]].y+Vertex[p[3]].y)/3;
             Vertex[CountVertex-1]:=SetVertex(u,v,t,true);
           end;
        2: begin
             with Face[ip] do
             t:=(Vertex[p[1]].x+Vertex[p[2]].x+Vertex[p[3]].x)/3;
             Vertex[CountVertex-1]:=SetVertex(t,v,w,true);
           end;
```

```
        3: begin
             with Face[ip] do
             t:=(Vertex[p[1]].z+Vertex[p[2]].z+Vertex[p[3]].z)/3;
             Vertex[CountVertex-1]:=SetVertex(u,v,t,true);
           end;
      end;

      CountEdge:=Length(Edge);
      SetLength(Edge,CountEdge+3);
      CountEdge:=CountEdge+1;
      Edge[CountEdge-1]:=SetEdge(Face[ip].p[1],CountVertex-1);
      CountEdge:=CountEdge+1;
      Edge[CountEdge-1]:=SetEdge(Face[ip].p[2],CountVertex-1);
      CountEdge:=CountEdge+1;
      Edge[CountEdge-1]:=SetEdge(Face[ip].p[3],CountVertex-1);

      CountSide:=Length(Face);
      SetLength(Face,CountSide+3);
      CountSide:=CountSide+1;
      Face[CountSide-1]:=
        SetSide(Face[ip].p[2],Face[ip].p[3],CountVertex-1);
      CountSide:=CountSide+1;
      Face[CountSide-1]:=
        SetSide(Face[ip].p[3],Face[ip].p[1],CountVertex-1);
      CountSide:=CountSide+1;
      Face[CountSide-1]:=
        SetSide(Face[ip].p[1],Face[ip].p[2],CountVertex-1);
    end;
  end;
end;
```

12.1.6. Переключение инструментов

На инструментальной форме и на ее вкладках расположены 12 кнопок типа `TSpeedButton`, 9 из которых образуют группу (их свойство `GroupIndex` равно `1`). Это значит, что в любой момент времени в нажатом состоянии может быть только одна из них. При нажатии одной из этих кнопок в обработчиках типа `MoveButtonClick` меняется значение переменной `fl_tools` (листинг 12.14). Эта переменная используется в обработчиках нажатия и перемещения указателя мыши на дочерних формах с проекциями. В табл. 12.1 приведены названия кнопок и значения, которые присваиваются переменной `fl_tools` при нажатии этих кнопок.

Листинг 12.14. Процедура изменения *fl_tools*

```
procedure TForm1.MoveButtonClick(Sender: TObject);
begin
  fl_tools:=0;
end;
```

Таблица 12.1. *Значения переменной fl_tools*

Кнопка	Назначение	*Fl_tools*
OffButton	Отключить функции	255
MoveButton	Перемещать вершины	0
SelectButton	Выделять вершины	1
RectButton	Растянуть рисунок до выделенного прямоугольника	2
Div2Button	Уменьшить рисунок вдвое	6
Mult2Button	Увеличить рисунок вдвое	7
AddButton	Добавить точку к выделенной грани	3
MoveXYZButton	Переместить систему координат	4
MovePictButton	Переместить рисунок	5

12.1.7. Выравнивание дочерних окон

Во время выполнения проекта можно изменять размеры и положение любого дочернего окна. Нажатие кнопки мыши в пункте меню **View/AllWind** главной формы приведет к выравниванию всех дочерних окон, при этом они поделят клиентскую часть родительского окна на четыре равных части (листинг 12.15).

Листинг 12.15. Процедура выравнивания дочерних окон

```
procedure TFormMain.AllWind1Click(Sender: TObject);
begin
  FormXZ.Left:=(FormMain.Width-fx) div 2;
  FormXZ.Top:=0;
  FormXZ.Width:=(FormMain.Width-fx) div 2;
  FormXZ.Height:=(FormMain.Height-fy) div 2;
  FormYZ.Top:=0;
  FormYZ.Left:=0;
  FormYZ.Width:=(FormMain.Width-fx) div 2;
  FormYZ.Height:=(FormMain.Height-fy) div 2;
```

```
  FormXY.Left:=0;
  FormXY.Top:=(FormMain.Height-fy) div 2;
  FormXY.Width:=(FormMain.Width-fx) div 2;
  FormXY.Height:=(FormMain.Height-fy) div 2;
  FormXYZ.Left:=(FormMain.Width-fx) div 2;
  FormXYZ.Top:=(FormMain.Height-fy) div 2;
  FormXYZ.Width:=(FormMain.Width-fx) div 2;
  FormXYZ.Height:=(FormMain.Height-fy) div 2;
end;
```

12.1.8. Нажатие кнопки мыши на дочерних формах

Нажатие кнопки мыши и перемещение указателя мыши на каждой дочерней форме обрабатывается процедурами `MouseDown`, `MouseUp`, `MouseMove`. При нажатых кнопках `MoveButton` и `SelectButton`, т. е. при перемещении и выделении вершины, необходимо определить номер захваченной точки. В процедурах `MouseDown` это реализовано вызовом процедуры `FindPoint`. В процедуре `FindPoint` определяется номер точки, расстояние до которой от точки `(u,v)` минимально. Функция `Dist(u,v)` определяет это расстояние на каждой проекции (листинг 12.16).

Листинг 12.16. Процедура определения ближайшей точки

```
procedure FindPoint(II,JJ: TIJ; fl: byte; x,y: integer);
var i,m,k,Min: integer;

  function Dist(i,k: byte; u,v: integer): integer;
  begin
    with Body[k].Vertex[i] do
    case fl of
      1: Dist:=Sqr(II(y)-u)+Sqr(JJ(z)-v);
      2: Dist:=Sqr(II(x)-u)+Sqr(JJ(z)-v);
      3: Dist:=Sqr(II(y)-u)+Sqr(JJ(x)-v);
    end;
  end;

begin
  IndexPoint:=0; IndexBody:=0;
  Min:=Dist(0,0,x,y);
  for k:=0 to CountBody-1 do
  with Body[k] do
  if visible then
  for i:=0 to Length(Vertex)-1 do begin
```

```
    M:=Dist(i,k,x,y);
    if m<Min then begin
      IndexPoint:=i; Min:=m; IndexBody:=k;
    end;
  end;
end;
```

В процедуре `FormMouseDown` при нажатой кнопке `MoveButton` (`fl_tools=0`) поднимается флаг перемещения `drawing:=true`. При нажатой кнопке `SelectButton` (`fl_tools=1`) меняется значение поля `Vp[].v`: `Vp[NumPoint].v:=not Vp[NumPoint].v`. Это поле используется при рисовании многогранника и служит признаком рисования вершины. При нажатой кнопке `RectButton` (`fl_tools=2`) поднимается флаг перемещения и начинается рисование фокусного прямоугольника. При нажатой кнопке `AddButton` (`fl_tools=3`) вызывается метод `Form1.AddPoint` основной формы для добавления новой точки. Приведем код всей процедуры, обрабатывающей начало перемещения указателя мыши (листинг 12.17).

Листинг 12.17. Процедура нажатия кнопки мыши на дочерней форме

```
procedure TFormXZ.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  if fl_tools in [0,1] then
    FindPoint(II,JJ,2,x,y);
  case fl_tools of
    0: drawing:=true;
    1: with Body[IndexBody] do
       Vertex[IndexPoint].Select:=
         not Vertex[IndexPoint].Select;
    2: begin
         drawing:=true;
         i0:=x; j0:=y; it:=x; jt:=y;
         Canvas.DrawFocusRect(Rect(i0,j0,it,jt));
       end;
    3: AddPoint(1,XX(x),0,YY(y));
    4,5:
       begin
         drawing:=true;
         i0:=x; j0:=y; it:=x; jt:=y;
         Canvas.Pen.Mode:=pmNotXor;
       end;
  end;
  if not (fl_tools in [4,5]) then FormMain.Show3D;
end;
```

12.1.9. Обработка перемещения указателя мыши на формах

Обработка перемещения указателя мыши возможна, если поднят флаг перемещения `drawing:=true`. Флаг перемещения поднимается при `fl_tools`, равном `0`, `2`, `4` или `5`, т. е. при нажатии следующих кнопок:

- `MoveButton` — перемещать вершины;
- `RectButton` — растянуть рисунок до выделенного прямоугольника;
- `MoveXYZButton` — переместить систему координат;
- `MovePictButton` — переместить рисунок.

Основное назначение обработчика события при перемещении указателя мыши — это "нарисовать-стереть". Эти функции реализуются разными способами:

- при нажатой кнопке `MoveButton` фиксируется новое положение вершины, и перерисовываются изображения во всех окнах вызовом метода `Form1.Show3D`;
- при нажатой кнопке `RectButton` повторный вызов `DrawFocusRect` стирает фокусный прямоугольник, изменяются временные координаты (`it`, `jt`) и вновь рисуется фокусный прямоугольник `DrawFocusRect`;
- при нажатых кнопках `MoveXYZButton` или `MovePictButton` в режиме перемещения системы координат или рисунка повторное рисование линии стирает ее (этот эффект уже обсуждался в *разд. 7.6*), так как предварительно установлен режим `Canvas.Pen.Mode:=pmNotXor`, изменяются временные координаты (`it`, `jt`) и вновь рисуется линия.

Обработчик перемещения указателя мыши представлен в листинге 12.18. Три перечисленных способа реализации функций "нарисовать-стереть" соответствуют трем вариантам в операторе выбора `case` и определяются со значениями переменной `fl_tools`.

Листинг 12.18. Процедура обработки перемещения указателя мыши на форме *XZ*

```
procedure TFormXZ.FormMouseMove(Sender: TObject;
  Shift: TShiftState; X,Y: Integer);
begin
  if drawing then
  case fl_tools of
    0: with Body[IndexBody] do begin
         Vertex[IndexPoint].x:=XX(X);
         Vertex[IndexPoint].z:=YY(Y);
         FormMain.Show3D;
       end;
```

```
    2:  with Canvas do begin
          DrawFocusRect(Rect(i0,j0,it,jt));
          it:=x; jt:=y;
          DrawFocusRect(Rect(i0,j0,it,jt));
        end;
    4,5:with Canvas do begin
          MoveTo(i0,j0); LineTo(it,jt);
          it:=x; jt:=y;
          MoveTo(i0,j0); LineTo(it,jt);
        end;
  end;
end;
```

Итак, при нажатой кнопке `MoveButton` (`fl_tools=0`) меняются координаты захваченной точки и перерисовываются все окна `Form1.Show3D`. При нажатой кнопке `RectButton` (`fl_tools=2`) первый вызов метода `DrawFocusRect` стирает старый прямоугольник, а второй вызов рисует прямоугольник в новом месте.

Процедура обработки перемещения указателя мыши для `fl_tools=0` на дочерней форме с проекцией `XYZ` несколько отличается от аналогичных обработчиков для остальных проекций. При перемещении мыши происходит изменение углов поворота системы координат: один угол пропорционален расстоянию от точки, в которой находится указатель мыши до центра формы, а второй угол равен углу между положительным направлением оси X и прямой, соединяющей центр формы с точкой, в которой находится указатель мыши (листинг 12.19).

Листинг 12.19. Процедура обработки перемещения мыши на форме *XYZ*

```
procedure TFormXYZ.Image1MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
var a,b,Alf,Bet: real;
begin
  if drawing then
  case fl_tools of
    0: begin
         a:=x-Width div 2; b:=y-Height div 2;
         Alf:=Abs(ArcTan2(b,a));
         Bet:=Sqrt(Sqr(a/10)+Sqr(b/10));
         Al:=Round(Alf*180/Pi) mod 72;
         Be:=Round(Bet) mod 72;
         Show3D;
       end;
    2: with Canvas do begin
         DrawFocusRect(Rect(i0,j0,it,jt));
         it:=x; jt:=y;
```

```
        DrawFocusRect(Rect(i0,j0,it,jt));
      end;
  end;
end;
```

При отпускании кнопки мыши вызывается процедура `FormMouseUp`, в которой реализованы завершающие действия. При нажатой кнопке `RectButton` (`fl_tools=2`) стирается фокусный прямоугольник и меняются габариты окна вывода (`x1, x2, z1, z2`). При перемещении окна вывода или тела относительно системы координат (`fl_tools=4` или `fl_tools=5`) меняются габариты окна или координаты вершин тела (листинг 12.20).

Листинг 12.20. Процедура обработки отпускания кнопки мыши

```
procedure TFormXZ.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var dx,dy: real;
    i: integer;
begin
  drawing:=false;
  case fl_tools of
    2: begin
         Canvas.DrawFocusRect(Rect(i0,j0,it,jt));
         Rew(i0,it); Rew(j0,jt);
         x1:=XX(i0); z1:=YY(jt);
         x2:=XX(it); z2:=YY(j0);
       end;
    4,5:
       with Canvas do begin
         MoveTo(i0,j0); LineTo(it,jt); Pen.Mode:=pmCopy;
         dx:=XX(it)-XX(i0); dy:=YY(jt)-YY(j0);
         if fl_tools=4 then begin
           x1:=x1-dx; x2:=x2-dx;
           z1:=z1-dy; z2:=z2-dy;
         end
         else
         with Body[IndexBody] do
         for i:=0 to Length(Vertex)-1 do begin
           Vertex[i].x:=Vertex[i].x+dx;
           Vertex[i].z:=Vertex[i].z+dy;
         end;
       end;
  end;
  FormMain.Show3D;
end;
```

12.2. Редактор для топологически связанных трехмерных тел

12.2.1. Структура данных

В геоинформационных системах (ГИС) существует проблема описания дву- и трехмерных тел (многоугольников на плоскости и многогранников в пространстве), топологически связанных между собой. Это означает, что граничные вершины и ребра многоугольников (на плоскости) или граничные вершины, ребра и грани (в трехмерном пространстве) должны быть общими для соседних тел. В этом случае изменение координат общей вершины двух соседних тел не приводит к появлению "ничейной" области между телами. На плоскости эта задача решена во многих ГИС, например, в ArcGIS (**www.dataplus.ru/win/ESRI**). В проекте "Трехмерный редактор многогранников", рассматриваемом в данном разделе, предлагается структура данных, описывающая систему топологически связанных тел для трехмерного пространства.

Стандартное описание системы тел в трехмерном пространстве выглядит следующим образом (рис. 12.5): эта система состоит из массива тел-многогранников. Для каждого тела (`Body`) определены свои независимые массивы вершин (`Vertex`), ребер (`Edge`) и граней (`Side`), опирающихся на вершины. Такое описание данных использовалось ранее, например, в проектах "Платоновы тела" (*см. разд. 11.1*) и "Редактор многогранников" (*см. разд. 12.1*). Ясно, что такая древовидная структура данных проще обрабатывается, чем граф, но независимость вершин приводит к появлению "ничейных" зон при редактировании вершин соседних тел.

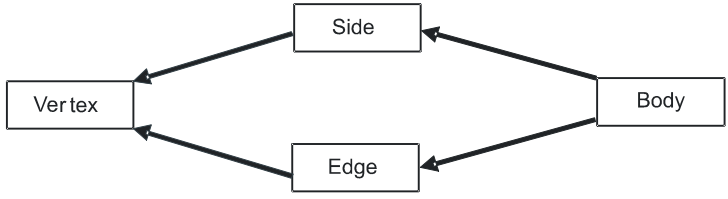


Рис. 12.5. Стандартная структура для описания тел

12.2.2. Структура данных проекта

В проекте "Трехмерный редактор многогранников" предлагается структура данных (рис. 12.6) в виде массивов вершин, ребер, граней и тел (листинг 12.21).

Предполагается, что каждое тело — многогранник, а каждая грань — треугольник. Под проектом мы будем понимать всю совокупность данных о телах, гранях, ребрах и вершинах, связанных между собой.

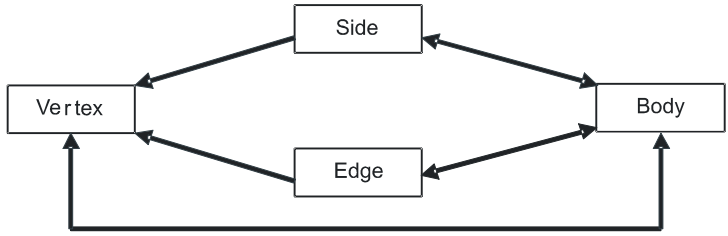


Рис. 12.6. Структура для описания тел в виде графа в проекте "Трехмерный редактор многогранников"

Листинг 12.21. Структура проекта "Трехмерный редактор многогранников"

```
TProjectBody=record
  uG1,vG1,wG1,uG2,vG2,wG2: single; // куб в пространстве 3D
  Vertexs  : array of TVertex;     // массив вершин всех тел
  Edge     : array of TEdge;       // массив ребер всех тел
  Face     : array of TFace;       // массив граней всех тел
  Body     : array of TBody;
end;
```

Между всеми объектами существует отношение "многие ко многим":

- поверхность каждого тела состоит из многих граней. Каждая грань может принадлежать поверхности нескольких тел;
- каждое тело опирается на несколько ребер. Каждое ребро может принадлежать нескольким телам;
- каждое тело опирается на несколько вершин. Каждая вершина может принадлежать нескольким телам.

Структура проекта (листинг 12.21) соответствует структуре данных в виде ориентированного графа (рис.12.6), так как описания и граней и ребер содержат ссылки на вершины.

Тело

Для описания одного тела будем использовать следующую структуру (листинг 12.22).

Листинг 12.22. Структура для описания тела

```
TBody=record
  Name        : string;              // имя тела
  Visible     : boolean;             // признак видимости тела
  IndexFace   : array of TIndexFace; // номера граней тела
  IndexVertex : array of word;       // номера вершин тела
end;
```

Тип номера грани тела `TIndexFace` определен в листинге 12.23 и содержит два поля: номер грани `Index` и знак направления внешней нормали `SignNorm`.

Листинг 12.23. Номера граней

```
  TIndexFace=record
    Index       : word;                // номер грани
    Norm        : shortint;            // направление нормали
  end;
```

Структура грани

Структура грани определена в листинге 12.24. Поле `p: array[0..2] of word` показывает на номера трех вершин. Обратную связь с телами реализует массив `NumBodys: TANumBodys`.

Листинг 12.24. Структура грани

```
TFace=record
  IndexEdge   : array[0..2] of word; // номера ребер
  AIndexVertex: array of word;       // номера точек
  A,B,C,D     : single;      // коэффициенты плоскости
  IndexBody   : TAIndexBody; // тела, в которые входит грань
  Po          : TAVertex;    // точки пересечения на грани
end;
```

Структура ребра

Структура ребра определена в листинге 12.25. Поля `p1` и `p2` показывают на номера двух вершин. Обратную связь с телами реализует массив `NumBodys: TANumBodys`.

Листинг 12.25. Структура ребра

```
TEdge=record
  p1,p2: word;               // номера вершин
  IndexBody   : TAIndexBody; // номера тел, в которые входит ребро
  IndexOfEdge : byte;        // индекс ребра
end;
```

Структура вершины

Структура вершины определена в листинге 12.26. Поля `x`, `y`, `z` показывают координаты вершины. Обратную связь с телами реализует массив `NumBodys: TANumBodys`.

Листинг 12.26. Структура вершины

```
TVertex=record
  x,y,z       : single;
  IndexEdgeOrFace: word;      // номер ребра или номер вершины
  typeVertex  : byte;         // 0 - вершина, 1 - ребро, 2 - внутри
  IndexBody   : TAIndexBody; // тела, в которые входит точка
  IndexOfVertex: byte;        // индекс точки
end;
```

Поля `IndexBody`, показывающие, в какие тела входят грани, ребра и вершины, во всех структурах являются избыточными. Но они позволяют значительно быстрее реализовывать алгоритмы рисования и поиска.

12.2.3. Трехмерный редактор многогранников

Для тестовой отработки методов предлагаемой структуры реализован трехмерный редактор, в который входит 12 модулей:

- `UnitMain` — главная форма;
- `UnitXZ` — дочерняя форма с проекцией XZ;
- `UnitYZ` — дочерняя форма с проекцией YZ;
- `UnitXY` — дочерняя форма с проекцией XY;
- `Lib` — модуль с основными процедурами;
- `UnitTools` — инструментальная форма;
- `UnitXYZ` — дочерняя форма с проекцией XYZ с повернутой системой координат;
- `UnitSurface` — вспомогательная форма для вывода списков граней и их вершин;
- `UnitProperty` — форма для вывода списков граней, ребер и вершин;
- `LibTr` — модуль с процедурами триангуляции;
- `UnitOpenGL` — форма с проекцией XYZ с повернутой системой координат, использующая библиотеку OpenGL;
- `UnitToolsOpenGL` — инструментальная форма для работы с OpenGL.

Структура проекта "Трехмерный редактор многогранников" показана на рис. 12.7.

В трех окнах главной формы `FormXY`, `FormXZ`, `FormYZ` изображаются проекции, в четвертом окне `FormXYZ` — повернутая система координат (рис. 12.8). В трех окнах с проекциями можно перемещать вершины и тела (режим переключается на вкладке **Edit** формы **Tools**). В четвертом окне тела можно вращать с помощью указателя мыши. Данные можно записывать и читать.

Инструментальная форма `FormTools` (с внешним именем **Tools**) содержит 8 вкладок: **New** (рис. 12.9), **Edit** (рис. 12.10), **Bodys** (рис. 12.11), **Общие** (рис. 12.12), **Операции** (рис. 12.13), **Vertex** (рис. 12.14), **Edge** (рис. 12.15), **Face** (рис. 12.16).

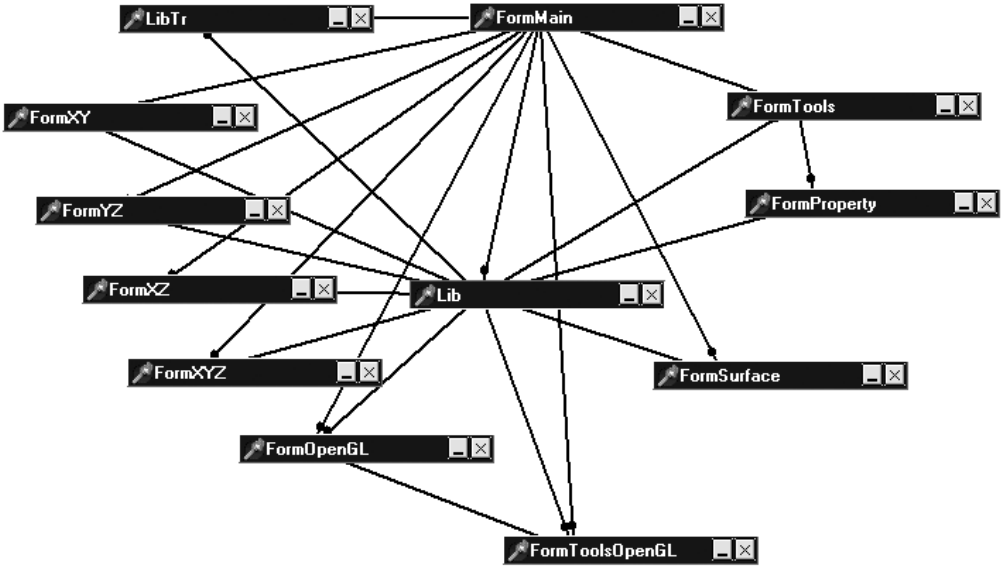


Рис. 12.7. Структура проекта "Трехмерный редактор многогранников"

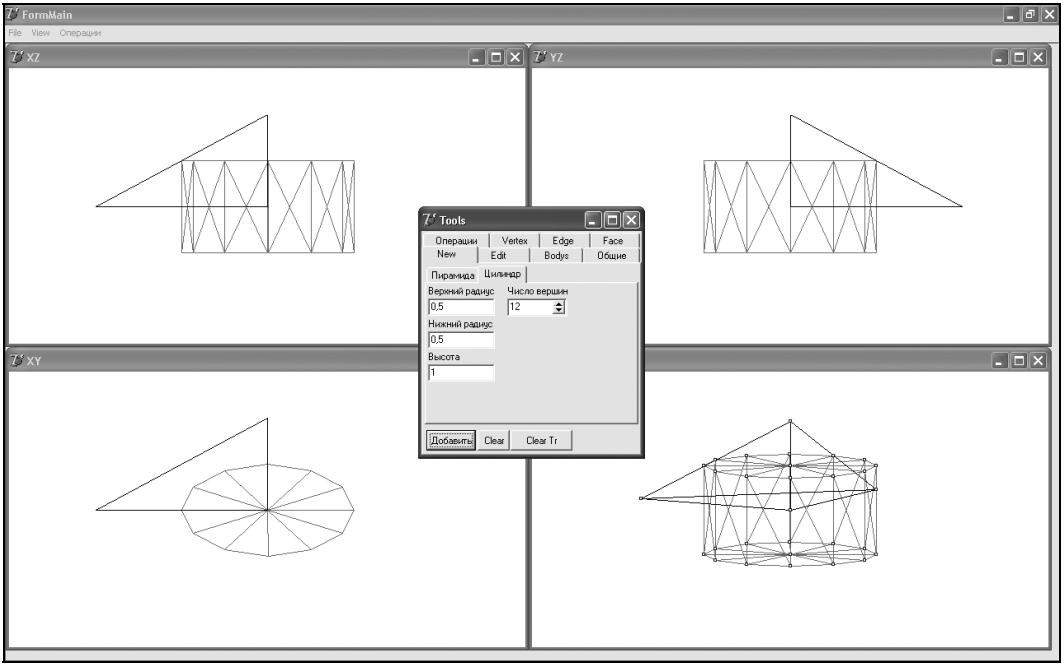


Рис. 12.8. Внешний вид проекта "Трехмерный редактор многогранников"

На вкладке **New** формы **Tools** (рис. 12.9) можно добавлять новые тела — пирамиды и цилиндры.

Рис. 12.9. Вкладка **New** формы **Tools**

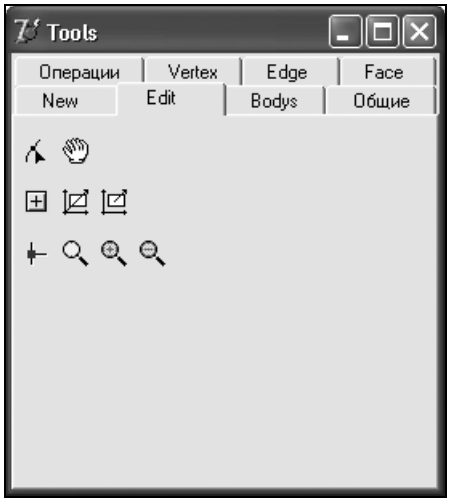


Рис. 12.10. Вкладка **Edit** формы **Tools**

На вкладке **Edit** формы **Tools** (рис. 12.10) можно нажатим кнопок типа TSpeedButton выбирать режимы перемещения тел или точек.

На вкладке **Bodys** формы **Tools** (рис. 12.11) можно выделять одно из тел и двойным щелчком указателя мыши изменять видимость тел.

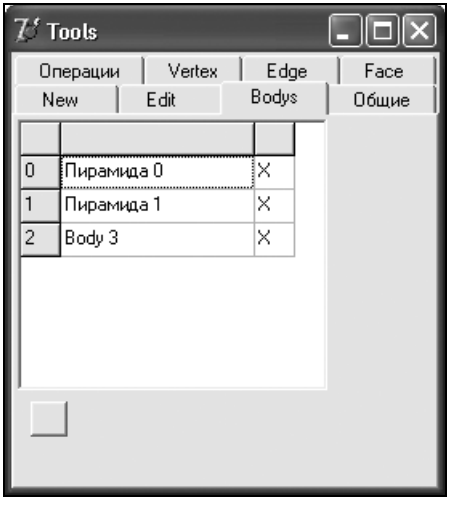


Рис. 12.11. Вкладка **Bodys** формы **Tools**

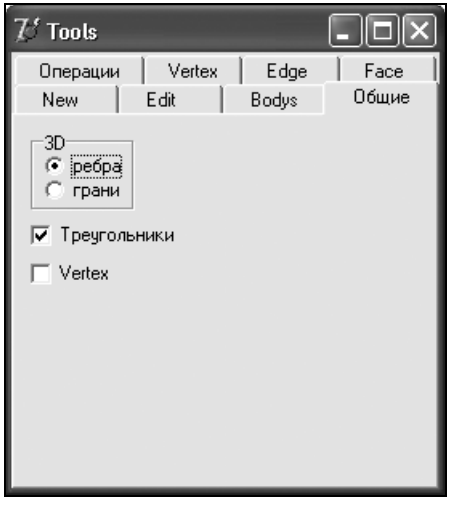


Рис. 12.12. Вкладка **Общие** формы **Tools**

На вкладке **Общие** формы **Tools** (рис. 12.12) можно менять видимость треугольников триангуляции, номеров вершин и переключать режим рисования (ребра / грани).

На вкладке **Операции** формы **Tools** (рис. 12.13) можно выбрать два тела и одну из 16-ти бинарных операций для этих тел. Нажатием кнопки **Пересечение** можно найти точки пересечения граней и ребер выбранных тел, разбить ребра и треугольные грани на новые треугольники. Нажатием кнопки **Add** создается новое тело — результат выбранной бинарной операции, в котором собираются соответствующие вершины, ребра и грани.

На вкладке **Vertex** (рис. 12.14) приведены значения элементов массива вершин `Vertex`.

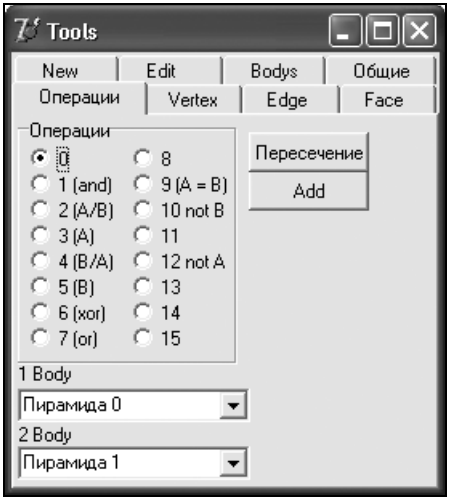


Рис. 12.13. Вкладка **Операции** формы **Tools**

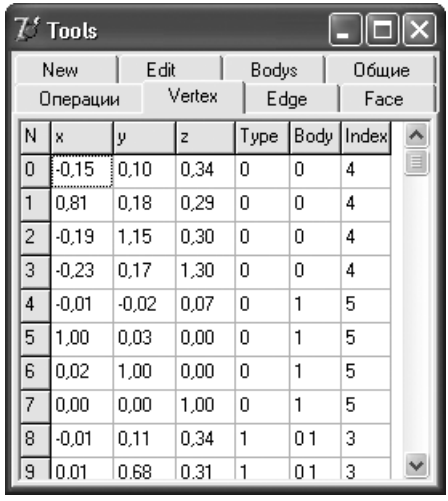


Рис. 12.14. Вкладка **Vertex** формы **Tools**

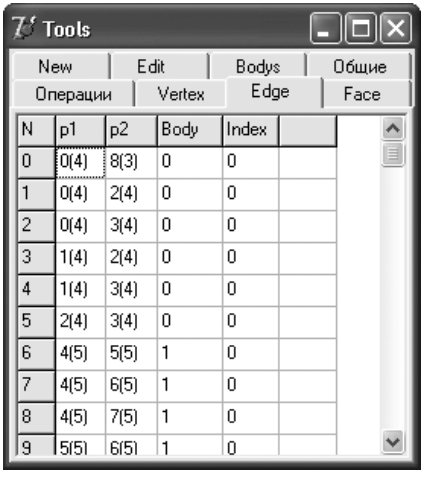


Рис. 12.15. Вкладка **Edge** формы **Tools**

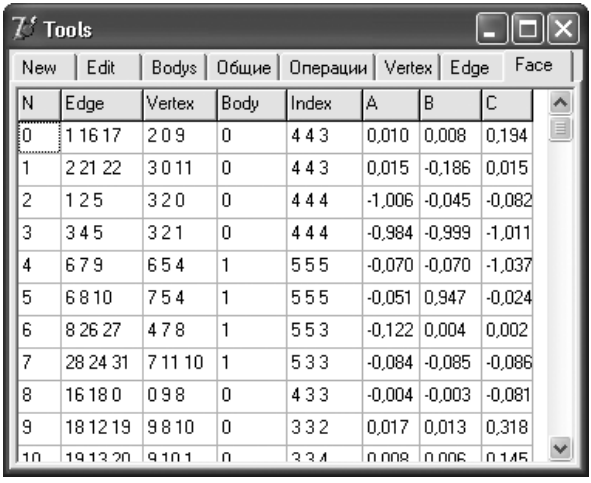


Рис. 12.16. Вкладка **Face** формы **Tools**

На вкладке **Edge** (рис. 12.15) приведены значения элементов массива ребер `Edge`.

На вкладке **Face** (рис. 12.16) приведены значения элементов массива граней `Face`.

12.2.4. Пересечение двух тел

Вид формы проекта "Трехмерный редактор многогранников", на которой решается задача нахождения вершин и ребер пересечения двух тел, показан на рис. 12.17.

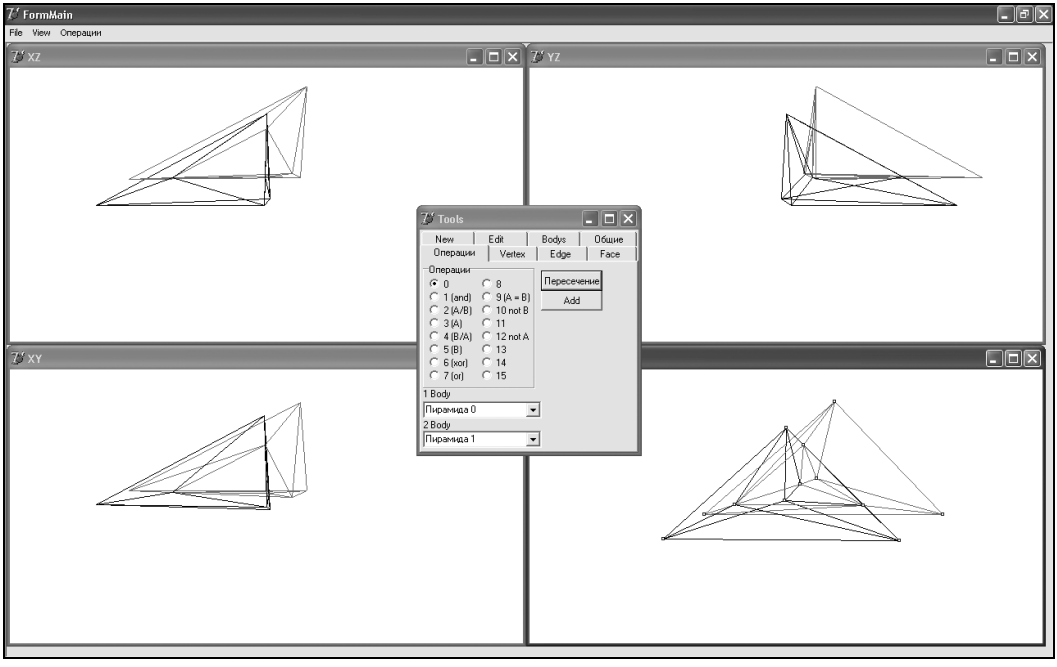


Рис. 12.17. Пересечение двух тел в проекте "Трехмерный редактор многогранников"

На рис. 12.18 показана пирамида с вершинами 0, 1, 2, 3, ребра которой пересеклись с гранями другой пирамиды (на рисунке показана только ее часть, "врезавшаяся" в первую пирамиду) по выделенным точкам 8, 9, 10, 11. Точки 8 и 10 расположены на ребре 01, точка 9 — на грани 012, точка 11 — на грани 013. Каждая из этих граней разбита на пять новых граней.

Задача разбиения на треугольники и добавления новых ребер в общем случае не тривиальна. На рис. 12.19 показан случай, когда в грань 012 по треугольнику 345 "врезалось" второе тело. Грань 012 приходится разбивать на шесть треугольников.

Эта задача сводится к построению триангуляции по заданным точкам, но при этом недопустима "правильная" в смысле Делоне триангуляция, если хотя бы одна из сторон треугольника пересекает полигон пересечения тел, как это показано на

рис. 12.20. Как известно, триангуляция Делоне предполагает соединение между собой набора точек непересекающимися отрезками прямых линий таким образом, чтобы сформированные треугольники стремились к равноугольности [103]. В случае, показанном на рис. 12.20, линия триангуляции 23 проходит через полигон пересечения тел 345, т. е. проходит через "отверстие", образованное в грани 012 в результате такого пересечения.

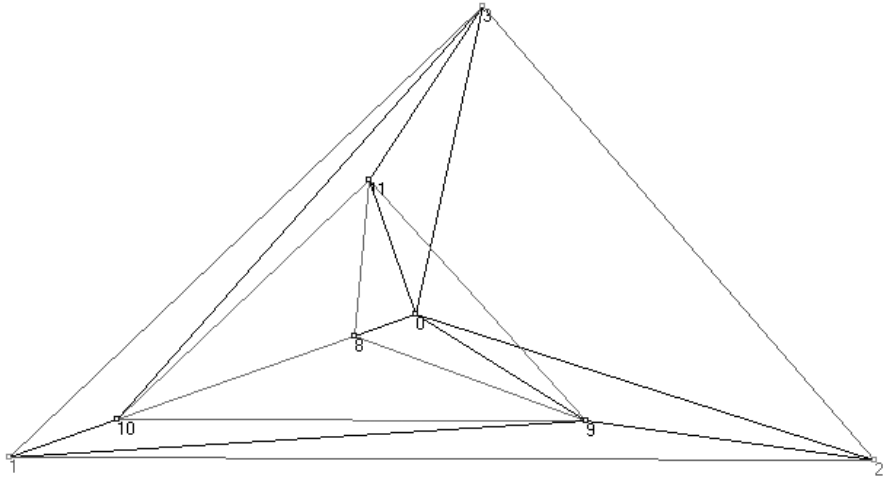


Рис. 12.18. Пирамида с добавленными точками и ребрами

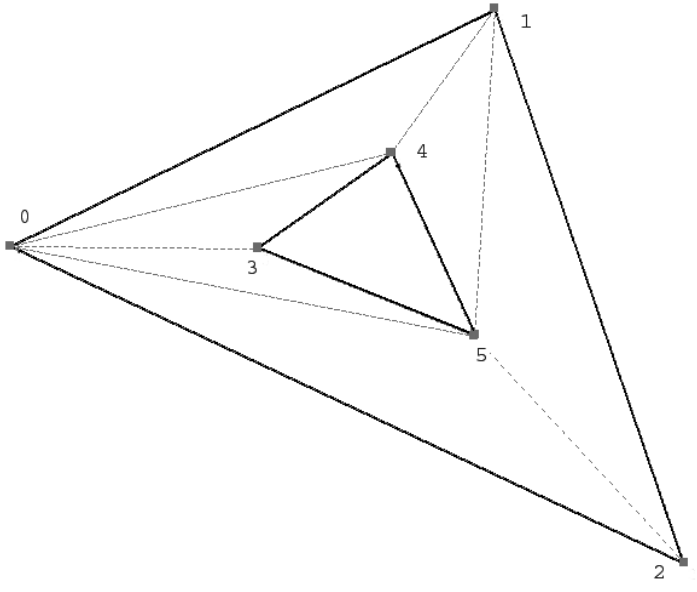


Рис. 12.19. Один из вариантов разбиения грани

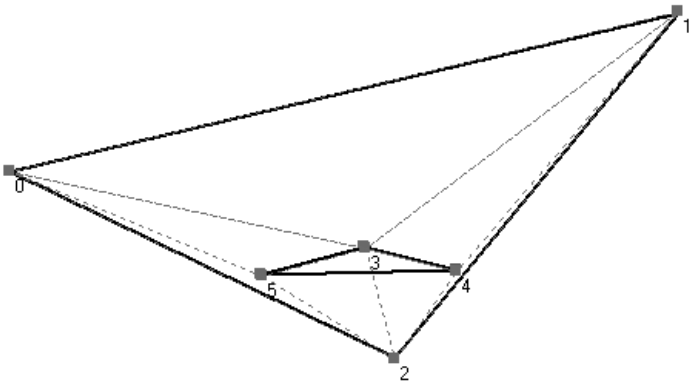


Рис. 12.20. Недопустимая триангуляция грани

В общем случае область пересечения грани с телом может быть невыпуклой и неодносвязной (рис. 12.21).

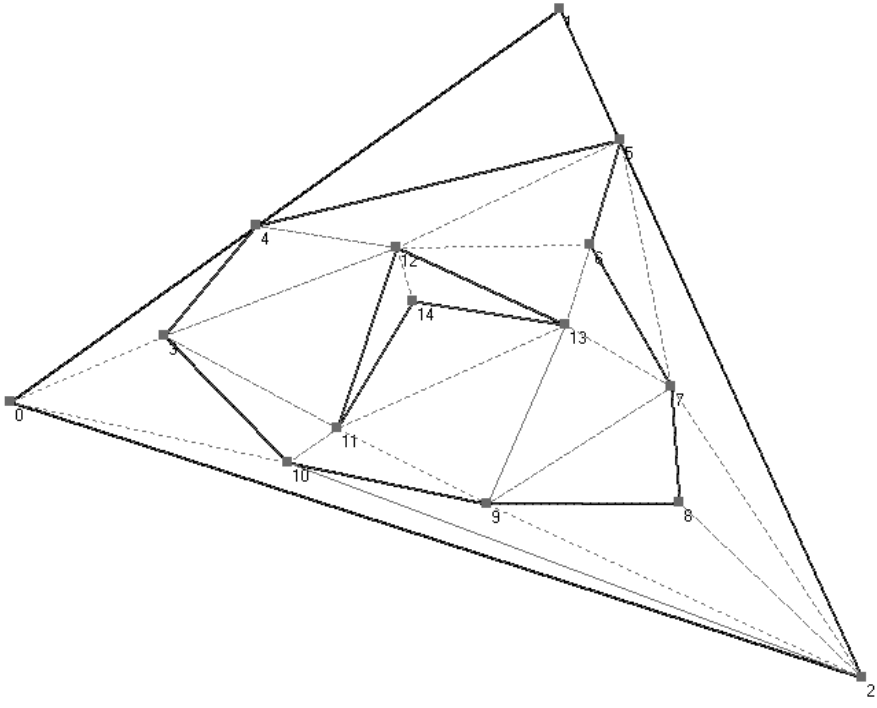


Рис. 12.21. Невыпуклая неодносвязная область пересечения грани с телом

Стороны треугольников триангуляции не должны пересекать контура пересечения грани с телом. Более подробное обсуждение этой проблемы отложим до *главы 14*, в которой изложены различные алгоритмы триангуляции.

А пока вернемся к задаче определения точек пересечения двух тел с номерами n1 и n2. Для этого предназначена процедура SetPointToBody (листинг 12.27).

Листинг 12.27. Определение точек пересечения двух тел

```
procedure SetPointToBody(n1,n2: word); // n1,n2 - номера тел
var
  i,j,Ls,L: integer;
  fl: byte;
  //Q: TVertex;
  Err: byte;
begin
  with ProjectBody do begin
    if Length(Body)=0 then Exit;
    Ls:=Length(Face);
    for i:=0 to Ls-1 do SetABCD(i);
    // найти точки пересечения граней тела n1 с ребрами грани n2
    i:=-1;
    while i<Length(Body[n1].IndexFace)-1 do begin
      Inc(i); j:=-1;
      while j<Length(Body[n2].IndexFace)-1 do begin
        Inc(j);
        if Body[n1].IndexFace[i].Index<>
           Body[n2].IndexFace[j].Index then
          SetPointToFace(n1,n2, Body[n1].IndexFace[i].Index,
                                Body[n2].IndexFace[j].Index);
      end;
    end;

    // расставить IndexOfVertex для Vertex
    Ls:=Length(Body[n1].IndexVertex);
    if Ls>0 then
    for i:=0 to Ls-1 do
      GetIndexOfVertex(Vertex[Body[n1].IndexVertex[i]],n2,0,Err);
    Ls:=Length(Body[n2].IndexVertex);
    if Ls>0 then
    for i:=0 to Ls-1 do
      GetIndexOfVertex(Vertex[Body[n2].IndexVertex[i]],n1,1,Err);

    with Body[n1] do begin
      Ls:=Length(IndexFace);
      for i:=0 to Ls-1 do begin
        L:=Length(Face[IndexFace[i].Index].AIndexVertex);
        if L>3 then begin
          fl:=GetFl(Face[IndexFace[i].Index]);
```

```
          Triangle(fl,n1,IndexFace[i].Index);
        end;
      end;
    end;
    with Body[n2] do begin
      Ls:=Length(IndexFace);
      for i:=0 to Ls-1 do begin
        L:=Length(Face[IndexFace[i].Index].AIndexVertex);
        if L>3 then begin
          fl:=GetFl(Face[IndexFace[i].Index]);
          Triangle(fl,n2,IndexFace[i].Index);
        end;
      end;
    end;
  end;
end;
```

Вычисления призводятся по следующему алгоритму из 4 шагов:

1. Процедура `SetABCD(i)` вычисляет коэффициенты `A`, `B`, `C`, `D` уравнения плоскости для *i*-ой грани `Face[i]`.
2. Процедура `SetPointToFace(nb1,nb2: word; n1,n2: word)` определяет точки пересечения ребер грани `n1` тела `nb1` с гранями тела `nb2` и наоборот — точки пересечения ребер грани `n2` тела `nb2` с гранями тела `nb1`.
3. Процедурой `GetIndexOfVertex` расставляются индексы для вершин первого и второго тела.
4. Процедура `Triangle` выполняет триангуляцию для всех граней первого и второго тела, у которых число вершин больше трех.

Определение точек пересечения ребер с гранями

Для определения точек пересечения ребер с гранями предназначена процедура `SetPointToFace` (листинг 12.28).

Листинг 12.28. Процедура определения точек пересечения ребер с гранями

```
procedure SetPointToFace(nb1,nb2: word; n1,n2: word);
// nb1,nb2 - Numbe Body n1,n2 - NumbeFace
// находит точки пересечения .Po[] двух граней
var
  i,{L1,L2,}L: integer;
  inFace: boolean;
  n,k: word;
  t: array of word;
```

```
begin
  with ProjectBody do begin
    k:=0;
    with Face[n1] do
    for i:=0 to 2 do begin
      // Ищет пересечение грани Face2 и ребра
      // на точках m1,m2 грани Face1
      SetNewPoint(nb1,nb2,IndexEdge[i],
                  Edge[IndexEdge[i]].p1,Edge[IndexEdge[i]].p2,
                  Face[n1],Face[n2],n,inFace);
      if inFace then begin
        Inc(k); SetLength(t,k); t[k-1]:=n;
      end;
    end;
    if k=2 then begin
      with Face[n1] do begin
        L:=Length(TempEdge); Inc(L); SetLength(TempEdge,L);
        TempEdge[L-1].p1:=t[0];
        TempEdge[L-1].p2:=t[1];
      end;
      with Face[n2] do begin
        L:=Length(TempEdge);
        Inc(L);
        SetLength(TempEdge,L);
        TempEdge[L-1].p1:=t[0];
        TempEdge[L-1].p2:=t[1];
      end;
    end;

    k:=0;
    with Face[n2] do
    for i:=0 to 2 do begin
      // Ищет пересечение грани Face1 и ребра
      // на точках m1,m2 грани Face2
      SetNewPoint(nb1,nb2,IndexEdge[i],
                  Edge[IndexEdge[i]].p1,Edge[IndexEdge[i]].p2,
                  Face[n2],Face[n1],n,inFace);
      if inFace then begin
        Inc(k); SetLength(t,k); t[k-1]:=n;
      end;
    end;
    if k=2 then begin
      with Face[n1] do begin
        L:=Length(TempEdge); Inc(L); SetLength(TempEdge,L);
        TempEdge[L-1].p1:=t[0];
```

```
        TempEdge[L-1].p2:=t[1];
      end;
      with Face[n2] do begin
        L:=Length(TempEdge); Inc(L); SetLength(TempEdge,L);
        TempEdge[L-1].p1:=t[0];
        TempEdge[L-1].p2:=t[1];
      end;
    end;
  end;
end;
```

Процедура `SetPointToFace` работает по следующему алгоритму: для каждого из двух тел для каждой грани процедура `SetNewPoint` ищет пересечение грани `Face2` и ребра на точках `m1`, `m2` грани `Face1`. Если получаются две точки пересечения грани с гранью, то в массив добавляется ребро пересечения граней. В последующем при выполнении триангуляции необходимо будет учитывать ограничения на способы разбиения такой грани из-за наличия в ней дополнительного ребра.

Поиск точек пересечения грани и ребра

Процедура поиска точки пересечения грани и ребра `SetNewPoint` представлена в листинге 12.29. В ней анализируются следующие случаи: точка пересечения может совпадать с вершиной, находиться на ребре и внутри грани.

Листинг 12.29. Пересечения грани и ребра

```
// Ищет пересечение грани Face2 и ребра
// на точках m1,m2 грани Face1
procedure SetNewPoint(
            NewIndexEdge: word;     // номер ребра
            m1,m2: word;            // номера точек ребра 1 грани
            var Face1: TFace;       //
            var Face2: TFace        // грани
            );
const Eps=1e-6;
var t,Q1,Q2: single;
    P1,P2,Pq: TVertex;
    S,S1,S2,S3: single;
    i: integer;
    n: word;
begin
  with ProjectBody do begin
    P1:=Vertex[m1]; P2:=Vertex[m2];
  end;
  with ProjectBody,Face2 do begin
    Q1:=A*P1.x+B*P1.y+C*P1.z;
```

```
Q2:=A*P2.x+B*P2.y+C*P2.z;
if Q1<>Q2 then begin
  t:=(-D-Q1)/(Q2-Q1);
  Pq.x:=(P2.x-P1.x)*t+P1.x;
  Pq.y:=(P2.y-P1.y)*t+P1.y;
  Pq.z:=(P2.z-P1.z)*t+P1.z;
  if PointInTr(Pq,Vertex[AIndexVertex[0]],
     Vertex[AIndexVertex[1]],Vertex[AIndexVertex[2]])
  then begin
    Ok:=(-Eps<t) and (t<1+Eps);
    if Ok then begin
      if not FindPointInPo(Pq,Face1.Po,n) then begin
        if (Eps<t) and (t<1-Eps) then begin // внутри
          Face1.Po[n].typeVertex:=1;        // на ребре
          Face1.Po[n].IndexEdgeOrFace:=NewIndexEdge;
        end
        else
        if Abs(t)<Eps then begin            // в вершине P1
          Face1.Po[n].typeVertex:=0;        // вершина
          Face1.Po[n].IndexEdgeOrFace:=m1;
        end
        else
        if Abs(t-1)<Eps then begin          // в вершине P2
          Face1.Po[n].typeVertex:=0;        // вершина
          Face1.Po[n].IndexEdgeOrFace:=m2;
        end;
      end;

      if not FindPointInPo(Pq,Face2.Po,n) then begin
        with ProjectBody do begin
          i:=-1; Ok:=false;
          while (i<2) and not Ok do begin
            Inc(i);
            Ok:=Dist(Pq,Vertex[AIndexVertex[i]])<Eps;
          end;
          if Ok then begin
            Po[n].typeVertex:=0;    // вершина
            Po[n].IndexEdgeOrFace:=AIndexVertex[i];
          end
          else begin
            i:=-1; Ok:=false;
            while (i<2) and not Ok do begin
              Inc(i);
              P1:=Vertex[Edge[IndexEdge[i]].p1];
              P2:=Vertex[Edge[IndexEdge[i]].p2];
```

```
                Ok:=DistLine(Pq,P1,P2)<Eps;
              end;
              if Ok then begin  // на ребре
                Po[n].typeVertex:=1;
                Po[n].IndexEdgeOrFace:=IndexEdge[i];
              end
              else begin        // внутри
                Po[n].typeVertex:=2;
                Po[n].IndexEdgeOrFace:=$FFFF;
              end;
            end;
          end;
        end;
      end;
    end;
  end;
end;
```

Прежде чем добавлять точку `Pq` в массив точек `Po`, необходимо убедиться, что точка еще не принадлежит этому массиву. Такую проверку выполняет функция `FindPointInPo` (листинг 12.30).

Листинг 12.30. Проверка принадлежности точки массиву точек

```
function FindPointInPo(Pq: TVertex; var Po: TAVertex;
                       var n: word): boolean;
const Eps=1e-6;
var L,i: integer;
begin
  L:=Length(Po); i:=-1; Result:=false;
  while (i<L-1) and not Result do begin
    Inc(i);
    Result:=Dist(Pq,Po[i])<Eps;
  end;
  if Result then n:=i
  else begin
    Inc(L); SetLength(Po,L); Po[L-1]:=Pq;
    n:=L-1;
  end;
end;
```

Функция `PointInTr` определяет, лежит ли точка `Pq` в треугольнике `P1`, `P2`, `P3`. Если сумма площадей трех треугольников с вершиной в точке `Pq` равна площади треугольника, то точка принадлежит треугольнику (листинг 12.31).

Листинг 12.31. Принадлежность точки треугольнику

```
function PointInTr(Pq,P1,P2,P3: TVertex): boolean;
const Eps=1e-6;
var S,S1,S2,S3: single;
begin
  S:=SurfTr(P1,P2,P3);
  S1:=SurfTr(Pq,P3,P2);
  S2:=SurfTr(P1,Pq,P3);
  S3:=SurfTr(P2,P1,Pq);
  Result:=(Abs(S)>Eps) and (Abs(S1+S2+S3-S)<Eps);
end;
```

12.2.5. Создание нового тела

Пересечение двух тел

На следующем этапе необходимо решить задачу разбиения ребер и граней при пересечении двух тел.

При разбиении ребер используется следующий алгоритм из трех шагов:

1. Процедурой `SetTempVertex(e)` для ребра `Edge[e]` формируется временный массив `TempVertex: array of word` номеров точек, принадлежащих ребру. В этой же процедуре массив `TempVertex` сортируется по координатам.
2. Если длина `Lt` массива больше `0`, то во временный массив ребер `Ed` добавляется `Lt-1` новых ребер.
3. Перебираются ребра треугольников триангуляции `Tr0`, и если они ранее не входили в массив `Ed`, то добавляются в него.

При разбиении граней формируется временный массив `Fa: array of Tface`, в который сначала копируются существующие грани, а затем добавляется по `Lt-1` треугольников триангуляции `Tr0`, если длина массива `Lt` треугольников больше нуля (листинг 12.32).

Листинг 12.32. Добавление ребер и граней

```
procedure AddEdgeFace;
var
  i,e,v,t,Le,Lee,Lb,Lt,f,Lf,Lf0,L,NumBody: integer;
  TempVertex: array of word;
  Ed: array of TEdge;
  Fa: array of TFace;
begin
  with ProjectBody do begin
    Le:=Length(Edge); Lee:=Le; SetLength(Ed,Lee);
```

```
for e:=0 to Le-1 do
  Ed[e]:=AssignEdge(Edge[e]);
for e:=0 to Le-1 do begin
  //найти все реберные точки TempVertex, принадлежащию ребру e
  SetTempVertex(e);
  if Lt>0 then begin // есть точки на ребре: разбить
    Lee:=Lee+Lt;
    SetLength(Ed,Lee);
    t:=Edge[e].p2;
    Ed[e].p2:=TempVertex[0];
    for v:=0 to Lt-1 do begin
      Ed[Lee-Lt+v].p1:=TempVertex[v];
      if v<Lt-1 then
        Ed[Lee-Lt+v].p2:=TempVertex[v+1]
      else
        Ed[Lee-Lt+v].p2:=t;
      SetEdgeIndexBody(Ed[Lee-Lt+v]);
    end;
  end;
end;

// ребра от треугольников
Lf:=Length(Face);
for f:=0 to Lf-1 do
with Face[f] do begin
  Lt:=Length(Tr0);
  if Lt>0 then
  for v:=0 to Lt-1 do
  with Tr0[v] do begin
    for i:=0 to 2 do
    if not EdgeInArr(p[i],p[(i+1) mod 3]) then begin
      Lee:=Lee+1; SetLength(Ed,Lee);
      Ed[Lee-1].p1:=p[i];
      Ed[Lee-1].p2:=p[(i+1) mod 3];
      SetEdgeIndexBody(Ed[Lee-1]);
    end;
  end;
end;

SetLength(Edge,Lee);
for e:=0 to Lee-1 do
  Edge[e]:=AssignEdge(Ed[e]);
SetLength(Ed,0);
SetLength(TempVertex,0);
```

```
    Lf0:=Length(Face); SetLength(Fa,Lf0);
    for f:=0 to Lf0-1 do
      Fa[f]:=AssignFace(Face[f]);

    Lf:=Lf0;
    for f:=0 to Lf0-1 do
    with Face[f] do begin
      Fa[f]:=AssignFace(Face[f]);
      Lt:=Length(Tr0);
      if Lt>0 then begin
        NumBody:=IndexBody[0];  // ???
        L:=Length(Body[NumBody].IndexFace);
        SetLength(Body[NumBody].IndexFace,L+Lt-1);
        for v:=1 to Lt-1 do
          Body[NumBody].IndexFace[L-1+v].Index:=Lf-1+v;

        Lf:=Lf+Lt-1; SetLength(Fa,Lf);
        SetLength(Fa[f].AIndexVertex,3);
        for v:=0 to 2 do
          Fa[f].AIndexVertex[v]:=Tr0[0].p[v];
        for v:=0 to 2 do
          Fa[f].IndexEdge[v]:=
            FindInEdge(Tr0[0].p[v],Tr0[0].p[(v+1) mod 3]);
        Fa[f].IndexBaseFace:=f;

        for v:=1 to Lt-1 do
        with Tr0[v] do begin
          Fa[Lf-Lt+v]:=AssignFace(Face[f]);
          SetLength(Fa[Lf-Lt+v].AIndexVertex,3);
          for i:=0 to 2 do
            Fa[Lf-Lt+v].AIndexVertex[i]:=Tr0[v].p[i];
          for i:=0 to 2 do
            Fa[Lf-Lt+v].IndexEdge[i]:=
              FindInEdge(Tr0[v].p[i],Tr0[v].p[(i+1) mod 3]);
          Fa[Lf-Lt+v].IndexBaseFace:=f;
        end;
      end;
    end;
    SetLength(Face,Lf);
    for f:=0 to Lf-1 do
      Face[f]:=AssignFace(Fa[f]);
    SetLength(Fa,0);
  end;
end;
```

Бинарные операции над телами

Для создания нового тела, полученного в результате одной из 16-ти бинарных операций, необходимо реализовать задачу сбора в результирующее тело граней и ребер по номеру бинарной операции над двумя телами. Алгоритм решения задачи сбора в результирующее тело граней и ребер при любой бинарной операции предложен в работах [100, 104], особых проблем не вызывает и для плоского случая обсуждался в *разд. 11.4*.

Новые вершины, грани и ребра добавляются в новое тело, если их индексы входят во множество сумм различных битов номера бинарной операции. За добавление номера нового тела в массивы `IndexBody` для граней и ребер отвечает обработчик события `onClick` для кнопки `BtnAdd`. При сборе ребер процедура `BtnAddBodyClick` перебирает все ребра, анализирует индексы вершин. Если оба индекса принадлежат множеству сумм различных битов номера бинарной операции, то ребро входит в новое тело. Возможна исключительная ситуация для вершин ребра с индексом 3, т. е. общих для двух тел вершин. В этом случае в качестве второй точки приходится использовать середину ребра. Аналогично решается задача сбора граней. Принадлежность грани определяется по ее трем вершинам или по двум вершинам и серединной точке. Реализация описанного алгоритма представлена в листинге 12.33. Сначала здесь просматривается массив ребер `Edge` и производится отбор подходящих ребер. Затем аналогичная операция выполняется для массива граней `Face`.

Листинг 12.33. Добавление ребер и граней в новое тело

```
procedure TFormTools.BtnAddBodyClick(Sender: TObject);
var
  Lb,k,Le,Lf,i,L: integer;
  Index1,Index2,Index3: byte;
  Q: TVertex;
  fl1,fl2: byte;
begin
  with ProjectBody do begin
    k:=Rg.ItemIndex;
    Lb:=Length(Body); Inc(Lb); SetLength(Body,Lb);
    Body[Lb-1].Name:='Body '+IntToStr(Lb);
    Body[Lb-1].Visible:=true;

    Le:=Length(Edge);  // сбор подходящих ребер
    for i:=0 to Le-1 do
    with Edge[i] do begin
      q_1:=Edge[i].p1; // номер 1-ой точки
      q_2:=Edge[i].p2; // номер 2-ой точки
      Index1:=Vertex[q_1].IndexOfVertex;
      Index2:=Vertex[q_2].IndexOfVertex;
```

```
  if (Index1<>3) or (Index2<>3) then
    Ok:=Op_In_Set(k, Index1, Index2)
  else begin       // Q - середина ребра
    Q.x:=(Vertex[p1].x+Vertex[p2].x)/2;
    Q.y:=(Vertex[p1].y+Vertex[p2].y)/2;
    Q.z:=(Vertex[p1].z+Vertex[p2].z)/2;
    GetIndexOfVertex(Q,Body2,0,fl2);
    GetIndexOfVertex(Q,Body1,1,fl1);
    case fl1+3*fl2 of
      1: Index2:=4;
      3: Index2:=5;
      4: Index2:=3;
      5: Index2:=1;
      7: Index2:=2;
    else
      Index2:=$FF;
    end;
    Ok:=Op_In_Set(k, Index1, Index2);
  end;
  if Ok then begin // добавляем номер ребра в IndexBody
    L:=Length(IndexBody);
    L:=L+1; SetLength(IndexBody,L);
    IndexBody[L-1]:=Lb-1;
  end;
end;

Le:=Length(Face);  // сбор подходящих граней
for i:=0 to Le-1 do
with Face[i] do begin
  q_1:=Face[i].AIndexVertex[0]; // номер 1-ой точки
  q_2:=Face[i].AIndexVertex[1]; // номер 2-ой точки
  q_3:=Face[i].AIndexVertex[2]; // номер 3-ой точки
  Index1:=Vertex[q_1].IndexOfVertex;
  Index2:=Vertex[q_2].IndexOfVertex;
  Index3:=Vertex[q_3].IndexOfVertex;
  if (Index1=3) and (Index2=3) and (Index3=3) then begin
    // Q - середина грани
    Q.x:=(Vertex[q_1].x+Vertex[q_2].x+Vertex[q_3].x)/3;
    Q.y:=(Vertex[q_1].y+Vertex[q_2].y+Vertex[q_3].y)/3;
    Q.z:=(Vertex[q_1].z+Vertex[q_2].z+Vertex[q_3].z)/3;
    GetIndexOfVertex(Q,Body2,0,fl2);
    GetIndexOfVertex(Q,Body1,1,fl1);
    case fl1+3*fl2 of
      1: Index3:=4;
      3: Index3:=5;
```

```
          4: Index3:=3;
          5: Index3:=1;
          7: Index3:=2;
        else
          Index3:=$FF;
        end;
        Ok:=Op_In_Set3(k, Index1, Index2, Index3);
      end
      else
        Ok:=Op_In_Set3(k, Index1, Index2, Index3);
      if Ok then begin
        Lf:=Length(IndexBody);
        Lf:=Lf+1; SetLength(IndexBody,Lf);
        IndexBody[Lf-1]:=Lb-1;
        L:=Length(Body[Lb-1].IndexFace);
        L:=L+1; SetLength(Body[Lb-1].IndexFace,L);
        Body[Lb-1].IndexFace[L-1].Index:=i;
      end;
    end;
  end;
  FormMain.DrawBody;
end;
```

Для определения индекса точки `VVertex` и признака ее положения относительно тела `n` используется функция `GetIndexOfVertex`. Эта функция возвращает значение `0`, если точка находится вне тела, значение `1`, если точка является вершиной тела или лежит на ребре или на грани, и значение `2`, если точка принадлежит телу. Определение принадлежности точки телу происходит по четности числа пересечений луча вдоль оси OX с гранями тела (листинг 12.34).

Листинг 12.34. Определение положения точки относительно тела

```
function GetIndexOfVertex(var VVertex: TVertex; n,fl: word): byte;
// определить IndexOfVertex для Vertex[Index_Vertex] в теле n
const
  Eps=1e-6;
var
  Lf,f,k,Lv,i: integer;
  Pq: TVertex;
  r: shortint;
begin
  VVertex.IndexOfVertex:=3;
  with ProjectBody do begin
    Lv:=Length(Body[n].IndexVertex); Ok:=false; i:=-1;
    while (i<Lv-1) and not Ok do begin
      Inc(i);
```

```
      Ok:=Dist(VVertex,Vertex[Body[n].IndexVertex[i]])<Eps;
    end;
    if Ok then                          // если вершина
      Result:=1
    else begin
      if PointInEdge(VVertex,n) then    // если ребро
        Result:=1
      else
      if PointInFace0(VVertex,n) then   // если грань
        Result:=1
      else begin
        // считаем k - число пересечений с гранями тела Body[n]
        Lf:=Length(Body[n].IndexFace);
        k:=0; Result:=2;
        for f:=0 to Lf-1 do begin
          r:=PointInFace(VVertex, Body[n].IndexFace[f].Index,Pq);
          if r=1 then k:=k+1;
        end;
        if k mod 2<>0 then begin
          VVertex.IndexOfVertex:=2-fl;  // внутри
          Result:=2;
        end
        else begin                      // вне тела
          VVertex.IndexOfVertex:=4+fl;
          Result:=0;
        end;
      end;
    end;
  end;
end; // GetIndexOfVertex
```

Функция `Op_In_Set` получает номер бинарной операции `k`, индексы двух вершин `n` и `m`, строит множество `V` сумм различных битов номера бинарной операции `k` и проверяет, входят ли `n` и `m` во множество `V` (листинг 12.35).

Листинг 12.35. Определение принадлежности ребра новому телу

```
function Op_In_Set(k,n,m: Byte): Boolean;
// отрезок n удовлетворяет операции k
// n:=Vertex[Edge[i].p1].IndexOfVertex;
// m:=Vertex[Edge[i].p2].IndexOfVertex;
var
  V: set of byte;
  i,j: byte;
begin
  V:=[3];
```

```
  for i:=0 to 2 do
  for j:=i+1 to 3 do
  if ((k and (1 shl i))<>0) xor
     ((k and (1 shl j))<>0) then V:=V+[i+j];
  Result:=(n in V) and (m in V);
end;  // Op_In_Set
```

Функция `Op_In_Set3` получает номер бинарной операции `k`, индексы трех вершин `n`, `m` и `q`, строит множество `V` сумм различных битов номера бинарной операции `k` и проверяет, входят ли `n`, `m` и `q` во множество `V` (листинг 12.36).

Листинг 12.36. Определение принадлежности грани новому телу

```
function Op_In_Set3(k,n,m,q: Byte): Boolean;
var
  V: set of byte;
  i,j: byte;
begin
  V:=[3];
  for i:=0 to 2 do
  for j:=i+1 to 3 do
  if ((k and (1 shl i))<>0) xor
     ((k and (1 shl j))<>0) then V:=V+[i+j];
  Result:=(n in V) and (m in V) and (q in V);
end;  // Op_In_Set3
```

Графическая трехмерная визуализация

Для реалистичного изображения тела на форме `FormOpenGL` используется библиотека OpenGL (рис. 12.22). Описание использования этой библиотеки приведено далее в *главе 13*.

Процедура `Show`, приведенная в листинге 12.37, демонстрирует процесс построения примитивов-треугольников OpenGL по граням видимых тел.

Листинг 12.37. Построение примитивов-треугольников OpenGL по граням

```
procedure TFormOpenGL.Show;
var
  i,j: integer;
  n: TVertex;
begin
  glViewPort((ClientWidth-ClientHeight) div 2-Zoom, 0-Zoom,
             ClientHeight+2*Zoom, ClientHeight+2*Zoom);
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
```

```
  with ProjectBody do begin
    Lb:=Length(Body);
    if Lb>0 then begin
      for i:=0 to Lb-1 do
      with Body[i] do
      if Visible then begin
        Ls:=Length(IndexFace);
        for j:=0 to Ls-1 do
        with Face[IndexFace[j].Index] do begin
          n:=getNormal(Vertex[AIndexVertex[0]],
                       Vertex[AIndexVertex[1]],
                       Vertex[AIndexVertex[2]]);
          glBegin(GL_TRIANGLES);
            glNormal3f(n.x,n.y,n.z);
            glVertex3f(Vertex[AIndexVertex[0]].x,
                       Vertex[AIndexVertex[0]].y,
                       Vertex[AIndexVertex[0]].z);
            glVertex3f(Vertex[AIndexVertex[1]].x,
                       Vertex[AIndexVertex[1]].y,
                       Vertex[AIndexVertex[1]].z);
            glVertex3f(Vertex[AIndexVertex[2]].x,
                       Vertex[AIndexVertex[2]].y,
                       Vertex[AIndexVertex[2]].z);
          glEnd;
        end;
      end;
    end;
  end;
  SwapBuffers(DC);
  glFlush();
end;
```

Для еще более реалистичной трехмерной визуализации тел можно в дальнейшем использовать следующие три варианта:

- стереоизображения, например, через разноцветные очки;
- лазерное изображение в трехмерном пространстве;
- использование двухслойных жидкокристаллических мониторов, выпуск которых начат в 2005 году.

В этой главе представлены упрощенные варианты проектов для работы с многогранниками. Оба проекта находятся на диске в папках **Примеры | Глава 12 | 3D** и **Примеры | Глава 12 | 3D Edition**.

Многие функции, реализация которых необходима и возможна, вынесены в задачи для самостоятельного решения. Список этих задач вы найдете в конце книги (*см. прил. 1*). Рекомендуем вам попытаться решить предложенные задачи.

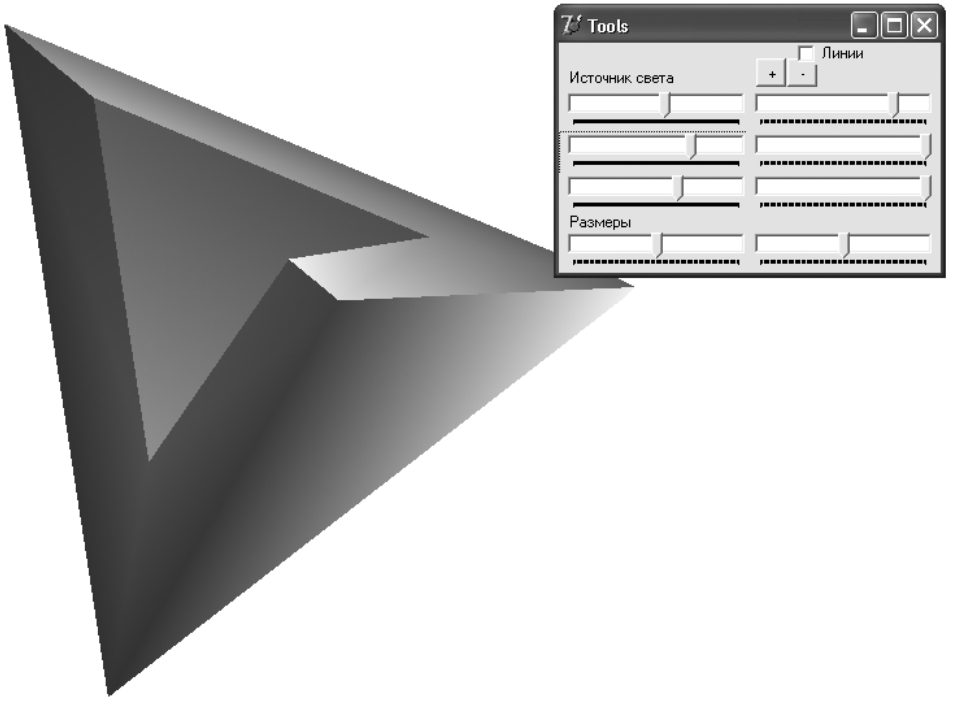


Рис. 12.22. Изображение тела с помощью библиотеки OpenGL

ГЛАВА 13

Использование графической библиотеки OpenGL

13.1. Введение

Графический стандарт и библиотека OpenGL (Open **G**raphics **L**ibrary — открытая графическая библиотека) разработаны в 1991–1993 гг. несколькими ведущими фирмами, в том числе Silicon Graphics Incorporated (SGI), IBM, Microsoft, Apple, Dell, Hewlett-Packard, Sun Microsystems и другими [24]. За основу была взята графическая библиотека IRIS GL, разработанная компанией Silicon Graphics. В состав библиотеки OpenGL для Windows входят библиотеки OpenGL32.dll и glu32.dll. Подобно другим графическим системам, OpenGL предлагает интерфейс между программным обеспечением и аппаратурой.

В основе модели интерпретации команд OpenGL лежит технология клиент-сервер: программа-клиент дает команды серверу OpenGL, который размещается на том же или на другом компьютере. На нижнем уровне команды OpenGL обрабатываются оконной системой, отвечающей за размещение ресурсов буфера кадра. Именно оконная система определяет, какая часть буфера может быть доступна для обработки программами, использующими OpenGL, и как осуществляется связь с этим буфером. Вывод содержимого буфера кадра на экран монитора не зависит от OpenGL.

OpenGL состоит из нескольких сотен процедур и функций, позволяющих описывать трехмерные графические объекты и операции над ними. Вместе с Delphi поставляется файл OpenGL.dcu и OpenGL.pas, в котором перечислены обращения этих процедур и функций к библиотекам OpenGL32.dll и glu32.dll[12].

Эти процедуры и функции позволяют: рисовать геометрические и растровые примитивы; работать с цветом в режиме RGBA и в индексном режиме; реализовать видовые преобразования и задавать модель освещения; удалять невидимые ребра и грани; использовать различные эффекты (сопряжения цветов, прозрачности, наложения текстуры, освещения распределенными и точечными источниками света, "тумана"); использовать B-сплайны.

Работа с цветом в режиме RGBA, наряду с установками долей красного (Red), зеленого (Green) и синего (Blue) цветов, предполагает использование еще одного параметра — альфа-параметра (Alpha), отвечающего за прозрачность (значение 1 этого параметра соответствует полной непрозрачности, значение 0 — полной прозрачности).

Под видовыми преобразованиями подразумеваются обычные геометрические преобразования системы координат, ее смещения и повороты.

В OpenGL предусмотрены следующие графические примитивы: точки, отрезки прямых линий, многоугольники, пиксельные прямоугольники или битовые карты. Каждый примитив задается множеством координат своих вершин, с которыми связываются следующие атрибуты: цвет, координаты вектора нормали и координаты текстуры. Все вершины обрабатываются независимо на основе текущих установок и режимов. В OpenGL предусмотрено множество операций над плоскими и трехмерными объектами: вращение, изменение размеров, перспективные преобразования, вариация освещенности и т. п. Так как стандарт отвечает только за визуализацию, осуществляемую через полностью независимый от производителя прикладной интерфейс пользователя, то в нем нет средств описания структур сложных объектов. Поэтому в библиотеку не включены средства построения невыпуклых многоугольников. Рекомендуется невыпуклые многоугольники предварительно разбить на выпуклые (например, треугольники).

Для задания геометрических объектов в OpenGL необходимо включить установки координат, цветов, векторов нормали и точек привязки текстур между командами `glBegin(DrawMode)` и `glEnd`. Для построения в трехмерной системе координат треугольника с вершинами в точках (`10`, `10`, `0`), (`10`, `20`, `0`) и (`20`, `0`, `10`), например, можно написать такую последовательность:

```
glBegin(GL_TRIANGLES);
  glVertex3i(10,10,0);
  glVertex3i(10,20,0);
  glVertex3i(20,0,10);
glEnd(); .
```

В OpenGL предусмотрены десять различных геометрических объектов:

- `GL_POINTS` — последовательность точек;
- `GL_LINE_STRIP` — ломаная линия;
- `GL_LINE_LOOP` — замкнутая ломаная линия;
- `GL_LINES` — отрезок линии;
- `GL_TRIANGLE_STRIP` — множество связных треугольников;
- `GL_TRIANGLE_FAN` — множество треугольников, связанных веером;
- `GL_TRIANGLES` — отдельный треугольник;
- `GL_QUAD_STRIP` — связные четырехугольники;

- `GL_QUADS` — отдельный четырехугольник;
- `GL_POLYGON` — выпуклый многоугольник.

На основе этих простых объектов можно строить сколь угодно сложные объекты.

Кроме координат, каждой вершине объекта могут присваиваться: вектор нормали, текущие координаты текстуры и цвет. Вектор нормали — трехмерный вектор, необходимый для вычисления ориентации, используемой, например, для формирования освещенности поверхности.

Исходные координаты вершины подвергаются видовому преобразованию однородной системы координат, умножением на матрицу 4×4, содержащую операции сдвига, поворота и масштабирования. После загрузки в стек одной или нескольких матриц преобразования можно управлять последовательностью их умножения на координаты вершины объекта с помощью специальных команд. В OpenGL предусмотрена возможность проводить геометрические преобразования не только путем последовательных поворотов вокруг осей X, Y или Z, но и относительно произвольной оси.

Цвет задается в модели RGBA, причем для задания доли каждого цвета можно использовать и целые числа в диапазоне от 0 до 255, и нормализованное вещественное число от 0,0 до 1,0. При расчете освещенности цвет объекта может изменяться в зависимости от следующих параметров: свойств материала, отражающих свойств поверхности объекта и параметров источника света. Вид изображения можно изменять, варьируя параметры модели освещенности, управляя расположением источников, их мощностью, свойствами среды и т. д.

Команды OpenGL, которые не используются для описания вершин и ассоциируемой с ними информации, могут быть размещены и вне блока `glBegin/glEnd`, что позволяет выполнять приложения в оптимизированном режиме, используя максимально возможный для данного оборудования режим вывода примитивов.

При работе с текстами в OpenGL предусмотрена возможность преобразования шрифтов из Windows непосредственно в свой буфер.

Обработка текстур является одним из наиболее мощных средств OpenGL, в число возможностей которых входят: управление временем выполнения преобразований для достижения требуемого качества изображения, автоматическое задание координат и трансформация текстур с помощью матрицы 4×4, привязка текстуры к поверхности произвольной формы. К текстуре, используя функцию `fog`, можно применить эффект "тумана". Эффект размытия или тумана достигается с помощью переменной, зависящей от расстояния до наблюдателя. Использование экспоненциальной зависимости изменения этой переменной позволяет моделировать утренний туман либо легкую дымку, а линейная зависимость дает эффект уменьшения освещения при удалении от источника света.

Базовой моделью интерпретации команд в OpenGL является режим непосредственного вывода, при котором команды выполняются сервером сразу после их получения. Например, визуализация вершины может осуществляться еще до того, как по-

лучены все атрибуты объекта, включающего данную вершину. Этот режим используется в интерактивных приложениях, требующих постоянных изменений примитивов и режимов их вывода. Благодаря непосредственному выводу достигается максимально высокая на данном графическом оборудовании производительность, а структура командного интерфейса OpenGL только способствует этому: даже отдельная операция по изменению параметров освещенности объекта (например, отражающих свойств материала поверхности) или вывод изображения текстуры выполняются отдельными командами.

Несмотря на свою гибкость, режим непосредственного вывода может оказаться неэффективным, если требуется переопределить объекты или параметры, неизменяемые интерактивно. В этом случае используется проверенный временем способ вывода через дисплейный список, содержащий последовательность команд OpenGL.

Поименованный буфер памяти с дисплейным списком хранится на сервере, а его содержимое выводится на экран при поступлении команды `glCallList`, содержащей приказ от программы клиента. Данный механизм удобен в случае необходимости многократного выполнения группы команд, заключенных между скобками `glBeginList/glEndList`.

Мы не ставим перед собой задачу описать все возможности библиотеки OpenGL. Более подробную информацию можно получить в документации OpenGL или, например, в прекрасной книге Ю. В. Тихомирова [105], в которой приведено множество примеров, разработанных в среде Visual C++.

13.2. Установка и завершение работы с OpenGL

Одним из наиболее сложных моментов при работе с библиотекой OpenGL является инициализация этой библиотеки. Во время инициализации необходимо [28]:

- с помощью функции `GetDC` получить указатель (дескриптор) `DC` на контекст воспроизведения (`rendering context`), то есть на поверхность рисования;
- задав поля специальной структуры `TPixelFormatDescriptor`, установить формат пикселов;
- функцией `wglCreateContext(DC)` создать новый контекст воспроизведения;
- с помощью процедуры `wglMakeCurrent` сделать данный контекст воспроизведения текущим;
- с помощью процедуры `glViewPort` определить область вывода на поверхности рисования;
- установить один из видов проекции;
- подготовить цвета для очистки буфера цвета.

13.2.1. Получение дескриптора контекста воспроизведения

Для получения дескриптора `DC` контекста воспроизведения (компонента вывода графической информации) используется функция:

```
function GetDC(hWnd: HWND): HDC; .
```

Единственный параметр `hWnd: HWND` этой функции должен быть указателем на окно.

13.2.2. Установка формата пикселов

В библиотеке OpenGL для описания каждого пиксела отводится несколько битов. Для описания долей RGB-цветов обычно отводится 16 битов, некоторое количество битов отводится для описания прозрачности (альфа-канала — `cAlphaBits`). Как правило, 32 бита отводится для учета глубины (`cDepthBits`), еще 8 битов отводится для наложения трафарета и еще 32 бита на каждый пиксел отводится для аккумулятора (`aAccumBits`).

Для задания конкретных значений этих параметров в модуле OpenGL предусмотрена структура `TPixelFormatDescriptor`, поля которой описывают формат пикселов (листинг 13.1). Описание полей этой структуры приведено в табл. 13.1.

Листинг 13.1. Структура *TPixelFormatDescriptor*

```
TPixelFormatDescriptor = packed record
  nSize: Word;
  nVersion: Word;
  dwFlags: DWORD;
  iPixelType: Byte;
  cColorBits: Byte;
  cRedBits: Byte;
  cRedShift: Byte;
  cGreenBits: Byte;
  cGreenShift: Byte;
  cBlueBits: Byte;
  cBlueShift: Byte;
  cAlphaBits: Byte;
  cAlphaShift: Byte;
  cAccumBits: Byte;
  cAccumRedBits: Byte;
  cAccumGreenBits: Byte;
  cAccumBlueBits: Byte;
  cAccumAlphaBits: Byte;
  cDepthBits: Byte;
  cStencilBits: Byte;
```

```
    cAuxBuffers: Byte;
    iLayerType: Byte;
    bReserved: Byte;
    dwLayerMask: DWORD;
    dwVisibleMask: DWORD;
    dwDamageMask: DWORD;
  end;
```

Таблица 13.1. *Поля структуры формата пикселов*

Название поля	Комментарий
nSize:	Размер структуры
nVersion:	Номер версии. Должен быть равным 1
dwFlags	Множество битовых флагов, определяющих устройство и интерфейс, с которыми совместим формат пикселов. С помощью операции or можно использовать следующие значения: PFD_DRAW_TO_WINDOW — разрешено рисование в окне или на поверхности другого компонента; PFD_DRAW_TO_BITMAP — разрешено рисование на битовой матрице PFD_SUPPORT_GDI — рисование поддерживается средствами интерфейса графического устройства (GDI); PFD_SUPPORT_OPENGL — поддерживается рисование OPENGL; PFD_GENERIC_FORMAT — формат пиксела определяется интерфейсом графического устройства (GDI)
iPixelType	Определяет режим изображения цветов: PFD_TYPE_RGBA — цвет пиксела определяется стандартом RGB и параметром альфа; PFD_TYPE_COLORINDEX — цвет пиксела определяется индексом в специальной таблице
cColorBits	Определяет число битовых плоскостей в каждом буфере цвета. Для режима RGBA определяет размер буфера цвета, исключая битовую плоскость альфа. Под битовой плоскостью подразумевается прямоугольный массив битов: одному пикселу соответствует один элемент битового массива
cRedBits	Определяет число битовых плоскостей красного цвета в каждом буфере RGBA
cRedShift	Определяет смещение от начала числа битовых плоскостей красного цвета в каждом буфере RGBA
cGreenBits	Определяет число битовых плоскостей зеленого цвета в каждом буфере RGBA
cGreenShift	Определяет смещение от начала числа битовых плоскостей зеленого цвета в каждом буфере RGBA
cBlueBits	Определяет число битовых плоскостей синего цвета в каждом буфере RGBA

Таблица 13.1 *(окончание)*

Название поля	Комментарий
cBlueShift	Определяет смещение от начала числа битовых плоскостей синего цвета в каждом буфере RGBA
cAlphaBits	Определяет число битовых плоскостей альфа в каждом буфере RGBA
cAlphaShift	Определяет смещение от начала числа битовых плоскостей альфа в каждом буфере RGBA
cAccumBits	Определяет общее число битовых плоскостей в буфере аккумулятора
cAccumRedBits	Число битовых плоскостей красного цвета в буфере аккумулятора
cAccumGreenBits	Число битовых плоскостей зеленого цвета в буфере аккумулятора
cAccumBlueBits	Число битовых плоскостей синего цвета в буфере аккумулятора
cAccumAlphaBits	Число битовых плоскостей альфа в буфере аккумулятора
cDepthBits	Определяет размер буфера глубины (ось Z)
cStencilBits	Размер буферов трафарета
cAuxBuffers	Число вспомогательных буферов
iLayerType	Тип слоя. Может принимать одно из трех значений: PFD_MAIN_PLANE = 0; PFD_OVERLAY_PLANE = 1; PFD_UNDERLAY_PLANE = LongWord(-1)
bReserved	Число плоскостей переднего и заднего плана
dwLayerMask	Игнорируется
dwVisibleMask	Определяет индекс или цвет прозрачности нижней плоскости
dwDamageMask	Игнорируется

Для выбора наиболее подходящего формата пикселов и его установки предназначена процедура `SetupPixelFormat` (листинг 13.2).

Листинг 13.2. Установка формата пикселов

```
procedure SetupPixelFormat;
var PixelFormat: integer;
    PFD:TPixelFormatDescriptor;
begin
  FillChar(PFD,SizeOf(TPixelFormatDescriptor),0);
  with PFD do
  begin
   nSize:=SizeOf(TPixelFormatDescriptor); // размер структуры
   nVersion:=1;            // номер версии. Должно быть 1
```

```
 dwFlags:=
   PFD_DRAW_TO_WINDOW or // разрешено рисование в окне
   PFD_SUPPORT_OPENGL or // поддерживается рисование OPENGL
   PFD_SUPPORT_GDI;      // поддерживается рисование GDI

 iPixelType:=PFD_TYPE_RGBA;
 // определяет режим изображения цветов
 // PFD_TYPE_RGBA - цвет пиксела определяется RGB и альфа
 // PFD_TYPE_COLORINDEX - цвет пиксела определяется
 // индексом в специальной таблице

 cColorBits:=16;
 // определяет число битовых плоскостей в каждом буфере
 // цвета. Для режима RGBA определяет размер буфера цвета,
 // исключая битовую плоскость альфа.

 cDepthBits:=32; //определяет размер буфера глубины (ось Z)
 iLayerType:=PFD_MAIN_PLANE;// PFD_MAIN_PLANE = 0;
                            // PFD_OVERLAY_PLANE = 1;
                            // PFD_UNDERLAY_PLANE
end;

PixelFormat:=ChoosePixelFormat(DC,@PFD);
// Функция ChoosePixelFormat запрашивает систему -
// поддерживается ли установленный
// формат пикселов и возвращает наиболее подходящий

if PixelFormat=0 then // ошибка
 Raise
  Exception.Create('Формат пикселов не поддерживается!');

SetPixelFormat(DC,PixelFormat,@PFD);
// устанавливает формат пикселов в контексте устройства

DescribePixelFormat(DC,PixelFormat,
  SizeOf(TPixelFormatDescriptor),PFD);
// получаем информацию о параметрах формата пикселов

with PFD do
if ((dwFlags and PFD_DRAW_TO_WINDOW) = 0) or
   ((dwFlags and PFD_SUPPORT_OPENGL) = 0) or
   ((dwFlags and PFD_SUPPORT_GDI) = 0) or
   (iPixelType <> PFD_TYPE_RGBA) or
```

```
     (cColorBits < 16) then
    Raise
     Exception.Create('Выбран неподходящий формат пикселов');
end;
```

13.2.3. Инициализация библиотеки OpenGL

Получение дескриптора DC контекста воспроизведения и установку формата пикселов можно реализовать, например, при создании следующей формы (листинг 13.3).

Листинг 13.3. Инициализация OpenGL

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  DC:=GetDC(Handle);
  SetupPixelFormat(DC{,Palette});
  // устанавливает формат пикселов
  RC:=wglCreateContext(DC);
  // создает новый контекст воспроизведения
  wglMakeCurrent(DC,RC);
  // устанавливает текущий контекст воспроизведения
  glViewPort(0,0,ClientWidth,ClientHeight);
  // определяет область вывода

  glOrtho(0,ClientWidth,0,ClientHeight,0,1);
  // glOrtho (устанавливает параллельную проекцию
  //    left,     - координаты левой плоскости отсечения
  //    right,    - правой
  //    bottom,   - нижней
  //    top,      - верхней
  //    zNear,    - расстояние до ближней плоскости отсечения
  //    zFar: GLdouble) - до дальней
  ConvertWinColor(Color,r,g,b);
  glClearColor(r,g,b,1);
  // определяет RGBA-цвета, используемые при очистке
  // буферов цвета
  glClear(GL_COLOR_BUFFER_BIT);
  // установка RGBA-цвета
  // GL_COLOR_BUFFER_BIT - буферы, доступные для записи цвета
  // GL_DEPTH_BUFFER_BIT - буфер глубины
  // GL_ACCUM_BUFFER_BIT - буфер аккумулятора
  // GL_STENCIL_BUFFER_BIT - буфер трафарета
  //
```

```
  glFlush;
  glFinish;
  // гарантирует завершение команд за определенное или
  // за конечное время
end;
```

В этом же обработчике события можно выполнить остальные подготовительные действия:

- функция `wglCreateContext(DC: HDC): HGLRC` создает новый контекст воспроизведения. Контекст воспроизведения имеет такой же формат пикселов, как и контекст устройства. Параметр `DC: HDC` — указатель на контекст устройства;
- функция `wglMakeCurrent(DC: HDC; RC: HGLRC): boolean` устанавливает текущий контекст воспроизведения. Параметр `DC: HDC` — указатель на контекст устройства, а `RC` — указатель на контекст воспроизведения;
- процедура `glViewport(x: Glint; y: Glint; width: GLsizei; height: GLsizei)` определяет область вывода. Параметры `x, y` определяют левый верхний угол прямоугольника окна экрана в пикселах. По умолчанию они принимают значения `(0, 0)`. Параметры `width` и `height` определяют ширину и высоту окна экрана. При подключении контекста GL к окну, ширина и высота устанавливаются по размерам этого окна. Ошибка `GL_INVALID_VALUE` генерируется, если ширина или высота имеют отрицательные значения. Ошибка `GL_INVALID_OPERATION` генерируется, если `glViewport` вызывается между командой `glBegin` и соответствующей командой `glEnd`;
- процедура `glOrtho(left, right, bottom, top, near, far: GLdouble)` умножает текущую матрицу на ортографическую матрицу. Параметры `left, right` определяют координаты левой и правой вертикальных плоскостей отсечения. Параметры `bottom, top` определяют координаты нижней и верхней горизонтальных плоскостей отсечения. Параметры `near, far` определяют расстояния до ближней и дальней плоскостей отсечения. Эти расстояния отрицательны, если плоскости находятся за наблюдателем. Текущая матрица `M` умножается на ортографическую матрицу `O: M=M·O`. Для сохранения и восстановления текущей матрицы можно использовать процедуры `glPushMatrix` и `glPopMatrix`;
- процедура `ConvertWinColor(AColor: TColor; var red, green, blue: GLFloat)` конвертирует цвет контекста устройства (окна) в доли красного, зеленого и синего цветов. Эти доли будут использоваться в следующей процедуре установки цветов очистки буферов цвета;
- процедура `glClear(mask: GLbitfield)` очищает буферы в пределах окна экрана. Параметр `mask` образуется с помощью битовой операции `or` из масок, указывающих на буферы, которые должны очищаться. Допустимы четыре маски: `GL_COLOR_BUFFER_BIT` — указывает на буферы, в настоящее временя доступные для записи цвета, `GL_DEPTH_BUFFER_BIT` — указывает буфер глубины, `GL_ACCUM_BUFFER_BIT` — указывает на буфер накопления, `GL_STENCIL_BUFFER_BIT` — указывает на буфер шаблона. Ошибка `GL_INVALID_VALUE` генерируется, если в

маске устанавливается любой бит кроме четырех определенных битов. Ошибка `GL_INVALID_OPERATION` генерируется, если `glClear` вызывается между вызовом `glBegin` и соответствующим вызовом `glEnd`;

- команды `glFlush` и `glFinish` гарантируют завершение команд за определенное или за конечное время. Команда `glFinish` блокирует дальнейшее выполнение программы, пока все предыдущие команды OpenGL не будут завершены. Команда `glFlush` предписывает выполнять команды OpenGL за конечное время. Эта функция может вызываться в любое время. Она не ожидает завершения всех ранее запущенных команд.

13.2.4. Завершение работы с OpenGL

Завершение работы с OpenGL можно реализовать, например, при закрытии следующей формы (листинг 13.4).

Листинг 13.4. Завершение работы с OpenGL

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  wglMakeCurrent(0,0);
  wglDeleteContext(RC);
  ReleaseDC(Handle,RC);
  if Palette<>0 then DeleteObject(Palette);
end;
```

При завершении работы необходимо:

- освободить контекст воспроизведения `wglMakeCurrent(0,0);`
- удалить контекст воспроизведения `wglDeleteContext(RC);`
- освободить контекст устройства, связанный с контекстом воспроизведения `ReleaseDC(Handle,RC);`
- если использовалась палитра, то удалить ее.

13.3. Команды и примитивы OpenGL

13.3.1. Синтаксис команд

Многие команды в OpenGL используются в различных вариантах. В названии команд отражается число аргументов и их тип. Символом `v` указывается, что в качестве аргумента используется указатель на массив. В общем случае синтаксис таких команд можно представить следующей обобщенной формулой [67]:

```
NameCommand [1|2|3|4] [b|s|i|f|d|ub|us|ui] [v]
```

Рассмотрим строение команды:

- одна из цифр `1|2|3|4` указывает на число аргументов;
- символы `b|s|i|f|d|ub|us|ui` указывают на тип аргументов. Типы аргументов, допустимые в OpenGL, приведены в табл. 13.2;
- символ `v` указывает на то, что в качестве аргумента используется указатель на массив.

Таблица 13.2. *Соответствие символов и типов OpenGL и Pascal*

Символы	Тип OpenGL	Тип Pascal
b	GLbyte	Shortint
s	GLshort	SmallInt
u	GLint	Integer
f	GLfloat	Single
d	GLdouble	Double
ub	GLubyte	Byte
us	GLushort	Word
ui	GLuint	Cardinal

Для полноты картины приведем также таблицу соответствия типов OpenGL и Pascal (табл. 13.3). Эти типы не участвуют в именах команд OpenGL.

Таблица 13.3. *Соответствие типов OpenGL и Pascal*

Тип OpenGL	Тип Pascal	Тип OpenGL	Тип Pascal
GLenum	Cardinal	GLsizei	Integer
GLboolean	BYTEBOOL	GLclampf	Single
GLbitfield	Cardinal	GLclampd	Double

В качестве примера приведем команду, определяющую координаты одной точки на плоскости: `glVertex2f(300,300)`.

13.3.2. Вершины

Вершина — это точка в двумерном или в трехмерном пространстве, координаты которой задаются командами:

```
glVertex[2 3 4][s i f d](coord: GLtype);
glVertex[2 3 4][s i f d]v(coord: PGLtype);
```

Параметр `coord` задает четыре однородные координаты вершины (`x, y, z, w`). Использование команды `glVertex2` задает координаты (`x, y, 0, 1`), использование

команды `glVertex3` задает координаты (`x, y, z, 1`), а команда `glVertex4` задает все четыре однородные координаты (`x, y, z, w`).

13.3.3. Примитивы

В графической библиотеке OpenGL возможно использование следующих примитивов: точек, линий, многоугольников, прямоугольников, пикселов. Каждый примитив или группа однородных примитивов задается вершинами и связанными с этой группой свойствами. Группа определяется командными скобками `glBegin .. glEnd`:

```
glBegin(mode: GLenum);
  …
glEnd;
```

Параметр `mode` определяет тип примитивов. Возможные значения для этого параметра перечислены в табл. 13.4.

Таблица 13.4. *Значения параметра mode, определяющего тип примитива*

Значение	Число	Результат
`GL_POINTS`	1	Каждая вершина рассматривается как независимая точка
`GL_LINES`	2	Каждая пара вершин представляет собой независимый отрезок. Всего будет нарисовано `N div 2` отрезков
`GL_LINE_LOOP`	2	Рисуется замкнутая ломаная линия (многоугольник с `N` вершинами)
`GL_LINE_STRIP`	2	Задается ломаная линия. Если в командных скобках `glBegin/glEnd` заключено `N` вершин, то будет нарисовано `N-1` отрезков
`GL_TRIANGLES`	3	Каждые три вершины задают независимый треугольник. Всего будет нарисовано `N div 3` треугольников
`GL_TRIANGLE_STRIP`	3	Рисуется группа связанных треугольников. Треугольник с номером `N` опирается на вершины с номерами `N+2`, `N+1` и `N`
`GL_TRIANGLE_FAN`	3	Рисуется группа связанных треугольников с общей вершиной. Треугольник с номером `N` опирается на вершины с номерами `N+2`, `N+1` и `1`
`GL_QUADS`	4	Каждые четыре вершины задают независимый четырехугольник
`GL_QUAD_STRIP`	4	Рисуется группа связанных четырехугольников. Четырехугольник с номером `N` опирается на вершины с номерами `2*N-1`, `2*N`, `2*N+2` и `2*N+1`. Всего изображается `(N-2) div 2` четырехугольника
`GL_POLYGON`	3	Рисуется выпуклый многоугольник

С группой примитивов связываются следующие свойства.

- Цвет. Если нет источников света, то команда `glColor*` для режима RGBA или `glIndex*` для индексного режима определяют цвет примитивов. При активизированных источниках света цвет примитивов определяется взаимодействием цвета источников и цвета примитивов.
- Вектор нормали. Для каждой вершины команда `glNormal*` определяет вектор нормали к плоскости, которой принадлежит эта вершина.
- Материал. Команда `glMaterial` определяет свойства материала (рассеянный цвет, цвет диффузионного отражения, цвет зеркального отражения, степень зеркального отражения).
- Позиция растра. Команда `glRasterPos*` устанавливает координаты растра при использовании битовых массивов.
- Координаты текстуры. Координаты текстуры являются частью данных, которые связываются с вершинами многоугольника. Они устанавливаются командой `glTexCoord*`. Команда `glTexCoord` позволяет определить координаты текстуры от одной до четырех размерностей и может вызываться между командами `glBegin` и `glEnd`.

13.4. Плоская графика

На прилагаемом к книге компакт-диске в папке **Примеры | Глава 13 | 2d Open GL** представлен проект "Плоская графика в OpenGL", использующий плоские примитивы OpenGL. В этом проекте демонстрируется возможность построения плоских примитивов, координаты которых выбираются случайным образом. Выбор примитива осуществляется с помощью соответствующих кнопок на инструментальной форме `FormTools`, а изображение строится на главной форме `FormMain`.

В листинге 13.5 приведен фрагмент кода этого проекта, выполняющего прорисовку точек случайного цвета со случайными координатами (рис. 13.1). Перед назначением цвета точек командой `glColor*` необходимо отключить источники света командой `glDisable`.

Листинг 13.5. Рисование точек

```
glDisable(GL_LIGHTING);  // отключить источники света
// включить режим сглаживания
glEnable(GL_POINT_SMOOTH);
//glDisable(GL_POINT_SMOOTH);
for i:=1 to 10 do begin
  glPointSize(i);        // задать размер точек
  glBegin(GL_POINTS);    // примитив - точки
    glColor3f(Random,Random,Random);
```

```
    glVertex2f(Random(700),Random(600));
  glEND;
end;
glFinish;
```

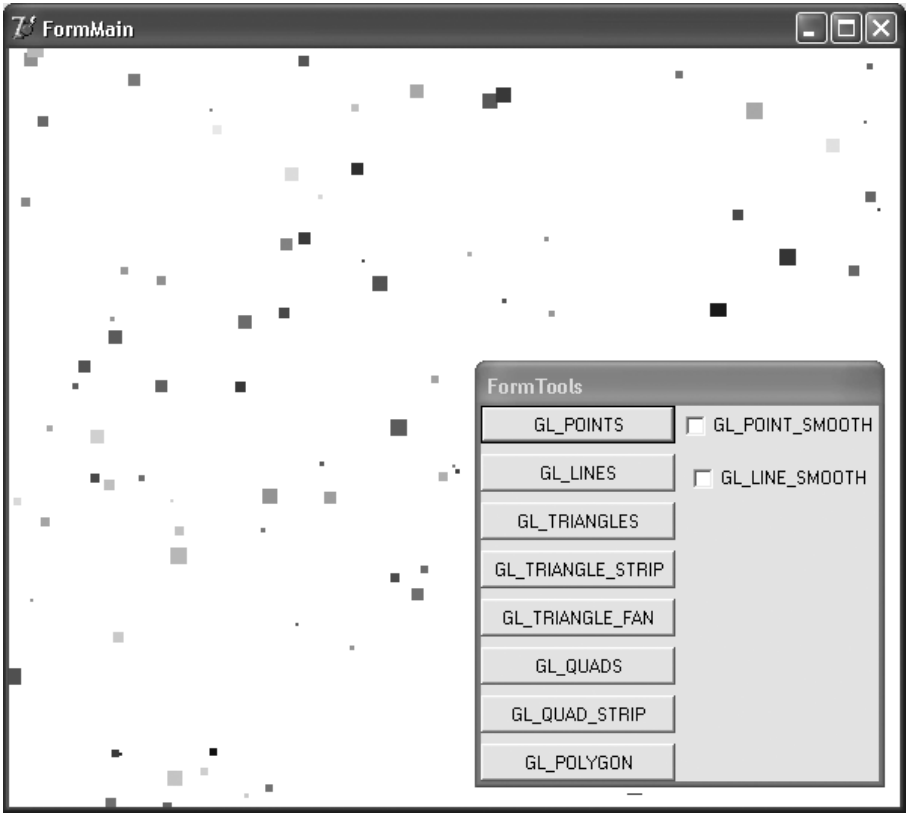


Рис. 13.1. Рисование точек средствами OpenGL

Точки могут изображаться прямоугольниками, если отключен режим сглаживания командой `glDisable` с параметром `GL_POINT_SMOOTH`, или окружностями, если этот режим включен. Команда `glPointSize` задает размер точек. В режиме сглаживания размер точек-прямоугольников не может быть любым и, обычно, ограничен величиной `10`.

В листинге 13.6 приведен фрагмент кода, рисующего отрезки со случайными координатами и случайными цветами для каждой вершины отрезка (рис. 13.2).

Листинг 13.6. Рисование отрезков

```
glDisable(GL_LIGHTING);
glColor3f(1,0,0);
```

```
// включаем режим устранения ступенчатости
glEnable(GL_LINE_SMOOTH);
//glDisable(GL_LINE_SMOOTH);
// задаем толщину линий
glLineWidth(4);
glBegin(GL_LINES);
  for i:=1 to Random(20) do begin
    // задаем случайный цвет каждой вершины
    glColor3f(Random,Random,Random);
    // задаем координаты вершин
    glVertex2f(Random(800),Random(600));
  end;
glEND;
glFinish;
```

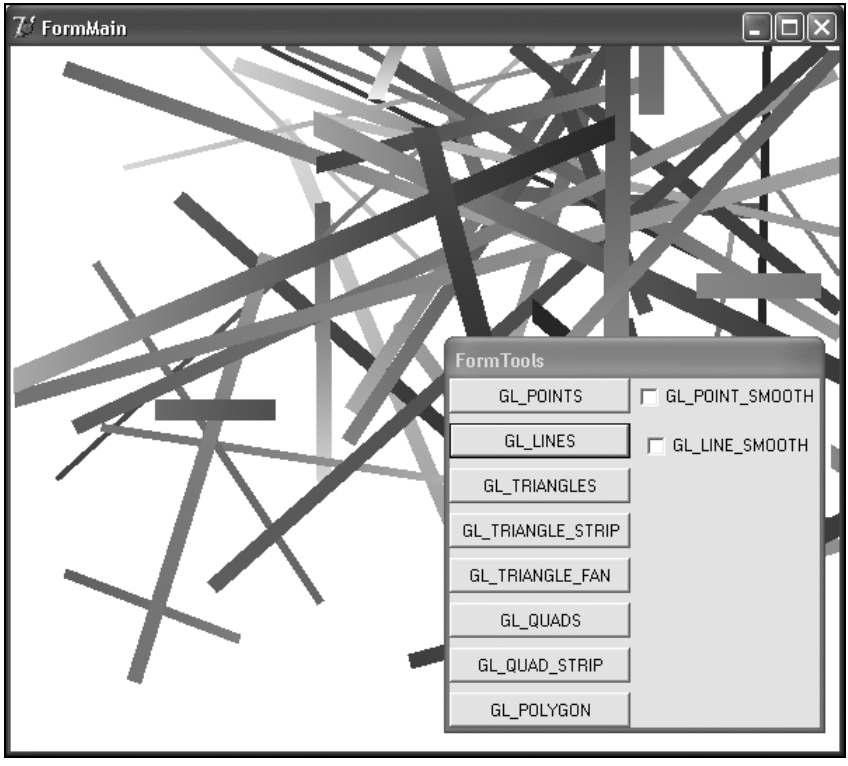


Рис. 13.2. Рисование отрезков средствами OpenGL

Отрезки изображаются в виде параллелограммов, меньшие стороны которых горизонтальны. При включенном режиме сглаживания (устранения ступенчатости) `glEnable(GL_LINE_SMOOTH)` линии рисуются прямоугольниками. Этот режим требует дополнительных временных затрат. Каждой вершине приписывается свой цвет,

поэтому цвета точек отрезков плавно меняются от цвета одного конца отрезка до цвета, заданного на другом конце отрезка.

В следующем примере рисования треугольников (листинг 13.7) плавное изменение цветов, заданных для каждой вершины треугольника, дает интересный визуальный эффект (рис. 13.3).

Листинг 13.7. Рисование треугольников

```
glDisable(GL_LIGHTING);
glBegin(GL_TRIANGLES);
  for i:=1 to Random(20) do
  begin
    glColor3f(Random,Random,Random);
    glVertex2f(Random(800),Random(600));
  end;
glEND;
glFinish;
```

Рисование остальных плоских фигур также несложно, поэтому не рассматривается.

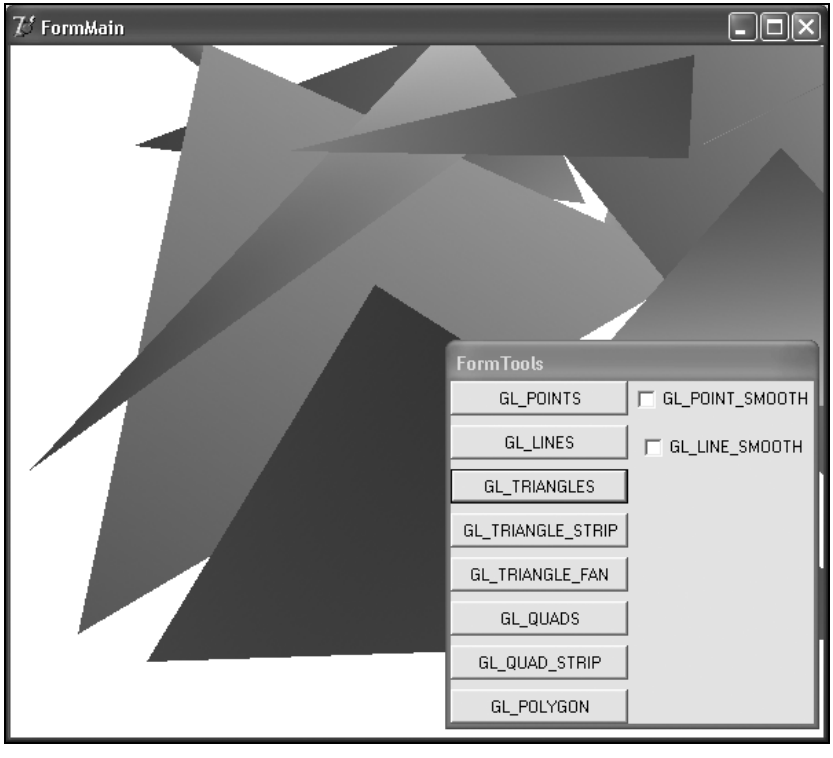


Рис. 13.3. Рисование треугольников средствами OpenGL

13.5. Трехмерная графика

Для демонстрации работы с трехмерными объектами предназначен проект "Трехмерная графика в OpenGL", полный текст которого представлен на компакт-диске в папке **Примеры | Глава 13 | 3d Open GL**.

Проект позволяет рисовать многогранники модуля DGLUT, квадратичные поверхности, заданные параметрическими уравнениями, поверхность, заданную табличным способом, импортировать различные фигуры, созданные в других графических пакетах, накладывать на поверхности текстуры из файлов BMP. Построение изображений производится на главной форме проекта `FormMain`. Форма инструментов `FormTools`, содержащая несколько вкладок, содержит наборы кнопок, радиокнопок и флажков для задания вида объекта изображения и его параметров.

Внешний вид форм проекта представлен на рис. 13.4.

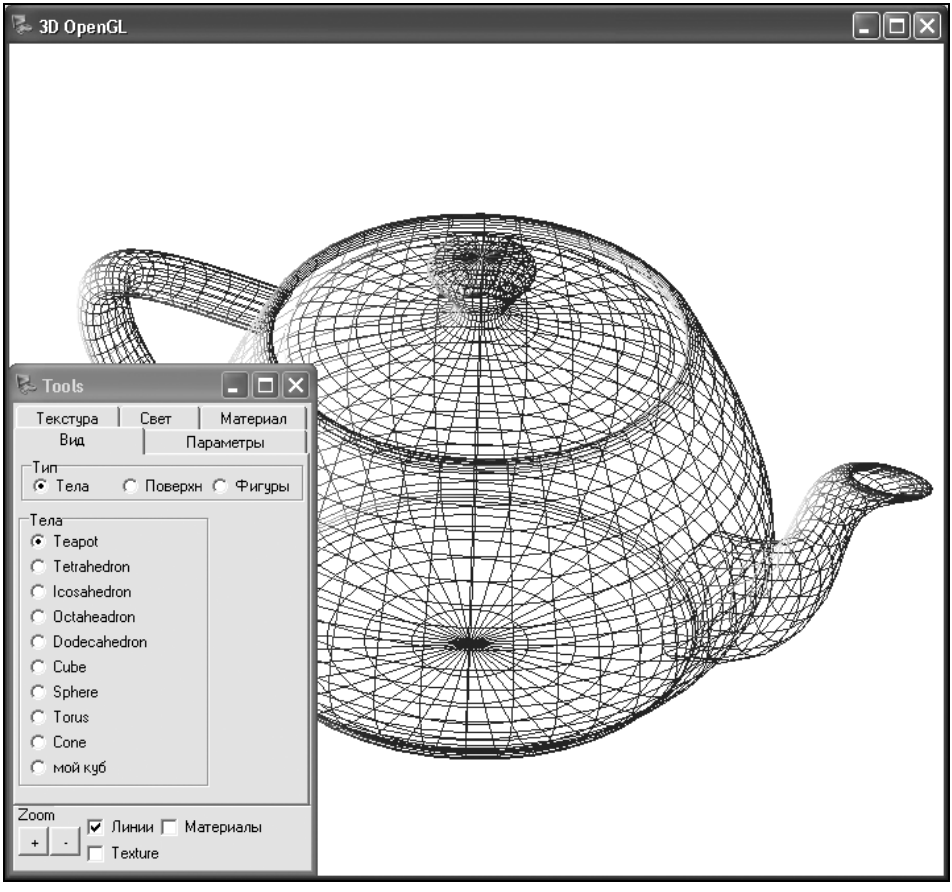


Рис. 13.4. Проект "Трехмерная графика в OpenGL"

13.5.1. Инициализация OpenGL

Прежде чем вызывать процедуры для рисования, необходимо в момент запуска проекта инициализировать библиотеку OpenGL (листинг 13.8).

Листинг 13.8. Инициализация библиотеки OpenGL

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  //получение дискриптора на контекст воспроизведения
  DC := GetDC (Handle);
  //создаем новый контекст воспроизведения
  SetDCPixelFormat(DC);
  //устанавливаем текущий контекст воспроизведения
  hrc := wglCreateContext(DC);
  wglMakeCurrent(DC,hrc);

  glClearColor ( 1.0 , 1.0 , 1.0 , 1.0 );

  // разрешается проведение теста на сравнение
  // параметров глубины объектов
  // и производится обновление буфера глубины
  glEnable ( GL_DEPTH_TEST );

  glEnable ( GL_COLOR_MATERIAL );

  glLightfv(GL_LIGHT0, GL_POSITION, @ligth_position0);
  glLightfv(GL_LIGHT0, GL_DIFFUSE, @ambient);

  glEnable(GL_LIGHTING);
  glEnable(GL_NORMALIZE);

  glEnable(GL_LIGHT0);
  MyDrawBoxList(0.5);
end;
```

Перед рисованием необходимо настроить параметры OpenGL, т. е. установить свойства материала и командой `glLightfv` расставить источники света. Подробный разговор об этих командах пойдет в следующих разделах. А пока приведем самый простой вариант процедуры рисования (листинг 13.9).

Листинг 13.9. Настройка параметров OpenGL перед рисованием

```
procedure TForm1.Show;
begin
  glViewPort(((ClientWidth div 2)-(ClientHeight div 2))-Zoom,
```

```
    0-Zoom,ClientHeight+(2*Zoom), ClientHeight+(2*Zoom));
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);

  // рисование
  // glutWireCube(Size);

  SwapBuffers(DC);
  glFlush();
end;
```

13.5.2. Многогранники модуля DGLUT

Небольшой набор многогранников представляет сама библиотека OpenGL[31]. Она, в частности, позволяет рисовать сферы, цилиндры и диски. Для рисования этих фигур предназначены следующие команды:

```
procedure gluCylinder;
procedure gluDisk;
procedure gluPartialDisk;
procedure gluSphere;
```

В 1997 г. Боб Кроуфорд (Bob Crawford), ссылаясь на Марка Килгарда (Mark J. Kilgard) и компанию Silicon Graphics, предложил модуль DGLUT.PAS, в котором представлен достаточно широкий набор многогранников: куб, конус, тор, октаэдр, икосаэдр, додекаэдр, тетраэдр и достаточно сложный многогранник — чайник, описываемый во многих книгах по компьютерной графике. Для построения изображений этих тел предназначены следующие процедуры.

Для построения куба:

```
procedure DrawBox;
procedure glutWireCube;
procedure glutSolidCube.
```

Для построения сферы:

```
procedure glutWireSphere;
procedure glutSolidSphere.
```

Для построения конуса:

```
procedure glutWireCone;
procedure glutSolidCone.
```

Для построения тора:

```
procedure glutWireTorus;
procedure glutSolidTorus.
```

Для построения додекаэдра:

```
procedure glutWireDodecahedron;
procedure glutSolidDodecahedron.
```

Для построения октаэдра:

```
procedure Octaheadron;
procedure glutWireOctaheadron;
procedure glutSolidOctaheadron.
```

Для построения икосаэдра:

```
procedure Icosahedron;
procedure glutWireIcosahedron;
procedure glutSolidIcosahedron.
```

Для построения тетраэдра:

```
procedure Tetrahedron;
procedure glutWireTetrahedron;
procedure glutSolidTetrahedron.
```

Для построения изображения чайника:

```
procedure Teapot;
procedure glutWireTeapot;
procedure glutSolidTeapot.
```

Для каждого вида многогранников используются процедуры двух типов: `Wire` и `Solid`, реализующие, соответственно, их каркасное и сплошное представление. Если параметр режима рисования принимает значение `GL_LINE_LOOP`, то реализуется рисование ребрами. Если же параметр режима рисования принимает значение `GL_TRIANGLES` или `GL_QUADS`, то реализуется рисование гранями — треугольниками или четырехугольниками.

Наиболее сложной из перечисленных фигур является чайник, изображение которого представлено на рис. 13.5.

Если использовать готовые многогранники модуля DGLUT, то процедура рисования куба будет выглядеть так (листинг 13.10).

Листинг 13.10. Процедура рисования куба

```
procedure TForm1.Show;
begin
  glViewPort(((ClientWidth div 2)-(ClientHeight div 2))-Zoom,
    0-Zoom,ClientHeight+(2*Zoom), ClientHeight+(2*Zoom));
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
  glutWireCube(Size);
  SwapBuffers(DC);
  glFlush();
end;
```

Задачу построения многогранников рассмотрим на примере построения тетраэдра (рис. 13.7).

Рис. 13.5. Изображение "чайник"

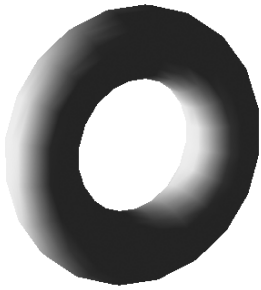


Рис. 13.6. Изображение "тор"

На рис. 13.6 представлено изображение тора, построенное средствами модуля DGLUT.PAS.

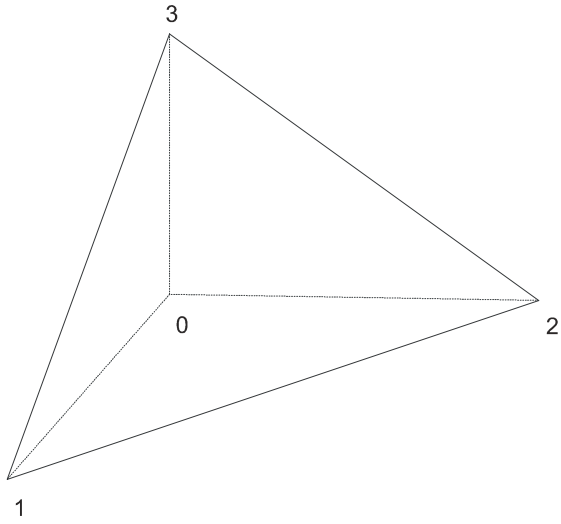


Рис. 13.7. Построение тетраэдра

Для определения параметров тетраэдра потребуются определить вершины, грани и нормали к граням.

Координаты четырех вершин заданы массивом `V0`.

```
V0: array[0..3,0..2] of GLfloat=((1,0,0),(0,0,0),(0,1,0),(0,0,1));
```

Для задания номеров точек, на которые опираются четыре треугольные грани тетраэдра, в модуле DGLUT введен двумерный массив граней `TetFaces`.

```
TetFaces:  Array[0..3,0..2] of GLint=((0,1,2),(0,3,1),(3,2,1),(2,3,0));
```

Наконец, для задания четырех нормалей к граням тетраэдра введен массив `TetPoints`.

```
TetPoints: Array[0..3,0..2] of GLfloat=((0,0,-1),(0,-1,0),
                                        (-1,0,0),(1,1,1));
```

Последовательность команд построения тетраэдра для OpenGL представлена в процедуре `TetraBox` (листинг 13.11).

Листинг 13.11. Построение тетраэдра

```
procedure TetraBox(Size : GLfloat; DrawType : GLenum);
var  V : array[0..3, 0..2] of GLfloat;
  I,J : GLint;begin { TetraBox }
  for i:=0 to 3 do
    for j:=0 to 2 do V[i,j]:=Size*V0[i,j];
  for I := 0 to 3 do
    glBegin(DrawType);
      glNormal3fv(@TetPoints[I,0]);
      glVertex3fv(@V[TetFaces[I,0],0]);
      glVertex3fv(@V[TetFaces[I,1],0]);
      glVertex3fv(@V[TetFaces[I,2],0]);
    glEnd;
end; { TetraBox }
```

В этой процедуре, после масштабирования координат вершин, определяются четыре грани. Описание каждой грани начинается командой `glBegin(DrawType)`. Параметр `DrawType` может принимать два значения: `GL_LINE_LOOP` (рисовать ребра) или `GL_TRIANGLES` (рисовать треугольники). Для каждой грани командой `glNormal3fv` задается нормаль. Командами `glVertex3fv` задаются три вершины.

Для рисования тетраэдра ребрами и гранями предназначены две процедуры (листинг 13.12).

Листинг 13.12. Рисование тетраэдра ребрами и гранями

```
procedure glutWireTetra(Size : GLDouble);
begin { glutWireTetra }
  TetraBox(Size, GL_LINE_LOOP);
end; { glutWireTetra }

procedure glutSolidTetra(Size : GLDouble);
begin { glutSolidCube }
  TetraBox(Size, GL_TRIANGLES);
end; { glutSolidTetra }
```

13.5.3. Списки команд

Другой способ рисования заключается в следующем: в начале работы программы командой `glNewList..glEndList` подготавливается список или несколько списков команд, в которых перечислены команды-примитивы (листинг 13.13). Рисование

может быть вызвано в любой момент, пока существует список, командой `glCallList(N)`, с указанием номера списка.

В листинге 13.13 создается список с номером `1`, задающий две пересекающиеся фигуры: куб и тетраэдр. Во время рисования будет показана новая фигура, получающаяся в результате объединения куба и тетраэдра (рис. 13.8).

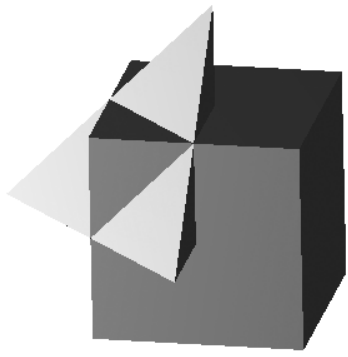


Рис. 13.8. Объединение куба и тетраэдра

Листинг 13.13. Подготовка списка примитивов

```
procedure MyDrawBoxList(Size : GLfloat);
var V3  : array[0..3, 0..2] of GLfloat;
    V,N : array[0..7, 0..2] of GLfloat;
    I,J : GLint;
begin { MyDrawBoxList }
  for i:=0 to 7 do
    for j:=0 to 2 do V[i,j]:=Size*BoxVertex[i,j]/2;
  V3[0,0]:=Size; V3[1,0]:=0; V3[2,0]:=0; V3[3,0]:=0;
  V3[0,1]:=0; V3[1,1]:=0; V3[2,1]:=Size; V3[3,1]:=0;
  V3[0,2]:=0; V3[3,2]:=Size; V3[1,2]:=0; V3[2,2]:=0;
  glNewList(1,GL_COMPILE);
    for I := 0 to 5 do begin
      glNormal3fv(@BoxPoints[I, 0]);
      glBegin(GL_QUADS);
        for j:=0 to 3 do glVertex3fv(@V[BoxFaces[I,j],0]);
      glEnd;
    end;
    for I := 0 to 3 do begin
      glBegin(GL_TRIANGLES);
        glNormal3fv(@TetPoints[I, 0]);
        glVertex3fv(@V3[TetFaces[I, 0], 0]);
        glVertex3fv(@V3[TetFaces[I, 1], 0]);
```

```
        glVertex3fv(@V3[TetFaces[I, 2], 0]);
      glEnd;
    end;
  glEndList;
end; { MyDrawBoxList }
```

13.5.4. Изображение квадратичных поверхностей

В проекте возможно построение восьми квадратичных поверхностей, заданных параметрически. Уравнения этих поверхностей задает следующая процедура (листинг 13.14).

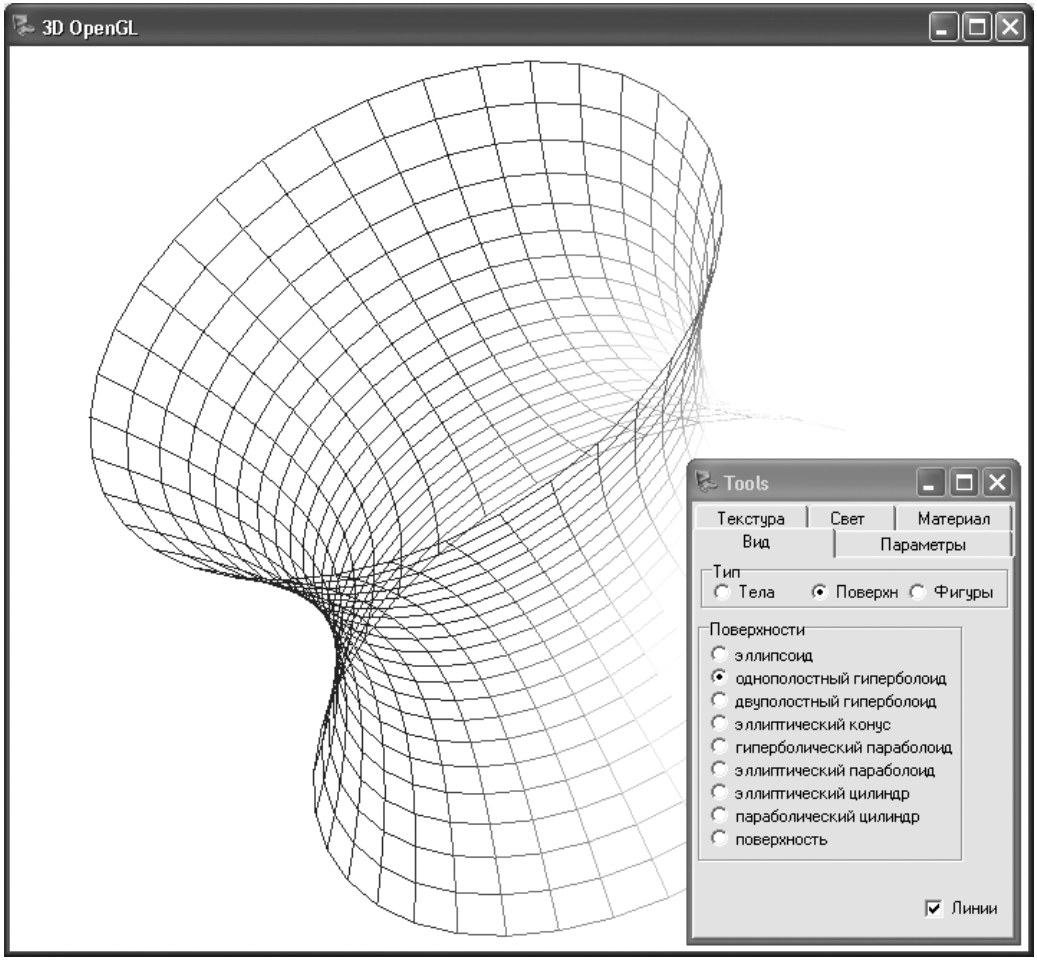


Рис. 13.9. Однополостной гиперболоид

Листинг 13.14. Уравнения квадратичных поверхностей

```
procedure XYZ(n: integer; t1,t2: real; var x,y,z: GLfloat);
begin
  case n of
    0: begin    // эллипсоид
         x:=a*Sin(t1)*Sin(t2);
         y:=b*Cos(t1);
         z:=c*Sin(t1)*Cos(t2);
       end;
    …
    7: begin    // параболический цилиндр
         x:=a*t2;
         y:=b*t1;
         z:=c*t2*t2;
       end;
  end;
end;
```

Одна из поверхностей — однополостной гиперболоид, изображена на рис. 13.9.

Последовательность действий при рисовании квадратичных поверхностей представлена в листинге 13.15.

Листинг 13.15. Рисование квадратичных поверхностей

```
      InitData(k);
      h1:=(a2-a1)/(n2-n1); h2:=(b2-b1)/(m2-m1);
      for j:=n1 to n2-1 do
      for i:=m1 to m2-1 do begin
        t1:=h1*j; t2:=h2*i; XYZ(k,t1,t2,x0[0],x0[1],x0[2]);
        t1:=h1*j; t2:=h2*(i+1); XYZ(k,t1,t2,x1[0],x1[1],x1[2]);
        t1:=h1*(j+1);
        t2:=h2*(i+1); XYZ(k,t1,t2,x2[0],x2[1],x2[2]);
        t1:=h1*(j+1); t2:=h2*i; XYZ(k,t1,t2,x3[0],x3[1],x3[2]);

        N:=getNormal(k,x0,x1,x2,t1);

        glBegin(DrawType);
          glNormal3fv(@N);
          glVertex3fv(@x0[0]);
          glVertex3fv(@x1[0]);
          glVertex3fv(@x2[0]);
          glVertex3fv(@x3[0]);
        glEnd;
        if sg<>0 then begin
          x0[1]:=-x0[1]; x1[1]:=-x1[1];
          x2[1]:=-x2[1]; x3[1]:=-x3[1];
          N:=getNormal(k,x0,x1,x2,t1);
          glBegin(DrawType);
            glNormal3fv(@N);
            glVertex3fv(@x0[0]);
```

```
          glVertex3fv(@x1[0]);
          glVertex3fv(@x2[0]);
          glVertex3fv(@x3[0]);
        glEnd;
      end;
```

По прямоугольной сетке `n1*m1` вычисляются координаты четырех точек, и строится четурехугольник. Многие поверхности двухсвязны (это показывает переменная `sg`, принимающая отрицательное значение): вторая половина поверхности получается изменением знака координаты `y`.

13.5.5. Изображение поверхности, заданной табличным способом

В геоинформационных системах часто возникает проблема изображения участка земной поверхности или слоя породы [111, 112]. Мы рассмотрим случай, когда на прямоугольном участке (`Gx1`, `Gx2`) (`Gy1`, `Gy2`) в узлах регулярной сетки заданы значения `V[i,j]: real`. По оси `X` прямоугольник разбит на `nCol` частей, а по оси `Y` — на `nRow` частей. Данные такого типа представлены, например, в файле XYZ.TXT. Для чтения файла предназначена процедура `Read_V` (листинг 13.16).

Листинг 13.16. Чтение табличных данных

```
procedure ReadXYZ(FileName: string);
const d=1.7;
var
  f: textfile;
  i,j,L,Lp: integer;
  h1,h2: real;
begin
  AssignFile(f,FileName); Reset(f);  // чтение файла
  Readln(f);
  Readln(f,nCol,nRow); SetLength(V,nCol,nRow);
  Readln(f,Gx1,Gx2);
  Readln(f,Gy1,Gy2);
  Readln(f,Gz1,Gz2);
  for j:=0 to nRow-1 do begin
    for i:=0 to nCol-1 do begin
      Read(f,V[i,j]);
      if V[i,j]<Gz1 then Gz1:=V[i,j];
      if V[i,j]>Gz2 then Gz2:=V[i,j];
    end;
    Readln(f);
  end;
  CloseFile(f);
```

```
h1:=d/NRow; h2:=d/NCol;
with MyBody do begin
  // заполнение массива точек
  Lp:=nRow*nCol;
  SetLength(Points,Lp);
  SetLength(Texture,Lp);
  for j:=0 to nRow-1 do
  for i:=0 to nCol-1 do
  with Points[j*nCol+i] do begin
    x:=-d/2+j*h1;
    y:=-d/2+i*h2;
    z:=(V[i,j]-(Gz2+Gz1)/2)/(Gz2-Gz1)*Size;
    Texture[j*nCol+i].x:=j/NRow;
    Texture[j*nCol+i].y:=i/NCol;
  end;
  // заполнение массива граней
  L:=2*(nRow-1)*(nCol-1);
  SetLength(Faces,L);
  for j:=0 to nRow-2 do
  for i:=0 to nCol-2 do begin
    // первый треугольник
    with Faces[j*(nCol-1)+i] do begin
      Vertex[0]:=j*nCol+i;
      Vertex[1]:=j*nCol+i+1;
      Vertex[2]:=(j+1)*nCol+i+1;
      Norm:=getNormalP(Points[Vertex[0]],Points[Vertex[1]],
                       Points[Vertex[2]]);
    end;
    // второй треугольник
    with Faces[j*(nCol-1)+i+(nRow-1)*(nCol-1)] do begin
      Vertex[0]:=j*nCol+i;
      Vertex[1]:=(j+1)*nCol+i+1;
      Vertex[2]:=(j+1)*nCol+i;
      Norm:=getNormalP(Points[Vertex[0]],Points[Vertex[1]],
                       Points[Vertex[2]]);
    end;
  end;
  // вычисление нормали в точках
  SetLength(Norms,Lp);
  for i:=0 to Lp-1 do
  with Norms[i] do begin
    x:=0; y:=0; z:=0;
    for j:=0 to L-1 do
    with Faces[j] do
    if (i=Vertex[0]) or (i=Vertex[1]) or (i=Vertex[2]) then
```

```
        Norms[i]:=SumPoint(Norms[i],Norm);
      Norms[i]:=NormaTo1(Norms[i]);
    end;
  end;
end;
```

После чтения текстового файла заполняются поля переменной `MyBody: TmyBody`:

```
  TMyBody = record
    Points : TAPoints;
    Faces  : TFaces;
    Norms  : TAPoints;
    Texture: array of TPoints2;
  end; ,
```

где

```
  TFace=record
    Vertex: array[0..2] of integer;
    Norm  : TPoints;
  end;
```

Поверхность будет задаваться треугольниками и эти `2*(nCol-1)*(nRow-1)` треугольников задаются в процедуре `DrawFace` (листинг 13.17).

Листинг 13.17. Задание поверхности треугольниками

```
procedure DrawFace;
// поверхности
var
  NumFace,i: integer;
begin
  NumFace:=Length(MyBody.Faces);
  if NumFace>0 then begin
    if FormTools.CheckBox1.Checked
      then DrawType:=GL_LINE_LOOP
      else DrawType:=GL_TRIANGLES;
    with MyBody do begin
      for i:=0 to NumFace-1 do
      with Faces[i] do begin
        glBegin(DrawType);
          glNormal3f(Norms[Vertex[0]].x,Norms[Vertex[0]].y,
                     Norms[Vertex[0]].z);
          glTexCoord2d(Texture[Vertex[0]].x,Texture[Vertex[0]].y);
          glVertex3f(Points[Vertex[0]].x,Points[Vertex[0]].y,
                     Points[Vertex[0]].z);
          glNormal3f(Norms[Vertex[1]].x,Norms[Vertex[1]].y,
                     Norms[Vertex[1]].z);
          glTexCoord2d(Texture[Vertex[1]].x,Texture[Vertex[1]].y);
          glVertex3f(Points[Vertex[1]].x,Points[Vertex[1]].y,
```

```
                    Points[Vertex[1]].z);
          glNormal3f(Norms[Vertex[2]].x,Norms[Vertex[2]].y,
                    Norms[Vertex[2]].z);
          glTexCoord2d(Texture[Vertex[2]].x,Texture[Vertex[2]].y);
          glVertex3f(Points[Vertex[2]].x,Points[Vertex[2]].y,
                    Points[Vertex[2]].z);
        glEnd;
      end;
    end;
  end;
end;
```

Поверхность, заданная в файле XYZ.TXT, изображена на рис. 13.10.

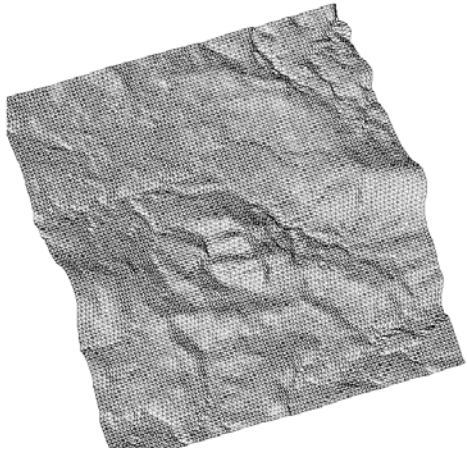


Рис. 13.10. Поверхность, заданная табличным способом

13.6. Геометрические преобразования

Будем считать, что ось X расположена в плоскости экрана и смотрит вправо, а ось Y, также расположенная в плоскости экрана, смотрит вверх. Если ось Z направлена на наблюдателя, то система координат — правосторонняя, если же она направлена от наблюдателя, то левосторонняя.

Координаты точек любого объекта описываются в общепринятой правосторонней системе координат. Для вывода изображения на экране необходимо преобразовать эти координаты к видовой левосторонней системе координат. Для этого нужно повернуть тело, учитывая, что нумерация строк пикселов на экране идет сверху вниз, и спроектировать его на экран (картинную плоскость).

Для того чтобы представлять все виды преобразований в единой форме, будем использовать однородные координаты. Для описания положения точки в трехмерном

пространстве в однородных координатах вводится масштабный множитель W, и формула записывается в виде $P(W \times x, W \times y, W \times z)$ или $P(X, Y, Z, W)$.

Преобразование однородных координат можно описать с помощью матрицы T матричным соотношением:

$$\|X \quad Y \quad Z \quad W\| = T \begin{Vmatrix} x \\ y \\ z \\ 1 \end{Vmatrix}.$$

Приведем матрицы часто используемых преобразований системы координат: параллельного переноса T_M и изменения масштаба T_D:

$$T_M = \begin{Vmatrix} 1 & 0 & 0 & x0 \\ 0 & 1 & 0 & y0 \\ 0 & 0 & 1 & z0 \\ 0 & 0 & 0 & 1 \end{Vmatrix}; \quad T_D = \begin{Vmatrix} k1 & 0 & 0 & 0 \\ 0 & k2 & 0 & 0 \\ 0 & 0 & k3 & 0 \\ 0 & 0 & 0 & 1 \end{Vmatrix}.$$

В OpenGL используется три матрицы (4×4): видовая матрица, матрица проекций и матрица текстуры. Прежде чем работать с матрицей, ее необходимо сделать текущей, используя команду

```
GlMatrixMode(mode: GLenum)
```

Параметр mode может принимать три значения:

- `GL_MODELVIEW` — устанавливается матрица видовых преобразований;
- `GL_PROJECTION` — устанавливается матрица проекций;
- `GL_TEXTURE` — устанавливается матрица текстуры.

Сама матрица загружается командой `glLoadMatrix[f d](p: PGLtype)`, в которой параметр `p` указывает на матрицу, хранящуюся в памяти по столбцам, как 16 последовательных элементов. Для загрузки единичной матрицы предусмотрена специальная команда `glLoadIdentity`.

Очень важной является команда `glMultMatrix[f d](p: PGLtype)`. По этой команде матрица, на которую указывает параметр `p`, умножается на текущую матрицу.

Существует несколько специализированных команд для переноса, вращения и масштабирования системы координат:

- команда `glTranslate[f d](x, y, z: GLtype)` — переносит объект на вектор `(x, y, z)`;
- команда `glRotate[f d](angle, x, y, z: GLtype)` — поворачивает объект на угол `angle` вокруг вектора `(x, y, z)`;

- команда `glScale[f d](x, y, z: GLtype)` — масштабирует объект с коэффициентами `x`, `y`, `z`.

Рассмотрим более подробно другой вид преобразования — проецирование. OpenGL поддерживает два вида проекций: ортографическую (параллельную) и перспективную с одной точкой схода.

Ортографическая проекция задается командой

`glOrtho(left, right, bottom, top, near, far: GLdouble).`

Параметры `left`, `right`, `bottom`, `top`, `near`, `far` задают параллелепипед видимости.

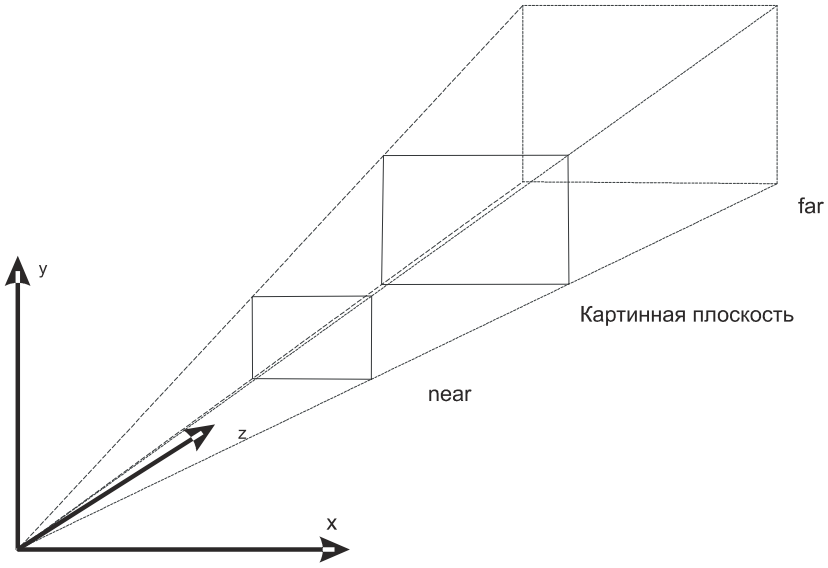


Рис. 13.11. Усеченная пирамида видимости

Эта команда генерирует матрицу проектирования M:

$$M = \begin{Vmatrix} 2/(right-left) & 0 & 0 & t_x \\ 0 & 2/(top-bottom) & 0 & t_y \\ 0 & 0 & -2/(far-near) & t_z \\ 0 & 0 & 0 & 1 \end{Vmatrix},$$

где $t_x = -\dfrac{right+left}{right-lef}$; $t_y = -\dfrac{top+bottom}{top-bottom}$; $t_z = -\dfrac{far+near}{far-near}$.

Наиболее "капризными" параметрами являются расстояние до ближней и дальней плоскостей отсечения по оси OZ — *near* и *far*. При малых значениях этих параметров во время выполнения возникает ошибка, если тело не попадает в заданный параллелепипед.

Для перспективного проецирования необходимо задать усеченную пирамиду видимости (рис. 13.11).

Задать эту пирамиду можно двумя командами: `glFrustum` и `gluPerspective`. Первая команда явно задает параметры пирамиды:

```
glFrustum (left, right, bottom, top, near, far: GLdouble).
```

Параметры `left`, `right`, `bottom`, `top` определяют размеры ближней секущей плоскости. Оба расстояния до секущих плоскостей должны быть положительными. Команда создает следующую матрицу преобразования:

$$M = \begin{Vmatrix} \frac{2\times near}{right-left} & 0 & A & 0 \\ 0 & \frac{2\times near}{top-bottom} & B & 0 \\ 0 & 0 & C & D \\ 0 & 0 & -1 & 0 \end{Vmatrix},$$

где

$$A = \frac{right+left}{right-lef}; \quad B = \frac{top+bottom}{top-bottom}; \quad C = -\frac{far+near}{far-near}; \quad D = \frac{2\times far\times near}{far-near}.$$

Вторая команда `gluPerspective(angley, aspect, near, far: GLdouble)` задает угол видимости в градусах `angley`, отношение сторон `aspect` и расстояния до плоскостей отсечения. Эта команда эквивалентна команде `glFrustum` при `Left=-right, bottom=-top, tg(angley/2)=top/nead, aspect = right/top`.

13.7. Цвет, освещение, свойства материала

Когда OpenGL рисует поверхности, освещенность поверхностей зависит от свойств диффузионного и зеркального отражения и от свойств диффузионного и направленного пропускания света. Поэтому при расчете освещенности поверхности необходимо учитывать вектор нормали, свойства материала, параметры источников света и параметры освещения.

13.7.1. Цвет

В библиотеке OpenGL используется цветовая модель RGBA. Возможно два режима установки цветов: индексный (использование цвета из палитры по индексу) и задание значений долей красного, зеленого и синего цветов.

Индексный режим используется реже и его использование может быть вызвано или ограниченным числом битовых плоскостей, или использованием цветовой анимации. В индексном режиме две команды устанавливают новое значение индекса в палитре:

```
glIndex[s i f d](index: GLtype)
glIndex[s i f d]v(index: GLtype)
```

Для работы в режиме RGBA также предоставлены две команды:

```
glColor[3 4][b s i f d](components: GLtype)
glColor[3 4][b s i f d]v(components: GLtype)
```

В первой из этих команд задаются доли красного, зеленого и синего цветов, а во второй устанавливается еще и значение альфа. Внутренним представлением параметров `components` является представление в вещественном виде из диапазона от 0 до 1. При записи команд допустимо использование целых параметров: при этом происходит автоматическое приведение их значений к диапазону от 0 до 1.

Назначая текущий цвет, необходимо помнить, что он действует на все создаваемые после этого примитивы. В частности, при рисовании треугольника можно задать одинаковый цвет для всех вершин (листинг 13.18).

Листинг 13.18. Задание однородного цвета фигуры

```
glColor3f(0.0,0.0,1.0);
glBegin(GL_TRIANGLES);
  glVertex2f(100,100);
  glVertex2f(100,200);
  glVertex2f(200,200);
glEND;
glFinish;
```

Или, в режиме плавного совмещения цветов `glEnable(GL_SMOOTH)`, можно назначить для всех вершин разные цвета (листинг 13.19).

Листинг 13.19. Задание неоднородного цвета фигуры

```
glEnable(GL_SMOOTH);
glBegin(GL_TRIANGLES);
  glColor3f(0.0,0.0,1.0);
  glVertex2f(100,100);
  glColor3f(0.0,1.0,0.0);
  glVertex2f(100,200);
  glColor3f(1.0,0.0,0.0);
  glVertex2f(200,200);
glEND;
glFinish;
```

Этот прием можно использовать для сглаживания углов между гранями многогранника, определяя для вершин цвета в соответствии с нормалями к поверхности гладкого тела, а не к грани.

13.7.2. Нормали

Нормаль к поверхности определяет направление зеркального отражения и направленного пропускания света. В OpenGL нормаль можно ассоциировать с каждой вершиной, и для определения ее координат предусмотрены две команды:

```
glNormal3[b s i f d](coords: type)
glNormal3[b s i f d]v(coords: type)
```

Целочисленные аргументы приводятся в вещественный формат в диапазоне от 0 до 1. При включенном режиме нормализации `glEnable(GL_NORMALIZE)` вектор нормали автоматически приводится к единичной длине.

13.7.3. Свойства материала

Материал может обладать рассеянным и диффузионным цветом, цветом зеркального отражения и излучаемым светом.

Для определения таких свойств материала предназначены две функции:

```
glMaterial[i f](face, pname: Glenum; param: GLtype)
glMaterial[i f]v(face, pname: Glenum; params: PGLtype)
```

Первая команда может определять только степень зеркального отражения, а вторая команда — более "мощная".

Параметры этих команд имеют следующий смысл:

- `face` — может принимать значения `GL_FRONT` (лицевые стороны поверхности), `GL_BACK` (обратные) и `GL_FRONT_AND_BACK` (и те и другие);
- `pname` — определяет, какой параметр будет обновляться;
- `params` — ссылка на массив, содержащий характеристики цвета в представлении RGBA либо индексном.

Для первой команды `glMaterial[i f]` аргумент `pname` может принимать только одно значение — `GL_SHININESS`, позволяющее определять степень зеркального отражения, которая задается аргументом `param`. Для второй (векторной) команды аргумент `pname` может принимать несколько значений, для каждого из которых задаются характеристики цвета. В табл. 13.5 представлены возможные значения аргумента `pname` и соответствующие цветовые характеристики, устанавливаемые по умолчанию.

Таблица 13.5. *Значения параметра pname и характеристики цвета*

Значение *pname*	Характеристики цвета
GL_AMBIENT	4 значения RGBA, определяющие рассеянный цвет. По умолчанию (0,2, 0,2, 0,2, 1,0)
GL_DIFFUSE	4 значения RGBA, определяющие диффузионный цвет. По умолчанию (0,8, 0,8, 0,8, 1,0)
GL_SPECULAR	4 значения RGBA, определяющие цвет зеркального отражения. По умолчанию (0,0, 0,0, 0,0, 1,0)
GL_EMISSION	4 значения RGBA, определяющие интенсивность излучаемого света. По умолчанию (0,0, 0,0, 0,0, 1,0)
GL_SHININESS	Одно целое или вещественное значение (0..128), определяющее степень зеркального отражения
GL_ AMBIENT_ AND_DIFFUSE	Эквивалентно GL_AMBIENT и GL_DIFFUSE с одинаковыми значениями параметра
GL_COLOR_INDEXES	3 целых или вещественных значения, определяющих индексы цветов для рассеянного, диффузионного и зеркального отражения

В листинге 13.20 приведен пример установки параметров материала.

Листинг 13.20. Установка параметров материала

```
const mat_ambient : array[1..4] of GLfloat =
        (0.2,0.2,0.0,0.5); // - рассеянный цвет материала
      mat_diffuse : array[1..4] of GLfloat =
        (0.7,0.7,0.0,0.5);
        // - цвет диффузионного отражения материала
      mat_specular : array[1..4] of GLfloat =
        (0.1,0.1,0.0,0.5);
        // - цвет зеркального отражения материала
      mat_shininess : array[1..1] of GLfloat = (20.0);
        // - степень зеркального отражения
begin
  glMaterialfv(GL_FRONT, GL_AMBIENT,@mat_ambient);
  glMaterialfv(GL_FRONT, GL_DIFFUSE,@mat_diffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR,@mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS,@mat_shininess);
end;
```

13.7.4. Источники света

При рисовании с помощью OpenGL могут использоваться несколько точечных источников света. Каждый из них задается командами:

```
glLight[i f](light, pname: GLenum; param: GLfloat)
glLight[i f]v(light, pname: GLenum; params: PGLfloat).
```

Эти команды отличаются тем, что первая задает единственное значение параметра `param`, а вторая, векторная, определяет четыре значения — для каждого цвета RGBA.

Аргумент `light` определяет номер источника света в диапазоне `0..GL_MAX_LIGTS=8`. Аргумент `pname` определяет устанавливаемый параметр. Возможные значения аргументов `pname` и `param` для невекторной команды приведены в табл. 13.6.

Таблица 13.6. *Значения аргументов команды glLight*

Значение *pname*	**Значения *param***
`GL_SPOT_EXPONENT`	Единственное целое [0..128] или вещественное значение, задающее распределение интенсивности света. Чем больше параметр, тем сильнее сфокусирован источник света
`GL_SPOT_CUTOFF`	Единственное целое [0..90,180] или вещественное значение, задающее максимальный угол разброса света

`GL_CONSTANT_ATTENUATION`
`GL_LINEAR_ATTENUATION`
`GL_QUADRATIC_ATTENUATION`
	По одному целому или вещественному положительному значению, задающему факторы постоянного, линейного и квадратичного ослабления света

Для векторной команды `GlLight[i f]v` аргумент `pname` может принимать значения, указанные в табл. 13.6, и, кроме того, пять дополнительных значений. Эти значения, а также соответствующие им цветовые характеристики приведены в табл. 13.7.

Таблица 13.7. *Дополнительные значения аргументов для команды glLightv*

Значения *pname*	**Характеристики цвета**
`GL_AMBIENT`	4 целых или вещественных значения RGBA, определяющих интенсивность фонового освещения
`GL_DIFFUSE`	4 целых или вещественных значения RGBA, определяющих интенсивность диффузионного освещения
`GL_SPECULAR`	4 целых или вещественных значения RGBA, определяющих интенсивность зеркального освещения
`GL_POSITION`	4 целых или вещественных значения, определяющих положение источника света
`GL_SPOT_DIRECTION`	4 целых или вещественных значения, определяющих вектор направления света

Если параметры i-го источника света заданы, то их можно включать и выключать командами glEnable(GL_LIGHTi) и glDisable(GL_LIGHTi) (листинг 13.21).

Листинг 13.21. Установка одного источника света

```
procedure TForm1.SetLight;
const light_position : array[1..4] of GLfloat =
        (0.0,5.0,20.0,1.0);
      light_ambient1 : array[1..4] of GLfloat =
        (1.0,1.0,1.0,1.0);
      light_diffuse1 : array[1..4] of GLfloat =
        (1.0,1.0,1.0,1.0);
      light_specular1: array[1..4] of GLfloat =
        (1.0,1.0,1.0,1.0);
begin
  glLightfv(GL_LIGHT1, GL_POSITION,@light_position);
  glLightfv(GL_LIGHT1, GL_AMBIENT,@light_ambient1);
  glLightfv(GL_LIGHT1, GL_DIFFUSE,@light_diffuse1);
  glLightfv(GL_LIGHT1, GL_SPECULAR,@light_specular1);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT1);
end;
```

В OpenGL есть еще две команды, определяющие модель освещения:

```
glLightModel[i f](pname: Glenum; param: GLenum)
glLightModel[i f]v(pname: Glenum; params: PGLenum)
```

В этих командах аргумент pname определяет характеристику модели освещения, а аргумент param (params) — ее значения (табл. 13.8).

Таблица 13.8. *Значение аргументов команды glLightModel*

Значение *pname*	**Значения *param* (*params*)**
GL_LIGHT_MODEL_LOCAL_VIEWER	Одно логическое значение, определяющее положение наблюдателя. При false направление обзора всегда направлено вдоль оси Z. При true наблюдатель находится в начале системы координат
GL_ LIGHT_MODEL_TWO_SIDE	Одно логическое значение, определяющее одно- (false) или двустороннее (true) освещение граней
GL_ LIGHT_MODEL_AMBIENT	4 целых или вещественных значения, определяющих интенсивность фонового освещения

13.8. Текстура

Текстура — техника, которая накладывает изображение на поверхность объекта. Текстура включает в себя одномерное или двумерное изображение и ряд парамет-

ров, определяющих, как образцы будут получены из изображения. Одномерная текстура может использоваться только для нанесения узора в виде полосок, двумерная позволяет накладывать на грани прямоугольные изображения.

Процесс наложения текстуры на грани состоит из следующих обязательных пунктов:

- каждой вершине грани необходимо назначить точку карты текстуры;
- с помощью процедуры `glTexImage1d` или `glTexImage2d` создать одномерную или двумерную карту текстуры, передавая ей, например, массив долей цветов RGB из изображения формата `BMP`;
- процедурой `glTexParameter[f|i|fv|iv]` задать параметры;
- командой `glEnable(GL_TEXTURE_1|2D)` разрешить пользоваться текстурой.

13.8.1. Назначение точки карты текстуры вершине

Назначение точки карты текстуры реализуется командой `glTexCoord2d`. Так, например, вершине `x` назначается точка карты `s,t`:

```
glTexCoord2d (s,t);
glVertex3fv(x)
```

Параметры `t` и `s` могут меняться в диапазоне [0,1] и накрывают всю прямоугольную карту текстуры.

> ***ЗАМЕЧАНИЕ***
>
> Некоторые многогранники модуля OpenGL (сфера — `gluSphere` и цилиндр — `gluCylinder`) и модуля DGLUT (чайник — `glutSolidTeapot`) уже сопоставляют свои вершины с точками карты текстуры.

> ***ЗАМЕЧАНИЕ***
>
> Для двумерной текстуры размеры массива образа должны быть степенью двух.

13.8.2. Задание параметров текстуры

Функция `glTexParameter[f|i|fv|iv]` назначает одно значение или несколько значений `param` параметру структуры `NameParam`. Параметр `target` определяет тип текстуры и, для одномерной текстуры, должен принимать значение `GL_TEXTURE_1D`, а для двумерной — `GL_TEXTURE_2D`.

Для назначения одного вещественного или целого параметра `param` предназначены две процедуры:

```
procedure glTexParameterf(
target   : GLenum;
```

```
NameParam: GLenum;
param    : GLfloat
);
procedure glTexParameteri(
target   : GLenum;
NameParam: GLenum;
param    : GLint
);
```

Параметр `NameParam` может принимать следующие значения:

`GL_TEXTURE_MIN_FILTER`,
`GL_TEXTURE_MAG_FILTER`,
`GL_TEXTURE_WRAP_S`,
`GL_TEXTURE_WRAP_T`.

Еще две процедуры предназначены для назначения многих параметров с помощью указателя на вещественные или целые данные `params`:

```
procedure glTexParameterfv(
target       : GLenum;
NameParam    : GLenum;
const *params: GLfloat
);
procedure glTexParameteriv(
target       : GLenum;
NameParam    : GLenum;
const *params: GLint
);
```

Для последних двух процедур параметр `NameParam` может принимать одно из следующих значений:

`GL_TEXTURE_MIN_FILTER`,
`GL_TEXTURE_MAG_FILTER`,
`GL_TEXTURE_WRAP_S`,
`GL_TEXTURE_WRAP_T`,
`GL_TEXTURE_BORDER_COLOR`.

Обсудим значения параметра `NameParam` более подробно.

При значении `GL_TEXTURE_MIN_FILTER` используется уменьшающая функция, если одному пикселу соответствует более чем один элемент текстуры.

Существует шесть функций уменьшения. Две из них используют для вычисления значения текстуры один самый близкий или четыре самые близких элемента текстуры. Другие четыре функции используют так называемые "мипмапы" (от слова `mipmap`).

"Мипмап" — это набор множеств, представляющих то же самое изображение в более низких разрешениях. Если текстура имеет размер $2^n \times 2^m$, то число "мипмапов"

равно $Max(n,m)+1$. Первый "мипмап" — это оригинальная текстура с размерами $2^n \times 2^m$. Каждый последующий "мипмап" имеет размер $2^{k-1} \times 2^{L-1}$, где $2^k \times 2^L$ — размер предыдущего "мипмапа", до $k=0$ или до $L=0$. В этом случае "мипмапы" имеют размер $1 \times 2^{L-1}$ или $2^{k-1} \times 1$ до последнего "мипмапа", который имеет размер 1×1. "Мипмапы" определяются командой `glTexImage1D` или `glTexImage2D` с аргументом уровня детализации, указывающим число "мипмапов". Уровень 0 — первоначальная структура, имеющая размер $2^n \times 2^m$, последний уровень соответствует размеру 1×1. Наличие "мипмапов" позволяет OpenGL использовать для мелких объектов подходящие по размеру образы.

Параметр `params` в этом случае предоставляет функцию для уменьшения текстуры по одному из следующих вариантов:

- `GL_NEAREST` — возвращает значение элемента текстуры, который является самым близким к центру пиксела;
- `GL_LINEAR` — возвращает взвешенное среднее число четырех элементов текстуры, которые являются самыми близкими к центру пиксела. Они могут включить элементы границы текстуры в зависимости от значений `GL_TEXTURE_WRAP_S` и `GL_TEXTURE_WRAP_T`;
- `GL_NEAREST_MIPMAP_NEAREST` — выбирает "мипмап", который наиболее близко соответствует размеру пиксела, и использует критерий `GL_NEAREST` (элемент структуры, самый близкий к центру пиксела) для вычисления значения текстуры;
- `GL_LINEAR_MIPMAP_NEAREST` — выбирает "мипмап", который наиболее близко соответствует размеру пиксела, и использует критерий `GL_LINEAR` (взвешенное среднее число четырех элементов структуры, являющихся самыми близкими к центру пиксела) для вычисления значения текстуры;
- `GL_NEAREST_MIPMAP_LINEAR` — выбирает два "мипмапа", которые наиболее близко соответствуют размеру пиксела, и использует критерий `GL_NEAREST` (элемент структуры, самый близкий к центру пиксела), чтобы произвести значение текстуры от каждого `mipmap`. Окончательное значение текстуры — среднее число этих двух значений;
- `GL_LINEAR_MIPMAP_LINEAR` — выбирает два "мипмапа", которые наиболее близко соответствуют размеру пиксела, и использует критерий `GL_LINEAR` (взвешенное среднее число четырех элементов текстуры, которые являются самыми близкими к центру пиксела) для вычисления значения текстуры от каждого "мипмапа". Окончательное значение текстуры — среднее от этих двух значений.

Если текстурируемый пиксел меньше одного элемента текстуры или равен ему, используется функция увеличения текстуры `GL_TEXTURE_MAG_FILTER`. В этом случае можно устанавливать функцию увеличения текстуры в одно из двух значений:

- `GL_NEAREST` — возвращает значение элемента текстуры, который является самым близким к центру текстурируемого пиксела.

- GL_LINEAR — возвращает среднее от четырех элементов текстуры, которые являются самыми близкими к центру текстурируемого пиксела. Они могут включить элементы текстуры границы, в зависимости от значений функций GL_TEXTURE_WRAP_S и GL_TEXTURE_WRAP_T.

Значением по умолчанию для функции GL_TEXTURE_MAG_FILTER назначается GL_LINEAR.

Функция GL_TEXTURE_WRAP_S устанавливает параметр обертки для координаты s структуры в значения GL_CLAMP или GL_REPEAT. Значение GL_CLAMP заставляет координату s принадлежать диапазону [0,1] и используется для гарантированного нанесения единственного изображения на объект. Значение GL_REPEAT заставляет игнорироваться целую часть числа координаты s; в этом случае GL использует только дробную часть, создавая повторяющийся образец. По умолчанию для функции GL_TEXTURE_WRAP_S выбирается значение GL_REPEAT.

Функция GL_TEXTURE_WRAP_T устанавливает параметр обертки для координаты t структуры в значения GL_CLAMP или GL_REPEAT аналогично функции GL_TEXTURE_WRAP_S. Первоначально для функции GL_TEXTURE_WRAP_T назначается значение GL_REPEAT.

Функция GL_TEXTURE_BORDER_COLOR устанавливает цвет границы. Параметр params содержит четыре значения, которые включают цвет RGBA границы текстуры. Значения должны быть определены в диапазоне [0,1]. Первоначально, цвет границы задан как (0, 0, 0, 0).

13.8.3. Создание двумерной карты текстуры

Предположим, что текстура берется из изображения шириной Width и высотой Height, а информация о долях цветов RGB каждого пиксела находится в элементах динамического массива pixels.

После того как массив pixels заполнен, процедурой glTexImage2d устанавливается двумерная текстура. У этой процедуры имеются следующие параметры (листинг 13.22).

Листинг 13.22. Параметры процедуры установления текстуры из изображения

```
glTexImage2d(
  target: GLenum;     // размерность текстуры (GL_TEXTURE_2D)
  level,              // 0, число уровней детализации текстуры
  components: GLint; // 3
  width,              // ширина изображения
  height: GLsizei;    // высота изображения
  border: GLint;      // 0 ширина границы
  format,             // GL_RGB
```

```
  _type: GLenum;      // GL_UNSIGNED_BYTE
  pixels: Pointer     // указатель на данные о цветах
);
```

Первый параметр — `target` — указывает на размерность текстуры. Значение второго параметра `level` — число уровней детализации текстуры, обычно задается равным нулю, что соответствует базовому уровню. Если этот параметр отличен от `0`, то образ уменьшается в `level` раз.

Параметру `components`, который определяет число цветовых компонентов, можно задать значение `1`, `2`, `3` или `4`. Если выбирается `1`, то используется только красный цвет, при значении `2` — красный и альфа, при значении `3` — красный, зеленый и синий, при значении `4` — все четыре составляющих цвета.

Параметры `width=2`^{n}`+2*border` и `height=2`^{m}`+2*border` задают ширину и высоту изображения. Ширина границы задается параметром `border` и обычно принимается равной нулю или `1`. Параметр `format=GL_RGB` указывает на то, что в массиве `pixels` для каждого пиксела отводится по три байта. Параметр `_type=GL_UNSIGNED_BYTE` указывает на то, что каждый элемент массива данных имеет тип `byte`. На данные о цветах точек изображения показывает указатель `pixels`.

Далее приводится код процедуры `GetTextureFromFile` (листинг 13.23), которая загружает изображение из файла с именем `NameFile` в `Bitmap`, формирует массив `pixels` и передает эти данные в карту текстуры.

Листинг 13.23. Загрузка изображения из файла

```
procedure GetTextureFromFile(NameFile: string);
type
  PByteArray = ^TByteArray;
  TByteArray = array [0..0] of byte;
var
  i,j   : integer;
  Color : TColor;
  pixels: PByteArray;
begin
  Bitmap := Graphics.TBitmap.Create;
  Bitmap.LoadFromFile(NameFile);    // считываем картинку из файла

  with Bitmap do begin
    GetMem(pixels, Width*Height*3); // создаем динамический массив
    for i:=0 to Width-1 do
    for j:=0 to Height-1 do begin
      Color:=Canvas.Pixels[i,j];
      // доля красного
      pixels[j*Width*3+3*i+0]:= Color and $FF;
      // доля зеленого
      pixels[j*Width*3+3*i+1]:=(Color and $FF00) shr 8;
```

```
      // доля синего
      pixels[j*Width*3+3*i+2]:=(Color and $FF0000) shr 16;
    end;
    glTexImage2d(GL_TEXTURE_2D,0,3,Width,Height,0,GL_RGB,
                 GL_UNSIGNED_BYTE,pixels);
  end;
  FreeMem(pixels);
  Bitmap.Free;
end;
```

13.8.4. Включение режима наложения текстуры

Режим наложения двумерной текстуры включается командой `glEnable(GL_TEXTURE_2D)`.

13.8.5. Текстура на сфере, конусе и чайнике

В модуле OpenGL предусмотрена следующая процедура рисования сферы:

```
procedure gluSphere(
  quadObject: GLUquadricObj;
  radius: GLdouble;
  slices, loops: GLint);
```

В рассматриваемой процедуре каждой вершине сферы назначена точка карты текстуры. Вызов данной процедуры предполагает задание параметра `quadObject: GLUquadricObj`. Поэтому в модуле DGLUT эта процедура переопределена процедурой `glutSolidSphere`, создающей и использующей динамическую переменную `quadObj` (листинг 13.24).

Листинг 13.24. Процедура рисования сферы

```
procedure glutSolidSphere(   Radius : GLdouble;
  Slices : GLint;   Stacks : GLint);
begin { glutSolidSphere }
  if quadObj = nil then   quadObj := gluNewQuadric;
  gluQuadricDrawStyle(quadObj, GLU_FILL);
  gluQuadricNormals(quadObj, GLU_SMOOTH);
  gluSphere(quadObj, Radius, Slices, Stacks);
end;{glutSolidSphere}
```

Для рисования сферы с текстурой достаточно разрешить переменной `quadObj` из модуля DGLUT обладать текстурой. Сделать это можно только после вызова процедуры `glutSolidSphere`, в которой создается динамическая переменная `quadObj` (листинг 13.25).

Листинг 13.25. Рисование сферы с текстурой

```
GetTextureFromFile(NameFile);
glTexParameteri(GL_TEXTURE_2D,
  GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glEnable(GL_TEXTURE_2D);
glutSolidSphere(Size, 50, 50);
gluQuadricTexture (quadObj, TRUE);
```

Рис. 13.12. Шаблон из файла Earth.BMP

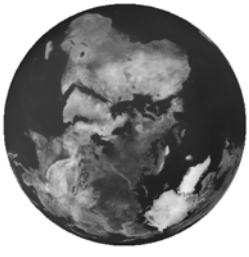


Рис. 13.13. Сфера с текстурой

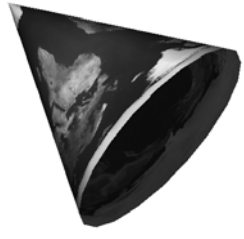


Рис. 13.14. Конус с текстурой

Рис. 13.15. Чайник с текстурой

В качестве изображения можно использовать любой файл формата BMP с размерами, кратными степени 2, например, файл `Earth.BMP` размером 1024×1024 (рис. 13.12).

В этом случае сфера будет выглядеть так, как показано на рис. 13.13.

Аналогично рисуется конус (листинг 13.26). Изображение конуса с текстурой приведено на рис. 13.14.

Листинг 13.26. Рисование конуса с текстурой

```
GetTextureFromFile(NameFile);
glTexParameteri(GL_TEXTURE_2D,
  GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glEnable(GL_TEXTURE_2D);
glutSolidCone(Size1, Size,20,20);
gluQuadricTexture (quadObj, TRUE);
```

Чайник рисуется еще проще (рис. 13.15), так как команда `glutSolidTeapot` не использует параметра `quadObj` (листинг 13.27).

Листинг 13.27. Рисование чайника с текстурой

```
GetTextureFromFile(NameFile);
glTexParameteri(GL_TEXTURE_2D,
  GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glEnable(GL_TEXTURE_2D);
glutSolidTeapot(Size);
```

Возможность наложения текстуры на остальные многогранники модуля DGLUT не предусмотрена.

13.8.6. Привязка текстуры к многоугольникам

Можно достаточно легко изменить, например, процедуру рисования куба `DrawBox` из модуля DGLUT, привязав к вершинам узловые точки `(0,0)`, `(0,1)`, `(1,1)`, `(1,0)` карты текстуры, соблюдая порядок обхода. Для этого достаточно воспользоваться командой `glTexCoord2d` (листинг 13.28).

Листинг 13.28. Задание куба с привязкой к шаблону текстуры

```
procedure MyDrawBox(Size: GLfloat; DrawType : GLenum);
var
  V : array[0..7, 0..2] of GLfloat;
  I : GLint;
begin { MyDrawBox }
  …
  for I := 0 to 5 do   begin
```

```
    glBegin(DrawType);
      glNormal3fv(@BoxPoints[I, 0]);
      glTexCoord2d(0,0);
      glVertex3fv(@V[BoxFaces[I, 0], 0]);
      glTexCoord2d(0,1);
      glVertex3fv(@V[BoxFaces[I, 1], 0]);
      glTexCoord2d(1,1);
      glVertex3fv(@V[BoxFaces[I, 2], 0]);
      glTexCoord2d(1,0);
      glVertex3fv(@V[BoxFaces[I, 3], 0]);
    glEnd;
  end;
end; // MyDrawBox
```

В качестве изображения выбран файл `24.BMP` размером 512×512. Результат рисования показан на рис. 13.16.

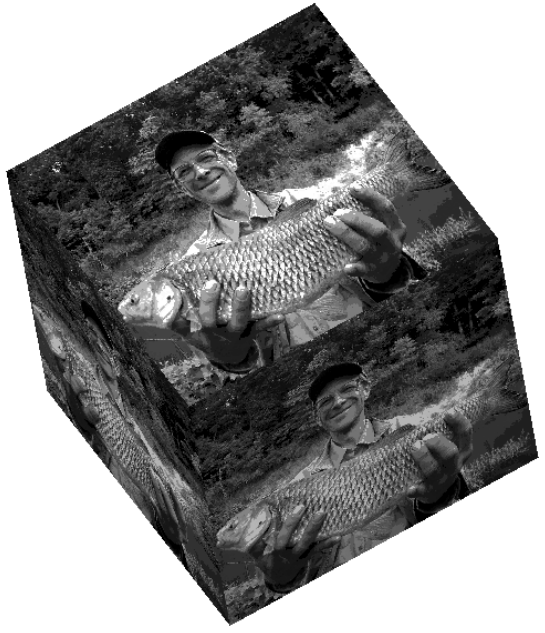


Рис. 13.16. Куб с текстурой

13.8.7. Текстура на поверхности, заданной табличным способом

Также просто накладывается текстура на поверхность, заданную табличным способом, типа той, что изображена на рис. 13.10.

Возьмем в качестве примера файл 8.BMP размером 512×512 с изображением карты местности (рис. 13.17).

Рис. 13.17. Изображение карты местности

Используем процедуру `GetSurface` (листинг 13.29), снабдив каждую вершину привязкой к карте текстуры. При вычислении параметров команды `glTexCoord2d(s,t)` надо помнить, что параметры `s` и `t` меняются от `0` до `1` и эти единичные интервалы необходимо разбивать на `nRow` и `nCol` частей, соответственно.

Листинг 13.29. Рисование поверхности с текстурой

```
procedure GetSurface(Size: GLfloat);
const
  d=1.7;
var
  i,j: integer;
  h1,h2: GLfloat;
  x0,x1,x2: TVect;
  N: TVect;
begin
  h1:=d/NRow; h2:=d/NCol;
```

```
  for j:=0 to nRow-2 do
  for i:=0 to nCol-2 do begin
    x0[0]:=-d/2+j*h1;
    x0[1]:=-d/2+i*h2;
    x0[2]:=(V[i,j]-(Gz2+Gz1)/2)/(Gz2-Gz1)*Size;
    x1[0]:=-d/2+(j+1)*h1;
    x1[1]:=-d/2+i*h2;
    x1[2]:=(V[i,j+1]-(Gz2+Gz1)/2)/(Gz2-Gz1)*Size;
    x2[0]:=-d/2+(j+1)*h1;
    x2[1]:=-d/2+(i+1)*h2;
    x2[2]:=(V[i+1,j+1]-(Gz2+Gz1)/2)/(Gz2-Gz1)*Size;
    N:=getNormal(7,x0,x1,x2,0,0);
    glBegin(DrawType);
      glNormal3fv(@N);
      glTexCoord2d (j/NRow, i/NCol);
      glVertex3fv(@x0[0]);
      glTexCoord2d ((j+1)/NRow, i/NCol);
      glVertex3fv(@x1[0]);
      glTexCoord2d ((j+1)/NRow, (i+1)/NCol);
      glVertex3fv(@x2[0]);
    glEnd;

    x0[0]:=-d/2+j*h1;
    x0[1]:=-d/2+i*h2;
    x0[2]:=(V[i,j]-(Gz2+Gz1)/2)/(Gz2-Gz1)*Size;
    x1[0]:=-d/2+(j+1)*h1;
    x1[1]:=-d/2+(i+1)*h2;
    x1[2]:=(V[i+1,j+1]-(Gz2+Gz1)/2)/(Gz2-Gz1)*Size;
    x2[0]:=-d/2+j*h1;
    x2[1]:=-d/2+(i+1)*h2;
    x2[2]:=(V[i+1,j]-(Gz2+Gz1)/2)/(Gz2-Gz1)*Size;
    N:=getNormal(7,x0,x1,x2,0,0);
    glBegin(DrawType);
      glNormal3fv(@N);
      glTexCoord2d (j/NRow, i/NCol);
      glVertex3fv(@x0[0]);
      glTexCoord2d ((j+1)/NRow, (i+1)/NCol);
      glVertex3fv(@x1[0]);
      glTexCoord2d ((j+0)/NRow, (i+1)/NCol);
      glVertex3fv(@x2[0]);
    glEnd;
  end;
end;
```

Результат наложения текстуры на поверхность, заданную табличным способом, представлен на рис. 13.18.

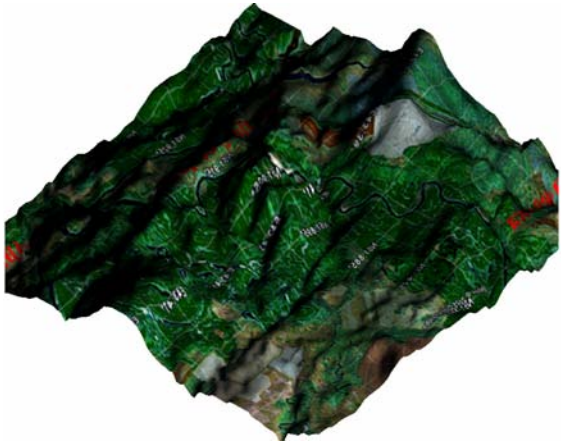


Рис. 13.18. Поверхность с текстурой карты местности

13.9. Чтение данных из текстового файла

Программа `Caligari™`, использующая графическую библиотеку DirectX, может записывать данные о многогранниках в текстовых файлах с расширением X. На прилагаемом к книге компакт-диске в папке **Фигуры** находится 20 таких файлов:

гидра.X;

кольцо тора 2.X;

кольцо тора 3.X;

кольцо тора 4.X;

кольцо тора 5.X;

кольцо тора 6.X;

кольцо 1.X;

корабль.X;

куб1.X;

лицо.X;

плоскость1.X;

пушка.X;

скейтбордист.X;

сфера1.X;

сфера2.X;

танк.X;

танк2.X;

фаска.X;

фигура.X;

человек.X.

Файлы содержат, в частности, информацию о координатах вершин, треугольниках поверхности, нормалях к каждой точке, координатах привязки текстуры к каждой вершине, информацию о материале и имеют следующую структуру:

```
Mesh Hedra01Mesh {
 42;
-0.000000;-0.000000;4.000000;,
-2.000000;3.236068;-1.236068;,
          . . . .
-1.788854;1.105573;0.000000;,
1.788854;-1.105573;-0.000000;;
80;
3;1,8,0;,
3;2,9,1;,
          . . . .
3;27,21,41;,
3;2,27,41;;
MeshNormals {
       42;
       -0.000000;-0.000000;1.000000;,
       -0.500000;0.809017;-0.309017;,
          . . . .
       -0.850651;0.525731;0.000000;,
       0.850651;-0.525731;-0.000000;;
 }

MeshTextureCoords {
       42;
       0.000000;0.500000;
       0.654508;0.904509;
          . . . .
       0.500000;0.638197;
       0.500000;0.361803;;
 }

MeshMaterialList {
       1;
       80;
       0,
```

Данные, находящиеся в начале файла, не содержат информации о вершинах и поэтому интереса для нас не представляют. В приведенном фрагменте кода из файла гидра.X показано несколько строк, содержащих такую информацию. Информация о вершинах содержится в нескольких секциях, каждая секция начинается после

строки, у которой во второй позиции есть слово Mesh. В следующей строке указано количество вершин (в приведенном примере их 42). Затем следует указанное количество строк (в нашем случае 42), в каждой из которых присутствует три координаты вершины. В следующей строке находится число граней (в нашем случае их 80). Затем следует указанное (80) количество строк, в каждой из которых присутствует число вершин грани и номера этих вершин. После строки со словом MeshNormals идет строка с числом нормалей, а затем следует указанное (в нашем случае 42) количество строк, в каждой из которых присутствует три координаты единичной нормали. После строки со словом MeshTextureCoords идет строка с числом точек привязки карты текстуры. Затем следует указанное (в нашем случае 42) количество строк, в каждой из которых присутствует две координаты точки привязки.

Для чтения текстового файла предназначена процедура ReadObject(var MyBody: TMyBody; FileName: string), которая заполняет поля переменной MyBody: TMyBody (листинг 13.30).

Листинг 13.30. Структура данных для текстового файла

```
  TMyBody = record
    Points: TAPoints;
    Faces : TFaces;
    Norms : TAPoints;
    Texture: array of TPoints2;
  end;
```

Алгоритм процедуры ReadObject достаточно прост, находится в модуле Lbr.pas и поэтому здесь не приводится.

Подготовка данных для рисования фигуры треугольниками осуществляется процедурой DrawFace, которая для каждой из трех вершин задает вектор нормали и координату карты текстуры (листинг 13.31).

Листинг 13.31. Подготовка данных для рисования фигуры

```
procedure DrawFace;
// поверхности
var
  NumFace,i: integer;
begin
  NumFace:=Length(MyBody.Faces);
  if NumFace>0 then begin
    if FormTools.CheckBox1.Checked
      then DrawType:=GL_LINE_LOOP
      else DrawType:=GL_TRIANGLES;
    with MyBody do begin
      for i:=0 to NumFace-1 do
      with Faces[i] do begin
        glBegin(DrawType);
          glNormal3f(Norms[Vertex[0]].x,Norms[Vertex[0]].y,
```

```
                    Norms[Vertex[0]].z);
          glTexCoord2d(Texture[Vertex[0]].x,Texture[Vertex[0]].y);
          glVertex3f(Points[Vertex[0]].x,Points[Vertex[0]].y,
                    Points[Vertex[0]].z);
          glNormal3f(Norms[Vertex[1]].x,Norms[Vertex[1]].y,
                    Norms[Vertex[1]].z);
          glTexCoord2d(Texture[Vertex[1]].x,Texture[Vertex[1]].y);
          glVertex3f(Points[Vertex[1]].x,Points[Vertex[1]].y,
                    Points[Vertex[1]].z);
          glNormal3f(Norms[Vertex[2]].x,Norms[Vertex[2]].y,
                    Norms[Vertex[2]].z);
          glTexCoord2d(Texture[Vertex[2]].x,Texture[Vertex[2]].y);
          glVertex3f(Points[Vertex[2]].x,Points[Vertex[2]].y,
                    Points[Vertex[2]].z);
        glEnd;
      end;
    end;
  end;
end;
```

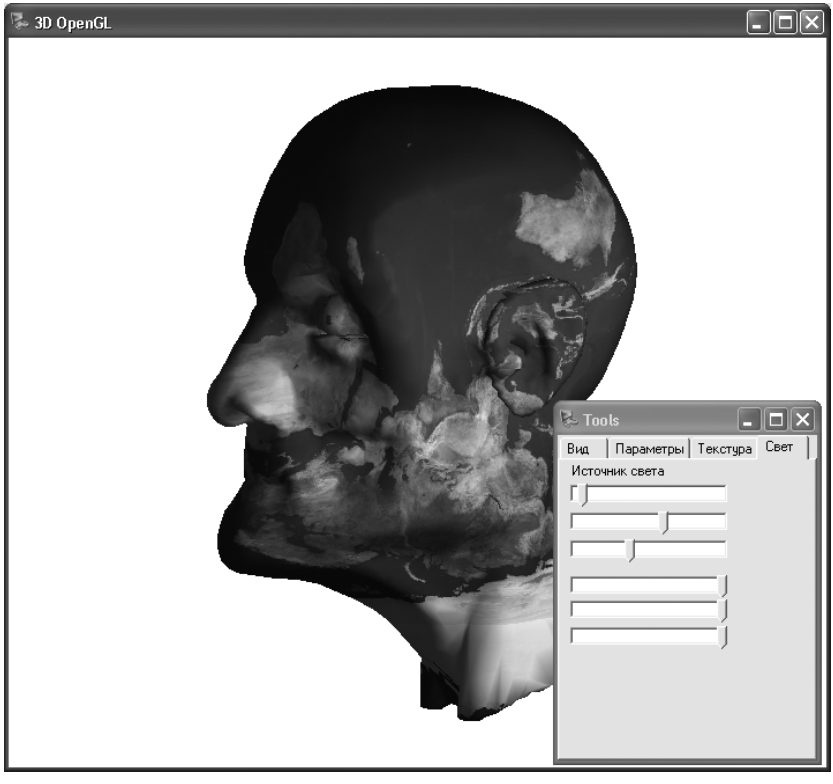


Рис. 13.19. Одна из фигур, данные для которой взяты из текстового файла **Лицо.X**

Задание вектора нормали для каждой вершины позволяет получить сглаженное изображение фигуры с наложением текстуры (рис. 13.19).

В заключении раздела 13.9 напомним, что текст процедуры `ReadObject`, осуществляющей чтение данных из текстового файла, находится в модуле `Lbr.pas` на компакт-диске в папке **Примеры | Глава 13 | 3D OpenGL**.

13.10. Проект "Редактор многогранников"

Предлагаемый проект, находящийся на компакт-диске в папке **Примеры | Глава 13 | Edit Body**, сложно назвать редактором. Скорее это манипулятор простейших 3D-тел: конусов и параллелепипедов. Проект позволяет:

- добавлять тела или группы тел;
- менять размеры выбранного тела;
- менять положение выбранного тела или группы тел;
- менять видимость тел или группы тел;
- изменять цвет тел;
- изменять цвет и положение источника света;
- поворачивать сцену.

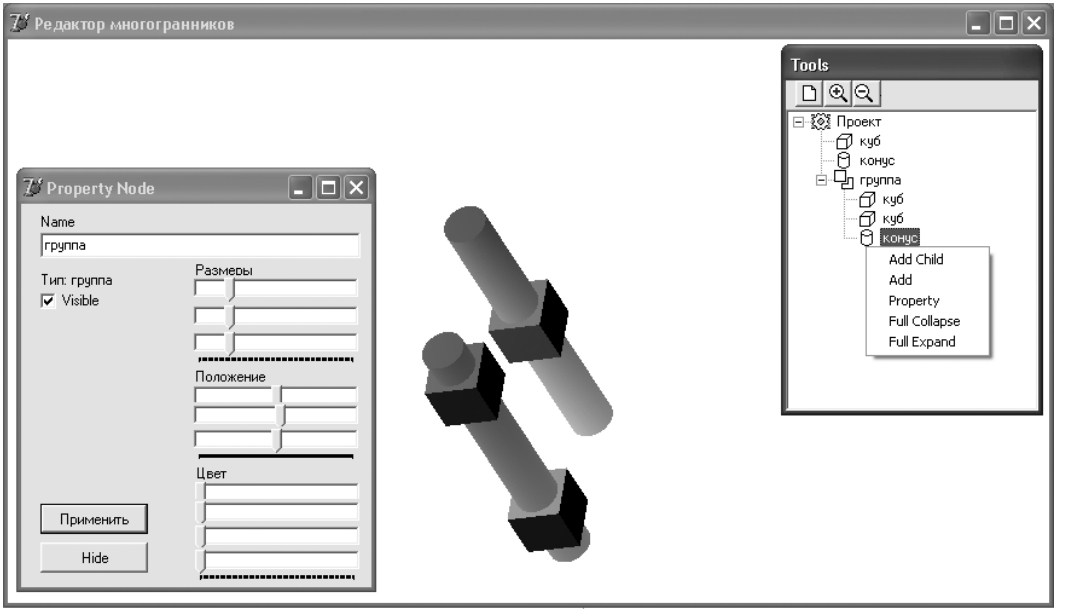


Рис. 13.20. Формы проекта "Редактор многогранников"

Проект состоит из главной формы `FormMain` (с внешним именем **Редактор многогранников**), формы для редактирования свойств узла `FormProperty` (с внешним

именем **Property Node**), инструментальной формы `FormTools` (с внешним именем **Tools**), на которую помещен компонент `TreeView` (рис. 13.20), и формы выбора типа нового узла `FormAdd` (с внешним именем **Добавить**). Форма `FormAdd` имеет тривиальный вид и содержит три радиокнопки для выбора типа добавляемого узла.

Все основные структуры и процедуры собраны в модуле `LibEdit`. В частности, в этом модуле находится рекурсивная структура `TNode`, предназначенная для описания узла, который может быть вершиной древовидной структуры, называемой проектом, группой тел, отдельным телом — кубом или конусом (листинг 13.32).

Листинг 13.32. Структура узла

```
PNode=^TNode;
TNode=record
  Name       : string;
  ParentNode : PNode;
  typeNode   : byte;//0- куб 1- конус $FE - группа $FF - проект
  Visible    : boolean;
  Size       : array[0..2] of GLfloat;
  x0         : array[0..2] of GLfloat;
  slices,
  stacks     : GLint;
  mat_ambient: array[1..4] of GLfloat;
  Child      : PNode;
  Next       : PNode;
end;
```

Поля этой структуры, описывающей бинарное дерево, содержат следующую информацию:

- `Name` — имя узла;
- `ParentNode` — указатель на родительский узел;
- `typeNode` — тип тела: куб или конус;
- `Visible` — видимость узла;
- `Size` — размеры тела;
- `x0` — координаты точки центра;
- `slices, stacks` — числа разбиений;
- `mat_ambient` — цвет тела;
- `Child` — указатель на дочерний узел;
- `Next` — указатель на следующий узел.

Поле типа узла typeNode может принимать следующие значения: 0 — куб; 1 — конус; $FE — группа и $FF — проект. Узлы типа куб или конус не могут содержать другие объекты или группы, то есть для них всегда Child=nil.

На вершину дерева, в котором содержится вся информация об узлах, показывает указатель Head: PNode. Вершину необходимо создать нажатием кнопки ButtonInitTree — крайней левой кнопки на инструментальной панели формы **Tools** (листинг 13.33).

Листинг 13.33. Создание вершины дерева

```
procedure TFormTools.ButtonInitTreeClick(Sender: TObject);
begin
  Head:=InitNode($FF,'Проект',nil);
  TreeView1.Items.Clear;
  TrNode1 := TreeView1.Items.AddObject(nil,Head.Name,Head);
  TrNode1.ImageIndex:=3;
  TrNode1.SelectedIndex:=3;
end;
```

Функция InitNode создает узел и назначает начальные значения его полям (листинг 13.34).

Листинг 13.34. Инициализация узла

```
function InitNode(fl: byte; NameNode: string;
  Parent: PNode): PNode;
begin
  New(Result);
  with Result^ do begin
    Name:=NameNode;
    Visible:=true;
    ParentNode:=Parent;
    typeNode:=fl;
    Size[0]:=0.2; Size[1]:=0.2; Size[2]:=0.2;
    slices:=48; stacks:=20;
    x0[0]:=0; x0[1]:=0; x0[2]:=0;
    mat_ambient[1]:=0; mat_ambient[2]:=0;
    mat_ambient[3]:=0; mat_ambient[4]:=0;
    Child:=nil;
    Next:=nil;
  end;
end;
```

При выборе пункта **Add Child** из контекстного меню узлов типа "группа" или "проект" вызывается процедура AddChild1Click, в которой производится выбор некоторого узла SelectNode, создается новый узел NewNode и процедурой Add-

`Node(SelectNode,NewNode)` вновь созданный узел присоединяется к выбранному (листинг 13.35).

Листинг 13.35. Добавление дочернего узла

```
procedure TFormTools.AddChild1Click(Sender: TObject);
begin
  with TreeView1 do begin
    if PNode(Selected.Data).typeNode in [$FE,$FF] then begin
      SelectNode:=PNode(Selected.Data);
      NewNode:=InitNode(0,'No Name',SelectNode);
      if FormAdd.ShowModal=mrOk then begin
        ChangeNode(FormAdd.RadioGroup1.ItemIndex,NewNode);
        TrNode1 :=
          Items.AddChildObject(Selected,NewNode.Name,NewNode);
        TrNode1.ImageIndex:=Rew(NewNode.typeNode);
        TrNode1.SelectedIndex:=Rew(NewNode.typeNode);
        AddNode(SelectNode,NewNode);
        FormMain.DrawProject;
      end;
    end
    else ShowMessage('У объекта не может быть группы!');
  end;
end;
```

Напомним, что дочерние узлы можно добавлять только к проекту и к группе.

Процедура `AddNode` добавляет дочерний узел `VNode` к узлу `Node` (листинг 13.36).

Листинг 13.36. Добавление узла

```
procedure AddNode(var Node: PNode; var VNode: PNode);
var
  t: PNode;
begin
  with Node^ do begin
    if Child=nil then Child:=VNode
    else begin
      t:=Child; while t.Next<>nil do t:=t.Next;
      t.Next:=VNode;
    end;
  end;
end;
```

При выборе пункта **Add Next** из контестного меню узла вызывается процедура `AddNextClick`, в которой определяется узел `SelectNode`, являющийся родительским для текущего и создается новый узел `NewNode`, который процедурой `AddNode(SelectNode,NewNode)` добавляется в дерево (листинг 13.37).

Листинг 13.37. Добавление следующего узла

```
procedure TFormTools.AddNextClick(Sender: TObject);
begin
  with TreeView1 do begin
    if PNode(Selected.Data).typeNode <> $FF then begin
      SelectNode:=PNode(Selected.Data).ParentNode;
      NewNode:=InitNode(0,'No Name',SelectNode);
      if FormAdd.ShowModal=mrOk then begin
        ChangeNode(FormAdd.RadioGroup1.ItemIndex,NewNode);
        TrNode1 := Items.AddObject(Selected,NewNode.Name,NewNode);
        TrNode1.ImageIndex:=Rew(NewNode.typeNode);
        TrNode1.SelectedIndex:=Rew(NewNode.typeNode);
        AddNode(SelectNode.ParentNode,NewNode);
        FormMain.DrawProject;
      end;
    end
    else
      ShowMessage('У проекта могут быть только дочерние узлы!');
  end;
end;
```

К вершине дерева разрешено добавлять только дочерние узлы.

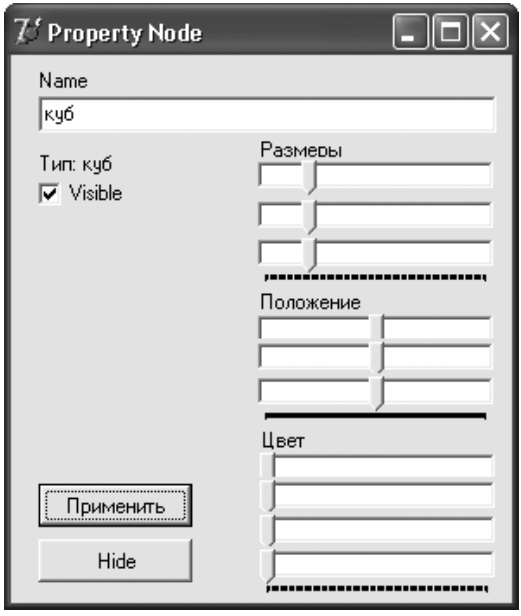


Рис. 13.21. Форма для изменения свойств узла

Вызов формы `FormProperty` (с внешним именем **Property Node**) (рис. 13.21) происходит при выборе пункта **Property** контекстного меню узла (листинг 13.38).

Листинг 13.38. Вызов формы для изменения свойств узла

```
procedure TFormTools.Property1Click(Sender: TObject);
begin
  NewNode:=PNode(TreeView1.Selected.Data);
  FormProperty.Show;
end;
```

Все 10 компонентов `TtrackBar` на представленной форме имеют разные значения свойства `tag` от `0` до `9` и вызывают один обработчик события (листинг 13.39).

Листинг 13.39. Изменение свойств объекта

```
procedure TFormProperty.TrackBar4Change(Sender: TObject);
var k: byte;
begin
  k:=(Sender as TTrackBar).tag;
  with NewNode^,Sender as TTrackBar do
  case k of
    0..2: // изменение размеров объектов
       Size[k]:=Position/100;
    3..5: // изменение положения объектов
       case typeNode of
         0,1: x0[k-3]:=Position/100;
         $FF: begin
                ligth_position0[k-3]:=Position/100;
                glLightfv(GL_LIGHT0, GL_POSITION,
                  @ligth_position0);
              end;
         $FE: begin
                dx:=x0[k-3]-Position/100;
                x0[k-3]:=Position/100;
                ChangePositionChild(NewNode,k-3,dx);
              end;
       end;
    6..9: // изменение цвета объектов и источника света
       if typeNode<>$FF then // цвет объектов
         mat_ambient[k-5]:=Position/100
       else begin            // цвет источника света
         ambient[k-5]:=Position/100;
         glLightfv(GL_LIGHT0, GL_DIFFUSE, @ambient);
       end;
```

```
  end;
  FormMain.DrawProject;
end;
```

Первые три компонента `TtrackBar` предназначены для изменения размеров объекта, следующие три — для изменения положения объекта, группы или источника света, последние четыре — для изменения цветов объекта или источника света, который приписан к вершине дерева, то есть к проекту.

Перемещение группы объектов на `dx` в `k`-ом направлении реализует рекурсивная процедура `ChangePositionChild` (листинг 13.40).

Листинг 13.40. Перемещение группы объектов

```
procedure ChangePositionChild(Node: PNode; k: byte; dx: real);

  procedure ChangePos(Node: PNode);
  var
    t: PNode;
  begin
    if Node=nil then Exit;
    t:=Node;
    repeat
      t.x0[k]:=t.x0[k]+dx;
      t:=t.Next;
    until t=nil;
    ChangePos(Node.Child);
  end;

begin
  if Node<>nil then ChangePos(Node.Child);
end;
```

Вызов процедуры рисования узлов `DrawNode` осуществляет процедура `DrawProject`, назначающая окно видимости и очищающая буфер рисования (листинг 13.41).

Листинг 13.41. Начало рисования

```
procedure TFormMain.DrawProject;
var i,L: integer;
begin
  glViewPort(((ClientWidth div 2)-(ClientHeight div 2))-Zoom,
    Zoom, ClientHeight+(2*Zoom), ClientHeight+(2*Zoom));
  glClear(GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT);
  if Head<>nil then DrawNode(Head);
  SwapBuffers(DC);
  glFlush();
end;
```

Рекурсивная процедура рисования узлов `DrawNode` в зависимости от типа узла `typeNode` вызывает процедуры рисования куба `MyDrawBox` или конуса `MyCylinder` (листинг 13.42).

Листинг 13.42. Рекурсивное рисование узлов

```
procedure DrawNode(Node: PNode);
begin
  if Node<>nil then
  repeat
    with Node^ do
    if Visible then begin
      case typeNode of
        0: MyDrawBox (Size[0],Size[1],Size[2], x0[0],x0[1],x0[2],
             GL_QUADS,@mat_ambient);
        1: MyCylinder(Size[0],Size[1],Size[2], x0[0],x0[1],x0[2],
             slices, stacks, GL_QUADS,@mat_ambient);
      end;
      DrawNode(Node.Child);
    end;
    Node:=Node.Next;
  until Node=nil;
end;
```

Тело "куб" на самом деле представляет собой параллелепипед, с размерами `Size[0],Size[1],Size[2]` и центром в точке `x0[0],x0[0],z[0]`. Поэтому при рисовании куба процедуру `DrawBox` из модуля DGLUT пришлось слегка изменить (листинг 13.43).

Листинг 13.43. Рисование куба

```
procedure MyDrawBox(Size0,Size1,Size2,x0,y0,z0: GLfloat;
  DrawType : GLenum; mat_ambient: pointer);
var
  V : array[0..7, 0..2] of GLfloat;
  I : GLint;
begin { MyDrawBox }
  V[0, 0] := -Size0/2+x0;
  V[1, 0] := -Size0/2+x0;
  …
  V[0, 1] := -Size1/2+y0;
  V[1, 1] := -Size1/2+y0;
  …
  V[0, 2] := -Size2/2+z0;
```

```
  V[3, 2] := -Size2/2+z0;
  …
  glLightModelfv(GL_LIGHT_MODEL_AMBIENT, mat_ambient);
  for I := 0 to 5 do   begin
    glBegin(DrawType);
      glNormal3fv(@BoxPoints[I, 0]);
      glVertex3fv(@V[BoxFaces[I, 0], 0]);
      glVertex3fv(@V[BoxFaces[I, 1], 0]);
      glVertex3fv(@V[BoxFaces[I, 2], 0]);
      glVertex3fv(@V[BoxFaces[I, 3], 0]);
    glEnd;
  end;
end; // MyDrawBox
```

Полностью проект "Редактор многогранников" приведен на компакт-диске в папке **Примеры | Глава 13 | Edit Body**.

ГЛАВА 14

Алгоритмы триангуляции поверхностей в трехмерном пространстве

В этой главе рассматриваются различные задачи триангуляции, реализация которых показана в проектах: "Триангуляция", "Триангуляция слоев", "Сглаживание триангуляции", "Изолинии".

Полный текст этих проектов представлен на прилагаемом к настоящей книге компакт-диске в папке **Примеры | Глава 14**.

14.1. Триангуляция поверхности

В проекте "Триангуляция" строится триангуляция треугольника, в котором содержится невыпуклый многосвязный многоугольник. Стороны треугольников триангуляции не должны пересекаться с ребрами внутреннего многоугольника.

Задача триангуляции поверхности по точкам — одна из самых распространенных задач, возникающих при проектировании геоинформационных систем (ГИС). Считается, что впервые триангуляцию применил Снеллиус в 1615 г. при строительстве каналов в Нидерландах. В настоящее время она используется в географии и конструировании (метод конечных элементов). Отсутствие единого масштаба и двух выделенных направлений, характерных для прямоугольной сетки, дает триангуляции преимущества при аппроксимации быстро меняющихся функций [12].

Триангуляцией называется разбиение поверхности на плоские треугольники в трехмерном пространстве. Будем рассматривать задачу построения триангуляции по заданному набору S трехмерных точек. Эта задача состоит в соединении заданных точек из набора S прямыми отрезками так, чтобы никакие отрезки не пересекались. Решение этой задачи неоднозначно, поэтому возникает проблема построения оптимальной триангуляции. Оптимальной называют такую триангуляцию, у которой сумма всех длин всех ребер минимальна, поэтому возникает необходимость создания методов триангуляции, дающих оптимальный результат.

Триангуляция для конечного набора точек S является задачей триангуляции выпуклой оболочки, охватывающей все точки набора S. Отрезки прямых линий при триангуляции не могут пересекаться — они могут только встречаться в общих точках, принадлежащих набору S. Поскольку отрезки прямых линий замыкают треугольники, мы будем считать их ребрами. На рис. 14.1 показаны два различных варианта триангуляции для одного и того же набора точек. Пока не будем обращать внимания на окружности, проведенные через некоторые точки на этих рисунках.

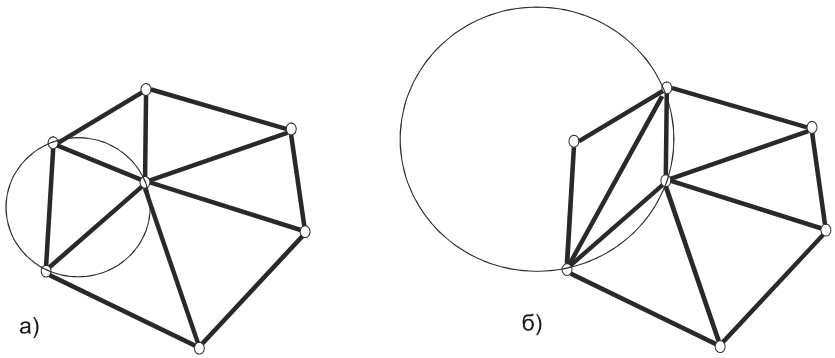


Рис. 14.1. Два различных варианта триангуляции для одного набора точек

Для данного набора точек S мы можем видеть, что все точки из этого набора могут быть подразделены на *граничные точки*, т. е. точки, которые лежат на границе выпуклой оболочки, и *внутренние точки*, лежащие внутри выпуклой оболочки. Также можно классифицировать и ребра, полученные в результате триангуляции S, как *ребра оболочки* и *внутренние ребра*. К ребрам оболочки относятся ребра, расположенные вдоль границы выпуклой оболочки, а к внутренним ребрам — все остальные ребра, образующие сеть треугольников внутри выпуклой оболочки. Отметим, что каждое ребро оболочки соединяет две соседние граничные точки, тогда как внутренние ребра могут соединять две точки любого типа. В частности, если внутреннее ребро соединяет две граничные точки, то оно является хордой выпуклой оболочки. Заметим также, что каждое ребро триангуляции является границей двух областей: каждое внутреннее ребро находится между двумя треугольниками, а каждое ребро оболочки — между треугольником и бесконечной плоскостью.

Любой набор точек, за исключением некоторых тривиальных случаев, допускает более одного способа триангуляции. Но при этом существует замечательное свойство: любой способ триангуляции для данного набора определяет одинаковое число треугольников. Данное свойство следует из широко известной теоремы о триангуляции набора точек. Рассмотрим эту теорему.

Предположим, что набор точек S содержит $n > 3$ точек и не все они лежат на одной прямой. Кроме того, m точек из них являются внутренними (т. е. лежащими

внутри выпуклой оболочки). Тогда при любом способе триангуляции набора S будет получено точно $n+m-2$ треугольников.

14.1.1. Алгоритмы триангуляции

Алгоритм построения триангуляции меняется в зависимости от целей, достигаемых при разбиении области на треугольники:

- *минимальная взвешенная триангуляция* — требует минимизировать суммарную длину ребер;
- *"жадная" триангуляция* — последовательно порождает ребра, не изменяя предыдущих разбиений. Вычисляются и сортируются линии, соединяющие все точки между собой, по возрастанию их длин. Ближайшие пары точек соединяются линией, если эта линия не пересекает полученные на предыдущих шагах линии. Процесс завершается, когда невозможно создать ни одной новой линии. В результате получается нерегулярное разбиение, в котором будет много остроугольных треугольников;
- *триангуляция Делоне* (ТД) — однозначно разбивает область на сеть максимально правильных треугольников, наиболее близких к равноугольным.

Наибольшей популярностью пользуется триангуляция Делоне, для которой существует три эквивалентных определения:

- первое определение: ТД — это множество треугольников с центрами в соседних *областях Вороного*. Областью Вороного узловой точки p называется множество точек, расстояние от которых до точки p меньше, чем до любой другой узловой точки;
- второе определение: триангуляция является ТД тогда и только тогда, когда в любой паре треугольников с общим ребром это ребро можно перебросить (сделать флип) и при этом минимальный угол из шести углов пары треугольников не увеличивается;
- третье определение (круговой критерий): триангуляция является ТД тогда и только тогда, когда внутри окружности, описанной вокруг любого треугольника, нет ни одной узловой точки. Этот метод позволяет получить треугольники, наиболее близкие к равноугольным. Так, например, триангуляцию, изображенную на рис. 14.1, *а*, можно отнести к типу триангуляции Делоне, а на рис. 14.1, *б* триангуляция содержит несколько сильно вытянутых треугольников и ее нельзя отнести к типу Делоне. На рис. 14.10 показан пример триангуляции Делоне для набора большого числа точек.

Любое из этих определений может быть основой для построения ТД. В частности, ТД может предшествовать разбиение на области Вороного. Как следует из определения, любая грань области Вороного является перпендикуляром к середине отрез-

ка, соединяющего две узловые точки (рис. 14.2), а треугольники ТД получаются после соединения центров соседних областей Вороного (рис. 14.3).

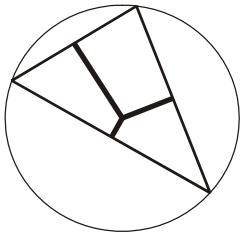


Рис. 14.2. Грани области Вороного для треугольника

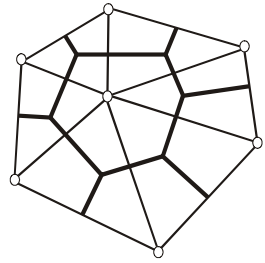


Рис. 14.3. Треугольники триангуляции Делоне

Построение ТД при помощи областей Вороного обладает двумя недостатками: во-первых, алгоритм неустойчив к погрешностям вычислений (способен обрабатывать только до 1000 точек); во-вторых, время счета квадратично зависит от числа точек. Поэтому был разработан инкрементный (итерационный) алгоритм ТД, основанный на втором и третьем определении.

Алгоритм инкрементной ТД основан на предположении, что имеется ТД из n узловых точек и добавляется новая $(n+1)$-я узловая точка:

- для новой точки определяется треугольник, в который она попадает;
- точка соединяется с вершинами этого треугольника;
- перебираются все треугольники против часовой стрелки, и проверяется для них выполнение кругового критерия. Если критерий не выполнен, то делается флип (рис. 14.4).

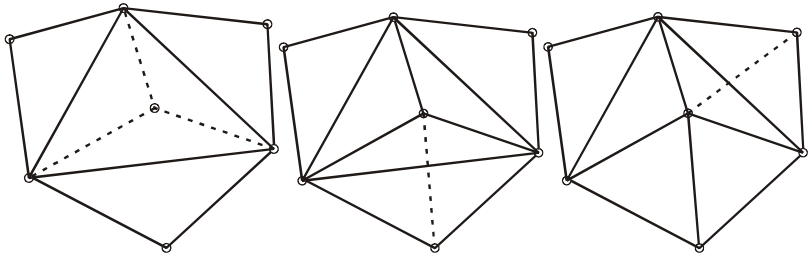


Рис. 14.4. Алгоритм инкрементной триангуляции Делоне

Для каждой новой точки число флипов невелико, но, в случае, если узловые точки лежат на одной окружности, может начаться лавина флипов по всей сетке старых узловых точек.

Для формирования триангуляции Делоне потребуется несколько новых определений. Набор точек считается *круговым*, если существует некоторая окружность, на которой лежат все точки набора. Такая окружность будет *описанной* для данного набора точек. Описанная окружность для треугольника проходит через все три его (не коллинеарные) вершины. Говорят, что окружность будет *свободной от точек* в отношении к заданному набору точек S, если внутри окружности нет ни одной точки из набора S. Но, однако, точки из набора S могут располагаться на самой свободной от точек окружности.

Триангуляция набора точек S будет триангуляцией Делоне, если описанная окружность для каждого треугольника будет свободна от точек. На схеме триангуляции (рис. 14.1, *а*), показаны две окружности, которые явно не содержат внутри себя других точек (можно провести окружности и для других треугольников, чтобы убедиться, что они также свободны от точек набора). Это правило не соблюдается на схеме (рис. 14.2) — внутрь проведенной окружности попала одна точка другого треугольника, следовательно, эта триангуляция не относится к типу Делоне.

Чтобы упростить алгоритм триангуляции, можно сделать два предположения относительно точек в наборе S. Во-первых, чтобы вообще существовала триангуляция, мы должны полагать, что набор S содержит, по крайней мере, три точки и они не коллинеарны. Во-вторых, для уникальности триангуляции Делоне необходимо, чтобы никакие четыре точки из набора S не лежали на одной описанной окружности. Легко видеть, что без такого предположения триангуляция Делоне не будет уникальной, ибо 4 точки на одной описанной окружности позволяют реализовать две различные триангуляции Делоне.

Наш алгоритм работает путем постоянного наращивания текущей триангуляции по одному треугольнику за один шаг. Вначале текущая триангуляция состоит из единственного ребра оболочки, по окончании работы алгоритма текущая триангуляция становится триангуляцией Делоне. На каждой итерации алгоритм ищет новый треугольник, который подключается к *границе* текущей триангуляции.

Определение границы зависит от следующей схемы классификации ребер триангуляции Делоне относительно текущей триангуляции. Каждое ребро может быть *спящим*, *живым* или *мертвым*:

- *спящие ребра*: ребро триангуляции Делоне является спящим, если оно еще не было обнаружено алгоритмом;
- *живые ребра*: ребро живое, если оно обнаружено, но известна только одна примыкающая к нему область;
- *мертвые ребра*: ребро считается мертвым, если оно обнаружено и известны обе примыкающие к нему области.

Вначале живым является единственное ребро, принадлежащее выпуклой оболочке — к нему примыкает неограниченная плоскость, а все остальные ребра — спящие. По мере работы алгоритма ребра из спящих становятся живыми, затем мертвыми. Граница на каждом этапе состоит из набора живых ребер.

На каждой итерации выбирается любое одно из ребер e границы, и оно подвергается обработке, заключающейся в поиске неизвестной области, которой принадлежит ребро e. Если эта область окажется треугольником f, определяемым концевыми точками ребра e и некоторой третьей вершиной v, то ребро e становится мертвым, поскольку теперь известны обе примыкающие к нему области. Каждое из двух других ребер треугольника t переводится в следующее состояние: из спящего в живое или из живого в мертвое. Здесь вершина v будет называться *сопряженной* с ребром e. В противном случае, если неизвестная область оказывается бесконечной плоскостью, то ребро e просто "умирает". В этом случае ребро e не имеет сопряженной вершины.

Триангуляция Делоне является универсальным способом триангуляции поверхностей, существует ряд реализаций этого способа (алгоритмы прямого построения, алгоритмы слияния, двухпроходные алгоритмы, итеративные алгоритмы). Однако могут возникнуть задачи, при решении которых нецелесообразна реализация триангуляции Делоне, т. к. можно построить менее трудоемкий в реализации алгоритм.

Данный раздел рассматривает один из способов построения триангуляции поверхности по заданному набору точек, полученных в результатах измерений. Кроме этого, в разделе приводится реализация алгоритма на языке Object Pascal и структуры данных, применимые к данной реализации.

Измерения проводятся для участка, на котором проводится бурение скважин (рис. 14.5). Скважины обозначены точками. Каждая точка имеет три координаты (X, Y и H — глубина). Таких слоев может быть несколько, по ним впоследствии строится разрез земной поверхности. Будем рассматривать триангуляцию только одного набора точек.

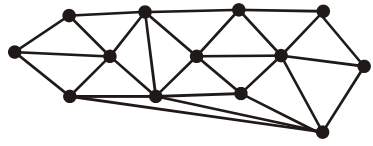


Рис. 14.5. Задача построения триангуляции по замерам

Особенностью задачи может служить то, что при построении триангуляции могут возникнуть "лишние" вытянутые треугольники. Это происходит из-за того, что крайние ряды скважин в полосе, в которой проводится бурение, в большинстве случаев расположены не на одной прямой (расстояния между скважинами — сотни метров, отклонение буровых установок от прямой может достигать нескольких метров из-за рельефа местности и т. п.).

Мы рассмотрим итерационный алгоритм построения триангуляции, в котором делается попытка получения треугольников, наиболее близких к равноугольным.

Триангуляция производится для двумерного случая, а третья координата учитывается лишь при построении поверхности.

Разберем данный алгоритм по шагам.

Шаг 1: находим две точки P_1 и P_2 такие, что все остальные точки лежат в одной полуплоскости относительно прямой P_1P_2.

Шаг 2: вводится фиктивная точка P_0, лежащая в другой полуплоскости относительно P_1P_2.

Шаг 3: помещаем в стек точки P_1, P_2, P_0.

Шаг 4: вынимаем из стека три точки и присваиваем им имена P_1, P_2, P_3.

Шаг 5: ищется точка M, такая, чтобы $\angle P_1MP_2$ был максимальным. Если такой точки нет, то выполняется переход к шагу 7.

Шаг 6: сравниваем ΔP_1MP_2 с найденными ранее. Если совпадений не обнаружено, запоминаем найденный треугольник и помещаем в стек точки P_1, M, P_2, и P_2, M, P_1.

Шаг 7: если стек не пуст, то возвращаемся к шагу 4.

14.1.2. Структура данных

Для реализации алгоритма потребуется стек, основанный на динамическом массиве, в который необходимо помещать номера трех точек. Поэтому для описания стека введем следующую структуру (листинг 14.1).

Листинг 14.1. Структура стека

```
type
  TStack=record
    q1,q2,q3: word;
  end;
  TAStack=array of TStack;
var
  Stack: TAStack;
```

Для добавления данных в стек и извлечения данных из стека предназначены процедуры `PushStack` и `PopStack` (листинг 14.2).

Листинг 14.2. Процедуры для работы со стеком

```
procedure InitStack(var s: TAStack);
// инициализация стека
```

```
begin
  SetLength(s,0);
end;

function StackIsEmpty(s: TAStack): Boolean;
// проверка на пустоту
begin
  Result:=Length(s)=0;
end;

procedure PopStack(var s: TAStack; var p1,p2,p3: integer);
// взять из стека
var L: integer;
begin
  L:=Length(s);
  p1:=S[L-1].q1; p2:=S[L-1].q2; p3:=S[L-1].q3;
  L:=L-1; SetLength(s,L);
end;

procedure PushStack(var s: TAStack; p1,p2,p3: word);
// положить в стек
var L: integer;
begin
  L:=Length(s); L:=L+1; SetLength(s,L);
  S[L-1].q1:=p1;
  S[L-1].q2:=p2;
  S[L-1].q3:=p3;
end;
```

Для описания точек введем динамический массив `Pt: TAXYZ=array of TXYZ`, где `TXYZ` определяется следующей структурой (листинг 14.3).

Листинг 14.3. Структура точки *TXYZ*

```
  TXYZ=record
    x,y,z: single;
  end;
  TAXYZ=array of TXYZ;
```

Введем структуру для описания массива треугольников (листинг 14.4).

Листинг 14.4. Структура треугольников

```
  TTr=record
    BColor: integer;
    Norm: TXYZ;
```

```
    P: array[0..2] of word;
  end;
  TATr=array of TTr;
```

В этой структуре для построения триангуляции потребуются только номера точек, на которые опирается треугольник `P: array[0..2] of word`.

14.1.3. Реализация алгоритма

Шаг 1: найти две точки P_1 и P_2, такие, чтобы все остальные точки лежали в одной полуплоскости относительно прямой P_1P_2.

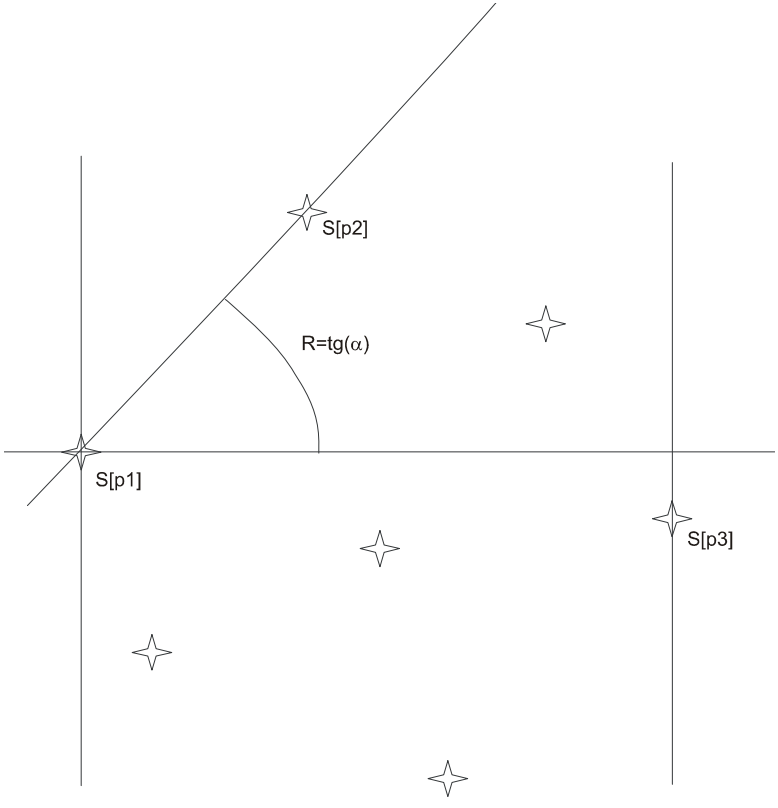


Рис. 14.6. Определение точек с номерами P_1, P_2 и P_3

Сначала определим номера точек P_1 и P_3, для которых координата x будет минимальной и максимальной, соответственно (рис. 14.6). На рисунке использованы следующие обозначения: отдельные точки представлены звездочками, $S[p_1]$, $S[p_2]$, $S[p_3]$ — точки с номерами P_1, P_2 и P_3, R — тангенс угла наклона прямой,

проходящей через точки P_1 и P_2. Код, соответствующий определению номеров точек P_1 и P_3, представлен в листинге 14.5.

Листинг 14.5. Поиск номеров точек с *MinX* и *MaxX*

```
// Найти m1 и m3: min=x[m1]<=x[i]<=max=x[m3]
i:=1; MinX:=Point[0].x; MaxX:=Point[0].x;
m1:=0; m3:=0;
repeat
  if Point[i].x<=MinX then begin
    MinX:=Point[i].x; m1:=i;
  end
  else
    if Point[i].x>MaxX then begin
      MaxX:=Point[i].x; m3:=i;
    end;
  Inc(i);
until i>n;
```

Затем найдем точку с номером P_2, для которой тангенс угла наклона относительно точки P_1 будет максимальным (рис. 14.6 и листинг 14.6).

Листинг 14.6. Нахождение точек *P*1[*m*1] и *P*2[*m*2]

```
// ШАГ 1 Находим такие две точки P1[m1] и P2[m2], чтобы все
// остальные точки лежали по одну сторону от прямой P1P2
// с максимальным tg(a)
R:=-1e24; i:=-1; m2:=m3; Ok:=false;
while i<=n do begin
  Inc(i);
  d:=DistToLine(Point[i].x,Point[i].y,
                Point[m1].x,Point[m1].y,
                Point[m3].x,Point[m3].y);
    if (i<>m1) and (i<>m3) and (Point[i].x<>Point[m1].x) and
      not IsPointOfLine(m1,i) and (Abs(d)>Eps) then begin
      t:=(Point[i].y-Point[m1].y)/(Point[i].x-Point[m1].x);
      if t>=R then begin
        m2:=i; R:=t;
      end;
    end;
  end;
```

Ясно, что все точки будут лежать ниже прямой P_1P_2, т. е. шаг 1 выполнен.

Шаг 2: ввести фиктивную точку с номером `n+1`, лежащую в другой полуплоскости относительно прямой `m1`, `m2`. Будем считать, что точка является четвертой верши-

ной параллелограмма, остальные вершины которого — `m1`, `m2`, `m3` (листинг 14.7, рис. 14.7).

Листинг 14.7. Ввод фиктивной точки *P*0[*n*+1]

```
// ШАГ 2 Вводим фиктивную точку P0[n+1],
// лежащую по другую сторону от прямой P1P2
Point[n+1].x:=Point[m1].x+Point[m2].x-Point[m3].x;
Point[n+1].y:=Point[m1].y+Point[m2].y-Point[m3].y;
```

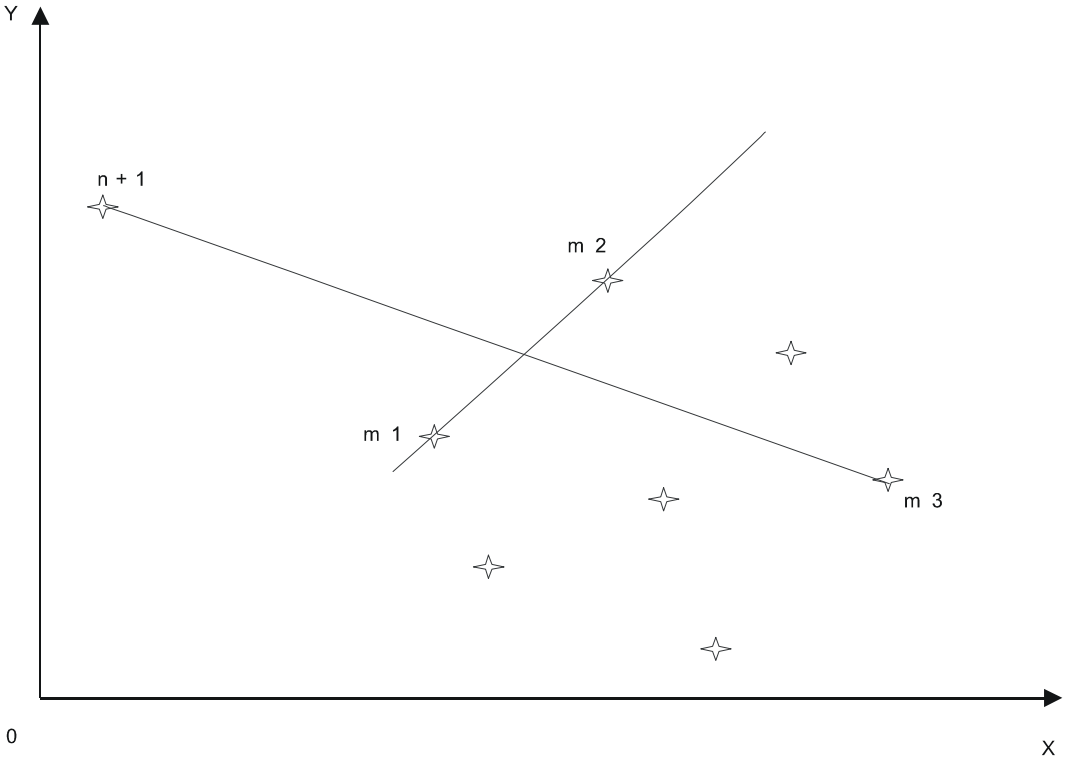


Рис. 14.7. Введение фиктивной точки *n*+1

Шаг 3: поместить в стек номера точек P_1, P_2 и `n+1` (листинг 14.8).

Листинг 14.8. Помещение в стек номеров точек P_1, P_2 и *n*+1

```
InitStack(Stack);              //  инициализация стека
// ШАГ 3 Помещаем в стек номера точек P1, P2, P0
PushStack(Stack, m1,m2,n+1);  //  положить в стек
```

Шаг 4: вынимаем из стека три точки и присваиваем им имена P_1, P_2, P_3 (листинг 14.9).

Листинг 14.9. Извлечение из стека трех точек

```
PopStack(Stack, p1,p2,p3);      //  взять из стека
```

Шаг 5: находится такая точка M, отделенная от точки P_3 прямой P_1P_2, чтобы $\angle P_1MP_2$ был максимальным (листинг 14.10). Если такой точки нет, то выполняется переход к шагу 7.

Листинг 14.10. Поиск точки *M*, отделенной от точки *P*3 прямой *P*1*P*2

```
// Ищем такую точку M, отделенную от точки P3 прямой P1P2,
// чтобы угол P1-M-P2 был максимальный;
// если такой точки нет, то переход на ШАГ 6
m:=-1; R1:=1e24; i:=0;
repeat
  if (i<>p1) and (i<>p2) and (i<>p3) then begin
    Ok:=not IsPointsOnOneSide(Point[p3].x,Point[p3].y,
                              Point[ i].x,Point[ i].y,
                              Point[p1].x,Point[p1].y,
                              Point[p2].x,Point[p2].y);
    if Ok then begin
      d1:=Sqrt(Sqr(Point[i].x-Point[p1].x)+
               Sqr(Point[i].y-Point[p1].y));
      d2:=Sqrt(Sqr(Point[i].x-Point[p2].x)+
               Sqr(Point[i].y-Point[p2].y));
      R:=((Point[p1].x-Point[i].x)*(Point[p2].x-Point[i].x)+
          (Point[p1].y-Point[i].y)*(Point[p2].y-Point[i].y))/
          d1/d2;
      if (R<R1) and
         (Abs(R+1)>Eps) and
         not TestEdges(Point[p1],Point[p2],Point[i]) and
         not TestEdgesTr(Point[p1],Point[p2],Point[i])
      then begin
        R1:=R; m:=i;
      end;
    end;
  end;
  Inc(i);
 until i>n;
```

Функцию `IsPointsOnOneSide`, проверяющую, лежат ли точки с одной стороны прямой P_1P_2, и функцию `Distance`, вычисляющую расстояние между точками, мы рассмотрим далее (листинги 14.13 и 14.14).

Шаг 6: сравниваем треугольник P_1MP_2 с найденными ранее треугольниками. Если совпадений не обнаружено, запоминаем найденный треугольник и помещаем в стек точки P_1, M, P_2 и P_2, M, P_1 (листинг 14.11).

Листинг 14.11. Сравнение треугольника *P*1*MP*2 с треугольниками, найденными ранее

```
// ШАГ 5 Сравниваем треугольник P1-M-P2 с найденными ранее;
//если совпадения не обнаружено, то запоминаем треугольник P1MP2
// и помещаем в стек номера точек P1, M, P2 и P2, M, P1
if m<>-1 then begin
  i:=-1; Ok:=false; L:=Length(Tr0);
  while (i<L-1) and not Ok do begin
    Inc(i);
    with Tr0[i] do Ok:=Eqv(p1,p2,m,P[0],P[1],P[2]);
  end;
  if not Ok then begin
    L:=L+1; SetLength(Tr0,L);
    with Tr0[L-1] do begin
      P[0]:=p1; P[1]:=p2; P[2]:=m;
    end;
    PushStack(Stack, p1,m,p2); //  положить в стек
    PushStack(Stack, p2,m,p1); //  положить в стек
  end;
end;
```

Запоминание найденного треугольника происходит только в том случае, если этот треугольник не был найден ранее. Проверку равенства треугольников реализует функция `Eqv` (листинг 14.12).

Листинг 14.12. Проверка совпадения треугольников

```
  function Eqv(p1,p2,p3,q1,q2,q3: integer): boolean;
  begin
    Result:=((p1=q1) and (p2=q2) and (p3=q3)) or
            ((p1=q2) and (p2=q3) and (p3=q1)) or
            ((p1=q3) and (p2=q1) and (p3=q2)) or
            ((p1=q1) and (p2=q3) and (p3=q2)) or
            ((p1=q3) and (p2=q2) and (p3=q1)) or
            ((p1=q2) and (p2=q1) and (p3=q3));
  end;
```

На следующем шаге итерационного процесса точки P_2, M, P_1, попавшие в стек последними, будут извлечены из стека и использованы для построения следующего треугольника (рис. 14.8).

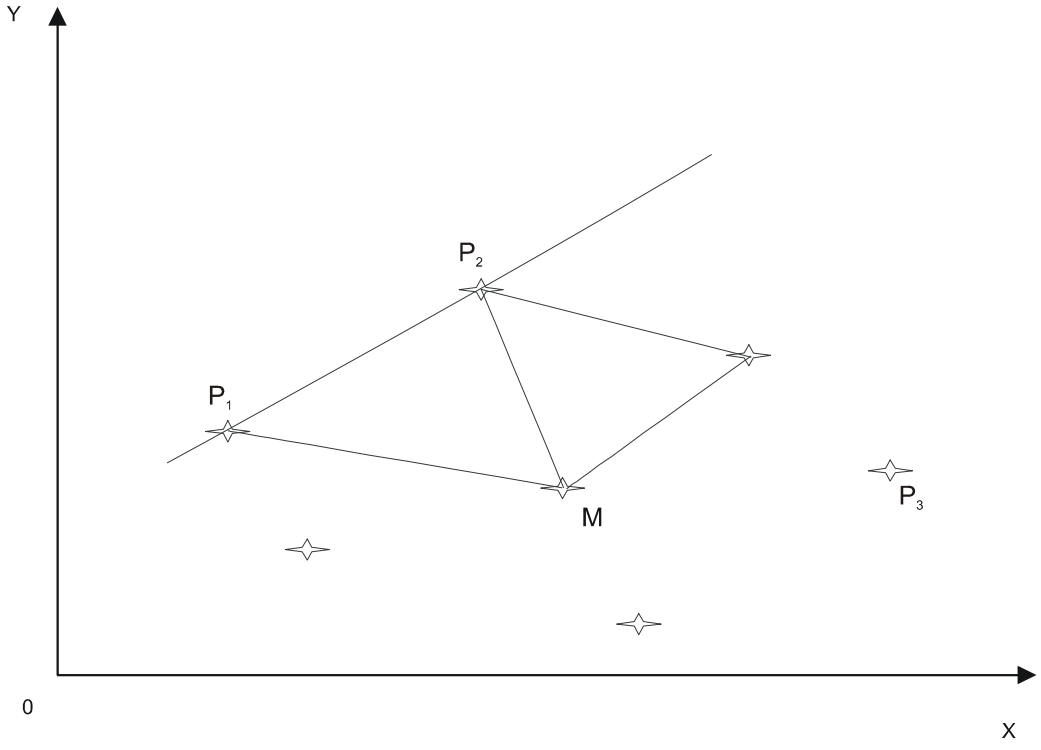


Рис. 14.8. Построение следующего треугольника

Функция, вычисляющая расстояние между точками, проста (листинг 14.13).

Листинг 14.13. Расстояние между точками

```
function Distance(n1,n2: word): real;
begin
  Result:=Sqrt(Sqr(S[n1].Point.x-S[n2].Point.x)+
               Sqr(S[n1].Point.y-S[n2].Point.y));
end;
```

Для проверки условия, лежат ли точки с одной стороны прямой, используется функция `IsPointsOnOneSide` (листинг 14.14).

Листинг 14.14. Проверка условия, лежат ли точки с одной стороны прямой

```
  function IsPointsOnOneSide(
       x1  : Double;
       y1  : Double;
       x2  : Double;
       y2  : Double;
```

```
      xL1 : Double;
      yL1 : Double;
      xL2 : Double;
      yL2 : Double): Boolean;
var A,d1,d2: Double;
begin
  d1:=DistToLine(x1,y1,xL1,yL1,xL2,yL2);
  d2:=DistToLine(x2,y2,xL1,yL1,xL2,yL2);
  if (Abs(d2)<Eps) or (Abs(d1)<Eps) then begin
    A:=0; d:=A; Result:=false;
  end;
  Result:=(Abs(d1)<Eps) or (Abs(d2)<Eps) or (d1*d2>0);
end;
```

Результаты работы алгоритма представлены на рис. 14.9.

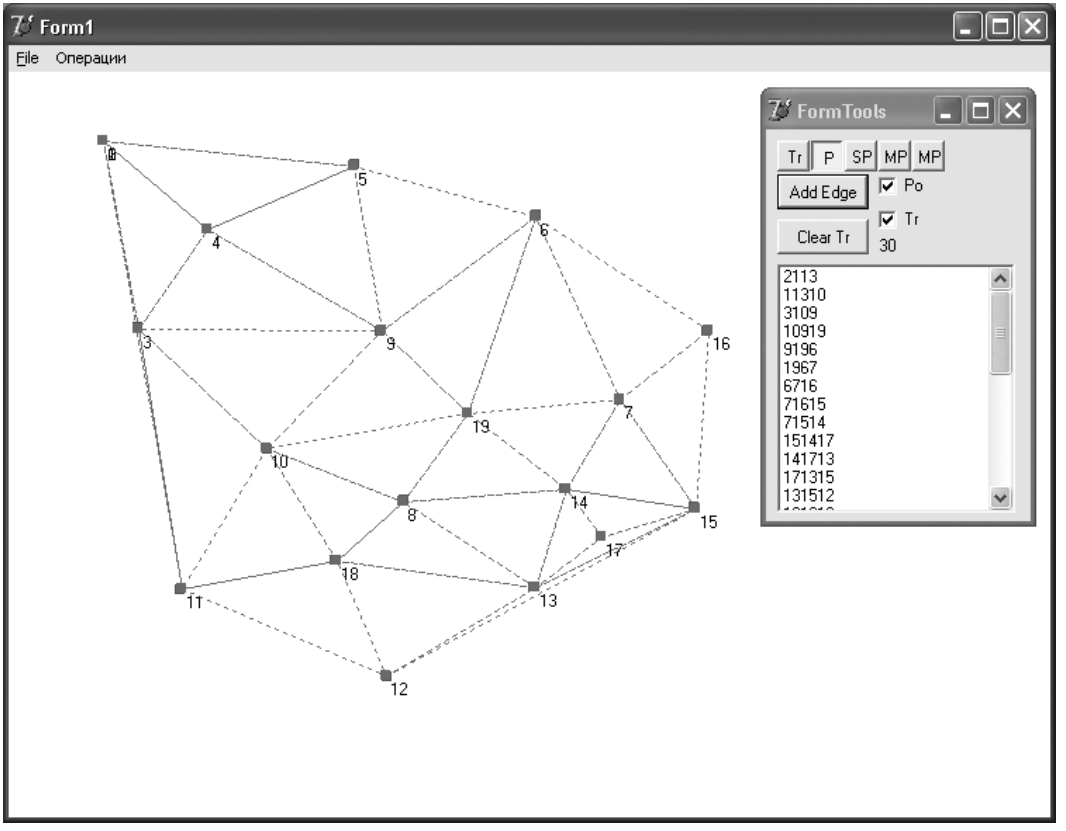


Рис. 14.9. Построение триангуляции по точкам

14.1.4. Удаление "лишних" треугольников

Алгоритм, рассмотренный в предыдущем разделе, строит множество треугольников, которые образуют выпуклую область. Некоторые из этих треугольников слишком вытянуты. Вытянутый треугольник (skinny triangle) — треугольник, имеющий одну сторону, намного отличающуюся от других двух. То есть треугольник слишком вытянут вдоль некоторой прямой. В качестве критерия для удаления треугольника выберем ограничение для всех косинусов углов треугольника `(cos1>cosMax) and (cos2>cosMax) and (cos3>cosMax)`, где cosMax = –0,3. Если это условие не выполняется, то треугольник удаляется из массива (листинг 14.15).

Листинг 14.15. Удаление вытянутых треугольников

```
procedure DeleteTr(Eps: real; var Tr: TATr; var PointH: TAXYZNorma);
const cosMax=-0.3;
var i,j: integer;
    cos1, cos2, cos3: single;
    Ok: boolean;
var a,b,c: single;
begin
  L:=Length(Tr); i:=-1;
  while (i<L-1) do begin
    Inc(i);
    with Tr[i] do begin
      a:=Distance(P[1],P[2]);
      b:=Distance(P[0],P[2]);
      c:=Distance(P[0],P[1]);
      cos1:=(b*b+c*c-a*a)/(2*b*c);
      cos2:=(a*a+c*c-b*b)/(2*a*c);
      cos3:=(b*b+a*a-c*c)/(2*b*a);
      Ok:=(cos1>cosMax) and (cos2>cosMax) and (cos3>cosMax);
      if not Ok then begin
        L:=Length(Tr);
        for j:=i+1 to L-1 do Tr[j-1]:=Tr[j];
        Dec(L); SetLength(Tr,L);
        Dec(i);
      end;
    end;
  end;
end;
```

На рис. 14.10 приведен пример работы алгоритма с удалением слишком вытянутых треугольников.

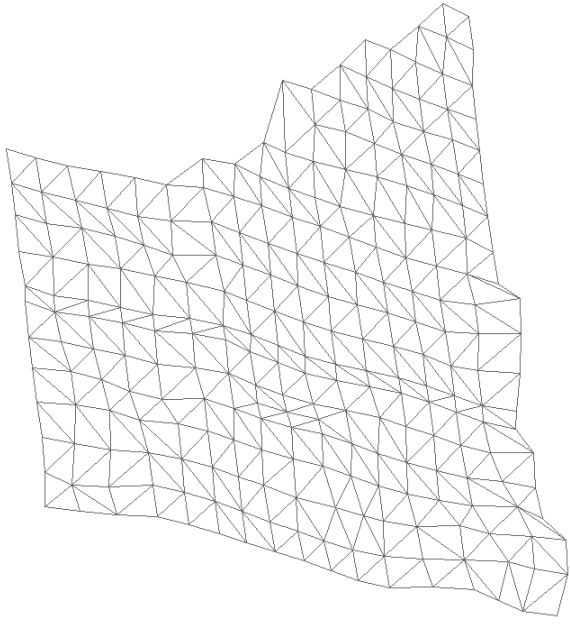


Рис. 14.10. Удаление слишком вытянутых треугольников

Полный текст данного проекта приведен на компакт-диске в папке **Примеры | Глава 14 | Триангуляция**.

14.2. Триангуляция всех слоев участка

Триангуляция вершин скважин, описанная в *разд. 14.1*, может быть использована для триангуляции слоев породы (проект "Триангуляция слоев"). Каждая порода обладает возрастом (стратиграфией) и составом (литологией). При бурении скважин получают данные в виде интервалов пород `From..To`. Для каждого интервала известно: начало интервала (`From`); конец интервала (`To`); возраст, точнее указатель `NodeStat` на структуру, которая описывает возраст (название, цвет, индекс литологии (`LCod`) и др.); состав, точнее указатель `NodeLith` на структуру, которая описывает состав (название, векторный шаблон изображения, `LCod` и др.).

Задача построения триангуляции верхней и нижней поверхности слоя осложняется тем, что не на всех скважинах есть все интервалы пород. В природе редко встречается случай, когда толщина слоя меняется резко. Поэтому необходимо свести толщину слоя в ноль на "соседних" скважинах. Фактически на этих скважинах необходимо ввести фиктивный интервал, для которого начало и конец совпадают. На рис. 14.11 показан плоский разрез с линзами слоев.

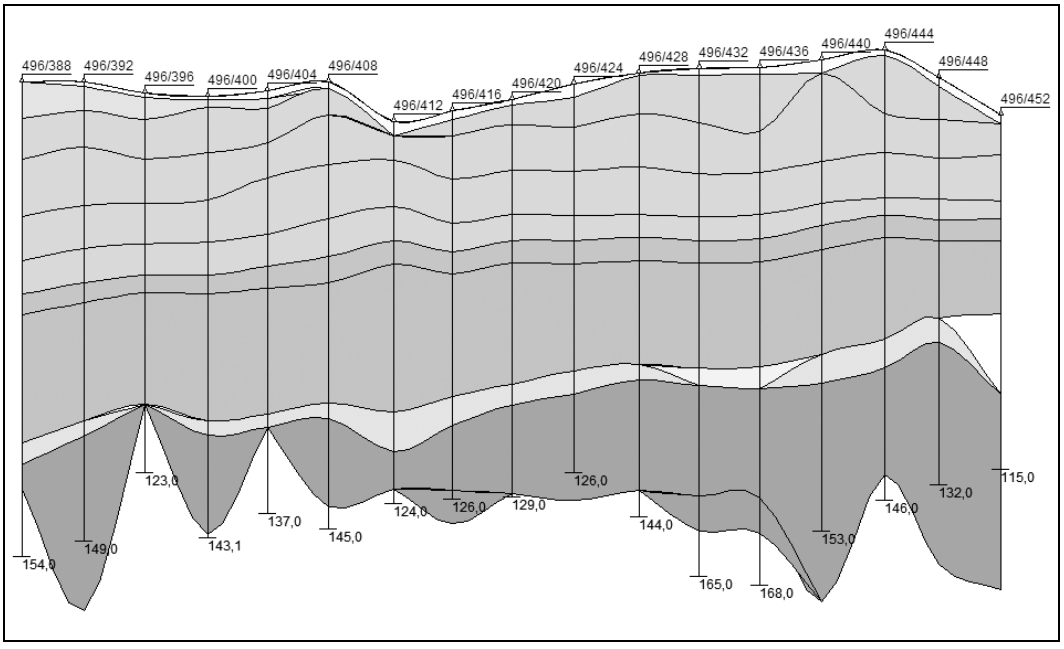


Рис. 14.11. Плоский разрез с линзами слоев

При сведении толщины слоя в ноль возникает две проблемы:

- определение "соседних" скважин;
- определение глубины залегания фиктивного слоя.

Первая проблема — определение "соседних" скважин, решается достаточно просто, если построена базовая триангуляция скважин, а вторая проблема — определение глубины залегания фиктивной точки слоя, решается только в случае упорядоченности слоев, например, по `Lcod`-ам стратиграфии.

14.2.1. Структура данных

Для каждого проекта определяется набор стратиграфий со следующей структурой (листинг 14.16).

Листинг 14.16. Структура стратиграфии

```
TSprStrat=record
  Node      : PNode;
  TrFrom    : TATr;       // треугольники триангуляции скважин
  PointFrom : TAXYZNorma; // точки триангуляции скважин
  TrTo      : TATr;       // треугольники триангуляции скважин
```

```
    PointTo   : TAXYZNorma; // точки триангуляции скважин
  end;
  TASprStrat=array of TSprStrat;
```

Поле `Node` указывает на структуру, которая описывает возраст (название, цвет, `LCod` и др.). В частности, этот набор для одного из участков может состоять из 12 стратиграфических слоев (рис. 14.12).

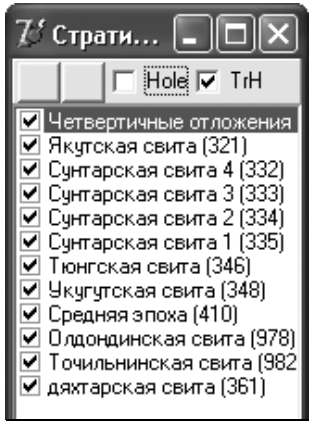


Рис. 14.12. Набор стратиграфий для одного из участков

Необходимо построить два массива треугольников `TrFrom` и `TrTo`, ограничивающих каждый стратиграфический слой сверху и снизу и опирающихся на точки `PointFrom` и `PointTo`. Структура треугольников `TTr` и `TAXYZ` описана в листинге 14.4. Для нашей задачи необходима несколько другая структура для описания точки (листинг 14.17).

Листинг 14.17. Структура точки

```
TXYZNorma=record
  isInterval: byte;
  // 0-есть интервал; 1-граничная точка; 2-нет интервала
  Point     : TXYZ;
  Norma     : TXYZ;
  NumHole   : word; // номер скважины
end;
```

В этой структуре поле `Point` описывает координаты `(x,y,z)`, поле `Norma` необходимо для вычисления псевдонормали в точке и будет использоваться в OpenGL для плавной закраски поверхности, поле `NumHole` содержит номер скважины, на которой находится эта точка.

И, наконец, вычисляемое поле `isInterval`, принимающее значения от `0` до `2`, показывает на характер точки: `0` — внутренняя точка, т. е. на соответствующей скважи-

не есть стратиграфический интервал на этой скважине; 1 — граничная точка, т. е. на соответствующей скважине интервала нет, но точка входит в треугольник, в котором хотя бы одна точка содержит интервал; 2 — на скважине нет интервала, и точка является внешней, т. е. и не внутренней и не граничной.

Будем использовать поле isInterval для вычисления поля NumHoleWithInterval каждого треугольника (листинг 14.18), показывающего число внутренних точек. Это поле позволит просто отфильтровать ненужные треугольники по условию NumHoleWithInterval=0.

Листинг 14.18. Структура для описания треугольников

```
TTr=record
  NumHoleWithInterval : byte;
  //3-внутренний; 2-внутренние; 1-внутренняя; 0-внутренних точек
  Norm: TXYZ;
  P: array[0..2] of word;
end;
```

14.2.2. Алгоритм построения триангуляции слоев

Алгоритм построения верхней триангуляции TrFrom и нижней триангуляции TrTo для каждого стратиграфического слоя состоит из следующих шагов.

Шаг 1: скопировать все точки базовой триангуляции PointH в точки верхней триангуляции PointTo и нижней триангуляции PointFrom слоя.

Шаг 2: для каждой точки верхней триангуляции PointTo и нижней триангуляции PointFrom вычисляем признак существования интервала (0 или 2) и новое значение координаты z.

Шаг 3: скопировать все треугольники базовой триангуляции TrH в треугольники верхней триангуляции TrTo и нижней триангуляции TrFrom слоя.

Шаг 4: для каждого треугольника верхней триангуляции TrTo и нижней триангуляции TrFrom вычислить число внутренних точек NumHoleWithInterval.

Шаг 5: удалить все треугольники, для которых число внутренних точек NumHoleWithInterval равно 0.

Код процедуры, реализующей этот алгоритм, приведен в листинге 14.19.

Листинг 14.19. Алгоритм построения триангуляции слоев

```
Lp:=Length(PointH); L:=Length(SprStrat);
for i:=0 to L-1 do
with SprStrat[i] do begin
  Lt:=Length(TrH);
  SetLength(TrFrom,Lt); SetLength(PointFrom,Lp);
```

```
  SetLength(TrTo,Lt); SetLength(PointTo,Lp);
  for j:=0 to Lp-1 do begin
    PointTo[j]:=PointH[j]; PointFrom[j]:=PointH[j];
    PointTo[j].isInterval:=
      SetIsInterval(2,Node,PointTo[j].NumHole,
         PointTo[j].Point.z);
    PointFrom[j].isInterval:=
       SetIsInterval(1,Node,PointTo[j].NumHole,
         PointFrom[j].Point.z);
  end;
  for j:=0 to Lt-1 do begin
    TrTo[j]:=TrH[j];
    TrTo[j].NumHoleWithInterval:=
      SetNumHoleWithInterval(TrTo[j],PointTo);
    TrFrom[j]:=TrH[j];
    TrFrom[j].NumHoleWithInterval:=
      SetNumHoleWithInterval(TrFrom[j],PointFrom);
  end;
  j:=0; Lt:=Length(TrH);
  while j<=Lt-1 do begin
    if TrTo[j].NumHoleWithInterval=0 then begin
      for k:=j+1 to Lt-1 do begin
        TrTo[k-1]:=TrTo[k];
        TrFrom[k-1]:=TrFrom[k];
      end;
      Dec(Lt);
      SetLength(TrTo,Lt); SetLength(TrFrom,Lt);
    end
    else Inc(j);
  end;
end;
```

Особого обсуждения заслуживает шаг 2, в котором для каждой точки верхней триангуляции `PointTo` и нижней триангуляции `PointFrom` вычисляется признак существования `isInterval` (`0` или `2`) стратиграфического интервала с указателем `Node` на скважине `NumHole` и новое значение координаты `z`.

На скважине может быть несколько (идущих подряд) стратиграфических интервалов, поэтому поиск интервала для верхней триангуляции должен начинаться сверху и закончиться нахождением верхней границы всех стратиграфических слоев `Node`, поиск интервала для нижней триангуляции должен начинаться снизу и закончиться нахождением нижней границы всех стратиграфических слоев `Node`.

Если интервал `Node` не найден, то будем искать интервал, у которого `LCod` меньше, чем `Node.LCod`.

Итак, алгоритм вычисления признака существования `isInterval` стратиграфического интервала и новое значение координаты `z` точки состоит из следующих шагов.

Шаг 1: если ищется верхняя граница слоя, то выполнить шаг 2, иначе выполнить шаг 3.

Шаг 2: начиная с верхнего интервала, найти интервал, для которого `Interval_[i].NodeStrat=Node`. Перейти на шаг 4.

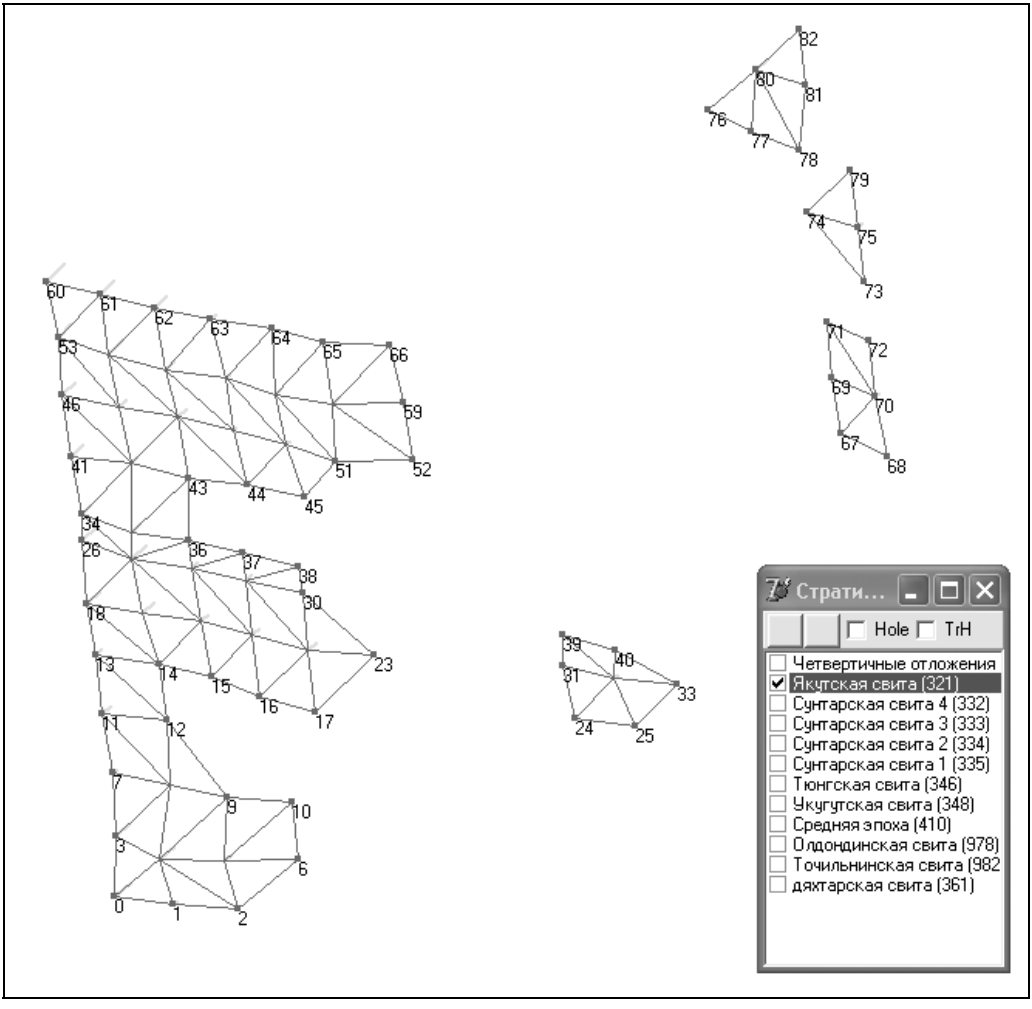


Рис. 14.13. Пример работы алгоритма для слоя "Якутская свита"

Шаг 3: начиная с последнего интервала найти интервал, для которого `Interval_[i].NodeStrat=Node`.

Шаг 4: если интервал найден, то функция вернет значение `0` и при поиске верхней границы новое значение `z` будет равно `Hole[NumHole].Elevation-Interval_[i].From`, а при поиске нижней границы `z` будет равно `Hole[NumHole].Elevation-Interval_[i].To`.

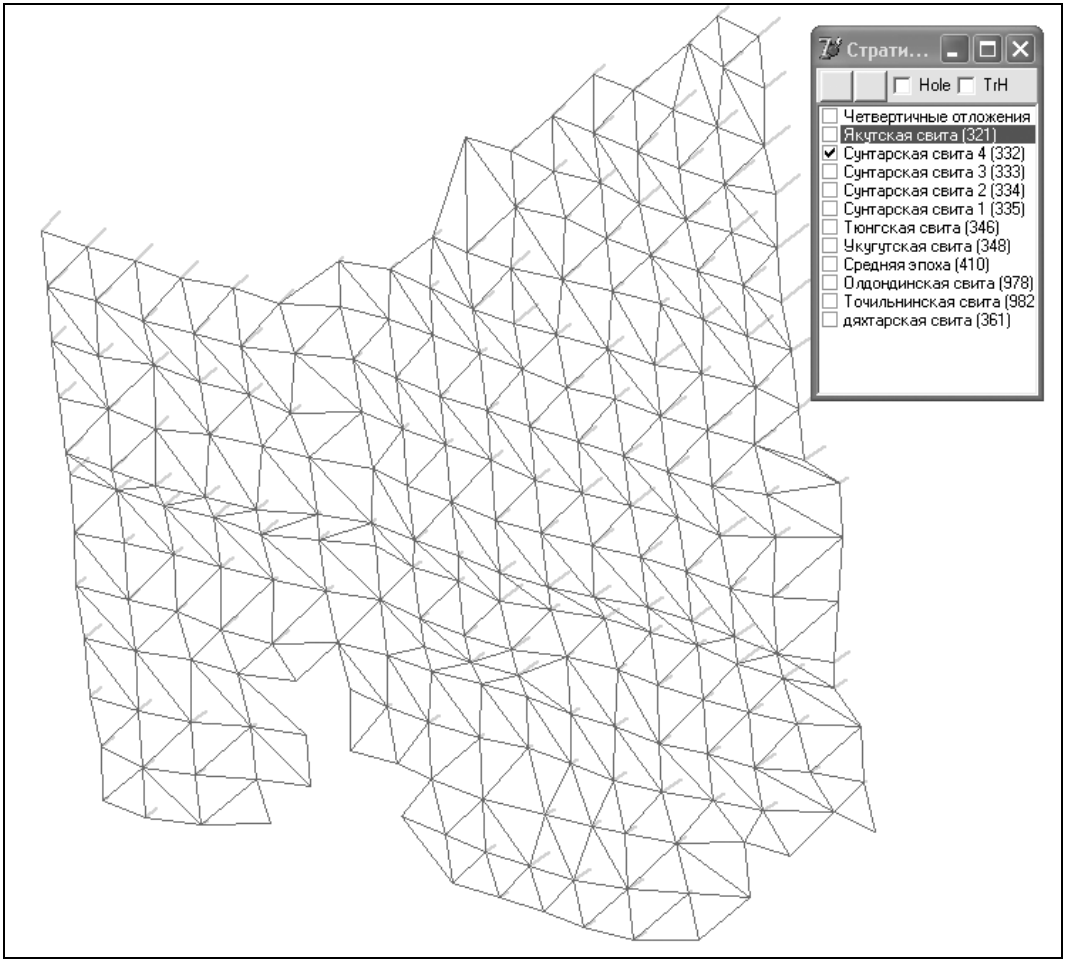


Рис. 14.14. Пример работы алгоритма для слоя "Сунтарская свита 4"

Шаг 5: если интервал не найден, то попытаемся, начиная поиск с последнего интервала, найти интервал, для которого будет выполняться условие `Interval_[i].NodeStrat.LCod<Node.LCod`. Для этого условия существенным является предположение об упорядоченности слоев по `LCod`-ам стратиграфии. Если интервал с номером `i` найден, то новое значение `z` будет равно `Hole[NumHole].Elevation-Interval_[i].To`, иначе новое значение будет равно координате вершины скважины над уровнем моря `Hole[NumHole].Elevation`.

Полный текст функции для этого алгоритма приведен в листинге 14.20.

Листинг 14.20. Вычисление признака существования интервала и нового значения координаты *Z*

```
function SetIsInterval(fl: byte; Node: PNode; NumHole: word; var Value: sin-
gle): byte;
var L,i: integer;
    Ok: boolean;
begin
  Result:=2;
  with PrGIS.Hole[NumHole].LayerHole[0] do begin
    L:=Length(Interval_); Ok:=false;
    case fl of
      1: begin
           i:=-1;
           while (i<L-1) and not Ok do begin
             Inc(i);
             Ok:=Interval_[i].NodeStrat=Node;
           end;
         end;
      2: begin
           i:=L;
           while (i>0) and not Ok do begin
             Dec(i);
             Ok:=Interval_[i].NodeStrat=Node;
           end;
         end;
    end; // case
    if Ok then begin
      Result:=0;
      if fl=1 then
        Value:=PrGIS.Hole[NumHole].Elevation-Interval_[i].From
      else
        Value:=PrGIS.Hole[NumHole].Elevation-Interval_[i].To_;
    end
    else begin
      i:=L;
      while (i>0) and not Ok do begin
        Dec(i);
        Ok:=Interval_[i].NodeStrat.LCod<Node.LCod;
      end;
      if Ok then Value:=PrGIS.Hole[NumHole].Elevation-
        Interval_[i].To_
```

```
      else Value:=PrGIS.Hole[NumHole].Elevation;
    end;
  end;
end;
```

Результаты работы алгоритмы для слоев "Якутская свита" и "Сунтарская свита 4" приведены на рис. 14.13 и 14.14.

Полный текст данного проекта приведен на компакт-диске в папке **Примеры | Глава 14 | Автопостроение слоев**.

14.3. Сглаживание триангуляции

Скважины бурят достаточно далеко друг от друга. Линии скважин находятся на расстоянии 400—500 метров друг от друга, расстояние между отдельными скважинами, принадлежащими каждой из этих линий, примерно такое же. Поэтому триангуляция поверхностей, построенная на скважинах, образует редкую сетку треугольников (рис. 14.15).

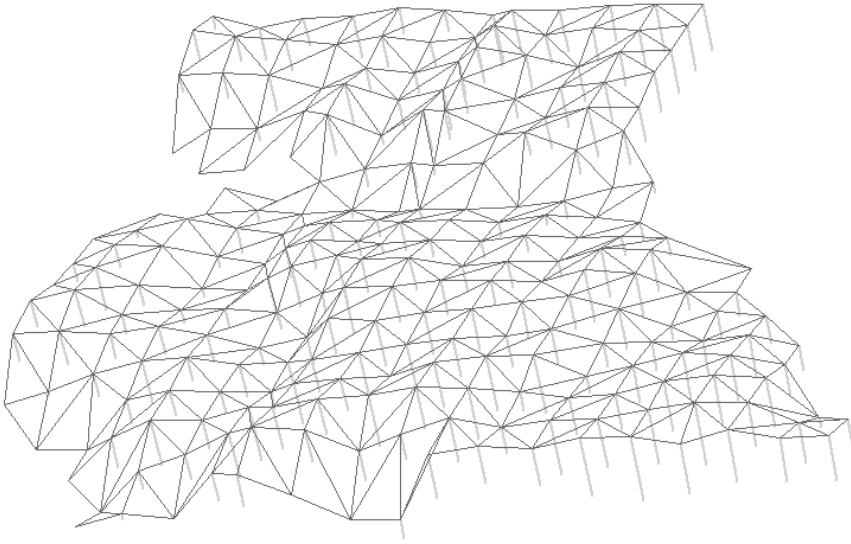


Рис. 14.15. Триангуляция, построенная по скважинам

Поэтому возникает задача введения дополнительного разбиения каждого треугольника на несколько новых треугольников. Естественно, новые узловые точки не должны лежать в плоскости треугольника, а должны сглаживать поверхность (рис. 14.16). В данном разделе рассматривается проект "Сглаживание триангуляции", в котором решается такая задача.

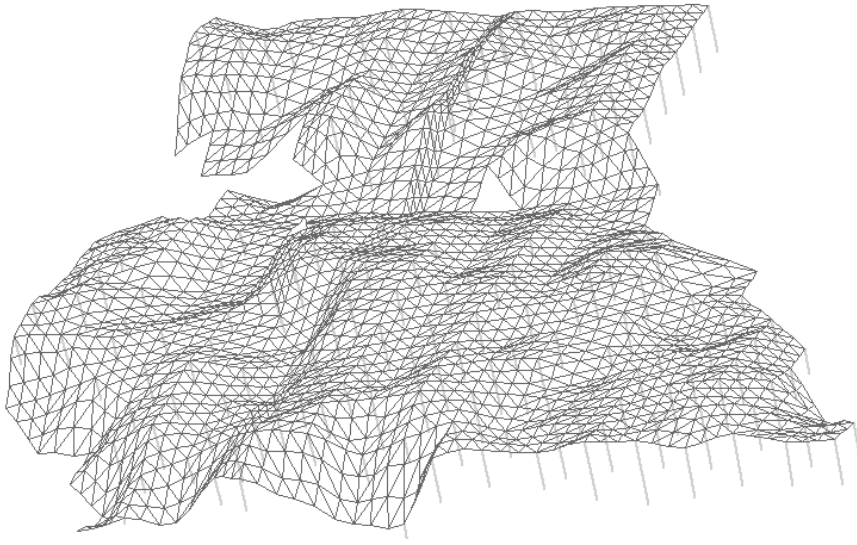


Рис. 14.16. Поверхность,
сглаженная добавленными треугольниками

Для вычисления координат вставляемых точек предлагается следующий алгоритм.

Шаг 1: разбить каждое ребро триангуляции на `nP` частей и через эти точки провести в каждом треугольнике линии, порождающие новые треугольники триангуляции (рис. 14.17).

Шаг 2: в каждой узловой точке вычислить псевдонормаль к поверхности, как среднее арифметическое единичных нормалей к треугольникам, сходящимся в этой вершине.

Шаг 3: вычислить касательные векторы к поверхности вдоль ребер триангуляции, перпендикулярные псевдонормалям.

Шаг 4: на трети расстояния вдоль касательных векторов ввести промежуточные точки кривой Безье.

Шаг 5: вычислить координаты промежуточных точек по уравнениям, описывающим кривую Безье.

14.3.1. Структура данных

Для описания кривой Безье используется следующая структура (листинг 14.21).

Листинг 14.21. Описание кривой Безье

```
TLine2=record
  p1,p2: word;
```

```
    A1,A2: TXYZ; // два касательных вектора
  end;
  TALine2=array of TLine2;
```

Каждый треугольник порождает три кривые Безье и в нем есть некая серединная точка `Pc`:

```
  TTrN=record
    Li   : array[0..2] of word;
    LineT: array[0..2] of TLine2;
    Pc   : TXYZ;    // серединная точка
  end;
```

14.3.2. Бикубическая поверхность Безье

Элементарной бикубической поверхностью Безье, определяемой матрицей $V_{i,j}$, V_{ij}, $i = 0..3,\ \ j = 0..3,$ называется поверхность [120], определяемая векторным параметрическим уравнением

$$r(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} C_3^i C_3^j u^i (1-u)^{3-i} v^j (1-v)^{3-j} V_{ij}, \qquad (14.1)$$

где $u, v \in [0,1]$, а число сочетаний из n по i определяется равенством $C_n^i = \dfrac{n!}{i!(n-i)!}$.

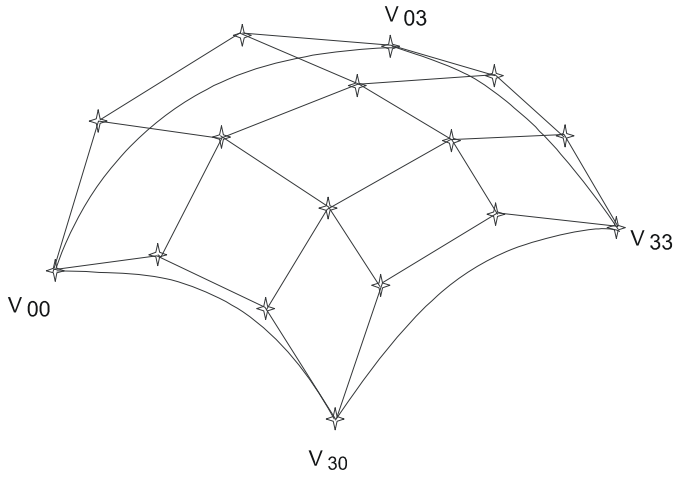


Рис. 14.17. Поверхность Безье

Поверхность Безье обладает следующими свойствами (рис 14.17):

- она является гладкой поверхностью;

- проходит через точки V_{00}, V_{30}, V_{03}, V_{33} и касается отрезков, выходящих из этих точек.

14.3.3. Вспомогательные функции

В дальнейшем потребуется несколько вспомогательных функций, реализующих операции векторной алгебры.

Функция `Summa` определяет сумму двух векторов $\overline{A}+t\times\overline{B}$ (листинг 14.22).

Листинг 14.22. Сумма двух векторов

```
function Summa(t: real; A,B: TXYZ): TXYZ;
begin
  Result.x:=A.x+t*B.x;
  Result.y:=A.y+t*B.y;
  Result.z:=A.z+t*B.z;
end;
```

Процедура `SetNormaTo1` возвращает единичный вектор (листинг 14.23).

Листинг 14.23. Единичный вектор

```
procedure SetNormaTo1(var A: TXYZ);
var n: real;
begin
  n:=Sqrt(Sqr(A.x)+Sqr(A.y)+Sqr(A.z));
  if n<>0 then begin
    A.x:=A.x/n; A.y:=A.y/n; A.z:=A.z/n;
  end;
end;
```

Функция `MultVect` находит векторное произведение двух векторов и нормирует результат (листинг 14.24).

Листинг 14.24. Векторное произведение двух векторов

```
function MultVect(A,B: TXYZ): TXYZ;
var n: real;
begin
  Result.x:= A.y*B.z-B.z*A.y;
  Result.y:=-A.x*B.z+B.x*A.z;
  Result.z:= A.x*B.y-A.y*B.x;
  n:=Sqrt(Sqr(Result.x)+Sqr(Result.y)+Sqr(Result.z));
  if n<>0 then begin
```

```
    Result.x:=Result.x/n; Result.y:=Result.y/n;
    Result.z:=Result.z/n;
  end
  else begin
    Result.x:=0; Result.y:=0; Result.z:=1;
  end;
end;
```

Функция `MultT` умножает вектор $\overline{A}$ на скаляр t (листинг 14.25).

Листинг 14.25. Умножение вектора *A* на скаляр *t*

```
function MultT(t: real; A: TXYZ): TXYZ;
begin
  Result.x:=t*A.x;
  Result.y:=t*A.y;
  Result.z:=t*A.z;
end;
```

Функция `Pr_A_in_B` находит вектор проекции вектора $\overline{A}$ на $\overline{B}$ (листинг 14.26).

Листинг 14.26. Вектор проекции вектора *A* на *B*

```
function Pr_A_in_B(A,B: TXYZ): TXYZ;
// проекция вектора A на вектор B
var k,m: real;
begin
  m:=(B.x*B.x+B.y*B.y+B.z*B.z);
  k:=(A.x*B.x+A.y*B.y+A.z*B.z)/(m*m);
  Result.x:=B.x*k;
  Result.y:=B.y*k;
  Result.z:=B.z*k;
end;
```

Функция `SetNorm` определяет вектор нормали к треугольнику, заданному вершинами `r1`, `r2`, `r3` (листинг 14.27).

Листинг 14.27. Вектор нормали к треугольнику

```
function SetNorm(r1,r2,r3: TXYZ): TXYZ;
var A,B: TXYZ;
begin
  A:=Summa(-1,r2,r1); B:=Summa(-1,r3,r1);
  Result:=MultVect(A,B);
end;
```

14.3.4. Алгоритм сглаживания триангуляции

Алгоритм состоит из восьми шагов.

Шаг 1: вычислить единичную нормаль для каждого треугольника.

Шаг 2: вычислить единичную псевдонормаль в каждой точке триангуляции.

Шаг 3: сформировать массив `Line2` и массив треугольников `TrN`, опирающихся на эти линии.

Шаг 4: для каждого отрезка массива `Line2` вычислить касательные векторы.

Шаг 5: для каждого треугольника `TrN` построить матрицу векторов Безье и новые треугольники.

Шаг 5.1: построить упорядоченные по обходу линии в треугольнике.

Шаг 5.2: переставить точки и касательные в `LineT`.

Шаг 5.3: вычислить серединную точку `Pc=A1-P1+A2-P2`.

Шаг 5.4: вычислить матрицу векторов Безье.

Шаг 5.5: вычислить остальные векторы Безье

```
InitBez2(MultT(0.5,Summa(1,Vs[0,0],Vs[3,3])));
```

Шаг 5.6: построение новых треугольников.

Шаг 5.6.1: вычислить координаты промежуточных точек.

Шаг 5.6.2: вычислить номера точек треугольника.

Шаг 6: склеить точки.

Шаг 7: вычислить для всех треугольников нормальный вектор.

Шаг 8: вычислить для всех новых точек вектор нормали.

Рассмотрим реализацию каждого шага алгоритма более подробно.

Шаг 1: вычисление единичной нормали для каждого треугольника

Для каждого треугольника нормаль будем вычислять с помощью функции `SetNorm`. Если проекция нормали на ось `z` меньше `0`, то меняется порядок обхода вершин и направление нормали с помощью функции `MultT` (умножение вектора на `-1`) (листинг 14.28).

Листинг 14.28. Вычисление единичной нормали для каждого треугольника

```
// вычислить единичную нормаль для каждого треугольника
Lt:=Length(TrH^); SetLength(Norm,Lt);
for i:=0 to Lt-1 do
```

```
with TrH^[i] do begin
  Norm:=SetNorm(PointH[P[0]].Point,PointH[P[1]].Point,PointH[P[2]].Point);
  if Norm.z<0 then begin
    j:=P[0]; P[0]:=P[1]; P[1]:=j; // меняем порядок обхода вершин
    Norm:=MultT(-1,Norm);         // меняем направление нормали
  end;
end;
```

Менять порядок обхода вершин необходимо, так как для любого треугольника проекция вектора нормали на ось `z` должна быть неотрицательна.

Шаг 2: вычисление единичной нормали в каждой точке

Каждая точка триангуляции `PointH[i]` является вершиной нескольких треугольников. Нормированный вектор суммы нормалей к этим треугольникам можно считать некоторым приближением (псевдонормалью) к вектору нормали к поверхности в точке `PointH[i]`. В листинге 14.29 приведен фрагмент программы, вычисляющей псевдонормали для всех точек триангуляции.

Листинг 14.29. Вычисление единичной псевдонормали в каждой точке

```
// вычислить единичную нормаль в каждой точке
L:=Length(PointH); SetLength(NormaH,L);
for i:=0 to L-1 do
with NormaH[i] do begin
  x:=0; y:=0; z:=0;
  for j:=0 to Lt-1 do
  with TrH^[j] do
  if (P[0]=i) or (P[1]=i) or (P[2]=i) then
    NormaH[i]:=Summa(1,NormaH[i],Norm);
  SetNormaTo1(NormaH[i]);
end;
```

Шаг 3: формирование массива *Line2* и массива треугольников *TrN*

В дальнейшем потребуется массив отрезков `Line2`, каждый элемент которого имеет структуру:

```
  TLine2=record
    p1,p2: word;
    A1,A2: TXYZ; // два касательных вектора
  end;
```

Поля `p1` и `p2` показывают на номера точек начала и конца отрезка, а векторные поля `A1` и `A2`, которые будут вычислены в дальнейшем, имеют смысл касательных векторов к кривой Безье.

Так же потребуется вспомогательный массив треугольников, который опирается не на точки, а на отрезки массива Line2. Каждый элемент этого массива содержит следующие поля:

```
  TTrN=record
    NumLine2 : array[0..2] of word;
    LineT    : array[0..2] of TLine2;
    Pc       : TXYZ;    // срединная точка
  end;
```

В этой структуре поле NumLine2 показывает номер элемента в массиве отрезков Line2.

Алгоритм шага 3 реализован в процедуре AddLines (листинг 14.30) для каждого треугольника для всех пар точек P_1, P_2:

Шаг 3.1: функцией FindLine определить номер n элемента массива Line2, содержащего точки P_1, P_2;

Шаг 3.2: если элемент массива Line2 не найден, то функцией AddLine добавить точки P_1, P_2 в массив;

Шаг 3.3: задать номер соответствующего отрезка в поле TrN[t].NumLine2[0]:=n.

Листинг 14.30. Формирование массива *Line2* и массива треугольников *TrN*

```
procedure AddLines;
var t,n,Ls: integer;
begin // AddLine
  L:=Length(TrH^); SetLength(TrN,L);
  Ls:=0;
  for t:=0 to L-1 do
  with TrH^[t] do begin
    n:=FindLine(P[0],P[1]);
    if n=-1 then n:=AddLine(P[0],P[1]);
    TrN[t].NumLine2[0]:=n;
    n:=FindLine(P[1],P[2]);
    if n=-1 then n:=AddLine(P[1],P[2]);
    TrN[t].NumLine2[1]:=n;
    n:=FindLine(P[2],P[0]);
    if n=-1 then n:=AddLine(P[2],P[0]);
    TrN[t].NumLine2[2]:=n;
  end;
end; // AddLine
```

Код функций FindLine и AddLine, использованных в листинге 14.30, приводится в листингах 14.31 и 14.32.

Листинг 14.31. Нахождение отрезка *p1p2*

```
function FindLine(p1,p2: word): integer;
var i: integer; Ok: boolean;
begin
  i:=-1; Ok:=false;
  while (i<Ls-1) and not Ok do begin
    Inc(i); Ok:=((p1=Line2[i].p1) and (p2=Line2[i].p2)) or
               ((p2=Line2[i].p1) and (p1=Line2[i].p2));
  end;
  if not Ok then Result:=-1 else Result:=i;
end;
```

Листинг 14.32. Добавление отрезка *p1p2*

```
function AddLine(p1,p2: word): integer;
begin
  Ls:=Ls+1; SetLength(Line2,Ls);
  Line2[Ls-1].p1:=p1; Line2[Ls-1].p2:=p2;
  Result:=Ls-1;
end;
```

Шаг 4: вычисление касательных векторов для каждого отрезка

Касательные векторы A_1 и A_2 в точках P_1 и P_2 вычислим по следующему алгоритму (рис. 14.18):

Шаг 1: вычислим вектор отрезка $\overline{A} = \overline{P}_2 - \overline{P}_1$;

Шаг 2: вычислим радиус-векторы точек B_1 и B_2: $\overline{B}_1 = \overline{P}_1 + \overline{A}/3$, $\overline{B}_2 = \overline{P}_2 - \overline{A}/3$;

Шаг 3: с помощью функции `Pr_A_in_B` вычислим проекции векторов $\overline{B}_1 - \overline{P}_1$ и $\overline{B}_2 - \overline{P}_2$ на нормали N_1 и N_2;

Шаг 4: найдем касательные векторы $\overline{A}_1 = \overline{B}_1 - \overline{Q}_1$ и $\overline{A}_2 = \overline{B}_2 - \overline{Q}_2$.

Этот алгоритм реализован во фрагменте кода, приведенном в листинге 14.33.

Листинг 14.33. Вычисление касательных векторов

```
  L:=Length(Line2);
  for k:=0 to L-1 do
  with Line2[k] do begin
    A:=Summa(-1,PointH[P2].Point,PointH[P1].Point);
    B1:=Summa(0.33,PointH[P1].Point,A);
```

```
    Q1:=Pr_A_in_B(Summa(-1,B1,PointH[P1].Point),NormaH[p1]);
    A1:=Summa(-1,B1,Q1);
    B2:=Summa(-0.33,PointH[P2].Point,A);
    Q2:=Pr_A_in_B(Summa(-1,B2,PointH[P2].Point),NormaH[p2]);
    A2:=Summa(-1,B2,Q2);
  end;
```

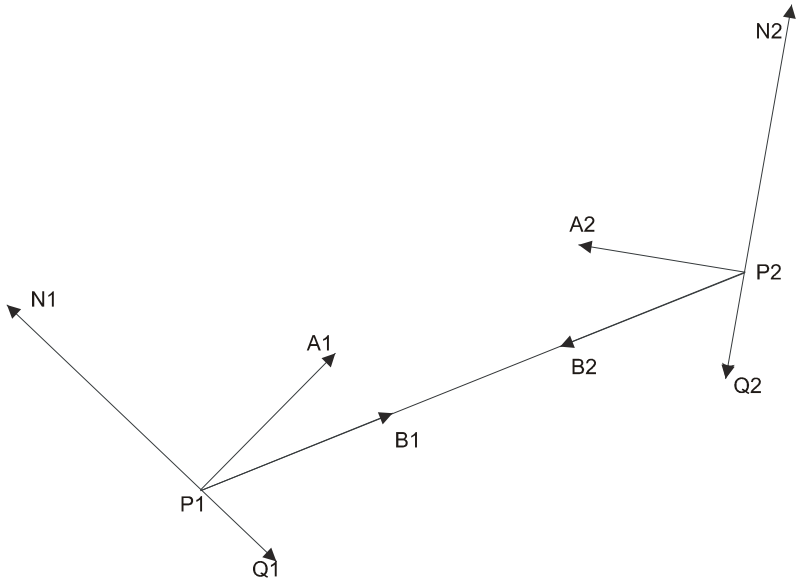


Рис. 14.18. Касательные векторы A_1 и A_2 в точках P_1 и P_2

Шаг 5: построение матрицы векторов Безье и новых треугольников для каждого треугольника *TrN*

Этот этап состоит из шести шагов.

Шаги 5.1 и 5.2: построить упорядоченные по обходу линии в треугольнике.

Вспомогательный массив `LineT: array[0..2] of TLine2` содержит для каждого треугольника упорядоченную по обходу информацию о трех отрезках, образующих треугольник: конец каждого отрезка `LineT` является началом для следующего отрезка. Фрагменты программы, предназначенные для построения этого массива, приведены в листингах 14.34 и 14.35.

Листинг 14.34. Построение упорядоченного массива отрезков треугольника

```
  LineT[0]:=Line2[Li[0]];
  if ((Line2[Li[2]].p1=LineT[0].p2) or (Line2[Li[2]].p2=LineT[0].p2))
```

```
then begin
  LineT[1]:=Line2[Li[2]]; LineT[2]:=Line2[Li[1]];
end
else begin
  LineT[1]:=Line2[Li[1]]; LineT[2]:=Line2[Li[2]];
end;
```

Листинг 14.35. Перестановка точек и касательных на отрезках *LineT*

```
for j:=1 to 2 do
with LineT[j] do
if p2=LineT[j-1].p2 then begin
  t:=p2; p2:=p1; p1:=t;
  Q:=A1; A1:=A2; A2:=Q;
end;
```

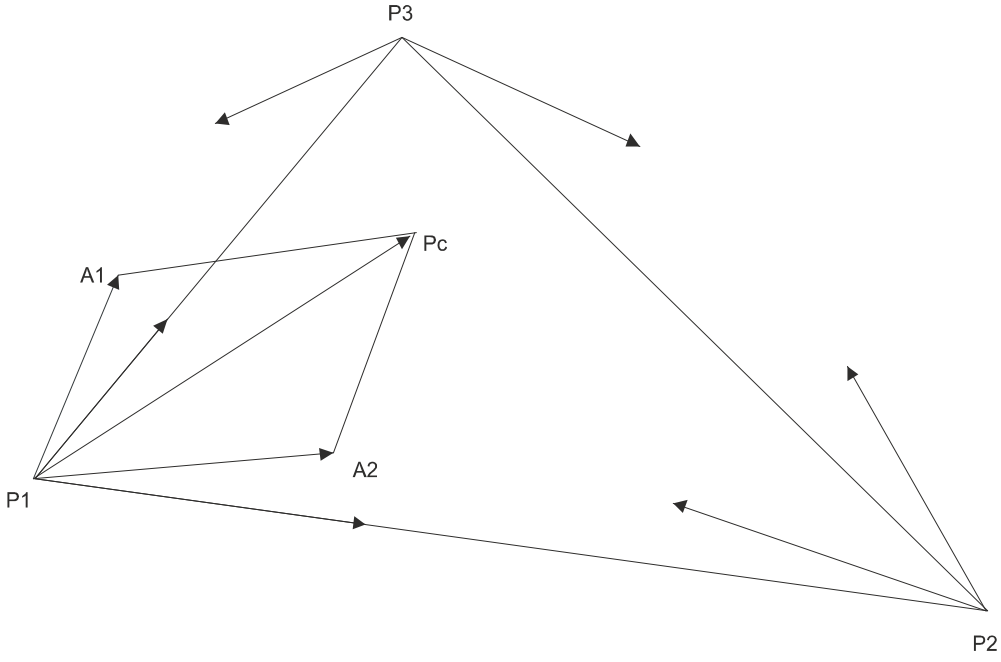


Рис. 14.19. Серединная точка *Pc* над поверхностью треугольника

Шаг 5.3: вычислить (рис. 14.19) серединную точку $P_c = A_1 - P_1 + A_2 - P_2$ над поверхностью треугольника. Серединная точка P_c потребуется для построения матрицы векторов Безье и должна находиться примерно над серединой треугольника. Прообразы касательных векторов — векторы B_1 и B_2 равны по длине 1/3 длины своих сторон треугольника. Поэтому сумма векторов $B_1 + B_2$ попадает в центр треугольника (точку пересечения медиан). Сумма касательных векторов

$A_1 - P_1 + A_2 - P_2$, выходящих из точки P_1, будет находиться примерно над серединой треугольника (рис. 14.19).

Необходимо отметить, что серединная точка, построенная на основе касательных векторов из точки P_2, наверняка не совпадет с точкой P_c.

В листинге 14.36 представлен фрагмент программы, вычисляющей координаты серединной точки над поверхностью треугольника.

Листинг 14.36. Вычисление координаты серединной точки над поверхностью треугольника

```
Pc.x:=0; Pc.y:=0; Pc.z:=0;
with LineT[0] do
  Pc:=Summa(1,Pc,Summa(-1,A1,PointH[p1].Point));
with LineT[2] do
  Pc:=Summa(1,Pc,Summa(-1,A2,PointH[p2].Point));
Pc:=Summa(1,PointH[LineT[0].p1].Point,Pc);
```

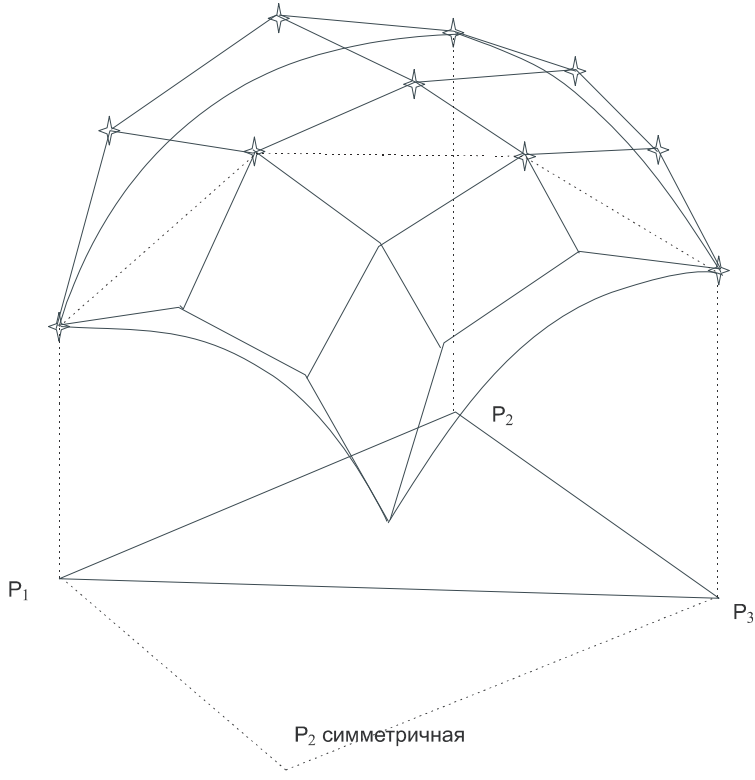


Рис. 14.20. Половина матрицы векторов Безье

Шаг 5.4: вычислить половину матрицы векторов Безье. Гладкая поверхность Безье, описываемая уравнением (14.1), определена над четырехугольником. Построим этот четырехугольник, введя точку $P_{2\ симметричная}$, симметричную точке P_2. Знание касательных векторов, выходящих из вершин треугольника P_1, P_2 и P_3, и серединной точки P_c позволяет построить 10 из 16 векторов Безье (рис. 14.20).

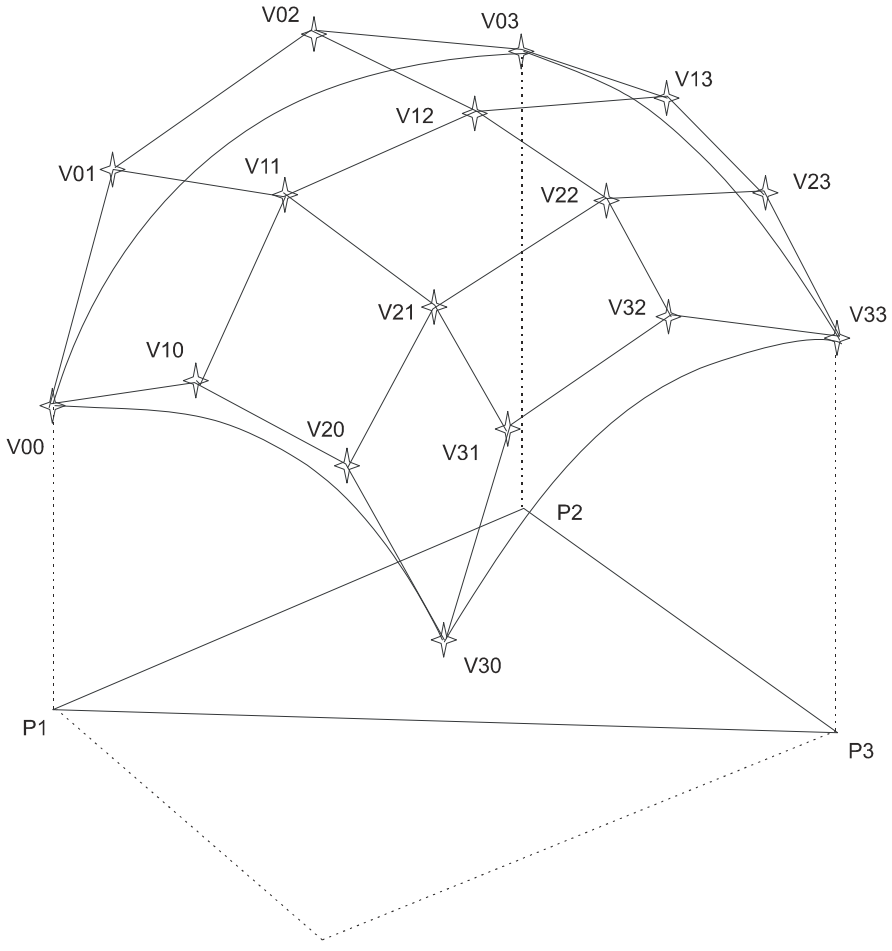


Рис. 14.21. Построение остальных векторов Безье

В листинге 14.37 приведен фрагмент программы, в котором вычисляются 10 векторов Безье.

Листинг 14.37. Вычисление 10 векторов Безье

```
Vs[0,0]:=PointH[LineT[0].p1].Point;
Vs[0,3]:=PointH[LineT[0].p2].Point;
```

```
  Vs[3,3]:=PointH[LineT[1].p2].Point;
  Vs[0,1]:=LineT[0].A1;
  Vs[0,2]:=LineT[0].A2;
  Vs[1,3]:=LineT[1].A1;
  Vs[2,3]:=LineT[1].A2;
  Vs[2,2]:=LineT[2].A1;
  Vs[1,1]:=LineT[2].A2;
  Vs[1,2]:=Pc;
```

Шаг 5.5: вычислить остальные векторы Безье. Остальные 6 векторов Безье можно вычислить, считая, что они симметричны соответствующим векторам матрицы Безье относительно середины отрезка P_1P_3 (рис. 14.21). Вычисление этих векторов реализовано в процедуре `InitBez2` (листинг 14.38).

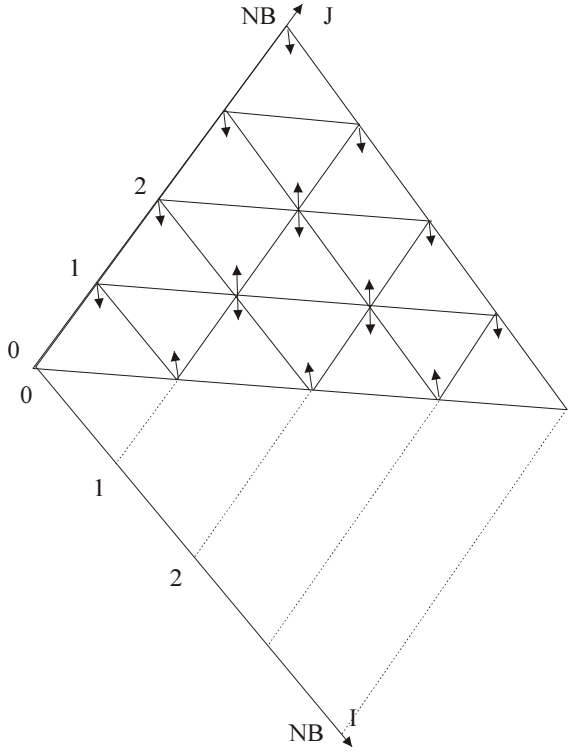


Рис. 14.22. Новые точки и треугольники

Листинг 14.38. Вычисление симметричных векторов Безье

```
procedure InitBez2(Po: TXYZ); //Po=(Vs[0,0]+Vs[3,3])/2
  function Simetr(P: TXYZ): TXYZ;
  begin
```

```
    Result:=Summa(-1,MultT(2,Po),P);
  end;
begin
  Vs[3,1]:=Simetr(Vs[0,2]); // = Vs[0,0]+Vs[3,3]-Vs[0,2]
  Vs[3,2]:=Simetr(Vs[0,1]);
  Vs[3,0]:=Simetr(Vs[0,3]);
  Vs[2,0]:=Simetr(Vs[1,3]);
  Vs[1,0]:=Simetr(Vs[2,3]);
  Vs[2,1]:=Simetr(Vs[1,2]);
end;
```

Шаг 5.6: построение новых треугольников. Будем считать, что каждая сторона треугольника разбивается на `nB` частей. Это приведет к появлению `nN:=(nB+1)*(nB+2)/2` новых точек (рис. 14.22).

Пройдем двойным циклом по всем новым точкам и вычислим их координаты. Занесем их координаты в массив `Po` (листинг 14.39).

Листинг 14.39. Вычисление координат новых точек

```
    h:=1.0/nB;
    for i:=0 to nB do begin
      u:=i*h;
      for j:=i to nB do begin
        v:=j*h;
        Ls:=Ls+1; SetLength(Po,Ls);
        Po[Ls-1]:=Bez2(u,v,Vs);
        …
      end;
    end;
```

Шаг 5.6.1: вычисление координат промежуточных точек будем проводить с помощью уравнения (14.1), которое реализовано в виде функции `Bez2` (листинг 14.40)

Листинг 14.40. Вычисление координат промежуточных точек

```
function Bez2(u,v: real; Vs: TV4_4): TXYZ;
const C: array[0..3] of real=(1,3,3,1);
var i,j: byte;
begin
  ua[0]:=1; u_a[0]:=1;
  va[0]:=1; v_a[0]:=1;
  for i:=1 to 3 do begin
    ua[i]:=ua[i-1]*u; u_a[i]:=u_a[i-1]*(1-u);
    va[i]:=va[i-1]*v; v_a[i]:=v_a[i-1]*(1-v);
  end;
  Result.x:=0;
  for i:=0 to 3 do
```

```
  for j:=0 to 3 do
    Result.x:=Result.x+
      C[i]*C[j]*ua[i]*u_a[3-i]*va[j]*v_a[3-j]*Vs[i,j].x;
  Result.y:=0;
  for i:=0 to 3 do
  for j:=0 to 3 do
    Result.y:=Result.y+
      C[i]*C[j]*ua[i]*u_a[3-i]*va[j]*v_a[3-j]*Vs[i,j].y;
  Result.z:=0;
  for i:=0 to 3 do
  for j:=0 to 3 do
    Result.z:=Result.z+
      C[i]*C[j]*ua[i]*u_a[3-i]*va[j]*v_a[3-j]*Vs[i,j].z;
end;
```

Все новые точки будем собирать в массиве `Po: array of TXYZ`. Поэтому для следующего шага построения новых треугольников, которые опираются на точки массива `Po`, потребуются две функции:

- функция `IJ_in_N`, которая по номеру строки точек `i0` и номеру точки в строке `j0` возвращает абсолютный номер точки в массиве `Po`;
- обратная процедура `N_in_IJ`, которая по абсолютному номеру точки вычисляет номер строки `i` и номер точки в строке `j`.

Тексты этих функций приведены в листинге 14.41.

Листинг 14.41. Определение абсолютных номеров точек

```
function IJ_in_N(i0,j0: integer): word;
var m,                // количество точек в каждой строке
    i: integer;
begin
  m:=nB; Result:=j0;
  for i:=0 to i0-1 do begin
    Result:=Result+m;
    m:=m-1;
  end;
end;
procedure N_in_IJ(n: word; var i,j: word);
var m: word;
begin
  m:=nB+1; i:=0; j:=n;
  while j>=m do begin
    j:=j-m; m:=m-1; i:=i+1;
  end;
  j:=j+i;
end;
```

Шаг 5.6.2: вычисление номеров точек треугольника и добавление новых треугольников. Из рис. 14.22 видно, что точки с номерами `(0,0)` и `(nB,nB)` не добавляют новых треугольников. Часть точек добавляет по одному треугольнику, остальные добавляют по два новых треугольника (листинг 14.42).

Листинг 14.42. Добавление треугольников

```
if not ((i=0) and (j=0)) or ((i=nB) and (j=nB))) then begin
  if (i=0) or (j=nB) then AddTr(i,j,i+1,j,I,j+1)
  else
  if i=j then AddTr(i,j,i-1,j,i,j-1)
  else begin
    AddTr(i,j,i+1,j,I,j+1);
    AddTr(i,j,i-1,j,i,j-1);
  end;
end;
```

Процедура `AddTr` добавления нового треугольника, использующая функцию `IJ_in_N` определения абсолютного номера точки по номеру строки `i0` и номеру точки в строке `j0`, приведена в листинге 14.43.

Листинг 14.43. Добавление нового треугольника

```
procedure AddTr(n1,m2,n2,m2,n3,m3: word);
begin
  Lp:=Lp+1; SetLength(TrH^,Lp);
  with TrH^[Lp-1] do begin
    P[0]:=IJ_in_N(n1,m1)+k*nN; P[1]:=IJ_in_N(n2,m2)+k*nN;
    P[2]:=IJ_in_N(n3,m3)+k*nN;
  end;
end;
```

Завершающие шаги алгоритма

Шаг 6: склеить точки. Построенный массив точек `Po` является вспомогательным и избыточным: может оказаться, что для двух соседних треугольников начальной триангуляции соседние точки на границе близки. Потому при построении основного массива точек `PointH` необходимо добавлять только существенно отличающиеся точки, отстоящие друг от друга на расстояние больше `Eps`. Реализация добавления отличающихся точек показана в листинге 14.44.

Листинг 14.44. Добавление существенно отличающихся точек

```
  L:=0;
  for i:=0 to Lp-1 do
```

```
with TrH^[i] do begin
  for j:=0 to 2 do begin
    n:=FindPoint_in_Po1(P[j]);
    if n<>-1 then P[j]:=n
    else begin
      L:=L+1; SetLength(PointH,L);
      PointH[L-1].Point:=Po[P[j]];
      P[j]:=L-1;
    end;
  end;
end;
```

На шаге 7 алгоритма осталось только для всех треугольников с помощью функции `SetNorm` вычислить нормальный вектор (листинг 14.45).

Листинг 14.45. Вычисление нормального вектора для всех треугольников

```
for i:=0 to Lp-1 do
with TrH^[i] do begin
  Norm:=SetNorm(PointH[P[0]].Point,PointH[P[1]].Point,
                PointH[P[2]].Point);
  if Norm.z<0 then begin
    j:=P[0]; P[0]:=P[1]; P[1]:=j; Norm:=MultT(-1,Norm);
  end;
end;
```

И, наконец, осталось выполнить шаг 8: для всех новых точек с помощью функций `Summa` и `SetNormaTo1` вычислить усредненный вектор нормали (листинг 14.46).

Листинг 14.46. Вычисление усредненного вектора нормали

```
L:=Length(PointH); SetLength(NormaH,L);
for i:=0 to L-1 do
with NormaH[i] do begin
  x:=0; y:=0; z:=0;
  for j:=0 to Lt-1 do
  with TrH^[j] do
  if (P[0]=i) or (P[1]=i) or (P[2]=i) then
    NormaH[i]:=Summa(1,NormaH[i],Norm);
  SetNormaTo1(NormaH[i]);
end;
```

Полный текст данного проекта приведен на компакт-диске в папке **Примеры | Глава 14 | Сглаживание поверхности**.

14.4. Триангуляция боковой поверхности слоя

В *разд. 14.2* представлен алгоритм триангуляции верней и нижней поверхности слоя. Необходимо построить триангуляцию боковой поверхности слоя, а для этого придется решить задачу построения упорядоченного по обходу массива номеров граничных точек верхней или нижней поверхности. На самом деле, триангулированная поверхность, как это видно на рис. 14.23 для Якутской свиты, может быть многосвязной, и поэтому необходимо для каждого фрагмента поверхности построить свой упорядоченный массив номеров граничных точек.

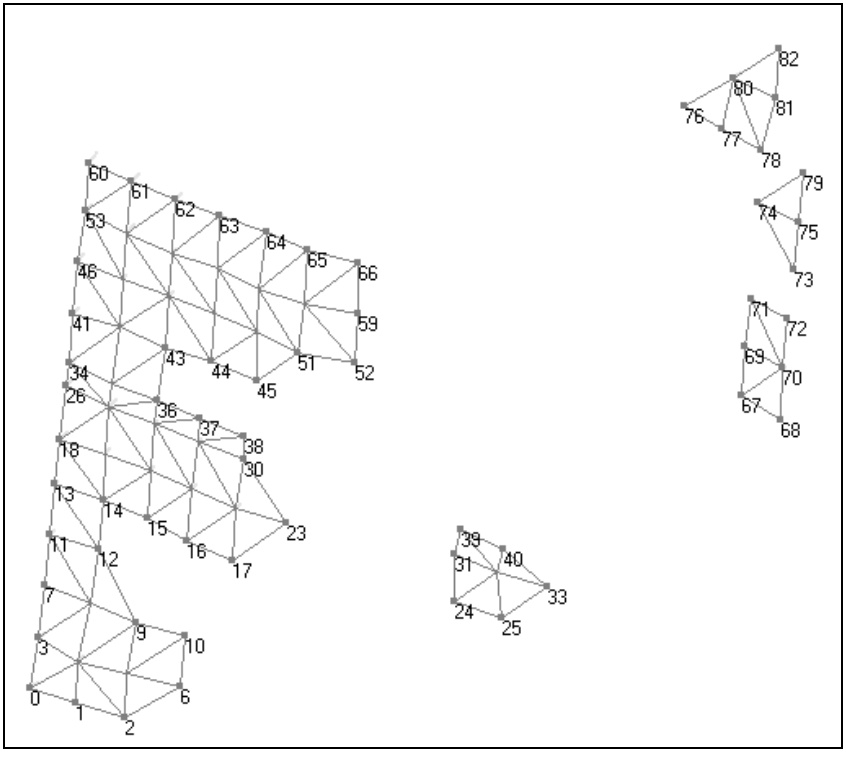


Рис. 14.23. Упорядоченные по обходу массивы номеров граничных точек

Так для поверхности, представленной на рис. 14.14, массивы номеров граничных точек должны содержать следующие значения:

```
(0,1,2,6,10,9,12,14,15,16,17,23,30,38,37,36,43,44,45,51,52,59,66,65,64,63,62,
61,60,53,46,41,34,26,18,13,11,7,3,0),(24,25,33,40,39,31,24),(67,68,70,72,71,
69,67), (73,75,79,74,73),(76,77,78,81,82,80,76)
```

14.4.1. Структура данных

Алгоритм определения этих массивов опирается только на топологические свойства данных, то есть принадлежность точек треугольникам. Поэтому алгоритм использует массив точек `PointH: array of TPointMy`, где структура `TPointMy` имеет следующий вид:

```
TPointMy =record
  Neighbor: array of word; // номера соседних точек
end;
```

В этой структуре значения поля `Neighbor` будут указывать на номера соседних точек, то есть на номера точек, входящих в треугольник, одной из вершин которой является точка.

Сначала соберем граничные точки в массив `pBorder: array of TBorderPoint`, а затем распределим эти точки в двумерный динамический массив `S_` стратиграфического `Border: array of array of word`, определенный для каждого значения справочника.

Граничные точки `TBorderPoint` образуют граф, узлы которого соединены дугами (ветвями). Чаще всего из каждого узла выходит две дуги, соединяющие граничную точку с двумя соседними. Но для многосвязных поверхностей с дырками возможны случаи, когда из узла выходят четыре или более дуг. Структура этого графа описана в листинге 14.47.

Листинг 14.47. Структура графа

```
TEdge=record
  Index: word;          // номер соседней граничной точки
  Visit: boolean;       // признак прохождения по ветви
end;
TBorderPoint=record
  Index: word;          // N в массиве PointH
  Edge: array of TEdge; // массив ветвей на соседние
                        // граничные точки
end;
```

Поле `Index` указывает на номер точки в массиве `PointH`, а поле `Edge: array of TEdge` образует массив ветвей на соседние граничные точки. В структуре `Edge` поле `Index: word` показывает на номер соседней точки, а поле `Visit: boolean` является признаком прохождения ветви.

14.4.2. Алгоритм определения номеров граничных точек

Алгоритм определения номеров граничных точек состоит из следующих пяти шагов:

Шаг 1: подготовить вспомогательный массив точек `PointH`;

Шаг 2: для каждой точки `PointH[i]` установить соседей;

Шаг 3: построить массив граничных точек `pBorder`;

Шаг 4: построить ветви для каждой граничной точки `pBorder[i]`;

Шаг 5: построить упорядоченные массивы номеров граничных точек `S_Border`.

Рассмотрим каждый шаг алгоритма более подробно.

Шаг 1: подготовка вспомогательного массива точек

Зададим длину массива `PointH` совпадающей с длиной массива исходных точек `Points`. Для каждой точки `PointH[i]` обнулим длину массива соседних точек `Neighbor` (листинг 14.48).

Листинг 14.48. Подготовка вспомогательного массива точек

```
Len:=Length(Points);
SetLength(PointH,Len);
for i:=0 to Len-1 do
  SetLength(PointH[i].Neighbor,0);
```

Шаг 2: установление соседних точек для каждой точки

Алгоритм второго шага тоже достаточно прост. Необходимо перебрать все треугольники и для каждой пары точек `n1` и `n2` проверить условие: если точки `n2` нет среди соседей точки `n1`, то включить ее в число соседей точки `n1`. Процедура `SetNeighbor`, представленная в листинге 14.49, реализует этот алгоритм, используя процедуру `Include`.

Листинг 14.49. Установление соседних точек для каждой точки

```
  procedure SetNeighbor;
  var i: integer;

    procedure Include(n1,n2: word);
    // если точки n2 нет среди соседей точки n1, то включить
    var L,i: integer;
        Ok: boolean;
    begin
      L:=Length(PointH[n1].Neighbor); i:=-1; Ok:=false;
      while (i<L-1) and not Ok do begin
        Inc(i);
        Ok:=n2=PointH[n1].Neighbor[i];
      end;
      if not Ok then begin
```

```
        Inc(L); SetLength(PointH[n1].Neighbor,L);
        PointH[n1].Neighbor[L-1]:=n2;
      end;
    end;

  begin
    L:=Length(TrH);
    for i:=0 to L-1 do
    with TrH[i] do begin
      Include(P[0],P[1]);
      Include(P[0],P[2]);
      Include(P[1],P[0]);
      Include(P[1],P[2]);
      Include(P[2],P[0]);
      Include(P[2],P[1]);
    end;
  end;
```

Шаг 3: построение массива граничных точек

Построение массива граничных точек основано на утверждении: если у точки с номером `i` среди ее соседей существует сосед с одним общим соседом, то `i`-ая точка — граничная. Поэтому для каждой точки `PointH[i]` необходимо выполнить следующие шаги алгоритма.

Шаг 1: просмотрим все соседние точки `Neighbor[j]`.

Шаг 2: вычислим число общих соседей с помощью функции `NumNeighbor`.

Шаг 3: если число общих соседей равно `1`, то включим точку с номером `i` в массив граничных точек `pBorder`.

Алгоритм представлен в листинге 14.50.

Листинг 14.50. Построение массива граничных точек *pBorder*

```
Lp:=0; L:=Length(PointH);
for i:=0 to L-1 do
with PointH[i] do begin
  L0:=Length(PointH[i].Neighbor);
  Ok:=false; j:=-1;
  while (j<L0-1) and not Ok do begin
    Inc(j);
    k:=PointH[i].Neighbor[j]; Ok:=NumNeighbor(i,k)=1;    // !!!
  end;
  if Ok then begin
    Inc(Lp); SetLength(pBorder,Lp);
```

```
    pBorder[Lp-1].Index:=i;
    SetLength(pBorder[Lp-1].Edge,0);
  end;
end;
```

Функция `NumNeighbor` двойным циклом проверяет совпадение номеров соседей для точек `n1` и `n2` и в случае совпадения увеличивает значение переменной `Result` на `1` (листинг 14.51).

Листинг 14.51. Вычисление числа общих соседей

```
function NumNeighbor(n1,n2: integer): integer;
// число общих соседей
var i,j,L1,L2: integer;
begin
  Result:=0;
  L1:=Length(PointH[n1].Neighbor);
  L2:=Length(PointH[n2].Neighbor);
  for i:=0 to L1-1 do
  for j:=0 to L2-1 do
  if (PointH[n1].Neighbor[i]=PointH[n2].Neighbor[j]) then
    Inc(Result);
end;
```

Шаг 4: построение ветвей для каждой граничной точки

При построении ветвей для каждой граничной точки снова воспользуемся признаком граничной точки: если у `i`-й точки среди ее соседей существует сосед с одним общим соседом, то `i`-ая точка — граничная. Поэтому для всех граничных точек и для всех ветвей каждой точки выполним следующие действия (листинги 14.52, 14.53, 14.54).

Шаг 1: проверим условие: у `i`-й точки среди ее соседей существует сосед с одним общим соседом.

Шаг 2: если условие не выполнено, то переходим на шаг 1, иначе — на шаг 3.

Шаг 3: по номеру соседа точки `PointH` определить номер `m` в массиве `pBorder`.

Шаг 4: для точки `i` добавить ветвь на точку `m` и для точки `m` добавить ветвь на точку `i`.

Листинг 14.52. Построение ветвей для каждой граничной точки

```
// строит ветви для каждой граничной точки
for i:=0 to Lp-1 do begin
  L0:=Length(PointH[pBorder[i].Index].Neighbor);
  for j:=0 to L0-1 do begin
    k:=PointH[pBorder[i].Index].Neighbor[j];     //
```

```
    if NumNeighbor(pBorder[i].Index,k)=1 then begin
      // точки pBorder[i].Index и k - соседние граничные
      m:=Find_pBorder(k);
      if m<>-1 then begin
        AddEdge(i,m);
        AddEdge(m,i);
      end;
    end;
  end;
end;
```

Теперь по номеру точки `k` в массиве `PointH` нужно определить номер точки в массиве граничных элементов.

Листинг 14.53. Определить номер точки в массиве граничных элементов

```
function Find_pBorder(k: word): integer;
var i,L: integer;
    Ok: boolean;
begin
  L:=Length(pBorder); i:=-1; Ok:=false;
  while (i<L-1) and not Ok do begin
    Inc(i); Ok:=k=pBorder[i].Index;
  end;
  if Ok then Result:=i else Result:=-1;
end;
```

Для точки `n` добавляем ветвь на точку `m`, если ее нет в списке ветвей (листинг 14.54).

Листинг 14.54. Добавление ветви для точки *n* на точку *m*

```
procedure AddEdge(n,m: word);
var i,L: integer;
    Ok: boolean;
begin
  L:=Length(pBorder[n].Edge);
  i:=-1; Ok:=false;
  while (i<L-1) and not Ok do begin
    Inc(i); Ok:=pBorder[n].Edge[i].Index=m;
  end;
  if not Ok then begin
    Inc(L); SetLength(pBorder[n].Edge,L);
    pBorder[n].Edge[L-1].Index:=m;
    pBorder[n].Edge[L-1].Visit:=false;
  end;
end;
```

Шаг 5: построение упорядоченных массивов номеров граничных точек

Пятый шаг алгоритма, реализующий построение упорядоченных массивов номеров граничных точек, наиболее сложен и состоит из шести шагов.

Шаг 1: среди граничных точек ищется такая точка, у которой ветви не проходились.

Шаг 2: номер границы `CountC` увеличивается на `1` и номер найденной точки добавляется к массиву номеров границы.

Шаг 3: ищется граничная точка, соседняя с найденной граничной точкой.

Шаг 4: если точка найдена, то она добавляется к массиву номеров границы, и мы переходим на шаг 3. Иначе переходим на шаг 5.

Шаг 5: среди граничных точек ищется такая точка, у которой ветви не проходились.

Шаг 6: если точка найдена, то возвращаемся на шаг 2, иначе завершаем алгоритм.

Фрагмент программы, реализующий этот алгоритм, представлен в листинге 14.55.

Листинг 14.55. Построение упорядоченных массивов номеров граничных точек

```
// строим упорядоченные массивы номеров граничных точек
CountC:=0; CountP:=0;
NumP:=FindPBorder;
// среди граничных точек ищется такая,
// у которой ветви не проходились
while NumP<>-1 do begin
  Inc(CountC); Inc(CountP);
  SetLength(ABorder,CountC);
  SetLength(ABorder[CountC-1],CountP);
  ABorder[CountC-1,CountP-1]:=pBorder[NumP].Index;
  NumP:=FindEdge(NumP);
  while NumP<>-1 do begin
    Inc(CountP); SetLength(ABorder[CountC-1],CountP);
    ABorder[CountC-1,CountP-1]:=pBorder[NumP].Index;
    NumP:=FindEdge(NumP);
  end;
  CountP:=0;
  NumP:=FindPBorder;
end;
```

Алгоритм использует две функции: функцию `FindPBorder`, которая среди граничных точек ищет такую точку, у которой ветви не проходились, и функцию `FindEdge`, которая для точки ищет непройденную ветвь.

Функция `FindPBorder` (листинг 14.56) просматривает граничные точки до тех пор, пока среди ветвей точки не найдет непройденную точку `pBorder[i].Edge[j].Visit=false`.

Листинг 14.56. Поиск точки, у которой ветви не проходились

```
function FindPBorder: integer;
var L,i,L0,j: integer;
    Ok: boolean;
begin
  L:=Length(pBorder); i:=-1; Ok:=false;
  while (i<L-1) and not Ok do begin
    Inc(i);
    L0:=Length(pBorder[i].Edge); j:=-1;
    while (j<L0-1) and not Ok do begin
      Inc(j);
      Ok:=not pBorder[i].Edge[j].Visit;
    end;
  end;
  if Ok then Result:=i else Result:=-1;
end;
```

Функция `FindEdge(n: word)` для точки `n` пытается найти непройденную ветвь. Если такая ветвь найдена, то функция отмечает ее как пройденную, ищет обратную ей и отмечает ее как пройденную (листинг 14.57).

Листинг 14.57. Поиск непройденной ветви

```
function FindEdge(n: word): integer;
// для точки n найти непройденную ветвь
// Если такая ветвь найдена, то отметить ее и обратную как пройденные
var L,i: integer;
    Ok: boolean;
begin
  L:=Length(pBorder[n].Edge); Result:=-1;
  i:=-1; Ok:=false;
  while (i<L-1) and not Ok do begin
    Inc(i);
    Ok:=not pBorder[n].Edge[i].Visit;
  end;
  if Ok then begin
    pBorder[n].Edge[i].Visit:=true;
    Result:=pBorder[n].Edge[i].Index;
    L:=Length(pBorder[Result].Edge);
    i:=-1; Ok:=false;
    while (i<L-1) and not Ok do begin
      Inc(i);
```

```
      Ok:=pBorder[Result].Edge[i].Index=n;
    end;
    if Ok then
      pBorder[Result].Edge[i].Visit:=true;
  end;
end;
```

14.4.3. Построение треугольников боковой поверхности

В предыдущих разделах был построен массив треугольников `TrP`, описывающих слой сверху и снизу. В двумерном массиве `P_Border` накоплены номера граничных точек слоя. Необходимо добавить к массиву треугольников треугольники боковой поверхности, опирающиеся на граничные точки.

Каждая граничная точка слоя порождает два новых треугольника. Поэтому если длина массива граничных точек равна `Lp`, то массив треугольников увеличится на `2*Lp` элементов. В листинге 14.58 приведен фрагмент программы, задающий треугольники боковой поверхности.

Листинг 14.58. Определение треугольников боковой поверхности

```
Lp:=0; Lh:=Length(TrP);
for j:=0 to Length(P_Border)-1 do begin
  Ls:=Length(P_Border[j]);
  Lp:=Lp+Ls;
end;

SetLength(TrP,Lh+2*Lp); p:=-2;
for j:=0 to Length(P_Border)-1 do begin
  Ls:=Length(P_Border[j]);
  if Ls>0 then
  for i:=0 to Ls-1 do begin
    Inc(p,2);
    with TrP[Lh+p] do begin
      P[0]:=P_Border[j,i];
      P[1]:=P_Border[j,(i+1) mod Ls];
      P[2]:=P_Border[j,i]+Lg;
    end;
    with TrP[Lh+p+1] do begin
      P[0]:=P_Border[j,i]+Lg;
      P[1]:=P_Border[j,(i+1) mod Ls];
      P[2]:=P_Border[j,(i+1) mod Ls]+Lg;
    end;
  end;
end;
```

14.5. Триангуляция невыпуклого многоугольника

Библиотека OpenGL не может изображать невыпуклые многоугольники.

Слой разреза породы представляет собой невыпуклый многоугольник, в общем случае многосвязный (рис. 14.24).

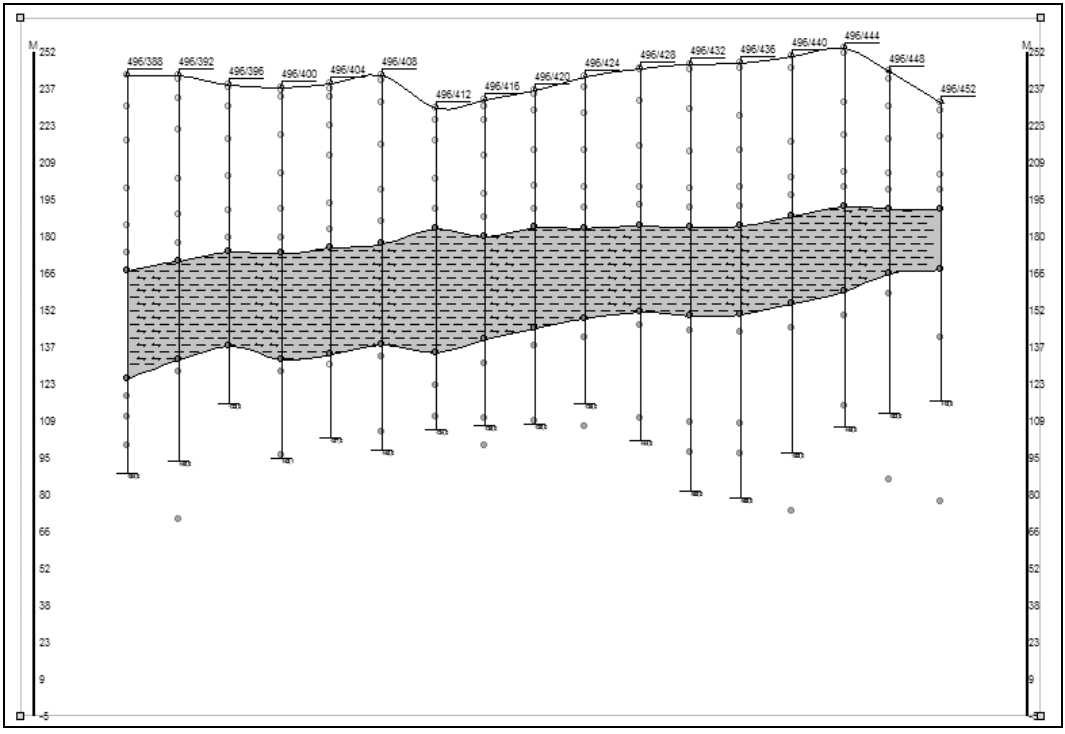


Рис. 14.24. Слой разреза породы

Рассматриваемый в данном разделе алгоритм производит разбиение произвольного (в смысле выпуклости) многоугольника на треугольники.

Пусть некоторый многоугольник задан набором координат своих вершин $P_0 = (x_0, y_0)$, $P_1 = (x_1, y_1)$, ..., $P_n = (x_n, y_n)$. При этом будем полагать, что вершины пронумерованы по часовой стрелке. В случае если многоугольник выпуклый, задача нахождения разбиения тривиальна, можно взять набор диагоналей, выходящих из какой-либо вершины (рис. 14.25).

В случае невыпуклого многоугольника задача существенно усложняется. Будем действовать следующим образом: на каждом шаге алгоритма будем отсекать один треугольник $P_{i-1}P_iP_{i+1}$ и исключать вершину P_i из дальнейшего рассмотрения.

Чтобы определить, можно ли использовать диагональ $P_{i-1}P_{i+1}$ для отсечения треугольника, будем вначале проверять знак синуса угла $P_{i-1}P_iP_{i+1}$, учитывая при этом последовательность обхода вершин. Так, для многоугольника, изображенного на рис. 14.26, при $i = 0, 1, 2, 3, 4, 7$ синус будет отрицателен, а при $i = 5, 6$ он будет положителен.

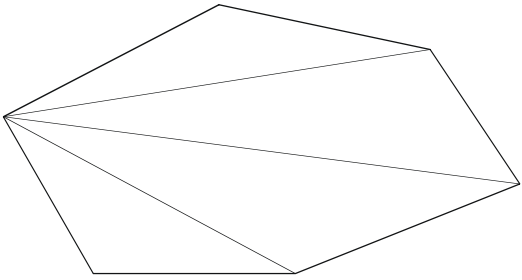


Рис. 14.25. Триангуляция выпуклого многоугольника

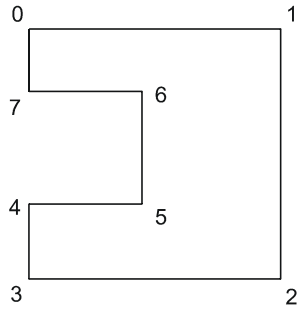


Рис. 14.26. Невыпуклый многоугольник

Далее, найдя i, для которого синус отрицателен, проверим, не входят ли в треугольник $P_{i-1}P_iP_{i+1}$ какие-либо точки из многоугольника (естественно, кроме точек P_{i-1}, P_i, P_{i+1}). Если точки из многоугольника не входят в треугольник $P_{i-1}P_iP_{i+1}$, то отсечем этот треугольник, иначе ищем следующую вершину с отрицательным синусом и проверяем ее. Проверку принадлежности точки треугольнику можно осуществить одним из алгоритмов, рассмотренных в данном разделе, переписав его для частного случая треугольника.

Алгоритм можно оптимизировать по скорости, введя, например массив, содержащий знаки синусов, и вычисляя на каждом шаге только i-ый знак. Возможны и другие усовершенствования.

Пусть дан упорядоченный массив координат вершин невыпуклого многоугольника `P: array of TXYZ`. Необходимо построить массив треугольников `Tr: array of TTr`, опирающихся на эти точки.

Введем вспомогательный массив `Numbe: array of word`, в котором будут содержаться номера вершин многоугольника. Алгоритм заключается в поиске треугольников с острыми углами, опирающихся на соседние вершины, и состоит из следующих шагов (рис. 14.27).

Шаг 1: заполним массив `Numbe` номерами точек от `0` до `n-1`.

Шаг 2: найдем точку с номером i, такую, что треугольник P_{i-1}, P_i, P_{i+1} будет иметь угол с неотрицательным косинусом.

Шаг 3: исключим вершину P_i из дальнейшего рассмотрения.

Шаг 4: если число оставшихся вершин больше трех, то перейти на шаг 2.

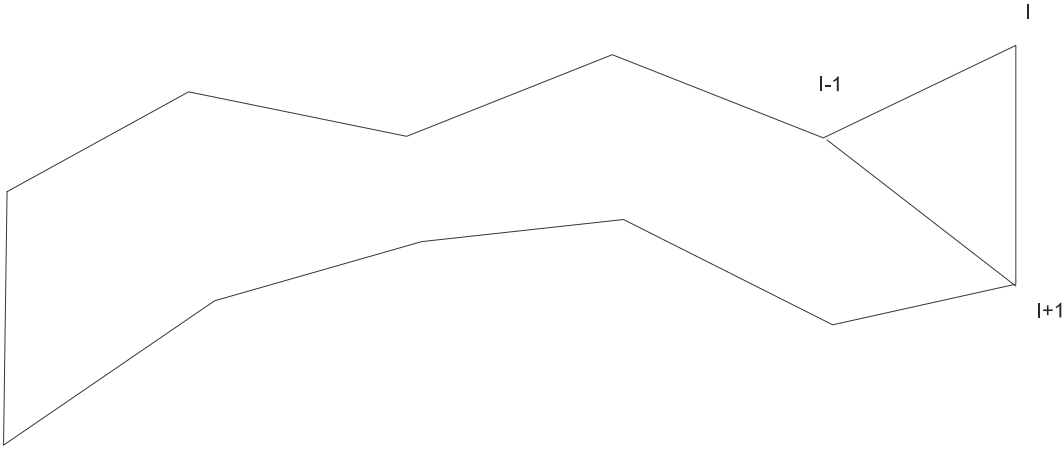


Рис. 14.27. Триангуляция невыпуклого многоугольника

Рассмотренный алгоритм реализован в процедуре `PolygonToTriangle`, текст которой представлен в листинге 14.59.

Листинг 14.59. Триангуляция невыпуклого многоугольника

```
procedure PolygonToTriangle(P: TAXYZ; var Tr: TATr);
var n,i,m,L,k,i0,i1,i2: integer;
    Numbe: array of word;
    Ok: boolean;
    d: real;
begin
  n:=Length(P); SetLength(Numbe,n);
  for i:=0 to n-1 do Numbe [i]:=i;
  m:=n;
  L:=1; SetLength(Tr,L);
  while m>3 do begin
    // отсекать один треугольник Pi-1,Pi,Pi+1
    // и исключать вершину Pi из дальнейшего рассмотрения
    i:=0; Ok:=false;
    while not Ok do begin
      d:=-1; i:=-1;
      // проверять знак синуса угла Pi-1,Pi,Pi+1
      while (i<m-1) and (d<0) do begin
        Inc(i);
        i2:= Numbe[(i+1) mod m];
        i1:= Numbe[i];
        i0:= Numbe[(i-1+m) mod m];
        d:=Sqr(Dist(P[i1],P[i2]))+Sqr(Dist(P[i1],P[i0]))-
           Sqr(Dist(P[i2],P[i0]));
      end;
```

```
      // не входят ли в треугольник Pi-1PiPi+1 какие-либо точки
      // из многоугольника
      // (естественно кроме точек Pi-1, Pi, Pi+1),
      // если нет, значит, отсечем этот треугольник,
      // иначе ищем следующую вершину с отрицательным синусом
      // и проверяем ее
      k:=-1; Ok:=true;
      while (k<m-1) and Ok do begin
        Inc(k);
        if (Numbe[k]<>i0) and (Numbe[k]<>i1) and (Numbe[k]<>i2)
        then
          Ok:=not IsPoIntoTr(P[Numbe[k]],P[i0],P[i1],P[i2]);
      end;
    end;
    Tr[L-1].P[0]:=i2; Tr[L-1].P[1]:=i1; Tr[L-1].P[2]:=i0;

    for k:=i+1 to m-1 do Numbe[k-1]:= Numbe[k];
    m:=m-1;
    L:=L+1; SetLength(Tr,L);
  end;
  Tr[L-1].P[0]:=Numbe[0]; Tr[L-1].P[1]:=Numbe[1]; Tr[L-1].P[2]:=Numbe[2];
  SetLength(Numbe,0);
end;
```

Проверка условия принадлежности точки P треугольнику основана на очевидном утверждении: только для внутренних точек треугольника выполняется равенство площади треугольника сумме площадей трех треугольников с общей вершиной в точке P. Это условие проверяется функцией `IsPoIntoTr`, текст которой приведен в листинге 14.60.

Листинг 14.60. Проверка условия принадлежности точки треугольнику

```
function IsPoIntoTr(P,P1,P2,P3: TXYZ): boolean;
var s,s1,s2,s3: real;
begin
  s:=Surf(P1,P2,P3); s1:=Surf(P,P2,P3); s2:=Surf(P1,P,P3);
  s3:=Surf(P1,P2,P);
  Result:=s1+s2+s3<=s;
end;
```

Площадь треугольника вычисляется по формуле Герона и реализуется функцией `Surf` (листинг 14.61). Используемая в данном листинге функция `Dist` приведена в листинге 14.62.

Листинг 14.61. Функция вычисления площади треугольника

```
function Surf(P1,P2,P3: TXYZ): real;
var a,b,c,p: real;
```

```
begin
  a:=Dist(P1,P2); b:=Dist(P3,P2); c:=Dist(P3,P1);
  p:=(a+b+c)/2;
  Result:=Sqrt(p*(p-a)*(p-b)*(p-c));
end;
```

Листинг 14.62. Функция вычисления расстояния между точками

```
function Dist(P1,P2: TXYZ): real;
begin
  Result:=Sqrt(Sqr(P1.x-P2.x)+Sqr(P1.y-P2.y)+
               Sqr(P1.z-P2.z));
end;
```

14.6. Изолинии

В проекте "Изолинии" рассматривается задача построения изолиний и топографической карты поверхности. Эта поверхность может быть дневной поверхностью земли или поверхностью слоя породы или значением объемных геофизических измерений, сделанных на определенной глубине.

Будем считать, что поверхность описана массивом треугольников `Tr: array of TTr` (рис. 14.28), полученных в результате выполнения задачи триангуляции. Эти треугольники опираются на массив точек `Points: array of TPoints`.

Вертикальные координаты точек меняются от $Z_{\min}$ до $Z_{\max}$. Необходимо построить `MaxP` сечений плоскостями $Z = H_k$. Все треугольники, попавшие между плоскостями H_k и H_{k+1}, нужно закрасить в свой цвет c_k. На рис. 14.29 представлена топографическая раскраска поверхности Сунгарской свиты. Белые треугольники говорят о том, что в соответствующих скважинах не было зафиксировано породы Сунгарской свиты. На дневной поверхности (рис. 14.30) белые треугольники отсутствуют, т. к. для любой скважины известна высота ее устья.

Алгоритм заключается в выполнении такой последовательности шагов.

Шаг 1: все треугольники закрашиваются в начальный цвет c_0.

Шаг 2: для всех сечений H_k, $k = 0...MaxP$ выполняются следующие действия.

Шаг 2.1: если треугольник полностью находится выше плоскости H_k, то он перекрашивается в цвет c_k.

Шаг 2.2: если треугольник пересекается плоскостью H_k, то его необходимо разбить на три новых треугольника и назначить цвет c_k для тех треугольников, которые расположены выше плоскости H_k.

Алгоритм реализован в процедуре `SetNew_Tr` (листинг 14.63).

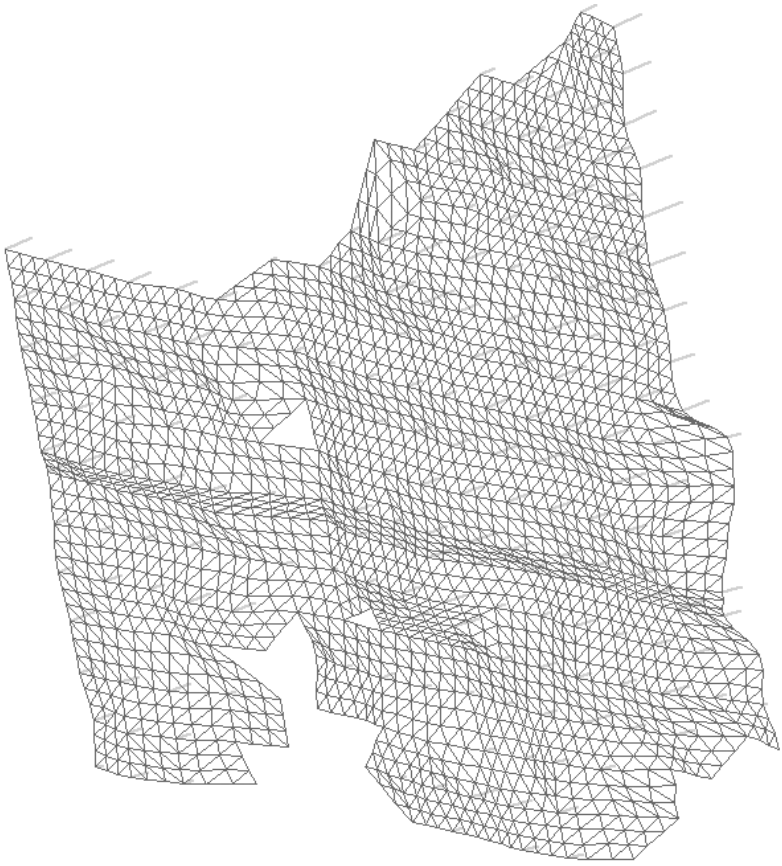


Рис. 14.28. Поверхность, описанная треугольниками

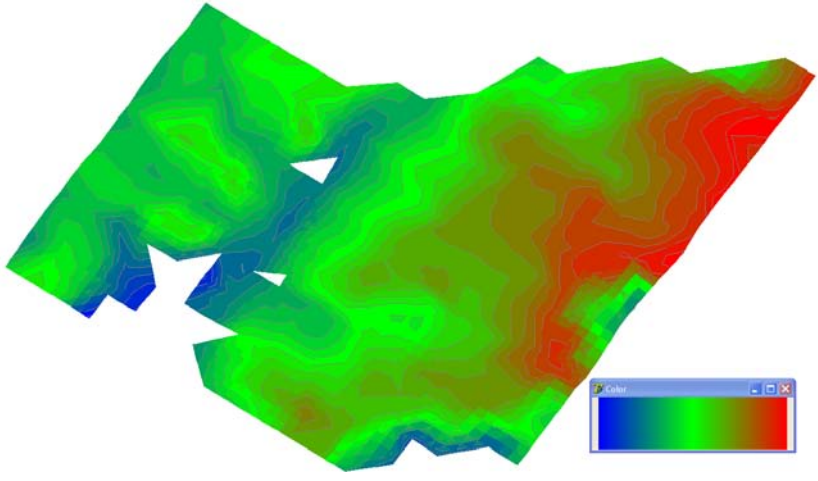


Рис. 14.29. Топографическая раскраска поверхности Сунгарской свиты

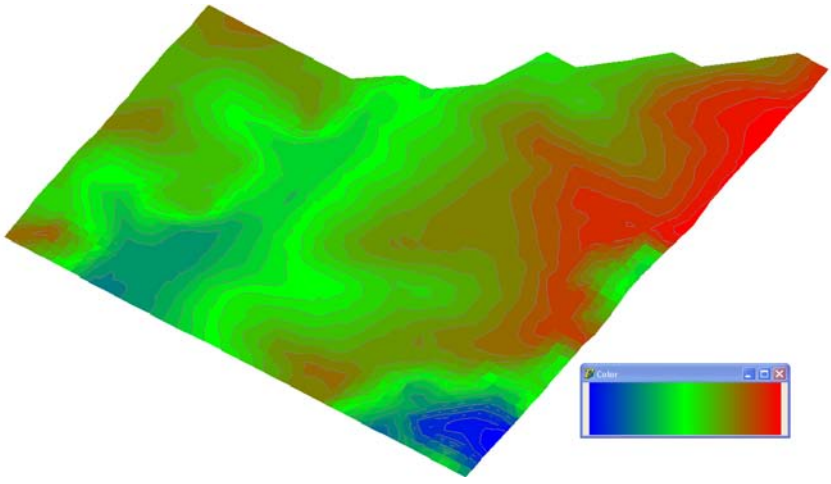


Рис. 14.30. Топографическая раскраска дневной поверхности

Листинг 14.63. Добавление новых треугольников

```
procedure SetNew_Tr(MaxP: integer);
var i,L,j,c,k: integer;
    zp: array[0..2] of real;
    H: real;
    NumLine: byte;
    P1,P2: TXYZ;
begin
  L:=Length(Tr);
  for i:=0 to L-1 do Tr[i].BColor:=SetColor(0,MaxP);
  for k:=0 to MaxP do begin
    H:=z1+k*(z2-z1)/MaxP; c:=SetColor(k,MaxP); i:=-1;
    while i<L-1 do begin
      Inc(i);
      with Tr[i] do begin
        for j:=0 to 2 do zp[j]:=Points[p[j]].z;
        if (zp[0]>=H) and (zp[1]>=H) and
           (zp[2]>=H) then Tr[i].BColor:=c
        else
          if not ((zp[0]<=H) and (zp[1]<=H) and (zp[2]<=H))
             and (NumPointTr(H,Tr[i],NumLine,P1,P2)=2) then
             // добавить треугольники
             AddTr(NumLine,i,c,H,P1,P2);
      end;
    end;
  end;
end;
```

Назначение цвета c_k будем производить в зависимости от значения z, которое меняется от $z_{\min}$ до $z_{\max}$, или в зависимости от k, изменяющегося от 0 до $MaxP$ по следующему алгоритму: доля каждого цвета меняется по кусочно-линейному закону, описываемому двумя отрезками прямых линий. Отрезки опираются на три точки, координаты которых хранятся в массиве `Pc: array[0..2,0..2] of TXY`. Графическое изображение этого алгоритма представлено на рис. 14.31, а реализация в виде функции `SetColor` приводится в листинге 14.64.

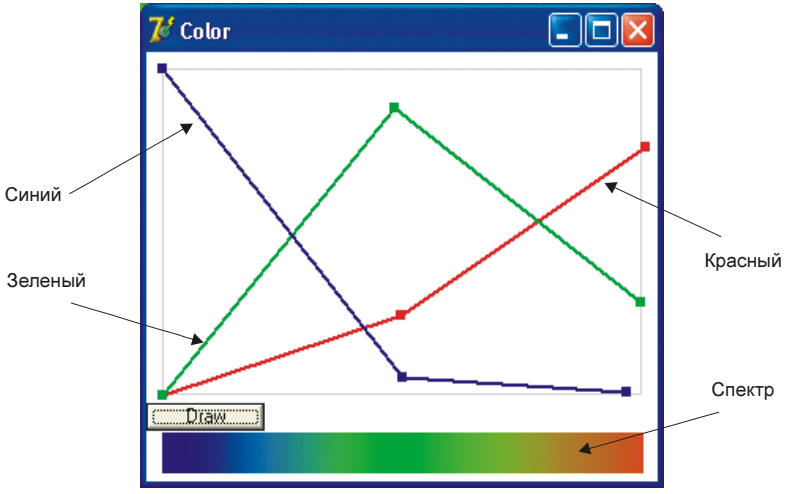


Рис. 14.31. Закон изменения цвета в зависимости от глубины

Листинг 14.64. Вычисление цвета в зависимости от глубины

```
function SetColor(n,MaxP: integer): integer;
var i: integer;
    x,y: single;
begin
  x:=n/MaxP; Result:=0;
  for i:=0 to 2 do
    if x<Pc[i,1].x then
      y:=Pc[i,0].y+(x-Pc[i,0].x)*(Pc[i,1].y-Pc[i,0].y)/
                                 (Pc[i,1].x-Pc[i,0].x)
    else
      y:=Pc[i,1].y+(x-Pc[i,1].x)*(Pc[i,2].y-Pc[i,1].y)/
                                 (Pc[i,2].x-Pc[i,1].x);
    Result:=Result+Trunc(255*y) shl (i*8);
end;
```

При решении задачи пересечения треугольника с плоскостью рассмотрим три случая (рис. 14.32) в зависимости от параметра `NumLine`:

- Отрезок P_1P_2 отсекает нулевую вершину треугольника (F1=0);
- Отрезок P_1P_2 отсекает первую вершину треугольника (F1=1);
- Отрезок P_1P_2 отсекает вторую вершину треугольника (F1=2).

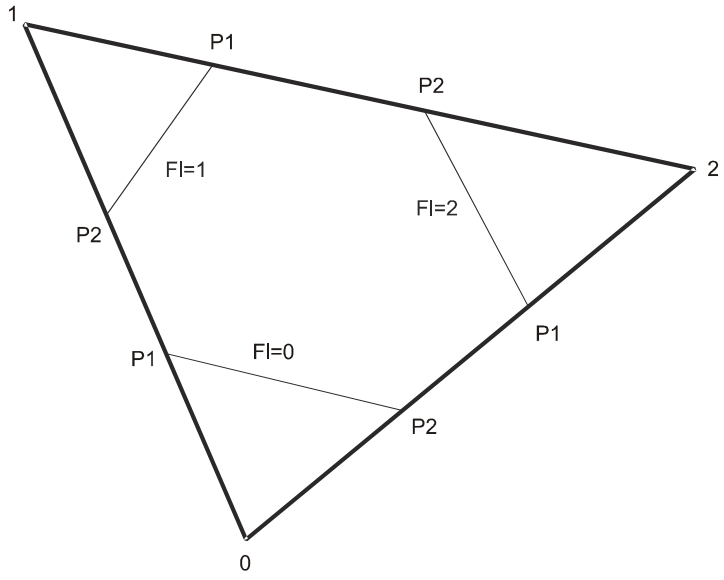


Рис. 14.32. Три случая пересечения треугольника с плоскостью

Число точек пересечений сторон треугольника с плоскостью $z = H_k$ и их координаты определяет функция `NumPointTr` (листинг 14.65). Функция вернет значение `0`, если треугольник с плоскостью не пересекается, и значение `2` в противном случае.

Листинг 14.65. Число точек пересечений сторон треугольника с плоскостью *z*

```
function NumPointTr(H: real; Tr: TTr;
           var NumLine: byte;      // номер стороны
           var P1,P2: TXYZ): byte; // координаты точек пересечения
var T0,T1,T2: TXYZ;
    k0,k1,k2: byte;
begin
  Result:=0;
  with Tr do begin
    k0:=LineToH(p[1],p[2],T0);
    k1:=LineToH(p[0],p[2],T1);
    k2:=LineToH(p[0],p[1],T2);
```

```
    if (k0=1) and (k1<>1) and (k2=1) then begin
      Result:=2; P1:=T0; P2:=T2; NumLine:=1;
    end;
    if (k0=1) and (k1=1) and (k2<>1) then begin
      Result:=2; P1:=T1; P2:=T0; NumLine:=2;
    end;
    if (k0<>1) and (k1=1) and (k2=1) then begin
      Result:=2; P1:=T2; P2:=T1; NumLine:=0;
    end;
  end;
end;
```

Функция `NumPointTr` обращается к функции `LineToH`, решающей более частную задачу: определения числа точек пересечений отрезка с плоскостью и координат этих точек, если они существуют. Текст функции приведен в листинге 14.66. Функция вернет `1`, если есть точка пересечения.

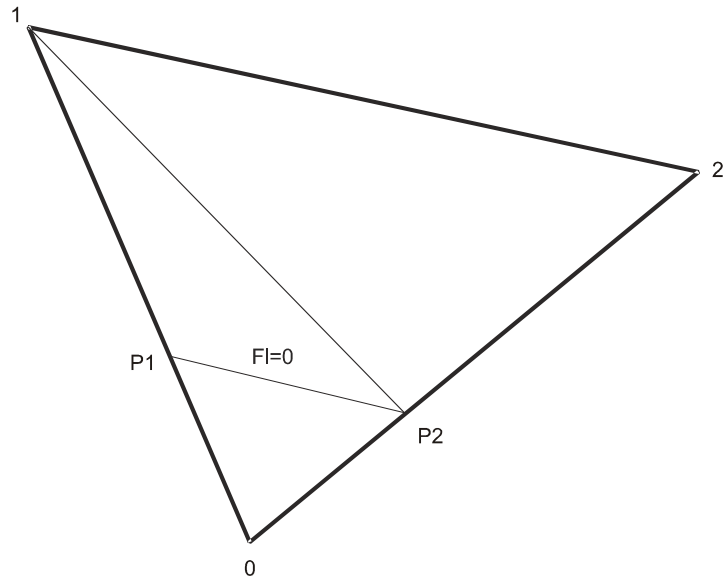


Рис. 14.33. Добавление треугольников

Листинг 14.66. Число точек пересечений отрезка с плоскостью *z*

```
function LineToH(n1,n2: word; var P: TXYZ): byte;
var t,a1,b1,a2,b2: real;
begin
  Result:=0;
  if Abs(Points[n2].z-Points[n1].z)>Eps then begin
    t:=(H-Points[n1].z)/(Points[n2].z-Points[n1].z);
```

```
      if (0<t) and (t<1) then begin
        a1:=Points[n1].x; a2:=Points[n2].x;
        b1:=Points[n1].y; b2:=Points[n2].y;
        P.x:=a1+t*(a2-a1); P.y:=b1+t*(b2-b1); P.z:=H;
        Result:=1;
      end;
    end;
  end;
```

Если треугольник пересекается с плоскостью H_k, то в массив `Points` необходимо добавить новые точки P_1 и P_2, а исходный треугольник разбить на три новых (рис. 14.33).

Процедура `AddTr` (листинг 14.67) добавляет точки P_1 и P_2 и, в зависимости от параметра `NumLine`, добавляет два новых треугольника и перестраивает старый.

Листинг 14.67. Добавление треугольников

```
procedure AddTr(NumLine: byte; i,c: integer; H: real; P1,P2: TXYZ);
var L,Lp: integer;
begin
  L:=Length(Tr); Inc(L,2); SetLength(Tr,L);
  Lp:=Length(Points); Lp:=Lp+2; SetLength(Points,Lp);
  Points[Lp-2]:=P1; Points[Lp-1]:=P2;
  Tr[L-2].BColor:=Tr[i].BColor;
  Tr[L-1].BColor:=Tr[i].BColor;
  // добавляем первый новый треугольник
  Tr[L-2].P[NumLine]:=Lp-2;                    // P1
  Tr[L-2].P[(NumLine+1) mod 3]:=Tr[i].p[(1+NumLine) mod 3];
  Tr[L-2].P[(NumLine+2) mod 3]:=Lp-1;          // P2
  // добавляем второй новый треугольник
  Tr[L-1].P[NumLine]:=Lp-1;                    // P2
  Tr[L-1].P[(NumLine+1) mod 3]:=Tr[i].p[(1+NumLine) mod 3];
  Tr[L-1].P[(NumLine+2) mod 3]:=Tr[i].p[(2+NumLine) mod 3];
  // изменяем номера точек старого треугольника
  Tr[i].P[(NumLine+1) mod 3]:=Lp-2;            // P1
  Tr[i].P[(NumLine+2) mod 3]:=Lp-1;            // P2
  // изменяем цвет треугольников
  if PointFrom[Tr[i].p[NumLine]].Point.z>H
  then Tr[i].BColor:=c
  else begin
    Tr[L-2].BColor:=c; Tr[L-1].BColor:=c;
  end;
end;
```

Полный текст проекта "Изолинии" приведен на компакт-диске в папке **Примеры | Глава 14 | Изолинии**.

Приложения

Приложение 1

Задания для самостоятельной работы

Задания по темам главы 3

Построить рисунки, используя все примитивы канвы (`Canvas`). Каждый рисунок должен содержать 3—4 строки надписей, использовать различные цвета линий и кисти, различные шаблоны кисти. Темы рисунков: человек, велосипед, автомобиль, компьютер, интерьер комнаты, чертеж детали, дом, план квартала, стол, самолет и т. п.

Задания по темам главы 4

Задания по темам "Компонент *Animate*", "Процедуры воспроизведения звуков *Beep*, *MessageBeep* и *PlaySound* "

Литература: [60], стр. 426–440, [102], стр. 61–65.

Перед выполнением задания ознакомиться с соответствием звуков и событий в системе (**Пуск — Панель управления** — **Звук и мультимедиа** — **Звуки**) и с разделом реестра **HKEY_CURRENT_USER — AppEvents — EventLabels**.

1. Использовать в приложении стандартную мультипликацию `FindFolder`, озвучив начало выполнения операции. Собственно процесс должен сопровождаться звуком, который принудительно заканчивается в момент окончания анимации.
2. Использовать в приложении стандартную мультипликацию `FindFile`. По окончании анимации автоматически запускается процесс воспроизведения звукового файла. Собственно процесс должен сопровождаться другим звуком, который принудительно заканчивается в момент окончания анимации.
3. Использовать в приложении стандартную мультипликацию `FindComputer`. Воспроизвести последовательно несколько звуковых файлов, каждый из которых должен звучать определенное время, заданное пользователем.

4. Использовать в приложении стандартную мультипликацию `CopyFiles`. Проиграть последовательно установленные системные события. В начале воспроизведения каждого из них должно выдаваться сообщение.
5. Использовать в приложении стандартную мультипликацию `RecycleFile`. Циклически воспроизводить заданный звук определенное время.
6. Использовать в приложении стандартную мультипликацию `EmptyRecycle`. После заданного числа повторений сделать изображение невидимым и воспроизвести заданное число раз другую анимацию. В промежутке между анимациями воспроизвести звук.
7. Использовать в приложении стандартную мультипликацию `DeleteFiles`. В момент открытия компонента выдать сообщение и трижды проиграть заданную мелодию.
8. Использовать в приложении стандартную мультипликацию `FindFolder`. В момент закрытия компонента выдать звуковое сообщение и перейти к воспроизведению следующей произвольной анимации из файла.
9. Использовать в приложении стандартную мультипликацию `FindFile`, воспроизводя ее, начиная с заданного кадра. В момент начала воспроизведения анимации выдать сообщение и звуковой сигнал.
10. Использовать в приложении стандартную мультипликацию `FindComputer`, воспроизводя ее, начиная с заданного кадра. При окончании воспроизведения выдать сообщение, продублировав его звуковым сигналом
11. Использовать в приложении стандартные мультипликации, каждая из которых воспроизводится определенное время. По окончании воспроизведения каждой из них выдается сообщение и воспроизводится различные звуковые сигналы.
12. Воспроизвести звуковой файл. По окончании воспроизвести стандартную мультипликацию `RecycleFile`. Процесс анимации должен сопровождаться звуком, который принудительно заканчивается в момент ее окончания.
13. Воспроизвести поочередно несколько звуковых и анимационных файлов, так, чтобы окончание одного из них автоматически приводило к вызову другого.
14. Воспроизвести поочередно несколько звуковых и анимационных файлов, так, чтобы пользователь мог задавать время выполнения для каждого из них.
15. Использовать в приложении две панели, на каждой из которых воспроизводится анимация из заданного пользователем файла `AVI`. По прекращению одной анимации автоматически начинает воспроизводиться вторая и наоборот. Предусмотреть кнопку остановки. Проигрывание каждой анимации должно сопровождаться своим звуковым файлом.
16. Разработать приложение с несколькими панелями, в каждой из которых воспроизводится свой файл AVI. Выполнение программы должно сопровождаться воспроизведением звука.

Задания по теме "Компонент TMediaPlayer"

Литература: [60], стр. 440–445, [112], стр. 83–114.

1. Разработать приложение, позволяющее воспроизводить различные файлы `AVI` по выбору пользователя. На главной форме должна присутствовать панель для воспроизведения видео и индикатор, показывающий прошедшее и оставшееся время воспроизведения в процентах.
2. Разработать приложение, последовательно воспроизводящее на форме несколько файлов AVI без участия пользователя. Компонент `MediaPlayer` не должен присутствовать на форме. Каждый файл должен воспроизводиться в течение определенного времени, которое пользователь может задавать с панели.
3. Разработать приложение, выдающее информацию о диске `AudioCD` (общая длительность записи, количество дорожек). Предусмотреть возможность воспроизведения диска. На форме должны присутствовать индикатор, показывающий прошедшее время воспроизведения, и текстовая информация о состоянии устройства.
4. Разработать приложение, для воспроизведения файлов MIDI. Предусмотреть на форме вывод текстовой информации о файле.
5. Разработать приложение для просмотра видео и звуковых файлов.
6. Разработать приложение для воспроизведения видеоинформации. Использовать в приложении две панели, на каждой из которых воспроизводится анимация из заданного пользователем файла AVI. По прекращению воспроизведения одной из них автоматически начинает воспроизводиться вторая и наоборот.
7. Использовать в приложении несколько панелей (до четырех), на каждой из которых воспроизводится анимация из заданного пользователем файла AVI. Предусмотреть варианты синхронного и последовательного воспроизведения файлов на всех панелях (в случае последовательного воспроизведения после завершения одного файла начинает воспроизводиться другой на своей панели).
8. Разработать проигрыватель Audio CD, предусмотрев режим "просмотра", когда каждый трек проигрывается 5–10 секунд, после чего происходит автоматический переход к следующему. По окончании проигрывания всех треков выдается информационное сообщение.
9. Разработать проигрыватель звуковых файлов форматов WAV и MIDI, предусмотрев режим "просмотра", когда каждый файл проигрывается 5–10 секунд, после чего происходит автоматический переход к следующему. По окончании проигрывания всех файлов выдается информационное сообщение.
10. Разработать проигрыватель видеофайлов, предусмотрев для пользователя возможность перемещения вперед и назад на заданное число кадров.

11. Разработать приложение для последовательного воспроизведения разнородных медиафайлов (в форматах MIDI, WAV, AVI). По окончании воспроизведения заданной последовательности предусмотреть выдачу информационного сообщения.
12. Разработать приложение для последовательного воспроизведения звуковых файлов (в форматах MIDI и WAV). Предусмотреть выдачу сообщений при выполнении каждого метода управления. По окончании воспроизведения заданной последовательности файлов предусмотреть выдачу информационного сообщения.
13. Использовать в приложении четыре панели, на каждой из которых воспроизводится анимация из заданного пользователем файла AVI. Воспроизведение начинается на панели, на которую указывает указатель мыши. При переводе указателя мыши на другую панель, воспроизведение начинается на новой панели, а на старой прекращается.
14. Разработать приложение с анимированной озвученной заставкой, которая меняется в зависимости от текущей даты.
15. Анимировать кнопку. Проигрывание начинается при нажатии кнопки и продолжается, пока кнопка не будет отпущена.

Задания по теме "Интерфейс управления мультимедийными устройствами – MCI"

> ***Внимание!***
>
> Задания по теме выполняются ТОЛЬКО с использованием функций интерфейса MCI.

Литература: [60], стр. 13–61, [77], [116].

Для каждого задания необходимо предусмотреть реализацию с использованием интерфейса командных строк (`mciSendString`) и (или) интерфейса командных сообщений (`mciSendCommand`).

Перед выполнением задания ознакомиться с именами устройств MCI, заданными в секции **[mci]** файла **SYSTEM.INI**. Необходимо соответствие расширений мультимедийных файлов именам устройств MCI, заданных в секции [**mci extensions**] файла **SYSTEM.INI**.

1. Разработать проигрыватель для AudioCD, реализовав функции воспроизведения музыки, произвольного выбора дорожки, режим воспроизведения заданной последовательности дорожек, индикацию общего числа дорожек на диске, индикацию процесса воспроизведения, индикацию текущего режима работы устройства.

2. Разработать проигрыватель файлов AVI, реализовав функции: "воспроизведение", "стоп", "пауза", выдачу информации об устройстве, выдачу динамической информации о состоянии устройства, воспроизведение файла в заданных временных пределах, выдачу информации о возможностях устройства, запрещение (разрешение) на ввод или вывод аудиосигнала.
3. Разработать проигрыватель файлов MIDI, реализовав функции: "воспроизведение", "стоп", "пауза", "воспроизведение с заданной позиции".
4. Разработать проигрыватель WAV-файлов, реализовав функции: "воспроизведение", "стоп", "пауза", "воспроизведение с заданной позиции" (по времени), вывод информации об устройстве управления цифровым звуком, управление звуком (вкл./выкл.).
5. Разработать проигрыватель для AudioCD, реализовав функцию воспроизведения музыки, индикацию общего времени записи на диске, позиционирование во временном формате, воспроизведение заданного временного промежутка на диске, индикацию процесса воспроизведения, отключение и включение вывода аудиосигнала (отдельно по правому и левому каналам), индикацию текущего режима работы устройства.
6. Разработать проигрыватель файлов AVI, реализовав функции: "воспроизведение", "стоп", "пауза", выдачу информации об устройстве, выдачу динамической информации о состоянии устройства, воспроизведение файла в заданных пределах (по кадрам), переключение воспроизведения из оконного в полноэкранный режим, режим перемотки изображения до заданной позиции (в прямом и обратном направлении).
7. Разработать проигрыватель файлов MIDI, реализовав функции: "воспроизведение", "стоп", "пауза", "воспроизведение в заданных пределах".
8. Разработать проигрыватель файлов WAV, реализовав функции: "воспроизведение", "стоп", "пауза", "воспроизведение с заданной позиции" (по отсчетам), вывод информации о текущем состоянии устройства с цифровым звуком (`status`), управление звуком (раздельно по каналам).
9. Разработать универсальное устройство для воспроизведения мультимедийной информации — WAV, MIDI, AudioCD.
10. Разработать проигрыватель файлов AVI, реализовав функции: "воспроизведение", "стоп", "пауза", выдачу информации об устройстве, выдачу динамической информации о состоянии устройства, проигрывание файла в заданных пределах (по времени), показ заданного кадра.
11. Разработать проигрыватель файлов MIDI, реализовав функции: "воспроизведение", "стоп", "пауза", "воспроизведение с заданной позиции", выдачу информации состоянии устройства воспроизведения, динамическую индикацию процесса воспроизведения.

12. Разработать записывающее устройство файлов WAV, реализовав функции: "запись", "стоп", "воспроизведение", "пауза", индикацию уровня записи, сохранение записанного файла.
13. Разработать цифровой магнитофон с функциями записи и воспроизведения.
14. Разработать проигрыватель файлов AVI, реализовав функции: "воспроизведение", "стоп", "пауза", выдачу информации об устройстве, выдачу динамической информации о состоянии устройства, проигрывание файла в заданных пределах (по кадрам), запрещение (разрешение) на ввод или вывод аудиосигнала (по отдельным каналам).
15. Разработать проигрыватель файлов MIDI, реализовав функции: "воспроизведение", "стоп", "пауза", вывод информации о состоянии устройства, управление звуком (вкл./выкл.), воспроизведение заданной последовательности файлов MIDI.
16. Разработать записывающее устройство файлов WAV, реализовав функции: "запись", "стоп", "воспроизведение", "пауза", последовательная запись нескольких файлов в один ("склейка" записи).

Задания по теме "Программирование мультимедийных приложений с использованием WinAPI"

> ***Внимание!***
>
> Задания по теме выполняются ТОЛЬКО с использованием функций Win32 API.

Литература: [60], стр. 61–135, [77], стр. 216–247, [40], [68].

1. Определить количество устройств ввода и вывода волновых файлов, установленных в системе и их возможности (при наличии нескольких устройств — определяется для каждого). Учесть наличие в системе основных и дополнительных устройств ввода и вывода.
2. Разработать регулятор баланса для воспроизведения стереосигнала. Необходимо предусмотреть: проверку типа файла (WAV), проверку количества каналов в файле, активизацию регулятора баланса (только при наличии в файле стереозаписи).
3. Разработать программу, представляющую структуру файла WAV в виде дерева с указанием содержимого разделов. Навигация осуществляется визуально либо по заданному смещению. Структура сэмплов в разделе `data` должна соответствовать структуре звукозаписи.

4. Разработать визуальный "проигрыватель" файлов WAV (уровень каждого сэмпла визуально отображается на панели). Предусмотреть отображение уровня для произвольного сэмпла и во время "проигрывания".
5. Реализовать воспроизведение файла WAV. Варианты: использовать функцию `mmio` и (или) стандартные функции управления файлами.
6. Реализовать запись файла WAV.
7. Реализовать редактирование файлов WAV.
8. Разработать программу, представляющую структуру файла MIDI в виде дерева с указанием содержимого разделов. Навигация осуществляется визуально либо по заданному смещению.
9. Реализовать визуальный "проигрыватель" файлов MIDI.
10. Реализовать воспроизведение файла MIDI.
11. Реализовать эмулятор MIDI-клавиатуры.
12. Разработать программу для получения информации о возможностях установленной в системе библиотеки ACM.
13. Разработать программу преобразования формата записи.
14. Разработать программу для работы с файлами формата RIFF (структуру и содержащиеся в ней данные).
15. Разработать программу для записи файла MIDI.
16. Разработать программу для редактирования файлов MIDI.
17. Разработать программу обработки RIFF-файлов с помощью функций ввода/вывода низкого уровня (`mmio`).

Задания по темам главы 6

С помощью свойства `Canvas.Pixels[x,y]` написать процедуры рисования линии методом аппроксимации Брезенхейма линий и дуги окружности или эллипса.

Задания по темам главы 7

1. Написать программу "Солнечная система".
2. Написать программу перемещения изображения по экрану с отражением от стенок.
3. Написать программу управления перемещением рисованного объекта с клавиатуры.

4. Написать программу одновременного управления перемещением двух рисованных объектов с клавиатуры.
5. Написать программу-мультфильм "Восход солнца".
6. Написать программу-мультфильм "Едущий велосипед".
7. Написать программу-мультфильм "Превращение одного объекта в другой".
8. Написать программу-мультфильм "Открывающаяся дверь".
9. Написать программу, постепенно рисующую график функции и выводящую надписи.
10. Написать программу-мультфильм "Колеса паровоза".
11. Написать программу "Полет сквозь звезды".
12. Разработать формат файла для записи векторных шрифтов и программу, редактирующую любой символ и записывающую данные в файл.
13. Написать программу для бинарных операций над трехмерными телами.
14. Написать программу построения пересечения платоновых тел с произвольной плоскостью.
15. Написать программу триангуляции Делоне по произвольному набору точек.
16. Написать программу построения трехмерного тела с цилиндрическими и конусными поверхностями по трем (двум) проекциям.

Задания по темам главы 9

Функция `f(x,y)` задана в узловых точках (`xi=x0+i*h1, yj=y0+j*h2`), `i=0,1,...,n`, `j=0,1,...,m`. Построить один из видов объемной диаграммы функции (см. компонент `PChart`).

Задания по темам главы 11

1. Разработать формат текстового файла для записи команд построения примитивов с помощью процедур модуля `Graphics`. Написать программу, читающую такие файлы.
2. Написать процедуру построения многоугольника со скруглениями радиуса R.
3. Написать программу изображения трехмерных тел с двухточечным проецированием.
4. Написать процедуру пересечения двух произвольных треугольников, заданных в трехмерном пространстве.

5. Написать программу изображения стеклянной бутылки с учетом коэффициента преломления и коэффициентов диффузии и отражения.
6. Написать программу построения тени от произвольного тела, заданного плоскими гранями.
7. Написать программу построения 2D-графика функции, заданной параметрически.
8. Написать программу динамического изменения 2D-графика функции.
9. Написать программу динамического поворота 3D-графика функции.
10. Написать процедуру, выводящую строку векторных символов с коэффициентом увеличения под любым углом.
11. Построить на экране изображения платоновых тел: тетраэдра, куба, октаэдра, икосаэдра, додекаэдра.
12. Написать процедуру построения точки пересечения двух произвольных отрезков.
13. Написать процедуру постепенного превращения одной кривой, заданной набором точек, в другую кривую, заданную другим набором точек.
14. Написать программу, вращающую два платоновых тела вокруг общего центра.
15. Написать программу освещения 3D-поверхности двумя источниками света.

Задания по темам главы 12

1. Построить график функции, заданной параметрическими уравнениями. При выполнении этого проекта предполагается развитый интерфейс, позволяющий изменять масштаб, сдвигать окно на бумаге, менять цвета фона и линий. Также должна быть предусмотрена возможность вывода координаты указателя мыши и параметра t при нажатии на правую кнопку мыши.
2. Создать проект, в котором моделируется траектория движения снаряда при разных начальных углах и скоростях выстрела, возможно, с нелинейностью (сопротивление пропорционально квадрату скорости с некоторым коэффициентом). Траектория движения определяется из задачи Коши для системы обыкновенных дифференциальных уравнений (уравнения движения материальной точки в поле тяжести). Интерфейс проекта должен позволять менять начальный угол траектории, начальную скорость полета, позволять выбирать модель полета и параметры модели.
3. Создать проект для рисования поверхностей второго порядка. Проект должен позволять выбирать в трехмерном пространстве параллелепипед просмотра, перемещать и поворачивать его, изменять масштаб. Поверхность второго порядка может быть выбрана из полного списка таких поверхностей с возможностью поворота, смещения и растягивания по любой оси, или поверхность может быть задана уравнением второго порядка.

4. Создать проект для рисования линий уровня. Этот проект должен позволять вводить функцию двух переменных либо в аналитическом виде, либо в табличном — значениями в узловых точках. Если функция вводится в аналитическом виде, то понадобится интерпретатор, или, в крайнем случае, можно менять исходный текст и перетранслировать его. В случае табличного задания функции придется применять подходящую двумерную интерполяцию. Должна быть предусмотрена возможность изменения прямоугольника проектирования и числа линий уровня. Рекомендуется кроме линий применить подкраску: определенному промежутку изменения значений функции поставить в соответствие некоторый цвет и этим цветом закрасить область между линиями уровня, так, как это делают на топографических картах (чем ниже — тем зеленее, чем выше — тем коричневее).
5. Создать инструмент для вращения многогранников. В проекте необходимо предусмотреть возможность ввода вектора вращения и угловой скорости.
6. Создать инструмент для показа вращения платоновых тел. Для этого проекта необходимо создать свой компонент, определяющий платоновы тела. Проект должен позволять выбирать одно и несколько платоновых тел и для каждого задавать пространственное положение, вектор вращения и угловую скорость. После выбора этих параметров в режиме просмотра тела должны вращаться.
7. Разработать векторный шрифт и инструмент для него. Проект предусматривает разработку формата файла для сохранения символов векторного шрифта и создание инструмента для ввода и редактирования символов мышью.
8. Создать инструмент для плавного превращения одной кривой в другую, изоморфную первой (морфинг). Проект предусматривает ввод двух одинаковых по числу множеств точек на плоскости. Каждому множеству точек соответствует гладкая линия, которая может быть замкнута. В режиме просмотра одна линия должна плавно переходить в другую.
9. Создать инструмент для изображения пересечения многогранника с плоскостью. В этом проекте необходимо предусмотреть возможность ввода плоскости. Результатом работы этого проекта должен быть новый многогранник, полученный в результате пересечения исходного многогранника с плоскостью.
10. Создать проект, реализующий возможность удаления невидимых ребер. В проекте должна быть реализована возможность выбора одного из алгоритмов: алгоритма Сазерленда—Кохена для отсечения отрезка, алгоритма Робертса для удаления невидимых линий, алгоритма Аппеля для удаления невидимых линий.
11. Создать инструмент для вращения многогранника с использованием перспективы. В проекте необходимо предусмотреть возможность выбора вектора вращения, угловой скорости, типа и параметров перспективы.
12. Разработать проект для оптимального раскроя материала. Проект должен поддерживать возможность создания и редактирования двумерных деталей, инст-

рументарий для расстановки этих деталей по заготовке. Проект должен часть оптимизационной работы выполнять автоматически.

13. Визуальный инструмент для работы с семантическими сетями. Кроме стандартного набора функций (добавить / изменить / удалить узел / дугу) проект должен позволять перемещать указателем мыши узлы и дуги, а также должен позволять настраивать цвета и внешний вид графа. Предполагается, что дуга изображается трехзвенной ломаной с линиями, параллельными осям координат.
14. Разработать проект для построения графика любой функции одной переменной (интерпретатор). Проект должен позволять выбирать на плоскости прямоугольник просмотра, перемещать его, изменять масштаб. Уравнение функции задается аналитическим выражением.
15. Тренажер для управляемых ракет. Проект представляет собой игровую программу, в которой по движущейся цели производится выстрел управляемой ракетой. Ракета движется с постоянной скоростью в поле тяжести, а ее управление реализуется по вертикали и горизонтали. Ракета видна из вашей "пусковой установки".
16. Сплайны 2-го порядка. Функция двух переменных задается своими значениями в узлах прямоугольной сетки. Проект должен позволять вводить с помощью мыши и клавиатуры значение функции в узлах, а затем решать задачу интерполяции двумерными сплайнами.
17. Программа для работы со сплайнами: кубические сплайны, кривые Безье, B-сплайны, beta-сплайны. В этой программе должен быть предусмотрен ввод, редактирование и удаление точек, а также выбор одного из методов.
18. Триангуляция плоских фигур. Программа должна позволять вводить плоскую многосвязную фигуру, ограниченную прямыми линиями или кривыми, а затем проводить разбиение этой области на треугольники. После автоматического разбиения должна быть предоставлена возможность просмотра с масштабированием и ручного редактирования триангуляции.
19. Создать программу разработки проекта газификации домов. Проект должен поддерживать возможность создания и редактирования двумерного плана дома и газовой трассы, создания и редактирования элементов газового оборудования, инструментарий для расстановки этих элементов по плану дома и газовой трассы.
20. Создать программу разработки проекта трехмерных локальных вычислительных сетей (ЛВС). Проект должен поддерживать возможность создания и редактирования трехмерного плана здания, создание и редактирования элементов ЛВС, инструментарий для расстановки этих элементов на плане здания.
21. Создать программу разработки многослойной карты города. Этот проект должен предоставлять возможность создания и редактирования слоя карты города. Необходимо поддерживать библиотеку условных обозначений и элементов.

В проекте должны быть реализованы функции масштабирования, рисования и удаления примитивов и элементов.

22. Создать программу разработки эскизов декораций. Эта программа должна создавать изображение из нескольких картинок. Каждая картинка может быть перемещена, у нее могут быть изменены размеры и один из цветов назначен прозрачным. Должны работать эффекты затемнения и освещения прожекторами разных цветов.
23. Разработать программу, в которой можно вырезать "ножницами" фрагмент одного изображения и поместить вырезанный фрагмент в другое изображение. В качестве основы можно взять графический редактор, описанный в *главе 7*, и добавить вырезание "ножницами".
24. Создать проект для рисования эскиза расстановки мебели. Проект должен поддерживать возможность создания и редактирования трехмерного плана квартиры, создания и редактирования эскиза мебели, а также инструментарий для расстановки мебели по квартире.
25. Разработать проект для рисования фракталов.
26. Создать программу, моделирующую алгоритм Брезенхейма для рисования отрезка и демонстрирующий работу этого алгоритма в увеличенном виде на экране.
27. Удаление невидимых граней. За основу этого проекта также можно взять пример, рассмотренный в *главе 10*, позволяющий редактировать многогранники. В модифицированном проекте необходимо добавить возможность удаления невидимых граней при выводе трехмерного изображения.
28. Программа макетирования визитных карточек. Проект должен поддерживать возможность создания и редактирования макета визитных карточек: менять текст и расположение надписей, подключать картинки и фотографии, выводить подготовленный макет на печать.

Приложение 2

Описание прилагаемого компакт-диска

На компакт-диске в 13-ти папках находятся 45 проектов.

Проекты находятся в папках, названных в соответствии с нумерацией глав, к которым они относятся: Глава 1, Глава 2, …, Глава 14 (см. табл. П2.1).

Таблица П2.1. *Структура компакт-диска*

Папка	Описание проекта
Глава 1 | Кривые Лиссажу	Простой проект для рисования кривых Лиссажу
Глава 2 | Цветовые модели	Проект для демонстрации работы с различными цветовыми моделями
Глава 2 | Данные из текстового файла	Чтение данных для рисования из текстового файла
Глава 2 | Вывод строки	Вывод строки под углом
Глава 2 | На экране	Рисование на экране с помощью API-функций
Глава 4 | Использование компонента Animate	Воспроизведение стандартных мультипликаций. Проекты демонстрируют основные методы работы с мультимедийными компонентами системы Delphi
Глава 4 | Консольное выполнение команд MCI	Проект, позволяющий в консольном режиме выполнять любые команды интерфейса MCI
Глава 4 | Проигрыватель аудио-CD	Проект реализует функциональность проигрывателя компакт-дисков с использованием команд MCI
Глава 4 | Запись звука с использованием команд MCI	Проект реализует функциональность магнитофона с использованием команд MCI
Глава 4 | Низкоуровневое чтение файла	В проекте осуществляется чтение звукового файла формата WAVE с использованием функций и структур WinApi
Глава 4 | Низкоуровневое воспроизведение файла	Проект реализует функциональность магнитофона с использованием функций WinAPI

Таблица П2.1 *(продолжение)*

Папка	Описание проекта
Глава 5 | Chart	Проект, демонстрирующий работу с диаграммами
Глава 6 | Алгоритмы Брезенхейма	Проект, реализующий алгоритмы Брезенхейма
Глава 6 | Проекции	Приложение для построения проекций
Глава 7 | View JPG	Приложение для просмотра графических файлов
Глава 7 | Сортировка	Мультипликация сортировки
Глава 7 | Морфинг	Превращение стула в стол
Глава 7 | Падение мяча	Мультипликация с шаром
Глава 7 | Velo	Мультипликация с велосипедом
Глава 7 | Произвольный стиль линии	Проект для рисования линии произвольным стилем
Глава 7 | Деформация BMP	Деформация прямоугольных изображений
Глава 7 | Pain	Приложение — простой растровый редактор
Глава 7 | LVS	Проектирование плоских схем
Глава 7 | Графы	Проектирование графов
Глава 7 | Газификация	Проектирование газификации домов
Глава 8 | MyCorel	Приложение — векторный редактор
Глава 9 | 2D	Проект для построения графика функции одной переменной
Глава 9 | 3D-графика	Проект для построения графика функции двух переменных
Глава 9 | Интерполяция	Проект для построения интерполяционных кривых
Глава 9 | Параметрические кривые	Проект для построения трансцендентных кривых, заданных параметрически
Глава 9 | Интерпретатор	Приложение для построения графика функции, введенной в виде строки
Глава 10 | EditReport	Проект для создания шаблонов отчетов
Глава 11 | Платоновы тела	Проект для рисования платоновых тел
Глава 11 | Квадратичные поверхности	Проект для изображения квадратичных поверхностей
Глава 11 | Проекции	"Восстановление трехмерного тела по проекциям"
Глава 11 | Бинарные операции	Проект, демонстрирующий бинарные операции
Глава 12 | 3D	Редактор трехмерных тел
Глава 12 | 3D Edition	Редактор топологически связанных трехмерных тел

Таблица П2.1 *(окончание)*

Папка	**Описание проекта**
Глава 13 | 2D OpenGL	Проект для рисования двумерных примитивов OpenGL
Глава 13 | 3D OpenGL	Проект для рисования трехмерных объектов с помощью OpenGL
Глава 13 | EditBody	Редактор трехмерных тел, реализованный на основе OpenGL
Глава 14 | Триангуляция	Проект для построения триангуляции
Глава 14 | Автопостроение слоев	Проект для автоматического построения слоев
Глава 14 | Сглаживание поверхности	Проект для построения сглаживающей триангуляции
Глава 14 | Изолинии	Проект для построения изолиний поверхности

Список литературы

1. Adobe Systems Inc. PostScript Language Reference Manual. — Addison-Wesley, Reading, Mass., 1986.
2. Adrian Nye. X Window System User's Guide, volume 3of TheDefinitive Guides to the X Window System. — O'Reilly & Associates, Sebastopol, Ca., 1987.
3. Bresenham J., A Linear Algoritm for Incremental Didgital Display of Circular Arcs. — CACM, vol. 20, pp. 100—106, 1977.
4. Bresenham J. E. Algorithm for computer control of a digital plotter. — IBM Systems Journal, vol. 4, N. 1, pp. 25—30, 1965.
5. Bui-Tuong Phong. Illumination for Computer-Generated Pictures. — Communication of the ASM, 18(6), pp. 311—317, June 1975.
6. Clark, J. H. A VLSI geometry Processor for Graphics. — IEEE Computer, 12(7).
7. Cyrus M., Beck J. Generalized two- and threedimensional clipping. — Computer and Graphics, Vol. 3, pp. 23—28, 1978.
8. Encarnacao J. Einfurung in die Graphische Datenverarbeiterung. — Eurographics '89. Tutorial Notes 1. Hamburg, FRG, 1989.
9. Fontenier Guy, Pascal Gros Pascal. Architectures of Graphic Processors for Interactive 2D Graphics. — Computer Graphics Forum 7, 1988.
10. Hans Joseph, Max Mehl. Computer Graphics Hardware: Introduction and State of the Art. — Eurographics '91. Tutorial Note 9. Viena, 1991.
11. http://cgm.graphicon.ru:8080/ — Графика и мультимедиа. Научно-образовательный сетевой журнал о компьютерной графике, машинном зрении и обработке изображений.
12. http://gamemaker.webservis.ru/
13. http://graphics.cs.msu.su/courses/cg/library/ — все, что связано с компьютерной графикой, обработкой изображений и мультимедиа. Сайт поддерживается сотрудниками и аспирантами лаборатории компьютерной графики и мультимедиа при факультете ВМиК МГУ.
14. http://propro.ru/go/gallery.html — Автор — Геннадий Обухов.
15. http://users.cybercity.dk/~bbl6194/delphi3dx.htm
16. http://www.artist-3d.com — Free 3D models, 3ds max (англ.) — подборка бесплатных 3D моделей.
17. http://www.b3d.mezon.ru — Blender3D — Сайт по пакету Blender.
18. http://www.delphi-jedi.org/DelphiGraphics/OpenGL/OpenGL.zip — Альтернативный заголовочный файл opengl.pas. Автор — Mike Lischke.

19. http://www.delphikingdom.com/ — Сайт "Королевство Delphi", на котором есть раздел с примерами по использованию OpenGL.
20. http://www.delphisources.ru/pages/sources_list.html — Графика и мультимедиа.
21. http://www.fantasyartdesign.com — Digital Fantasy art 3d wallpapers (англ.) — Коллекция трехмерной графики международных дизайнеров.
22. http://www.freeartsoftware.com — Free 3D design, 2D art software (англ.) — Подборка бесплатных 3D и 2D компьютерных программ.
23. http://www.gamedeveloper.org/delphi3d — Автор — Tom Nuydens. — Содержит пакет CgLib, примеры и документацию. Здесь можно получить заголовочный файл для использования библиотеки GLUT.
24. http://www.ispu.ru/library/math/sem1/index.html — Интерактивный компьютерный учебник по высшей математике.
25. http://www.lischke-online.de —Автор — Mike Lischke. Содержит Opener, программу просмотра 3DS-файлов, а также пакет GLScene.
26. http://www.neosurrealismart.com — Коллекция трехмерной графики.
27. http://www.opengl.org — Начальное знакомство с библиотекой OpenGL.
28. http://www.p-m.org/delphi/
29. http://www.render.ru — Онлайн-журнал, посвященный трехмерной графике и анимации.
30. http://www.scitechsoft.com — Библиотека программирования графики SciTech MGL.
31. http://www.sgl.com/software/opengl — Курсы программирования для OpenGL. Альтернативная версия OpenGL.
32. http://www.signsoft.com/downloadcenter/index.html — Набор компонентов VisIt.
33. http://www.softimage.ru — Русскоязычное сообщество пользователей пакета Softimage|XSI.
34. http://www.torry.ru/samples/saniples/primscr.zip — Пример использования OpenGL в Delphi и модуль DGLUT.pas.
35. http://www.torry.ru/vcl/mmedia/ogl.zip — Автор — Enzo Piombo. Редактор на основе компонента TOpenGL.
36. http://www.torry.ru/vcl/mmedia/ogld lO.zip — Заголовочные файлы gl.pas и glu.pas. Автор — Alexander Staubo: http ://home.powertech. no/alex/
37. http://wwwl.math.luc.edu/~jlayous/opengl/index.html
38. International Standards Organization. International standard information processing systems — computer graphics — graphical kernel system for three dimensions (GKS-3D) functional description. — Technical Report ISO Document Number 9905:1988(E), American National Standards Institute, New York, 1988.

39. Martti Mantyla. An Introduction to Solid Modeling. — Computer Science Press, 1988.

40. Microsoft, "Multimedia Programmer's Reference for the Microsoft Windows Operating System". — Microsoft Windows Software Development Kit, 1992.

41. Pinkman R., Novak M., Guttag K. Video-RAM exels at fast graphics. — Electronics Design, pp. 161—171, August 1983.

42. Seidel H. P. PC Graphics Hardware. — Eurographics '88. Tutorial/Cours 8, France, Nice, September 1988.

43. Smit A. R., Tint Fill. SIGGRAPH '79 Proceedings. — Computer Graphics, Vol. 13(2), pp. 276—283, 1979.

44. Sobkow Mark S., Pospisil Paul, Yang Yee-Hong. A Fast Two-Dimensional Line Clipping Algoritm via Line Encoding. — Computer & Graphics, Vol. 11, N. 4, pp. 459—467, 1987.

45. Sproull Robert F., Sutherland Ivan E. A Clipping Divider. — AFIP Fall Joint Computer Conference, San Francisco, 1968.

46. Stevenson Jeff. PEXlib specification and C language binding, version 5.1P. — The X Resource, Special issue B, September 1992.

47. Stralunsfreier Flacbildschirm. MC, Die MikrocomputerZeitschrift. N. 8, 1989.

48. Strauss Paul S., Carey R. An object-oriented 3D graphics toolkit. — In Proceeding of SIGGRAPH '92, pp. 341-349, 1992.

49. Sutherland I. E., Hodgman G.W. Reentrant Polygon Clipping. — Communications of the ACM, 17(1), pp. 32-42.

50. Tina M. Nicholl, D.T.Lee and Robin A. Nicholl. An efficient new algoritm for 2-D line clipping: its development and analysis. — Computer Graphics, Vol. 21, N. 4, pp. 253—262, July 1987.

51. Weiler K., Atherton P. Hidden Surface Removal Using Polygon Area Sorting. — SIGGGRAPH'77 Proceedings, Computer Graphics, Vol. 11, N. 2, pp. 214—222, 1977.

52. Weiler K., Polygon Comparision Using a Graph Representation // SIGGGRAPH'80 Proceedings, Computer Graphics, Vol. 14, pp. 10—18, 1980.

53. Wiegand G., Covey B. HOOPS Reference Manual, Version 3.0. — Ithaca Software, 1991.

54. Womack Paula, ed. PEX protocol specification and encoding, version 5.1P. — The X Resource, Special Issue A, May 1992.

55. You-Dong Liang and Brian A. Barsky. A new concept and method for line clipping. — ACM Transaction on Graphics, Vol. 3, N. 1, pp. 1—22, January 1984.

56. Абраш Майкл. Программирование графики. Таинства. — Киев: EvroSYB, 1995.

57. Адамс Джим. DirectX: продвинутая анимация. — М.: КУДИЦ-ОБРАЗ, 2004.

58. Аммерал Л. Принципы программирования в машинной графике: монография. —М.: Сол Систем, 1992.
59. Антонофф М., Линдерхолм О. Лазерные принтеры. — Компьютер Пресс, сборник N 1, с. 3—8, 1989.
60. Архангельский А. Я. Программирование в Delphi 6. — М.: Бином, 2002.
61. Боресков А. В. Графика трехмерной компьютерной игры на основе OpenGL. — М.: Диалог-МИФИ, 2004.
62. Боресков А. В. Расширение OpenGL. — СПб.: БХВ-Петербург, 2005.
63. Вельтмандер П. В. Основные алгоритмы компьютерной графики: Учебное пособие в 3-х книгах. — Новосибирский государственный университет, 1997.
64. Верма Р. Д. Введение в Open GL. Горячая линия — Телеком, 2004.
65. Вирт Н. Алгоритмы + структуры данных = программы. — М.: Мир, 1985.
66. Гилой В. Интерактивная машинная графика. — М.: Мир, 1981.
67. Гончаров Д. DirectX 7.0 для программистов: Учебный курс. — СПб.: Питер, 2001.
68. Гордеев О. Программирование звука в Windows. — СПб.: БХВ-Петербург, 2000.
69. Горнаков С. Г. Инструментальные средства программирования и отладки шейдеров в DirectX и OpenGL. Профессиональное программирование. — СПб.: БХВ-Петербург, 2005.
70. Дарахвелидзе П. Delphi — среда визуального программирования. — СПб.: БХВ-Петербург, 1995.
71. Джамбруно Марк. Трехмерная графика и анимация. — М.: Издательский дом "Вильямс", 2002.
72. Дональд Херн, М. Паулин Бейкер. Компьютерная графика и стандарт OpenGL. — М.: Издательский дом "Вильямс", 2005.
73. Евченко А. И. OpenGL и DirectX. Программирование графики. — СПб.: Питер, 2006.
74. Иванов В. П., Батраков А. С. Трехмерная компьютерная графика. — М.: Радио и связь, 1995.
75. Кинтцель Т. Программирование звука на ПК. — М.: ДМК Пресс, 2005.
76. Климов А. С. Форматы графических файлов. — Киев: НИПФ "ДиаСофт Лтд.", 1995.
77. Комягин В. Б. Программирование мультимедиа-приложений. — М.: ЭКОМ, 1995.
78. Корриган Джон. Компьютерная графика: Секреты и решения. — М.: Энтроп, 1995.

79. Краснов М. Direct X: Графика в проектах Delphi. — СПб.: БХВ-Петербург, 2003.

80. Краснов М. OpenGL. Графика в проектах DELPHI. — СПб.: БХВ-Петербург, 2004.

81. Кречко Ю. А. Автокад: Курс практической работы. — М.: ДИАЛОГ-МИФИ, 1994.

82. Маров М. Н. Энциклопедия 3D Studio Max 6. — СПб.: Питер, 2006.

83. Мартинес Ф. Синтез изображений. Принципы, аппаратное и программное обеспечение. — М.: Радио и связь, 1990.

84. Математика и САПР. Кн. 1. Основные методы. Теория полюсов: В 2-х книгах. — М.: Мир, 1988.

85. Миллер Том. DirectX 9 с управляемым кодом. Программирование игр и графика. — М.: КомБук, 2005.

86. Мюррей Джеймс Д. Энциклопедия форматов графических файлов. — Киев: БХВ-Петербург, 1997.

87. Никулин Е. А. Компьютерная геометрия и алгоритмы машинной графики. — СПб: БХВ-Петербург, 2005.

88. Ньюмен У., Спрулл Р. Основы интерактивной машинной графики. — М.: Мир, 1976.

89. OpenGL: Официальный справочник. — М.: СПб.: Киев: DiaSoft, 2002.

90. OpenGL: руководство по программированию. — СПб.: Питер, 2006.

91. Павлидис Т. Алгоритмы машинной графики и обработки изображений. — М.: Радио и связь, 1986.

92. Петзолд Ч. Программирование для Windows 95 в 2-х томах. — СПб.: БХВ-Петербург, 1997.

93. Печатающие устройства персональных ЭВМ: Справочник. — М.: Радио и связь, 1992.

94. Поляков А. Ю. Программирование графики. GDI+ и DirectX. — БХВ-Петербург, 2005.

95. Прэтт У. Цифровая обработка изображений. — М.: Мир, 1982.

96. Райт Р. OpenGL. Суперкнига. — М.: Издательский дом "Вильямс", 2006.

97. Роджерс Д. Алгоритмические основы машинной графики. — М.: Мир, 1989.

98. Роджерс Д., Адамс Д. Математические основы машинной графики. — М.: Мир, 2001.

99. Романов В. Ю. Популярные форматы файлов для хранения графических изображений на IBM PC. — М.: Унитех, 1992.

100. Рост Рэнди Дж. OpenGL: трехмерная графика и язык программирования шейдеров. — СПб.: Питер, 2005.

101. Сван Том. Форматы файлов Windows. — М.: БИНОМ, 1995.

102. Секунов Н. Обработка звука на PC. — СПб.: БХВ-Петербург, 2001.

103. Скворцов А. В., Мирза Н. С. Алгоритмы построения и анализа триангуляции. — Томск: Изд-во Томского университета, 2006.

104. Справочник по машинной графике в проектировании. — Киев: Будівельник, 1984.

105. Тихомиров Ю. Программирование трехмерной графики. — СПб.: БХВ-Петербург, 2000.

106. Томпсон Найджел. Секреты программирования трехмерной графики для Windows 95. — СПб.: Питер, 1997.

107. Трухильо Стен. Графика для Windows средствами DirectDraw. — СПб.: Питер, 1998.

108. Тюкачев Н. Введение в компьютерную графику. — Воронеж: изд-во ВГТУ, 1996.

109. Тюкачев Н. А. Программирование в Delphi для начинающих: учебное пособие для студентов вузов. — СПб.: БХВ-Петербург, 2007.

110. Тюкачев Н. А. Сглаживание триангуляции трехмерных поверхностей. Информатика: проблемы, методология, технологии: материалы 6 международной научно-методической конференции. — Воронеж: ВГУ, 2006.

111. Тюкачев Н. А. Структура данных для топологически связанных трехмерных тел. Вестник ВГУ. Серия: Системный анализ и информационные технологии, № 1, с. 141—144, 2006.

112. Тюкачев Н. А., Свиридов Ю. Т. Delhpi 5. Создание мультимедийных приложений. — М.: Нолидж, 2000.

113. Тюкачев Н. А., Свиридов Ю. Т. Delhpi 5. Создание мультимедийных приложений: Учебный курс. — СПб.: Питер, 2001.

114. Фоли Дж., Вэн Дэм А. Основы интерактивной машинной графики в 2-х книгах. — М.: Мир, 1985.

115. Фостер Дж. Обработка списков. — М.: Мир, 1974.

116. Фролов А. В. Мультимедиа для WINDOWS: Руководство для программиста. — М.: ДИАЛОГ МИФИ, 1994.

117. Фролов А. В. Программирование видеоадаптеров CGA, EGA и VGA. — М.: ДИАЛОГ-МИФИ, 1994.

118. Хокс Барри. Автоматизированное проектирование и производство. — М.: Мир, 1991.

119. Хоу Артур. LightWave 3D 8. — М.: NT Press, 2004.

120. Шикин Е. В., Боресков А. В. Компьютерная графика. Полигональные модели. — М.: ДИАЛОГ-МИФИ, 2005.

121. Шикин Е. В., Боресков А. В. Компьютерная графика: Динамика, реалистические изображения. — М.: ДИАЛОГ-МИФИ, 1995.

122. Шикин Е. В., Боресков А. В., Зайцев А. А. Начала компьютерной графики. — М.: Диалог-МИФИ, 1993.

123. Шлихт Ганс Юрген. Цифровая обработка цветных изображений. — М.: ЭКОМ, 1997.

124. Эйнджел Эдвард. Интерактивная компьютерная графика: Вводный курс на базе OpenGL. — М.: Издательский дом "Вильямс", 2001.

125. Юань Фень. Программирование графики для Windows. — СПб.: Питер, 2002.

126. Янг Майкл. Программирование графики в Windows 95: Векторная графика на языке C+. — М.: БИНОМ, 1997.

127. Яншин В. В., Калинин Г. А. Обработка изображений на языке Си для IBM PC. — М.: Мир, 1994.

Предметный указатель

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W

А

Б

В

Г

Д

Ж

З

И

К

Л

М

Н

О

П

Р

С

Т

У

Ф

Ц

Ч, Ш

Э

Я