Михаил Краснов

bhv
www.bhv.ru
www.bhv.kiev.ua

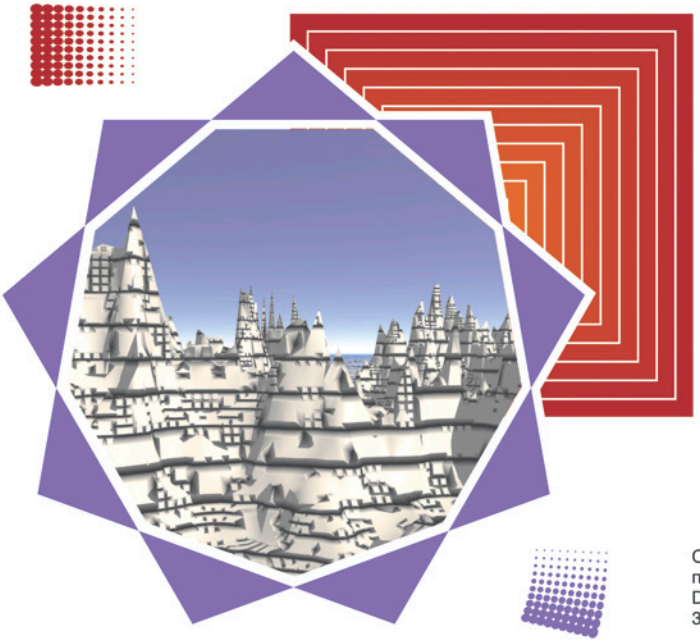
DirectX

Графика в проектах Delphi

МАСТЕР

ПРАКТИЧЕСКОЕ РУКОВОДСТВО

- Библиотеки DirectDraw и Direct3D
- Программирование игр
- Двумерная и трехмерная графика



CD-ROM содержит примеры проектов, дистрибутив ядра DirectX 8.0a и демо-версию 3D Exploration

Михаил Краснов

DirectX.

Графика в проектах *Delphi*

Дюссельдорф ◆ Киев ◆ Москва ◆ Санкт-Петербург

УДК 681.3.06

Книга посвящена использованию модулей DirectX в приложениях, разрабатываемых в Delphi. Начиная с простых примеров, последовательно и подробно рассматривается создание объектов двумерной и трехмерной графики, визуальные и цветовые эффекты, а также обсуждаются дополнительные темы, такие как быстрая работа с устройствами ввода. Большое внимание уделяется вопросам оптимизации и ускорения работы приложений. Книга содержит практические решения проблем, возникающих при программировании игр и других приложений, нуждающихся в высокой скорости вывода графики в среде Windows. Прилагается компакт-диск с инструментальными средствами, кодами и демонстрационными версиями рассматриваемых примеров.

Для широкого круга программистов

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Наталья Таркова*
Редактор	*Анна Кузьмина*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Краснов М. В.
DirectX. Графика в проектах Delphi. — СПб.: БХВ-Петербург, 2001. — 416 с.: ил.

ISBN 5-94157-033-3

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 06.06.01.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 33,54.
Тираж 4000 экз. Заказ
"БХВ-Петербург", 198005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар, № 77.99.1.953.П.950.3.99
от 01.03.1999 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН.
199034, Санкт-Петербург, 9-я линия, 12.

Содержание

Введение

Главной темой книги, которую вы держите в руках, является компьютерная графика, а именно использование в Delphi модулей DirectX, связанных с двумерной и трехмерной графикой.

DirectX — это набор драйверов, образующий интерфейс между программами в среде Windows и аппаратными средствами. Состоит он из набора компонентов, поддерживающих непосредственную работу с устройствами, и служит в качестве средства разработки быстродействующих мультимедийных приложений. Для программиста применение DirectX заключается в использовании набора низкоуровневых интерфейсов (API).

Развитие DirectX происходит беспрерывно и корпорация Microsoft ежегодно выпускает новую или обновленную версию этого продукта. Очередная версия включает в себя возможности предыдущих, но некоторые предлагают подходы, кардинально отличающиеся от концепций ранних версий. Так, в восьмой версии не произошло обновления модуля, связываемого с двумерной графикой, и разработчикам предложено использовать объединенный подход к графике, в котором чистая двумерная графика является частным случаем трехмерной. В этой версии единый набор API обслуживает оба подраздела компьютерной графики.

В данной книге обсуждается API седьмой и восьмой версий DirectX. В начале в ней изложено применение модуля DirectDraw для создания приложений чистой двумерной графики. DirectDraw используется как набор интерфейсов седьмой версии DirectX. Во второй части книги рассматривается компонент DirectX Graphics, как набор интерфейсов восьмой версии.

Читателю не стоит относиться к материалу книги о DirectDraw, как к устаревшему анахронизму. Во-первых, при работе с этим модулем приложения двумерной графики гарантированно используют возможности видеокарт, поддерживающих только 2D-акселерацию, а пользователей, имеющих именно такие карты, в ближайшие годы будет оставаться еще очень много.

Во-вторых, в планах разработчиков корпорации Microsoft было заявлено о том, что в девятой версии DirectX будет возобновлена поддержка DirectDraw, как обновленных интерфейсов, и этот материал наверняка будет легко "приспособить" также к новой версии. И, в-третьих, вы получите здесь представление о базовых механизмах и приемах, лежащих в основе и трехмерной графики, при переходе к которой вы встретите уже знакомые вам принципы и подходы.

Теперь я должен сказать несколько важных вещей непосредственно о книге.

Прежде всего, хочу предупредить, что книга не охватывает целиком ни DirectX, ни даже модули, относящиеся напрямую к графике. Материал чрезвычайно обширен, чтобы охватить его одним изданием, поэтому я вам не могу обещать, что вы после знакомства с моей книгой будете уметь абсолютно все, но я обещаю, что вы научитесь очень многому.

Как мне кажется, большинство читателей купят книгу в расчете на то, чтобы с ее помощью научиться создавать игры. Вы найдете в ней примеры простых игр, которые не стоит рассматривать как профессиональные. Но, познакомившись с ними, вы сможете написать и более масштабные проекты. Думаю, что книга может оказаться полезной и тем, кто не собирается программировать игры, а нуждается в средстве, позволяющем максимально быстро построить, например, диаграмму или график.

Не утверждаю, что я открыл новый жанр, но должен предупредить вас, что эта книга может показаться вам своеобразной: главный упор в ней делается на практические примеры. Среди прочитанных мною изданий по программированию самыми полезными оказались те, которые содержат не пространные рассуждения и сложные диаграммы, а те, где предлагаются готовые решения и масса примеров. Поэтому и здесь я постарался выдержать изложение в том же духе в надежде, что она принесет вам действительную пользу. В книге масса примеров, многие из которых не стоит рассматривать слишком бегло, иначе в какой-то момент вы можете потерять нить понимания. По мере неспешного ознакомления с примерами в каждом из них попробуйте что-то изменить или добавить, и тогда у вас появится ощущение полного понимания.

Эта книга и примеры к ней также очень сильно отличаются от обычных книг по программированию на Delphi. Например, здесь нет обзора компонентов, в большинстве примеров на форме располагается один компонент, а код модулей может вам поначалу показаться непривычно длинным и странным по синтаксису.

Одна из целей, которую я преследовал, состоит в том, чтобы книга читалась легко теми, кто впервые сталкивается с данной темой. Но я должен предупредить, что если вы программируете на Delphi меньше года, вам, возможно, будет очень тяжело изучать эту книгу. Вы должны хорошо знать Delphi, причем подразумевается не умение ориентироваться в палитре компонентов,

а наличие опыта в кодировании. Минимальный уровень, который вы должны иметь, чтобы приступать к чтению этой книги, таков: читатель должен свободно владеть навыками работы с невизуальными объектами, такими как объекты класса `TBitmap`. Если вы можете с помощью подобного объекта вывести на форме содержимое растрового файла, то, я надеюсь, сможете легко и быстро разобраться в материале книги.

Наверняка вы наслышаны о DirectX, и его существование не стало для вас откровением, пришедшим в вашу жизнь с этой книгой. Вы знаете, что данное средство предназначено для создания мультимедийных приложений, работающих максимально быстро, и у вас, наверное, нет вопроса ко мне, почему я написал книгу об использовании DirectX. Но, скорее всего, мне необходимо упомянуть, почему для освещения этой темы мною выбрана среда программирования Delphi. Ведь если в ней и написаны масштабные игры профессионального уровня, то их очень немного. Программисты, знающие среду Delphi поверхностно, несправедливо считают, что с ее помощью можно создавать только СУБД. А между тем, это очень мощное средство, которое годится для решения достаточно широкого круга задач. Прочитав книгу, вы убедитесь в этом. Delphi — очень популярная среда программирования, о которой разработчики DirectX позаботились не в первую очередь: для программистов, использующих C++ или Visual Basic, имеется богатый источник информации по разработке программ, комплект документации и примеров, SDK; для программистов же, использующих Delphi, таких источников информации мало. Чтобы помочь именно этой огромной армии программистов и написана данная книга. Это не руководство для тех, кто использует C++, или не умеет программировать вообще, но хочет научиться писать игры. Это учебник для тех, кто хорошо знает Delphi, но пока не умеет использовать DirectX.

Поскольку в Delphi отсутствует стандартная поддержка DirectX, нам приходится выбирать среди решений, предложенных сторонними разработчиками, главным образом, энтузиастами. Среди таких решений есть и привычное для Delphi, в виде наборов компонентов, например WDirectX и DelphiX. Но я предлагаю другое решение: мы будем использовать набор заголовочных файлов проекта JEDI. Это перенесенные энтузиастами заголовочные файлы из состава DirectX SDK корпорации Microsoft, изначально написанные на C. Такой подход хоть и приводит к кажущемуся поначалу чрезмерно громоздкому коду, но облегчит вам жизнь, когда, например, вы захотите разобраться в коде игр, написанных профессионалами. Очень многое для вас в чужом коде станет знакомым и понятным.

Обновления комплекта заголовочных файлов, а также дополнительные примеры использования DirectX в Delphi вы можете найти по ссылке **http://www.delphi-jedi.org/DelphiGraphics/**.

Заголовочные файлы, которыми я пользовался в настоящей книге, взяты мною с этого сайта, сведения об авторах трансляции указаны в коде модулей.

Здесь же вы можете найти файлы справки из состава DirectX SDK. Обратите внимание, что в файлах справки восьмой версии отсутствует информация о функциях DirectDraw, поэтому вам необходимо найти соответствующие файлы седьмой версии. Пока вы читаете первую главу книги, постарайтесь скачать эти файлы по указанному адресу.

Также вам после прочтения книги очень пригодятся переложения на Delphi примеров из DirectX SDK. Их вы найдете по адресу **http://groups.yahoo.com/group/JEDI-DirectXExamples**.

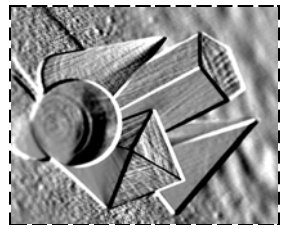
Прилагающийся к книге компакт-диск содержит инсталляцию ядра DirectX восьмой версии, но, возможно, к тому времени, когда вы начали читать эту книгу, уже появились новые версии. Их вы можете найти здесь:

- **http://www.microsoft.com/directx**
- **http://msdn.microsoft.com/directx**

По тем же адресам, наверняка, вы найдете и документацию по текущей версии.

Если у вас возникли какие-либо технические вопросы, такие, например, как проблемы с компакт-диском, обратитесь на сайт издательства **http://www.bhv.ru** (или **mail@bhv.ru**).

ГЛАВА 1



Понятие о COM

Глава является первой в серии глав, посвященных подсистеме DirectDraw. Эта библиотека, как и остальные модули DirectX, реализована в соответствии со спецификацией COM. Глава представляет собой краткий курс по этой спецификации и содержит минимальные сведения, необходимые читателю для понимания механизмов функционирования DirectDraw и других частей DirectX.

Примеры к данной главе располагаются в каталоге \Examples\Chapter01, для изучения они должны быть скопированы на диск, допускающий перезапись.

Библиотеки динамической компоновки

Ключевым понятием операционной системы Windows, позволяющим понять любую технологию, использующуюся в ней, является понятие *библиотеки динамической компоновки* (DLL, Dynamic Link Library). Любое полноценное приложение этой операционной системы (32-разрядное приложение, имеющее собственное окно) использует DLL-файлы. По мере необходимости приложение обращается к библиотекам, вызывая из них нужные функции. Например, выполнимый модуль приложения не содержит кода по отображению окна, вывода в окно и реакции на большинство событий. Перечисленные действия реализуются в системных DLL. В частности, использованием такой технологии удается экономить драгоценные ресурсы, один и тот же код не дублируется многократно, а размещается в памяти единожды.

К одной библиотеке, как правило, может обращаться одновременно несколько приложений. Библиотеку в такой схеме называют *сервером*, а обслуживаемое им приложение — *клиентом*. Сервером и клиентом в общем случае могут являться и библиотека, и приложение. В частности, это означает, что некоторая библиотека, в свою очередь, может "подгружать" функции из другой библиотеки.

Продемонстрируем работу операционной системы следующим примером. Создадим библиотеку, содержащую полезную функцию, выводящую на окне вызывающего клиента растровое изображение. Дальше приведем инструкцию ваших действий в среде Delphi. Готовый результат содержится в каталоге Ex01.

В главном меню выберите пункт **File | New** и в появившемся окне **New Items** щелкните на значке с подписью "DLL".

Чтобы выводимый растр не оказался легко доступным для посторонних глаз, скроем его, поместив в библиотеку. Для этого с помощью редактора ресурсов Image Editor (для вызова его выберите соответствующую команду меню **Tools**) создайте новый файл ресурсов с единственным ресурсом — нужным растром. Присвойте имя ресурсу — BMP1.

Для подготовки этого примера было взято одно из растровых изображений, поставляемых в составе пакета DirectX SDK, скопированное из окна редактора Microsoft Paint через буфер обмена.

Закончив редактировать растр, res-файл запишите в каталог, предназначающийся для проекта библиотеки под именем DLLRes.res.

Код DLL-проекта приведите к следующему виду:

```
library Project1;    // Проект библиотеки
uses
  Windows, Graphics;
{$R DLLRes.res}      // Подключение файла ресурсов
// Описание экспортируемой функции, размещаемой в DLL (export) и
// вызываемой стандартно (stdcall)
procedure DrawBMP (Handle : THandle); export; stdcall;
var
  wrkBitmap : TBitmap;
  wrkCanvas : TCanvas;
begin
  wrkBitmap := TBitmap.Create;
  wrkCanvas := TCanvas.Create;
  try
    // Растр загружается из ресурсов, идентифицируется именем
    wrkBitmap.LoadFromResourceName (HInstance, 'BMP1');
    wrkCanvas.Handle := Handle;
    wrkCanvas.Draw(0, 0, wrkBitmap);
  finally
    wrkCanvas.Free;
    wrkBitmap.Free;
  end;
end;
```

```
// Список экспортируемых функций
// Функция у нас единственная
exports
  DrawBMP;
// Следующий блок соответствует инициализации библиотеки
begin
end.
```

Не будет лишним привести некоторые пояснения. Аргументом функции должен являться идентификатор канвы вызываемой формы. У вспомогательного объекта процедуры, класса `TCanvas`, значение этого идентификатора устанавливается в передаваемое значение, и теперь все его методы будут работать на канве окна, вызывающего функцию приложения.

А сейчас создайте DLL, откомпилировав проект.

Внимание!

Откомпилируйте проект, но не запускайте его. Нельзя запустить DLL в понятии, привычном для обычного приложения.

В каталоге должен появиться файл Project1.dll. Исследуйте библиотеку: поставьте курсор на ее значок, нажмите правую кнопку мыши и в появившемся контекстном меню выберите команду **Быстрый просмотр**.

Примечание

Если данная команда отсутствует, вам необходимо установить соответствующий компонент, входящий в дистрибутив операционной системы.

В окне отобразится информация о содержимом библиотеки, разбитая по секциям, среди которых нас особо интересует секция экспортируемых функций (рис. 1.1).

Если с помощью утилиты быстрого просмотра вы взглянете на содержимое модуля обычного приложения, то не найдете там секции экспортируемых функций. Это принципиальное отличие библиотек динамической компоновки от обычных исполняемых файлов.

Примечание

Некоторые библиотеки скрывают секцию экспортируемых функций от обычного просмотра, но она там обязательно присутствует, даже если библиотека содержит только ресурсы. Пример подобной библиотеки — системная библиотека moricons.dll.

Итак, созданная нами библиотека содержит код экспортируемой функции с именем `DrawBMP` и растровое изображение. Сервер готов. Теперь создайте клиента. Организуйте новый проект, сохраните его в другом каталоге (готовый проект содержится в каталоге Ex02).

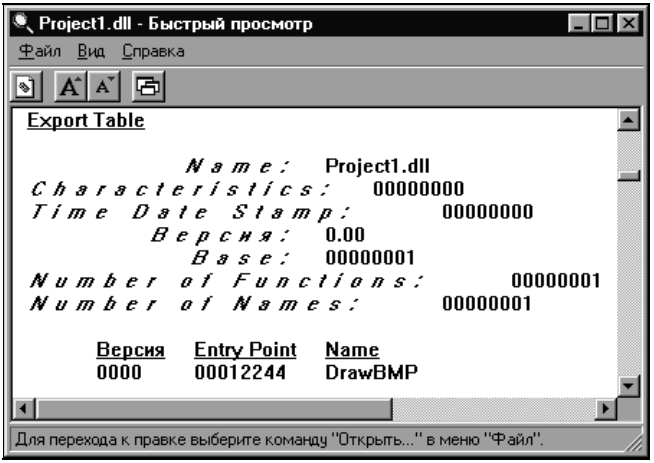


Рис. 1.1. Убеждаемся, что в секции экспортируемых функций созданной библиотеки присутствует название DrawBMP

В секции `implementation` введите следующую строку:

```
procedure DrawBMP (Handle : THandle); stdcall; external 'Project1.dll';
```

Этим мы декларируем нужную нам функцию. Ключевое слово `external` указывает, что данная функция размещена в библиотеке с указанным далее именем. Ключевое слово `stdcall` определяет вызов функции стандартным для операционной системы образом. При использовании импортируемых функций такие параметры задаются обязательно.

На форме разместите кнопку, в процедуре обработки события щелчка кнопки мыши которой введите строку:

```
DrawBMP (Canvas.Handle);
```

Аргументом вызываемой функции передаем ссылку канвы окна. Основной смысл этой величины — идентификация полотна окна.

Откомпилируйте проект, но пока не запускайте. С помощью утилиты быстрого просмотра исследуйте содержимое откомпилированного модуля: найдите в списке импортируемых функций следы того, что приложение использует функцию `DrawBMP` (рис. 1.2).

Обратите внимание, что имя известной нам функции находится в длинном ряду имен других импортируемых приложением функций. То есть модуль даже минимального приложения использует целый сонм функций, подключаемых из DLL. Именно о них упоминалось в начале главы как о функциях, ответственных за появление окна приложения и вывод на канве окна, а также отвечающих за реакцию окна на события. Эти функции именуются *системными*. Так же называются и библиотеки, хранящие код таких функций.

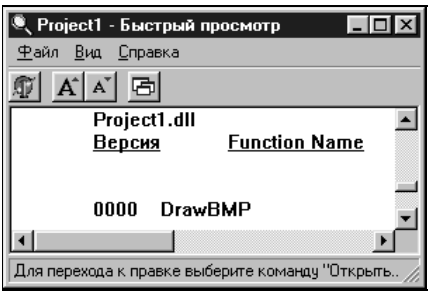


Рис. 1.2. Информация обо всех импортируемых приложением функциях доступна для анализа

Примечание

Другое название системных функций — функции API (Application Program Interface).

При исследовании содержимого созданной нами библиотеки вы могли обратить внимание, что она ко всему прочему импортирует массу функций из системных библиотек.

Delphi позволяет нам писать краткий и удобочитаемый код, но при компиляции этот код преобразуется к вызову массы системных функций, и подчас одна строка кода "расшифровывается" вызовом десятка функций API. Программирование на Delphi образно можно себе представить как общение с операционной системой посредством ловкого переводчика, способного нам одной фразой передать длинную тираду, перевести без потери смысла, но некоторые потери мы все-таки имеем. Прежде всего, мы расплачиваемся тем, что приложения, созданные в Delphi, как правило, имеют сравнительно большой размер. Другая потеря — скорость работы приложения. При использовании библиотеки VCL и концепции объектно-ориентированного программирования вообще, мы жертвуем скоростью работы приложения.

В тех случаях, когда скорость работы приложения чрезвычайно важна, как в случае с обработкой графики, выход может состоять в том, чтобы отказаться от применения "переводчика", писать код, основанный исключительно на использовании функций API. Но такие программы плохо понятны новичкам, требуют специальной подготовки, поэтому мы не будем злоупотреблять этим. Если вы испытаете необходимость подробного разговора о том, как создавать в Delphi приложения без вызова библиотеки классов VCL, то автор может посоветовать вам свою предыдущую книгу, в списке литературы она поставлена на первое место. В ней вы найдете достаточно примеров подобных проектов. Ну а в этой книге постараемся не приводить таких примеров.

Вернемся к нашему примеру. Если вы сейчас запустите приложение, то вас постигнет неудача: сразу после запуска появится системное окно с сообще-

нием о том, что необходимый файл библиотеки не найден. Ничего удивительного, но обратите внимание, что сообщение об ошибке появляется сразу же после запуска приложения, а не после вызова функции, вслед за нажатием кнопки.

Скопируйте в этот же каталог скомпилированную библиотеку и снова запустите приложение. Теперь при запуске все должно быть в порядке, никаких сообщений не появится, а после нажатия кнопки на поверхности окна должна отобразиться картинка (рис. 1.3).

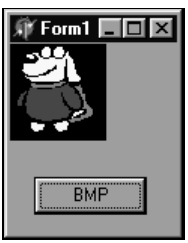


Рис. 1.3. Особенность примера состоит в том, что код для вывода картинки не содержится в модуле приложения

Главное в рассмотренном примере заключается в том, что код приложения не содержит напрямую ничего, связанного с отображаемой в окне картинкой. Приложение обращается к указанной нами библиотеке динамической компоновки, которая выполняет всю работу по выводу изображения.

Первый наш пример является моделью диалога приложения с библиотеками вообще. Каждый раз, когда нам это необходимо, работает библиотека, путем вызова нужной функции.

Посмотрим внимательнее на работу приложения. Картинка исчезает при каждой перерисовке окна, например, если минимизировать, а затем восстановить окно, то картинка "пропадет". Объяснить это легко: при перерисовке окна вызывается собственный обработчик события `OnPaint` окна, а мы позаботились о наличии в нем кода, с помощью которого можно было бы запоминать текущий вид окна. Операционная система подобную услугу не предоставляет, поскольку на нее требуется слишком много ресурсов. Шлифовать код этого примера не станем, мы получили от него почти все, что требовалось для нас.

Запустите несколько копий клиентов и протестируйте вывод картинки на поверхность каждого из них. Пример упрощенный, но я, надеюсь, он смог достичь главной цели, преследуемой мною. Использование динамических библиотек является действительно эффективной технологией построения архитектуры программных систем: код клиентов освобожден от дублирования.

Еще одно важное свойство динамических библиотек состоит в том, что при их использовании безразлично, в какой программной системе созданы клиенты и сами библиотеки. Этим мы пользуемся во время применения DirectX

в проектах Delphi точно так же, как и при использовании любой системной библиотеки.

В коде клиента указывается имя вызываемой функции, но во время работы откомпилированного приложения клиент при вызове динамической библиотеки ориентируется не по имени функции, а по соответствующей функции точке входа, адрес которой он получает при инициализации библиотеки. Взгляните снова на рис. 1.1. Слева от имени экспортируемой функции вы найдете адрес точки входа. Клиент при инициализации библиотеки получает это значение в качестве опорного для вызова функции.

Вспомним, что при запуске исполнимого модуля клиента происходит исключение при отсутствии необходимой библиотеки, рассмотренная компоновка приложения называется *статическим связыванием*.

Динамическое связывание отличается тем, что клиент загружает библиотеку не сразу же после своего размещения в памяти, т. е. запуска, а по мере надобности. Примером такого подхода является проект каталога Ex03. В разделе `implementation` модуля записано следующее:

```
type     // Процедурный тип функции, подгружаемой из библиотеки
  TDrawBMP = procedure (Handle : THandle); stdcall;
// Щелчок кнопки с надписью BMP
procedure TForm1.Button1Click(Sender: TObject);
var
  hcDll : THandle;                          // Указатель на библиотеку
  procDrawBMP : TDrawBMP;                   // Подгружаемая функция
begin
  hcDll := LoadLibrary('Project1.dll');     // Динамическая загрузка DLL
  if hcDll <= HINSTANCE_ERROR then begin    // Загрузка не удалась
     MessageDlg ('Отсутствует библиотека Project1!', mtError, [mbOK], 0);
     Exit;
  end;
  // Библиотека загружена. Получаем адрес точки входа нужной функции
  procDrawBMP := GetProcAddress(hCDll, 'DrawBMP');
  // проверка на успешность операции связывания
  if not Assigned (procDrawBMP) then begin
     MessageDlg (В библиотеке Project1.dll отсутствует нужная функция!,
                 mtError, [mbOK], 0);
     Exit;
  end;
  procDrawBMP (Canvas.Handle);    // Вызываем функцию
  FreeLibrary(hcDll);             // Выгружаем библиотеку
end;
```

Схема наших действий теперь такова: загружаем библиотеку только в момент, когда она действительно необходима, получаем адрес требуемой функции и обращаемся к ней. Обратите внимание, что успешная загрузка

библиотеки не является окончательным признаком того, что мы можем успешно использовать необходимую нам функцию. В каталог этого проекта автор поместил "испорченную" библиотеку Project1.dll, в ней отсутствует нужная нам функция.

Подобная ситуация на практике вполне возможна, если у одной и той же библиотеки есть несколько версий, различающихся набором функций. Производитель подчас распространяет несколько версий программы, различающихся функциональностью, и использующие их клиенты должны перед вызовом функций производить проверку на действительное присутствие этих функций в библиотеке.

Протестируйте работу проекта, заменив библиотеку в его каталоге "правильной", из каталога самого первого примера.

Динамическая загрузка библиотек используется знакомыми вам приложениями очень часто. Например, при проверке правописания текстовый редактор загружает соответствующую библиотеку только при установленном режиме проверки.

COM-модель

Технология, основанная на динамических библиотеках, является очень эффективной, потому и стала основой программной архитектуры операционной системы. Однако ей присуще ограничение, не позволяющее использовать парадигму объектно-ориентированного программирования (ООП): библиотеки могут содержать код функций и процедур, а также ресурсы, но не способны содержать описания классов. Это утверждение верно отчасти, я говорю пока о DLL "в чистом виде". По мере развития программирования как технологии, возникла необходимость поддержки ООП на уровне операционной системы.

Самым ходовым примером такого использования идей ООП на уровне операционной являются составные документы. Вставляя в текстовый документ электронную таблицу или записывая в нем математическую формулу с помощью редактора формул, пользователь текстового процессора как раз встречается со зримым воплощением ООП. Вставленный, внедренный документ является объектом со своими свойствами и методами. Это пример зримого воплощения технологии COM (Component Object Model, модель компонентных объектов). Хотя я и упомянул в примере составные документы, COM предоставляет концепцию взаимодействия программ любых типов: библиотек, приложений, системного программного обеспечения и др. Для нашей темы важно подчеркнуть, что COM стала частью технологий, не имеющих никакого отношения к составным документам.

COM может применяться для создания программ любых типов, в частности DirectX использует эту технологию. Поэтому мы и вынуждены сделать небольшой экскурс в эту тему.

Первоначально для всей группы технологий, в основе которых лежит COM, корпорацией Microsoft было предложено общее имя — OLE. Затем, по мере развития и дополнения технологии, это название менялось. Например, однажды оно стало ActiveX, но программисты со стажем часто так и продолжают пользоваться термином OLE (сейчас это не является аббревиатурой) для обозначения данной группы технологий.

COM — не язык, не протокол. Это метод взаимодействия между программами и способ создания программ.

Функции программы, доступные для использования другим программам, называются *сервисами*. COM определяет стандартный механизм, с помощью которого одна часть программного обеспечения предоставляет свои сервисы другой.

Для нас особенно важно то, что технология COM также является независимой от языка программирования. Физически приложение, предоставляющее сервисы, может быть реализовано в виде обычного выполнимого модуля, либо, чаще всего, реализовано в виде библиотеки. Как и в случае обычных библиотек, неважно, в какой программной системе созданы серверы и использующие их клиенты. В случае с обычной DLL-библиотекой клиенту достаточно знать адрес точки входа нужной функции и в определенный момент передать управление по этому адресу. Тот факт, что библиотека должна предоставлять не обычные функции, а методы объектов, внес в эту схему некоторые изменения, о которых мы поговорим позже.

Контроль версии

Сервером может быть целый программный комплекс, а не один-единственный файл.

При распространении библиотеки ее обычно помещают в какой-либо общедоступный каталог, например системный. Это вам знакомо, поскольку встречалось при установке программ. Наверняка вам известны и возникающие при этом проблемы. Например, при самостоятельном удалении таких программ сопутствующие библиотеки могут остаться, хотя больше никто их не использует.

Если же сервер представляет собой не один файл, а внушительный набор модулей, то размещение его целиком в системном каталоге принесло бы массу дополнительных проблем пользователю, который не сможет правильно определить назначение каждого файла из десятка установленных в системном каталоге.

Перед разработчиками операционной системы стояла следующая задача: необходимо предоставить клиенту возможность доступа к серверу независимо от места его физического расположения. Пусть пользователь устанавли-

вает программы там, где это ему необходимо, хоть и не в общедоступном каталоге, а клиентские программы должны получать доступ к серверу, где бы он ни располагался.

Один из способов решения задачи таков: при установке программы в файл автозагрузки дописывается строка, объявляющая каталог устанавливаемой программы доступным для всех приложений. Теперь при каждом поиске файла система будет заглядывать и в этот каталог. Подобное решение малоэффективно и удовлетворительным являлось лишь два десятилетия назад, когда на одном компьютере установить больше десятка крупных программ практически было невозможно. Сегодня же на компьютере пользователя могут быть установлены одновременно сотни приложений, и блуждание по каталогам может оказаться чересчур долгим. К тому же библиотеки разных производителей, с различным набором функций, могут быть случайно названы одинаково, и клиенту первым может попасться не тот сервер, который он ищет.

Итак, клиент в любой момент должен иметь точную информацию о текущем расположении нужного ему сейчас сервера.

Найденное разработчиками решение состоит в использовании базы данных установленных программ. Такая база данных носит название *реестр*. Функции ее гораздо шире названной мною, но я сосредоточусь только на ней. При установке сервер записывает в реестр свой уникальный идентификатор и, как минимум, информацию о собственном физическом расположении. Клиент при вызове сервера обращается к базе данных установленных программ, ориентируясь по идентификатору, находит и загружает либо запускает сервер. Клиенту, в принципе, можно и не знать имени необходимой библиотеки, главное должен быть известен ее идентификатор. Схема взаимодействия клиента и сервера мною упрощена, напрямую они не общаются, но, надеюсь, основное я сумел донести.

Глобальный вопрос, мучающий впервые прикоснувшихся к этой теме, можно сформулировать так: "Почему это здесь?". DirectX является частью операционной системы, он неизбежно присутствует в ней сразу же после установки. Хоть он и реализован в виде набора файлов, но помещаются они всегда в системный каталог, и ничего зазорного в этом для системных файлов нет. Первый, но не самый главный, ответ на этот вопрос вы уже получили: разработчики стремились отразить требование сегодняшнего дня, связанное с поддержкой ООП на уровне операционной системы. Приступая к разработке DirectX, разработчики корпорации Microsoft задались целью создать набор объектно-ориентированных библиотек, и COM-модель подходит здесь как нельзя лучше.

Подчеркну, что данная книга не является официальным документом, все, что вы в ней читаете, является мыслями автора, и не более. Многие мысли основаны на официальных документах, но далеко не все.

Например, я твердо убежден, что DirectX можно было бы и не строить на основе COM-модели. Для обеспечения его функциональности технологии использования "обычных" библиотек вполне достаточно, а для графической части системы ООП является подспорьем незначительным. Тем более что в технологии COM имеются ограничения с точки зрения традиционного ООП, а новичкам изучение COM часто тяжело дается. Нередко для наглядности при изучении парадигмы ООП прибегают к визуальным иллюстрациям, но сама техника программирования компьютерной графики очень хорошо описывается и стародавним процедурным подходом.

Итак, если бы DirectX не был основан на COM, он в чем-то, может быть, и выиграл. Но это не значит, что весомых оснований в решении разработчиков построить DirectX именно на основе COM-технологии нет.

Существенное преимущество COM-серверов перед обычными библиотеками состоит в облегчении контроля версии сервера. С самого начала работы над DirectX его разработчики были убеждены в том, что одной версией они не ограничатся, и каждая последующая версия продукта будет снабжена новыми, дополнительными функциями. А некоторые прежние функции будут изменяться, например, в связи с устранением ошибок.

В случае с традиционными DLL каждое новое обновление продукта порождает у разработчиков массу проблем. Можно новые функции располагать в библиотеках с новым именем, а старые функции клиентами будут загружаться из прежних библиотек. Это плохо, поскольку влечет потери времени.

Если же новая версия сервера реализована физически в файлах с прежним названием, как серверу узнать, запрашивает ли клиент старую версию функции или новую? Ведь наряду с клиентами, появившимися после выхода новой версии сервера, его будут использовать и клиенты, созданные до появления новой версии. Эти клиенты ничего не знают о новых функциях и изменениях в реализации функций, носящих прежнее имя. Конечно, в библиотеку можно поместить информацию о версии продукта, но тогда в коде каждой функции надо хранить информацию о том, к какой версии сервера она относится. Если добавляется очень много функций, то все это выливается в массу проблем для разработчиков сервера и клиентов.

Вдобавок остается проблема с беспорядочным размещением файлов библиотек на диске: одни и те же файлы могут многократно копироваться на жестком диске в разные каталоги. Или поверх обновленной версии может быть установлена более старая.

Технология COM тем и отличается от традиционных библиотек, что хорошо приспособлена к решению проблемы контроля версии сервера. Хочу подчеркнуть, что все эти проблемы устранимы и в схеме традиционных DLL, но решения получаются громоздкими и способны привести к ошибкам. С использованием же технологии COM появляется гарантия, что сервер не будет установлен многократно, а клиент станет получать именно запрашиваемый набор функций.

COM-объекты

Как уже отмечалось, технология COM появилась вслед за возникшей потребностью программистов получить реализацию парадигмы ООП. В COM любая часть программного обеспечения реализует свои сервисы как один или несколько объектов COM.

COM-объекты представляют собой двоичные программные компоненты, подобно компонентам Delphi, устанавливаемым на уровне операционной системы и доступным для использования в любой среде программирования. COM-объекты для Object Pascal ничем, по сути, не отличаются от обычных объектов, или, по крайней мере, очень похожи на обычные невизуальные объекты, такие как объекты класса `TBitmap`. Изучение DirectX позволит нам разобраться с методами невизуальных объектов особых типов. Только необходимо сразу же запомнить, что у COM-объектов нет свойств, есть только методы. Вдобавок, коренное отличие таких объектов состоит в использовании конструкторов и деструкторов.

Для создания COM-объекта не вызывается функция конструктора, как для обычных объектов в Delphi. Первым нашим действием будет создание *главного объекта*, который имеет методы, использующиеся для создания других объектов и получения необходимых интерфейсов.

Для удаления COM-объекта вместо метода `Free` обычно предназначен метод `_Release`. Это справедливо в общем случае, но иногда для освобождения памяти, занятой COM-объектом, будем просто присваивать значение `nil` соответствующей переменной.

Интерфейсы

Интерфейсом обозначается набор функций, предоставляемый некоторым предложением. Обычные приложения предоставляют один интерфейс, т. е. весь тот набор функций, который реализован в вашей, к примеру, бухгалтерской программе, является в такой терминологии единым интерфейсом. Если бы ваша бухгалтерская программа могла предоставлять несколько наборов функций, то она имела бы несколько интерфейсов.

Здесь начинающие обычно испытывают затруднение. Вопрос, зачем же DirectX предоставляет несколько интерфейсов, кажется резонным.

Вспомним еще раз проблему контроля версии. Клиент может запрашивать функцию или набор функций, реализованных в различных версиях DirectX, по-разному. Очень важно предоставлять ему эти функции именно в той реализации, как он того ожидает. Например, если в какой-либо предыдущей версии функция реализована с известной ошибкой, то клиент при использовании этой функции может делать поправку на данную ошибку. Тогда,

если клиент получит уже скорректированную функцию, такая поправка может только испортить все дело.

DirectX предоставляет несколько интерфейсов, связанных с различными версиями. Клиент запрашивает именно тот интерфейс, который ему известен. Например, клиент создан пару лет назад и просто ничего не знает о новых функциях, появившихся в DirectX с тех пор. Функции, чьи имена не изменились, но реализация в последующих версиях сервера претерпела изменения, должны работать именно так, как они работали во времена создания клиента, и как того ожидает клиент.

Конечно, подобная схема отнюдь не идеальна. Например, если функция в новой версии реализована эффективнее, то "старый" клиент просто не сможет ею воспользоваться, он запустит ее старую версию. Поэтому при установке новой версии DirectX не приходится ожидать, что ранее установленные игры автоматически станут выглядеть иначе. Но все же это одно из самых эффективных решений.

Итак, сервер поддерживает один или несколько интерфейсов, состоящих из методов. Клиенты могут получить доступ к сервисам только через вызовы методов интерфейсов объекта. Иного непосредственного доступа к данным объекта у них нет.

Все COM-интерфейсы унаследованы от интерфейса, называемого `IUnknown`, обладающего тремя методами: `QueryInterface`, `AddRef` и `_Release`. О них нам надо знать совсем немного, ведь непосредственно к графике они отношения не имеют.

Последний в этом списке метод мы уже вскользь обсуждали — удаление объекта. Часто использование его будем заменять простым освобождением памяти.

Предпоследний метод предназначен для подсчета ссылок на интерфейсы. Клиент явно инициирует начало работы экземпляра COM-объекта, а для завершения его работы он вызывает метод `_Release`. Объект ведет подсчет клиентов, использующих его, и когда количество клиентов становится равным нулю, т. е. когда счетчик ссылок становится нулевым, объект уничтожает себя сам. Новичок может здесь растеряться, поэтому я уточню, что мы всем этим не будем пользоваться часто, и вы можете особо не напрягать внимание, если все это кажется сложным. Просто клиент, получив указатели на интерфейсы объекта, способен передать один из них другому клиенту, без ведома сервера. В такой ситуации ни один из клиентов не может закончить работу объекта с гарантией того, что делает это преждевременно. Пара методов `AddRef` и `_Release` дает гарантию того, что объект исчезнет только тогда, когда никто его не использует.

Обычно свой первый указатель на интерфейс объекта клиент приобретает при создании главного объекта. Имея первый указатель, клиент получает

указатели на другие интерфейсы объекта, методы которых ему необходимо вызывать, запрашивая у объекта эти указатели с помощью метода `QueryInterface`.

Перейдем к иллюстрации. В этом проекте мы должны сообщить пользователю, возможно ли применять на данном компьютере DirectX седьмой версии. Это самое простое приложение, использующее DirectDraw, и здесь нет графики, мы только определяемся, возможна ли в принципе дальнейшая работа. У DirectDraw нет интерфейсов восьмой версии, и наше приложение не различит седьмую и последующие версии. Позже мы сможем распознать присутствие именно восьмой версии, а пока что наши приложения пусть довольствуются и предыдущей.

Можете взять готовый проект из каталога Ex04, но будет лучше, если вы повторите все необходимые действия сами.

Создайте новый проект и в его опции **Search path** запишите путь к каталогу, содержащему заголовочный файл DirectDraw.pas, в моем примере там записано "..\..\DUnits".

В разделе `private` опишите две переменные:

```
FDD : IDirectDraw;
FDD7 : IDirectDraw7;
```

Первая из них является главным объектом DirectDraw. Вторая переменная нужна, чтобы проиллюстрировать применение метода `QueryInterface`. В используемом нами сейчас модуле `DirectDraw.pas` найдите строки, раскрывающие смысл новых для нас типов:

```
IDirectDraw = interface;
DirectDraw7 = interface;
```

Ключевое слово `interface` здесь, конечно, является не началом секции модуля, а типом, соответствующим интерфейсам COM-объектов.

Обработчик создания окна приведите к следующему виду:

```
procedure TForm1.FormCreate(Sender: TObject);
var
    hRet    : HRESULT;     // Вспомогательная переменная
begin
    // Создание главного объекта DirectDraw
    hRet := DirectDrawCreate (nil, FDD, nil);
    if Failed (hRet)      // Проверка успешности предыдущего действия
       then ShowMessage ('Ошибка при выполнении DirectDrawCreate')
       // Поддерживается ли интерфейс 7-й версии DirectX
       else hRet := FDD.QueryInterface (IID_IDirectDraw7, FDD7);
    if Failed (hRet)        // Или один из двух,
                            // или оба интерфейса не получены
```

```
     then ShowMessage ('DirectX 7-й версии не доступен')
     else ShowMessage ('DirectX 7-й версии доступен');
    // Освобождение памяти, занятой объектами
    if Assigned (FDD7) then FDD7 := nil;
    if Assigned (FDD) then FDD := nil;
end;
```

Уже при подготовке этого, простейшего, примера я прибегнул к некоторым упрощениям, но все равно у каждого новичка здесь появится масса вопросов. Попробую предвосхитить и разрешить их.

Итак, первая строка кода — создание главного объекта, через интерфейсы которого выполняются действия по созданию остальных объектов. Как я говорил, для COM-объектов нельзя использовать обычный конструктор.

Переменная `DirectDrawCreate` описывается в заголовочном файле DirectDraw.pas так:

```
DirectDrawCreate : function (lpGUID: PGUID;
                             out lplpDD: IDirectDraw;
                             pUnkOuter: IUnknown) : HResult; stdcall;
```

При инициализации модуля происходит связывание переменной и получение адреса точки входа:

```
DirectDrawCreate := GetProcAddress(DDrawDLL,'DirectDrawCreate');
```

Это нам немного знакомо по первому примеру. Здесь происходит динамическая загрузка функции из библиотеки. Ссылка на библиотеку описывается так:

```
var
  DDrawDLL : HMODULE = 0;
```

Первое действие при инициализации модуля — загрузка библиотеки:

```
DDrawDLL := LoadLibrary('DDraw.dll');
```

С помощью утилиты быстрого просмотра можем убедиться, что действительно в списке экспортируемых функций данной библиотеки (обратите внимание, что этот список сравнительно невелик) присутствует имя функции `DirectDrawCreate`. Напоминаю, что сам файл библиотеки содержится в системном каталоге, как правило, это C:\Windows\System\. Оттуда загружается функция. Но каков смысл ее аргументов и возвращаемой ею величины? Начнем с возвращаемой величины. Из описания ясно, что тип ее — `HRESULT`, который имеет результат всех функций, связанных с OLE. Обрабатывается результат таких функций для проверки успешности каких-либо действий, как в данном случае, для того, чтобы выяснить, успешно ли выполнена операция получения интерфейса.

Это 32-битное целое значение, описание типа которого вы можете найти в модуле system.pas:

```
HRESULT = type Longint;
```

HRESULT — общий для OLE тип, соответствующий коду ошибки. Каждый сервер по-своему распределяет возможные ошибки и возвращаемый код. Общим является то, что нулевое значение эквивалентно отсутствию ошибки.

Коды ошибок, возвращаемых функциями, связанными с DirectDraw, можно интерпретировать в осмысленную фразу с помощью функции

```
function DDErrorString(Value: HResult) : string;
```

Эта функция описана в модуле DirectDraw.pas. Аргументом ее является код ошибки, результатом — строка, раскрывающая смысл произошедшей неудачи. Равенство нулю кода выступает признаком успешно выполненной операции. Анализ успешности операции часто выполняется просто сравнением возвращаемой величины с константой DD_OK, равной нулю.

Примечание

Константа S_OK, равная нулю, также может применяться во всех модулях, использующих OLE, но обычно каждый из них определяет собственную нулевую константу.

В примере для оценки успешности операции я пользуюсь системной функцией, описанной в модуле windows.pas:

```
function Failed(Status: HRESULT): BOOL;
```

Функция возвращает значение True, если аргумент отличен от нуля. Есть и обратная ей функция, возвращающая значение True при отсутствии ошибок:

```
function Succeeded(Status: HRESULT): BOOL;
```

Теперь вернемся к аргументам функции DirectDrawCreate. Первый из них задает параметры работы приложения, если задавать значение его в nil, то при работе будет применяться текущий видеодрайвер. Если же необходимо строго оговорить, чтобы приложение не использовало все преимущества аппаратного ускорения, то это значение нужно установить так:

```
PGUID(DDCREATE_EMULATIONONLY)
```

Если же требуется оговорить, что создаваемый объект DirectDraw не будет эмулировать особенности, не поддерживаемые аппаратно, надо использовать в качестве этого параметра константу DDCREATE_HARDWAREONLY. Функция DirectDrawCreate тогда "проглотит" аргумент в любом случае, но, в будущем, попытка вызвать методы, требующие неподдерживаемые особенности, приведет к генерации ошибки с кодом DDERR_UNSUPPORTED.

Второй параметр функции — собственно наш объект, который примет данные.

Последним аргументом всегда надо указывать `nil`. Этот параметр зарезервирован для будущих нужд, чтобы старые приложения смогли в перспективе работать при измененной COM-модели.

Так, все, связанное с первым действием, — созданием главного объекта, — разобрали. Функция `DirectDrawCreate` вряд ли когда-либо возвратит ненулевое значение. Это будет соответствовать ситуации серьезного сбоя в работе системы. Однако после каждого действия необходимо проверять успешность его выполнения. Приходится сразу же привыкнуть к тому, что код будет испещрен подобной проверкой, анализом возвращаемого функцией значения. Некоторые действия вполне безболезненно можно выполнять без проверки на неудачу, поскольку ошибки при их выполнении если и возможны, то крайне редки. Ключевые же операции следует обязательно снабжать подобным кодом, поскольку ошибки при их выполнении вполне возможны и даже ожидаемы. Появляются эти исключения не по причине неустойчивого поведения системы или приложения, а закономерно в ответ на изменения окружения работы приложения. Например, пользователь может временно переключиться на другое приложение или поменять параметры рабочего стола по ходу работы вашего приложения. Анализ исключений позволяет вашему приложению отслеживать такие моменты и реагировать на изменившиеся условия работы.

Дальше в коде примера идет следующая строка:

```
FDD.QueryInterface (IID_IDirectDraw7, FDD7);
```

Вызываем метод `QueryInterface` главного объекта для получения нужного нам интерфейса, соответствующего седьмой версии DirectX. Заглянем в описание этого интерфейса. Начало выглядит так:

```
IDirectDraw7 = interface (IUnknown)
  ['{15e65ec0-3b9c-11d2-b92f-00609797ea5b}']
```

Все интерфейсы строятся на базе интерфейса `IUnknown`, в следующей строке указывается идентификатор конкретного интерфейса, за которой приведено перечисление его методов. Идентификаторы интерфейсов используются при взаимодействии клиента с сервером. Следы наиболее важных идентификаторов мы можем обнаружить в реестре. Например, в заголовочном файле вы можете найти такие строки описания идентификаторов:

```
const
  CLSID_DirectDraw: TGUID = '{D7B70EE0-4340-11CF-B063-0020AFC2CD35}';
  CLSID_DirectDraw7: TGUID = '{3c305196-50db-11d3-9cfe-00c04fd930c5}';
```

Запустив системную программу редактирования реестра regedit.exe и активизировав поиск любого из этих идентификаторов, вы способны найти соответствующие записи в базе данных (рис. 1.4).

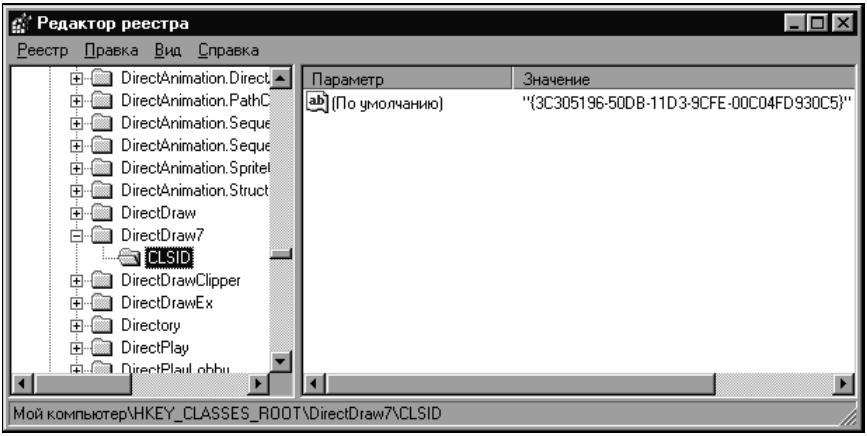


Рис. 1.4. По значению идентификатора интерфейса клиент находит запись о сервере в реестре

Я изрядно упрощаю рассмотрение тонких вопросов, связанных с COM-моделью, но для успешного использования DirectX нам таких общих представлений о ней будет вполне достаточно.

Аргументов у метода `QueryInterface` два: запрашиваемый интерфейс и объект, в который должен помещаться результат.

Дальше в нашей программе идет проверка успешности предыдущего действия, по традиционной схеме. Обратите внимание, что другой признак провала конкретно этой операции заключается в том, что значение `FDD7` окажется равным `nil`. COM-объекты в этом плане для нас будут такими же, как и обычные объекты в Delphi, признаком связанности объектов является наличие каких-либо данных в них.

Попутно еще одно важное замечание. В начале работы необходимо установить в `nil` значение всех переменных, соответствующих COM-объектам. Только из желания упростить код я не сделал этого в программе, но в последующих примерах будем строго следить за выполнением данного правила. Все подобные мероприятия кажутся необязательными, но невыполнение их только повышает вероятность некорректной работы вашего приложения.

Тот факт, что нам не удастся получить указатель нужного интерфейса, является вполне возможным, например, у пользователя просто не установлен DirectX необходимой нам версии. Клиент запрашивает интерфейс седьмой версии, и получит его именно в таком виде, как он того ожидает, даже если установлен DirectX старшей версии.

После информирования пользователя о том, установлен ли у него DirectX нужной нам версии, работа программы завершается, и память, занятая COM-объектами, освобождается. Последнее действие тоже является процедурой, обязательной для всех наших примеров. Если этого не делать, то

приложение может при выходе порождать исключения. Другая возможная ситуация: приложение корректно работает при первом запуске, а после его закрытия ни то же самое приложение, ни любое другое, использующее DirectX, корректно работать уже не может. Каждый раз, когда вы встречаетесь с подобной ситуацией, помните, что вина за это целиком лежит на вашем приложении. Такие простые программы, как разбираемая нами сейчас, навряд ли приведут к похожим авариям, но будем привыкать делать все правильно.

Память, занятую объектами, мы освобождаем в порядке, обратном порядку их связывания. Данное правило тоже очень важно соблюдать. Использование функции `Assigned` вполне можно заменить сравнением значения переменной с `nil`, в этом плане все выглядит также обычно, как и при работе с самыми заурядными объектами Delphi.

Из всех предопределенных методов интерфейсов метод `QueryInterface` является самым важным. Но и им мы, в дальнейших примерах, пользоваться не будем.

Рассматриваемый пример может подсказать нам, какие действия надо предпринимать в распространяемых приложениях, чтобы они корректно работали в ситуации отсутствия на пользовательском компьютере нужной нам версии DirectX. Но в остальных примерах инициализацию DirectDraw подобным образом проводить не будем, подразумевая, что нужные интерфейсы присутствуют.

Важное замечание: рассмотренный порядок действий в начале работы приложения является самым надежным для случаев, если приложение может быть запущено на компьютерах, не располагающих DirectX версии 7 и выше. Если в такой ситуации вам надо сообщить пользователю о необходимости установить DirectX нужной версии, то действуйте именно так, как мы рассмотрели выше. Описываемый далее способ, предлагаемый разработчиками, для такой ситуации не совсем хорош, поскольку опирается на принципиально новые функции, отсутствующие в библиотеках ранних версий DirectX. При попытке загрузки отсутствующей функции будет генерироваться исключение. Поэтому ваше приложение может просто не добраться до информирования пользователя.

Для старших версий DirectX разработчики рекомендуют пользоваться функцией

```
DirectDrawCreateEx : function (lpGUID: PGUID;
                               out lplpDD: IDirectDraw7; const iid: TGUID;
                               pUnkOuter: IUnknown) : HResult; stdcall;
```

Главный объект теперь должен быть типа `IDirectDraw7`, здесь же мы указываем требуемый нами интерфейс. То есть эта функция объединяет два действия, рассмотренные в предыдущем примере.

Очередным примером является проект каталога Ex05. Код немного отягощен включением защищенного режима, но приложение будет корректно работать на компьютере со старой версией DirectX.

Главный объект здесь имеет тип `IDirectDraw7`, а обработчик события `OnCreate` формы выглядит так:

```
procedure TForm1.FormCreate(Sender: TObject);
var
  hRet : HRESULT;   // Вспомогательная переменная для анализа результата
begin
  FDD := nil;       // Это обязательно для повышения надежности работы
  try               // Включаем защищенный режим
  try               // ...finally
  // Создание главного объекта DirectDraw
  hRet := DirectDrawCreateEx (nil, FDD, IDirectDraw7, nil);
  if Failed (hRet)  // В случае ошибки наверняка сюда не доберемся
     then ShowMessage ('DirectX 7-й версии не доступен')
     else ShowMessage ('DirectX 7-й версии доступен');
  finally           // В любом случае производим освобождение памяти
  if Assigned (FDD) then FDD := nil;
  end;
  except            // В случае ошибки информируем о неудаче
     ShowMessage ('DirectX 7-й версии не доступен')
  end;
end;
```

Как видно из комментариев, анализ значения переменной `hRet` здесь можно и не производить, обращение к функции `DirectDrawCreateEx` на компьютере с установленным DirectX версии младше седьмой приведет к появлению исключения.

В наших последующих примерах мы, как правило, будем пользоваться именно функцией `DirectDrawCreateEx`, чтобы иметь доступ ко всем возможностям, предоставляемым последними версиями DirectX. Так рекомендуют разработчики. Защищенный режим в такой ситуации включать не будем, но только в погоне за удобочитаемостью кода.

Что вы узнали в этой главе

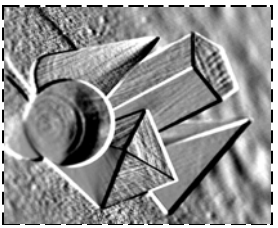
Первая, вводная глава посвятила читателей в программную архитектуру операционной системы и напомнила о важной роли динамических библиотек в этой архитектуре. COM-модель будем считать развитием технологии "традиционных" DLL, позволяющей использовать парадигму ООП на уровне операционной системы, функций API.

Изучение DirectX сводится к знакомству с методами невизуальных объектов.

DirectX, как основа построения графики, пока еще не рассматривался. Но мы уже познакомились с действиями, обязательными при его использовании:

- первое действие инициализации — создание главного объекта, осуществляется специальной функцией;
- методы главного объекта служат для получения интерфейсов;
- функции DirectX возвращают код ошибки.

ГЛАВА 2



Обзор библиотеки DirectDraw

Настоящая глава содержит краткие сведения о DirectDraw API, вводит читателя в круг базовых терминов и понятий.

Обучение начинается с самых простейших программ. Приводятся приемы отладки приложений, использующих DirectDraw.

Примеры к данной главе располагаются в каталоге \Examples\Chapter02.

Поверхности

В первой главе мы узнали, что работа начинается с создания главного объекта DirectDraw, а его методы используются для создания остальных необходимых нам объектов. Здесь у вас не должно сложиться впечатление, что все методы главного объекта предназначены только для создания других объектов. Однако он имеет также методы с иными предназначениями.

Рассмотрим проект каталога Ex01. Имя формы `frmDD` задано, в разделе `private` описания класса формы объявлены две переменные и вспомогательная функция:

```
FDD  : IDirectDraw7;                                    // Главный объект
FDDSPrimary  : IDirectDrawSurface7;                     // Поверхность
procedure ErrorOut(hRet : HRESULT; FuncName : String); // Вывод сообщений
```

Вспомогательная функция вывода сообщений получает в качестве аргументов код ошибки и поясняющее сообщение и выводит расшифровку кода, используя функцию `DDErrorString`:

```
procedure TfrmDD.ErrorOut(hRet : HRESULT; FuncName : String);
begin
  MessageBox(0, PChar(FuncName + ': ' + #13 + DDErrorString(hRet)),
             PChar(Caption), MB_OK or MB_ICONSTOP);
end;
```

Обработчик события OnCreate формы дополнился новыми для нас действиями, которые мы должны очень внимательно разобрать:

```
procedure TfrmDD.FormCreate(Sender: TObject);
var
  hRet : HRESULT;           // Для анализа успешности действий
  ddsd : TDDSurfaceDesc2;   // Вспомогательная структура
begin
  FDDSPrimary := nil;       // В начале работы обнуляем все COM-объекты
  FDD := nil;
  // Создание главного объекта DirectDraw
  hRet := DirectDrawCreateEx (nil, FDD, IDirectDraw7, nil);
  if hRet <> DD_OK then begin
    ErrorOut(hRet, 'DirectDrawCreateEx');
    Exit;
  end;
  // Задаем уровень кооперации
  hRet := FDD.SetCooperativeLevel(Handle, DDSCL_FULLSCREEN or
                                  DDSCL_EXCLUSIVE);
  if hRet <> DD_OK then begin
    ErrorOut(hRet, 'SetCooperativeLevel');
    Exit;
  end;
  // Заполняем поля вспомогательной структуры
  FillChar(ddsd, SizeOf(ddsd), 0);   // Для начала все поля обнуляем
  ddsd.dwSize := SizeOf(ddsd);       // Поле размера структуры
  ddsd.dwFlags := DDSD_CAPS;
  // Будет создаваться первичная поверхность
  ddsd.ddsCaps.dwCaps := DDSCAPS_PRIMARYSURFACE;
  // Собственно создание первичной поверхности
  hRet := FDD.CreateSurface(ddsd, FDDSPrimary, nil);
  if hRet <> DD_OK then begin
    ErrorOut(hRet, 'Create Primary Surface');
    Exit;
  end;
end;
```

Прежде чем мы разберем подробно все новое, обращаю внимание, что при завершении работы приложения привычным для нас способом освобождаем переменные в порядке, обратном их связыванию:

```
procedure TfrmDD.FormDestroy(Sender: TObject);
begin
  if Assigned(FDD) then begin // Связана ли переменная главного объекта
  // Связана ли переменная первичной поверхности
    if Assigned(FDDSPrimary) then FDDSPrimary := nil;
```

```
    FDD := nil;
  end;
end;
```

При запуске проекта чего-либо особенного не происходит, только окно оказывается распахнутым на весь экран, хотя в свойствах формы ничего подобного не указано. Обратите внимание также, что окно хоть и распахнуто, но не максимизировано, мы можем изменять его размеры привычным способом.

Примечание

Возьмите за правило не запускать проекты, использующие DirectDraw, под управлением среды Delphi. Запускайте непосредственно откомпилированный модуль.

Сразу после создания главного объекта мы задаем уровень кооперации, используя метод `SetCooperativeLevel`. Уровень кооперации определяет уровень взаимодействия приложения с экраном и с другими приложениями. И это действие — задание уровня кооперации — является обязательным для всех приложений, использующих DirectDraw. Метод имеет два параметра: первый является идентификатором окна приложения, здесь мы передаем значение свойства `Handle` формы, второй параметр представляет собой одно из строго определенных значений, либо комбинацию таких значений.

Используемая комбинация битовых флагов записана в этом примере как "`DDSCL_FULLSCREEN or DDSCL_EXCLUSIVE`". В подобных случаях всегда слово `or` можно заменить знаком `+`. Первый флаг задает полноэкранный режим работы приложения, второй — монопольный режим доступа к экрану. Оба флага в комбинации должны присутствовать.

Примечание

В Delphi подобные комбинации битовых значений редко встречаются, поэтому для начинающих требуются пояснения. Такие комбинации используются для передачи одним аргументом нескольких параметров. Константы, указанные в комбинации, представляют собой степени двойки. Из суммы таких чисел легко выделить присутствие каждой константы: наличие единицы в разряде двоичного числа.

Итак, в рассматриваемом примере запрашивается полноэкранный режим работы приложения, и по правилам, установленным DirectX, необходимо для такого режима установить эксклюзивный уровень доступа. Теперь понятно, почему окно распахивается на весь экран.

Дальше по коду создается первичная поверхность. Поверхность является одним из основных понятий DirectDraw, за этим термином скрывается прямоугольный блок памяти. Поясню смысл этого понятия на примере анимации, которую можно организовать, скажем, следующим образом: кадры

анимации хранятся в отдельных блоках памяти, и с течением времени на экран выводится нужный блок. Блок, хранящий отдельный кадр, и блок, соответствующий экрану, называются *поверхностями*. О поверхностях можете думать как о растрах, размещенных в памяти, или как об объектах класса `TBitmap`.

После изрядной практики в использовании компонентов Delphi и рисовании с помощью методов `Canvas`, начинающие программисты в DirectDraw испытывают при знакомстве с термином поверхности вполне объяснимую неловкость. Некоторые вещи здесь могут показаться новичку неудобными и неясными. В дальнейшем, с практикой, придет полное понимание всех моментов смысла поверхности и работы с ней, а сейчас вы должны твердо для себя уяснить, что экран для будущей работы вы обязаны подготовить сами, и производится это в терминах поверхности. Даже если у вас не будет на экране картинок, а вы хотите просто порисовать так, как привыкли это делать с помощью методов `Canvas`.

Создаются поверхности с помощью метода `CreateSurface` главного объекта, но для задания свойств создаваемой поверхности используется вспомогательная величина типа `TDDSurfaceDesc2`. Представляет она собой структуру — запись в традиционной для Delphi терминологии. Значения ее полей содержат параметры создаваемой поверхности. Только в редких случаях необходимо определять значения абсолютно всех параметров. Как правило, можно обойтись только парой из них. Остальные поля DirectX заполнит за нас. В любом случае, следует обязательно обнулить все поля структуры. Это очень важное правило. В нашем примере поля обнуляются с помощью функции `FillChar`, но помните, что это же можно сделать посредством другой функции, `ZeroMemory`:

```
ZeroMemory (@ddsd, SizeOf(ddsd));
```

Следующее действие также является обязательным: в поле `dwSize` надо записать размер структуры. Оба действия, обнуление всех полей и установка размера, необходимо выполнять в начале работы с любой структурой, встречающейся нам в DirectX. Пренебречь каким-либо из них у нас просто не получится. Проверьте сейчас же: удалите любую из этих строк и запустите проект. Выполнение кода обработчика `OnCreate` завершится аварийно. Авария, впрочем, для этого проекта не является фатальной. Исключение связано с DirectDraw и пока не приводит к полному провалу работы приложения. Запомните, как выглядит окно выдачи расшифровки ошибки — работа нашей пользовательской функции `ErrorOut`, чтобы сразу же отличать аварийные ситуации, связанные с DirectX.

Итак, неиспользуемые поля структуры будут нулевыми. Главное поле, которое нам надо обязательно заполнить, — это поле `ddsCaps`, представляющее описание возможностей (capabilities) — наиважнейших характеристик по-

верхности. Поле это также является структурой, из всех полей которой мы обязаны задать, как минимум, значение поля `dwCaps`.

Поле `dwFlags` структуры `TDDSurfaceDesc2` содержит указания, какие из ее полей заполнены нами и должны быть приняты системой в расчет. Присвоив этому полю значение здесь `DDSD_CAPS`, мы указываем DirectDraw, что нами заполнено именно поле `ddsCaps`, а все остальные параметры создаваемой поверхности отдаются на откуп графической системе, и она будет распоряжаться ими по собственному усмотрению. Если мы поместим в поля структуры значения, но забудем указать это в поле флагов, графическая система установленные значения в расчет принимать не будет.

Вывод в DirectDraw осуществляется на поверхностях, которых может быть столько, сколько нам надо. Предназначение и параметры каждой из них мы задаем, исходя из логики приложения. Поверхности можно использовать и как хранилища вспомогательных ресурсов — битовых растров, чем мы и будем пользоваться достаточно часто.

Но среди всех поверхностей есть одна, особая, связанная непосредственно с экраном. Все, что на нее будет выведено, окажется отображенным прямо на экране монитора. Называется эта поверхность *первичной*. Она обязательно создается в любом проекте, рисующем что-либо на экране с помощью DirectDraw.

В поле `dwCaps` структуры `ddsCaps`, являющейся, в свою очередь, частью структуры `ddsd`, заносим значение `DDSCAPS_PRIMARYSURFACE`, вызывая метод `CreateSurface` главного объекта. У метода три аргумента: адрес структуры, описывающей параметры создаваемой структуры, переменная, в которую помещается результат — созданная поверхность. Третий параметр зарезервирован для будущих нужд, а пока обязан быть установлен в `nil`.

Далее мы традиционным образом оцениваем успешность проделанной операции, и в случае ее провала сообщаем об ошибке и выполняем выход из программы. Строго говоря, здесь мы завершаем работу приложения в любом случае, независимо от успеха создания поверхности — ведь код на этом заканчивается.

Точно так же, как и в примере первой главы, равенство `nil` переменной поверхности является признаком неудачи предыдущей операции.

Мы разобрали все действия в нашем примере, и все они должны быть вам понятны.

Давайте повторим порядок действий:

- создается главный объект;
- задается уровень взаимодействия приложения с системой и другими приложениями;
- заполняются поля структуры, хранящей параметры поверхности;

- создается, как минимум, одна поверхность — первичная, связанная с экраном.

Поначалу все это может показаться чересчур громоздким, но по мере накопления опыта у вас появится легкость понимания кода.

Начинающих программистов может смущать кажущаяся запутанность с цифрами в типах переменных, скажем, тип `TDDSurfaceDesc2` заканчивается не на 7. Одни типы, например интерфейсы, менялись с каждой версией, другие же вспомогательные типы модифицировались реже, поэтому их цифры "отстают" от нумерации используемых интерфейсов.

Еще один вопрос, который надо разрешить, тоже связан с версией DirectX. Многое из того, что мы применяем, присутствует и в более ранних версиях, и для массы примеров вполне можно использовать интерфейсы не седьмой, а, например, пятой версии. Легко поддаться такому соблазну, ведь тогда круг потенциальных пользователей вашей программы существенно расширяется. Я не смогу придерживаться этого, и вам советую делать подобное лишь в случае крайней необходимости. Ведь наверняка со временем разношерстность кода приведет к полному беспорядку в ваших проектах. Согласно спецификации COM-интерфейс не может меняться после его определения. Новый интерфейс должен поддерживать как старые, так и новые возможности. Но интерфейсы со старшими номерами версий не обязательно должны быть образованы от соответствующих интерфейсов с меньшими номерами.

Еще одна потенциальная проблема связана со вспомогательными структурами. Мы уже начали с ними знакомиться. Доступ к этим структурам, используемым в методах интерфейсов, осуществляется в программе непосредственно, а не через интерфейс (мы сами заполняем поля записи). Структуры с каждой новой версией могут обрастать новыми полями, следовательно, размер их меняется.

Примечание

Именно по этой причине каждая функция должна обязательно получать и размер передаваемой структуры.

Более старая версия DirectX не сможет обработать новые поля знакомой ей структуры, она просто ничего не знает о появившихся в ней новых полях. Разработчики попытались снять часть возникших проблем, вводя новые типы структур, те самые двойки в имени их появились из-за запрета на применение одноименных методов разных интерфейсов. Но самым лучшим решением для нас будет использование текущей версии DirectX.

Двигаемся дальше. Попробуем порисовать что-нибудь в привычном для нас антураже. Переходим к проекту каталога Ex02, отличающемуся от предыдущего тем, что здесь добавился обработчик события `OnPaint` формы:

```
procedure TfrmDD.FormPaint(Sender: TObject);
var
  // Вспомогательный дескриптор, идентификатор устройства вывода GDI
  DC : HDC;
  wrkCanvas : TCanvas; // Вспомогательный объект, рабочая канва
begin
  // Получение дескриптора, необходимого для функций GDI
  if FDDSPrimary.GetDC(DC) = DD_OK then begin
     wrkCanvas := TCanvas.Create; // Создаем вспомогательную канву
     wrkCanvas.Handle := DC; // Задаем идентификатор канвы = DC
     // Рисуем на канве кружок
     wrkCanvas.Ellipse (Left + 50, Top + 50, Left + 100, Top + 100);
     wrkCanvas.Free; // Освобождение памяти, удаление канвы
     FDDSPrimary.ReleaseDC (DC); // Освобождение контекста устройства
  end;
end;
```

Пример учебный, не ожидайте от него ничего выдающегося. Впереди нас ждут еще более впечатляющие программы. При запуске проекта ничего особенного не происходит, на поверхности окна рисуется кружок. Причем вы можете даже обнаружить, что появляется он на экране медленнее, чем нарисованный обычным способом.

Канва Delphi является оболочкой системных функций вывода GDI, ее свойство `Handle` в точности соответствует типу `HDC`, ссылке на устройство вывода. В этой величине нуждаются все функции GDI для идентификации устройства, окна или блока памяти, в который осуществляется вывод.

Как видно из кода, метод поверхности `GetDC` позволяет в нужную переменную поместить значение такого идентификатора. Установив значение `Handle` канвы в найденное значение, мы добьемся того, что вывод на канве физически будет осуществляться на необходимое нам устройство. Все это нам знакомо по предыдущей главе.

Первичная поверхность в нашей программе является канвой рабочего стола, так мы сами задали ее свойства. Поэтому канва связана с областью всего экрана, и прямоугольник, ограничивающий кружок, мы задаем в координатах рабочего стола, а не в координатах окна приложения. Затем мы освобождаем память и контекст вывода. Это действие является предельно важным. Если его не выполнить, то приложение просто закроет доступ к рабочему столу для всех остальных приложений и для операционной системы. Страшная ситуация!

Поработайте с проектом. При перемещении окна все работает так, как мы того ожидаем, но вот если размеры окна сделать слишком маленькими, то кружок может выходить за его пределы. Это объяснимо. Мы знаем, что круг рисуется на поверхности всего экрана. Если же окно приложения перекрыть

другим окном или минимизировать, а затем распахнуть, то кружок уже не появляется.

Мы подходим к очень важным вопросам, специфичным именно для DirectDraw. Ситуацию, когда приложение временно убирается с экрана, а затем восстанавливается, необходимо в приложениях фиксировать.

Посмотрите проект каталога Ex03, в котором отслеживается ошибка, возникающая при такой ситуации, и пользователю выдается осмысленное сообщение (рис. 2.1).

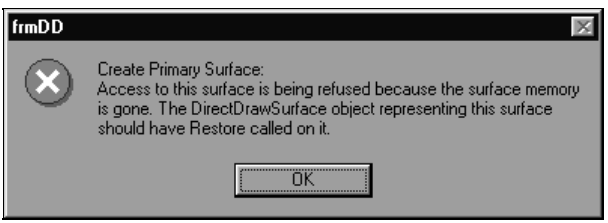


Рис. 2.1. При использовании DirectDraw необходимо следить за потерей доступа к поверхности

Код ошибки — `DDERR_SURFACELOST`. Как сообщается в его расшифровке, в таком случае необходимо использовать метод `Restore` поверхности.

Пример делает прозрачным причины потери поверхности. Первичную поверхность мы связали с рабочим столом, и, конечно, после того, как окно "ушло" с экрана, оно освободило память поверхности для других приложений. Теперь ему надо заново вернуть свои исключительные права на область памяти. Такая же ситуация возникает при изменении параметров рабочего стола по ходу работы приложения.

Также многие из функций DirectDraw могут вернуть код ошибки `DDERR_WASSTILLDRAWING`, означающий, что аппаратное обеспечение занято и запрос необходимо повторять до тех пор, пока не добьемся успеха или не получим иного сообщения об ошибке.

Взгляните на проект каталога Ex04, здесь решены все эти проблемы, и первое, что изменилось, — это код, связанный с перерисовкой окна. Теперь код собственно воспроизведения заключен внутрь цикла, из которого мы выходим либо в случае успешного воспроизведения, либо если поверхность восстановить не удается, либо код ошибки отличен от `DDERR_WASSTILLDRAWING`:

```
while True do begin возможно,       // Код придется повторять неоднократно
  hRet := FDDSPrimary.GetDC(DC);    // Заново получаем дескриптор
  if Succeeded (hRet) then begin
    wrkCanvas := TCanvas.Create;
    wrkCanvas.Handle := DC;
    wrkCanvas.Ellipse (Left + 50, Top + 50, Left + 100, Top + 100);
```

```
    wrkCanvas.Free;
    FDDSPrimary.ReleaseDC (DC);
    Break;
  end;
  // Поверхность потеряна, надо восстановить
  if hRet = DDERR_SURFACELOST then begin
    hRet := FDDSPrimary._Restore;
    // Если не удалось восстановить, дальше продолжать нельзя
    if hRet <> DD_OK then Break;
  end;
  // Ошибка отлична от DDERR_WASSTILLDRAWING, следовательно непоправима
  if hRet <> DDERR_WASSTILLDRAWING then Break;
end;
```

Чтобы кружок не рисовался за пределами окна приложения, можно просто не разрешать уменьшать высоту окна. Таким образом, появился обработчик события `OnCanResize`:

```
procedure TfrmDD.FormCanResize(Sender: TObject; var NewWidth,
  NewHeight: Integer; var Resize: Boolean);
begin
  if NewHeight < 110      // Высота окна не должна быть меньше 110
    then Resize := False
    else Resize := True;
end;
```

Что еще надо сделать, так это при обработке события `OnResize` окна вызывать тот же код, что и при событии `OnPaint`.

Для обработки тех ситуаций, когда восстановить поверхность не удается, в проект добавлен компонент класса `TApplicationEvents`, на события `OnActivate` и `OnRestore` которого вызывается такой же код, как и при создании окна. То есть при восстановлении минимизированного окна и каждой активизации окна приложения заново создаем первичную поверхность.

Хорошенько поработайте с проектом: протестируйте его работу в самых различных ситуациях, минимизируйте и восстанавливайте окно, активизируйте самыми различными способами, поменяйте установки экрана по ходу работы этого приложения. Кружок должен появляться всегда, когда мы его ожидаем. При деактивизации окно может вести себя непривычно для обычных приложений, можете записать `Application.Minimize` в обработчике события `OnDeactivate` единственного компонента проекта. Восстанавливается окно тоже особым образом, распахиваясь на весь экран.

Такое использование полноэкранной первичной поверхности, как в этом примере, когда воспроизведение осуществляется только функциями GDI в пределах окна приложения, редко применяется в практических задачах.

Примечание

В примере есть небольшое упрощение. Так как при восстановлении окна приложения и его активизации (пользователь переходит на него с помощью комбинации клавиш <Alt>+<Tab>) поверхность создается заново, то она никогда не будет потеряна. Такой прием можно использовать только для простейших приложений, поскольку весьма неэкономно тратить время подготовки работы при каждой активизации приложения.

Блиттинг

Блиттингом называется копирование графических изображений в память видеоустройств, и DirectDraw представляет собой просто механизм блиттинга.

Начнем знакомство с этим механизмом с помощью проекта каталога Ex05. Пример является продолжением предыдущей программы. Первичная поверхность все также полноэкранная, на нее с помощью вспомогательной канвы выводится растровое изображение. Это изображение, используемое здесь и во многих последующих примерах, взято из DirectX SDK производства Microsoft. Содержит эта картинка потрясающий по своей красоте пейзаж.

Главное отличие данного примера от предыдущего состоит в том, что поверхность, служащая фоном нашего растра, закрашивается. Работа приложения теперь выглядит естественной для полноэкранных приложений вообще, и для приложений, использующих DirectDraw, в частности.

Поскольку это полноэкранное приложение, то обработчики событий `OnCanResize` и `OnResize` ему не нужны. Не пропустите также, что свойство `BorderStyle` формы я установил в `bsNone`. Это важно, если этого не сделать, то приложение будет работать прекрасно, но при движении курсора вблизи границ экрана и в районе системного заголовка окна приложения сквозь "экран" будет проглядывать другое окно. Обязательно проверьте это, вернув обычное значение указанного свойства формы. Полноэкранная поверхность занимает рабочий стол, загораживает собой все окна, но они продолжают реагировать на поступающие им сообщения.

Чтобы при восстановлении приложения его окно появлялось распахнутым на весь экран, появился обработчик события `OnRestore` компонента `ApplicationEvents1`:

```
procedure TfrmDD.ApplicationEvents1Restore(Sender: TObject);
begin
  WindowState := wsMaximized;
end;
```

Следует задавать это свойство именно динамически, т. е. принудительно распахивать окно при каждом его восстановлении.

Главные изменения коснулись кода, связанного с перерисовкой окна. Часть этого кода, предназначенную для вывода изображения, рассматривать не будем. Здесь ничего нового для нас нет. Посмотрим внимательно на то, что ему предшествует, а именно на заполнение фона черным цветом. Кстати, обратите внимание, что порядок действий такой: закрашиваем фон, на закрашенный фон выводим растр. Замечу, что работа с растром в примере выполнена не оптимально, растр загружается при каждой перерисовке окна. Пока наши примеры очень просты и вполне терпят подобное, но при интенсивной работе с растровыми изображениями, конечно, так писать программы не будем.

Процедура обработчика перерисовки окна имеет локальную вспомогательную переменную `ddbltfx` типа `TDDBLTFX`, который представляет собой запись и является вспомогательным, подобным использованному нам при создании поверхности типу `TDDSurfaceDesc2`, но применяется лишь для задания параметров блиттинга. Поля этой структуры мы изучим по ходу дальнейшего изложения. Пока ограничимся тем, что значение поля `dwFillColor` задает цвет заполнения поверхности. Как указывалось ранее, работа с подобными структурами начинается с обнуления всех полей и задания ее размера указанием значения поля `dwSize`:

```
ZeroMemory(@ddbltfx, SizeOf (ddbltfx));   // Обнуляем все поля
ddbltfx.dwSize := SizeOf (ddbltfx);       // Обязательно задаем размер
ddbltfx.dwFillColor := 0;                 // Цвет заполнения — черный
```

Для блиттинга используем метод `Blt` первичной поверхности:

```
hRet := FDDSPrimary.Blt(nil, nil, nil, DDBLT_COLORFILL or DDBLT_WAIT,
                        @ddbltfx);
```

Первым трем аргументам присваиваем значение `nil` (смысл их раскроим позже). Четвертый аргумент метода является битовым флагом: у нас это комбинация двух констант. Константа `DDBLT_WAIT` задает режим ожидания для вывода. В этом режиме вывод станет осуществляться, когда устройство будет готово к нему, и поэтому метод не возвратит никогда значение `DDERR_WASSTILLDRAWING`. Флаг `DDBLT_COLORFILL` сообщает методу `Blt` о том, что вместо блиттинга выполняется цветовое заполнение. Последний аргумент указывает адрес переменной, хранящей параметры блиттинга.

Возвращаемое методом значение мы анализируем, подобно тому, как это делали в предыдущем примере. Только здесь мы покидаем цикл при любой ошибке, отличной от ошибки, связанной с потерей поверхности, или при отсутствии ошибок.

Примечание

Строго говоря, нам следовало бы выделить ситуацию с неустранимой ошибкой, ведь в этом случае надо прекращать выполнение кода, а не просто завершать цикл.

Протестируйте пример. Приложение должно функционировать надежно в самых различных ситуациях, но если по ходу его работы изменить настройки экрана и установить его новые размеры больше предыдущих, то при восстановлении приложения оно не будет закрывать весь экран, а размеры его станут такими же, как при первоначальном запуске. Заметна еще одна проблема: если до запуска приложения установить режим экрана в 256 цветов, образ выводится совсем не красочным. Чтобы избежать таких ошибок, обычно полноэкранные приложения устанавливают разрешения экрана сообразно со своими нуждами.

Переходим к иллюстрации — проекту каталога Ex06. От предыдущего он отличается только тем, что код обработчика `OnCreate` формы дополнился строками:

```
hRet := FDD.SetDisplayMode (640, 480, 16, 0, 0); // Задаем режим
if hRet <> DD_OK then begin    // Обязательно анализируем результат
  ErrorOut(hRet, 'SetDisplayMode');
  Exit;
end;
```

Установка режима является методом главного объекта, поэтому должна происходить строго после его создания, но не обязательно первым же действием. Первые три аргумента метода, надеюсь, сразу же ясны: высота, ширина экрана и разрешение (число бит, необходимых для определения цвета пиксела). Последние два аргумента метода всегда будем задавать нулевыми. Первый из них определяет частоту регенерации. При нулевом значении параметра отдается на усмотрение DirectDraw. Последний аргумент задает дополнительные флаги, пока из них доступен только `DDSDM_STANDARDVGAMODE`, связанный с особым режимом Mode X (320×200×8).

Итак, на время работы приложения мы задаем режим 640×480×16. Эта тройка чисел не может браться наобум, а должна принадлежать набору поддерживаемых системой режимов.

Совет

Запустив утилиту диагностики DirectX, вы можете найти список поддерживаемых режимов.

Если на вашей карте выводимое изображение теряет в красочности по сравнению с исходным 24-разрядным растром, установите этот режим 24- или 32-битным, в зависимости от того, какой из них доступен.

Обратите внимание, что нет необходимости по окончании работы приложения возвращаться к режиму, установленному пользователем. Это произойдет автоматически.

Разобравшись с данным примером, мы можем перейти к рассмотрению процедуры настоящего блиттинга. Рассмотрим проект из каталога Ex07. Коренное отличие его от предыдущего состоит в том, что образ помещаем на

вспомогательную поверхность, а при воспроизведении осуществляется блиттинг на первичную поверхность.

Раздел `private` описания класса формы дополнился строкой объявления дополнительной поверхности, предназначенной для хранения образа:

```
FDDSImage  :  IDirectDrawSurface7;
```

Код обработчика события `OnCreate` начинается с того, что этой переменной присваивается значение `nil`, а при завершении работы приложения освобождается память в порядке, обратном связыванию переменных:

```
if Assigned(FDD) then begin
  if Assigned(FDDSImage) then FDDSImage := nil;     // Перед первичной
  if Assigned(FDDSPrimary) then FDDSPrimary := nil; // поверхностью
  FDD := nil;
end;
```

Растровое изображение считывается только один раз. У обработчика `OnCreate` появился вспомогательный объект `wrkBitmap` класса `TBitmap`. Вторичная поверхность создается после первичной и заполняется считанным растром:

```
wrkBitmap := TBitmap.Create;             // Создание объекта растра
wrkBitmap.LoadFromFile ('..\lake.bmp'); // Считывание файла растра
// Напоминаю, что обнулять поля можно и с помощью ZeroMemory
FillChar (ddsd, SizeOf(ddsd), 0);
with ddsd do begin // Как для любой записи, можно использовать with
  dwSize := SizeOf(ddsd); // Обязательное действие
  // Будем задавать размеры поверхности (+ DDSD_HEIGHT и DDSD_WIDTH)
  dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
  ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN; // Внеэкранная поверхность
  dwWidth  := wrkBitmap.Width;  // Ширина поверхности равна ширине растра
  dwHeight := wrkBitmap.Height; // Задаем высоту поверхности
end;     // with
// Собственно создание вспомогательной поверхности
hRet := FDD.CreateSurface(ddsd, FDDSImage, nil);
if hRet <> DD_OK then begin   // Анализируем на предмет успешности
  ErrorOut(hRet, 'Create Image Surface');
  Exit;
end;
// Копирование растра из wrkBitmap во вспомогательную поверхность
hRet := DDCopyBitmap (FDDSImage, wrkBitmap.Handle, 0, 0, wrkBitmap.Width,
                      wrkBitmap.Height);
if hRet <> DD_OK then begin   // Обязательно анализируем результат
  ErrorOut(hRet, 'DDCopyBitmap');
  Exit;
end;
// Удаление вспомогательного объекта
wrkBitmap.Free;
```

Вспомогательная, внеэкранная поверхность FDDSImage создается с описанием DDSCAPS_OFFSCREENPLAIN. Здесь есть некоторые нюансы, но пока рассматривать их не будем.

После создания вторичной поверхности заполняем ее растровым изображением с помощью вспомогательной функции DDCopyBitmap из модуля DDUtil, не забываем дописать имя модуля после uses. В тонкости того, как осуществляется копирование, можете не вникать или разберитесь позднее самостоятельно. Код данной функции основан на функциях API. Ключевым является вызов StretchBlt.

Вспомогательная поверхность создана и заполнена растром размером 256×256 пикселов. Среди аргументов операции блиттинга присутствуют структуры типа TRECT, задающие местоположение в принимающей поверхности и копируемую область. Поэтому код обработчика перерисовки окна дополнился переменными dstRect и srcRect типа TRECT. Заполняем их поля с помощью API-функции SetRect:

```
SetRect (dstRect, 100, 100, 356, 356);   // Для принимающей поверхности
SetRect (srcRect, 0, 0, 256, 256);       // Для источника
```

Теоретически эта операция также может привести к провалу. Для важных приложений рекомендую здесь анализировать возвращаемое булево значение. К тому же соглашусь, что в данном примере оптимальным решением было бы использование глобальных переменных, заполняемых один раз, а не при каждой перерисовке окна. Просто код в таком виде удобнее читать, а перерисовка не станет производиться интенсивно.

Канву для вывода растра не используем, делаем теперь все традиционным для DirectDraw способом:

```
while True do begin   // Возможно, придется производить неоднократно
  hRet := FDDSPrimary.Blt (@dstRect, FDDSImage, @srcRect, DDBLT_WAIT,
nil); // Собственно блиттинг
  if hRet = DDERR_SURFACELOST then begin     // Поверхность потеряна
    if Failed (RestoreAll) then Exit;        // Пытаемся восстановить
  end else Break;     // Или все прошло успешно, или неустранимая ошибка
end;
```

Потеря любой поверхности является верным признаком того, что надо восстанавливать все поверхности. Поэтому каждый раз в случае ошибки обращаемся к пользовательской функции RestoreAll:

```
function TfrmDD.RestoreAll : HRESULT;
begin
  Result := DD_FALSE;     // Определяемся с результатом
  // Пытаемся восстановить первичную поверхность
  if Succeeded (FDDSPrimary._Restore) then begin
```

```
    // Пытаемся восстановить вторичную поверхность
    if Failed (FDDSImage._Restore) then Exit;
    Result := DD_OK;       // Все прошло успешно
  end;
end;
```

Нажав комбинацию клавиш <Alt>+<Tab>, переключитесь с этого приложения, а затем верните ему фокус. Если восстановление поверхностей прошло успешно, вы увидите картинку с пейзажем. Но если это получилось с вашей картой, совсем не обязательно, что это произойдет и с другими. На иных компьютерах пользователи в такой ситуации могут получить бессмысленный узор. Согласно рекомендациям разработчиков, поверхности, содержащие растр, при восстановлении должны заново заполняться.

Если это окно из рассматриваемого примера у вас восстанавливается без потерь, можете двигаться дальше. Если же у вас картинка при восстановлении портится, функцию восстановления исправьте следующим образом:

```
function TfrmDD.RestoreAll : HRESULT;
var
  hRet : HRESULT;
begin
  hRet := FDDSPrimary._Restore;
  if Succeeded (hRet) then begin
    hRet := FDDSImage._Restore;
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    // Перезагружаем на поверхность содержимое растра
    Result := DDReLoadBitmap(FDDSImage, imageBMP);
  end else Result := hRet;
end;
```

Теперь мы можем узнать смысл первых трех аргументов метода `Blt` поверхности. Первый из них — указатель на структуру типа `TRECT`, задающую местоположение и размер области, в которую происходит копирование. Второй параметр — поверхность источника. Третий аргумент — указатель на структуру типа `TRECT`, задающую местоположение и размер области, из которой происходит копирование.

Флагом задаем константу `DDBLT_WAIT`, не комбинацию значений. Дополнительные параметры пока не указываем, поэтому последний аргумент метода устанавливаем в `nil`.

Пример простой, но очень важный. Осмыслим изученное. Естественным для DirectDraw способом воспроизведения является блиттинг. На вспомогательных поверхностях размещаем нужные нам образы, а в определенный момент времени копируем требуемые области с одной поверхности на дру-

гую, в простейшем случае — со вспомогательных поверхностей на первичную, связанную с экраном.

Вторичных поверхностей создают столько, сколько требуется приложению. Разработчик сам решает, что и где ему располагать, но здесь надо учесть небольшую тонкость: если видеокарта имеет малый размер памяти, то вторичную поверхность не получится создать размером больше первичной. Может быть, это происходит только с конкретными картами, но я действительно встречался с такой ситуацией.

Поверхностей вы можете создавать сколько угодно, но чем их меньше, тем быстрее будет осуществляться блиттинг. Поэтому лучше не плодить множество поверхностей, а, в идеальном случае, располагать все образы на одной большой поверхности.

В общем случае операция копирования блоков из системной памяти в видеопамять осуществляется медленнее, чем из видеопамяти в видеопамять. Поэтому образы, выводимые наиболее часто, старайтесь размещать в видеопамяти, а при ее нехватке те из образов, которые редко будут появляться, например заставки или меню, следует размещать в системной памяти.

Поддержка AGP если и стирает разницу в скоростях обмена, то незначительно. Если бы скорости работы с видеопамятью и системной памятью совпадали или были близки друг к другу, не было бы нужды производителям карт выпускать модели, различающиеся размером видеопамяти, а пользователям оставалось лишь наращивать размер системной памяти.

Для поверхности, создаваемой в видеопамяти, надо использовать комбинацию флагов `DDSCAPS_OFFSCREENPLAIN or DDSCAPS_VIDEOMEMORY`. И наоборот, флаг `DDSCAPS_SYSTEMMEMORY` указывает, что поверхность должна располагаться в системной памяти.

Метод `GetAvailableVidMem` главного объекта DirectDraw позволяет выяснить, сколько видеопамяти осталось в распоряжении приложения.

При нехватке видеопамяти операция создания поверхности завершится провалом, код соответствующей ошибки — `DDERR_OUTOFVIDEOMEMORY`. В этом случае необходимо поменять флаг и заново попытаться создать поверхность.

Теперь обсудим вопросы масштабирования растров. Вспомогательная функция копирования растра поддерживает масштабирование. Задайте размер вторичной поверхности больше, чем размер используемого растра, например, так:

```
ddsd.dwWidth  := wrkBitmap.Width * 2;
```

Теперь при воспроизведении мы увидим картинку, растянутую вдоль экрана, но не всю, а только ее половину. Для того чтобы вывести ее целиком, надо изменить значение поля `Right` структуры `dstRect`:

```
SetRect (dstRect, 100, 100, 100 + 256 * 2, 356);
```

Попробуем перемещать картинку по экрану (проект каталога Ex08). Переменная `lft` хранит текущее значение смещения картинки, на это значение опираемся при заполнении полей структуры `dstRect`:

```
SetRect (dstRect, lft, 100, lft + 256, 356);
```

Форма дополнилась обработчиком нажатия клавиши: демонстрационные программы, использующие DirectDraw, традиционно должны завершать работу при нажатии клавиши <Esc> или <F12>. Добавилась также обработка нажатий клавиш управления курсором:

```
case Key of
  VK_ESCAPE, VK_F12 : begin   // Традиция для DirectDraw
                        Close;
                        Exit;
                      end;
  VK_LEFT  : begin            // Клавиша "стрелка влево"
               Dec (lft, 1);     // Уменьшаем lft
               FormPaint (nil);  // Перерисовываем экран
             end;
     VK_RIGHT : begin         // Клавиша "стрелка вправо"
                  Inc (lft, 1);       // Увеличиваем lft
                  FormPaint (nil);    // Перерисовываем экран
                end;
end;
```

Обратите внимание, что для перерисовки окна метод `Refresh` не годится, иначе сквозь экран будет проглядывать мелькнувшее окно приложения. Картинка движется с малым шагом с целю убедить вас, что если хоть один пиксел растра не помещается на первичную поверхность, не воспроизводится ничего.

Также наверняка вам бросится в глаза то, что картинка появляется медленно на экране, при ее воспроизведении вы можете увидеть мельтешение черных полос. Пока старайтесь не обращать на это внимания. В будущем мы устраним это — такого не должно быть в наших серьезных проектах.

Сейчас сделайте следующее. Запишите строку, задающую параметры области вывода так:

```
SetRect (dstRect, lft, 100, lft + 512, 356);
```

Картинка выводится растянутой, из чего делаем важный вывод: метод `Blt` поверхности поддерживает операцию масштабирования. Удобное для нас свойство, им можно пользоваться, чтобы задавать в качестве фона растровое изображение любого размера. Для этого измените ту же строку вот так:

```
SetRect (dstRect, 0, 0, ClientWidth, ClientHeight);
```

Теперь код, заполняющий первичную поверхность черным цветом, можно просто удалить. Он не нужен, его выполнение только отнимает драгоценное время.

Плохо в получающемся примере то, что размер растра используется в нем дважды: при задании размеров первичной поверхности и при задании области вывода. Здесь нас ждет хорошая новость: во втором случае можно ничего не указывать, по умолчанию будет использоваться вся поверхность, и строку блиттинга можно записать так:

```
hRet := FDDSPrimary.Blt (@dstRect, FDDSImage, nil, DDBLT_WAIT, nil);
```

То есть третий параметр равен `nil`. Обязательно проверьте это, все должно работать как следует.

Совет

С точки зрения оптимизации лучше явно задавать размер копируемой поверхности.

Протестируйте работу программы при переключении и восстановлении и, если картинка пейзажа теряется, скорректируйте код функции `RestoreAll`.

Я воспользуюсь случаем, чтобы посвятить вас в еще одну важную тему: в любой момент времени мы можем получить информацию обо всех свойствах поверхности, в том числе и о ее размерах. Для этого предназначен метод поверхности `GetSurfaceDesc`.

Иллюстрацией служит проект каталога Ex09. Код обработчика `OnPaint` формы дополнился локальной переменной `ddsd2` типа `TDDSurfaceDesc2`, перед блиттингом с ней производятся обычные действия (обнуление всех полей и задание размера), используется она с целью хранения информации о параметрах поверхности, для получения которых и вызывается изучаемый метод:

```
// В ddsd2 занести данные о поверхности
FDDSImage.GetSurfaceDesc (ddsd2);
// Размеры srcRect устанавливаются равными размерам поверхности
SetRect (srcRect, 0, 0, ddsd2.dwWidth, ddsd2.dwHeight);
```

Сейчас в качестве упражнения рекомендую выполнить следующее задание: создайте простейшую программу просмотра bmp-файлов. После загрузки приложения пользователь выбирает нужный файл с помощью стандартного диалога. Растр выводится на полный экран.

Еще один простой пример по поводу блиттинга — проект каталога Ex10. Здесь экран раскрашивается подобно радуге (рис. 2.2).

Используется растр размером 1024×1, т. е. высотой в один пиксел. Не забывайте, что карты с небольшой видеопамятью не способны создать вторичную поверхность больше первичной. Некоторые читатели не смогут насла-

диться всей красотой этого примера, но ничего не потеряют, поскольку следующий проект выводит все тот же растр, и должен работать на всех картах.

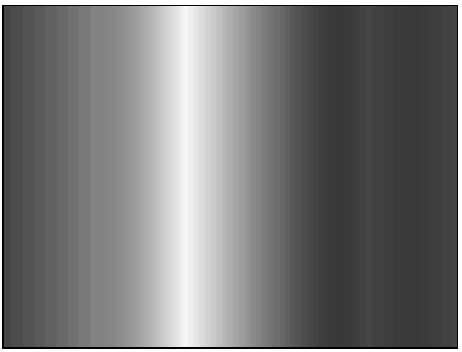


Рис. 2.2. Пример на масштабирование растра

В проекте каталога Ex11 я напоминаю о другом способе масштабирования растров, обычном для Delphi. При создании вторичной поверхности растровое изображение все также загружается в объект `wrkBitmap`. Затем создается вспомогательный объект `wrkBitmap1`, его ширина — 640 пикселов, высота — 1 пиксел. После чего "масштабируется" прежний растр и выводится на канве `wrkBitmap1` с помощью метода `StretchDraw`:

```
wrkBitmap1.Canvas.StretchDraw (Rect (0, 0, wrkBitmap1.Width,
                                wrkBitmap1.Height), wrkBitmap);
```

Размеры вторичной поверхности теперь должны опираться на размеры именно второго растра.

Такой способ масштабирования более эффективен. Задайте высоту растра равной 60 пикселам, и радуга должна заполнить экран гораздо быстрее, чем в двух предыдущих способах, поскольку меньше тратится времени при окончательном растяжении вторичной поверхности.

Упражнение: сделав `wrkBitmap1` глобальной переменной, добейтесь уверенного восстановления изображения.

Аналогичный прием со вспомогательным объектом класса `TBitmap` используется в очередном примере (проекте каталога Ex12), в котором образ загружается из jpg-файла, а при выводе картинка заключается в рамку (рис. 2.3).

В списке `uses` добавлены модули `extctrls` и `jpeg` для использования динамически создаваемого объекта `Image` класса `TImage`, в который будет загружаться jpg-файл :

```
Image := TImage.Create (nil);                // Создаем объект
Image.Picture.LoadFromFile ('..\lake.jpg');  // Загружаем jpg
```

```
// Непосредственно Image использовать не сможем
wrkBitmap := TBitmap.Create;    // Вспомогательный Bitmap
wrkBitmap.Width := 640;   // Размеры — все окно, чтобы не было искажений
wrkBitmap.Height := 480;
wrkBitmap.Canvas.Brush.Color := clBlue;   // Фон прямоугольника рамки
wrkBitmap.Canvas.Pen.Color := clRed;      // Рамка обрамляется красным
wrkBitmap.Canvas.Pen.Width := 5;          // Толщина карандаша
wrkBitmap.Canvas.Rectangle (150, 100, 490, 380); // Рамка
// Воспроизводим jpg на канве
wrkBitmap.Canvas.Draw (192, 112, Image.Picture.Graphic);
Image.Free; // Image больше не нужен
```

Рис. 2.3. В примере канва используется только при создании вторичной поверхности

Канва в примере используется только при подготовке поверхности, посему мы не потеряем в скорости при воспроизведении.

Будьте внимательны, основной фон экрана в рассматриваемом примере — серый, поскольку за нашей картинкой выступает поверхность основного окна. Такое сочетание вывода функциями GDI и командами DirectDraw вообще-то надо избегать, заполняя весь фон вторичной поверхности.

Если вы внимательно исследуете содержимое заголовочного файла DirectDraw.pas, то легко сможете обнаружить, что свойства блиттинга гораздо шире изученных нами. Например, поверхность можно вращать при выводе. Удобная возможность, но предоставляется только акселератором, причем далеко не каждым. Поэтому изучить вам это придется самостоятельно.

А мы перейдем к другому методу поверхности, осуществляющему блиттинг — методу `BltFast`. Рассмотрим пример, представленный в проекте ка-

талога Ex13. Картинка загружается из jpg-файла, внеэкранная поверхность должна закрывать собой весь экран:

```
wrkBitmap.Width := 640;       // По размерам совпадает с устанавливаемым
wrkBitmap.Height := 480;      // экранным режимом
wrkBitmap.Canvas.Brush.Color := clBlack;   // Фон экрана установим черным
wrkBitmap.Canvas.Rectangle (0, 0, 640, 480);     // Закрасим весь экран
wrkBitmap.Canvas.Draw (192, 112, Image.Picture.Graphic); // Вывод jpg
```

Воспроизведение основано на методе `BltFast`:

```
hRet := FDDSPrimary.BltFast (0, 0, FDDSImage, nil, DDBLTFAST_WAIT);
```

Первые два аргумента задают координаты (x, y) левого верхнего угла размещаемого блока в принимающей поверхности. Дальше указывается вставляемая поверхность. Предпоследний аргумент — величина типа `TRECT` — задает вырезаемую из вставляемой поверхности область. Точно так же, как и в случае с методом `Blt`, желательно явно задавать размеры, даже в случае, когда поверхность вставляется целиком. Последний аргумент определяет условия работы блиттинга. Пока мы задаем одиночное значение. Константа изменилась, но смысл ее использования аналогичен `DDBLT_WAIT`.

Метод `BltFast` более привлекателен в использовании и работает быстрее. Но он имеет некоторые ограничения в сравнении с методом `Blt`, например, не предоставляет возможности автоматического масштабирования, не может использоваться для заполнения фона так, как мы это делали раньше.

Буферы

Итак, будем стараться использовать метод `BltFast` всегда, когда это возможно, т. к. скорость работы приложения является для графики наиважнейшей характеристикой. Посмотрим, как выглядит с применением этого метода перерисовка в проекте каталога Ex14. С помощью клавиш перемещения курсора можно управлять положением картинки на экране по горизонтальной оси. Хотя скорость воспроизведения и увеличилась, но мельтешение полос при перерисовке картинки осталось. Экран мы заполняем в два этапа: вначале все закрашиваем черным цветом, затем накладываем картинку. В промежуток времени между этими действиями экран успевает обновиться, и мы видим эти раздражающие полосы.

Попробуем проделать следующее: создадим внеэкранную вспомогательную поверхность, по размерам равную первичной. Воспроизводить изображение будем на нее, а окончательную картинку перенесем на экранную поверхность. Иллюстрацией этого ловкого приема служит проект каталога Ex15.

Поскольку значения устанавливаемых параметров экрана в коде используются неоднократно, введем соответствующие константы:

```
ScreenWidth    = 640;
ScreenHeight   = 480;
ScreenBitDepth = 16;
```

Для загрузки растра вызовем вспомогательную функцию `DDLoadBitmap` из модуля `DDUtil`, объединяющую создание поверхности, собственно загрузку и копирование растра на поверхность:

```
FDDSImage := DDLoadBitmap(FDD, szBitmap, 0, 0);   // Укороченный код
if FDDSImage = nil then begin                     // Произошла ошибка
  ErrorOut(hRet, 'DDLoadBitmap');
  Exit;
end;
```

Функция пытается загрузить растр из ресурсов, в случае неудачи загружает файл. Первым аргументом задается главный объект DirectDraw, затем имя файла — у нас это константа `szBitmap`, — дальше указываются требуемые размеры поверхности или нули, если поверхность должна иметь размеры растра.

Вспомогательная поверхность носит имя `FDDSBack` и в начале и конце работы приложения "обнуляется" самой первой. Ее размеры задаются равными размерам первичной поверхности:

```
FillChar (ddsd, SizeOf(ddsd), 0);
with ddsd do begin
  dwSize := SizeOf(ddsd);
  dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
  ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
  dwWidth  := ScreenWidth;
  dwHeight := ScreenHeight;
end;
hRet := FDD.CreateSurface(ddsd, FDDSBack, nil);
if hRet <> DD_OK then begin
  ErrorOut(hRet, 'Create Back Surface');
  Exit;
end;
```

Перерисовка окна принципиально изменилась тем, что весь вывод осуществляется на вспомогательную поверхность, содержимое которой в законченном виде помещается на первичную:

```
ZeroMemory(@ddbltfx, SizeOf(ddbltfx));
ddbltfx.dwSize := SizeOf(ddbltfx);
ddbltfx.dwFillColor := 0;
while True do begin // Закрашиваем фон вторичной поверхности черным
  hRet := FDDSBack.Blt(nil, nil, nil, DDBLT_COLORFILL or DDBLT_WAIT,
@ddbltfx);
```

```
  // Внеэкранная поверхность также может быть потеряна
  if hRet = DDERR_SURFACELOST then begin
    if Failed (RestoreAll) then Exit;
  end
  else Break;
end;
// Помещаем растр на вспомогательную поверхность
while True do begin
  hRet := FDDSBack.BltFast (lft, 112, FDDSImage, nil, DDBLTFAST_WAIT);
  if hRet = DDERR_SURFACELOST then begin
    if Failed (RestoreAll) then Exit;
  end
  else Break;
end;
// Вспомогательная поверхность заполнена, блиттинг производится
// на первичную
while True do begin
  hRet := FDDSPrimary.BltFast (0, 0, FDDSBack, nil, DDBLTFAST_WAIT);
  if hRet = DDERR_SURFACELOST then begin
    if Failed (RestoreAll) then Exit;
  end
  else Break;
end;
```

Функция восстановления поверхности использует вспомогательную функцию перезагрузки растра `DDReLoadBitmap` модуля `DDUtil`:

```
function TfrmDD.RestoreAll : HRESULT;
begin
  Result := DD_FALSE;
  if Succeeded (FDDSPrimary._Restore) then begin
    if Failed (FDDSImage._Restore) then Exit;
    // Рекомендуется перезагрузить растр после восстановления
    if Failed (DDReLoadBitmap(FDDSImage, szBitmap)) then Exit;
    // Добавилось восстановление еще одной вспомогательной поверхности
    if Failed (FDDSBack._Restore) then Exit;
    Result := DD_OK;
  end;
end;
```

Протестируйте приложение: никаких полос не возникает, все выглядит прекрасно.

DirectDraw предлагает автоматизированный механизм двойной буферизации, аналогичный проделанному нами вручную. Посмотрим на примере проекта каталога Ex16, как это делается. При создании первичной поверхности указываем количество задних буферов. Вместо одной константы у нас

появилась комбинация нескольких флагов. Создаваемая поверхность является комплексной, состоящей из двух поверхностей — первичной и присоединенной к ней вторичной поверхности заднего буфера. Чтобы оговорить то, что "перебрасывание" (flipping) содержимого заднего буфера на первичную поверхность будет осуществляться DirectDraw без нашего участия, необходимо добавить флаг DDSCAPS_FLIP:

```
FillChar (ddsd, SizeOf(ddsd), 0);
ddsd.dwSize := SizeOf(ddsd);
// Сообщаем, что надо учесть наши пожелания о буфер заднего плана
ddsd.dwFlags := DDSD_CAPS or DDSD_BACKBUFFERCOUNT;
ddsd.ddsCaps.dwCaps := DDSCAPS_PRIMARYSURFACE or DDSCAPS_FLIP or
DDSCAPS_COMPLEX; // + комплексная поверхность + разрешить перебрасывание
ddsd.dwBackBufferCount := 1; // У поверхности есть один задний буфер
hRet := FDD.CreateSurface(ddsd, FDDSPrimary, nil);
if hRet <> DD_OK then begin
  ErrorOut(hRet, 'Create Primary Surface');
  Exit;
end;
```

Поверхность заднего буфера нам создавать не нужно, она появится без нашего участия, но для осуществления вывода на нее необходимо связать нашу переменную FDDSBack, для чего предназначен метод поверхности GetAttachedSurface. Первый аргумент метода — запись типа TDDSCaps2. С таким типом мы встречались, он является частью структуры TDDSurfaceDesc2. Здесь же указываем, что нам требуется адрес поверхности заднего буфера:

```
FillChar(ddscaps, SizeOf(ddscaps), 0); // Обнуляем все поля записи
// Оговариваем, что требуется адрес поверхности заднего буфера
ddscaps.dwCaps := DDSCAPS_BACKBUFFER;
// Получаем адрес присоединенной поверхности
hRet := FDDSPrimary.GetAttachedSurface(ddscaps, FDDSBack);
if hRet <> DD_OK then begin
  ErrorOut(hRet, 'GetAttachedSurface');
  Exit;
end;
```

Код воспроизведения изменился только в финальной части, вместо использования метода BltFast первичной поверхности вызываем ее метод Flip:

```
while True do begin
  hRet := FDDSPrimary.Flip(nil, DDFLIP_WAIT);
  if hRet = DDERR_SURFACELOST then begin
    if Failed (RestoreAll) then Exit;
  end
  else Break;
end;
```

Первым аргументом метода указывается, при необходимости, адрес конкретной поверхности; вторым аргументом задается набор параметров, определяющих режим переключения.

Примечание

Присоединенная поверхность не требует отдельного восстановления, она будет восстановлена как часть первичной, комплексной поверхности.

Обращаю внимание, что в программах, написанных на Delphi, необходимо обязательно при завершении работы освобождать присоединяемые поверхности, иначе возникнет исключение.

При использовании метода поверхности `Flip` не происходит, на самом деле, простого воспроизведения на ней так, как вытекает из моих предыдущих рассуждений. Буферы меняются местами, вернее, происходит переключение указателей (адресов). Сами объекты при этом местами не меняются.

Переходим к очередному примеру — проекту каталога Ex17. Смысл примера состоит в следующем: поместим на переднюю и заднюю поверхности разные образы и с течением времени будем только переключать их, не перерисовывая.

На переднюю поверхность я помещаю растянутую картинку с пейзажем, на задней поверхности (она как раз и является задним буфером) нарисован тот же пейзаж посреди черного поля. Код переключения буферов из обработчика `OnPaint` формы переместился в обработчик единственного события таймера, а начинается код, связанный с перерисовкой окна, с заполнения переднего буфера:

```
hRet := FDDSPrimary.Blt (nil, FDDSImage, nil, 0, nil);
```

Эта строка, а также код, связанный с заполнением заднего буфера, может спокойно перекочевать в обработчик `OnCreate` формы, он в этом примере вызывается и при восстановлении окна.

Чтобы таймер не работал при неактивном состоянии приложения, применяется событие объекта `ApplicationEvents1`, связанное с "уходом" окна приложения:

```
procedure TfrmDD.ApplicationEvents1Deactivate(Sender: TObject);
begin
  Timer1.Enabled := False;    // Выключаем таймер
  Application.Minimize;       // Минимизируем приложение
end;
```

Включается же таймер в обработчике `OnCreate`, специально для обработки восстановления окна.

Заполнение черным или любым другим цветом поверхности заднего плана для подобных примеров является обязательным. Если этого не делать, то в незаполненных участках буфера будет выводиться "мусор", искаженные

следы работы системы с канвой рабочего стола. Если вы удалите этот код, вам может показаться, что все работает прекрасно, но только потому, что перед этим вы запускали приложение, аккуратно заполнившее экран.

Итак, интересен для нас этот несложный пример демонстрацией того, что при переключении буферов их содержимое не теряется, а перекочевывает в следующий буфер. Буферы заполняются при создании и восстановлении окна. В обработчике таймера происходит только переключение буферов.

В рассмотренном примере существуют два буфера. Это обычно для приложений, использующих DirectDraw. Если же буферов больше, то содержимым они меняются по цепочке (рис. 2.4).

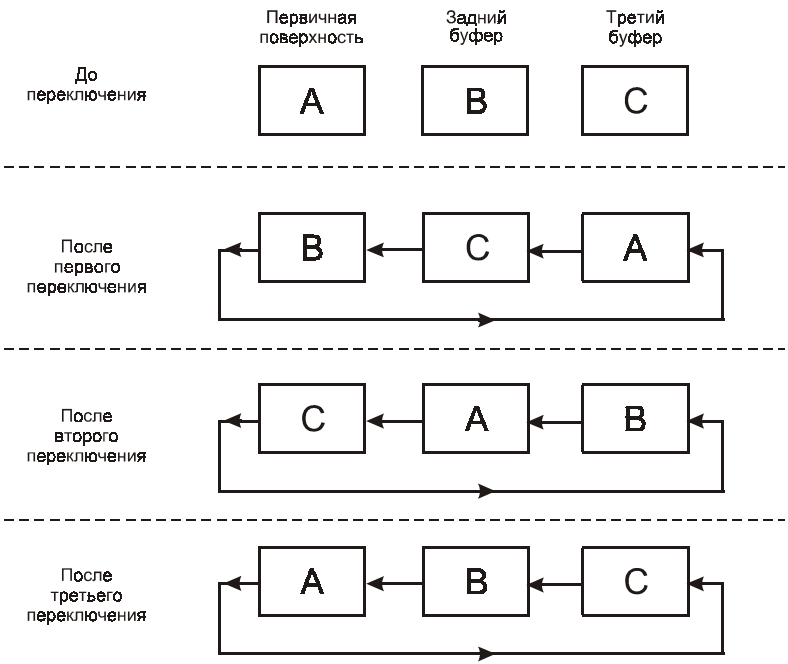


Рис. 2.4. Переключение буферов происходит по цепочке

Последний пример главы, проект каталога Ex18, на страницах книги разбирать не будем. Проект этот станет для вас полезным тогда, когда вам потребуется код получения информации о системе.

Отладка приложений

Надеюсь, у вас уже выработалась привычка запускать наши проекты, использующие DirectDraw, отдельно от среды Delphi. Полноэкранные приложения на основе DirectDraw тяжело отлаживать так, как вы привыкли это делать с обычными проектами.

Если вы установите точку останова в коде, то при достижении этой строки среда IDE попытается осуществить вывод на занятой поверхности, и ничего хорошего из этого не получится — система может зависнуть.

Ошибки, возникающие при создании и подготовке поверхностей, легко нами обрабатываются. Но как только полноэкранное приложение заняло канву рабочего стола, сообщения, выводимые нашей функцией `ErrorOut`, просто перестанут быть видны.

У каждого есть свои способы работы в такой ситуации. Я могу порекомендовать свой: пользуйтесь подачей звукового сигнала в тех точках, прохождение которых ставится под вопрос. Если есть сомнения в успешности каких-либо действий, подавайте различные сигналы, в зависимости от значения проверяемого выражения.

В ситуациях, когда такой способ существенно не поможет, придется использовать вывод в файл. Ведите протокол ваших действий, дописывая в отладочный файл информацию о выполненных операциях.

В остальных примерах использования DirectDraw расшифровка произошедшей ошибки будет выводиться в текстовый файл:

```
procedure TfrmDD.ErrorOut(hRet : HRESULT; FuncName : String);
var
  t : TextFile;
begin
  AssignFile (t, 'Debug.txt');
  Rewrite (t);
  WriteLn (t, FuncName + ': ' + DDErrorString (hRet));
  CloseFile (t);
  Destroy;
end;
```

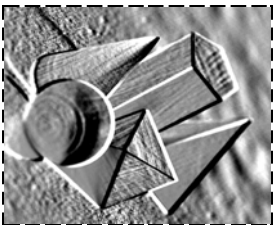
Что вы узнали в этой главе

Главное предназначение DirectDraw — вывод содержимого одной поверхности или части поверхности на другую. Поверхностей может быть столько, сколько требуется разработчику, но для построений должна быть, как минимум, одна, первичная поверхность. Первичная поверхность связана с экраном, и пользователь видит то, что в конечном итоге оказывается на ней.

Методы поверхности `Blt` и `BltFast` предназначены для осуществления блиттинга — вывода на поверхность.

В построениях, как правило, используется двойная буферизация: вывод осуществляется на поверхность заднего буфера, затем передний и задний буферы меняются местами.

ГЛАВА 3



Приемы использования DirectDraw

Данная глава является логическим продолжением предыдущей и предлагает описание основных типов приложений, использующих DirectDraw. Рассматриваемые примеры освещают основные приемы, используемые в дальнейших главах, и представляют собой более совершенные проекты.

В главе уделено много внимания вопросам доступа к содержимому поверхности, приводится масса вариантов манипуляции с этим содержимым для создания визуальных эффектов.

Примеры располагаются в каталоге \Examples\Chapter03.

Цветовой ключ

Вы должны четко определить для себя, что DirectDraw предназначен главным образом для быстрой смены растровых изображений на экране и ограничен по своим возможностям в действиях с канвой формы. Здесь нет каких-либо примитивов, команд рисования кругов, отрезков и т. п. В случае крайней необходимости можно использовать команды вывода GDI, но их желательно избегать, поскольку они слишком медленны для обычных методов DirectDraw.

Но если использовать только блиттинг прямоугольных блоков, то получается, что мы имеем дело лишь с прямоугольными вставками. Как же тогда рисуются картинки сложной формы, мы узнаем в этом разделе.

DirectDraw предоставляет на этот случай элегантный механизм, называемый *цветовым ключом* (color key). Заключается этот механизм в том, что оговариваемый цвет становится при выводе поверхности прозрачным.

Нам известны два метода для блиттинга. Посмотрим, как цветовой ключ может использоваться для метода `BltFast`, который мы стараемся использовать всегда, когда нам это позволительно.

В проекте каталога Ex01 в качестве фона используется знакомая нам по предыдущим примерам картинка. На ее фоне двигается стрелка, положение которой управляется мышью (рис. 3.1). Фактически, здесь мы заменили вид курсора приложения.

Рис. 3.1. Первый пример вывода образов непрямоугольной формы

В примере используется две вторичных поверхности: одна для вывода фона, другая — для хранения растра курсора:

```
FDDSBackGround : IDirectDrawSurface7;
FDDSImage      : IDirectDrawSurface7;
```

Для загрузки растров необходима пользовательская функция `DDLoadBitmap`:

```
// Обратите внимание, что загружаемый растр растягивается
FDDSBackGround := DDLoadBitmap(FDD, groundBmp, ScreenWidth,
    ScreenHeight);   // Загружаем фоновое изображение
if FDDSBackGround = nil then ErrorOut(DD_FALSE, 'DDLoadBitmap');
// Загружаем изображение со стрелкой
FDDSImage := DDLoadBitmap (FDD, imageBmp, 0, 0);
if FDDSImage = nil then ErrorOut (DD_FALSE, 'DDLoadBitmap');
```

После создания поверхности `FDDSImage` и загрузки в нее растра задаем цветовой ключ, используя вспомогательную функцию модуля `DDUtil`:

```
// Задаем цветовой ключ для поверхности с курсором
hRet := DDSetColorKey (FDDSImage, RGB(0, 0, 0));
if Failed (hRet) then ErrorOut(hRet, 'DDSetColorKey');
```

В качестве первого аргумента указывается имя нужной поверхности. Второй параметр — это тройка чисел, задающих цвет ключа. Все аргументы функции RGB равны нулю, поскольку в этом примере стрелка нарисована на черном фоне (рис. 3.2).

Рис. 3.2. Стрелка для курсора нарисована в квадратном растре, мы не можем использовать фигурные картинки

Цвет для ключа задается произвольным, но при рисовании картинки следует помнить, что все, закрашенное этим цветом, не будет отображаться при выводе растра. Растр в примере 24-битный, хоть и используется в нем всего два цвета: черный и синий.

При рисовании вначале с помощью метода BltFast выводим на поверхность заднего буфера фон — предварительно растянутую картинку:

```
while True do begin
  hRet := FDDSBack.BltFast (0, 0, FDDSBackGround, nil, DDBLTFAST_WAIT);
  if hRet = DDERR_SURFACELOST then begin
    if Failed (RestoreAll) then Exit;
  end
  else Break;
end;
```

Затем в позиции курсора появляется растровое изображение стрелки. Обратите внимание на новую для нас константу в комбинации флагов:

```
while True do begin
  hRet := FDDSBack.BltFast (mouseX, mouseY, FDDSImage, nil,
                            DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
  if hRet = DDERR_SURFACELOST then begin
    if Failed (RestoreAll) then Exit;
  end
  else Break;
end;
```

Добавленная константа заставляет при воспроизведении учитывать цветовой ключ источника. Данный ключ может задаваться для любой поверхности. При блиттинге можно определять, чей цветовой ключ работает — источника или приемника. Чаще всего применяется ключ источника.

Обычным делом для приложений, использующих DirectDraw, является отключение курсора или, как в рассматриваемом примере, замена его пользовательским. С системными черно-белыми курсорами проблем при воспроизведении обычно не возникает, цветные же курсоры могут мерцать или вовсе пропадать.

В описании класса формы добавлен раздел `protected`, в котором анонсирована процедура-ловушка сообщения, связанного с установкой курсора:

```
procedure FormSetCursor(var aMsg : TMessage); message WM_SETCURSOR;
```

Код процедуры совсем короткий:

```
procedure TfrmDD.FormSetCursor(var aMsg : TMessage);
begin
  SetCursor(0); // Не отображать курсор
end;
```

При перемещении курсора фиксируем его положение в глобальных переменных, следя, чтобы ни один пиксел стрелки не вышел за пределы окна:

```
procedure TfrmDD.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  if X <= ScreenWidth - 40 then mouseX := X;   // Ограничиваем размерами
  if Y <= ScreenHeight - 40 then mouseY := Y;  // растра стрелки
  FormPaint (nil);         // Вызываем код перерисовки окна
end;
```

Сам указатель, как видим, никогда не укажет на точку вблизи правой и нижней границы экрана. И есть еще одна серьезная проблема с указателем — если его передвигать быстро, то он может "застыть" далеко от границы окна. Связано это с медленной обработкой событий перемещения мыши, т. к. при быстром передвижении курсора приложение не успевает проследить все его положения. Потом мы займемся этой проблемой основательно.

Данные, размещаемые в видеопамяти, могут быть потеряны в ситуации временного ухода приложения. При его минимизации или включении энергосберегающих функций, поверхности, размещаемые в видеопамяти, должны быть восстановлены, для чего служит метод `Restore`. Содержимое их в таких ситуациях теряется и требует повторного заполнения.

Функция `DDReLoadBitmap` плохо справляется с перезагрузкой на масштабируемые поверхности, как в случае с фоном этого примера. Минимизируйте, а затем восстановите окно. Растр фона выведется с потерями, на нем появятся квадратики.

Работая с примерами предыдущей главы, вы наверняка заметили, что полноэкранные приложения, использующие DirectDraw, после своей работы оставляют в панели задач след — значок отработавшего приложения. Начиная с этого примера, для устранения такого следа в проектах полноэкранных приложений будем включать обработчик события `OnClose`, содержащий единственную строку с вызовом метода `Hide` формы.

Еще один важный момент. По завершении работы у объектов, связанных с DirectDraw, перед непосредственно высвобождением памяти будем теперь

вызывать метод _Release. Такая работа с интерфейсами является более корректной, академичной, но я обязан предупредить, что использование его в некоторых случаях может приводить к исключениям. Проблема плохо понятна, и возникает именно в приложениях, написанных на Delphi. Если вы столкнетесь с ней, то завершайте работу приложения так, как мы это делали раньше.

Обратите внимание, что в случае составной поверхности метод _Release вызывается только для первичного буфера, для заднего буфера отдельно этот метод вызывать нет необходимости:

```
procedure TfrmDD.FormDestroy(Sender: TObject);
begin
  if Assigned(FDD) then begin
    if Assigned(FDDSImage) then begin
      FDDSImage._Release;
      FDDSImage := nil;
    end;
    if Assigned(FDDSBackGround) then begin
      FDDSBackGround._Release;
      FDDSBackGround := nil;
    end;
    if Assigned(FDDSPrimary) then begin
      FDDSPrimary._Release;
      FDDSPrimary := nil;
    end;
    FDD._Release;
    FDD := nil;
  end;
end;
```

Примечание

В знак того, что наши примеры теперь становятся более совершенными, значок приложения устанавливаем отличным от принятого в Delphi по умолчанию, теперь этим значком будет логотип DirectX.

Посмотрим, как использовать цветовой ключ совместно с методом Blt поверхности, для чего переходим к проекту каталога Ex02.

По виду приложение ничем не отличается от предыдущего, изменения коснулись кода воспроизведения, в котором появилась вспомогательная переменная wrkRect типа TRECT:

```
while True do begin
  // Прямоугольник, связанный с пользовательским курсором
  SetRect (wrkRect, mouseX, mouseY, mouseX + 40, mouseY + 40);
```

```
  // Используется ключ; добавилась новая константа в комбинации флагов
  hRet := FDDSBack.Blt (@wrkRect, FDDSImage, nil,
                        DDBLT_WAIT or DDBLT_KEYSRC, nil);
  if hRet = DDERR_SURFACELOST then begin
    if Failed (RestoreAll) then Exit;
  end
  else Break;
end;
```

Как видим, для применения цветового ключа потребовалось добавить константу.

Все просто, но для этого метода есть небольшая тонкость. При масштабировании изображения DirectX интерполирует края закрашенных областей, сглаживает переходы между цветами. Так, по крайней мере, происходило у меня. Получается красиво, но при использовании цветового ключа интерполяция может немного подпортить картинку. Установите `nil` первым аргументом метода `Blt` и запустите проект. Стрелка растягивается на весь экран, а ее края красиво оттеняются темным оттенком синего. Выглядит симпатично, но, возможно, вы уже почувствовали подвох в том, что чистый синий цвет на границах стрелки потерян. Установите цветовой ключ для поверхности `FDDSImage` в чистый синий:

```
hRet := DDSetColorKey (FDDSImage, RGB(0, 0, 255));
if Failed (hRet) then ErrorOut(hRet, 'DDSetColorKey');
```

И снова запустите проект. Фон будет проглядывать только во внутренних частях стрелки, а не по всему ее силуэту.

С масштабированием связана еще одна попутно возникшая проблема. При использовании функции `DDLoadBitmap`, напоминаю, можно загружаемый растр масштабировать, задавая ненулевыми последние два аргумента. Но и при таком масштабировании края закрашенных контуров размываются, их цвет смешивается с цветом фона. При установлении ключа появляется характерный контур вокруг образов.

Выход простой — не использовать подобное масштабирование для растров, на которые предполагается накладывать ключ. В таких случаях нужно осуществлять масштабирование с помощью вспомогательных объектов класса `TBitmap`, с которыми мы уже сталкивались и сталкнемся не раз.

Полноэкранные приложения

Полноэкранные приложения являются самыми выигрышными для использования DirectDraw. Данный тип чаще всего и выбирают разработчики компьютерных игр. Главная причина, конечно, состоит в том, что полноэкранный режим позволяет обеспечивать максимальную скорость работы приложения.

Вы наверняка заметили, что профессионально написанные игры работают с удовлетворительной скоростью даже на компьютерах, оснащенных слабой видеокартой. И это при обилии графики, когда на экране мы видим десятки одновременно движущихся персонажей. Основной прием, которым достигается высокая скорость, заключается в том, что игра использует палитру из 256 цветов. Иногда кажется просто невероятным, но это действительно так. Профессиональные художники мастерски создают иллюзию богатства красок, опираясь всего лишь на 8-битную палитру. Чтобы закрепить эту иллюзию, заставки игр намеренно рисуются особенно красочными, подчас не ограничиваясь 256 цветами.

Конечно, при использовании 16-битного режима ваши приложения выиграют в эффектности, но если вы пишете масштабный проект и используете действительно много образов, то удовлетворительную скорость получите далеко не на каждом компьютере.

В проекте каталога Ex03, как и в большинстве остальных примеров книги, на основе DirectDraw используется режим в 256 цветов. Пример по функциональности очень похож на предыдущий, но вместо стрелки здесь мышью передвигается образ страшного дракона (рис. 3.3).

Рис. 3.3. Для фона используется 256-цветный рисунок

Чтобы не иметь проблем с масштабированием, размеры фонового рисунка равны 640×480 пикселов.

В проекте появилась свойственная всем приложениям, использующим 256-цветный режим, работа с палитрой. Для корректного вывода растра

нужно загрузить и установить на экране именно его палитру. Поэтому появился специальный объект:

```
FDDPal  : IDirectDrawPalette;
```

Напомню, что этому специальному объекту в начале работы приложения должно быть присвоено значение `nil`, а в конце работы перед аналогичным присвоением должен вызываться метод `_Release`.

Сразу после создания первичной поверхности устанавливаем в ней палитру, загружаемую из фонового изображения. Для загрузки набора цветов вызываем пользовательскую функцию незабвенного модуля `DDUtil`:

```
// Загружаем палитру растра
FDDPal := DDLoadPalette(FDD, groundBmp);
if FDDPal = nil then ErrorOut(DD_FALSE, 'DDLoadPalette');
```

Устанавливается палитра с помощью специального метода поверхности:

```
// Устанавливаем палитру
hRet := FDDSPrimary.SetPalette(FDDPal);
if Failed(hRet) then ErrorOut(hRet, 'SetPalette');
```

Растр намеренно выбран с подходящими размерами, чтобы не пришлось его масштабировать. Поэтому последние два аргумента `DDLoadBitmap` равны нулю:

```
FDDSBackGround := DDLoadBitmap(FDD, groundBmp, 0, 0);
if FDDSBackGround = nil then ErrorOut(DD_FALSE, 'DDLoadBitmap');
```

Дракон нарисован с черным контуром. Для цветового ключа берется цвет фона:

```
hRet := DDSetColorKey (FDDSImage, RGB(0, 255, 255));
if Failed (hRet) then ErrorOut(hRet, 'DDSetColorKey');
```

Поскольку фон не масштабируется, при восстановлении поверхностей перезагрузка фонового рисунка не приведет к зернистости. При восстановлении поверхностей следует также заново загружать и устанавливать палитру лишь при успешном восстановлении первичной поверхности:

```
function TfrmDD.RestoreAll : HRESULT;
var
  hRet : HRESULT;
begin
  hRet := FDDSPrimary._Restore;
  if Succeeded (hRet) then begin
    FDDPal := nil;                              // Удаляем старую палитру
    FDDPal := DDLoadPalette(FDD, groundBmp);  // Перезагружаем ее
    if FDDPal <> nil then begin          // Палитра перезагружена успешно
```

```
      // Заново ее устанавливаем
      hRet := FDDSPrimary.SetPalette(FDDPal);
      if Failed (hRet) then ErrorOut(hRet, 'SetPalette');
    end
    else ErrorOut(DDERR_PALETTEBUSY, 'DDLoadPalette');
    hRet := FDDSBackGround._Restore;
    if Failed (hRet) then begin
      Result := hRet;
    Exit;
  end;
  hRet := DDReLoadBitmap(FDDSBackGround, groundBmp);
  if Failed (hRet) then ErrorOut(hRet, 'DDReLoadBitmap');
    hRet := FDDSImage._Restore;
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    Result := DDReLoadBitmap(FDDSImage, imageBmp);
  end else Result := hRet;
end;
```

При неудачной перезагрузке и установлении палитры нет смысла продолжать работу приложения. Константу для вывода сообщения о фатальной ошибке я взял произвольно из ряда ошибок, связанных с неудачной работой с палитрами.

Также не имеет смысла продолжать работу приложения, если не удается попытка заново загрузить файл растра. Он ведь может быть просто удален.

Изменился немного и обработчик перемещения курсора. Теперь проблема с положением курсора вблизи границ решена:

```
procedure TfrmDD.FormMouseMove(Sender: TObject; Shift: TShiftState;
                               X, Y: Integer);
begin
  if X <= ScreenWidth - 64
    then mouseX := X
    else mouseX := ScreenWidth - 64;    // Добавилась эта ветвь
  if Y <= ScreenHeight - 64
    then mouseY := Y
    else mouseY := ScreenHeight - 64;   // Этого тоже не было
  FormPaint (nil);
end;
```

Новый пример (проект каталога Ex04) позволит нам плавно перейти к теме анимации в приложениях. Изменим предыдущий пример таким образом, чтобы изображение беспрерывно обновлялось.

Для получения максимальной скорости обновления необходим обработчик события `OnIdle` компонента класса `TApplicationEvents`. Код, записанный в этом обработчике, будет выполняться беспрерывно, пока приложение находится в режиме ожидания сообщений.

Нам нужно будет в этом обработчике записать код, связанный с перерисовкой первичной поверхности. Однако в ситуации, когда приложение не активно или минимизировано, тратить ресурсы компьютера на заведомо безуспешное и ненужное действо совершенно ни к чему. Поэтому состояние активности будем отслеживать:

```
FActive : BOOL;   // Переменная хранит информацию о текущем состоянии
```

Устанавливается эта переменная в `True` при активации окна приложения и при восстановлении его из минимизированного состояния:

```
procedure TfrmDD.ApplicationEvents1Restore(Sender: TObject);
begin
  WindowState := wsMaximized;
  // После распахивания окна считаем его готовым к воспроизведению
  FActive := True;
end;
// Появился новый обработчик
procedure TfrmDD.FormActivate(Sender: TObject);
begin
  FActive := True;   // После запуска приложения оно готово к работе
end;
```

Обработчик события `OnPaint` нам теперь не нужен, а код, связанный с перерисовкой окна, разделим для удобства на две функции, отвечающие за непосредственную перерисовку окна и за переключение страниц:

```
function TfrmDD.UpdateFrame : HRESULT;   // Функция перерисовки окна
var
  hRet : HRESULT;
begin
  // Заполняем фон
  hRet := FDDSBack.BltFast (0, 0, FDDSBackGround, nil, DDBLTFAST_WAIT);
  if hRet = DDERR_SURFACELOST then begin
    hRet := RestoreAll;
    if Failed (hRet) then begin          // Полная неудача
      Result := hRet;
      Exit;
    end;
  end;
  // Выводим изображение
  hRet := FDDSBack.BltFast (mouseX, mouseY, FDDSImage, nil,
                            DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
```

```
  if hRet = DDERR_SURFACELOST then begin
    hRet := RestoreAll;
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
  end;
  Result := DD_OK;
end;
// Функция переключения страниц
function TfrmDD.FlipPages : HRESULT;
begin
  Result := FDDSPrimary.Flip(nil, DDFLIP_WAIT);
  if Result = DDERR_SURFACELOST then Result := RestoreAll;
end;
```

Обращаю внимание, что зацикливание при каждом вызове методов блиттинга теперь не используется, при первой же неудаче покидаем функцию перерисовки окна. Код воспроизведения у нас и так выполняется беспрерывно, и на смену потерянному кадру тут же, после восстановления поверхностей, приходит новый кадр.

Но особая неприятность состоит в том, что при использовании зацикливания возникает неприятная проблема: после минимизации приложения оно перестает реагировать на сообщения и зацикливается на восстановлении поверхностей.

Последнее, и самое главное, что добавилось — это обработчик события ожидания:

```
procedure TfrmDD.ApplicationEvents1Idle(Sender: TObject;
  var Done: Boolean);
begin
  if FActive then              // Только при активном состоянии приложения
     if Succeeded (UpdateFrame)   // Перерисовка окна прошла успешно
        then FlipPages;           // Переключаем страницы
  // Посмотреть, не появились ли сообщения
  Done := False;
end;
```

Я видел немало приложений, использующих DirectDraw, написанных на Delphi. И очень многие из них не могли корректно минимизироваться или восстановиться. Хорошенько "погоняйте" этот пример, убедитесь, что в данных ситуациях он ведет себя корректно.

В примере непрерывно перерисовывается экран, положение образа на нем меняется только после передвижения курсора. Модифицируем проект, заставив двигаться картинку по кругу.

Переходим к проекту каталога Ex05. В нем удалены переменные, хранящие текущие координаты курсора, и введена вспомогательная переменная, содержащая текущее значение угла:

```
Angle : Single = 0;
```

Размер растра — 64×64 пиксела. Текущее положение на экране его центра опирается на значение переменной `Angle`:

```
FDDSBack.BltFast (320 + trunc (cos(Angle) * 150) — 32,
                  240 + trunc (sin(Angle) * 150) — 32,
                  FDDSImage, nil,
                  DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
```

Менять значение `Angle` при каждой перерисовке окна будет неправильным решением, т. к. частота перерисовки экрана сильно зависит от конфигурации компьютера, поэтому на различных компьютерах картинка начнет передвигаться с разной скоростью. И самое неприятное здесь то, что на быстродействующих компьютерах изображение будет двигаться так быстро, что пользователь не сможет разглядеть на экране вообще ничего.

Традиционное решение состоит в том, что процесс смены положений опирается на системный таймер. Экран перерисовывается так часто, как это позволяет компьютер, но положение образов меняется лишь через определенные промежутки времени.

Класс формы дополнился двумя переменными, предназначенными для контроля промежутка времени:

```
ThisTickCount  : DWORD;   // Текущее "время"
LastTickCount  : DWORD;   // Время последнего обновления
```

При активизации приложения запоминаем текущее значение системного времени. Функция `GetTickCount` возвращает количество миллисекунд, прошедших со времени запуска операционной системы:

```
LastTickCount := GetTickCount;
```

Функция перерисовки кадра начинается с того, что мы выясняем, подошло ли время смены положения образа:

```
ThisTickCount := GetTickCount;                        // Текущее "время"
if ThisTickCount — LastTickCount > 60 then begin    // Пора менять место
  Angle := Angle + 0.05;          // Для плавности смены положения образа
  // Для предотвращения переполнения
  if Angle > 2 * Pi then Angle := Angle — 2 * Pi;
  LastTickCount := GetTickCount;  // Запомнили время смены положения
end;
```

Итак, картинка сменяется с максимальной частотой, но образ передвигается только по истечении некоторого промежутка времени. Значение задержки мы задаем сами, добиваясь плавности движения.

Теперь займемся подсчетом количества воспроизводимых в секунду кадров (FPS, Frames Per Second) в проекте каталога Ex06. Здесь добавились вспомогательные переменные, связанные с подсчетом кадров:

```
Frames : Integer = 0;       // Счетчик кадров
FPS : PChar = '';           // Выводимая строка
```

При каждом воспроизведении увеличиваем счетчик, а через установленный промежуток времени подсчитываем частоту воспроизведения:

```
Inc (Frames); // Увеличиваем счетчик, воспроизводим очередной кадр
if ThisTickCount - LastTickCount > 60 then begin
  Angle := Angle + 0.05;
  if Angle > 2 * Pi then Angle := Angle - 2 * Pi;
  // Определяем и форматируем частоту
  FPS := PChar ('FPS = ' + Format('%6.2f',
                [Frames * 1000 / (ThisTickCount - LastTickCount)]));
  Frames := 0;        // Обнуляем счетчик
  LastTickCount := GetTickCount;
end;
```

Заполнив фон, выводим на экран найденную величину с помощью функции GDI `TextOut`. Не станем тратить время на особые украшения, текст выводится черным по белому:

```
if Succeeded (FDDSBack.GetDC (DC)) then begin       // DC получен
  TextOut (DC, 20, 20, FPS, 12);  // Выводим строку длиной в 12 символов
  FDDSBack.ReleaseDC (DC);        // DC обязательно должен освобождаться
end;
```

Найденная частота воспроизведения не соответствует, конечно, действительной частоте появления кадров на экране. Ведь, если эта цифра получается величиной несколько сотен, то она превышает максимальную частоту развертки монитора. Мы никак не сможем вывести на экран так много кадров за одну секунду. FPS в действительности отражает частоту обновления экранного буфера.

Конечно, чем больше эта величина, тем больше радостных чувств она вызывает у разработчика, если ваше масштабное приложение имеет FPS величиной в три десятка, это очень хорошая цифра. Большие значения свидетельствуют о том, что у проекта есть еще существенный запас для обогащения экрана или алгоритма.

Еще одна тонкость получаемой величины связана с использованием цикла ожидания. Наивысшая скорость воспроизведения нашего приложения будет

в случае, когда операционная система не слишком загружена, т. к. параллельная работа других приложений может серьезно снизить производительность нашего приложения.

Частичное обновление экрана

Частичное обновление экрана используется для повышения быстродействия, т. к. при каждой смене положения образа обновляется только участок поверхности, занимаемый им ранее.

Посмотрим на практике, как это можно осуществить. Проект каталога Ex07 является модификацией предыдущего примера, проекта с подсчетом FPS.

Получающееся теперь значение FPS может вам показаться огромным, но, с очень небольшой долю лукавства, его вполне можно считать истинным. Лукавство состоит в том, что экранный буфер обновляется частично, а не целиком.

Теперь только в начале работы и при восстановлении первичной поверхности на передний и задний буферы помещается растровое изображение, соответствующее фону:

```
if FDDSBack.BltFast (0, 0, FDDSBackGround, nil, DDBLTFAST_WAIT) =
DDERR_SURFACELOST  then Close;
if FDDSPrimary.BltFast (0, 0, FDDSBackGround, nil, DDBLTFAST_WAIT) =
DDERR_SURFACELOST  then Close;
```

Обновление кадра объединяет собственно воспроизведение и переключение страниц. Хоть функция перерисовки кадра и вызывается все также беспрерывно, но при *каждом* вызове на экране только выводится текущее значение FPS, а изменения в картинку вносятся через некоторые промежутки времени, при перемещении образа:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  DC : HDC;
  wrkRect : TRECT;
begin
  Result := DD_FALSE;
  ThisTickCount := GetTickCount;
  Inc (Frames);
  if ThisTickCount – LastTickCount > 60 then begin
    // Прямоугольник, соответствующий старому положению образа
    SetRect(wrkRect, 288 + trunc (cos(Angle) * 150),
                     208 + trunc (sin(Angle) * 150),
                     352 + trunc (cos(Angle) * 150),
                     272 + trunc (sin(Angle) * 150));
```

```
    Angle := Angle + 0.05;
    if Angle > 2 * Pi then Angle := Angle – 2 * Pi;
    // На задней поверхности выводим образ в новом месте
    if FDDSBack.BltFast (288 + trunc (cos(Angle) * 150),
                         208 + trunc (sin(Angle) * 150),
                         FDDSImage, nil,
                         DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY) =
       DDERR_SURFACELOST then if Failed (RestoreAll) then  Exit;
    FPS := PChar ('FPS = ' + Format('%6.2f',
                  [Frames * 1000 / (ThisTickCount – LastTickCount)]));
    Frames := 0;
    LastTickCount := GetTickCount;
    // Переключаем страницы, на переднем буфере образ в новом месте
    if FDDSPrimary.Flip(nil, DDFLIP_WAIT) = DDERR_SURFACELOST
       then if Failed (RestoreAll) then Exit;
    // Стираем образ на заднем буфере
    if FDDSBack.Blt (@wrkRect, FDDSBackGround, @wrkRect, DDBLT_WAIT, nil)
= DDERR_SURFACELOST then if Failed (RestoreAll) then  Exit;
  end;
  if Succeeded (FDDSPrimary.GetDC (DC)) then begin
    TextOut (DC, 20, 20, fps, 12);
    FDDSPrimary.ReleaseDC (DC);
  end;
  Result := DD_OK;
end;
```

Приводимый здесь код немного отличается от действительного, я сократил изнурительные проверки результата.

Значение FPS выводится непрерывно, при каждом обновлении кадра. Этого, в принципе, можно и не делать, а отображать его только при смене положения образа. Тогда значение FPS станет еще больше. Просто в этом случае под его значением нельзя понимать частоту обновления экранного буфера, ведь в экранную память бóльшую часть времени не будут вноситься вообще никакие изменения.

Вы можете значительно уменьшить интервал паузы между перемещениями образа, повышая тем самым частоту (частичного) обновления экрана, но получающееся значение FPS все равно будет значительным, всегда бóльшим, чем в предыдущем примере.

В данном разделе мы рассмотрели один из приемов, используемых профессиональными разработчиками игр. Иногда на медленных компьютерах, при скроллинге экрана или быстром перемещении, хорошо заметно "торможение" воспроизведения, вызванное тем, что при этом перерисовывается весь экран, а не его часть. Для ослабления такого эффекта дизайнеры часто уменьшают игровой экран, располагая по границе его различные панели и меню.

Непосредственный доступ к пикселам поверхности

Прямой доступ к графическим данным обеспечивает максимум быстродействия, и предоставляет разработчику возможность реализации любых, или почти любых, действий с изображением.

На время прямого доступа поверхность должна быть заблокирована, после работы поверхность необходимо разблокировать. Во время блокировки поверхности операционная система находится в особом режиме, поэтому блокировка должна применяться в течение максимально короткого промежутка времени.

Блокирование поверхности является одним из самых спорных моментов в DirectDraw. Фактически она означает исключительный доступ к разделу памяти, связанному с поверхностью. Если для работы с обычными переменными, например, при копировании одной строки в другую, нам не приходится блокировать память, ассоциированную с данными, то почему же при прямом доступе к памяти поверхности нам непременно следует блокировать эту память? Запирать поверхность необходимо, поскольку позиция поверхности в системной памяти может меняться, системный менеджер памяти по каким-то своим соображениям может перемещать блоки памяти.

Примечание

Работа с данными, размещаемыми в видеопамяти, в принципе отличается от привычной. Позже мы узнаем, как избавиться от блокирования поверхности, размещенной в системной памяти, если это необходимо.

Перейдем к проекту каталога Ex08. Смысл примера таков: не будем использовать растровое изображение в качестве фона, а для заполнения его, получив адрес поверхности заднего буфера в памяти, заполним нулем блок памяти этого буфера.

Память поверхности всегда организована линейно, поэтому обращение к данным сильно упрощается.

В коде удалены все фрагменты, связанные в предыдущем примере с фоном, включая палитру. Добавилась функция быстрой очистки заднего буфера:

```
function TfrmDD.Clear : HRESULT;
var
  desc : TDDSURFACEDESC2;            // Вспомогательная структура
  hRet : HRESULT;
begin
  Result := DD_FALSE;
  ZeroMemory (@desc, SizeOf(desc));  // Обычные действия с записью
  desc.dwSize := SizeOf(desc);
```

```
  // Запираем задний буфер
  hRet := FDDSBack.Lock (nil, desc, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Заполняем нулем блок памяти заднего буфера
  FillChar (desc.lpSurface^, 307200, 0);
  // В конце работы обязательно необходимо открыть запертую поверхность
  Result := FDDSBack.Unlock (nil);
end;
```

Действие метода `Lock` очень похоже на действие знакомого нам метода `GetSurfaceDesc`, в полях указанной структуры типа `TDDSURFACEDESC2` хранится информация о поверхности, в частности поле `lpSurface` содержит ее адрес.

Единственное действие, производимое нами в этой функции с блокированной поверхностью, состоит в том, что мы заполняем нулем весь блок памяти заднего буфера. Используется 8-битный режим, значение 307 200 — размер блока памяти, ассоциированного с поверхностью — получилось путем перемножения 640 на 480 и на 1 (размер единицы хранения, байт).

Первый параметр метода `Lock` — указатель на величину типа `TRECT`, задающую запираемый регион, если блокируется только часть поверхности.

Второй параметр ясен. Это структура, хранящая данные для вывода на поверхность.

Третий — флаг или комбинация флагов. Применяемый здесь традиционен для нас и указывает, что необходимо дожидаться готовности устройства.

Последний аргумент не используется.

Парный метод поверхности `UnLock` имеет один аргумент, аналогичный первому аргументу метода `Lock` и указываемый в том случае, если запирается не вся поверхность целиком.

Обратите внимание, как важно анализировать возвращаемое значение. Если этого не делать для метода `Lock`, то при щелчке по кнопке минимизированного окна фон "не восстановится", и первичная поверхность окажется потерянной безвозвратно.

Итак, мы изучили быстрый способ заполнения фона черным цветом. Для 8-битного режима можете использовать любое число в пределах до 255. Но заранее предсказать, каким цветом будет заполняться фон, мы не можем, за исключением первого и последнего чисел диапазона. Тонкости палитры мы осветим позднее. Для прочих разрешений имеются свои особенности, о которых мы поговорим также чуть позже. А пока будем опираться на режим в 256 цветов, а фон использовать черный.

Посмотрим проект каталога Ex09, в котором экран с течением времени заполняется точками случайного цвета и случайными координатами. Ключевой является функция, перекрашивающая конкретную точку на экране в указанный цвет:

```
function TfrmDD.PutPixel (const X, Y : Integer;
                          const Value : Byte) : HRESULT;
var
  desc : TDDSURFACEDESC2;
  hRet : HRESULT;
begin
  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf(desc);
  // Всегда, всегда анализируйте результат
  hRet := FDDSBack.Lock (nil, desc, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Находим адрес нужного пиксела и устанавливаем его значение
  PByte (Integer(desc.lpSurface) + Y * desc.lPitch + X)^ := Value;
  Result := FDDSBack.Unlock (nil);
end;
```

Поле `lPitch` записи `TDDSURFACEDESC2` содержит расстояние до начала следующей строки. Для 8-битного режима это будет, конечно, 640 (ширина поверхности умножить на размер одной ячейки). Но мы подготавливаем универсальный код, для других режимов есть существенное отличие.

Код перерисовки кадра совсем прост, ставим очередную точку:

```
Result := PutPixel (random (ScreenWidth),
                    random (ScreenHeight), random (255));
```

Для того чтобы нарисованные точки не пропадали, экран очищать необходимо только один раз. У нас это делается сразу после подготовки поверхностей. Обратите внимание, как все происходит:

```
if Failed (Clear) then Close;       // Очищаем задний буфер
if Failed (FlipPages) then Close;   // Переставляем буферы
// Очищаем то, что раньше находилось в переднем буфере
if Failed (Clear) then Close;
```

Нельзя забывать и о ситуации восстановления окна, после восстановления поверхностей опять следует очистить оба буфера:

```
function TfrmDD.RestoreAll : HRESULT;
var
  hRet : HRESULT;
```

```
begin
  hRet := FDDSPrimary._Restore;
  if Succeeded (hRet) then begin      // Только при успехе этого действия
    if Failed (Clear) then Close;
    if Failed (FlipPages) then Close; // Здесь неудача уже непоправима
    if Failed (Clear) then Close;
    Result := DD_OK
  end else Result := hRet;
end;
```

Чтобы избежать рекурсии, процедура восстановления поверхностей вызывается не в функции переключения поверхностей, а в цикле ожидания:

```
procedure TfrmDD.ApplicationEvents1Idle(Sender: TObject;
                                        var Done: Boolean);
begin
  if FActive then begin
    if Succeeded (UpdateFrame)
      then FlipPages
    else RestoreAll
  end;
  Done := False;
end;
```

Ну что же, если мы в состоянии поставить отдельную точку на экране, то можем нарисовать, в принципе, любой примитив. Иллюстрацией такого утверждения служит проект каталога Ex10, где экран с течением времени "усеивается" окружностями (рис. 3.4).

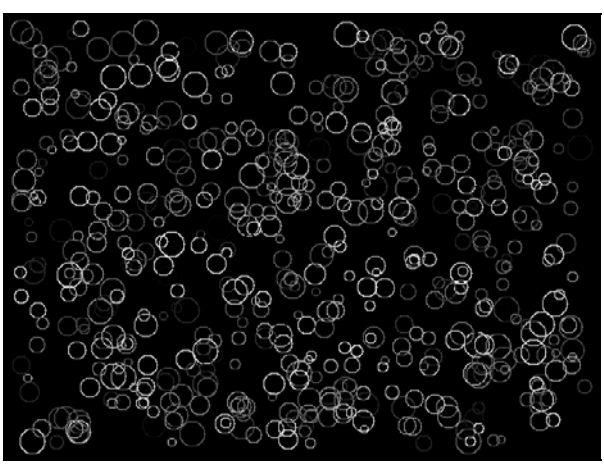


Рис. 3.4. В DirectDraw нет готовых примитивов, и эти окружности строятся по отдельным точкам

Здесь я не пользуюсь процедурой предыдущего примера, перекрашивающей один пиксел экрана, чтобы не запирать поверхность для каждой точки окружности. Введена функция, блокирующая поверхность на все время рисования окружности:

```
function TfrmDD.Circle (const X, Y, R : Integer;
                        const Color : Byte) : HRESULT;
// Локальная процедура для одной точки
// Поверхность должна быть предварительно заперта
procedure PutPixel (const Surf, lPitch, X, Y : Integer;
                    const Value : Byte);
begin
  PByte (Surf + Y * lPitch + X)^ := Value;
end;
var
  desc : TDDSURFACEDESC2;
  a : 0..359;              // Угол
  hRet : HRESULT;
begin
  Result := DD_FALSE;
  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf(desc);
  hRet := FDDSBack.Lock (nil, desc, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  for a := 0 to 359 do     // Берем значения углов полного круга
    PutPixel (Integer(desc.lpSurface), desc.lPitch,
              X + trunc (cos (a) * R), Y + trunc (sin (a) * R),
              Color);
  Result := FDDSBack.Unlock (nil);
end;
```

При перерисовке кадра диапазоны для параметров окружностей строго ограничиваются пределами экрана, чтобы ненароком не "залезть" в чужую область памяти:

```
Result := Circle (random (ScreenWidth — 30) + 15, random
         (ScreenHeight — 30) + 15, random (10) + 5, random (256));
```

Вы должны обратить внимание, как неприятно мерцает экран в данном и предыдущем примерах. Каждый новый примитив рисуется на поверхности заднего буфера, затем буферы меняются местами. Подвох очевиден: примитив мерцает, потому что он нарисован только на одной из двух поверхностей.

Согласование содержимого буферов

При каждом изменении фона экрана необходимо согласовывать содержимое обоих буферов. Запустите проект каталога Ex11 — модификацию предыдущего примера, но уже без неприятного мерцания экрана. Порядок воспроизведения в подобных ситуациях обсудим подробнее при рассмотрении следующего примера.

Отвлечемся немного от прямого доступа к памяти. Закрепим недавно пройденное. Мы ведь знаем и другой способ закраски, которым пользовались в самых первых примерах для заполнения фона.

Смотрим проект каталога Ex12, экран все также заполняется окружностями, но при разрешении экрана, поддерживающем 16-битный режим, и без операций непосредственного доступа к памяти поверхности.

Процедура очистки экрана основана на использовании метода `Blt`:

```
function TfrmDD.Clear : HRESULT;
var
  ddbltfx : TDDBLTFX;
begin
  ZeroMemory(@ddbltfx, SizeOf(ddbltfx));
  ddbltfx.dwSize := SizeOf(ddbltfx);
  ddbltfx.dwFillColor := 0;
  Result := FDDSBack.Blt(nil, nil, nil,
                         DDBLT_COLORFILL or DDBLT_WAIT, @ddbltfx);
end;
end;
```

Напрягите свою память — мы проходили уже такой способ.

Чтобы перекрасить один пиксел, воспользуемся все тем же приемом с применением метода `Blt`, но ограничим область перекрашивания небольшим квадратом:

```
function TfrmDD.Circle (const X, Y, R : Integer;
                        const Color : Byte) : HRESULT;
function DDPutPixel (const X, Y, R, G, B : Integer) : HRESULT;
var
  ddbfx : TDDBLTFX;
  rcDest : TRECT;
begin
  ZeroMemory (@ddbfx, SizeOf(ddbfx));
  ddbfx.dwSize := SizeOf(ddbfx);
  ddbfx.dwFillColor := RGB(R, G, B);
  // Перекрашиваться будет маленький квадрат
  SetRect(rcDest, X, Y, X + 1, Y + 1);
```

```
  Result := FDDSBack.Blt(@rcDest, nil, nil,
            DDBLT_WAIT or DDBLT_COLORFILL, @ddbfx);
end;
var
  a : 0..359;
  hRet : HRESULT;
begin
  for a := 0 to 359 do begin
    hRet := DDPutPixel(X + trunc (cos (a) * R), Y + trunc (sin (a) * R),
                       Color, Color, Color);
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
  end;
end;
```

Цвет задается тройкой одинаковых чисел. Для повышения красочности вы можете попробовать генерировать отдельное значение для каждой составляющей цвета. И если вы хорошенько поработаете с этим примером, то обнаружите небольшой обман: функция `RGB` в примере не работает должным образом, цвета получаются отнюдь не ожидаемые. Режим здесь 16-битный. Позднее, когда мы познакомимся с форматом пикселов, то найдем хорошее решение для этой проблемы.

Переключение буферов в данном примере из обработчика `OnIdle` перенесено непосредственно в код обновления кадра.

При воспроизведении, аналогично предыдущему примеру, рисуем окружность в заднем буфере, затем буферы переключаем, и повторяем рисование окружности на том же самом месте, но уже во втором буфере:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  X, Y, R : Integer;
  Color : Byte;
  hRet : HRESULT;
begin
  X := random (ScreenWidth - 30) + 15;
  Y := random (ScreenHeight - 30) + 15;
  R := random (10) + 5;
  Color := random (256);
  // Рисуем окружность в заднем буфере первый раз
  hRet := Circle (X, Y, R, Color);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
```

```
  if FDDSPrimary.Flip(nil, DDFLIP_WAIT) = DDERR_SURFACELOST
    then begin
      hRet := RestoreAll;
      if Failed (hRet) then begin
        Result := hRet;
        Exit;
      end;
    end;
  // Рисуем ту же окружность в заднем буфере второй раз
  Result := Circle (X, Y, R, Color);
end;
```

Поворот изображения

Такая эффектная операция, как я уже говорил, аппаратно поддерживается далеко не каждой видеокартой. Посмотрим, как можно использовать пикселные операции для осуществления поворота изображения (проект каталога Ex13). На экране вращается жуткое изображение (рис. 3.5).

Рис. 3.5. Очень страшный пример поворота растра

Не пугайтесь, хоть картинка и страшная, сам пример совершенно безобиден, если только вы не будете лицезреть его работу чересчур долго.

Используется картинка размером 256×256 пикселов, для работы с которыми введен пользовательский тип:

```
type
  TByteArray = Array [0..255, 0..255] of Byte;
```

Переменная `Pict` данного типа хранит растровое изображение, а массив заполняется в пользовательской процедуре, вызываемой в начале работы приложения и при каждом восстановлении поверхностей:

```
function TfrmDD.Prepare : HRESULT;
var
  desc : TDDSURFACEDESC2;
  i, j : Integer;
  hRet : HRESULT;
```

```
begin
  hRet := Clear;          // Очистка первичной поверхности
  if Failed (hRet) then begin
     Result := hRet;
     Exit;
  end;
  // Посередине экрана выводится картинка с черепом
  hRet := FDDSPrimary.BltFast (193, 113, FDDSImage, nil,
                               DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;

  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf(desc);
  // Запираем поверхность
  hRet := FDDSPrimary.Lock (nil, desc, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Считываем в массив Pict содержимое нужных пикселов экрана
  for i := 0 to 255 do
    for j := 0 to 255 do
      Pict [i, j] := PBYTE (Integer (desc.lpSurface) +
                            (j + 113) * desc.lPitch + (i + 193))^;
  Result := FDDSPrimary.Unlock (nil);
end;
```

Заполнить массив можно многими разными способами, например напрямую из растра. Также обращаю внимание, что массив можно заполнять и из содержимого поверхности `FDDSImage`, без промежуточного блиттинга на первичную. Если ключом является не черный цвет, следует анализировать цвет каждого пиксела и отбрасывать пиксел с цветом ключа, а при использовании черного цвета в качестве ключа можно просто копировать значения пикселов в массив. Так мы будем поступать в последующих примерах.

Переменная `Angle` хранит текущее значение угла поворота растрового изображения в радианах. Изменяется ее значение при обновлении окна через некоторый промежуток времени:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  hRet : HRESULT;
begin
  Result := DD_FALSE;
```

```
  ThisTickCount := GetTickCount;
  if ThisTickCount — LastTickCount > 30 then begin
    Angle := Angle + 0.1;                  // Угол в радианах
    // Надо уберечься от переполнения
    if Angle > 2 * Pi then Angle := Angle — 2 * Pi;
    while True do begin
      if Failed (Rotating) then begin      // Поворот на Angle
        hRet := RestoreAll;
        if Failed (hRet) then begin        // Неустранимая ошибка
          Result := hRet;
          Exit;
        end
      end else Break
    end;
    LastTickCount := GetTickCount;
  end;
  Result := DD_OK;
end;
```

Пользовательская функция `Rotating`, несмотря на свое название, не содержит кода самого поворота картинки, а лишь заменяет содержимое части экрана:

```
function TfrmDD.Rotating : HRESULT;
var
  desc : TDDSURFACEDESC2;
  i, j : Byte;
  Image : TByteArray;
  hRet : HRESULT;
begin
  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf(desc);
  // Получаем растр из первоначального путем
  // поворота на угол Alpha относительно середины растра
  Image := Rotate (Pict, 127, 127, Angle);
  hRet := FDDSPrimary.Lock (nil, desc, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Заполняем блок экрана новым растром
  for i := 0 to 255 do
    for j := 0 to 255 do
      PByte (Integer (desc.lpSurface) + (j + 113) * desc.lPitch +
             i + 193)^ := Image [i, j];
  Result := FDDSPrimary.Unlock (nil);
end;
```

Самая интересная функция примера — пользовательская функция, возвращающая растр, повернутый на заданный угол относительно указанной точки:

```
function TfrmDD.Rotate (const pictOriginal : TByteArray; // Исходный растр
                        // Точка в растре, задающая оси поворота
                         const iRotationAxis, jRotationAxis: Integer;
                         const ug : Single): TByteArray;  // Угол, радианы
type                    // Тип, соответствующий одной строке массива
  wrkByteArray = Array [0..255] of Byte;
var
  i, j        : Integer;
  iOriginal   : Integer;
  iPrime      : Integer;
  jOriginal   : Integer;
  jPrime      : Integer;
  RowOriginal : ^wrkByteArray;
  RowRotated  : ^wrkByteArray;
  sinTheta    : Single;
  cosTheta    : Single;
begin
  sinTheta := sin(ug);          // Для оптимизации синусы и косинусы
  cosTheta := cos(ug);          // Запоминаем в рабочих переменных
  for j := 255 downto 0 do begin     // Строки результирующего массива
    RowRotated := @result [j, 0];    // Указатель на очередную строку
    jPrime := j - jRotationAxis;     // Смещение от оси по Y
    for i := 255 downto 0 do begin   // Цикл по столбцам
      iPrime := i - iRotationAxis;   // Смещение от оси по X
      iOriginal := iRotationAxis + trunc(iPrime * cosTheta -
                   jPrime * sinTheta); // Координаты нужной точки по X
      jOriginal := jRotationAxis + trunc(iPrime * sinTheta +
                   jPrime * cosTheta); // Координаты нужной точки по Y
      // После поворота некоторые точки на границе
      // не имеют аналога в старом растре
      if (iOriginal >= 0) and (iOriginal <= 255) and    // Не границы
         (jOriginal >= 0) and (jOriginal <= 255) then begin
            // Копируем в новый растр точку
            RowOriginal := @pictOriginal[jOriginal, 0];
            RowRotated^[i] := RowOriginal^[iOriginal]
      end
      else RowRotated[i] := 0;   // Границы заполняем черным цветом
    end
  end;
end;
```

В этом и следующем примерах я не применяю двойную буферизацию. Если же с использованием вашей видеокарты по этой причине шоу разворачива-

ется слишком медленно, в качестве упражнения установите двойную буферизацию.

Визуальные эффекты

В данном разделе мы закрепим наши навыки непосредственного доступа к пикселам и научимся создавать некоторые несложные эффекты.

В проекте каталога Ex14 выводится тот же образ, что и в предыдущем примере, но уже весь покрытый "перцем", подобно изображению плохо настроенного телевизора (рис. 3.6).

Рис. 3.6. Эффект "перца"

Добиться эффекта очень легко — достаточно для вывода выбирать произвольные точки из массива образа, а остальные точки оставлять черными:

```
function TfrmDD.Effect : HRESULT;
var
  desc : TDDSURFACEDESC2;
  i, j : Byte;
  Image : TByteArray;   // Вспомогательный массив,
                        // размеры равны размеру растра
  k : Integer;
  hRet : HRESULT;
begin
  Result := DD_FALSE;
  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf(desc);
  // Локальные массивы надо всегда инициализировать
  ZeroMemory (@Image, SizeOf (Image));
  for k := 0 to 100000 do begin  // Верхний предел задает густоту перца
    i := random (255);           // Можно брать и меньший интервал
    j := random (255);           // Растр занимает не всю область 256x256
    Image [i, j] := Pict [i, j]; // Берем точку растра
  end;
```

```
  hRet := FDDSPrimary.Lock (nil, desc, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
   for i := 0 to 255 do
     for j := 0 to 255 do
       PByte (Integer (desc.lpSurface) + (j + 113) * desc.lPitch +
                       i + 193)^ := Image [i, j];
  Result := FDDSPrimary.Unlock (nil);
end;
```

Надеюсь, все просто и понятно, и в качестве упражнения модифицируйте пример таким образом, чтобы густота перца менялась с течением времени.

Двигаемся дальше. Рассмотрим проект каталога Ex15 — простой пример на смешивание цветов. Посередине экрана выводится картинка размером 64×64 пикселов, при обновлении кадра вызывается пользовательская процедура, усредняющая цвет для каждого пиксела внутри области растра. Для усреднения берется девять соседних точек:

```
function TfrmDD.Blend : HRESULT;
var
  desc : TDDSURFACEDESC2;
  i, j : Byte;
  Pict : Array [0..63, 0..63] of Byte;
  hRet : HRESULT;
begin
  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf(desc);
  hRet := FDDSBack.Lock (nil, desc, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Во вспомогательный массив заносится область растра
  for i := 0 to 63 do
    for j := 0 to 63 do
      Pict [i, j] := PBYTE (Integer (desc.lpSurface) +
                            (j + 208) * desc.lPitch + (i + 288))^;
// Для каждой точки внутри области растра значение пиксела берется
// усредненным значением девяти окружающих точек
  for i := 1 to 62 do
    for j := 1 to 62 do
      PByte (Integer (desc.lpSurface) +
            (j + 208) * desc.lPitch + i + 288)^ :=
              (Pict [i — 1, j — 1] +
```

```
                Pict [i, j - 1] +
                Pict [i + 1, j - 1] +
                Pict [i - 1, j] +
                Pict [i, j] +
                Pict [i + 1, j - 1] +
                Pict [i - 1, j + 1] +
                Pict [i, j + 1] +
                Pict [i + 1, j + 1])  div 9;
  Result := FDDSBack.Unlock (nil);
end;
```

Прием простой и очень действенный. Его эффектность поможет нам оценить готовый проект из каталога Ex16, во время работы которого на экране появляется феерическая картина (рис. 3.7).

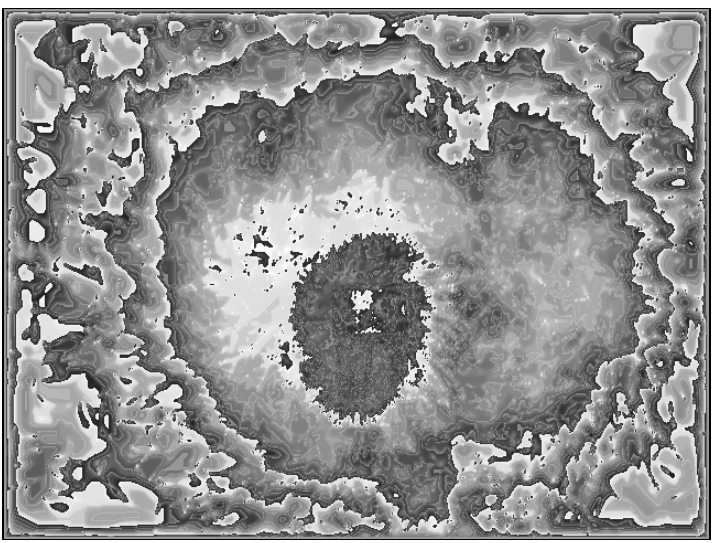


Рис. 3.7. Простым смешиванием цветов можно добиться очень сильных визуальных результатов

Алгоритм работы прост: по экрану двигаются частицы, за каждой из которых тянется след. Срок жизни любой частицы ограничен, новые точки появляются в месте расположения курсора:

```
const
  MaxParticles = 100000; // Верхнее ограничение по количеству точек
type
  TParticle = record     // Тип для описания отдельной точки
    X : Integer;         // Координаты точки на экране
    Y : Integer;
```

```
    Angle : Single;      // Угол направления движения
    Speed : Integer;     // Скорость движения
    Decay : Single;      // Время жизни
    HalfLife : Single;   // Срок существования
    // Величина сдвига для угла, движение по спирали
    AngleAdjustment : Single;
  end;
var    // Глобальные переменные модуля
  ParticleCount : Integer = 10000;    // Текущее количество точек
  Particle : Array [0..MaxParticles] of TParticle; // Массив частиц
  mouseX, mouseY : Integer;           // Координаты курсора
  // Растровый массив, хранит цвет для всех пикселов экрана
  Pict : Array [0..ScreenWidth — 1, 0..ScreenHeight — 1] of Byte;
  BlurFactor : Integer = 1;           // Задает величину размытости следа
```

При начале работы приложения массив частиц заполняется первоначальными данными, и частицы располагаются хаотически по всему экрану:

```
for Index := 0 to MaxParticles do
  with Particle [Index] do begin
    Speed := 1 + round (random (3));
    Angle := random * 2 * Pi;
    X := random (ScreenWidth — 1) + 1;
    Y := random (ScreenHeight — 1) + 1;
    Decay := random;
    HalfLife := random / 20;
    AngleAdjustment := random / 20;
end;
```

При каждом обновлении экрана отслеживаются новые позиции частиц и усредняются цвета пикселов, подобно предыдущему примеру:

```
for Index := 0 to ParticleCount do
  with Particle [Index] do begin
    Decay := Decay — HalfLife;            // Уменьшить время жизни
    // Срок существования прошел, появляется новая точка
    if Decay <= 0 then begin
      Decay := 1;
      X := mouseX;     // В позиции курсора
      Y := mouseY;
    end;
    Angle := Angle + AngleAdjustment;     // Движение по спирали
    If Angle >= 2 * Pi then Angle := 0;   // От переполнения
    X := X + round (cos(Angle) * Speed);  // Новая позиция
    Y := Y + round (sin(Angle) * Speed);
```

```
    // Точка, ушедшая за границу экрана
    if (X > ScreenWidth — 2) or (X < 2) then begin
      X := mouseX;     // Переместить в позицию курсора
      Y := mouseY;
      Angle := random * 2 * Pi;
    end
    else if (Y > ScreenHeight — 2) or (Y < 2) then begin
      X := mouseX;
      Y := mouseY;
      Angle := random * 2 * Pi;
    end;
    // "Отображение" точки
    Pict [X, Y] := Speed * 16 + 186;
  end;
  // Эффект размытости
  for Index := 1 to BlurFactor do
    for X := 2 to ScreenWidth — 2 do
      for Y := 2 to (ScreenHeight — 2) do begin
      // Усреднение значения девяти соседних элементов
        Accum := 0;
        Accum := Accum + Pict [X, Y] + Pict[X, Y + 1] +
                 Pict[X, Y — 1] + Pict[X + 1, Y] +
                 Pict[X — 1, Y] + Pict[X + 1, Y + 1] +
                 Pict[X — 1, Y — 1] + Pict[X + 1, Y — 1] +
                 Pict[X — 1, Y + 1];
        Accum := Accum div 9;  // Усреднение значений
                               // соседних пикселов
        Pict [X, Y] := Accum;
  end;
```

Чтобы изображение не съеживалось с течением времени, как в предыдущем примере, закрашиваясь черным цветом, граничные точки экрана заполняются ненулевыми значениями:

```
for Index := 0 to ScreenWidth — 1 do begin
  Pict[Index, 0] := 127;
  Pict[Index, ScreenHeight — 1] := 127;
  Pict[Index, 1] := 127;
  Pict[Index, ScreenHeight — 2] := 127;
end;
for Index := 0 to ScreenHeight — 1 do begin
  Pict[0, Index] := 127;
  Pict[ScreenWidth — 1, Index] := 127;
  Pict[1, Index] := 127;
  Pict[ScreenWidth — 2, Index] := 127;
end;
```

С помощью клавиш <Home> и <End> можно менять количество частиц, а с помощью клавиш <Page Up> и <Page Down> — управлять степенью усреднения пикселов.

Пример может работать при разных разрешениях и глубине цвета экрана. Обратите внимание, что при его очистке размер блока в таких случаях задается исходя из значения текущей глубины:

```
ZeroMemory (desc.lpSurface, desc.lPitch * ScreenHeight *
           (ScreenBitDepth div 8));
```

Также здесь нельзя использовать значение ширины экрана вместо `lPitch`, т. к. из-за выравнивания памяти это могут быть разные значения. Ширина поверхности "подгоняется" к границам параграфов[1], т. е. должна быть кратна 4-м байтам.

Массивы в видеопамять приходится переносить медленным способом — поэлементно. Одна ячейка массива занимает байт, при разрешении экрана в 16 разрядов на пиксел массив скопируется только в первую половину памяти поверхности. Если же вы в своем приложении не собираетесь менять разрешение, то вполне можете копировать массив целиком, одной командой `CopyMemory`.

Поскольку значения в массиве `Pict` лежат в пределах диапазона типа `Byte`, то для 16-битного режима картинка получится не очень выразительной и отображается оттенками одного цвета.

Сохранение растровых изображений

Наверняка перед вами рано или поздно встанет задача сохранения получающихся картинок. Если вы попытаетесь их скопировать в буфер обмена для дальнейшей вставки в рисунок графического редактора, то обнаружите проблему с 256-цветными приложениями. Картинки будут искажаться, поскольку палитра таких изображений будет отличной от палитры рисунка.

Я приведу простейшее решение проблемы, основанное на использовании объекта класса `TBitmap`. В предыдущем примере обработчик формы нажатия клавиши приведите к следующему виду:

```
procedure TfrmDD.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
var
  BitMap : TBitmap;     // Для записи картинок в файл
begin
  case Key of
    VK_NEXT : BlurFactor := BlurFactor + 1;
```

[1] Параграф — единица распределяемой памяти, равная 4 байтам. — *Ред.*

```
    VK_PRIOR : begin
      BlurFactor := BlurFactor - 1;
      if BlurFactor < 1 then BlurFactor := 1;
    end;
    VK_HOME : begin
      Inc (ParticleCount, 1000);
      if ParticleCount > MaxParticles then ParticleCount := MaxParticles;
    end;
    VK_END  : begin
      Dec (ParticleCount, 1000);
      if ParticleCount < 2000 then ParticleCount := 2000;
    end;
    // По нажатию пробела содержимое экрана сохраняется в файле
    VK_SPACE : begin
      BitMap := TBitmap.Create;
      BitMap.PixelFormat := pf24bit;        // Разрядность задаем 24
      BitMap.Height := ClientHeight;
      BitMap.Width := ClientWidth;
      // Копируем в BitMap содержимое экрана
      BitBlt(BitMap.Canvas.Handle, 0, 0, ClientWidth, ClientHeight,
             Canvas.Handle, 0, 0, SRCCOPY);
             BitMap.SaveToFile ('1.bmp');   // Записываем в файл
    end;
    VK_ESCAPE, VK_F12 : Close;
  end;
end;
```

Записываются 24-битные файлы, и информация о цвете не теряется в любом случае.

Доступ к пикселам в 16-битном режиме

В таком режиме информация о цвете пиксела разделяется на три цветовые составляющие, но шестнадцать на три нацело не делится, поэтому разработчики вынуждены прибегать к неравномерному распределению. Наиболее распространенной является схема 5–6–5. В этом формате первые пять битов хранят значение красного оттенка, следующие шесть битов отводятся под зеленую составляющую, ну и последние пять битов заняты оттенком синего. Всего получается 65 536 (2^{16}) различных цветов. Из них по 32 градации красного и синего, 64 градации зеленого.

Схема 5–6–5 является самой распространенной. Поэтому для начала будем опираться именно на нее. Как быть в случае другого формата, рассмотрим позднее.

Для примера возьмем цвет, образованный следующими значениями составляющих:

- красный, 5 бит: 00011;
- зеленый, 6 бит: 001011;
- синий, 5 бит: 00101.

Значение пиксела с таким цветом будет следующим (пробелы вставлены для удобочитаемости):

```
0001 1001 0110 0101
```

Все выглядит просто, имея значение трех составляющих, мы должны в пиксел заносить значение по следующей формуле:

`blue + green * 2^5 + red * 2^11` или `blue + green * 64 + red * 4096`

Операции умножения и деления с участием степени двойки лучше оптимизировать с помощью операции сдвига. Теперь окончательная формула выглядит так:

```
blue OR (green SHL 5) OR (red SHL 11)
```

Иллюстрация в виде примера последует позже, а сейчас задержимся на том, как вырезать из пиксела значения составляющих. Для этого применяются битовые маски. Так, для получения значения пяти битов красной составляющей надо использовать бинарное число

```
1111 1000 0000 0000
```

и логическую операцию `AND` для вырезания значения первых пяти битов. Вот так:

```
0001 1001 0110 0101 &
1111 1000 0000 0000
-------------------
0001 1000 0000 0000
```

Результат найден, как видим, верно, но ему предшествуют одиннадцать нулей. Чтобы получить значение составляющей, надо применить к этому выражению операцию битового сдвига вправо. Вот пример для красной составляющей:

```
Red : Byte;
Red := (pixel & $F800) SHR 11;
```

Или, если поменять порядок действий, вырезать ее можно так:

```
Red := (pixel SHR 11) AND $1f;
```

Маска в этом случае та же — пять единиц, но без завершающих одиннадцати нулей.

Перейдем к иллюстрации — проекту каталога Ex17. Работа его выглядит очень просто, на экране появляются вспышки синих и красных частиц. Работа с системой частиц во многом похожа на код предыдущего примера, но теперь воспользуемся концепцией ООП:

```
const
  MAX_ENERGY = 60;              // Максимальная энергия частицы
  DEFAULT_SIZE = 200;           // Количество частиц во вспышке
  DEFAULT_POWER = 30;           // Для зарядки энергии частицы
type
  TParticle = record          // Данные на отдельную частицу
    X, Y : Single;              // Позиция
    SpeedX, SpeedY : Single;    // Скорости по осям
    Energy : Integer;           // Энергия
    Angle : Integer;            // Направление движения
    R, G, B : Byte;             // Цвет
  end;
  TParticleSystem = class       // Класс системы частиц
    public
    procedure Init (NewSize, Power : Integer); // Инициализация
    procedure Calculate;            // Пересчет положений частиц
    function Render : HRESULT;      // Отображение вспышки
    private
      Particle : Array [0..1000] of TParticle; // Массив частиц
      Size : integer;               // Размер
  end;
```

Инициализация системы выглядит так:

```
procedure TParticleSystem.Init (NewSize, Power : Integer);
var
  i : Integer;
  X, Y : Integer;     // Стартовая точка вспышки
  Speed : Single;
begin
  Size := NewSize;    // Устанавливаем размер системы
  // Центр вспышки располагаем вдали от границ экрана
  X := random (ScreenWidth - 80) + 40;
  Y := random (ScreenHeight - 80) + 40;
  for i := 0 to Size do begin         // Частицы системы
    Particle[i].X := X;
    Particle[i].Y := Y;
    Particle[i].Energy := random (MAX_ENERGY); // Энергия
    Particle[i].Angle  := random (360);    // Угол движения
    Speed := random (Power) - Power / 2;
    Particle[i].SpeedX := sinA[Particle[i].Angle] * Speed;
```

```
    Particle[i].SpeedY := cosA[Particle[i].Angle] * Speed;
    Particle[i].r := random (256);     // Сине-красный цвет
    Particle[i].g := 0;
    Particle[i].b := random (256);
  end;
end;
```

Первый раз система инициализируется в начале работы приложения. Здесь же заполняются вспомогательные массивы, хранящие синусы и косинусы углов:

```
sinA : Array [0..360] of Single;
cosA : Array [0..360] of Single;
PS : TParticleSystem;
for j := 0 to 360 do begin           // Для оптимизации, чтобы вычислять
  sinA[j] := sin(j * Pi / 180);      // только один раз
  cosA[j] := cos(j * Pi / 180);
end;
PS := TParticleSystem.Create;             // Создание системы
PS.Init (DEFAULT_SIZE, DEFAULT_POWER);    // Инициализация системы
```

В методе `Calculate` класса вспышки пересчитываются текущие координаты частиц:

```
procedure TParticleSystem.Calculate;
var
  i : Integer;
begin
  for i := 0 to Size do begin
    if Particle[i].Energy > 0 then begin
      Particle[i].X := Particle [i].X + Particle [i].SpeedX;
      // Частицы отскакивают от границ экрана
      if Particle[i].X >= ScreenWidth – 1 then begin
        Particle[i].SpeedX := -0.5 * Particle [i].SpeedX;
        Particle[i].X := ScreenWidth – 1;
      end;
      if Particle[i].X < 0 then begin
        Particle[i].SpeedX := -0.5 * Particle [i].SpeedX;
        Particle[i].X := 0;
      end;
      Particle [i].Y := Particle[i].Y + Particle[i].SpeedY;
      if Particle[i].Y >= ScreenHeight – 1 then begin
        Particle[i].SpeedY := -0.3 * Particle[i].SpeedY;
        Particle[i].Y := ScreenHeight – 1;
      end;
      if Particle[i].Y < 0 then begin
        Particle[i].SpeedY := -Particle[i].SpeedY;
```

```
        Particle[i].Y := 0;
      end;
      Particle[i].Energy := Particle[i].Energy — 1;
      Particle[i].SpeedY := Particle[i].SpeedY + 0.2;
    end;
  end;
end;
```

Самый главный для нас метод — воспроизведение частиц системы:

```
function TParticleSystem.Render : HRESULT;
var
  i : Integer;
  desc : TDDSURFACEDESC2;
  hRet : HRESULT;
begin
  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf(desc);
  hRet := frmDD.FDDSBack.Lock (nil, desc, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Очистка экрана
  ZeroMemory (desc.lpSurface,
              desc.lPitch * ScreenHeight * (ScreenBitDepth div 8));
  // Заполняем пикселы в соответствии с состоянием системы частиц
   for i := 0 to Size do
     if (Particle[i].Energy > 0) then
       PWord (Integer(desc.lpSurface) +
       trunc (Particle[i].Y) * desc.lPitch +
       trunc (Particle[i].X) * (ScreenBitDepth div 8))^ :=
       Particle[i].B or (Particle[i].G shl 5) or (Particle[i].R shl 11);
   Result := frmDD.FDDSBack.Unlock(nil);
end;
```

При каждой перерисовке экрана отображается текущее состояние системы:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  hRet : HRESULT;
begin
  Result := DD_FALSE;
  PS.Calculate;                // Пересчитываем положения частиц
  // Воспроизведение состояния системы
  hRet := PS.Render;
  if Failed (hRet) then begin
```

```
    Result := hRet;
    Exit;
  end;
  Time := Time + 1;             // Простейший эмулятор таймера
  if Time > 15 then begin       // Прошел срок существования системы
    PS.Init(DEFAULT_SIZE, DEFAULT_POWER);   // Вспышка в новом месте
    Time := 0;
  end;
  Result := DD_OK;
end;
```

Полупрозрачность

Такой прием часто используется в играх. Автоматизации полупрозрачности DirectDraw не предоставляет, все необходимо делать самому разработчику, попикселно накладывая данные источника и приемника.

В общем случае формула вычисления значения цветовых компонентов выглядит так:

```
Result = Alpha * srcColor + (1 – Alpha) * destColor
```

Здесь `Alpha` — коэффициент прозрачности, принимающий вещественное значение в пределах от нуля до единицы; `srcColor` — цвет источника; `destColor` — цвет приемника.

Если `Alpha` равно нулю, то получаем цвет приемника; если `Alpha` имеет единичное значение, источник совершенно непрозрачен. Если мы имеем дело с образом, двигающимся по поверхности, то под источником подразумеваем образ, а фон считаем приемником.

Формулу можно оптимизировать. Начнем с того, что избавимся от присутствия двух операций умножения. Перестроим уравнение так, чтобы присутствовала лишь одна из них:

```
Result = Alpha * srcColor + destColor  – Alpha * destColor
```

или

```
Result = Alpha * (srcColor – destColor) + destColor
```

Коэффициент прозрачности имеет смысл представлять целым, чтобы все вычисления производить только с целыми числами. Считая `Alpha` целым в интервале 0 — 256, окончательную формулу расчета составляющей запишем так:

```
Result = (Alpha * (srcColor – destColor)) / 256 + destColor
```

Все предваряющие слова сказаны, можем перейти к иллюстрации — проекту каталога Ex18, при работе которого по знакомому фону перемещается полупрозрачный образ насекомого (рис. 3.8).

Рис. 3.8. Момент работы эффектного примера на полупрозрачность

Массив `Pict` содержит битовую карту растра:

```
const
  imageWidth  = 84;
  imageHeight = 80;
  Alpha       = 127;
var
  Pict : Array [0..imageWidth - 1, 0..imageHeight - 1] of Word;
  ColorKey : Word;     // Вспомогательный цветовой ключ
```

Поверхность образа не выводится на экран, а служит только для заполнения массива `Pict`:

```
function TfrmDD.Prepare : HRESULT;
var
  desc : TDDSURFACEDESC2;
  i, j : Integer;
  hRet : HRESULT;
begin
  Result := DD_FALSE;
  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf(desc);
  hRet := FDDSImage.Lock (nil, desc, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
     Result := hRet;
     Exit;
  end;
```

```
  // Заполнение массива Pict
  for i := 0 to imageWidth - 1 do
    for j := 0 to imageHeight - 1 do
      Pict [i, j] := PWORD (Integer (desc.lpSurface) + j * desc.lPitch +
      i * (ScreenBitDepth div 8))^;
  ColorKey := Pict [0, 0];        // Определяемся с цветовым ключом
  Result := FDDSImage.Unlock (nil);
end;
```

Для простоты в качестве цветового ключа возьмем значение самого первого пиксела образа, считая, что цвет его совпадает с цветом фона. Для ускорения работы примера воспользуемся приемом с частичным обновлением экрана:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  X, Y : Integer;
  wrkRect : TRECT;
  hRet : HRESULT;
begin
  ThisTickCount := GetTickCount;
  if ThisTickCount - LastTickCount > 60 then begin
    X := 288 + trunc (cos(Angle) * 150);
    Y := 208 + trunc (sin(Angle) * 150);
    // Старая позиция образа
    SetRect (wrkRect, X, Y, X + imageWidth, Y + imageHeight);
    Angle := Angle + 0.05;
    if Angle > 2 * Pi then Angle := Angle - 2 * Pi;
    // Вывод полупрозрачного образа в задний буфер
    hRet := Blend (288 + trunc (cos(Angle) * 150),
                   208 + trunc (sin(Angle) * 150));
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    // Переключаем страницы
    hRet := FlipPages;
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    // Стираем образ в заднем буфере
    hRet := FDDSBack.Blt (@wrkrect, FDDSBackGround, @wrkRect,
                          DDBLT_WAIT, nil);
    if Failed (hRet) then begin
```

```
      Result := hRet;
      Exit;
    end;
    LastTickCount := GetTickCount;
  end;
  Result := DD_OK;
end;
```

Итак, осталось рассмотреть собственно функцию вывода полупрозрачного образа:

```
function TfrmDD.Blend (const X, Y : Integer) : HRESULT;
var
  desc : TDDSURFACEDESC2;
  i, j : Integer;
  wrkPointer : PWORD;
  sTemp, dTemp : WORD;
  sb, db, sg, dg, sr, dr : Byte;
  blue, green, red : Byte;
  hRet : HRESULT;
begin
  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf(desc);
  hRet := FDDSBack.Lock (nil, desc, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
     Result := hRet;
     Exit;
  end;
  for i := 0 to imageWidth - 1 do
    for j := 0 to imageHeight - 1 do
    // Только для точек с цветом, отличным от цвета фона
      if Pict [i, j] <> ColorKey then begin
        wrkPointer := PWORD (Integer(desc.lpSurface) +
          (Y + j) * desc.lPitch + (X + i) * (ScreenBitDepth div 8));
        sTemp := Pict [i, j]; // Пиксел источника, точка образа
        dTemp := wrkPointer^; // Приемник, фоновая картинка
        sb := sTemp and $1f;  // Синий цвет источника
        db := dTemp and $1f;  // Синий цвет приемника
        sg := (sTemp shr 5)  and $3f; // Зеленый цвет источника
        dg := (dTemp shr 5)  and $3f; // Зеленый цвет приемника
        sr := (sTemp shr 11) and $1f; // Красный цвет источника
        dr := (dTemp shr 11) and $1f; // Красный цвет приемника
        blue  := (ALPHA * (sb - db) shr 8) + db; // Результат, синий
        green := (ALPHA * (sg - dg) shr 8) + dg; // Результат, зеленый
        red   := (ALPHA * (sr - dr) shr 8) + dr; // Результат, красный
```

```
        // Сложение цветовых компонентов в пикселе приемника
        wrkPointer^ := blue or (green shl 5) or (red shl 11);
      end;
  Result := FDDSBack.Unlock (nil);
end;
```

Вы должны обратить внимание, что фон в примере заполняется растянутым растровым изображением. Мы уже обсуждали проблему, связанную с использованием метода `DDReLoad` в таких случаях. Чтобы при распахивании минимизированного окна картинка не превращалась в мозаику, перезагрузим растр:

```
function TfrmDD.RestoreAll : HRESULT;
var
  hRet : HRESULT;
begin
  hRet := FDDSPrimary._Restore;
  if Succeeded (hRet) then begin
    FDDSBackGround := nil;           // Удаление поверхности
    FDDSBackGround := DDLoadBitmap(FDD, groundBmp, ScreenWidth,
                      ScreenHeight); // Заново создаем поверхность фона
    if FDDSBackGround = nil then ErrorOut(DD_FALSE, 'DDLoadBitmap');
    if FDDSBackGround = nil then ErrorOut(DD_FALSE, 'DDLoadBitmap');
    hRet := FDDSPrimary.Blt (nil, FDDSBackGround, nil, DDBLT_WAIT, nil);
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    Result := FDDSBack.Blt (nil, FDDSBackGround, nil, DDBLT_WAIT, nil);
  end else Result := hRet;
end;
```

Картинка загружается заново, и в случае неудачи загрузки программа заканчивает работу.

Обратите внимание, что в примере растр для заполнения фона берется 24-битным, а второй, накладываемый, растр имеет разрядность 8 бит, т. е. используется 256-цветный рисунок. В таких случаях не требуется загружать палитру из этого рисунка, поскольку все цвета при переносе на 24-битную поверхность отображаются корректно. Формат пиксела первичной поверхности задает формат пиксела и для всех остальных поверхностей. Не должна возникать ситуация, когда на 8-битную первичную поверхность помещается 16-битный образ. Также палитра, устанавливаемая для первичной поверхности, задается для всех остальных поверхностей. В таких примерах мы не загружали и не устанавливали палитры ни для одной поверхности, кроме первичной. Из-за этого в примерах с летающим драконом его цвета немного искажались, для отображения использовалась палитра фоновой поверхности.

Теоретически, DirectDraw сам проследит, чтобы не возникло разнобоя в установках поверхностей, но я думаю, что если вы будете явно устанавливать одинаковый формат для всех поверхностей, то только повысите корректность работы программы, особенно в случае оконных приложений.

Использование полупрозрачности позволит придать нашим проектам потрясающую эффектность, такую, как в следующем, очень интересном, примере — проекте каталога Ex19. Идея такова: после запуска приложения содержимое рабочего стола копируется на первичную поверхность, а по ходу работы появляется полупрозрачное изображение. У пользователя создается ощущение того, что приложение осуществляет вывод прямо на рабочий стол. Но мы этого не делаем, иначе окно приложения нарушит иллюзию.

Для простоты накладываем одно ограничение: считаем разрешение экрана 16-битным, размеры рабочего стола — 640×480 пикселов. Обратите внимание на это, при других установках рабочего стола пример работает не так эффектно.

Сразу после запуска приложения до появления на экране окна нашего приложения, копируем во вспомогательный объект класса `TBitmap` содержимое рабочего стола:

```
wrkBitmap := TBitmap.Create;
wrkBitmap.Height := 480;
wrkBitmap.Width := 640;
BitBlt(wrkBitmap.Canvas.Handle, 0, 0, 640, 480, GetDC (GetDesktopWindow),
       0, 0, SRCCOPY);
```

Поверхность фона создается "длинным" способом. При этом не загружаем ничего из растра:

```
ZeroMemory (ddsd, SizeOf(ddsd), 0);
with ddsd do begin
  dwSize := SizeOf(ddsd);
  dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
  ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
  dwWidth  := 640;
  dwHeight := 480;
end;
hRet := FDD.CreateSurface(ddsd, FDDSBackGround, nil);
if Failed(hRet) then ErrorOut(hRet, 'Create Back Surface');
// Копируем содержимое wrkBitmap на фоновую поверхность
hRet := DDCopyBitmap (FDDSBackGround, wrkBitmap.Handle, 0, 0,
                      wrkBitmap.Width, wrkBitmap.Height);
if Failed(hRet) then ErrorOut(hRet, 'DDCopyBitmap');
wrkBitmap.Free;      // wrkBitmap больше не требуется
```

Дальше все происходит традиционно: на первичную поверхность выводится содержимое поверхности фона, поверх которого накладывается полупро-

зрачный образ. Чтобы добиться зловещего эффекта призрачного колебания, первоначальный массив образа искажается по синусоиде:

```
function TfrmDD.Rotate (const pictOriginal : TWordArray) : TWordArray;
var
  i, j, k : Integer;
begin
  ZeroMemory (@Result, SizeOf (Result));
  for j := 0 to 255 do
    for i := 0 to 255 do begin
      k := trunc (sin (Angle + j * 3 * Pi / 255) * 10);   // Сдвиг точек
      if (i - k >= 0) and (i - k <= 255) then   // Помещается ли в растр
        Result [i, j] := pictOriginal [i - k, j];
    end;
  Angle := Angle + 0.2;                          // Периодичный сдвиг
  if Angle > 2 * Pi then Angle := Angle - 2 * Pi;// Избежать переполнения
end;
```

Пользователь может перемещать курсор, попытаться выполнить привычные действия, видя знакомое содержимое экрана, но реакции на его действия, конечно, не последует. Чтобы у него не возникало паники, закрываем приложение по нажатию любой клавиши, иначе у пользователя может возникнуть ощущение того, что система зависла.

Выбор объектов

В этом разделе мы познакомимся с простейшим способом организации выбора и выяснением, какой объект находится в определенной точке экрана, например под курсором. В простейших проектах, конечно, можно всегонавсего анализировать координаты нужной точки и перебирать все отображаемые объекты, чтобы выделить из них нужный. Но если объектов присутствует очень много, а сами они бесформенные или имеют сложную форму, то на перебор и анализ может уйти слишком много времени.

В таких случаях используется выбор по цвету, заключающийся в том, что объекты раскрашиваются в различные цвета, анализ цвета нужной точки дает ответ на вопрос: "Что в настоящий момент находится под курсором".

Рассмотрим пример из проекта каталога Ex20. На экране перемещаются три одинаковых образа, при этом образ, находящийся под курсором, перекрашивается (рис. 3.9).

Поскольку образы выводятся совершенно одинаковые, мы не можем напрямую различать их по цвету. Действуем так: на вспомогательной поверхности `FDDSDouble` отображаем образы такой же формы, что и на экране, но разные по цвету (в моем примере это три круга чистых цветов: красного, зеленого и синего). Выводятся они с теми же координатами, что и на экране. Перед

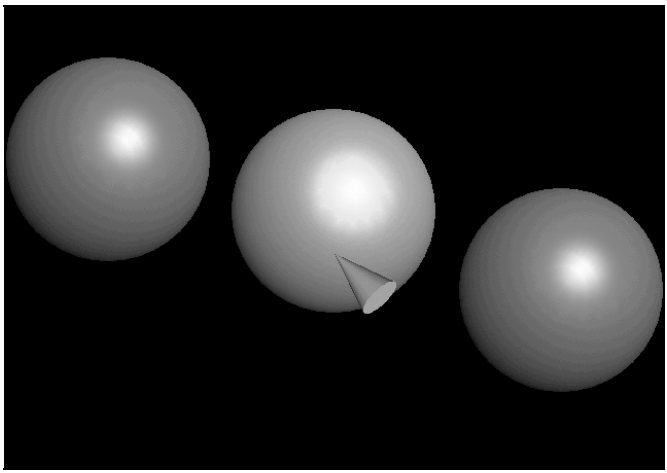


Рис. 3.9. Фрагмент работы проекта выбора объектов

тем, как отобразить сферы на экране, анализируем цвет нужного пиксела на вспомогательной поверхности:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  ddbltfx : TDDBLTFX;   // Для очистки экрана
  wrkI : Integer;       // Рабочая переменная
begin
  Result := DD_FALSE;
  ZeroMemory (@ddbltfx, SizeOf(ddbltfx));
  ddbltfx.dwSize := SizeOf(ddbltfx);
  ddbltfx.dwFillColor := 0;
  // Закрашиваем, очищаем обе поверхности
  FDDSBack.Blt(nil, nil, nil, DDBLT_COLORFILL or DDBLT_WAIT, @ddbltfx);
  FDDSDouble.Blt(nil, nil, nil, DDBLT_COLORFILL or DDBLT_WAIT, @ddbltfx);
  ThisTickCount := GetTickCount;
  // Пауза для смены положения сфер
  if ThisTickCount - LastTickCount > 10 then begin
     Angle := Angle + 0.02;
     if Angle > 2 * Pi then Angle := Angle - 2 * Pi;
     LastTickCount := GetTickCount;
  end;
  // Выводим три сферы на вспомогательную поверхность
  FDDSDouble.BltFast (0, 140 - trunc (sin (Angle) * 100),
                      FDDSImageRed, nil, DDBLTFAST_WAIT);
                     // Красная, соответствует первому образу
  FDDSDouble.BltFast (230, 140 - trunc (sin (Angle + Pi / 4) * 100),
                      FDDSImageGreen, nil, DDBLTFAST_WAIT);
```

```
                    // Зеленая, для второго образа
  FDDSDouble.BltFast (440, 140 - trunc (sin (Angle + Pi / 2) * 100),
                      FDDSImageBlue, nil, DDBLTFAST_WAIT);
                      // Синяя для третьего
  wrkI := Select (mouseX, mouseY);   // Выбор элемента под курсором
  if wrkI = -1 then begin            // Произошла авария
    Result := RestoreAll;
    Exit;
  end;
  if wrkI = 1    // Под курсором первая сфера, ее выводим помеченной
    then FDDSBack.BltFast (0, 140 - trunc (sin (Angle) * 100),
                           FDDSImageSelect, nil, DDBLTFAST_WAIT)
  // Под курсором не первая сфера, ее выводим обычной
     else FDDSBack.BltFast (0, 140 - trunc (sin (Angle) * 100),
          FDDSImageSphere, nil, DDBLTFAST_WAIT);
  // Аналогично с двумя оставшимися сферами
  if wrkI = 2
    then FDDSBack.BltFast (220, 140 - trunc (sin (Angle + Pi / 4) * 100),
         FDDSImageSelect, nil, DDBLTFAST_WAIT)
    else FDDSBack.BltFast (220, 140 - trunc (sin (Angle + Pi / 4) * 100),
         FDDSImageSphere, nil, DDBLTFAST_WAIT);
  if wrkI = 3
    then FDDSBack.BltFast (440, 140 - trunc (sin (Angle + Pi / 2) * 100),
         FDDSImageSelect, nil, DDBLTFAST_WAIT)
    else FDDSBack.BltFast (440, 140 - trunc (sin (Angle + Pi / 2) * 100),
         FDDSImageSphere, nil, DDBLTFAST_WAIT);
  // Вывод указателя курсора
  FDDSBack.BltFast (mouseX, mouseY, FDDSMouse, nil,
                    DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
  if Failed (FlipPages)
    then Result := RestoreAll
    else Result := DD_OK;
end;
```

Теперь посмотрим функцию выбора:

```
function TfrmDD.Select (const X, Y : Integer) : Integer;
var
  desc : TDDSURFACEDESC2;
  Red, Green, Blue : Byte;
  Pixel : Word;
begin
  Result := -1;
  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf(desc);
  if Failed (FDDSDouble.Lock (nil, desc, DDLOCK_WAIT, 0))
    then Exit;                          // Закрыть не удается, выходим
```

```
  Pixel := PWord (Integer (desc.lpSurface) + Y * desc.lPitch + X * 2)^;
  Blue := Pixel and $1F;                // Цветовые компоненты пиксела
  Green := (Pixel shr 5) and $3F;
  Red := (Pixel shr 11) and $1F;
  FDDSDouble.Unlock (nil);
  if Blue <> 0 then Result := 3 else   // Анализируем результат
  if Green <> 0 then Result := 2 else
  if Red <> 0 then Result := 1 else Result := 0;
end;
```

Конечно, для этого конкретного примера можно делать выбор просто по координате, но я надеюсь, что сумел достичь данной иллюстрацией понимания, как поступать в случаях, когда подобный анализ получается слишком длинным.

В рассмотренном примере фон не используется. Но если он потребуется, то учтите, что на вспомогательную поверхность его выводить совершенно не нужно.

В некоторых стратегических играх вы можете заметить, что на экране можно выбирать "спрятавшиеся" объекты, закрытые каким-то элементом пейзажа, деревом или горой. На вспомогательной поверхности они не рисуются, поэтому так и происходит.

Также часто возникают ситуации, когда выбор осуществляется неточно, в некотором районе объекта. Это происходит потому, что для повышения скорости работы на вспомогательную поверхность выводятся окрашенные прямоугольники, а не силуэт объекта.

Лупа

В данном разделе мы рассмотрим два любопытных примера, посвященных организации лупы. Задача сама по себе занимательна, но вдобавок мы узнаем кое-что интересное и загадочное.

Запустите проект, располагающийся в каталоге Ex21. По экрану перемещается "лупа", кружок, в пределах которого выводится увеличенный участок фона (рис. 3.10).

В качестве фона в примерах этого раздела я использую, с любезного разрешения автора, работы художника, имя которого присутствует в левом нижнем углу растрового изображения. Псевдоним автора — Beardo, а адрес его страницы **http://home5.swipnet.se/~w-57902/images/art/**.

Изобразить увеличенный участок фона — задача не из трудных, мы хорошо усвоили метод `Blt` поверхности. Проблема состоит в том, чтобы вывести не прямоугольную, а именно круглую лупу. Посмотрим, как это сделано в данном примере.

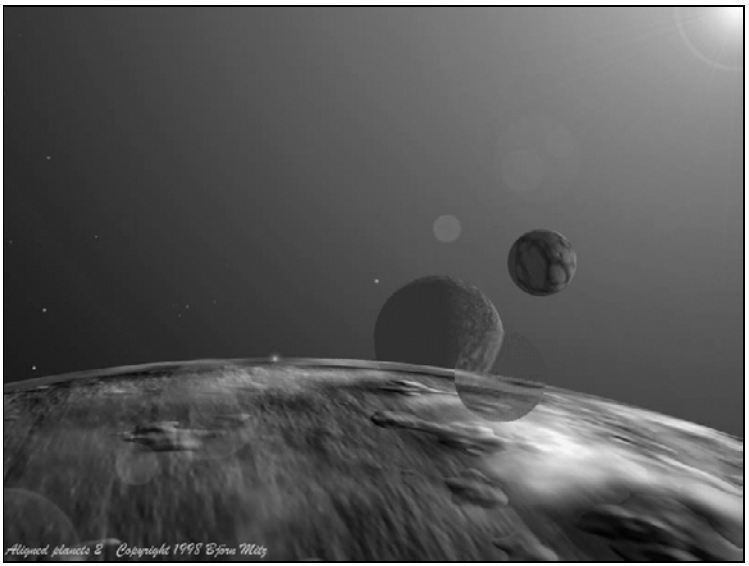


Рис. 3.10. Имитация лупы

Поверхность, связанная с лупой, называется `FDDSZoom`, для нее установлен цветовой ключ — черный цвет. Размер поверхности — 100×100 пикселов.

Все точки этой поверхности, находящиеся за пределами круга "лупы", окрашиваются черным:

```
function TfrmDD.Circle : HRESULT;
var
  desc : TDDSURFACEDESC2;
  i, j : Integer;
  hRet : HRESULT;
begin
  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf (desc);
  hRet := FDDSZoom.Lock (nil, desc, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  for i := 0 to 99 do          // Цикл по всем точкам поверхности
    for j := 0 to 99 do
    // Выделяем точки, располагающиеся за пределами круга "лупы"
      if sqr (i - 50) + sqr (j - 50) > 50 * 50 then   // Заполняем черным
        PWord (Integer(desc.lpSurface) + j * desc.lPitch + i * 2)^ := 0;
  Result := FDDSZoom.Unlock (nil);
end;
```

При отображении цветовой ключ позволяет ограничить вывод растянутой поверхности именно кругом:

```
// Квадрат, задающий степень увеличения
SetRect (wrkRect, mouseX + 25, mouseY + 25, mouseX + 75, mouseY + 75);
// Растягиваем участок фона
FDDSZoom.Blt (nil, FDDSBackGround, @wrkRect, DDBLT_WAIT, nil);
Circle;          // Заполняем черным часть квадрата
// Выводим с цветовым ключом
FDDSBack.BltFast (mouseX, mouseY, FDDSZoom, nil,
                  DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
```

Выглядит просто и эффектно, но в решении содержится проблема: оно подходит только для черного цвета. Если в качестве ключа использовать любой другой цвет, то на точки, заполненные цветом ключа "вручную", прозрачность распространяться не будет: прозрачными окажутся только участки этого же цвета, но окрашенные вызовом метода поверхности. Разрешить означенную проблему мне не удалось, поскольку плохо понятно, как DirectDraw удается различать такие участки.

Черный цвет для использования его в качестве ключа подходит для этого фона, но пример будет некрасиво работать с фоном, подобным рис. 3.11, где присутствует масса участков именно черного цвета.

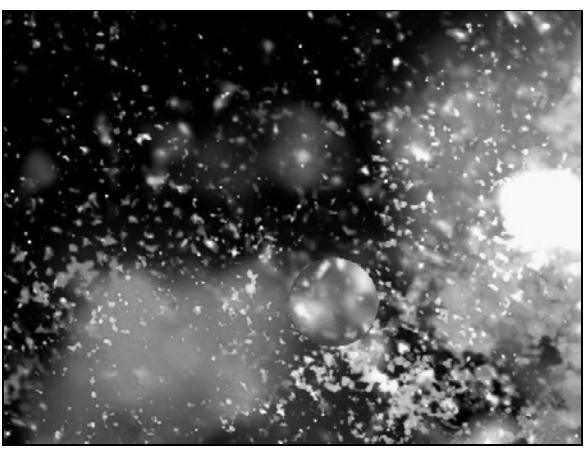


Рис. 3.11. Работа усложненного примера на создание лупы

В проекте каталога Ex22 приведено другое решение задачи, менее элегантное, но работающее с любыми цветовыми ключами.

Здесь, помимо поверхности `FDDSZoom`, введена поверхность `FDDSDouble`. Для первой из них в качестве ключа взят чистый зеленый цвет, как отсутствующий на фоне. Вторая поверхность создается путем загрузки изображения-

шаблона — зеленый квадрат с черным кругом посередине. Ключом для нее установлен черный цвет.

Теперь на поверхность лупы последовательно помещаем растянутый участок фона, затем ограничивающий шаблон:

```
SetRect (wrkRect, mouseX + 25, mouseY + 25, mouseX + 75, mouseY + 75);
// Растягиваем участок фона
FDDSZoom.Blt (nil, FDDSBackGround, @wrkRect, DDBLT_WAIT, nil);
// Вместо черных участков шаблона останется увеличенный фрагмент
FDDSZoom.BltFast (0, 0, FDDSDouble, nil,
                  DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
// Зеленая канва не воспроизведется
FDDSBack.BltFast (mouseX, mouseY, FDDSZoom, nil,
                  DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
```

Позже мы вернемся к задаче с лупой и получим искаженное изображение в ее круге.

Палитры

Для хранения отдельного набора цветовых составляющих палитры используется переменная типа `TPaletteEntry`. Переменная типа `IDirectDrawPalette`, как мы знаем из предыдущих примеров, служит для установления определенной палитры 8-битной поверхности. Обычно такая палитра загружается из растра.

В любой момент времени мы можем получить набор палитры и модифицировать его, как это делается в следующем нашем примере (проект каталога Ex23). За основу взят проект с перемещающимся драконом, но здесь с течением времени экран становится тусклым, имитируется суточная смена освещенности. Дойдя до некоторой фазы, восстанавливается первоначальная яркость. Такой эффект постепенного угасания называется *fade* (затухание). Разберем, как он создается.

Для хранения первоначальной палитры предназначен массив:

```
DefPal : Array[0..255] of TPaletteEntry;
```

Массив заполняется после загрузки палитры из растра, для чего вызывается метод палитры `GetEntries`:

```
hRet := FDDpal.GetEntries(0, 0, 256, @DefPal);
if Failed (hRet) then ErrorOut(hRet, 'Palette GetEntries');
```

При каждом перемещении образа во всех составляющих текущей палитры убавляются веса цветов, используется локальный массив `PalEntries`:

```
// Получаем составляющие текущей палитры экрана
FDDpal.GetEntries(0, 0, 256, @PalEntries);
```

```
for i := 0 to 255 do begin          // Цикл по всем элементам палитры
  if PalEntries[i].peRed > Step then PalEntries[i].peRed :=
     PalEntries[i].peRed – Step;
   if PalEntries[i].peGreen > Step then PalEntries[i].peGreen :=
     PalEntries[i].peGreen – Step;
   if PalEntries[i].peBlue > Step then PalEntries[i].peBlue :=
     PalEntries[i].peBlue – Step;
end;
// Устанавливаем текущей палитру, образованную элементами массива
FDDPal.SetEntries(0, 0, 256, @PalEntries);
Timer := (Timer + 1) mod 100;
// Восстанавливаем первоначальную палитру
if Timer = 0 then FDDpal.SetEntries(0, 0, 256, @DefPal);
```

Эффект угасания часто применяется для необычного завершения работы приложения.

Модификация палитры может использоваться также для создания эффекта *цветовой анимации*. Для этого различные участки поверхности рисуются в индивидуальных цветах, а при поочередном затемнении некоторых цветовых наборов палитры создается эффект перемещения, на экране последовательно появляются отдельные образы.

Рассмотрим простейший пример на эту тему — проект каталога Ex24. Фон представляет собой рисунок, построенный серией эллипсов, нарисованных оттенками серого; цвета повторяются в каждой серии (рис. 3.12).

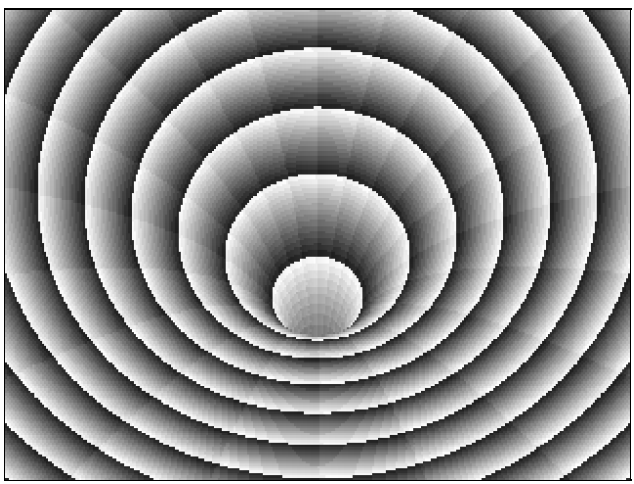


Рис. 3.12. Фон примера на палитровую анимацию

Равномерно удаленные компоненты палитры с течением времени последовательно заменяются желтоватым цветом, остальные элементы ее затемня-

ются. На экране по очереди появляются близко расположенные окружности и возникает иллюзия их движения.

Целочисленная переменная `kr` задает текущую незатемняемую палитру и изменяется от шестнадцати до двух, уменьшаясь на каждом шаге:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  k : Integer;
  DefPal : Array[0..255] of TPaletteEntry;    // Массив цветов палитры
  hRet : HRESULT;
begin
  ThisTickCount := GetTickCount;
  if ThisTickCount - LastTickCount > 10 then begin
    // Берем текущую палитру
    hRet := FDDPal.GetEntries(0, 0, 256, @DefPal);
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    for k := 0 to 14 do begin     // Затемняем предыдущий цвет палитры
      DefPal [kr * 15 + k].peBlue := 0;
      DefPal [kr * 15 + k].peRed := 0;
      DefPal [kr * 15 + k].peGreen := 0;
    end;
    Dec (kr);                     // Переходим к следующему цвету палитры
    if kr < 2 then kr := 16;
    for k := 0 to 14 do begin    // Подменяем текущий цвет желтоватым
      DefPal [kr * 15 + k].peBlue := 0;
      DefPal [kr * 15 + k].peRed := 128;
      DefPal [kr * 15 + k].peGreen := 100;
    end;
    // Устанавливаем измененную палитру
    hRet := FDDPal.SetEntries(0, 0, 256, @DefPal);
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    LastTickCount := GetTickCount;
    hRet := FDDSPrimary.Flip(nil, DDFLIP_WAIT);
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
  end;
  Result := DD_OK;
end;
```

Обратите внимание, что при запуске и восстановлении приложения появляется первоначальная фоновая картинка, поскольку затеняются "ненужные" цвета палитры только после первого прохождения значения `kr` по кругу.

Оконные приложения

Вы, должно быть, уже привыкли к тому, что наши примеры работают в полноэкранном режиме. Обычно оконные приложения создаются в DirectDraw только в случае крайней необходимости, т. к. слишком многое мы теряем при его использовании. Главная потеря здесь — скорость работы приложения.

Примечание

Скорость воспроизведения, в общем случае, выше у полноэкранных приложений. Конечно же, при отображении небольшого числа блоков в маленьком окне вы сможете добиться очень высокой скорости.

Для оконных приложений нельзя использовать переключение страниц или двойную буферизацию.

По своей сути, оконные приложения похожи на наши самые первые примеры, с выводом кружка на поверхности окна. Точно так же первичной поверхностью является весь рабочий стол экрана, и приложение должно отслеживать положение окна.

Рассмотрим простейший пример, располагающийся в каталоге Ex25. Работа его совсем проста, в пределах окна выводится хорошо нам знакомый растр с горным пейзажем.

Свойство `Borderstyle` формы приняло теперь свое обычное значение `bsSizeable`, удалены единственный компонент и все, связанное с курсором. Не можем мы также здесь задавать параметры экрана и устанавливать исключительный уровень кооперации, поскольку для оконных приложений задается обычный уровень доступа:

```
hRet := FDD.SetCooperativeLevel(Handle, DDSCL_NORMAL);
```

Появился обработчик перерисовки окна, в котором определяем текущее положение окна приложения и выводим на него масштабированный растр:

```
procedure TfrmDD.FormPaint(Sender: TObject);
var
  rcDest : TRECT;
  p : TPOINT;   // Вспомогательная точка для определения положения окна
begin
  p.X := 0;
  p.Y := 0;
```

```
  // Находим положение на экране точки левого верхнего угла
  // клиентской части окна приложения
  Windows.ClientToScreen(Handle, p);
  // Получаем прямоугольник размерами клиентской части окна
  Windows.GetClientRect(Handle, rcDest);
  OffsetRect(rcDest, p.X, p.Y);   // Сдвигаем прямоугольник на p.X, p.Y
  if Failed (FDDSPrimary.Blt (@rcDest, FDDSBackGround, nil,
             DDBLT_WAIT, nil))    // Выводим растр
    then RestoreAll;
end;
```

Перед именами некоторых процедур я указал, что имеются в виду процедуры именно модуля `Windows`. Если для вас привычнее использовать аналогичные методы формы, то эти строки можете записать так:

```
p := ClientToScreen(p);
rcDest := GetClientRect;
```

Хоть в рассматриваемом примере и приходится следовать рекомендациям разработчиков, пытаясь восстанавливать все поверхности в случае неудачи при воспроизведении, но сделано это большей частью формально. Если по ходу приложения поменять установки экрана, то оно не сумеет восстановить первичную поверхность. Это не является недостатком конкретно нашего примера. Точно так же ведут себя все оконные приложения, использующие DirectDraw.

Если вы внимательно посмотрите на работу приложения, то должны заметить, как плохо масштабируется картинка при изменении размеров окна. Для более удовлетворительной работы обработчик этого события должен вызывать код перерисовки окна.

Оконное приложение может рисовать в любом месте рабочего стола. Малейшая ошибка в коде приведет к очень некрасивым результатам. Обычно такие приложения ограничивают область вывода, для чего используется объект класса `IDirectDrawClipper`.

Посмотрим проект каталога Ex26, в коде модуля которого появилась переменная `FDDClipper` такого типа. В начале и конце работы приложения ее значение, как принято, устанавливается в `nil`.

Сразу после создания первичной поверхности формируется этот объект, ответственный за отсечение области вывода, и присоединяется к области вывода:

```
// Создание объекта отсечения
hRet := FDD.CreateClipper(0, FDDClipper, nil);
if Failed (hRet) then ErrorOut(hRet, 'CreateClipper FAILED');
// Определяем окно, связанное с отсечением области вывода
hRet := FDDClipper.SetHWnd(0, Handle);
if Failed (hRet) then ErrorOut(hRet, 'SetHWnd FAILED');
```

```
// Устанавливаем объект отсечения для первичной поверхности
hRet := FDDSPrimary.SetClipper(FDDClipper);
if Failed (hRet) then ErrorOut(hRet, 'SetClipper FAILED');
```

Можете для проверки работы отсечения удалить в коде строку с вызовом `OffsetRect`, чтобы принудить приложение воспроизводить за границами своей клиентской области. Картинка окажется искаженной, но приложение уже не испортит вид рабочего стола.

Одно небольшое замечание. В программах DirectX SDK можно обнаружить, что объект отсечения не удаляется по окончании работы, я же делаю это в моих примерах намеренно. Легко проверить это: объект, связанный с отсечением, имеет по окончании работы значение, отличное от `nil`, а в таком случае лучше будет явным образом освобождать память, занятую им. Также иногда можно встретить, что эта переменная присваивается `nil` сразу после присоединения к первичной поверхности.

Важно подчеркнуть, что при использовании отсечения нельзя применять для вывода на первичную поверхность метод `BltFast`.

Следующий пример, проект каталога Ex27, продолжает тему оконных приложений, отличается он от предыдущего пользовательским курсором (рис. 3.13).

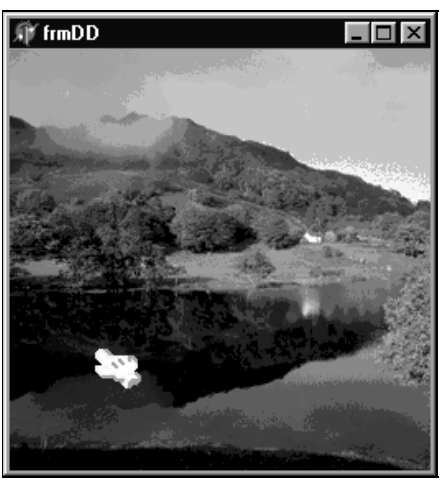


Рис. 3.13. Фрагмент работы примера с использованием буферизации в оконном приложении

Буферизацию приходится организовывать самостоятельно, для экономии памяти вспомогательная поверхность создается при каждом изменении размеров окна:

```
procedure TfrmDD.FormResize(Sender: TObject);
var
  hRet : HRESULT;
  ddsd : TDDSurfaceDesc2;
```

```
begin
  if Assigned(FDDSBack) then FDDSBack := nil;
  ZeroMemory(@ddsd, SizeOf(ddsd));
  with ddsd do begin
    dwSize := SizeOf(ddsd);
    dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
    ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
    dwWidth  := ClientWidth;    // Размеры совпадают с текущими размерами
    dwHeight := ClientHeight;   // окна
  end;
  hRet := FDD.CreateSurface(ddsd, FDDSBack, nil);
  if Failed(hRet) then ErrorOut(hRet, 'Create Back Surface');
  FormPaint (nil);
end;
```

Обратите внимание, что в этом примере пользовательским курсором можно указать на любую точку клиентской области окна. Для отсечения нужной части поверхности образа (ее размер 32×32 пиксела) объявлена переменная `rcMouse` типа `TRECT`. При перемещении курсора вблизи границы окна оставляем для воспроизведения только часть образа:

```
procedure TfrmDD.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
var
  wrkI, wrkJ : Integer;
begin
  mouseX := X;
  if X < ClientWidth - 32
    then wrkI := 32                // По X помещается весь растр
    else wrkI := ClientWidth - X;  // Воспроизводить только часть образа
  mouseY := Y;
  if Y < ClientHeight - 32
    then wrkJ := 32                // По Y помещается весь растр
    else wrkJ := ClientHeight - Y; // Воспроизводить только часть образа
  SetRect (rcMouse, 0, 0, wrkI, wrkJ);   // Итоговый прямоугольник образа
  FormPaint (nil);                 // Принудительно перерисовываем окно
end;
```

При перерисовке окна метод `BltFast` приходится использовать только для вывода растрового изображения курсора:

```
procedure TfrmDD.FormPaint(Sender: TObject);
var
  rcDest, wrkRect : TRECT;
  p : TPOINT;
begin
  p.X := 0;
  p.Y := 0;
```

```
  Windows.ClientToScreen(Handle, p);
  Windows.GetClientRect(Handle, rcDest);
  OffsetRect(rcDest, p.X, p.Y);
  SetRect (wrkRect, 0, 0, ClientWidth, ClientHeight);
  // На вспомогательную поверхность помещаем растровое изображение фона
  if Failed (FDDSBack.Blt (@wrkRect, FDDSBackGround, nil,
             DDBLT_WAIT, nil)) then if Failed (RestoreAll) then Exit;
  // Поверх фона размещаем растровое изображение курсора
  if Failed (FDDSBack.BltFast (mouseX, mouseY, FDDSImage, @rcMouse,
             DDBLTFAST_WAIT or  DDBLTFAST_SRCCOLORKEY))
    then if Failed (RestoreAll) then Exit;
  // Копируем содержимое вспомогательной поверхности на первичную
  if Failed (FDDSPrimary.Blt (@rcDest, FDDSBack, nil, DDBLT_WAIT, nil))
    then if Failed (RestoreAll) then Exit;
end;
```

Для отключения отображения курсора в этом примере прибегнем к альтернативному способу: воспользуемся процедурой `ShowCursor`. В начале работы вызовем ее с аргументом `False`. Однако с курсором осталась связанной одна проблема, возникающая при нахождении его в области заголовка и в пределах рамки окна, когда пользовательский курсор мы отобразить уже не можем, а системный отключен. Полного решения данной проблемы достичь нелегко, если включать курсор в ловушке сообщения `WM_NCMOUSEMOVE`, возникающего при нахождении курсора за пределами клиентской части окна, то результат может получиться неустойчивым, придется все равно отслеживать возвращение курсора в окно.

Самое простое решение — управлять видимостью курсора в обработчике `OnMouseMove`, включать его при нахождении курсора вблизи границ окна. Но для хорошего функционирования алгоритма надо либо беспрерывно перерисовывать окно, либо добиться более высокой скорости работы с мышью.

Вскользь я уже говорил, что для поверхностей можно принудительно устанавливать одинаковый формат пиксела. Посмотрим на примере проекта каталога Ex28, как это сделать. Здесь введена переменная `PixelFormat` типа `TDDPixelFormat`; после создания первичной поверхности заносим ее формат в данную переменную:

```
ZeroMemory(@PixelFormat, SizeOf(PixelFormat));
PixelFormat.dwSize := SizeOf(PixelFormat);
// Получаем формат пиксела
hRet := FDDSPrimary.GetPixelFormat(PixelFormat);
if Failed (hRet) then ErrorOut(hRet, 'GetPixelFormat');
```

При создании вспомогательной поверхности явно устанавливаем ее формат пиксела:

```
ZeroMemory(@ddsd, SizeOf(ddsd));
with ddsd do begin
```

```
  dwSize := SizeOf(ddsd);
  // Добавился новый флаг
  dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH or DDSD_PIXELFORMAT;
  ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
  ddpfPixelFormat := PixelFormat;   // Устанавливаем формат поверхности
  dwWidth  := ClientWidth;
  dwHeight := ClientHeight;
end;
```

Поверхность, хранящая образ пользовательского курсора, создается теперь явно, для нее также устанавливается формат, совпадающий с форматом первичной поверхности, растр загружается с помощью объекта класса `TBitmap` и копируется на созданную поверхность. Подобный прием мы уже рассматривали в первой главе.

Комбинированные приложения

Данный тип приложений позволяет переключаться между оконным и полноэкранным режимами по ходу работы приложения, которое можно запустить в оконном режиме. Если же оно работает слишком медленно, то пользователь способен переключиться в полноэкранный режим. Оба режима нами были рассмотрены ранее, и вы помните, что для каждого из них установки, определяющие режим работы, необходимо задавать при создании первичной поверхности. Поэтому для комбинированного приложения при переключении режимов следует повторять весь код инициализации заново, одной магической строкой переключение осуществить не удастся. Также при каждом переключении надо повторять все действия деактивации диалога с DirectDraw.

Вот и все тонкости, которые связаны с комбинированными приложениями, можем переходить к иллюстрации — проекту каталога Ex29. Этот пример является моим переложением на Delphi программы из пакета DirectX 7.0 SDK. Работа приложения очень простая: по экрану перемещается одинокий кружок, отскакивающий от границ окна подобно бильярдному шару. Приложение запускается в полноэкранном режиме, но в любой момент работы программы можно переключиться в альтернативный режим, нажав комбинацию клавиш <Alt>+<Enter>, о чем информирует пользователя подсказка, располагающаяся в левом верхнем углу экрана (рис. 3.14). Для упрощения кодирования поведения кружочка окно приложения устанавливаем 640×480 пикселов и не допускаем изменения его размеров:

```
procedure TfrmDD.FormCanResize(Sender: TObject; var NewWidth,
  NewHeight: Integer; var Resize: Boolean);
begin
  Resize := False;    // Запрещаем любые изменения размеров окна
end;
```

Вот почему для этого примера лучше задать размеры области экрана большими, чем принятые по умолчанию.

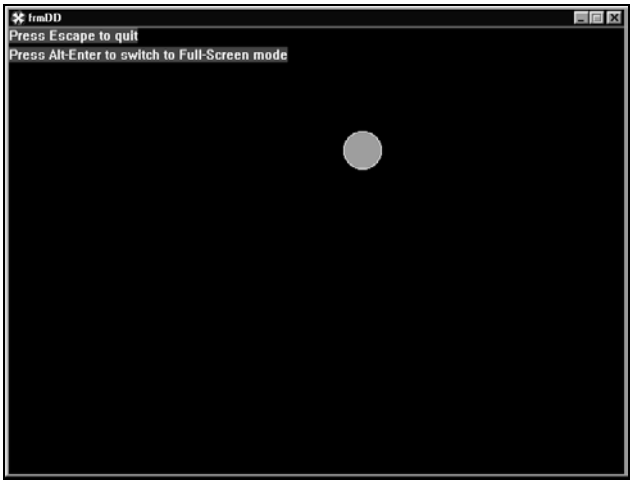


Рис. 3.14. Фрагмент работы комбинированного приложения

Вводимые глобальные переменные связаны с позицией круга на экране, направлением его движения и параметрами области вывода:

```
// Круг рисуется средствами GDI, вписанным в квадрат
x1 : Integer = 0;        // Левый верхний угол квадрата
y1 : Integer = 0;
x2 : Integer = 40;       // Правый нижний угол квадрата
y2 : Integer = 40;
xDir : Integer = 4;      // Текущее приращение координаты X
yDir : Integer = 4;      // Текущее приращение координаты Y
rcScreen    : TRECT;     // Позиция окна, используется для оконного режима
rcViewport  : TRECT;     // Область вывода, 640x480
rcWindow    : TRECT;     // Структура для хранения позиции окна на экране
flgWindowed : BOOL = False;   // Текущий режим работы приложения
```

Код обработчика создания окна будет вызываться при каждом переключении режима:

```
procedure TfrmDD.FormCreate(Sender: TObject);
var
  hRet : HRESULT;
begin
  // Обнуляем все объекты DirectDraw
  FDDClipper := nil;   // Объект отсечения будет удаляться дважды,
  FDDSBack := nil;     // можно этого и не выполнять, но для корректности
```

```
  FDDSPrimary := nil;  // первого вызова FormCreate лучше все-таки сделать
  FDD := nil;
  // В зависимости от режима задаем стиль рамки и видимость курсора
  if flgWindowed then begin
    BorderStyle := bsSizeable;   // Обычный стиль, с областью заголовка
    ShowCursor(True);
  end
  else begin
    BorderStyle := bsNone;       // Без рамки и области заголовка
    ShowCursor(False);
  end;
  // Создается главный объект DirectDraw
  hRet := DirectDrawCreateEx (nil, FDD, IDirectDraw7, nil);
  if Failed(hRet) then ErrorOut(hRet, 'DirectDrawCreateEx');
  // Инициализация поверхностей
  if Failed (InitSurfaces(Handle)) then Close;
  FActive := True;
end;
```

Процедура инициализации поверхностей объединяет в себе оба подхода, изученные нами для полноэкранного и оконного режимов:

```
function TfrmDD.InitSurfaces(Window : THandle) : HRESULT;
var
  hRet : HRESULT;
  ddsd : TDDSURFACEDESC2;
  ddscaps : TDDSCAPS2;
  p : TPoint;
begin
  if flgWindowed then begin      // Оконный режим
    // Получить обычный доступ
    hRet := FDD.SetCooperativeLevel(Window, DDSCL_NORMAL);
    if Failed(hRet) then begin
      Result := hRet;
      ErrorOut(hRet, 'SetCooperativeLevel');
      Exit;
    end;
    // Получаем размеры области вывода и границы экрана
    Windows.GetClientRect(Window, rcViewport);
    Windows.GetClientRect(Window, rcScreen);
    // Находим позицию клиентской области окна на экране
    p.X := rcScreen.Left;
    p.Y := rcScreen.Top;
    Windows.ClientToScreen(Window, p);
    OffsetRect(rcScreen, p.X, p.Y);
```

```
  // Создаем первичную поверхность
  ZeroMemory(@ddsd, SizeOf(ddsd));
  with ddsd do begin
    dwSize := SizeOf(ddsd);
    dwFlags := DDSD_CAPS;
    ddsCaps.dwCaps := DDSCAPS_PRIMARYSURFACE;
  end;
  hRet := FDD.CreateSurface(ddsd, FDDSPrimary, nil);
  if Failed(hRet) then ErrorOut(hRet, 'CreateSurface FAILED');
  // Для оконного приложения создаем объект отсечения
  hRet := FDD.CreateClipper(0, FDDClipper, nil);
  if Failed(hRet) then ErrorOut(hRet, 'CreateClipper FAILED');
  // Ассоциируем отсечение с окном приложения
  FDDClipper.SetHWnd(0, Window);
  FDDSPrimary.SetClipper(FDDClipper);
  FDDClipper := nil;
  // Создаем поверхность заднего буфера, непосредственного вывода
  with ddsd do begin
    dwFlags        := DDSD_WIDTH or DDSD_HEIGHT or DDSD_CAPS;
    dwWidth        := 640;
    dwHeight       := 480;
    ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
  end;
  hRet := FDD.CreateSurface(ddsd, FDDSBack, nil);
  if Failed(hRet) then ErrorOut(hRet, 'CreateSurface2 FAILED');
end
else begin         // Полноэкранный режим
  // Задаем режим исключительного доступа
  hRet := FDD.SetCooperativeLevel(Window, DDSCL_EXCLUSIVE or
                                          DDSCL_FULLSCREEN);
  if Failed(hRet) then ErrorOut(hRet, 'SetCooperativeLevel FAILED');
  // Видеорежим 640x480x8
  hRet := FDD.SetDisplayMode(640, 480, 8, 0, 0);
  if Failed(hRet) then ErrorOut(hRet, 'SetDisplayMode FAILED');
  // Размер области вывода и границ окна, одинаковые значения
  SetRect(rcViewport, 0, 0, 640, 480);
  CopyMemory (@rcScreen, @rcViewport, SizeOf(TRECT));
  // Создаем первичную поверхность с одним задним буфером
  ZeroMemory(@ddsd, SizeOf(ddsd));
  with ddsd do begin
    dwSize := SizeOf(ddsd);
    dwFlags := DDSD_CAPS or DDSD_BACKBUFFERCOUNT;
    ddsCaps.dwCaps := DDSCAPS_PRIMARYSURFACE or DDSCAPS_FLIP or
                      DDSCAPS_COMPLEX;
    dwBackBufferCount := 1;
  end;
```

```
    hRet := FDD.CreateSurface(ddsd, FDDSPrimary, nil);
    if Failed(hRet) then ErrorOut(hRet, 'CreateSurface FAILED');
    ZeroMemory(@ddscaps, SizeOf(ddscaps));
    ddscaps.dwCaps := DDSCAPS_BACKBUFFER;
    hRet := FDDSPrimary.GetAttachedSurface(ddscaps, FDDSBack);
    if Failed(hRet) then ErrorOut(hRet, 'GetAttachedSurface FAILED');
  end;
  Result := DD_OK;
end;
```

Как я уже говорил, код, связанный с созданием объектов, вызывается при каждом переключении режима:

```
procedure TfrmDD.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if (Key = VK_RETURN) and (ssAlt in Shift) then begin   // Переключение
    FActive := False;     // На время переключения запрещаем перерисовку
    flgWindowed := not flgWindowed;             // Меняем значение флага
    FormCreate(nil);      // Удаляем и заново восстанавливаем объекты
  end else
  if (Key = VK_ESCAPE) or (Key = VK_F12) then Close;
end;
```

При перерисовке окна отображаем и перемещаем круг, а затем выводим текст подсказки:

```
function TfrmDD.UpdateFrame : BOOL;
var
  ddbltfx : TDDBLTFX;   // Для очистки фона
  DC : HDC;             // Ссылка на контекст, нужна для функций GDI
  hOldBrush : HBrush;   // Объект фона
  hOldPen : HPen;       // Объект карандаша
begin
  // Очистка окна
  ZeroMemory(@ddbltfx, SizeOf(ddbltfx));
  ddbltfx.dwSize := SizeOf(ddbltfx);
  ddbltfx.dwFillColor := 0;
  FDDSBack.Blt(nil, nil, nil, DDBLT_COLORFILL or DDBLT_WAIT, @ddbltfx);
  // Получение контекста
  if FDDSBack.GetDC(DC) = DD_OK then begin
    // Вывод закрашенного круга
    SetBkColor(DC, RGB(0, 0, 255));      // Синий фон для текста
    SetTextColor(DC, RGB(255, 255, 0));  // Желтый цвет букв
    // Круг закрашивается серым
    hOldBrush := SelectObject(DC, GetStockObject(LTGRAY_BRUSH));
```

```
    // Сам круг — белый
    hOldPen := SelectObject(DC, GetStockObject(WHITE_PEN));
    Ellipse(DC, x1, y1, x2, y2);          // Рисуем круг
    SelectObject(DC, hOldPen);            // Восстанавливаем предыдущие
    SelectObject(DC, hOldBrush);          // параметры рисования
    // Перемещение круга на экране, учитываем границы экрана
    x1 := x1 + xDir;
    x2 := x2 + xDir;
    if x1 < 0 then begin
      x1 := 0;
      x2 := 40;
      xDir := -xDir;   // Меняется направление движения, круг отскакивает
    end;
    if x2 >= 640 then begin
      x1 := 640 — 1 — 40;
      x2 := 640 — 1;
      xDir := -xDir;
    end;
    y1 := y1 + yDir;
    y2 := y2 + yDir;
    if y1 < 0 then begin
      y1 := 0;
      y2 := 40;
      yDir := -yDir;
    end;
    if y2 >= 480 then begin
      y1 := 480 — 1 — 40;
      y2 := 480 — 1;
      yDir := -yDir;
    end;
    // Вывод подсказки
    TextOut(DC, 0, 0, 'Press Escape to quit', 20);
    if flgWindowed
      then TextOut(DC, 0, 20,
          'Press Alt-Enter to switch to Full-Screen mode', 45)
      else TextOut(DC, 0, 20,
          'Press Alt-Enter to switch to Windowed mode', 42);
    FDDSBack.ReleaseDC(DC);
    Result := True;
  end
  else Result := False;     // Поверхность потеряна
end;
```

В обработчике состояния ожидания сообщений переключаем буферы:

```
if FActive then begin
  if UpdateFrame then while TRUE do begin
```

```
    // Оконный режим, переключаем самостоятельно
    if flgWindowed
      then hRet := FDDSPrimary.Blt(@rcScreen, FDDSBack,
                                   @rcViewport, DDBLT_WAIT, nil)
      else
        // Полноэкранный режим, используем метод Flip
        hRet := FDDSPrimary.Flip(nil, 0);
    if hRet = DD_OK then Break;
    if hRet = DDERR_SURFACELOST then begin
      hRet := FDDSPrimary._Restore;
      if Failed(hRet) then Break;
    end;
    if hRet <> DDERR_WASSTILLDRAWING then Break;
  end
  else
  // Воспроизведение не получилось, восстанавливаем поверхность
  FDDSPrimary._Restore; // Для простоты не используем зацикливание
end;
```

Напоминаю, что приложение запускается в полноэкранном режиме. Если вы установите первоначальным оконный режим, то заметите присущий этому примеру недостаток: при первом переключении на полноэкранный режим приложение минимизируется. Эта странность проявляется именно при первом переключении, все остальные протекают без ошибок. Как я сказал, этот пример является переложением программы, написанной на C. В него внесены минимальные изменения по сравнению с первоисточником, но при каждом таком переносе[1] требуются дополнительные усилия для обеспечения полностью корректной работы приложения.

Проект, располагающийся в каталоге Ex30, является переделанным примером оконного приложения с пользовательским курсором в виде руки. Теперь приложение является комбинированным, запускается в оконном режиме.

Для решения проблемы с первым переключением введен специальный флаг, инициируемый тем же значением, что и первый флаг:

```
flgWindowed : BOOL = True;    // Для обоих флагов необходимо задавать
First       : BOOL = True;    // одно и то же первоначальное значение
```

При первой деактивизации полноэкранного приложения окно не минимизируем:

```
procedure TfrmDD.ApplicationEvents1Deactivate(Sender: TObject);
begin
  if flgWindowed
```

[1] Имеется в виду перенос с одного языка программирования на другой язык. — *Ред.*

```
    then begin
      GetWindowRect(Handle, rcWindow);  // Запомнили позицию окна
      if First then First := False;     // Прошла первая минимизация
    end
  else begin
    if First
      then First := False             // Пропускаем первую деактивизацию
      else Application.Minimize;
  end;
end;
```

В остальном код знаком по предыдущим примерам, не стоит, думаю, повторно его рассматривать, обращу внимание только на следующие отличия этого примера:

- поверхность заднего буфера для оконного приложения должна создаваться не в процедуре инициализации, а в обработчике `OnResize` окна, когда известны новые размеры окна;
- чтобы при изменении положения окна в оконном режиме его поверхность не покрывалась серыми пятнами, добавлена ловушка сообщения `WM_MOVE`, в котором определяю новую позицию окна и перерисовываю его;
- обработчики многих событий заканчиваются вызовом процедуры перерисовки окна `UpdateFrame`.

В целом, этот пример можно оценить "на отлично", я не заметил каких-либо отклонений в его работе. Но, конечно, в оконном режиме пользовательский курсор приносит хлопоты при нахождении на границах окна.

Осциллограф

Наверняка многие из читателей планируют использовать DirectDraw в серьезных целях, например, для быстрого отображения диаграмм или графиков.

В этом разделе мы рассмотрим решение подобной задачи несколькими методами и воспользуемся случаем, чтобы узнать еще много нового о DirectDraw. В наших примерах будет моделироваться осциллограф, показания которого представляют собой бегущую синусоиду.

Начнем с проекта каталога Ex31, в нем отдельные точки синусоиды ставятся с использованием метода `Blt` поверхности, подобно одному из примеров на построение окружностей. Ничего особо нового нет, за исключением того, что для точного задания цвета точки используется пользовательская функция `CreateRGB`, осуществляющая перевод тройки цветов в значение, соответствующее схеме 5–6–5.

Перед изучением следующего примера, проекта каталога Ex32, вы должны утроить внимание. Он иллюстрирует новый для нас способ непосредствен-

ного обращения к памяти поверхности. Новизна состоит в том, что мы не применяем запирание памяти, но такое можно производить корректно только с поверхностями, размещенными в системной памяти.

Итак, смотрим внимательно пример. Режим 640×480×8, для работы с пикселами поверхности заведен массив буфера кадра вспомогательной поверхности:

```
FrameBuffer : Array [0..99, 0..99] of Byte;
```

Поверхность, как видим, будет размером 100×100 пикселов, внимательно посмотрите, как она создается. Сами задаем значение `lPitch` и адрес содержимого буфера кадра:

```
ZeroMemory(@ddsd, SizeOf(ddsd));
with ddsd do begin
  dwSize := SizeOf(ddsd);
  dwFlags := DDSD_WIDTH or DDSD_HEIGHT or DDSD_LPSURFACE or
             DDSD_CAPS or DDSD_PITCH;   // Новые флаги!
  // Поверхность создается в СИСТЕМНОЙ памяти
  ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN or DDSCAPS_SYSTEMMEMORY;
  dwWidth := 100;
  dwHeight := 100;
  lpSurface := @FrameBuffer;   // Адрес поверхности равен адресу массива
  lPitch  := LongInt(100);     // Адрес поверхности равен ширине массива
end;
hRet := FDD.CreateSurface(ddsd, FDDSWork, nil);
if Failed(hRet) then ErrorOut(hRet, 'Create Surface');
// Цветовой ключ для вспомогательной поверхности
hRet := DDSetColorKey (FDDSWork, RGB(0, 0, 0));
if Failed (hRet) then ErrorOut(hRet, 'DDSetColorKey');
```

При воспроизведении кадра работаем непосредственно с элементами вспомогательного массива:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  i : Integer;
  hRet : HRESULT;
begin
  ThisTickCount := GetTickCount;
  if ThisTickCount - LastTickCount > 10 then begin
    Angle := Angle + 0.05;        // Сдвиг синусоиды
    if Angle > 2 * Pi then Angle := Angle - 2 * Pi;
    LastTickCount := GetTickCount;
  end;
  // Воспроизводим картинку фона
  hRet := FDDSBack.BltFast (0, 0, FDDSBackGround, nil, DDBLTFAST_WAIT);
```

```
  if Failed(hRet) then begin
    hRet := RestoreAll;
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
  end;
  // Обнуляем элементы массива
  ZeroMemory (@FrameBuffer, SizeOf (FrameBuffer));
  // Заполняем массив для получения синусоиды
  for i := 0 to 99 do
    FrameBuffer [50 - trunc (sin (Angle + i * 2 * Pi / 100) * 25), i] :=
                120;
  // Воспроизводим поверхность синусоиды
  hRet := FDDSBack.BltFast (0, 0, FDDSWork, nil,
          DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
  if Failed(hRet) then begin
    hRet := RestoreAll;
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
  end;
  Result := DD_OK;
end;
```

Пример действительно важен, показывает, как отображать данные, размещенные в системной памяти. В некоторых случаях, например при сложных вычислениях с матрицами, такой подход может облегчить решение задачи.

Проект каталога Ex33 принципиально ничем не отличается от предыдущего, только используется 16-битный режим, а синусоида выводится на весь экран. Здесь вам надо обратить внимание на изменения в описании массива:

```
FrameBuffer : Array [0..479, 0..639] of WORD;
```

Значение `lPitch` для 16-битной поверхности задаем 640×2 пикселов, как ширина поверхности, умноженная на размер одной ячейки. Синусоида располагается на всем экране, и поверхность фона теперь отсутствует. Для простоты подготовки синусоиду рисуем синим цветом:

```
// Очистка фона, она же — очистка экрана
ZeroMemory (@FrameBuffer, SizeOf (FrameBuffer));
for i := 0 to 639 do
FrameBuffer [240 - trunc (sin (Angle + i * 2 * Pi / 640) * 100), i] :=
            255;   // Для синего цвета достаточно поместить в ячейку 255
Result := FDDSBack.BltFast (0, 0, FDDSWork, nil, DDBLTFAST_WAIT);
```

Закончим самым тривиальным способом построения синусоиды, основанным на блиттинге (проект каталога Ex34). Важен этот простой пример тем, что иллюстрирует существование образов в таких количествах, сколько нам необходимо. Подобным многократным блиттингом мы активно будем пользоваться в следующей главе.

Отдельный образ загружается из растра, при воспроизведении кадра он копируется на экране 640 раз:

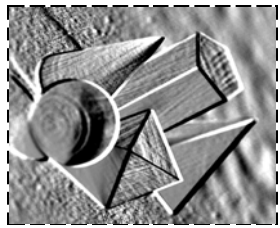
```
for i := 0 to 639 do begin
  hRet := FDDSBack.BltFast (i, 240 -
                            trunc (sin (Angle + i * 2 * Pi / 640) * 100),
          FDDSImage, nil, DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
end;
```

Что вы узнали в этой главе

Мы выяснили, что для задания прозрачности участков поверхности используется механизм цветового ключа.

Также познакомились с принципами построения анимации и приемами, применяемыми для создания визуальных эффектов. Один из важнейших механизмов, используемых для работы с содержимым поверхности, заключается в непосредственном доступе к ее пикселам. Осуществляется такой доступ для поверхностей, размещаемых в видеопамяти путем реализации механизма запирания поверхности. К тому же подобные поверхности требуют особого внимания в ситуациях восстановления минимизированного приложения.

ГЛАВА 4



Спрайтовая анимация

Данная глава содержит вводный материал по созданию анимации на основе спрайтов и на примерах демонстрирует используемые приемы ее программирования. Как правило, именно такие приемы используются при моделировании игр и схожих с ними приложений.

Примеры располагаются в каталоге \Examples\Chapter04.

Спрайты

В большинстве предыдущих примеров на экране присутствовал одинокий образ, вид которого не менялся с течением времени. Теперь нам предстоит узнать, как создавать движущиеся образы, меняющиеся со временем или в зависимости от обстоятельств. Также попутно нам предстоит узнать еще много нового о DirectDraw.

Разработчики восьмой версии этой замечательной библиотеки позаботились о программистах, переработав модуль `DDUtil` и предоставив в наше распоряжение объектно-ориентированную оболочку для использования DirectDraw. Код приложений выглядит удобочитаемым и легко воспринимаемым. Ваш покорный слуга перенес этот модуль на Delphi (назвав `DDUtil8`), и мы сможем воспользоваться удобными нововведениями. Однако в рассматриваемых до сих пор примерах эта библиотека не использовалась и во многих последующих примерах также не будет применяться.

Во-первых, использование такой, как и любой другой объектно-ориентированной библиотеки в Delphi приводит к значительным накладным расходам, потерям драгоценного времени, поэтому указанные библиотеки лучше включать лишь в простые примеры.

Во-вторых, подобные библиотеки не могут вместить в себя все возможности DirectDraw, программист не в состоянии только с их помощью реализовать все свои идеи и ему все равно потребуются знания более низкого уровня.

В-третьих, если опираться только на готовые библиотеки, теряется чувство понимания собственных действий, а вынужденное использование механизмов, не включенных в библиотеку, выглядит чуждым и неестественным. В принципе, такое возникает очень часто при программировании в среде Delphi, например создание ловушек сообщений для новичка выглядит вычурным и сложным.

Я надеюсь, что ознакомление с предыдущим материалом книги прошло для вас без проблем, и вы теперь можете свободно ориентироваться в этих программах с длинным и громоздким кодом. Если это так, вы будете легко разбираться и в чужих программах, написанных на другом языке. Вы можете встретить массу примеров по использованию DirectX в книгах, ориентированных на C-программиста в DirectX SDK или Сети. Код таких примеров вами должен легко пониматься, поскольку код наших предыдущих примеров был к нему очень близок.

Мы могли бы теперь и не отвлекаться на изучение нового для сегодняшнего дня подхода, но, поскольку такой подход предлагается разработчиками, то он фактически узаконивается в качестве стандарта, и, со временем, вам будут все чаще и чаще встречаться программы, построенные именно на подобном подходе. Нам надо обязательно познакомиться с ним, чтобы вы не чувствовали себя в такой ситуации неуютно.

Код теперь выглядит проще, но я подчеркну, что ваши программы только выиграют, если вы будете создавать их так, как мы делали это в предыдущих примерах.

Ну что же, после такого вступления можно переходить к рассмотрению первого примера, проекта каталога Ex01. Выглядит его работа несложной: по экрану времени перемещаются образы логотипа DirectX, отскакивая от стенок. Пример является моей трансляцией одного из примеров, входящих в DirectX 8.0 SDK, я внес в код минимум изменений по сравнению с первоисточником.

Поведение спрайтов не будем подробно рассматривать, проанализируем только то, что связано непосредственно с DirectDraw.

В коде отсутствуют многие знакомые нам типы, вместо них появились новые:

```
g_pDisplay     : CDisplay;   // Главный объект
g_pLogoSurface : CSurface;   // Поверхность образа
g_pTextSurface : CSurface;   // Поверхность текста
```

Я долго думал, изменять ли префикс таких типов на префикс "T", принятый для Delphi, и решил оставить все-таки его таким же, как и в первоисточнике.

Главный объект инкапсулирует методы и свойства, связанные с созданием и управлением поверхностями. Объекты присоединяемых к главному объекту поверхностей можно создавать пустыми, либо по содержимому растра, либо путем вывода текста:

```
g_pDisplay := CDisplay.Create;    // Создание главного объекта
// Метод создания полноэкранного дисплея
hr := g_pDisplay.CreateFullScreenDisplay(Handle, ScreenWidth,
                                         ScreenHeight, ScreenBitDepth);
// Анализ успешности действия
if FAILED(hr) then ErrorOut (hr, 'This display card does
                             not support 640x480x8.');
// Создание внеэкранной поверхности спрайта
hr := g_pDisplay.CreateSurfaceFromBitmap(g_pLogoSurface, imageBmp,
                                    SPRITE_DIAMETER, SPRITE_DIAMETER);
if(FAILED(hr)) then ErrorOut (hr, 'CreateSurfaceFromBitmap');
// Создание внеэкранной поверхности с текстом
hr := g_pDisplay.CreateSurfaceFromText(g_pTextSurface, Font.Handle,
                             HELPTEXT, RGB(0,0,0), RGB(255, 255, 0));
if(FAILED(hr)) then ErrorOut (hr, 'CreateSurfaceFromText');
// Метод поверхности для установки цветового ключа
hr := g_pLogoSurface.SetColorKey(0);     // Ключ — черный цвет
if(FAILED(hr)) then ErrorOut (hr, 'SetColorKey');
```

Как я и обещал, код действительно упростился, нет ни слова о первичной поверхности и вообще о буферах. Вся эта черновая работа скрыта от глаз разработчика. Воспроизведение также значительно упростилось. Основано оно целиком на использовании методов главного объекта:

```
for iSprite := 0 to NUM_SPRITES — 1 do     // Цикл вывода спрайтов
  g_pDisplay.ColorKeyBlt(g_Sprite[iSprite].fPosX,
    g_Sprite[iSprite].fPosY, g_pLogoSurface.GetDDrawSurface, nil);
// Вывод текста подсказки
g_pDisplay.Blt(10, 10, g_pTextSurface, nil);
// Завершение работы. Выполняем переключение поверхностей
Result := g_pDisplay.Present;
```

Выглядит код немного непривычно, но радует своей лаконичностью. Надеюсь, вы сейчас испытываете светлое чувство ясности понимания того, что скрыто за этой краткостью кода.

Самостоятельно разберите, как выглядит код восстановления потерянных поверхностей.

Вам не стоит обижаться на меня, что мы не начали сразу же писать подобный код, потому что, напоминаю, нам все равно не удастся уберечься от углубления в дебри, стоит только попробовать решить мало-мальски сложные задачи.

Вот первый пример такой задачи, проект каталога Ex02 — развитие предыдущего: те же мечущиеся логотипы, но желтоватые кресты их со временем меняют цвет. Пример также является моим переложением учебной программы из SDK.

Используется палитровый режим, поэтому добавилась переменная знакомого нам типа IDIRECTDRAWPALETTE. Для загрузки ее задействованы соответствующие методы главного объекта:

```
// Загружаем палитру из растра
hr := g_pDisplay.CreatePaletteFromBitmap(g_pDDPal, imageBmp);
if FAILED(hr) then ErrorOut (hr, 'CreatePaletteFromBitmap');
// Задаем палитру для экрана
hr := g_pDisplay.SetPalette(g_pDDPal);
if FAILED(hr) then ErrorOut (hr, 'SetPalette');
```

Для смены оттенков, образующих палитры, используются все знакомые нам действия: сохранение составляющих во вспомогательном массиве и циклическое изменение цветовых весов некоторых из них. В примере циклически модифицируются элементы палитры, отвечающие за оттенки желтого цвета. Все эти действия никак с появлением нового подхода не упростились, и этот код заметно выделяется среди прочего.

Обратите внимание, что разработчики рекомендуют синхронизировать обновление палитры с вертикальной разверткой экрана, чем я пренебрег в своем первом примере на тему цветовой анимации. Действие это аналогично использованию флага WAIT при блиттинге и запирании поверхности, но записывается вот так:

```
hr:=g_pDisplay.GetDirectDraw.WaitForVerticalBlank(DDWAITVB_BLOCKBEGIN,0);
if(FAILED(hr)) then ErrorOut (hr, 'WaitForVerticalBlank');
```

Результат будет равен константе E_NOTIMPL, если такая синхронизация аппаратно не поддерживается. Карты с отсутствием данной поддержки сейчас редко встречаются, но вот аппаратная поддержка следующего рассматриваемого нами приема на "устаревших" картах может и отсутствовать.

Гамма-контроль используется для обеспечения цветовых переходов в непалитровых режимах и управляет яркостью изображения. Примером создания fade-эффекта в 32-битном режиме является проект каталога Ex03. Здесь появилась переменная специального типа, связанного с гамма-контролем:

```
g_pGammaControl: IDIRECTDRAWGAMMACONTROL;
```

Поскольку не каждая видеокарта поддерживает эту возможность, необходимо определиться, возможна ли корректная работа приложения:

```
function TfrmDD.HasGammaSupport : BOOL;
var
  ddcaps : TDDCAPS;   // Структура описания возможностей драйвера
begin
  ZeroMemory(@ddcaps, sizeof(ddcaps));
  ddcaps.dwSize := sizeof(ddcaps);
  // Получаем список возможностей
  g_pDisplay.GetDirectDraw.GetCaps(@ddcaps, nil);
```

```
  // Поддерживается ли гамма-контроль аппаратно?
  if(ddcaps.dwCaps2 and DDCAPS2_PRIMARYGAMMA) <> 0
    then Result := TRUE
    else Result := FALSE;
end;
```

В этом примере при отсутствии аппаратной поддержки приложение завершает работу. В принципе этого можно не делать. Не должно возникать исключений в работе приложений при отсутствии аппаратной поддержки такой возможности. Просто на экране по ходу работы не будет заметно никаких изменений.

В примере на экране рисуются три красивые полосы, образованные плавным переходом черного цвета в каждый из тройки чистых цветов. Как это осуществляется, разберите самостоятельно. Чтобы полосы равномерно заполняли экран при любых установках, необходимо получить данные о формате пиксела, для чего предназначен метод поверхности (объекта типа `CSurface`) `GetBitMaskInfo`.

Проект, располагающийся в каталоге Ex04, отличается от предыдущего тем, что вместо полос на экран выводится система мечущихся логотипов.

Для задания текущей яркости служит целочисленная переменная:

```
g_lGammaRamp : LongInt = 256;
```

Работа по осуществлению гамма-контроля очень похожа на работу с палитрой:

```
function TfrmDD.UpdateGammaRamp : HRESULT;
var
  hr : HRESULT;
  ddgr : TDDGAMMARAMP;     // Набор значений яркости чистого цвета
  dwGamma : WORD;
  iColor : Integer;
begin
  ZeroMemory(@ddgr, sizeof(ddgr));
  // Получаем текущие значения яркостей
  hr := g_pGammaControl.GetGammaRamp(0, ddgr);
  if(FAILED(hr)) then begin
    Result := hr;
    Exit
  end;
  dwGamma := 0;
  // Последовательно наращиваем яркость цветовых составляющих
  for iColor := 0 to 255 do begin
    ddgr.red[iColor]   := dwGamma;
    ddgr.green[iColor] := dwGamma;
    ddgr.blue[iColor]  := dwGamma;
    dwGamma := dwGamma + g_lGammaRamp;
  end;
```

```
  // Устанавливаем текущую "палитру"
  hr := g_pGammaControl.SetGammaRamp(0, ddgr);
  if(FAILED(hr)) then begin
    Result := hr;
    Exit
  end;
  Result := S_OK;
end;
```

Привожу еще один вариант использования модуля `DDUtil8` (проект каталога Ex05) — иллюстрацию непосредственной работы с пикселами поверхности. Здесь таким способом подготавливаются поверхности спрайтов. Пример рассчитан на работу в 16-битном режиме и использует указатели `PWORD`. Принципиально ничего нового в коде не появилось. Библиотека DDUtil8 ничего не изменила в этой части, поэтому не стану подробно разбирать код. Только обращаю ваше внимание на то, что этот пример, подобно предыдущему, корректно работает с любым форматом пиксела, поскольку опирается на присутствующие битовые маски.

Надеюсь, такого беглого знакомства с библиотекой DDUtil8 для вас оказалось достаточным для того, чтобы получить представление о ней.

Вернемся к обычному для этой книги подходу, лишенному объектной ориентированности. Рассмотрим следующий пример — проект, располагающийся в каталоге Ex06. Пример можно отнести к разряду классических, он является моей интерпретацией программы stretch.cpp из DirectX 6.0 SDK. Это оконное приложение, на экране выводится образ вращающегося в пространстве тора (рис. 4.1).

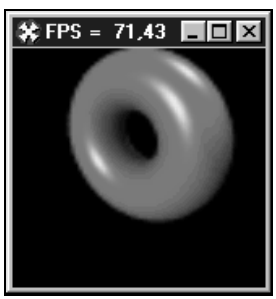


Рис. 4.1. Момент работы классического примера на тему меняющегося образа

Мультфильм намеренно подготовлен таким образом, чтобы создать у зрителя иллюзию трехмерной графики. На самом деле последовательно выводятся отдельные кадры с изображением различных фаз поворота тора.

Размер отдельного кадра 64×64 пиксела. Все кадры записаны в одно растровое изображение donut.bmp в шесть рядов по десять кадров. Растр загружа-

ется на поверхность FDDSImage. При перерисовке экрана на первичную поверхность выводится прямоугольник очередной фазы поворота тора:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  rcRect : TRECT;
begin
  Inc (Frames);
  ThisTickCount := GetTickCount;
  if ThisTickCount - LastTickCount > TimeDelay then begin
    FPS := PChar ('FPS = ' + Format('%6.2f',
                 [Frames * 1000 / (ThisTickCount - LastTickCount)]));
    Caption := FPS;
    Frames := 0;
    // Наращиваем текущий кадр; всего 60 кадров
    CurrentFrame := (CurrentFrame + 1) mod 61;
    // Прямоугольник очередного кадра; "shl 6" равносильно " * 64"
    SetRect (rcRect,
            (CurrentFrame mod 10) shl 6,
            (CurrentFrame div 10) shl 6,
            (CurrentFrame mod 10 + 1) shl 6,
            (CurrentFrame div 10 + 1) shl 6);
    LastTickCount := GetTickCount;
  end;
  // Вывод кадра на первичную поверхность
  Result := FDDSPrimary.Blt(@rcDest, FDDSImage, @rcRect,
                            DDBLT_WAIT, nil);
end;
```

Принципиально этот пример отличается от примеров предыдущей главы только тем, что при каждой перерисовке кадра выводится не вся вторичная поверхность, а только ее фрагмент.

Для оптимизации код, связанный с определением позиции окна вывода на экране, я перенес в обработчик OnResize формы. Для корректного поведения при перемещении окна этот же код вызывается в ловушке соответствующего сообщения WM_MOVE:

```
procedure TfrmDD.WindowMove (var Msg: TWMMove);    // Перемещение окна
begin
  FormResize (nil);   // Определение нового положения окна
end;
procedure TfrmDD.FormResize(Sender: TObject);
var
  p : TPoint;
begin
  p.X := 0;
  p.Y := 0;
```

```
  Windows.ClientToScreen(Handle, p);
  Windows.GetClientRect(Handle, rcDest);
  OffsetRect(rcDest, p.X, p.Y);
end;
```

Вдогонку рассмотренному примеру привожу проект каталога Ex07, идею которого я также позаимствовал из набора примеров SDK. В данном примере эмулируется наложение выводимого тора с содержимым рабочего стола, наподобие одной из программ предыдущей главы. Пример не очень хорош и предложен скорее "для массовости". Здесь содержимое экрана копируется только один раз, при запуске приложения. Поэтому при изменении подлинного фона тора возникает ощущение некорректности работы приложения. Если же копировать подложку при каждом обновлении фазы поворота тора, то вместе с фоном копируется изображение самого тора, оставшееся с предыдущего вывода. Попробуйте развить этот пример. Из него может получиться занятная программа, если изменения во всех кадрах будут находиться только в пределах первоначального силуэта.

Хранитель экрана

В этом разделе мы начнем работу по созданию оригинального хранителя экрана, попробуем реализовать наши полученные знания в проекте, имеющем не только познавательную ценность. Также мы решим для себя вопрос, возможно ли в принципе создать на Delphi приложение, использующее DirectDraw со множеством образов, выполняющееся с удовлетворительной скоростью.

Работа нашего хранителя экрана будет заключаться в том, что по экрану станут проплывать рыбки и полупрозрачные пузырьки воздуха (рис. 4.2).

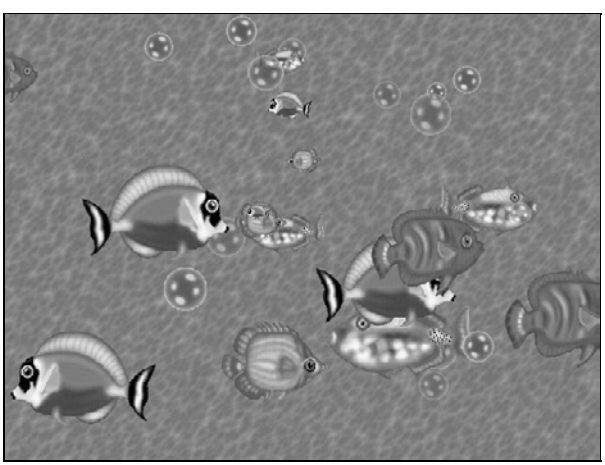


Рис. 4.2. Момент работы нашего хранителя экрана

Образы рыбок для этой программы любезно предоставлены Kelly Villoch. Рыбки будут сновать в разные стороны с различной скоростью, а пузырьки подниматься плавно вверх и лопаться, достигнув верхнего края экрана.

Особенно интересным для этого примера является использование в качестве фона "тайлового", зацикленного образа. Фон окна состоит из нескольких повторяющихся одинаковых фрагментов, состыкованных друг с другом. Так получается бесконечный образ фона.

Чтобы получить простейший хранитель экрана, достаточно переименовать исполняемый модуль любого полноэкранного приложения в файл с расширением scr и поместить его в папку System системного каталога (как правило, это C:\Windows\System\).

Но, чтобы хранитель экрана не выбивался из ряда других "скринсэйверов", необходимо хорошенько потрудиться. Так, следует обеспечить его работу в небольшом окне при предварительном просмотре, когда пользователь выбирает вкладку **Заставка** при задании свойств экрана (рис. 4.3).

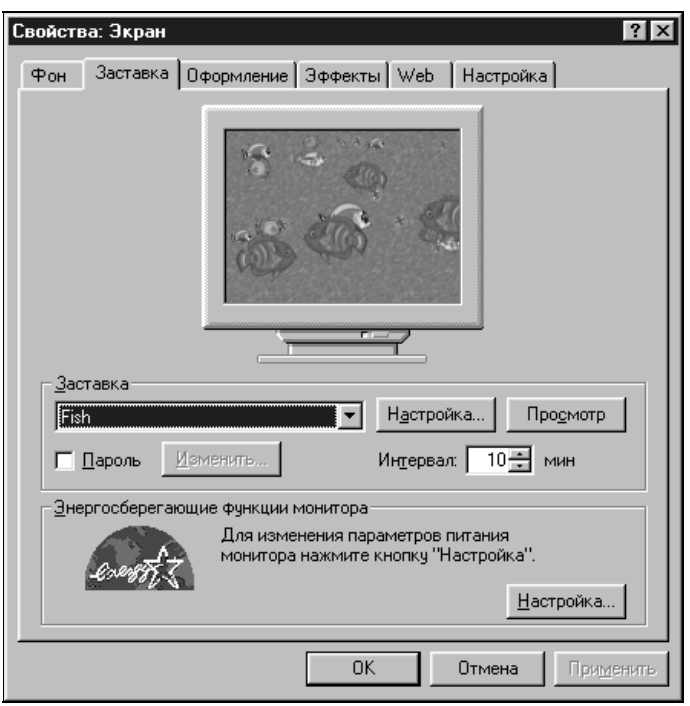


Рис. 4.3. Хранитель экрана обязан работать и в оконном режиме, и в окне предварительного просмотра

Итак, у хранителя экрана должны быть оконный и полноэкранный режимы работы. Причем, совсем не нужно делать приложение комбинированным,

поскольку переключения между режимами по ходу работы происходить не будет. Просто надо обеспечить работу в одном из этих режимов. Требуемый режим устанавливается один раз, в начале работы, и будет действительным до самого конца его выполнения.

Далее, необходимо снабдить приложение диалоговыми окнами настройки параметров работы и задания пароля для выхода. И еще, на тот случай, если указан пароль, следует предусмотреть режим просмотра, при выходе из которого пароль не вводится и комбинация клавиш <Ctrl>+<Alt>+<Del> при работе хранителя в этом режиме не блокируется.

Также нам надо обеспечить наличие единственного модуля. Все необходимые образы не должны загружаться из других файлов, что будет совершенно неудобно для пользователя.

В этом разделе мы только начнем работу над приложением, затем упростим нашу работу, немного ограничим функциональность хранителя экрана и не будем принимать в расчет ничего, что связанно с паролем.

Рассмотрение готовой работы, проекта каталога Ex08, начнем с разбора его сердцевины — механизма визуализации жизни подводного мира.

Для хранения образов использую компоненты класса `TImage`, располагающиеся на форме. Хранитель экрана разместится в единственном файле.

Мне потребовались четыре образа рыбок, один образ всплывающего пузырька воздуха и образ для построения фона (рис. 4.4).

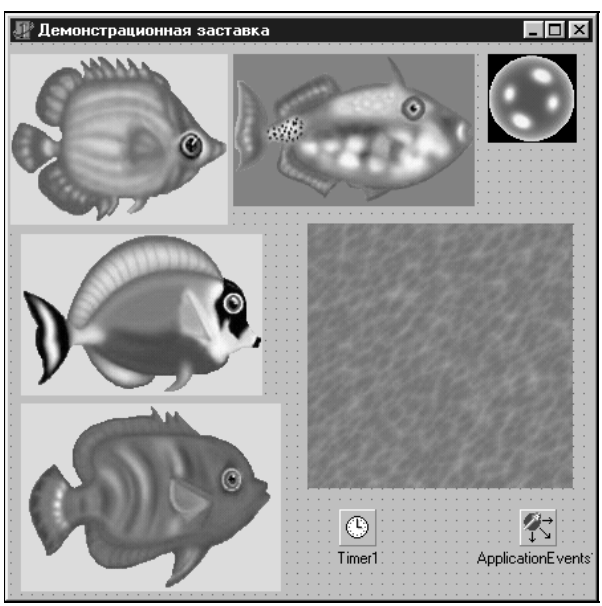


Рис. 4.4. Все образы хранятся в компонентах класса `TImage`, чтобы не загружать их из отдельных файлов

Для загрузки на поверхность образа из компонента класса `TImage` введена пользовательская функция `CreateFromImage`. Рыбки и пузырьки на экране будут иметь случайные размеры, чтобы при использовании цветового ключа не появлялась окантовка. Для корректного масштабирования приходится применять длинную процедуру перекладывания образа на вспомогательные объекты класса `TBitMap`:

```
function TfrmDD.CreateFromImage (var FDDS : IDirectDrawSurface7;
  const Image : TImage; const imgWidth, imgHeight : Integer) : HRESULT;
var
  DC : HDC;
  ddsd : TDDSurfaceDesc2;
  hRet : HResult;
  wrkBitmap1 : TBitMap;
  wrkBitmap2 : TBitMap;
begin
  ZeroMemory (@ddsd, SizeOf(ddsd));
  with ddsd do begin
    dwSize := SizeOf(ddsd);
    dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
    dwWidth := imgWidth;
    dwHeight := imgHeight;
    ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
  end;
  // Создаем поверхность нужных размеров
  hRet := FDD.CreateSurface(ddsd, FDDS, nil);
  if Failed(hRet) then ErrorOut(hRet, 'Create Surface');
  // Первое изображение хранит растр,
  // переложенный с компонента класса TImage
  wrkBitmap1 := TBitMap.Create;
  wrkBitmap1.Width := Image.Width;
  wrkBitmap1.Height := Image.Height;
  // Копирование растра, StretchBlt исказит образ
  BitBlt(wrkBitmap1.Canvas.Handle, 0, 0, wrkBitmap1.Width,
         wrkBitmap1.Height, Image.Canvas.Handle, 0, 0, SRCCOPY);
  // Второе изображение используется для корректного масштабирования
  wrkBitmap2 := TBitMap.Create;
  wrkBitmap2.Width := imgWidth;
  wrkBitmap2.Height := imgHeight;
  // Перекладываем растр во второй битмап
  wrkBitmap2.Canvas.StretchDraw (Rect (0, 0, imgWidth, imgHeight),
                                 wrkBitmap1);
  // Воспроизводим масштабированный растр на сформированной поверхности
  if FDDS.GetDC(DC) = DD_OK then begin
     BitBlt(DC, 0, 0, imgWidth, imgHeight,
         wrkBitmap2.Canvas.Handle, 0, 0, SRCCOPY);
```

```
    FDDS.ReleaseDC(DC);
  end;
  wrkBitmap1.Free;
  wrkBitmap2.Free;
  // Задаем ключ, берем цвет первого пиксела
  Result := DDSetColorKey (FDDS, Image.Canvas.Pixels [0, 0]);
end;
```

Класс `TFish` инкапсулирует свойства и методы наших рыбок:

```
TFish = class
  XFish, YFish   : Integer;     // Позиция на экране
  Direction      : 0..1;        // Направление движения
  WidthFish      : Integer;     // Ширина
  HeightFish     : Integer;     // Высота
  FDDSFish       : IDirectDrawSurface7; // Поверхность с образом
  SpeedFish      : Integer;     // Скорость движения
  procedure Init;               // Инициализация
  procedure Render;             // Воспроизведение
end;
```

При инициализации очередной рыбки ее размеры задаются случайно, чтобы создать иллюзию пространства, как будто некоторые рыбки удалены дальше от глаза наблюдателя. Но при указании размеров я соблюдаю пропорции первоначальной картинки. Чтобы рыбка могла плавать и слева направо, и справа налево, можно заготовить два образа на каждую рыбку. Но в этом случае размер файла хранителя экрана резко увеличится. Я выбрал другой путь: имеется одна картинка каждого вида рыбок, плывущих слева направо, а при необходимости содержимое поверхности зеркально переворачивается.

Размер исполнимого файла при таком подходе не увеличивается, но мы, конечно, при каждой инициализации рыбки теряем немного во времени:

```
procedure TFish.Init;
procedure Rotate;     // Зеркальный поворот поверхности рыбки
var
  desc : TDDSURFACEDESC2;
  i, j : Integer;
  wrkW : Word;
begin
  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf(desc);
  if Failed (FDDSFish.Lock (nil, desc, DDLOCK_WAIT, 0)) then Exit;
  for i := 0 to (WidthFish - 1) div 2 do    // Цикл по столбцам растра
    for j := 0 to HeightFish - 1 do begin   // Цикл по строкам растра
      wrkW := PWord (Integer (desc.lpSurface) + j * desc.lPitch +
                     i * 2)^;    // Переставляем пикселы растра
```

```
      PWord (Integer (desc.lpSurface) + j * desc.lPitch + i * 2)^ :=
      PWord (Integer (desc.lpSurface) + j * desc.lPitch +
            (WidthFish - 1 - i) * 2)^;
      PWord (Integer (desc.lpSurface) + j * desc.lPitch +
            (WidthFish - 1 - i) * 2)^ := wrkW;
  end;
  FDDSFish.Unlock (nil);
end;
begin
  case random (4) of    // Случайный выбор одного из четырех видов рыбок
    0 : begin
        WidthFish := random (141) + 24;
        HeightFish := WidthFish * 129 div 164;   // Сохранение пропорций
        if Failed (frmDD.CreateFromImage (FDDSFish, frmDD.imgFish1,
                   WidthFish, HeightFish))
                     then frmDD.ErrorOut(DD_FALSE, 'CreateFish');
        end;
    1 : begin
        WidthFish := random (161) + 22;
        HeightFish := WidthFish * 115 div 182;
        if Failed (frmDD.CreateFromImage (FDDSFish, frmDD.imgFish2,
                   WidthFish, HeightFish))
                     then frmDD.ErrorOut(DD_FALSE, 'CreateFish');
        end;
    2 : begin
        WidthFish := random (161) + 22;
        HeightFish := WidthFish * 122 div 182;
        if Failed (frmDD.CreateFromImage (FDDSFish, frmDD.imgFish3,
                   WidthFish, HeightFish))
                     then frmDD.ErrorOut(DD_FALSE, 'CreateFish');
        end;
    3 : begin
        WidthFish := random (175) + 22;
        HeightFish := WidthFish * 142 div 182;
        if Failed (frmDD.CreateFromImage (FDDSFish, frmDD.imgFish4,
                   WidthFish, HeightFish))
                     then frmDD.ErrorOut(DD_FALSE, 'CreateFish');
        end;
  end;
  Direction := random (2);       // Направление движения случайно
  SpeedFish := random (6) + 1;  // Следим, чтобы скорость была ненулевой
  if Direction = 0               // Плывет слева направо, значит,
                                 // должна появиться слева экрана
    then XFish := -WidthFish
```

```
    else begin
    XFish := ScreenWidth;       // Должна появиться справа экрана
    Rotate;                     // Требуется зеркальный поворот картинки
  end;
  YFish := random (360) + 5;    // Глубина, на которой поплывет рыбка
end;
```

При отображении проплывающей рыбки надо отдельно обрабатывать ситуацию вблизи границ, когда выводится только часть образа:

```
procedure TFish.Render;
var
  wrkRect : TRect;
begin
  case Direction of
    0 : begin
        XFish := XFish + SpeedFish;         // Рыбка плывет вправо
        if XFish > ScreenWidth then Init;   // Уплыла за границы экрана
        end;
    1 : begin
        XFish := XFish - SpeedFish;         // Рыбка плывет влево
        if XFish < -WidthFish then Init;
        end;
  end;
  if XFish <= 0 then begin
  SetRect (wrkRect, -XFish, 0, WidthFish, HeightFish);
  frmDD.FDDSBack.BltFast (0, YFish, FDDSFish,
            @wrkRect, DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
  end
  else begin
  // На экране помещается вся картинка целиком
  if XFish <= ScreenWidth - WidthFish then begin
  frmDD.FDDSBack.BltFast (XFish, YFish, FDDSFish,
            nil, DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
  end
  else begin
  SetRect (wrkRect, 0, 0, ScreenWidth - XFish, HeightFish);
  frmDD.FDDSBack.BltFast (XFish, YFish, FDDSFish,
            @wrkRect, DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
  end;
  end;
end;
```

Для описания пузырьков воздуха также используется концепция ООП:

```
TBubble = class
  X, Y : Integer;       // Позиция пузырька на экране
  Length : Integer;     // Образы квадратные, достаточно одной величины
```

```
  FDDSBubble : IDirectDrawSurface7;
  SpeedBubble : Integer;
  Pict : Array of Array of Word; // Массив образа, для полупрозрачности
  Alpha : Integer;               // Степень прозрачности пузырька
  procedure Init;                // Инициализация пузырька
  procedure Render;              // Воспроизведение
end;
```

Инициализацию пузырька можно упростить. Его поверхность используется только для заполнения массива `Pict`:

```
procedure TBubble.Init;
var
  desc : TDDSURFACEDESC2;
  i, j : Integer;
begin
  Length := random (30) + 20;
  if Failed (frmDD.CreateFromImage (FDDSBubble, frmDD.imgSphere,
             Length, Length))
               then frmDD.ErrorOut(DD_FALSE, 'Create Bubble');
  SetLength(Pict, Length);     // Задаем размер динамического массива
  for i := 0 to Length - 1 do
      SetLength(Pict [i], Length);
  ZeroMemory (@desc, SizeOf(desc));
  desc.dwSize := SizeOf(desc);
  if Failed (FDDSBubble.Lock (nil, desc, DDLOCK_WAIT, 0)) then Exit;
  for i := 0 to Length - 1 do     // Заполняем массив
    for j := 0 to Length - 1 do   // масштабированным образом
      Pict [i, j] := PWord (Integer (desc.lpSurface) +
                            j * desc.lPitch + i * 2)^;
  FDDSBubble.Unlock (nil);
  // Поверхность больше не нужна, будет использоваться массив Pict
  FDDSBubble := nil;
  Alpha := random (150) + 50;     // Степень прозрачности
  SpeedBubble := random (3) + 1;  // Скорость, ненулевая
  X := random (550) + Length;
  Y := ScreenHeight - Length;     // Появится внизу экрана
end;
```

Механизм обеспечения полупрозрачности поверхности вы должны хорошо помнить по предыдущим примерам, повторно разбирать его не будем. Код рисования пузырька за вычетом процедуры `Blend` выглядит совсем коротким:

```
procedure TBubble.Render;
begin
  Y := Y - SpeedBubble;    // Перемещение пузырька
```

```
  if Y < 0 then Init;       // Всплыл на поверхность
  Blend (X, Y);             // Собственно вывод полупрозрачного пузырька
end;
```

Пузырек появляется снизу экрана и доплывает до его верха, всегда находясь целиком в пределах экрана. Таким образом, мы оставляем меньше шансов возникновения ошибок при воспроизведении. Ошибки же, возникающие при воспроизведении рыбок, просто игнорируем.

Значение константы `MaxImages` задает максимально возможное число пар образов рыбки и пузырька, значение переменной `NumImages` устанавливает текущее количество образов (на экране должна присутствовать хотя бы одна рыбка и один пузырек). Позже мы узнаем, как устанавливается значение этой переменной, а пока просто посмотрим, какие переменные хранят состояние нашей системы:

```
Bubble : Array [0..MaxImages - 1] of TBubble;   // Массив пузырьков
Fish : Array [0..MaxImages - 1] of TFish;       // Массив рыбок
NumImages : 1..MaxImages;          // Текущее количество пар образов
```

Система инициализируется в начале работы приложения:

```
for i := 0 to NumImages - 1 do begin
  Bubble [i] := TBubble.Create;
  Bubble [i].Init;
  Fish [i] := TFish.Create;
  Fish [i].Init;
end;
```

Обратите внимание, что при воспроизведении кадра пары наших образов отображаются поочередно. Чтобы усилить иллюзию пространства, пузырьки будут загораживаться некоторыми рыбками, но проплывать "поверх" других:

```
for i := 0 to NumImages - 1 do begin
  Bubble [i].Render;
  Fish [i].Render;
end;
```

Теперь нам необходимо разобрать работу нашего приложения в режиме предварительного просмотра. В этом режиме система запускает очередную копию хранителя экрана. Для предотвращения работы нескольких копий приложения я сделал следующее: в модуле проекта в список `uses` добавил подключение модуля Windows и работу приложения начинаю с проверки наличия его запущенной копии:

```
var
  Wnd : HWND;
begin
  Wnd := FindWindow ('TfrmDD', 'Демонстрационная заставка');
```

Если есть такая копия, то следующее запущенное приложение закрывает его и посылает сообщение `WM_CLOSE`:

```
if Wnd <> 0 then PostMessage (Wnd, WM_CLOSE, 0, 0);
```

В режиме предварительного просмотра хранитель экрана запускается с ключом `/p`. Вторым параметром передается контекст окна предварительного просмотра. Если же пользователь выбрал режим задания параметров хранителя, он запускается с ключом `/c`. Параметр у нашего хранителя один — количество пар образов, и его значение будет задаваться пользователем в отдельном окне, с помощью компонента `tbFish` класса `TTrackBar`, а храниться в реестре.

Глобальная переменная `wrkHandle` предназначена для хранения значения дескриптора окна, в котором будет выводиться картинка. Сразу после запуска приложения стартует процедура, определяющая режим работы хранителя:

```
function TfrmDD.RunScreenSaver : BOOL;
const
  SECTION = 'Fish';        // Название секции в реестре
var
  S : string;
  FIniFile: TRegIniFile;   // Для работы с реестром
begin
  FIniFile := TRegIniFile.Create;
  // Считываем из реестра записанное значение
  NumImages := FIniFile.ReadInteger(SECTION, 'NumImages', MaxImages);
  S := ParamStr(1);          // Первый параметр при запуске хранителя
  if Length(S) > 1 then begin
    Delete(S, 1, 1);         // Удаляем значок "/"
    S[1] := UpCase(S[1]);    // Переводим в верхний регистр
    if S = 'P' then begin    // Режим предварительного просмотра
      flgWindowed := True;   // Задаем оконный режим
      // Второй параметр — ссылка на окно предварительного просмотра
      wrkHandle := StrToInt(ParamStr(2));
    end else
    if S[1] = 'C' then begin              // Выбран пункт "Настройка"
      with TfrmPar.Create (nil) do begin  // Выводим окно параметров
        tbFish.Max := MaxImages;          // Параметры ползунка
        tbFish.Position := NumImages;
        ShowModal;
        NumImages := tbFish.Position; // Выбранное пользователем значение
        Free;                 // Удаляем окно задания параметров хранителя
      end;
      // Записываем в реестр установленное значение параметра
      FIniFile.WriteInteger (SECTION, 'NumImages', NumImages);
```

```
      FIniFile.Free;
      Result := False;
      Exit;
    end;
  end;
  if Assigned (FIniFile) then FIniFile.Free;
  Result := True;
end;
```

После выполнения данной процедуры происходит инициализация DirectDraw. Код этого процесса очень объемный, но нами разобран достаточно хорошо. Здесь устанавливается оконный либо полноэкранный режим. Единственное замечание: в отличие от предыдущих оконных приложений в настоящем примере воспроизведение осуществляется не в собственном окне, поэтому его необходимо скрыть. Сделать это можно разными способами. Я выбрал тот, что основан на использовании региона:

```
var
  Rgn : THandle;
  Rgn := CreateRectRgn (0, 0, 0, 0);   // Пустой регион
  SetWindowRgn(Handle, Rgn, True);     // Убираем окно
```

Осталось последнее, на что следует обратить внимание — фон. Как я уже говорил, он состоит из зацикленных образов, размером 200×200 пикселов. Для оптимизации я не покрываю "паркетной плиткой" экран при каждой перерисовке кадра, а создаю поверхность фона размером 1000×800 пикселов и заполняю ее только один раз, при инициализации. По ходу работы приложения на экран выводятся фрагменты этого фона, размером 640×480 пикселов, и каждый раз происходит небольшой сдвиг координат некоторого фрагмента. Вспомогательный таймер задает величину этого сдвига случайным образом.

Возможные ошибки в работе любого хранителя экрана могут привести к тяжелым последствиям для пользователя, поэтому в случае возникновения исключений при восстановлении поверхностей наш хранитель экрана безоговорочно завершает свою работу. При включенных энергосберегающих функциях отключение воспроизведения приведет к такому аварийному завершению работы. Во избежание этого код функции восстановления поверхностей приведите к следующему виду:

```
function TfrmDD.RestoreAll : HRESULT;
var
  i : Integer;
  hRet : HRESULT;
begin
  Result := FDDSPrimary._Restore;
  if Succeeded (Result) then begin
```

```
    if flgWindowed then begin
      hRet := FDDSBack._Restore;
      if Failed (hRet) then begin
        Result := hRet;
        Exit;
      end;
    end;
    hRet := FDDSBackGround._Restore;
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    hRet := FDDSImage._Restore;
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    hRet := CreateFromImage (FDDSImage, imgBlue, 200, 200);
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    hRet := Prepare;    // Заполнение поверхности фона
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    // Восстановление поверхности рыбок
    for i := NumImages - 1 downto 0 do begin
      hRet := Fish [i].FDDSFish._Restore;
        if Failed (hRet) then begin
          Result := hRet;
          Exit;
        end;
      end;
      // Повторная инициализация
    for i := 0 to NumImages - 1 do Fish [i].Init;
  end;
end;
```

В качестве задания введите еще один параметр хранителя: яркость либо разрешение. Иначе у некоторых пользователей появится слишком блеклая картинка.

Итак, мы создали собственный хранитель экрана. Правда, нам придется вернуться к нему, чтобы снабдить его диалоговым окном ввода пароля. Но уже сейчас мы можем сделать некоторые оптимистические выводы.

Написали мы наше приложение на Delphi, выводим четыре десятка образов, половина из которых полупрозрачна, и работает наш хранитель экрана со вполне удовлетворительной скоростью даже на маломощных видеокартах, хотя код во многих местах совершенно не оптимизирован, и мы не используем ассемблерные вставки, да и располагаются образы на отдельных поверхностях.

Проверка столкновений

Такая задача относится к разряду наиболее распространенных, и ее рассмотрения нам не обойти. Как принято в настоящей книге, ознакомимся с решением на примере конкретного проекта. Располагается этот проект в каталоге Ex09, очередная вариация на бильярдную тему: по экрану мечутся, отскакивая от стенок и друг от друга, девять сфер и одна замысловатая фигура (рис. 4.5).

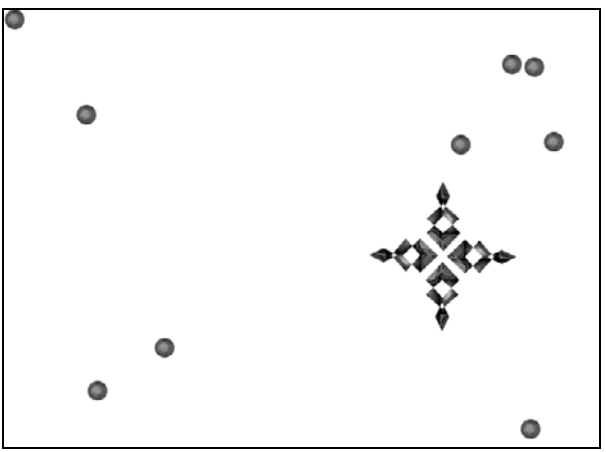


Рис. 4.5. Элегантный пример на проверку столкновений спрайтов

Если бы на экране присутствовали только круги, то, конечно, решать задачу можно было бы, просто опираясь на координаты их центров, но присутствие звездообразной фигуры усложняет задачу.

Код программы во многом похож на отдельные предыдущие примеры, однако имеет и некоторые отличия. Нам надо подробно разобрать этот пример, поскольку многое из него ляжет в основу некоторых последующих.

Введены такие типы, способствующие удобному оперированию со спрайтами:

```
type
  TCollideInfo = record
    X, Y : Integer;  // Вспомогательная запись, координаты столкновения
  end;
```

```
  TSprite = class               // Класс спрайта
    SpriteWidth : Integer;      // Размеры
    SpriteHeight : Integer;
    FSpriteSurface : IDirectDrawSurface7; // Поверхность
    PosX, PosY : Integer;       // Позиция
    Collide : BOOL;             // Флаг, связанный со столкновением
    function GetRect : TRect;   // Прямоугольник, ограничивающий спрайт
    function GetCenterX : Integer;  // Координаты центра
    function GetCenterY : Integer;
    // вывод спрайта на экран
    function Show (const FDDSBack : IDirectDrawSurface7) : HRESULT;
    procedure CalcVector;           // Инициализация направления движения
    procedure Update;               // Вычислить новые координаты
    procedure Init (const FDD : IDirectDraw7; const fileName : PChar);
    procedure Hit (const S : TSprite);   // Столкновение
  private
    Xinc : Integer;                // Приращения координат
    Yinc : Integer;
    CollideInfo : TCollideInfo;    // Координаты столкновения
  end;
```

В примере используется 8-битный режим, палитра одного из образов устанавливается для первичной поверхности и для всех спрайтов. Для разнообразия и закрепления пройденного не будем пользоваться готовой функцией загрузки образа на поверхность. Поверхность спрайтов создадим самостоятельно.

Обратите внимание, что в таком случае требуется формат пиксела "дочерней" поверхности задавать явно, и должен этот формат совпадать с форматом пиксела первичной поверхности. Иначе вполне может случиться так, что поверхности образов будут создаваться не 8-битными, и палитру на них установить не удастся:

```
const
  ScreenWidth    = 640;
  ScreenHeight   = 480;
  ScreenBitDepth = 8;
  NumSprites     = 10;  / Всего спрайтов, один из них — не круг, а фигура
var
  frmDD : TfrmDD;
  spr : Array [0..NumSprites — 1] of TSprite;         // Массив спрайтов
  PixelFormat : TDDPixelFormat;     // Для согласования форматов пиксела
```

Значение переменной `PixelFormat` устанавливается после создания первичной поверхности, до инициализации системы образов:

```
procedure TfrmDD.FormCreate(Sender: TObject);
var
  hRet : HRESULT;
```

```
  ddsd : TDDSurfaceDesc2;
  ddscaps : TDDSCaps2;
  i : Integer;
begin
  FDDPal := nil;
  FDDSBack := nil;
  FDDSPrimary := nil;
  FDD := nil;
  hRet := DirectDrawCreateEx (nil, FDD, IDirectDraw7, nil);
  if Failed (hRet) then ErrorOut(hRet, 'DirectDrawCreateEx');
  hRet := FDD.SetCooperativeLevel(Handle, DDSCL_FULLSCREEN or
          DDSCL_EXCLUSIVE);
  if Failed (hRet) then ErrorOut(hRet, 'SetCooperativeLevel');
  hRet := FDD.SetDisplayMode (ScreenWidth, ScreenHeight,
                              ScreenBitDepth, 0, 0);
  if Failed (hRet) then ErrorOut(hRet, 'SetDisplayMode');
  ZeroMemory(@ddsd, SizeOf(ddsd));
  with ddsd do begin
    dwSize := SizeOf(ddsd);
    dwFlags := DDSD_CAPS or DDSD_BACKBUFFERCOUNT;
    ddsCaps.dwCaps := DDSCAPS_PRIMARYSURFACE or DDSCAPS_FLIP or
                      DDSCAPS_COMPLEX;
    dwBackBufferCount := 1;
  end;
  hRet := FDD.CreateSurface(ddsd, FDDSPrimary, nil);
  if Failed (hRet) then ErrorOut(hRet, 'Create Primary Surface');
  ZeroMemory(@ddscaps, SizeOf(ddscaps));
  ddscaps.dwCaps := DDSCAPS_BACKBUFFER;
  hRet := FDDSPrimary.GetAttachedSurface(ddscaps, FDDSBack);
  if Failed (hRet) then ErrorOut(hRet, 'GetAttachedSurface');
  FDDSBack._AddRef;
  // Палитра должна быть считана до инициализации спрайтов
  FDDPal := DDLoadPalette(FDD, '1.bmp');
  if FDDPal = nil then ErrorOut(DD_FALSE, 'DDLoadPalette');
  hRet := FDDSPrimary.SetPalette(FDDPal);
  if Failed (hRet) then ErrorOut(hRet, 'SetPalette');
  // Определяемся с форматом пиксела первичной поверхности
  ZeroMemory(@PixelFormat, SizeOf(PixelFormat));
  PixelFormat.dwSize := SizeOf(PixelFormat);
  hRet := FDDSPrimary.GetPixelFormat(PixelFormat);
  if Failed (hRet) then ErrorOut(hRet, 'GetPixelFormat');
  Randomize;
  // Первый спрайт — фигура
  spr [0] := TSprite.Create;
  spr [0].Init (FDD, '1.bmp');
```

```
  // Остальные спрайты — сферы
  for i := 1 to NumSprites — 1 do begin
    spr [i] := TSprite.Create;
    spr [i].Init (FDD, '2.bmp');
  end;
end;
```

Инициализация спрайта реализована "длинным" кодом:

```
procedure TSprite.Init (const FDD : IDirectDraw7;
                        const fileName : PChar);
var
  Bitmap : TBitmap;
  hRet : HResult;
  DC : HDC;
  ddsd : TDDSurfaceDesc2;
begin
  FSpriteSurface := nil;
  Bitmap := TBitmap.Create;
  Bitmap.LoadFromFile(fileName);
  ZeroMemory(@ddsd, SizeOf(ddsd));
  with ddsd do begin
    dwSize := SizeOf(ddsd);
    dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH or
               DDSD_PIXELFORMAT;
    ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
    dwHeight := bitmap.Height;
    dwWidth := bitmap.width;
    ddpfPixelFormat := PixelFormat;   // Явно задаем 8-битный формат
  end;
  hRet := FDD.CreateSurface(ddsd, FSpriteSurface, nil);
  if Failed(hRet) then frmDD.ErrorOut(hRet, 'CreateSpriteSurface');
  // Воспроизведение картинки на поверхности спрайта
  if FSpriteSurface.GetDC(DC) = DD_OK then begin
    BitBlt(DC, 0, 0, Bitmap.Width, Bitmap.Height, Bitmap.Canvas.Handle,
           0, 0, SRCCOPY);
    FSpriteSurface.ReleaseDC(DC);
  end;
  // Цветовой ключ для всех спрайтов — белый
  hRet := DDSetColorKey (FSpriteSurface, RGB(255, 255, 255));
  if Failed (hRet) then frmDD.ErrorOut(hRet, 'DDSetColorKey');
  SpriteWidth := Bitmap.Width;            // Задаем размеры спрайта
  SpriteHeight := Bitmap.Height;
  Bitmap.Free;
  // Устанавливаем одну палитру для всех образов
  hRet := FSpriteSurface.SetPalette(frmDD.FDDPal);
```

```
  if Failed (hRet) then frmDD.ErrorOut(hRet, 'SetPalette');
  Collide := False;        // Явно инициализируем значение свойства
  PosX := random (500);    // Координаты задаются случайно
  PosY := random (300);
  CalcVector;              // Определяемся с направлением движения
end;
```

Инициализация направления движения вызывается только при создании спрайта, но намеренно вынесена в отдельный метод, чтобы добиться того, чтобы ни один из спрайтов не имел нулевой скорости по какой-либо оси:

```
procedure TSprite.CalcVector;
begin
  Xinc := random (7) — 3;    // Случайные значения в интервале [-3; 3]
  Yinc := random (7) — 3;
  if (Xinc = 0) or (Yinc = 0) then CalcVector; // Повторяем генерацию
end;
```

Методы спрайта с префиксом "Get" предназначены для получения информации о спрайте:

```
function TSprite.GetCenterX : Integer;   // Координаты центра
begin
  Result := PosX + SpriteWidth div 2;
end;
function TSprite.GetCenterY : Integer;
begin
  Result := PosY + SpriteHeight div 2;
end;
function TSprite.GetRect : TRect;        // Ограничивающий прямоугольник
begin
  SetRect (Result, PosX, PosY, PosX + SpriteWidth, PosY + SpriteHeight);
end;
```

В момент столкновения спрайта фиксируем текущую позицию, а в случае одновременного столкновения с несколькими спрайтами выполняем операцию один раз:

```
procedure TSprite.Hit(const S : TSprite);
begin
  if not Collide then begin     // На случай одновременного столкновения
    CollideInfo.X := S.GetCenterX;
    CollideInfo.Y := S.GetCenterY;
    Collide := True;
  end;
end;
```

При пересчете координат помним о том, что спрайт должен отскакивать от стенок и от других спрайтов.

```
procedure TSprite.Update;
var
  CenterX : Integer;
  CenterY : Integer;
  XVect : Integer;
  YVect : Integer;
begin
  if Collide then begin         // Столкновение
     CenterX := GetCenterX;     // Текущее положение
     CenterY := GetCenterY;
     XVect := CollideInfo.X — CenterX;    // Вектор из центра в точку
     YVect := CollideInfo.Y — CenterY;    // Столкновения
     // Для предотвращения залипания столкнувшихся спрайтов
     if ((Xinc > 0) and (Xvect > 0)) or ((Xinc < 0) and (XVect < 0))
          then Xinc := -Xinc;
     if ((Yinc > 0) and (YVect > 0) or (Yinc<0) and (YVect < 0))
        then  Yinc := -Yinc;
     Collide := False;
  end;
  // Собственно обновление позиции
  PosX := PosX + Xinc;
  PosY := PosY + Yinc;
  // Столкновение со стенками
  if PosX > ScreenWidth — SpriteWidth then begin
    Xinc := -Xinc;
    PosX := ScreenWidth — SpriteWidth;
    end else
    if PosX < 0 then begin
    Xinc := -Xinc;
    PosX := 0;
  end;
  if PosY > ScreenHeight — SpriteHeight then begin
    Yinc := -Yinc;
    PosY := ScreenHeight — SpriteHeight;
    end else
    if PosY < 0 then begin
    Yinc := -Yinc;
    PosY := 0;
  end;
end;
```

Функция воспроизведения лаконична:

```
function TSprite.Show (const FDDSBack : IDirectDrawSurface7) : HRESULT;
begin
  Result := FDDSBack.BltFast (PosX, PosY, FSpriteSurface, nil,
            DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
end;
```

Перерисовка кадра осуществляется с небольшим интервалом, поэтому переключение буферов переместилось в этот код, иначе появится мерцание картинки:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  i : Integer;
  s1, s2 : Integer;
  hRet : HRESULT;
begin
  ThisTickCount := GetTickCount;
  if ThisTickCount - LastTickCount > 10 then begin     // Время подошло
    hRet := Clear (255, 255, 255);          // Стираем фон белым цветом
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    for i := 0 to NumSprites - 1 do begin   // Цикл по спрайтам
      spr [i].Update;               // Определить новую позицию
      hRet := spr [i].Show (FDDSBack);      // Воспроизвести
      if Failed (hRet) then begin
        Result := hRet;
        Exit;
      end;
    end;
    // Ищем столкнувшиеся спрайты
    for s1 := 0 to NumSprites - 1 do
      for s2 := s1 + 1 to NumSprites - 1 do
        if SpritesCollidePixel (spr [s1], spr[s2]) then begin
          spr [s1].Hit (spr [s2]);
          spr [s2].Hit (spr [s1]);
        end;
    FlipPages;      // Переключение буферов
    LastTickCount := GetTickCount;
  end;
  Result := DD_OK;
end;
```

При восстановлении поверхностей аккуратно работаем с поверхностями спрайтов, вызываем метод `Restore` и переустанавливаем палитру для каждой из них:

```
function TfrmDD.RestoreAll : HRESULT;
var
  i : Integer;
  hRet : HRESULT;
```

```
begin
  hRet := FDDSPrimary._Restore;
  if Succeeded (hRet) then begin
    FDDPal := nil;
    FDDPal := DDLoadPalette(FDD, '1.bmp');
    // Восстанавливаем палитру
    if FDDPal <> nil then begin
      if Failed (FDDSPrimary.SetPalette(FDDPal))
        then ErrorOut(DDERR_PALETTEBUSY, 'SetPalette');
    end
    else ErrorOut(DDERR_PALETTEBUSY, 'DDLoadPalette');
      for i := 0 to NumSprites - 1 do begin
      // Восстанавливаем поверхность спрайтов
        hRet := spr [i].FSpriteSurface._Restore;
          if Failed(hRet) then begin
            Result := hRet;
            Exit;
          end;
          // Переустанавливаем поверхность спрайта
          if Failed (spr [i].FSpriteSurface.SetPalette(FDDPal))
            then ErrorOut(DDERR_PALETTEBUSY, 'SetPalette');
    // Восстанавливаем изображение
    if i = 0 then spr [i].Init (FDD, '1.bmp')
      else spr [i].Init (FDD, '2.bmp');
    end;
    Result := DD_OK
  end else Result := hRet;
end;
```

По завершении работы также нельзя забывать о поверхностях спрайтов:

```
procedure TfrmDD.FormDestroy(Sender: TObject);
var
  i : Integer;
begin
  if Assigned(FDD) then begin
    if Assigned(FDDPal) then FDDPal := nil;
    for i := 0 to NumSprites - 1 do begin
      if Assigned(spr [i].FSpriteSurface) then begin
        spr [i].FSpriteSurface._Release;
        spr [i].FSpriteSurface := nil;
      end;
      spr [i].Free;
    end;
    if Assigned(FDDSPrimary) then begin
      FDDSPrimary._Release;
```

```
      FDDSPrimary := nil;
    end;
    FDD._Release;
    FDD := nil;
  end;
end;
```

Теперь посмотрим ключевую функцию этого примера, определяющую, столкнулись ли два, передаваемые в параметрах, спрайта. Начинается она с определения пересечения ограничивающих спрайты прямоугольников. Если прямоугольники не пересекаются, дальнейший анализ проводить бессмысленно, спрайты располагаются в разных частях экрана. Если есть пересечение, определяем его позицию для каждого спрайта и последовательно просматриваем содержимое пикселов поверхностей спрайтов.

Опытным путем я определил, что пикселы фона для установленной палитры имеют значение 191, поэтому такие пикселы пропускаем. Как только встречается пиксел, по адресу которого в обеих поверхностях записывается значение, отличное от 191, перебор прекращается:

```
function TfrmDD.SpritesCollidePixel(Sprite1, Sprite2 : TSprite) : BOOL;
var
  Rect1 : TRect;
  Rect2 : TRect;
  IRect : TRect;
  r1target : TRect;
  r2target : TRect;
  locWidth : Integer;
  locHeight : Integer;
  Desc1, Desc2 : TDDSURFACEDESC2;
  Ret : BOOL;
  Surfptr1 : POINTER;  // Указатели на начало области памяти поверхности
  Surfptr2 : POINTER;
  Pixel1 : PBYTE;      // Пикселы поверхностей
  Pixel2 : PBYTE;
  XX, YY : Integer;
label
  Done;
begin
  // Прямоугольники, ограничивающие спрайты
  Rect1 := Sprite1.GetRect;
  Rect2 := Sprite2.GetRect;
  // Вычисляем точку пересечения прямоугольников
  IntersectRect (IRect, Rect1, Rect2);
  // Если нет пересечения прямоугольников, спрайты сталкиваться не могут
  if (IRect.Left = 0) and (IRect.Top = 0) and
```

```
    (IRect.Right = 0) and (IRect.Bottom = 0) then begin
     Result := FALSE;
     Exit;
  end;
  // Находим положение области пересечения для каждого спрайта
  IntersectRect (r1target, Rect1, IRect);
  OffsetRect(r1target, -Rect1.Left, -Rect1.Top);
  IntersectRect (r2target, Rect2, IRect);
  OffsetRect(r2target, -Rect2.Left, -Rect2.Top);
  r2target.Right := r2target.Right - 1;
  r2target.Bottom := r2target.Bottom - 1;
  // Предыдущие две строки обеспечивают корректное нахождение
  // размеров области пересечения
  locWidth := IRect.Right - IRect.Left;
  locHeight := IRect.Bottom - IRect.Top;
  // Подготавливаем структуры для работы с памятью поверхностей
  ZeroMemory (@desc1, SizeOf(desc1));
  desc1.dwSize := SizeOf(desc1);
  ZeroMemory (@desc2, SizeOf(desc2));
  desc2.dwSize := SizeOf(desc2);
  Ret := False;
  // Запираем поверхности спрайтов
  Sprite1.FSpriteSurface.Lock(nil, desc1, DDLOCK_WAIT, 0);
  Surfptr1 := desc1.lpSurface;
  Sprite2.FSpriteSurface.Lock(nil, desc2, DDLOCK_WAIT, 0);
  Surfptr2 := desc2.lpSurface;
  // Просмотр содержимого пикселов для каждого спрайта
  // в пределах области пересечения
  for YY := 0 to locHeight - 1 do
    for XX := 0 to locWidth - 1 do begin
      // Для оптимизации эти действия можно свернуть в одну строку
      Pixel1 := PByte (Integer(Surfptr1)+(yy+r1target.Top)*desc1.lPitch+
                       (xx+r1target.Left));
      Pixel2 := PByte (Integer(Surfptr2)+(yy+r2target.Top)*desc2.lPitch+
                       (xx+r2target.Left));
      if (Pixel1^ <> 191) and (Pixel2^ <> 191) then begin
        Ret := True;          // Найдено пересечение, выходим
        goto Done;
      end;
    end;
Done:
  Sprite2.FSpriteSurface.Unlock(nil);
  Sprite1.FSpriteSurface.Unlock(nil);
  Result := Ret;
end;
```

Спрайты и оконный режим

Взгляните на рис. 4.6, на котором запечатлен момент работы проекта из каталога Ex10, единственного примера этого раздела.

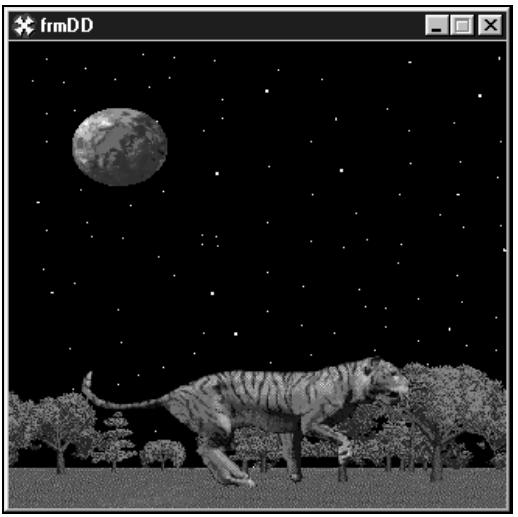


Рис. 4.6. Очень важный пример работы приложения в оконном режиме

На экране выводится фантастическая картинка с тигром, бегущим на фоне леса. На заднем плане отображается звездное небо, в небесах — вращающаяся планета. Все образы, кроме звездного неба, меняются со временем.

Для подготовки примера я взял, с любезного разрешения корпорации Intel, образы, поставляемые в составе RDX COM SDK.

Все используемые образы реализованы в 256-цветной палитре, а при оконном режиме нельзя явно задавать формат первичной поверхности. Поэтому в данном примере, подобно предыдущему, при создании поверхностей образов явно задается формат пиксела для каждой из них.

Разница хорошо заметна, если установить 256-цветную палитру рабочего стола, вариант с явным указанием формата пиксела выводит неискаженную картинку. Удалите в коде все, связанное с форматом пиксела, и запустите перекомпилированное приложение в 8-битной палитре. Вы должны увидеть, как сильны потери в качестве передачи цветов изображения.

В остальном, рассмотренный пример не сильно отличается от предыдущих. Ограничусь лишь небольшими замечаниями.

Образ леса по размерам совпадает с размерами окна и для создания иллюзии движения отображается в два этапа. На первом этапе выводится последовательно сужающийся фрагмент, примыкающий к правой границе образа,

а впритык к ней — расширяющийся фрагмент, примыкающий к левой границе образа леса.

Должен напомнить, что при использовании отсечения итоговый вывод на первичную поверхность должен осуществляться с помощью метода `Blt`. Отображение на "самодельном" заднем буфере выполняется с помощью метода `BltFast`. Таким же может быть и финальный вывод, только если не выполнять отсечение.

Неспешно поработайте с этим примером. Определите по коду, с помощью каких клавиш можно менять скорость бега тигра, вращения планеты и смещения фона.

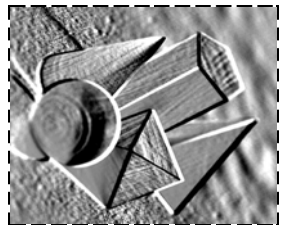
Не пропустите также, что этот пример совершенно безболезненно переживает моменты "засыпания" компьютера: функция восстановления поверхностей вызывается до тех пора, пока не возвратит успешный результат.

Что вы узнали в этой главе

Вы познакомились со вспомогательной библиотекой разработчика DDUtil8, предоставляющей объектно-ориентированный подход к применению DirectDraw и появившейся с восьмой версией DirectX; изучили принципы работы с меняющимися во времени образами; научились определять моменты столкновения спрайтов.

Главный вывод, который мы можем сделать из примеров этой главы, состоит в том, что Delphi вполне можно использовать для разработки больших проектов, требующих высокой скорости воспроизведения.

ГЛАВА 5



Пишем игру

В данной главе освещаются вопросы использования знаний, полученных в предыдущих главах, для разработки проекта-каркаса полноценной игры. Такие проекты обычно называются "движками".

Примеры располагаются в каталоге \Examples\Chapter05.

Оригинальный сплэш

Так в обиходе программисты называют заставки, появляющиеся в начале работы приложения (splash — красочное пятно, всплеск). Выводят их на время долгой инициализации основного модуля, как правило, для того, чтобы скрасить секунды (или минуты) ожидания пользователя. Иногда их использованием авторы преследуют цель поразить воображение зрителя, заявить на весь мир о своих выдающихся художественных и профессиональных способностях.

В этом разделе я приведу пример, который может стать основой для создания вашей собственной оригинальной заставки. Это проект каталога Ex01. Во время его работы посередине рабочего стола выводится изображение земного шара, на фоне которого вращается фраза "DirectX". На неискушенных пользователей окна непрямоугольной формы обычно производят сильное впечатление. Подобные окна можно создавать разными способами, например, с помощью регионов. Мы же решим задачу обычным для DirectDraw способом. Совсем необязательно должно получаться именно круглое окно, как в моем примере.

Приемы, используемые в проекте, во многом вам знакомы по примерам предыдущих глав, однако добавилось и кое-что новое.

Заставка должна появляться всегда посередине экрана, при любых установленных разрешениях, поэтому в начале работы нам необходимо определить

текущие размеры рабочего стола, относительно середины которого выверить координаты вывода нашего образа размером 256×256 пикселов:

```
HalfWidth  := (GetSystemMetrics(SM_CXSCREEN) – 256) div 2;
HalfHeight := (GetSystemMetrics(SM_CYSCREEN) – 256) div 2;
```

Примечание

Конечно, если по ходу работы заставки пользователь поменяет настройки рабочего стола, значения установок, полученные нами в начале работы, станут неактуальны. Но нет смысла вычислять их значения беспрерывно, ведь в ситуации смены режима дальнейший вывод будет невозможен, точно так же, как и для любого другого приложения, использующего DirectDraw.

Уровень кооперации устанавливается нормальным, а очередной кадр не выходит за границу предыдущего. Поэтому наша заставка эффектно располагается поверх всех окон, и нет необходимости производить манипуляций с ее фоном (запоминать и восстанавливать для каждого кадра).

Но для того, чтобы заставка не исчезла при изменениях на рабочем столе, ее необходимо постоянно перерисовывать. Чтобы перерисовка протекала с большим эффектом, работают с двумя образами: земного шара и вращающейся надписи.

Мы уже использовали прием с вращением изображения, основанный на непосредственном доступе к 8-битной поверхности. Пример этой главы рассчитан на, минимум, 16-разрядную глубину поверхности, а вызываемая нами тогда функция вращения для такого режима требует корректировки.

Я переписал эту функцию. Теперь поворачивается содержимое передаваемого объекта класса `TBitmap`, и возвращается объект такого же класса:

```
function TfrmDD.RotateBmp (const BitmapOriginal: TBitmap;
                           const iRotationAxis, jRotationAxis: Integer;
                           const AngleOfRotation: Single): TBitmap;
const
  MaxPixelCount = 32768;
type
  TRGBTripleArray = Array [0..MaxPixelCount-1] of TRGBTriple;
  pRGBTripleArray = ^TRGBTripleArray;
var
  cosTheta    : Single;
  i           : Integer;
  iOriginal   : Integer;
  iPrime      : Integer;
  j           : Integer;
  jOriginal   : Integer;
  jPrime      : Integer;
  RowOriginal : pRGBTripleArray;
```

```
  RowRotated  : pRGBTRipleArray;
  sinTheta    : Single;
begin
  Result := TBitmap.Create;      // Создание результирующего растра
  Result.Width := BitmapOriginal.Width;
  Result.Height := BitmapOriginal.Height;
  Result.PixelFormat := pf24bit; // Очень важно задать явно режим пиксела
  sinTheta := sin (AngleOfRotation);
  cosTheta := cos (AngleOfRotation);
  // Расчет источника для пикселов повернутого растра
  for j := Result.Height - 1 downto 0 do begin
    RowRotated := Result.Scanline[j];
    jPrime := j - jRotationAxis;
    for i := Result.Width-1 downto 0 do begin
      iPrime := i - iRotationAxis;
      iOriginal := iRotationAxis + round(iPrime * CosTheta - jPrime *
                   sinTheta);
      jOriginal := jRotationAxis + round(iPrime * sinTheta + jPrime *
                   cosTheta);
      if (iOriginal >= 0) and (iOriginal <= BitmapOriginal.Width-1) and
         (jOriginal >= 0) and (jOriginal <= BitmapOriginal.Height-1)
      then begin
        RowOriginal := BitmapOriginal.Scanline[jOriginal];
        RowRotated[i] := RowOriginal[iOriginal]
      end
      else begin    // "Новые" пикселы заполняются черным, цветом ключа
        RowRotated[i].rgbtBlue := 0;
        RowRotated[i].rgbtGreen := 0;
        RowRotated[i].rgbtRed := 0
     end
    end
  end;
end;
```

При перерисовке кадра поворачиваем первоначальное изображение на увеличивающийся угол, копируем полученный растр на вспомогательную поверхность, а затем формируем окончательную картинку:

```
function TfrmDD.UpdateFrame : HRESULT;
begin
  // Повернутый растр копируем на поверхность FDDSLogo
  with RotateBmp (wrkBitmap, 128, 128, Angle) do begin
    DDCopyBitmap (FDDSLogo, Handle, 0, 0, Width, Height);
    Free
  end;
  Angle := Angle - 0.1;     // Наращиваем угол поворота
  if Angle > - 2 * Pi then Angle := Angle + 2 * Pi;
```

```
  // Теоретически возможные ошибки блиттинга игнорируем
  // На заднем буфере подготавливаем итоговую картинку
  FDDSBack.BltFast(0, 0, FDDSImage, nil, DDBLTFAST_WAIT or
    DDBLTFAST_SRCCOLORKEY); // Вывод фона, земной шар
  FDDSBack.BltFast(0, 0, FDDSLogo, nil, DDBLTFAST_WAIT or
    DDBLTFAST_SRCCOLORKEY); // На фон накладываем повернутую надпись
  // Вывод посередине экрана заставки
  Result := FDDSPrimary.BltFast(HalfWidth, HalfHeight, FDDSBack,
            nil, DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
end;
```

Перед окончанием работы заставку необходимо убрать с экрана. Обращаю внимание, как это делается: появился обработчик события `OnClose`, в котором собственно окно приложения занимает всю область заставки:

```
procedure TfrmDD.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  Left := HalfWidth;
  Top := HalfHeight;
  Width := 256;
  Height := 256;
end;
```

Космический истребитель

В этом разделе я представляю свою небольшую заготовку увлекательной игры. Проект располагается в каталоге Ex02. Имитируется полет в космосе космического корабля (рис. 5.1).

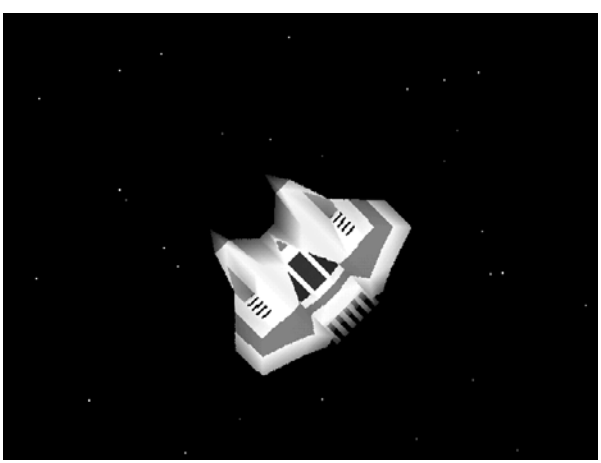


Рис. 5.1. Наконец-то мы подошли к серьезным примерам

С помощью клавиш перемещения курсора можно управлять направлением и скоростью полета истребителя.

Для создания эффекта пространства звезды, на фоне которых располагается корабль, разделены на три группы, различающиеся по яркости и скорости перемещения:

```
const
  NumStars        = 10;   // Количество звезд в каждой группе
var
  StepX : Integer = 0;    // Базовая скорость движения звезд
  StepY : Integer = 1;
type
  TCoord = record         // Тип описания текущего положения звезды
    X, Y : Integer;
  end;
var                      // Массивы звезд
  Stars1 : Array [0..NumStars - 1] of TCoord;
  Stars2 : Array [0..NumStars - 1] of TCoord;
  Stars3 : Array [0..NumStars - 1] of TCoord;
```

В начале координаты звезд задаются, конечно, случайно. При каждом обновлении кадра они циклически движутся по экрану:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  i : Integer;
begin
  ThisTickCount := GetTickCount;
  if ThisTickCount - LastTickCount > 5 then begin
    for i := 0 to NumStars - 1 do begin
    // Первая группа звезд, самое медленное движение
      Stars1 [i].X := (Stars1 [i].X + StepX);
      if Stars1 [i].X > ScreenWidth - 2 then Stars1 [i].X := 0 else
      if Stars1 [i].X < 0 then Stars1 [i].X := ScreenWidth - 2;
    // Вторая группа звезд движется в два раза быстрее
      Stars2 [i].X := (Stars2 [i].X + 2 * StepX);
      if Stars2 [i].X > ScreenWidth - 2 then Stars2 [i].X := 0 else
      if Stars2 [i].X < 0 then Stars2 [i].X := ScreenWidth - 2;
    // Третья группа движется в три раза быстрее
      Stars3 [i].X := (Stars3 [i].X + 3 * StepX);
      if Stars3 [i].X > ScreenWidth - 2 then Stars3 [i].X := 0 else
      if Stars3 [i].X < 0 then Stars3 [i].X := ScreenWidth - 2;
    // Аналогично по координате Y
      Stars1 [i].Y := (Stars1 [i].Y + StepY);
      if Stars1 [i].Y > ScreenHeight - 2 then Stars1 [i].Y := 0 else
      if Stars1 [i].Y < 0 then Stars1 [i].Y := ScreenHeight - 2;
```

```
      Stars2 [i].Y := (Stars2 [i].Y + 2 * StepY);
      if Stars2 [i].Y > ScreenHeight - 2 then Stars2 [i].Y := 0 else
      if Stars2 [i].Y < 0 then Stars2 [i].Y := ScreenHeight - 2;
      Stars3 [i].Y := (Stars3 [i].Y + 3 * StepY);
      if Stars3 [i].Y > ScreenHeight - 2 then Stars3 [i].Y := 0 else
      if Stars3 [i].Y < 0 then Stars3 [i].Y := ScreenHeight - 2;
    end;
    LastTickCount := GetTickCount;
  end;
  Clear;                                  // Очистка заднего буфера
  for i := 0 to NumStars - 1 do begin   // Цикл рисования звезд
    FDDSBack.BltFast (Stars1 [i].X, Stars1 [i].Y,
                      FDDSImage1, nil, DDBLTFAST_WAIT);
    FDDSBack.BltFast (Stars2 [i].X, Stars2 [i].Y,
                      FDDSImage2, nil, DDBLTFAST_WAIT);
    FDDSBack.BltFast (Stars3 [i].X, Stars3 [i].Y,
                      FDDSImage3, nil, DDBLTFAST_WAIT);
  end;
  // Рисование истребителя
  Result := FDDSBack.BltFast (150, 140,
            FDDSFighter, nil, DDBLTFAST_WAIT or DDBLTFAST_SRCCOLORKEY);
end;
```

Механизм поворота образа истребителя точно такой же, как в предыдущем примере, и основан на искажении первоначального растра. В данном примере нет необходимости обращаться к функции поворота в каждом кадре. Делается это только при нажатии клавиш:

```
procedure TfrmDD.FormKeyDown(Sender: TObject; var Key: Word;
  Shift: TShiftState);
begin
  if (Key = VK_ESCAPE) or (Key = VK_F12) then begin
    Close;
    Exit;
  end else
  if Key = VK_LEFT then StepX := StepX + 1 else
  if Key = VK_RIGHT then StepX := StepX - 1 else
  if Key = VK_UP then StepY := StepY + 1 else
  if Key = VK_DOWN then StepY := StepY - 1;
  // Ограничиваем углы поворота некоторыми пределами
  if StepY < 1 then StepY := 1 else
  if StepY > 3 then StepY := 3;
  if StepX < -4 then StepX := -4 else
  if StepX > 4 then StepX := 4;
  // Копируем на поверхность истребителя новое изображение
  with RotateBmp (wrkBitmap, 170, 135, arctan (StepX / StepY)) do begin
```

```
    DDCopyBitmap (FDDSFighter, Handle, 0, 0, Width, Height);
    Free
  end;
end;
```

При восстановлении поверхностей надо не просто восстановить содержимое поверхности истребителя, но и повернуть его образ соотвественно текущему положению:

```
function TfrmDD.RestoreAll : HRESULT;
var
  hRet : HRESULT;
begin
  hRet := FDDSPrimary._Restore;
  if Succeeded (hRet) then begin
    hRet := FDDSFighter._Restore;
    if Failed (hRet) then begin
      Result := hRet;
      Exit;
    end;
    // Поворот образа истребителя на текущий угол
    with RotateBmp (wrkBitmap, 170, 135, arctan (StepX / StepY)) do begin
      hRet := DDCopyBitmap (FDDSFighter, Handle, 0, 0, Width, Height);
      Free
    end;
    if Failed (hRet)
      then ErrorOut(hRet, 'DDCopyBitmap');
      hRet := FDDSImage3._Restore;
      if Failed (hRet) then begin
        Result := hRet;
        Exit;
      end;
      hRet := DDReLoadBitmap(FDDSImage3, starBmp3);
      if Failed (hRet) then ErrorOut(hRet, 'DDReLoadBitmap');
      hRet := FDDSImage2._Restore;
      if Failed (hRet) then begin
        Result := hRet;
        Exit;
      end;
      hRet := DDReLoadBitmap(FDDSImage2, starBmp2);
      if Failed (hRet) then ErrorOut(hRet, 'DDReLoadBitmap');
      hRet := FDDSImage1._Restore;
      if Failed (hRet) then begin
        Result := hRet;
        Exit;
      end;
```

```
      hRet := DDReLoadBitmap(FDDSImage1, starBmp1);
      if Failed (hRet)
        then ErrorOut(hRet, 'DDReLoadBitmap');
      Result := DD_OK
  end else Result := hRet;
end;
```

Ну что же, дорогой читатель, если вы истомились в ожидании "настоящих" примеров, то сейчас настало самое время встряхнуться и превратить этот пример в полноценную игру. Вам осталось добавить извергающиеся лучи мощного лазерного оружия, накатывающиеся астероиды и злобного и многочисленного противника. Сделайте все это, но не сейчас: прочтите до конца хотя бы эту главу.

Игра "Меткий стрелок"

При написании мало-мальски объемной игры необходимо применять приемы, которыми новички часто пренебрегают, считая их малозначимыми. Знакомясь с примерами настоящей главы, вы легко сможете убедиться, как важно придерживаться некоторых правил, своими глазами вы увидите, как сильно выигрывает в этом случае приложение.

Следующий наш пример, проект каталога Ex03, является уже вполне законченной игрой, хотя и носит своеобразный оттенок любительских творений.

Игра состоит в том, чтобы поразить всех монстров, беспрерывно появляющихся на экране, пока их количество не достигнет какого-то предела. Вооруженный мощным оружием воин располагается в нижней части экрана и способен передвигаться только по горизонтали. Он может стрелять влево, вправо или вверх; с помощью клавиш управления курсором можно передвигать его и задавать направление стрельбы (пробелом осуществляется вертикальное направление стрельбы).

Чудовища мечутся по экрану, отталкиваясь друг от друга и от границ окна (предел нижней границы области перемещений монстров чуть выше нижней границы окна).

Несмотря на свой ужасный вид, монстры вполне безобидны и не приносят никому никакого вреда.

В игре присутствует два вида чудовищ, после попадания в монстра пули на месте трагедии остается огненный сполох (рис. 5.2).

Данный пример иллюстрирует ваше умение создать, в принципе, несложную игру без каких-либо особых ухищрений, опираясь на полученные знания. Игра работает со вполне удовлетворительной скоростью даже на маломощных компьютерах (для достижения этого используется 8-битный режим), хотя имеется значительный запас для оптимизации ее работы.

Рис. 5.2. Пример захватывающей игры "Меткий стрелок"

Код построен на основе примера из предыдущей главы с проверкой столкновений. Класс `TBaseSprite` является базовым для других классов спрайтов. Следуя логике предыдущих примеров, каждый объект имеет собственную поверхность:

```
type
  TBaseSprite = class
    FSpriteSurface : IDirectDrawSurface7;   // Поверхность
    PosX, PosY : Integer;                   // Позиция
    SpriteWidth : Integer;                  // Размеры
    SpriteHeight : Integer;
    function GetRect : TRect;            // Охватывающий прямоугольник
    procedure Show; virtual; abstract;   // Вывод
  private
    rcRect : TRect;                      // Прямоугольник кадра
  end;
```

Фон загружается из отдельного растра, все остальные образы берутся из компонентов класса `TImage` (рис. 5.3).

Классы воина, монстров и пуль являются дочерними базового класса:

```
type
  TWarrior = class (TBaseSprite)                  // Класс воина
    Direction : (dirLeft, dirRight);              // Два направления
```

```
    constructor Create (const Image : TImage);   // Конструктор
    function Restore (const Image : TImage) : HRESULT; // Восстановление
    // Метод вывода определяется в каждом дочернем классе
    procedure Show; override;
  end;
```

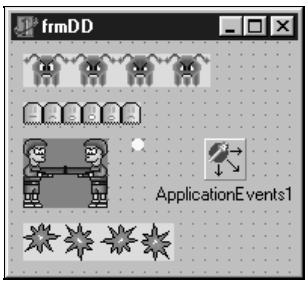


Рис. 5.3. Образы спрайтов располагаются в компонентах класса `TImage`

Обратите внимание, что каждая пуля в моей игре является отдельным спрайтом:

```
type
  TBullet = class (TBaseSprite)
    Delay : DWORD;     // Задержка, задает скорость полета пуль
    constructor Create (const Image : TImage);
    function Restore (const Image : TImage) : HRESULT;
    procedure Show; override;   // Вычисление нового положения и вывод
  private
    Xinc : Integer;             // Наращивание по каждой оси
    Yinc : Integer;
    ThisTickCount : DWORD;      // Локальный таймер для каждого спрайта
    LastTickCount : DWORD;
  end;
```

Для спрайтов монстров необходимо определять столкновения, их класс унаследовал очень многое от класса спрайтов из примера предыдущей главы:

```
type
  TCollideInfo = record
    X, Y : Integer;
  end;
  TSprite = class (TBaseSprite)
    Delay : DWORD;
    AnimFrame : Integer;     // Текущий кадр
    FrameCount : Integer;    // Всего кадров для этого вида монстров
    Collide : BOOL;
    Live : BOOL;             // Флаг, сигнализирующий, не убит ли монстр
    constructor Create (const Image : TImage; const SprDelay : DWORD;
                        const FrmCount : Integer);
```

```
    function GetCenterX : Integer;
    function GetCenterY : Integer;
    function Restore : HRESULT;
    procedure CalcVector;
    procedure Hit(S : TSprite);
    procedure Show; override;   // Вычисление нового положения и вывод
  private
    Xinc : Integer;
    Yinc : Integer;
    CollideInfo : TCollideInfo;
    ThisTickCount : DWORD;
    LastTickCount : DWORD;
  end;
```

На экране может присутствовать до двухсот спрайтов одновременно (сто монстров и сто пуль), это большая цифра, и вы в состоянии увеличить эту цифру еще больше. Программа будет дольше инициализироваться, но воспроизведение получалось у меня с очень хорошей скоростью даже тогда, когда весь экран был забит спрайтами до отказа. А тестировал я эту игру на очень скромной машине:

```
const
  DelayMonsters = 1000;// Через сколько миллисекунд появится новый монстр
  MaxSprites    = 100; // Ограничение количества спрайтов
var
  Monsters : Array [0..MaxSprites – 1] of TSprite;  // Массив чудовищ
  Bullets : Array [0..MaxSprites – 1] of TBullet;   // Массив пуль
  Warrior : TWarrior;               // Объект бойца
  GlobalThisTickCount : DWORD;      // Глобальный таймер
  GlobalLastTickCount : DWORD;
  NumMonsters : Integer = 0;        // Текущее количество монстров
  NumBullets : Integer = 0;         // Текущее количество пуль
```

Создание отдельного спрайта (имеющего собственную поверхность) происходит очень долго, поэтому массивы спрайтов заполняются в начале работы приложения. Если же поступать так, как подсказывает логика, и создавать объекты только непосредственно перед их появлением на экране, картинка в такие моменты будет замирать на долю секунды. Создание двух сотен объектов будет долгим. Чтобы скрасить время ожидания, перед началом этого процесса я вывожу на первичную поверхность картинку фона, но можно было бы использовать и специальную заставку:

```
FDDSBackGround := DDLoadBitmap(FDD, bkBitmap, 0, 0);   // Загружаем фон
if FDDSBackGround = nil then ErrorOut(hRet, 'DDLoadBitmap');
// Палитра предварительно загружена,
// устанавливается для всех поверхностей программы
hRet := FDDSBackGround.SetPalette(FDDPal);
```

```
if Failed (hRet) then ErrorOut(hRet, 'SetPalette');
// Прямоугольник, охватывающий весь экран
SetRect(bkRect, 0, 0, ScreenWidth, ScreenHeight);
// Сразу же после загрузки на экран выводится фон
FDDSPrimary.BltFast(0, 0, FDDSBackGround, @bkRect, DDBLTFAST_WAIT);
Randomize;
// Создание объекта воина
Warrior := TWarrior.Create (ImgWarrior);
// Заполняем массив монстров
for wrkI := Low (Monsters) to High (Monsters) do
if random > 0.5
  then Monsters [wrkI] := TSprite.Create (ImgMoster1,
                                          40 + random (40), 4)
else Monsters [wrkI] := TSprite.Create (ImgMoster2, 40 + random (20), 6);
// Заполняем массив пуль
for wrkI := Low (Bullets) to High (Bullets) do
  Bullets [wrkI] := TBullet.Create (ImgBullet);
```

Чудовища в игре динамичны, каждый из них со временем меняется: шевелит глазами или раскрывает рот. Заготовки монстров содержат ленту кадров для его отображения. При создании монстра какой-то произвольный из них берется за начало цепочки кадров. Сделано это для того, чтобы не получилось, будто бы все чудовища синхронно меняются в своем обличии:

```
constructor TSprite.Create (const Image : TImage; const SprDelay : DWORD;
                            const FrmCount : Integer);
var
  DC : HDC;
  ddsd : TDDSurfaceDesc2;
  hRet : HResult;
begin
  ZeroMemory (@ddsd, SizeOf(ddsd));
  with ddsd do begin
    dwSize := SizeOf(ddsd);
    dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
    dwHeight := Image.Height;
    dwWidth := Image.Width;
    ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
  end;
  hRet := frmDD.FDD.CreateSurface(ddsd, FSpriteSurface, nil);
  if Failed (hRet) then frmDD.ErrorOut(hRet, 'CreateSpriteSurface');
  if FSpriteSurface.GetDC(DC) = DD_OK then begin
    BitBlt(DC, 0, 0, Image.Width, Image.Height, Image.Canvas.Handle,
           0, 0, SRCCOPY);
    FSpriteSurface.ReleaseDC(DC);
  end;
```

```
  // Оба вида монстров нарисованы на зеленом фоне
  DDSetColorKey (FSpriteSurface, RGB(0, 255, 0));
  FSpriteSurface.SetPalette(frmDD.FDDPal);
  SpriteHeight := Image.Height;
  // Image содержит все кадры
  SpriteWidth := Image.Width div FrmCount;
  Collide := False;
  PosX := random (640 - SpriteWidth);
  PosY := random (426 - SpriteHeight);
  CalcVector;
  AnimFrame := random (FrmCount);   // Текущий кадр - случайно
  // Количество кадров для каждого вида монстров свое
  FrameCount := FrmCount;
  // Индивидуальная задержка смены кадров, передается случайное число
  Delay := SprDelay;
  // Прямоугольник кадра, фрагмент из ленты кадров
  SetRect (rcRect, AnimFrame * SpriteWidth, 0,
           AnimFrame * SpriteWidth + SpriteWidth, SpriteHeight);
  Live := True;
  LastTickCount := GetTickCount;
end;
```

Остальные методы классов спрайтов или схожи с предыдущими примерами, или тривиальны. Подробно разбирать их, думаю, не стоит, обращу внимание только на некоторые моменты.

Столкновение спрайтов определяется в программе просто выяснением наличия пересечения охватывающих прямоугольников. Так получается быстрее, а на глаз зазор между спрайтами в этом примере неразличим.

В рассматриваемом примере блиттинг спрайтов на задний буфер осуществляется с флагом `DDBLTFAST_DONOTWAIT`, что редко для примеров этой книги. Считаем, что задний буфер будет всегда доступным для вывода. При большом количестве отображаемых образов ожидание доступности устройства является слишком большой роскошью.

Каждый спрайт снабжен методом, связанным с восстановлением потерянной поверхности, в котором по высоте спрайта определяем, с какой картинкой ассоциирован конкретный объект:

```
function TSprite.Restore : HRESULT;
var
  DC : HDC;
  hRet : HRESULT;
  Image : TImage;
begin
  hRet := FSpriteSurface._Restore;
  if Failed (hRet) then begin
```

```
    Result := hRet;
    Exit;
  end;
  // Пользуемся тем, что высота трех образов различна
  if SpriteHeight = 15 then Image := frmDD.ImgMoster2 else
  if SpriteHeight = 22 then Image := frmDD.ImgMoster1
    else Image := frmDD.ImgDead;
  // Копируем нужный образ на восстанавливаемую поверхность
  if FSpriteSurface.GetDC(DC) = DD_OK then begin
    BitBlt(DC, 0, 0, Image.Width, Image.Height, Image.Canvas.Handle,
           0, 0, SRCCOPY);
    FSpriteSurface.ReleaseDC(DC);
  end;
  Result := FSpriteSurface.SetPalette(frmDD.FDDPal);
end;
```

Пули, долетевшие до края окна, должны удаляться из списка воспроизводимых образов:

```
procedure UpdateBul;
var
  wrkI, wrkJ : Integer;
begin
  for wrkI := 0 to NumBullets - 2 do
    if (Bullets [wrkI].PosX >= 632) or (Bullets [wrkI].PosX <= 0) or
       (Bullets [wrkI].PosY <= 0) then begin
    for wrkJ := wrkI to NumBullets - 1 do  // Сдвигаем содержимое массива
    with Bullets [wrkJ] do begin
      PosX := Bullets [wrkJ + 1].PosX;
      PosY := Bullets [wrkJ + 1].PosY;
      Xinc := Bullets [wrkJ + 1].Xinc;
      Yinc := Bullets [wrkJ + 1].Yinc;
    end;
    NumBullets := NumBullets - 1;
  end;
end;
```

Положение пули, попавшей в монстра, устанавливается за пределами экрана, чтобы она не летела дальше, поражая других чудовищ.

Для погибшего монстра необходимо заменить размеры спрайта и ленту кадров. Все эти действия следует производить максимально быстро. По возможности будем опираться на конкретные числа; проверки успешности, равно как и академическое пересоздание поверхности, опускаем:

```
procedure TfrmDD.DeadMonster (const Number : Integer);
var
  DC : HDC;
  ddsd : TDDSurfaceDesc2;
```

```
begin
  ZeroMemory (@ddsd, SizeOf(ddsd));
  with ddsd do begin
    dwSize := SizeOf(ddsd);
    dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
    dwHeight := ImgDead.Height;
    dwWidth := ImgDead.Width;
    ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
  end;
  with Monsters[Number] do begin
    // Пересоздаем поверхность (без := nil)
    FDD.CreateSurface(ddsd, FSpriteSurface, nil);
    // Считаем, что ошибок не будет
    FSpriteSurface.GetDC(DC);
    // Конкретные числа размеров копируемого образа
    BitBlt(DC, 0, 0, 100, 25, ImgDead.Canvas.Handle, 0, 0, SRCCOPY);
    FSpriteSurface.ReleaseDC(DC);
    // Ключ необходимо переустановить
    DDSetColorKey (FSpriteSurface, RGB(0, 255, 0));
    // Опять опираемся на конкретные числа
    SpriteHeight := 25;
    SpriteWidth := 25;
    AnimFrame := 0;
    FrameCount := 4;
    Xinc := 0;       // Погибший спрайт остается неподвижный
    Yinc := 0;
    Live := False;
  end;
end;
```

Кадр перерисовывается непрерывно, но изменения в нем вносятся в соответствии с принятыми задержками:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  wrkI, s1, s2 : Integer;
begin
  GlobalThisTickCount := GetTickCount;
  // Подошло время выпустить нового монстра
  FDDSBack.BltFast(0, 0, FDDSBackGround, @bkRect, DDBLTFAST_WAIT);
  if (GlobalThisTickCount - GlobalLastTickCount > DelayMonsters)
      and (NumMonsters < High (Monsters) - 1) then begin
        Inc (NumMonsters);
        GlobalLastTickCount := GlobalThisTickCount;
  end;
  // Обновить положения и воспроизвести монстров
  for wrkI := 0 to NumMonsters - 1 do Monsters [wrkI].Show;
```

```
  Warrior.Show;   // Вывод воина поверх пролетающих монстров
  UpdateBul;      // Удалить пули, вылетевшие за пределы экрана
  // Обновить положения и отобразить пули
  for wrkI := 0 to NumBullets - 1 do Bullets [wrkI].Show;
  // Определить столкновение монстров и пуль
  for s1 := 0 to NumMonsters - 1 do
    for s2 := 0 to NumBullets - 1 do
      if Monsters [s1].Live and SpritesCollidePixel (Monsters [s1],
          Bullets [s2]) then begin
        // Попавшая пуля перемещается за границы экрана
        Bullets [s2].PosY := -10;
        DeadMonster (s1);     // Заменить образ монстра
  end;
  // Столкновения монстров, берутся в расчет только живые чудовища
  for s1 := 0 to NumMonsters - 2 do
    for s2 := s1 + 1 to NumMonsters - 1 do
      if Monsters [s1].Live and Monsters [s2].Live and
         SpritesCollidePixel (Monsters [s1], Monsters[s2]) then begin
           Monsters [s1].Hit(Monsters [s2]);
           Monsters [s2].Hit(Monsters [s1]);
      end;
  Result := DD_OK;
end;
```

Больших усилий мне стоило при подготовке данного примера то, что не бросается сразу же в глаза — уверенное восстановление поля игры. Для достижения этого приходится восстанавливать поверхности всех спрайтов, и тех, что уже выводились, и тех, что ни разу не появлялись на экране. Вследствие чего процесс восстановления происходит тоже очень долго. Здесь я опять вывожу на первичную поверхность пустой фон.

Итак, воспроизведение множества спрайтов выполняется с удовлетворительной скоростью. Наиболее слабыми местами этой пробной игры являются чересчур долгая инициализация и восстановление объектов. Также при каждом попадании в чудовище смена цепочки кадров чересчур затягивается, и в такие моменты происходит ощутимое торможение в работе игры.

Очередной пример является развитием предыдущего — это игра аналогичного жанра, поменялись только фон и вид нашего бойца (рис. 5.4).

Скорость работы игры повысилась существенно, инициализация и восстановление происходят мгновенно, и нет ощутимой паузы в моментах замены картинки спрайтов.

Однажды я уже говорил, что, в случае применения множества образов, оптимальным решением является использование одной поверхности. Если в предыдущем примере каждый объект спрайта имеет собственную поверхность, то в этом проекте заведена одна единственная поверхность, хранящая

все используемые образы. Прием простой, но, как видим, очень эффективный.

Образы спрайтов хранятся в единственном компоненте класса `TImage` (рис. 5.5).

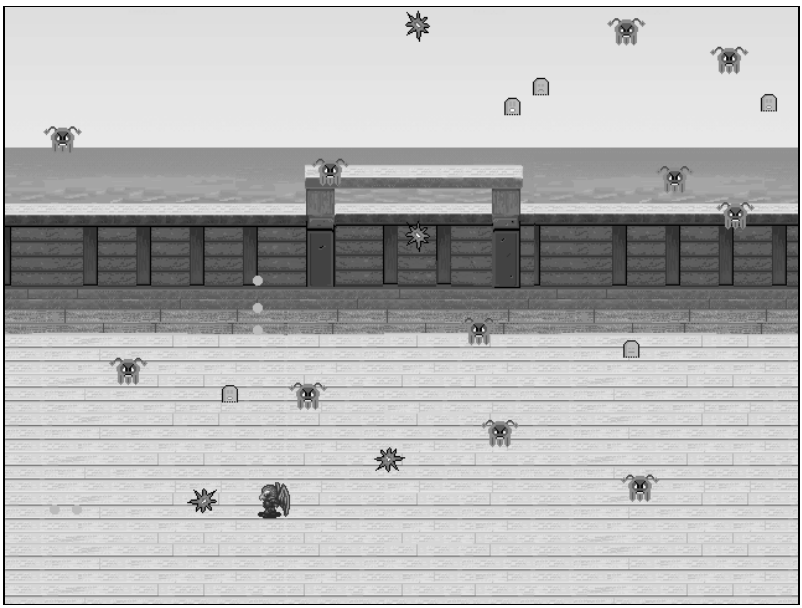


Рис. 5.4. В этом примере многое кардинально изменилось

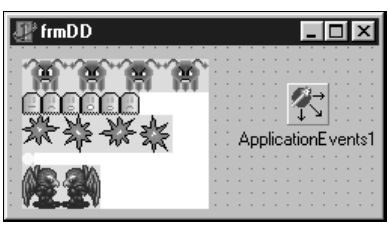


Рис. 5.5. Все используемые образы теперь хранятся в единственном компоненте класса `TImage`

В принципе, это и не обязательно. Главное, повторюсь, то, что используется одна поверхность. Она может заполняться отдельными образами или единым, как в нашем примере, но при выводе спрайтов применяется не индивидуальная поверхность спрайта, а поверхность `FDDSImages`, обслуживающая все спрайты. Вот как выглядит теперь код воспроизведения воина:

```
procedure TWarrior.Show;
begin
  if Direction = dirRight
```

```
  // rcRect устанавливается в координатах образа, хранящего все картинки
    then SetRect (rcRect, 0, 70, SpriteWidth, 70 + SpriteHeight)
    else SetRect (rcRect, SpriteWidth, 70, 2 * SpriteWidth, 70 +
                  SpriteHeight);
  // Осуществляется блиттинг FDDSImages, а не поверхности спрайта
  frmDD.FDDSBack.BltFast(PosX, PosY, frmDD.FDDSImages, @rcRect,
                         DDBLTFAST_DONOTWAIT or DDBLTFAST_SRCCOLORKEY);
end;
```

Также этот пример отличается от предыдущего тем, что пространство игры не ограничивается одним экраном, воин может продвигаться дальше правой границы, всего я использую два растровых фона, каждый размером 640×480 пикселов. Напоминаю, что некоторые видеокарты не позволяют создавать поверхности, превышающие в размерах первичную поверхность. Поэтому для хранения этих растров использую две поверхности — `FDDSOne` и `FDDSTwo`. Значение целочисленной переменной `lftRect` указывает ширину прямоугольника, вырезаемого из второй поверхности:

```
  SetRect(rcRectOne, lftRect, 0, ScreenWidth, ScreenHeight);
  // Первый фон
  FDDSBack.BltFast(0, 0, FDDSOne, @rcRectOne, DDBLTFAST_WAIT);
  if lftRect > 0 then begin   // Присутствует ли часть второго фона
    SetRect(rcRectTwo, 0, 0, lftRect, ScreenHeight);
    FDDSBack.BltFast(ScreenWidth – lftRect, 0, FDDSTwo, @rcRectTwo,
                     DDBLTFAST_WAIT);
  end;
```

Работа с клавиатурой

При изучении двух предыдущих примеров вам, наверняка, не понравилась скорость перемещения нашего воина, и, возможно, вы гадали, почему я не установил величину приращения побольше. Объяснение вы найдете в настоящем разделе.

Начинающие "игроделы" часто рассуждают так: если традиционный графический вывод совершенно не годится для масштабной игры, и для обеспечения быстрой графики надо искать другие пути, то управление, построенное на получении информации обычными способами, вполне подходит. Имея опыт разработки программ бухучета, вы не испытывали особых проблем со скоростью ввода данных, и, возможно, полагаете, что, если ваша игра использует для ввода только клавиатуру и мышь, вам не стоит напрягаться и изучать новые для вас методы организации ввода от традиционных устройств. Если это так, то вас ждет большой сюрприз, вы сами убедитесь, как сильно может улучшиться игра, если отказаться от привычных обработчиков событий, связанных с устройствами ввода.

Обычно игры используют функции библиотеки DirectInput для организации управления, с ними мы и бегло познакомимся в данном разделе. Эта библиотека является частью DirectX и содержит набор функций для обеспечения пользовательского ввода с максимальной скоростью. Высокая скорость работы даже с традиционными устройствами обеспечивается тем, что DirectInput обходит часто применяемые механизмы операционной системы и обращается к устройствам напрямую. Поэтому установленные в системе параметры, такие как частота повтора символов для клавиатуры или чувствительность мыши, не влияют на скорость ввода.

DirectInput использует модель COM. Посему, после изучения DirectDraw, нам будет легко знакомиться с ним: мы встретим здесь знакомые понятия главного объекта и интерфейсов.

Разбирая очередной пример (проект каталога Ex05), я попутно расскажу об основных понятиях библиотеки DirectInput. По виду пример представляет собой обычное оконное приложение, в компоненте класса `TMemo` выводятся скан-коды нажимаемых клавиш, нажатие кнопки **Clear** приводит к очистке его содержимого (рис. 5.6).

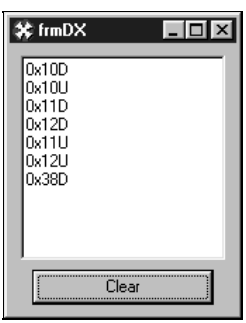


Рис. 5.6. Первый пример использования библиотеки DirectInput

В списке `uses` помимо обычных для Delphi модулей мною вписан `DirectInput8`.

Глобальная переменная `DInput` обеспечивает доступ к функциям `DirectInput`:

```
var
  DInput : IDIRECTINPUT8 = nil;           // Главный объект DirectInput
  // Интерфейс доступа к устройству ввода
  DIKeyboard : IDIRECTINPUTDEVICE8 = nil;
```

Примечание

Впервые в наших примерах мы обращаемся к интерфейсам именно восьмой версии DirectX. Обращу внимание на это событие, чтобы оно не прошло для вас незамеченным.

Следующая пользовательская функция предназначена для подготовки работы (обработку ошибок оставлю только для первого действия):

```
function TfrmDX.OnCreateDevice : HRESULT;
var
  hRet : HRESULT;       // Результат действий
  dipdw : TDIPROPDWORD; // Вспомогательная структура, задание параметров
begin
  // Создание главного объекта DirectInput
  hRet := DirectInput8Create (hInstance, DIRECTINPUT_VERSION,
                              IID_IDirectInput8, DInput, nil);
  if Failed (hRet) then begin
     Result := hRet;
     Exit
  end;
  // Создание объекта ввода информации от клавиатуры
  hRet := DInput.CreateDevice (GUID_SysKeyboard, DIKeyboard, nil);
  // Задаем формат данных, получаемых от устройства
  hRet := DIKeyboard.SetDataFormat(c_dfDIKeyboard);
  // Задаем уровень кооперации
  hRet := DIKeyboard.SetCooperativeLevel(Handle, DISCL_NONEXCLUSIVE or
                                         DISCL_BACKGROUND);
  // Параметры для буферной схемы получения данных
  ZeroMemory (@dipdw, SizeOf (dipdw));
  with dipdw do begin
    diph.dwSize       := SizeOf(TDIPROPDWORD);
    diph.dwHeaderSize := SizeOf(TDIPROPHEADER);
    diph.dwObj        := 0;
    diph.dwHow        := DIPH_DEVICE;
    dwData            := SAMPLE_BUFFER_SIZE;
  end;
  // Задаем параметры буфера
  hRet := DIKeyboard.SetProperty(DIPROP_BUFFERSIZE, dipdw.diph);
  // Установили связь с устройством ввода
  Result := DIKeyboard.Acquire;
end;
```

Для создания главного объекта из библиотеки DirectInput должна использоваться функция `DirectInput8Create`. Аргументы ее таковы:

- указатель на вызывающий поток;
- версия DirectX, в которой создано приложение;
- идентификатор требуемого интерфейса;
- переменная, в которую помещается результат.

Последний аргумент — указатель на показатель агрегирования (разновидность наследования; термин, специфичный для COM) — обычно равен `nil`.

В случае удачи функция возвращает ноль. Такому значению соответствует константа `DI_OK`, определенная в модуле `DirectInput8`.

Метод `CreateDevice` главного объекта используется для создания нового объекта устройства. У этого метода три аргумента:

- идентификатор нужного устройства;
- переменная, в которую помещается результат;
- показатель агрегирования.

В качестве идентификатора для клавиатуры передаем константу `GUID_SysKeyboard`.

Перед захватом устройства необходимо вызвать метод `SetDataFormat` объекта, связанного с устройством ввода. Здесь описывается формат, в котором вводимые данные возвращаются устройством. Для стандартного устройства задаем стандартный формат.

Также обязательным действием является определение степени контроля над устройством, задание уровня кооперации, другим словом. Для этого вызывается метод `SetCooperativeLevel`, первый аргумент которого — идентификатор окна приложения.

Прежде всего, необходимо указать, задается ли исключительный доступ к устройству или нет (флаги `DISCL_EXCLUSIVE` и `DISCL_NONEXCLUSIVE`). В этом примере устанавливаю неисключительный доступ. Для стандартного устройства разница между ними невелика, библиотека DirectInput не может позволить никакому приложению захватить клавиатуру монопольно. Просто эксклюзивный доступ может привести к помехам в работе с устройством других приложений.

Помимо эксклюзивности обязательно необходимо задать активность режима (указать один из флагов `DISCL_BACKGROUND` или `DISCL_FOREGROUND`). Первый флаг соответствует режиму, когда приложение имеет доступ к устройству ввода всегда, даже когда не имеет активности. Если вы запустите две копии этой программы, то обе они будут реагировать на нажатие клавиш, и по нажатии клавиши <Esc> завершат работу обе копии.

Следующие действия при инициализации связаны с выбранной схемой получения доступа к данным. Можно использовать данные двух видов: *непосредственные* (immediate) и *буферизованные* (buffered).

При работе с клавиатурой по первой схеме приложение периодически опрашивает клавиши, получая данные о каждой из них: нажата она или нет. Вторая схема состоит в том, что приложение считывает буфер, в котором хранятся данные о произошедших со времени последнего опроса событиях,

связанных с устройством: какие клавиши были нажаты, какие были отпущены.

Наш пример позволяет применить обе схемы, но первоначально настроен на вторую, буферизованную, схему. Для нее надо задать размер буфера, и поэтому используется вспомогательная структура, передающаяся аргументом метода `SetProperty`. Размер буфера мы задаем равным значению константы проекта:

```
const
  SAMPLE_BUFFER_SIZE = 8;
```

Запомните, что для схемы непосредственного опроса эти действия не нужны.

Заканчивается код инициализации захватом устройства, получением доступа к нему, вызовом метода `Acquire` объекта, связанного с устройством. Теперь мы можем получать данные с устройства, если оно доступно и все подготовительные шаги были успешны.

Вызывается код инициализации при создании формы, в случае неудачи выводится сообщение:

```
procedure TfrmDX.FormCreate(Sender: TObject);
var
  hRet : HRESULT;
begin
  hRet := OnCreateDevice;          // Инициализация устройства
  if Failed (hRet) then MessageDlg(DIErrorString(Error), mtError,
                                   [mbAbort], 0);
end;
```

Ошибки возможны при неверном указании параметров, также они появляются при занятости устройства. Если сейчас запущено приложение, имеющее исключительный доступ к клавиатуре, то у нас могут возникнуть проблемы с захватом устройства. В этой ситуации следует вызывать метод `Acquire` до тех пор, пока не будет установлена связь.

В нашем примере, после установления связи с устройством происходит беспрерывный вызов функции чтения буферизованных данных:

```
function TfrmDX.ReadBufferedData : HRESULT;
var
  didod : Array [0..SAMPLE_BUFFER_SIZE – 1] of TDIDEVICEOBJECTDATA;
  dwElements : DWORD;
  i : DWORD;
  hRet : HRESULT;
  s : String;
begin
  if DIKeyboard = nil then begin
```

```
    Result := DI_OK;
    Exit
  end;
  // Считываем данные из буфера
  hRet := DIKeyboard.GetDeviceData (SizeOf(TDIDEVICEOBJECTDATA),
                                    @didod, dwElements, 0);
  if Failed (hRet) then begin            // Восстанавливаем связь
    hRet := DIKeyboard.Acquire;
      while hRet = DIERR_INPUTLOST do
          hRet := DIKeyboard.Acquire;
  end;
  // Буфер не пустой
  if dwElements <> 0 then
    for i := 0 to dwElements - 1 do begin
     if didod[i].dwData and $80 <> 0    // Клавиша нажата
        then s := 'D'
        else s := 'U';
     Memo1.Lines.Add (Format ('0x%02x%s', [didod[i].dwOfs, s]));
     if didod[i].dwOfs = DIK_ESCAPE then Close;
  end;
  Result := DI_OK;           // Нулевое значение, признак успешности
end;
```

Метод `GetDeviceData` объекта, ассоциированного с устройством, позволяет осуществить собственно считывание данных из буфера. Смысл первого аргумента прозрачен: это размер структуры, предназначенной для хранения. Второй аргумент — указатель на массив элементов данной структуры. В качестве значения третьего аргумента устанавливается количество считанных из буфера данных. Последний аргумент может быть нулем или константой `DIGDD_PEEK` (во втором случае буфер не будет очищаться после считывания данных).

Если функция возвращает ненулевое значение, то, скорее всего, потеряна связь с устройством. Тогда необходимо снова установить эту связь, вызвав метод `Acquire`. В библиотеке DirectInput отсутствуют какие-либо специальные методы восстановления, а устанавливать связь можно сколько угодно раз, т. к. лишние вызовы этой функции игнорируются.

Скан-коды клавиш содержатся в поле `dwOfs` структуры `TDIDEVICEOBJECTDATA`, значение поля `dwData` позволяет узнать, какое событие произошло, нажата ли клавиша или отпущена. Если это значение равно 128, то клавиша опущена. В нашем примере к коду клавиши в этом случае приписывается буква "D", иначе — "U".

Вам не обязательно помнить наизусть коды всех клавиш, можете пользоваться символическими константами. Для примера я показал, как выделить нажатие клавиши <Esc>.

После завершения работы освобождаем устройство и память, занятую объектами:

```
procedure TfrmDX.FormDestroy(Sender: TObject);
begin
  if Assigned (DIKeyboard) then DIKeyboard.Unacquire; // Завершить диалог
  if Assigned (DIKeyboard) then DIKeyboard := nil;
  if Assigned (DInput) then DInput := nil;
end;
```

Поработайте с примером и обратите внимание, что можно отследить состояние максимум четырех клавиш одновременно.

Непосредственная схема работы с клавиатурой используется чаще, чем буферизованная, напоминаю, что состоит она в том, что в необходимые моменты происходит опрос всех клавиш. Удалите из кода обработчика `OnIdle` вызов процедуры буферного опроса клавиатуры и снимите комментарий со следующей далее строки. В коде инициализации удалите все, связанное с заданием размера буфера. Запустите проект и нажмите несколько клавиш (тоже максимум четыре) одновременно, в `Memo` выведутся коды всех нажатых клавиш:

```
function TfrmDX.ReadImmediateData : HRESULT;
var
  hRet : HRESULT;
  diks : Array [0..255] of BYTE;     // Массив состояния клавиатуры
  i : Integer;
  sMulti : String;
begin
  if DIKeyboard = nil then begin
     Result := DI_OK;
     Exit
  end;
  ZeroMemory(@diks, SizeOf(diks));   // Подготавливаем массив
  hRet := DIKeyboard.GetDeviceState(SizeOf(diks), @diks); // Заполняем
  if Failed (hRet) then begin        // Требуется восстановить связь
    hRet := DIKeyboard.Acquire;
      while hRet = DIERR_INPUTLOST do
        hRet := DIKeyboard.Acquire;
  end;
  sMulti := '';
  for i := 0 to 255 do               // Вывод кодов нажатых клавиш
    if diks[i] and $80 <> 0
      then sMulti := sMulti + ' ' + Format ('0x%02x', [i]);
  Memo1.Lines.Add (sMulti);
  Result := DI_OK;
end;
```

Непосредственная схема основана на использовании метода GetDeviceState, по вызову которого массив заполняется данными о состоянии клавиш, точно также здесь возможны значения 0 и 128.

В примере происходит опрос состояния всех клавиш, что не обязательно делать, если вас интересуют только некоторые из них. Например, если требуется осуществить выход по нажатии клавиши <Esc>, можно не пробегать в цикле по всем элементам массива, а обратиться только к единственному:

```
if diks [DIK_ESCAPE] = 128 then Close;
```

С помощью непосредственной схемы доступа легко обрабатывать нажатие нескольких клавиш одновременно, чем и пользуются часто в играх, как, например, в очередном примере — проекте каталога Ex06. От предыдущего варианта нашей тестовой игры пример отличается только тем, что здесь для управления используется библиотека DirectInput. Скорость ввода стала чрезвычайно стремительной, воин теперь резво передвигается по нажатии клавиш, буквально быстрее пули. Хотя шаг его нисколько не увеличился. Одним махом происходит обработка нескольких клавиш, и можно, например, стрелять вверх и вбок одновременно, или двигаться вправо и стрелять вверх. Пули тоже вылетают на порядок быстрее, и ограничение в сотню расходуется в считанные секунды.

Принципиально обработка ввода ничем не отличается от первого примера на основе библиотеки DirectInput. Здесь используется непосредственная схема доступа, только уровень доступа устанавливается в комбинацию DISCL_FOREGROUND or DISCL_EXCLUSIVE. Несмотря на негласное соглашение, что неактивное приложение не будет считывать данные с клавиатуры, при запуске двух копий программы обе они сильно замедлятся.

Работа с мышью

После знакомства с возможностями ввода с клавиатуры нам будет легко научиться работать с мышью, принципы обработки ввода здесь точно такие же. Непосредственную схему доступа изучим на конкретном примере — проекте каталога Ex07. Это еще один вариант создания эффекта лупы, но более эффектный, чем предыдущие, поскольку здесь добавлены сферические искажения пикселов (рис. 5.7).

Представленные ниже константы и переменные связаны с параметрами искажений:

```
const
  Diameter   = 180;    // Задает максимальный размер лупы
  Scale     = 35;      // Вспомогательный коэффициент
var
  Radius  : Integer = Diameter div 2;  // Текущий размер лупы
  SqrRad  : Integer;                   // Вспомогательные величины
  Sphere  : Integer;
```

Рис. 5.7. Пример иллюстрирует работу с мышью и создание сферических искажений

Вспомогательные переменные заполняются первоначально при создании формы:

```
SqrRad  := Radius * Radius;           // Квадрат радиуса
Sphere  := (Radius * Radius) - (Scale * Scale);   // Искажение
```

Во время перерисовки кадра накладываем фон, а искажения вносим сразу на поверхность заднего буфера:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  hRet : HRESULT;
begin
  // Блиттинг фона
  hRet := FDDSBack.BltFast (0, 0, FDDSBackGround, nil, DDBLTFAST_WAIT);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := Zoom;            // Вызов функции создания эффекта
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  Result := FlipPages;     // Переключение буферов
end;
```

Эффект построен на простейшей математике — уравнениях круга и сферы:

```
function TfrmDD.Zoom : HRESULT;
var
  desc1 : TDDSURFACEDESC2;
  desc2 : TDDSURFACEDESC2;
  X, Y : Integer;
  XX,YY,YYXX : Integer;
  mz : Single;
  hRet : HRESULT;
begin
  ZeroMemory (@desc1, SizeOf(desc1));
  desc1.dwSize := SizeOf (desc1);
  ZeroMemory (@desc2, SizeOf(desc2));
  desc2.dwSize := SizeOf (desc2);
  hRet := FDDSBack.Lock (nil, desc1, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := FDDSBackGround.Lock (nil, desc2, DDLOCK_WAIT, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  for Y := -Radius to Radius do begin
    YY := Y * Y;
    for X := -Radius to Radius do begin
      XX := X * X;
      YYXX := YY + XX;
      if YYXX < Sphere then begin           // Точка внутри круга
        mz := Scale / sqrt(SqrRad - YYXX);  // Масштаб по третьей оси
       // Пиксел на задней поверхности
       PWord (Integer(desc1.lpSurface) + (Y + mouseY) * desc1.lPitch +
             (mouseX + x) * 2)^ :=
       // Источник на поверхности фона
              PWord (Integer(desc2.lpSurface) +
              trunc (mz * Y + mouseY) * desc2.lPitch +
              trunc (mz * X + mouseX) * 2)^;
      end;
    end;
  end;
  FDDSBackGround.Unlock (nil);
  FDDSBack.Unlock (nil);
  Result := DD_OK;
end;
```

Для работы с устройством введены переменные уже знакомых нам типов:

```
DInput : IDIRECTINPUT8 = nil;
DIMouse : IDIRECTINPUTDEVICE8 = nil;
```

В коде подготовки устройства выполняются действия, аналогичные работе с клавиатурой, лишь поменялись константы:

```
function TfrmDD.OnCreateDevice : HRESULT;
var
  hRet : HRESULT;
begin
  hRet := DirectInput8Create (hInstance, DIRECTINPUT_VERSION,
                              IID_IDirectInput8, DInput, nil);
  // GUID соответствует устройству "мышь"
  hRet := DInput.CreateDevice (GUID_SysMouse, DIMouse, nil);
  hRet := DIMouse.SetDataFormat(c_dfDIMouse2);  // Задаем формат данных
  // Уровень кооперации задаем обычный
  hRet := DIMouse.SetCooperativeLevel(Handle, DISCL_NONEXCLUSIVE or
                                      DISCL_BACKGROUND);
  Result := DIMouse.Acquire;                    // Захватываем устройство
end;
```

Опрос состояния мыши происходит непрерывно, перед каждым обновлением кадра:

```
procedure TfrmDD.ApplicationEvents1Idle(Sender: TObject;
                                        var Done: Boolean);
begin
  if FActive then begin
     ReadImmediateData;     // Ошибки игнорируем
     if Failed (UpdateFrame) then RestoreAll;
  end;
  Done := False;
end;
```

При непосредственном доступе к мыши мы получаем данные о приращениях по осям, а не о координатах курсора на экране. При удерживаемой левой кнопки мыши радиус лупы увеличивается, в то время как правая кнопка позволяет его уменьшить:

```
function TfrmDD.ReadImmediateData : HRESULT;
var
  hRet : HRESULT;
  dims2 : TDIMOUSESTATE2;  // Структура хранения вводимых данных
begin
  ZeroMemory(@dims2, SizeOf(dims2));
```

```
  // Получаем сведения о состоянии мыши
  hRet := DIMouse.GetDeviceState(SizeOf(TDIMOUSESTATE2), @dims2);
  if Failed (hRet) then begin    // Связь потеряна
    hRet := DIMouse.Acquire;     // Устанавливаем связь заново
    while hRet = DIERR_INPUTLOST do hRet := DIMouse.Acquire;
  end;
  // Массив rgbButtons хранит состояние для каждой кнопки мыши
  if dims2.rgbButtons[0] = 128 then begin  // Нажата левая кнопка
    Radius := Radius + 1; // Радиус увеличивается до некоторых пределов
    if Radius > Diameter then Radius := Diameter;
    SqrRad  := Radius * Radius;
    Sphere  := (Radius * Radius) - (Scale * Scale);
  end;
  if dims2.rgbButtons[1] = 128 then begin // Нажата правая кнопка
    Radius := Radius - 1;                 // Радиус уменьшается
    if Radius < 0 then Radius := 0;
    SqrRad  := Radius * Radius;
    Sphere  := (Radius * Radius) - (Scale * Scale);
  end;
  // Полученное реальное приращение умножаем
  mouseX := mouseX + 2 * dims2.lX;
  if mouseX < Radius then mouseX := Radius else
  if mouseX > ScreenWidth - Radius then mouseX := ScreenWidth - Radius;
  mouseY := mouseY + 2 * dims2.lY;
  if mouseY < Radius then mouseY := Radius else
  if mouseY > ScreenHeight - Radius then mouseY := ScreenHeight - Radius;
  Result := DI_OK;
end;
```

Вывод текста

Текст можно выводить двумя способами: используя функции GDI и осуществляя блиттинг растров отдельных букв. Первый способ мы применяли неоднократно в предыдущих примерах. Рассмотрим второй.

В качестве примера я приготовил простую программу изучения английского языка. Один из методов пополнения словарного запаса состоит в том, чтобы выводить на экран строки словаря на очень маленький промежуток времени, меньший 1/24 секунды. Считается, что выводимый "в 25-м кадре" текст запоминается зрителем на подсознательном уровне. Метод не требует особых усилий от обучаемого, но я не могу сказать ничего определенного по поводу его реальной эффективности, и замечу, что применяться он должен только при условии, что пользователь информирован о работе подобных программ.

Программа проекта каталога Ex08 как раз относится к разряду подобных. После ее запуска можете выполнять текущую работу и заодно обогащать свой словарный запас.

Я подготовил небольшой файл словаря, на основе которого заполняется массив строк:

```
const
  imageBmp      = '..\font.bmp';     // Растр шрифта
  NumbLines = 70;                    // Количество строк в файле
  FileName = 'dictionary.txt';       // Файл словаря
  Delay = 50;                 // Пауза между появлениями очередной фразы
var
  OutLiteral : String;           // Очередная выводимая строка
  StrList : Array [0..NumbLines - 1] of String; // Массив строк словаря
  WinWidth, PosX  : Integer;     // Размеры экрана и позиция строки по X
  WinHeight, PosY : Integer;     // Размеры экрана и позиция строки по Y
  tmpRect : TRECT;         // Прямоугольник, связанный с текущей строкой
```

Избранные символы, с кодом большим 31, нарисованы в растре шрифта, высота каждого символа — 15 пикселов (рис. 5.8).

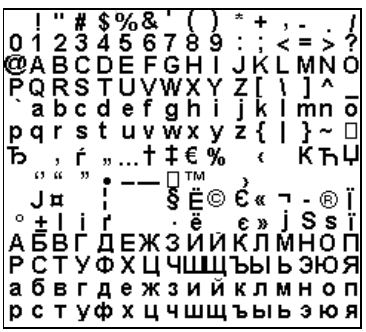


Рис. 5.8. В этой задаче не потребуются все 255 символов

Используется нормальный уровень кооперации. Для создания вспомогательной поверхности определяем текущие установки экрана:

```
procedure TfrmDD.FormCreate(Sender: TObject);
var
  hRet : HRESULT;
  ddsd : TDDSurfaceDesc2;
  t : TextFile;
  i, maxLength : Integer;
begin
  FDDSWork := nil;
  FDDSGround := nil;
  FDDSFont := nil;
  FDDSPrimary := nil;
```

```
FDD := nil;
hRet := DirectDrawCreateEx (nil, FDD, IDirectDraw7, nil);
if Failed(hRet) then ErrorOut(hRet, 'DirectDrawCreateEx');
// Уровень кооперации — нормальный
hRet := FDD.SetCooperativeLevel(Handle, DDSCL_NORMAL);
if Failed(hRet) then ErrorOut(hRet, 'SetCooperativeLevel');
ZeroMemory(@ddsd, SizeOf(ddsd));
with ddsd do begin
  dwSize := SizeOf(ddsd);
  dwFlags := DDSD_CAPS;
  ddsCaps.dwCaps := DDSCAPS_PRIMARYSURFACE;
end;
hRet := FDD.CreateSurface(ddsd, FDDSPrimary, nil);
if Failed(hRet) then ErrorOut(hRet, 'Create Primary Surface');
// Загружаем растр со шрифтом
FDDSFont := DDLoadBitmap(FDD, imageBmp, 0, 0);
if FDDSFont = nil then ErrorOut(hRet, 'DDLoadBitmap');
// Узнаем текущие размеры экрана
WinWidth  := GetSystemMetrics(SM_CXSCREEN);
WinHeight := GetSystemMetrics(SM_CYSCREEN);
// Поверхность для запоминания подложки выводимой фразы
ZeroMemory(@ddsd, SizeOf(ddsd));
with ddsd do begin
  dwSize := SizeOf(ddsd);
  dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
  ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
  dwWidth := WinWidth;
  dwHeight := WinHeight;
end;
hRet := FDD.CreateSurface(ddsd, FDDSGround, nil);
if Failed (hRet) then ErrorOut(hRet, 'CreateSurface');
// Считываем файл словаря, находим длину самой длинной фразы
AssignFile (t, FileName);
Reset (t);
maxLength := 0;
for i := 0 to NumbLines — 1 do begin
  ReadLn (t, StrList [i]);
  if length (StrList [i]) > maxLength then maxLength :=
                                          length (StrList [i]);
end;
CloseFile (t);
// Поверхность для хранения растра фразы
ZeroMemory(@ddsd, SizeOf(ddsd));
with ddsd do begin
  dwSize := SizeOf(ddsd);
  dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
```

```
    ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
    dwWidth := maxLength * 15;   // Должны вместиться все фразы
    dwHeight := 15;
  end;
  hRet := FDD.CreateSurface(ddsd, FDDSWork, nil);
  if Failed (hRet) then ErrorOut(hRet, 'CreateSurface');
  Randomize;
  OutLiteral := StrList [random (NumbLines)];  // Генерируем первую фразу
  GeneratePos;             // Случайно генерируем позицию фразы на экране
  LastTickCount := GetTickCount;
end;
```

Для обеспечения максимальной скорости ошибки вообще не обрабатываются. Фразы побуквенно выводятся на вспомогательную поверхность, чтобы затем на первичной поверхности отобразить всю строку целиком:

```
procedure TfrmDD.ApplicationEvents1Idle(Sender: TObject;
  var Done: Boolean);
var
  rcRect : TRECT;
  i, X, Y : Integer;
// Вывод одного символа на вспомогательную поверхность
procedure OutChar (ch : Char; PosX : Integer);
var
  chRect : TRECT;
  wrkI : integer;
begin
  // В растре шрифта представлены символы, начиная с пробела
  wrkI := ord (ch) - 32;
  chRect.Left := wrkI mod 16 * 15; // Прямоугольник буквы в растре шрифта
  chRect.Top := wrkI div 16 * 15;
  chRect.Right := chRect.Left + 15;
  chRect.Bottom := chRect.Top + 15;
  // Вывод буквы на вспомогательную поверхность
  FDDSWork.BltFast(PosX, 0, FDDSFont, @chRect, DDBLTFAST_DONOTWAIT);
end;
begin
  ThisTickCount := GetTickCount;
  Done := False;
  // Подошло время выводить очередную строку словаря
  if (ThisTickCount - LastTickCount) < Delay then Exit;
  // Ограничивающий  прямоугольник
SetRect (rcRect, PosX, PosY, PosX + length (OutLiteral) * 15, PosY + 15);
  // Запоминаем, что на экране находится в этом прямоугольнике
  FDDSGround.BltFast(PosX, PosY, FDDSPrimary, @rcRect, DDBLTFAST_WAIT);
  // Вывод строки
  FDDSPrimary.BltFast(PosX, PosY, FDDSWork, @tmpRect, DDBLTFAST_WAIT);
```

```
  // Запоминаем текущее положение строки
  X := PosX;
  Y := PosY;
  OutLiteral := StrList [random (NumbLines)];  // Генерация новой строки
  GeneratePos;              // Генерируем позицию на экране новой строки
  // Подготавливаем поверхность новой строки
  for i := 1 to length (OutLiteral) do
    OutChar (OutLiteral [i], (i - 1) * 15);
  SetRect (tmpRect, 0, 0, length (OutLiteral) * 15, 15);
  // Стираем старую фразу на экране
  FDDSPrimary.BltFast(X, Y, FDDSGround, @rcRect, DDBLTFAST_WAIT);
  LastTickCount := GetTickCount;
end;
```

Итак, фраза на экране присутствует, пока выполняется код подготовки новой строки. Это очень малый промежуток времени. Конечно, некоторые строки будут потеряны, появившись и исчезнув быстрее, чем произошло обновление экрана. Для замедления процесса можно вставить вызов системной функции `Sleep` с небольшой задержкой, но для небыстрых компьютеров это может привести к тому, что строки начнут неприятно мерцать по всему экрану.

Создание консоли

Консоль вы часто видели и использовали в профессиональных играх и, наверняка, захотите создать и в своей игре. Пример данного раздела — проект каталога Ex09 — поможет вам в этом. Он является развитием нашей пробной игры: теперь по нажатии клавиши <Tab> на экране появляется консоль, предназначенная для ввода команд (рис. 5.9).

Я предусмотрел реакцию только на одну команду, после ввода `Exit` приложение завершает работу, все остальные вводимые строки просто вызывают эхо[1] в консоли.

Моя консоль вмещает три строки, инициализируемые многозначительными фразами:

```
rcRectConsole : TRECT;              // Вспомогательный прямоугольник
ConsoleHeight : Integer = 0;        // Текущий размер консоли
ConsoleLive : BOOL = False;         // Флаг, связанный с присутствием
TextConsole1 : String = '> Initialization....OK'; // Строки вывода
TextConsole2 : String = '> Loading...........OK';
TextConsole3 : String = '>_';
```

[1] То есть вывод команды. — *Ред.*

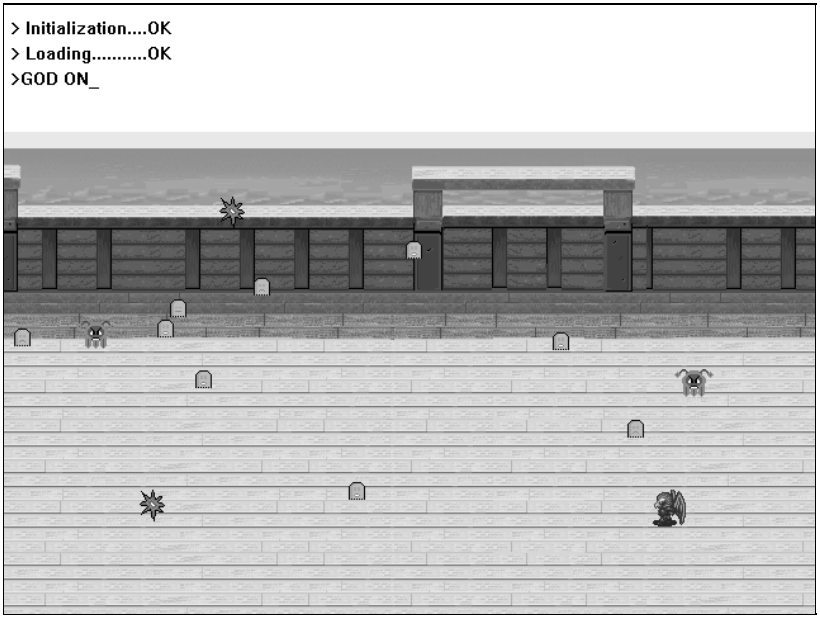


Рис. 5.9. Наша игра обзавелась консолью

Для функционирования консоли я завел отдельную поверхность, закрашиваемую при инициализации белым цветом:

```
ZeroMemory(@ddsd, SizeOf(ddsd));
with ddsd do begin
  dwSize := SizeOf(ddsd);
  dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
  ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
  dwWidth := 640;
  dwHeight := 100;
end;
hRet := FDD.CreateSurface(ddsd, FDDSConsole, nil);
if Failed (hRet) then ErrorOut(hRet, 'CreateSurface');

hRet := FDDSConsole.SetPalette(FDDPal);
if Failed (hRet) then ErrorOut(hRet, 'SetPalette');

ZeroMemory(@ddbltfx, SizeOf(ddbltfx));
ddbltfx.dwSize := SizeOf(ddbltfx);
ddbltfx.dwFillColor := RGB (255, 255, 255);
FDDSConsole.Blt(nil, nil, nil, DDBLT_COLORFILL or DDBLT_WAIT, @ddbltfx);

SetRect (rcRectConsole, 0, 0, 640, 100);
```

Код воспроизведения монстров и пуль немного подправил, чтобы они не появлялись в области консоли. Сама же консоль воспроизводится в последнюю очередь, непосредственно перед переключением буферов:

```
if ConsoleLive then begin   // Надо ли рисовать консоль
  if (GlobalThisTickCount – GlobalLastTickCount > DelayConsole) then
    begin                   // Плавное появление консоли
      Inc (ConsoleHeight, 5);
      if ConsoleHeight > 100 then ConsoleHeight := 100;
      SetRect (rcRectConsole, 0, 0, 640, ConsoleHeight);
    end;
    // Собственно воспроизведение консоли
    FDDSBack.BltFast(0, 0, FDDSConsole, @rcRectConsole, DDBLTFAST_WAIT);
end;
```

Текст в консоли выводится с помощью функций GDI:

```
procedure OutText (const X, Y : Integer; const TextCon : String);
var
  DC : HDC;
begin
  FDDSConsole.GetDC (DC);
  SetBkColor(DC, RGB (255, 255, 255));   // Цвета фона и букв необходимо
  SetTextColor (DC, 0);                  // задавать обязательно
  TextOut (DC, X, Y, PChar(TextCon), length (TextCon));
  FDDSConsole.ReleaseDC (DC);
end;
```

Немало хлопот принесла обработка нажатия клавиши <Backspace>: чтобы стереть старый текст, приходится воспроизводить ряд пробелов:

```
if diks [DIK_TAB] and $80 <> 0 then begin   // Клавиша <Tab>
  if not ConsoleLive then begin             // Включение консоли
    ConsoleHeight := 0;
    ConsoleLive := True;
  end
  else ConsoleLive := False;               // Выключить консоль
  Sleep(100);                              // Небольшая пауза
end;
if ConsoleLive then begin        // Обработка клавиш для консоли
  OutText (5, 10, TextConsole1);   // Вывод трех строк в консоли
  OutText (5, 30, TextConsole2);
  OutText (5, 50, TextConsole3);
  if diks [DIK_RETURN] and $80 <> 0 then begin   // Ввод команды
    // Введена команда "Exit"; выход из программы
    if (TextConsole3 = '>EXIT_') or (TextConsole3 = '> EXIT_')
      then Close;
```

```
    // Введена другая команда, строки стираются и поднимаются наверх
    TextConsole1 := '                                                    ';
    OutText (5, 10, TextConsole1);   // Затираем пробелами
    TextConsole1 := TextConsole2;    // Строка сдвигается вверх
    TextConsole2 := '                                                    ';
    OutText (5, 30, TextConsole2);
    TextConsole2 := '> Command : ' + Copy (TextConsole3, 2,
    length (TextConsole3) — 2);     // Реакция на все остальные команды -
                                    // вывод эха
    TextConsole3 := '                                                    ';
    OutText (5, 50, TextConsole3);
    TextConsole3 := '>_';  // Последняя строка превратилась в приглашение
    Sleep(100);
  end;
  if diks [DIK_BACKSPACE] and $80 <> 0 then begin   // Нажата клавиша
                                                    // <Backspace>
  TextConsole3 := '                                                  ';
  OutText (5, 50, TextConsole3);    // Стираем последнюю строку
  TextConsole3 := '>_';
  OutText (5, 50, TextConsole3);
end;
for i := DIK_Q to DIK_M do            // Просматриваем буквенные клавиши
  if diks [i] and $80 <> 0 then begin // Нажата какая-то клавиша с буквой
    if length (TextConsole3) < 20 then begin  // Ограничение длины строки
    // Перед символом подчеркивания вставляем букву нажатой клавиши
      TextConsole3 := Copy (TextConsole3, 1, length (TextConsole3) — 1) +
                      ScanToChar (i) +'_';
      OutText (5, 50, TextConsole3);  // Вывод получившейся строки
      Sleep(100);
      end;
    end;
  end;
```

Поскольку обработка клавиатуры происходит необычайно быстро, с помощью процедуры `Sleep` создаем искусственную паузу, чтобы не получить эффекта залипания клавиши. Эта пауза дает время пользователю отпустить нажатую клавишу.

Диалоговые окна

Многие приложения нуждаются в диалоговых окнах, поэтому уделим немного внимания этому вопросу. Пример данного раздела (проект каталога Ex10) представляет собой окончательную реализацию нашего хранителя экрана с плавающими рыбками. В развитие предыдущего состояния добавлена поддержка пароля для входа в систему.

Установка и запрос пароля хранителя экрана являются системными действиями, но вызов их не ограничивается одной строкой. Это означает, что диалоги установки и ввода пароля не должны реализовываться программистом, пароль назначается для всех хранителей экранов. Мы не можем самостоятельно запросить пароль, хранить его в реестре, в определенном разделе, и самостоятельно организовывать ввод пароля при нажатии клавиши или движении курсора мыши.

Следующая процедура предназначена для вызова системного диалога задания нового пароля:

```
procedure TfrmDD.RunSetPassword;
type    // Специальный тип функции, используется только в этой ситуации
  TPCPAFunc = function(A : PChar; Parent : hWnd; B, C : Integer) :
              Integer; stdcall;
var
  Lib : THandle;          // Ссылка на DLL
  PCPAFunc : TPCPAFunc;   // Загружаемая функция
begin
  Lib := LoadLibrary('MPR.DLL');   // Динамическая загрузка DLL
  if Lib > 32 then begin           // Проверка успешности загрузки
    // Получаем адрес точки входа нужной функции
    @PCPAFunc := GetProcAddress(Lib, 'PwdChangePasswordA');
    // Задаем пароль хранителей экрана
    if @PCPAFunc <> nil then PCPAFunc('SCRSAVE', StrToInt(ParamStr(2)),
       0, 0);
    FreeLibrary(Lib);    // Выгружаем библиотеку
  end;
end;
```

В нашей программе эта процедура вызывается, если приложение запущено с параметром `/a`, т. е. в ситуации, когда пользователь нажал кнопку **Изменить** на вкладке **Заставка** (см. рис. 4.3).

При нажатии клавиши или движении курсора программа должна сама определить, установлен ли пароль для хранителя экрана, и запустить системный диалог ввода пароля:

```
function TfrmDD.TestPassword : BOOL;
type
  // Специальный тип, тоже используется только в этом, особом случае
  TVSSPFunc = function(Parent : hWnd) : BOOL; stdcall;
var
  Key   : hKey;
  D1,D2 : Integer;
  Value : Integer;
  Lib   : THandle;
  VSSPFunc : TVSSPFunc;
```

```
begin
  Result := True;
  // Загружаем информацию из реестра, используя функции API
  if RegOpenKeyEx(hKey_Current_User,'Control Panel\Desktop', 0,
    Key_Read, Key) = Error_Success then begin
    D2 := SizeOf(Value);
    // Определяем, установлен ли пароль
    if RegQueryValueEx(Key, 'ScreenSaveUsePassword', nil, @D1,
      @Value,@D2) = Error_Success then begin
      if Value <> 0 then begin
        // Динамически загружаем библиотеку ввода пароля
        Lib := LoadLibrary('PASSWORD.CPL');
        if Lib > 32 then begin
        // Получаем адрес точки входа
          @VSSPFunc := GetProcAddress(Lib, 'VerifyScreenSavePwd');
          // На время работы диалога включаем курсор
          ShowCursor(True);
          // Запускаем системный диалог
          if @VSSPFunc <> nil then Result := VSSPFunc(Handle);
          ShowCursor(False);   // Это можно, в принципе, не делать
          FreeLibrary(Lib);    // Освобождаем память
        end;
      end;
    end;
    RegCloseKey(Key);
  end;
end;
```

И теперь самое главное: диалоговое окно должно работать "поверх" первичной поверхности (рис. 5.10).

Чтобы пользователь увидел его, перед вызовом нашей пользовательской функции `TestPassword` нужно переключиться на воспроизведение в режиме GDI:

```
FDD.FlipToGDISurface;
```

То есть в такой ситуации обязан вызываться метод главного объекта `FlipToGDISurface`, а перерисовка экрана не должна осуществляться. К сожалению, мне встречались хранители экрана, написанные профессионалами, авторы которых не позаботились о корректной работе системного диалога: пароль приходится вводить "вслепую", не видя окно ввода пароля, закрытое картинкой первичной поверхности. Памятуя об этом, я многократно проверял работу своего хранителя на самых разных видеокартах, и могу сказать, что не встретил ничего подобного.

Чтобы отключить клавиатуру, точнее, запретить работу комбинаций клавиш <Alt>+<Tab> и <Ctrl>+<Alt>+<Del>, на время работы приложения инфор-

мируем систему о том, что работает хранитель экрана, при запуске приложения выполняется следующая строка кода:

```
SystemParametersInfo(SPI_SCREENSAVERRUNNING, 1, nil, 0);
```

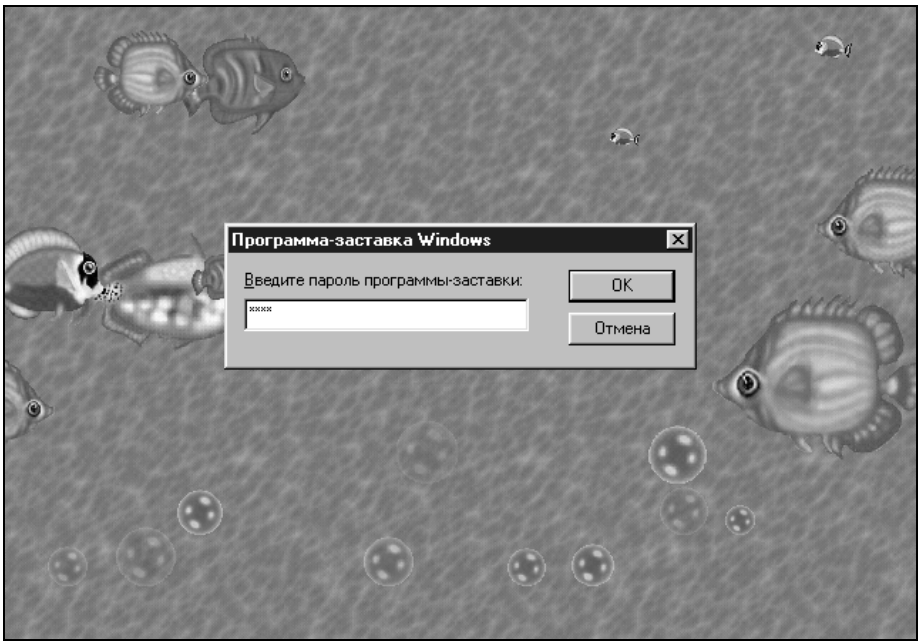


Рис. 5.10. В такой ситуации обычное окно должно всплыть перед нашим полноэкранным приложением

По окончании работы надо не забыть восстановить нормальную работу комбинаций этих клавиш, для чего вызывается та же команда, но второй аргумент задается нулевым значением.

Переключение на GDI-воспроизведение, надеюсь, сделает видимым диалоговое окно у каждого читателя книги, но полноценным такое решение назвать нельзя. При перемещении окна по экрану оно оставляет следы, расчищая первичную поверхность и обнажая окно приложения (поэтому экран становится серым).

Такой способ общения с пользователем ущербен еще по той причине, что он не работает в палитровом режиме. Из-за этого я не мог рекомендовать его для применения в функции вывода сообщения об ошибке, используемой нами в предыдущих примерах.

Учтите, что наш хранитель экрана требует небольших доработок, для оконного режима следует различать установленное разрешение рабочего стола, т. к. при 32-битной глубине возможно искаженное масштабирование образов.

Использование отсечения в полноэкранном приложении

Имея дело с Delphi, вам наверняка жалко будет потерять все ее прелести и мощь разработки диалоговых средств, расстаться с визуальными компонентами только потому, что при перемещении окна могут остаться серые пятна.

Специально для этого случая я подготовил пример, иллюстрирующий, как можно эффектно комбинировать обычный вывод средствами GDI и DirectDraw. Это проект каталога Ex11, где посередине экрана отображается репродукция самого популярного шедевра живописи, а по бокам разбросаны стандартные интерфейсные элементы (рис. 5.11).

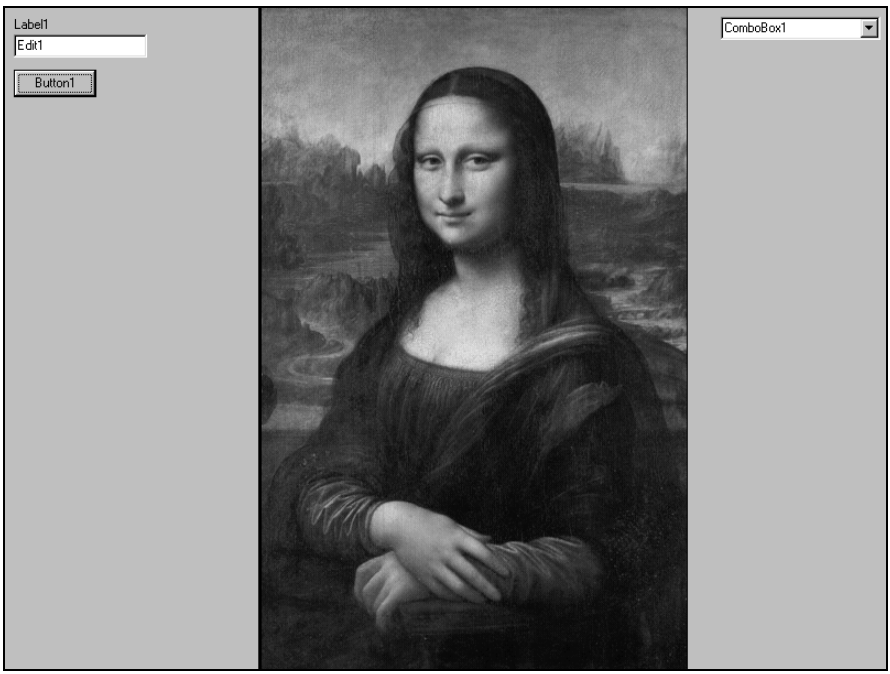


Рис. 5.11. Пример можно назвать шедевром, но не благодаря искусному коду

В программе задается режим 800×600 пикселов, уровень доступа — исключительный. При воспроизведении происходит блиттинг на первичную поверхность содержимого вспомогательной поверхности, а для того, чтобы воспроизведение осуществлялось только в пределах центральной области экрана, я использую отсечение, знакомое нам по оконным приложениям. Но теперь оно ограничивает строго определенную область экрана, а не связывается с окном приложения. Если ранее для связывания отсечения с окном использовался метод `SetHWnd`, то в этом примере вызывается метод

SetClipList. В общем случае последний метод может применяться для создания и непрямоугольных областей вывода, для описания их служат регионы:

```
var
  rgn : TRgnData; // Вспомогательная переменная, описывает набор регионов
  wrk : TRECT;    // Прямоугольник, описывающий наш единственный регион
...
  SetRect (wrk, 230, 0, 620, 600); // Задаем область вывода на экране
  with rgn.rdh do begin            // Заполняем поля структуры
    dwSize := SizeOf (RGNDATAHEADER); // Это обязательно, как всегда
    iType := RDH_RECTANGLES;       // Единственно возможное значение поля
    nCount := 1;                   // Количество задействованных регионов
    nRgnSize := Sizeof(TRECT);     // Размер единицы информации
  end;
  PRECT(@rgn.Buffer)^ := wrk;  // Заносим в буфер наш единственный регион
  if FDD.CreateClipper(0, FDDClipper, nil) = DD_OK then begin
    FDDClipper.SetClipList (@rgn, 0);  // Задаем область отсечения
    FDDSPrimary.SetClipper(FDDClipper);
  end;
```

Замечу, что вспомогательная структура, представленная здесь, является системной и не связана исключительно с DirectX. Заполняя поле buffer этой структуры, вы можете получить холсты замысловатой формы.

Приведу еще один способ работы с методом SetClipList. Вот код, который способствует отсечению, аналогичному отсечению предыдущего примера:

```
var
  hrg : HRGN;                                  // Регион
  rgnDataBuffer: Array [0..1023] of BYTE;   // Массив списка регионов
...
  hrg := CreateRectRgn (230, 0, 620, 600);  // Создание нужного региона
  // Заполняем массив данными
  GetRegionData(hrg, SizeOf(rgnDataBuffer), @rgnDataBuffer);
  DeleteObject(hrg);
  if FDD.CreateClipper(0, FDDClipper, nil) = DD_OK then begin
    FDDClipper.SetClipList (@rgnDataBuffer, 0);   // Задаем отсечение
    FDDSPrimary.SetClipper(FDDClipper);
  end;
```

Отсечение для полноэкранных приложений может использоваться и для того, чтобы полностью решить проблему с выводом образов вблизи границ. В проекте каталога Ex12 курсор заменен логотипом DirectX, динамически меняющим свой размер. Для такой ситуации задание положения курсора становится трудной задачей. Мы не можем пользоваться решениями предыдущих примеров с замененным курсором приложения. Присовокупление отсечения к экрану позволяет совершенно не задумываться о текущем размере образа курсора и его положении, при его воспроизведении вблизи границ образ отсекается совершенно корректно (рис. 5.12).

Рис. 5.12. Пример использования отсечения в полноэкранном приложении

Аналогично предыдущему примеру, объект отсечения строится на основе региона, но здесь регион представляет собой простой прямоугольник, связанный с размерами экрана:

```
SetRect (wrk, 0, 0, 800, 600);
with rgn.rdh do begin
  dwSize := SizeOf (RGNDATAHEADER);
  iType := RDH_RECTANGLES;
  nCount := 1;
  nRgnSize := Sizeof(TRECT);
end;
PRECT(@rgn.Buffer)^ := wrk;
if FDD.CreateClipper(0, FDDClipper, nil) = DD_OK then begin
  FDDClipper.SetClipList (@rgn, 0);
  FDDSBack.SetClipper(FDDClipper);
end;
```

При перерисовке кадра образ курсора растягивается на величину `Scale`:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  hRet : HRESULT;
  wrkRect : TRECT;
begin
  // Вывод фона
  hRet := FDDSBack.Blt (nil, FDDSBackGround, nil, DDBLT_WAIT, nil);
```

```
  if Failed (hRet) then begin
     Result := hRet;
     Exit;
  end;
  // Прямоугольник области образа курсора
  SetRect (wrkRect, mouseX, mouseY, mouseX + Scale, mouseY + Scale);
  // Масштабирование образа курсора, используется цветовой ключ
  hRet := FDDSBack.Blt (@wrkRect, FDDSImage, nil, DDBLT_WAIT or
                        DDBLT_KEYSRC, nil);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  Result := FDDSPrimary.Flip(nil, DDFLIP_WAIT)
end;
```

Библиотека CDX

Наверняка многие из читателей этой книги захотят создать собственную вспомогательную библиотеку, облегчающую программирование приложений на основе DirectDraw. Прежде чем приступать к этому мероприятию, стоит познакомиться с уже готовыми решениями. В данном разделе вас ждет краткий экскурс по функциям очень популярной библиотеки CDX, реализованной по лицензии GNU. Библиотека написана на C, и я перевел на Delphi лишь небольшую ее часть, а полностью библиотеку вы можете получить по адресу **http://www.cdx.sk/**.

Начнем с проекта каталога Ex13, простейшего примера, выводящего на экране по ходу своей работы разноцветные прямоугольники (рис. 5.13).

В списке `uses` добавилось подключение модуля `CDXScreenX`, а перечень переменных, связанных с графикой, совсем короткий:

```
GameScreen : CDXScreen;
```

Вообще, код сильно упрощается, а инициализация графики сведена к одному оператору:

```
GameScreen := CDXScreen.CreateCDXScreenCustomBPP(Handle, ScreenWidth,
                ScreenHeight, ScreenBitDepth);
```

Код перерисовки окна также состоит из одного действия:

```
GameScreen.GetAppFrontBuffer.Rect(random(ScreenWidth),
                                  random(ScreenHeight),
                                  random(ScreenWidth),
                                  random(ScreenHeight),
                                  random(254));
```

То есть, рисуем очередной прямоугольник прямо на переднем буфере.

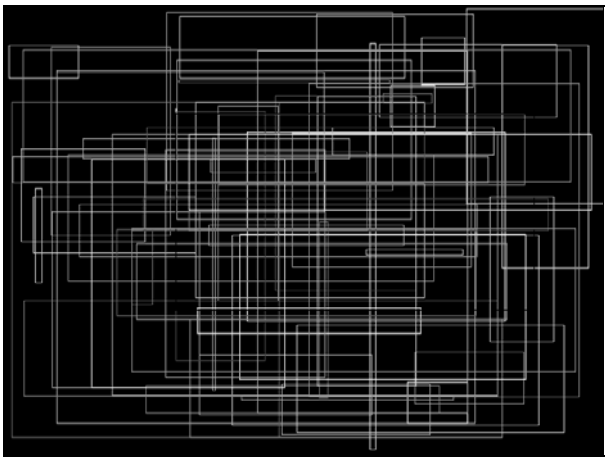


Рис. 5.13. Простейший пример на использование библиотеки CDX

Работа приложения завершается после нажатия любой клавиши. По окончании работы приложения удаляется наш единственный объект:

```
if Assigned (GameScreen) then GameScreen.Destroy;
```

Подход, предлагаемый CDX, вам сильно напомнит то, что мы встречали в реализации модуля `DDUtil` для восьмой версии DirectX, но исторически первой появилась CDX. Библиотека специально предназначена для разработки игр и охватывает не только DirectDraw, но и все остальные модули DirectX.

Следующий пример, проект каталога Ex14, иллюстрирует некоторые моменты работы с фоном. Экран разбит на четыре части, в каждой из которых выводится своя фоновая картинка (рис. 5.14).

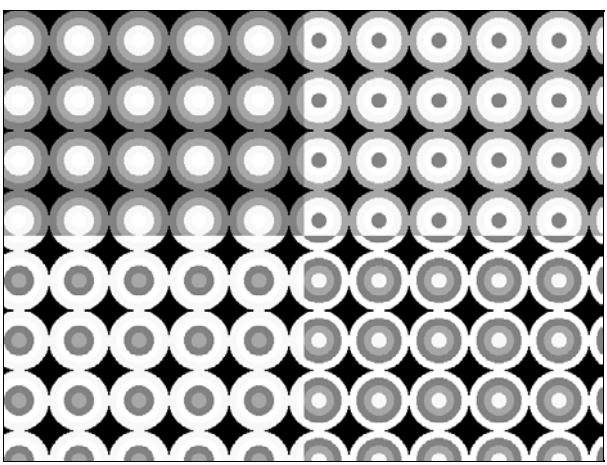


Рис. 5.14. Пример сложного заполнения фона

По нажатии клавиш управления курсором ландшафт в каждом секторе циклически сдвигается. В программе появились дополнительные переменные, связанные с фоном и обработкой клавиатуры:

```
GameInput : CDXInput;            // Объект, связанный с вводом
GameScreen : CDXScreen;          // Главный объект вывода
LandScape : CDXTile;             // Загружаемая картинка
Map1 : CDXMap;                   // Секторы, отдельные окна экрана
Map2 : CDXMap;
Map3 : CDXMap;
Map4 : CDXMap;
MapScrollSpeed : Integer = 4;    // Скорость передвижения фона
```

Загружаемая картинка объединяет четыре шаблона заполнения окна, каждый имеет размеры 64×64 пикселов:

```
procedure TfrmDD.FormCreate(Sender: TObject);
begin
  GameInput := CDXInput.CreateCDXInput;   // Инициализация ввода
  GameInput.Create(HInstance, Handle);
  GameScreen := CDXScreen.CreateCDXScreenCustomBPP(Handle, ScreenWidth,
                ScreenHeight, ScreenBitDepth);
  GameScreen.LoadPalette('Anim.bmp');    // Палитровый режим
  // Загрузка картинки с 4-мя шаблонами заполнения экрана, 64x64 пиксела
  LandScape := CDXTile.CDXTileCustom(GameScreen,'Anim.bmp',64, 64, 4);
  // Создаем четыре схемы заполнения фона
  Map1 := CDXMap.CDXMap(LandScape, GameScreen);
  Map1.CreateMap(64, 64, 1);
  Map2 := CDXMap.CDXMap(LandScape, GameScreen);
  Map2.CreateMap(64, 64, 2);
  Map3 := CDXMap.CDXMap(LandScape, GameScreen);
  Map3.CreateMap(64, 64, 3);
  Map4 := CDXMap.CDXMap(LandScape, GameScreen);
  Map4.CreateMap(64, 64, 4);
end;
```

Поскольку нумерация шаблонов основана на нуле, в картинки включают дополнительный, неиспользуемый фон, идущий первым.

Код обработки клавиш легко читается, смысл всех действий вам должен быть понятен:

```
function KeyDown (Key : Byte): BOOL;   // Вспомогательная функция
begin
  Result := GameInput.Keys[Key] = 128; // Нажата ли клавиша
end;
procedure UpdateKeys;    // Процедура обработки клавиатуры
begin
```

```
  if KeyDown(DIK_RIGHT) then begin          // Стрелка вправо
    Map1.WrapScrollRight(MapScrollSpeed);   // Сдвиг вправо содержимого
    Map2.WrapScrollRight(MapScrollSpeed);   // Всех четырех окон
    Map3.WrapScrollRight(MapScrollSpeed);
    Map4.WrapScrollRight(MapScrollSpeed);
  end;
  if KeyDown(DIK_LEFT) then begin           // Стрелка влево
    Map1.WrapScrollLeft(MapScrollSpeed);
    Map2.WrapScrollLeft(MapScrollSpeed);
    Map3.WrapScrollLeft(MapScrollSpeed);
    Map4.WrapScrollLeft(MapScrollSpeed);
  end;
  if KeyDown(DIK_UP) then begin             // Стрелка вверх
    Map1.WrapScrollUp(MapScrollSpeed);
    Map2.WrapScrollUp(MapScrollSpeed);
    Map3.WrapScrollUp(MapScrollSpeed);
    Map4.WrapScrollUp(MapScrollSpeed);
  end;
  if KeyDown(DIK_DOWN) then begin           // Стрелка вниз
    Map1.WrapScrollDown(MapScrollSpeed);
    Map2.WrapScrollDown(MapScrollSpeed);
    Map3.WrapScrollDown(MapScrollSpeed);
    Map4.WrapScrollDown(MapScrollSpeed);
  end;
  if KeyDown(DIK_ESCAPE) then begin         // Выход
    GameScreen.FadeTo(255, 255, 255, 0);    // Эффект угасания
    GameScreen.FadeOut(4);
    frmDD.Close;
  end;
end;
```

Обрабатывается нажатие нескольких клавиш одновременно, образы можно передвигать по диагонали.

Вывод осуществляется в задний буфер, каждая карта отсекается по своему сектору:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  Window1 : TRECT;     // Секторы окна
  Window2 : TRECT;
  Window3 : TRECT;
  Window4 : TRECT;
begin
  SetRect (Window1, 0, 0, 320, 240);  // Четыре равные части экрана
  SetRect (Window2, 320, 0, 640, 240);
```

```
  SetRect (Window3, 0, 240, 640, 480);
  SetRect (Window4, 320, 240, 640, 480);
  GameInput.Update;        // Обновить данные о клавиатуре
  UpdateKeys;              // Обслужить нажатые клавиши
  // Вывод в задний кадр четырех карт, отсекаемых по секторам
  Map1.DrawClipped(GameScreen.GetAppBackBuffer, Window1);
  Map2.DrawClipped(GameScreen.GetAppBackBuffer, Window2);
  Map3.DrawClipped(GameScreen.GetAppBackBuffer, Window3);
  Map4.DrawClipped(GameScreen.GetAppBackBuffer, Window4);
  Result := GameScreen.Flip;  // Переключение страниц
end;
```

Для восстановления поверхностей используется метод `Restore`.

В продолжение нашего знакомства с библиотекой CDX разберем проект каталога Ex15, помогающий постигнуть создание анимации. В качестве фона здесь используются те же круги, что и в предыдущем примере, которые сменяют друг друга на экране.

Существует одна переменная, связанная с фоном, в нее загружаются различные фрагменты растра:

```
GameMap := CDXMap.CDXMap(LandScape, GameScreen); // Создание лоскута
GameMap.CreateMap(MapSizeX, MapSizeY, 1);
GameMap.MoveTo(0, 0);
Tile := 1;
for i := 0 to 63 do               // Цикл заполнения карты
  for j := 0 to 62 do begin       // разными фрагментами
    GameMap.SetTile(i, j, Tile);
    Tile := Tile + 1;
    if Tile > 4 then Tile := 1;
  end;
```

Через некоторый промежуток времени экран заполняется новым фоном:

```
var
  Delay : Integer = 0;    // Счетчик кадров
function TfrmDD.UpdateFrame : HRESULT;
var
  wrk : TRECT;            // Прямоугольник экрана
  i, j, Tile : Integer;
begin
  GameInput.Update;
  UpdateKeys;
  SetRect (wrk, 0, 0, ScreenWidth, ScreenHeight);
  // Вывести текущее состояние фона
  GameMap.DrawClipped (GameScreen.GetAppBackBuffer, wrk);
  Inc (Delay);
```

```
  if Delay > 40 then begin    // Прошло 40 кадров
    for i := 0 to 62 do
      for j := 0 to 62 do begin
        Tile := GameMap.GetTile(i, j);  // Получить номер фрагмента
        Inc (Tile);       // Циклический сдвинуть в цепочке фрагментов
        if Tile > 4 then Tile := 1;
        GameMap.SetTile(i, j, Tile);    // Задать новый фрагмент
      end;
    Delay := 0;
  end;
  Result := GameScreen.Flip;
end;
```

Код обработки клавиатуры в примере заметно короче по сравнению с предыдущим:

```
procedure UpdateKeys;
begin
  if KeyDown(DIK_RIGHT) then GameMap.WrapScrollRight(MapScrollSpeed);
  if KeyDown(DIK_LEFT) then GameMap.WrapScrollLeft(MapScrollSpeed);
  if KeyDown(DIK_UP) then GameMap.WrapScrollUp(MapScrollSpeed);
  if KeyDown(DIK_DOWN) then GameMap.WrapScrollDown(MapScrollSpeed);
  if KeyDown(DIK_ESCAPE) then frmDD.Close;
end;
```

На рис. 5.15 запечатлен момент работы нашего очередного примера (проекта каталога Ex16), в котором на экране выводятся координаты пользовательского курсора.

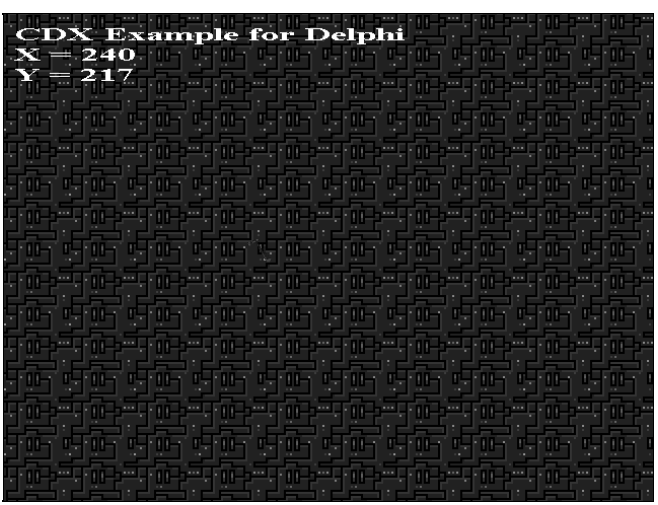


Рис. 5.15. Пример вывода текста и обработки событий мыши

Для изображения курсора предназначена отдельная поверхность, для которой задается ключ:

```
GameCursor := CDXSurface.Create;
GameCursor.CreateCDXSurfaceFromFile(GameScreen,'Cur.bmp');
GameCursor.ColorKey(0);
```

Для заднего буфера задается конкретный шрифт:

```
GameScreen.GetAppBackBuffer.ChangeFont('Times', 16, 20, FW_BOLD);
```

Аналогично процедуре обработки клавиатуры, требуется процедура, связанная с событиями мыши. Обратите внимание, как организована прокрутка изображения:

```
procedure UpDateMouse;
var
  TempX, TempY : Integer;
begin
  TempX := GameInput.Mouse.X;      // Смещение по осям
  TempY := GameInput.Mouse.Y;
  CurX := CurX + 3 * TempX;        // Текущие координаты курсора
  CurY := CurY + 3 * TempY;
  // Анализ положения курсора вблизи границ экрана
  if CurX < 0 then CurX := 0 else
  if CurX > ScreenWidth - MapSizeX then CurX := ScreenWidth - MapSizeX;
  if CurY < 0 then CurY := 0 else
  if CurY > ScreenHeight - MapSizeY then CurY := ScreenHeight - MapSizeY;
  if CurX = 0 then begin
    if TempX < 0 then GameMap.WrapScrollLeft(-TempX);
  end else
  if CurX = ScreenWidth - MapSizeX then
    if TempX > 0 then GameMap.WrapScrollRight(TempX);
  if CurY = 0 then begin
    if TempY < 0 then GameMap.WrapScrollUp(-TempY);
  end else
  if CurY = ScreenHeight - MapSizeY then
    if TempY > 0 then GameMap.WrapScrollDown(TempY);
end;
```

Вывод текста на экран осуществляется с помощью метода `TextXY` заднего буфера:

```
function TfrmDD.UpdateFrame : HRESULT;
var
  wrk : TRECT;
begin
  GameInput.Update;
```

```
  UpdateKeys;
  UpdateMouse;
  SetRect (wrk, 0, 0, ScreenWidth, ScreenHeight);
  GameMap.DrawClipped (GameScreen.GetAppBackBuffer, wrk);
  // Вывод курсора
  GameCursor.DrawFast(CurX, CurY, GameScreen.GetAppBackBuffer);
  // Вывод текста
  GameScreen.GetAppBackBuffer.TextXY(10, 10, 255,
                                     'CDX Example for Delphi');
  GameScreen.GetAppBackBuffer.TextXY(10, 30, 255, PChar('X = ' +
                                      IntToStr(CurX)));
  GameScreen.GetAppBackBuffer.TextXY(10, 50, 255, PChar('Y = ' +
                                      IntToStr(CurY)));
  Result := GameScreen.Flip;
end;
```

Последний пример на эту тему (проект каталога Ex17) поможет нам разобраться в организации спрайтовой анимации. Здесь вид курсора меняется со временем так, что получается изображение страшного животного, раскрывающего зубастую пасть (рис. 5.16).

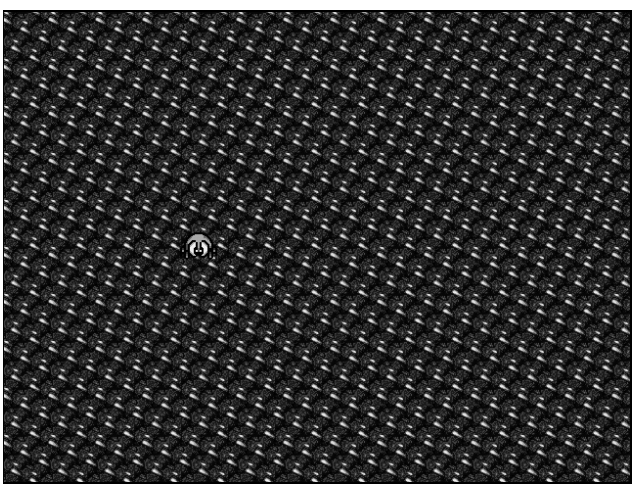


Рис. 5.16. При работе примера чудовище раскрывает и закрывает свою пасть

Фазу можно менять, используя метод `SetTile`, как в одном из предыдущих примеров, или же напрямую задавая прямоугольник источника:

```
Inc (Delay);
if Delay = 10 then begin      // Прошло 10 кадров
  // Меняем прямоугольник в источнике
  SetRect (GameCursor.SrcRect, 39 * wrkI, 0, 39 * (wrkI + 1), 36);
```

```
  wrkI := (wrkI + 1) mod 3;
  Delay := 0;
end;
```

В данном разделе мы рассмотрели лишь основные функции библиотеки CDX, все остальные остаются вам для самостоятельного изучения.

Я не думаю, что здесь вы встретите особые проблемы, т. к. после проведенного нами детального изучения механизмов DirectDraw знакомство с подобными библиотеками (и исправление ошибок в исходном коде) превращается в приятное времяпрепровождение.

Что вы узнали в этой главе

Главное, что мы смогли выяснить в данной главе, можно сформулировать следующей торжественной фразой: узнали все необходимое для программирования собственной эффектной игры.

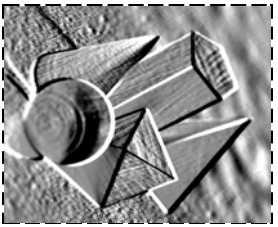
Мы научились работать с устройствами ввода настолько быстро, что приложение мгновенно реагирует на изменение состояний устройств.

Спрайтовая анимация изучена нами до уровня, нужного для разработки игр.

Примеры несложных игр убедительно демонстрируют достигнутые нами высоты мастерства.

Мы познакомились с готовой библиотекой, использующейся профессиональными разработчиками игр.

ГЛАВА 6



Работа с AVI-файлами

Эта небольшая глава посвящена вопросам, относящимся к воспроизведению и созданию видео. Многие аспекты напрямую не связаны с DirectX и поэтому вынесены в отдельную часть.

Примеры к главе располагаются в каталоге \Examples\Chapter06.

Модуль *VFW*

Существует много способов работы с видео. Предназначенные для этого компоненты Delphi не очень помогут в решении задачи воспроизведения подобных файлов на поверхностях, поэтому нам придется воспользоваться нестандартными методами.

Прежде всего мы познакомимся с модулем vfw.pas, не входящим в набор стандартных модулей Delphi, но работающим со стандартной системной библиотекой avifil32.dll.

Вместе с Delphi поставляется набор справочных файлов системного программиста, корпорации Microsoft. С их помощью вы сможете разобраться во всех тонкостях использования этого модуля. Я же приведу конкретные решения с краткими пояснениями. Для большей части читателей этого объема должно вполне хватать.

В проекте каталога Ex01 на знакомом нам фоне воспроизводится AVI-файл, взятый мною из набора файлов, поставляемых в составе DirectX SDK (рис. 6.1).

В списке подключаемых модулей добавлен модуль `VFW`, а в установках проекта — путь к файлу ole2.dcu, в котором нуждается подключаемый модуль.

Следующие переменные связаны со спецификой примера:

```
var
  TmpBmp : TBitmap;          // Вспомогательное изображение
  AviStream : PAVISTREAM;    // AVI-поток
```

```
  Frame : PGetFrame;          // Кадр видео
  pbmi : PBITMAPINFOHEADER;   // Указатель на заголовок растра
  bits : Pointer;             // Указатель на картинку растра
  CurrFrame : DWORD = 0;      // Счетчик кадров
  AVIClock  : DWORD;          // Эмулятор таймера
  AVIDelay  : DWORD;          // Величина паузы между кадрами
  AVIWidth  : DWORD;          // Характеристики кадра
  AVIHeight : DWORD;
  AVILength : DWORD;          // Количество кадров в AVI
```

Рис. 6.1. Простой пример воспроизведения видео на поверхности

Перед инициализацией DirectDraw нам необходимо узнать характеристики отображаемого видео, для этого используем функции рассматриваемого модуля, связанные с работой AVI как с файлом:

```
var
  AVIFile : PAVIFile;         // Обрабатываем AVI как файл
  AVIInfo : TAVIFileInfo;     // Заголовок файла, характеристики AVI
begin
  TmpBmp := TBitmap.Create;  // Создаем вспомогательный растр
  AVIFileOpen(AVIFile, AVIName, OF_READ, nil);    // Открытие AVI
  // Считываем заголовочную информацию, заполняются поля AVIInfo
  AVIFileInfo(AVIFile, AVIInfo, SizeOf (AVIInfo));
  AVIWidth := AVIInfo.dwWidth;     // Запоминаем размеры кадра
  AVIHeight := AVIInfo.dwHeight;
  AVILength := AVIInfo.dwLength;   // Количество кадров
```

```
  // Вычисляем паузу между очередными кадрами
  AVIDelay := 1000 div (AVIInfo.dwRate div AVIInfo.dwScale);
  AVIFileRelease(AVIFile); // Освобождаем AVI
```

Поверхность FDDSImage, на которой будет воспроизводиться видео, создается с размерами, равными размерам кадра видео:

```
ZeroMemory (@ddsd, SizeOf(ddsd));
with ddsd do begin
  dwSize := SizeOf(ddsd);
  dwFlags := DDSD_CAPS or DDSD_HEIGHT or DDSD_WIDTH;
  ddsCaps.dwCaps := DDSCAPS_OFFSCREENPLAIN;
  dwWidth  := AviWidth;
  dwHeight := AVIHeight;
end;
hRet := FDD.CreateSurface(ddsd, FDDSImage, nil);
if Failed(hRet) then ErrorOut(hRet, 'Create Second Surface');
```

Совет

В общем случае заводить отдельную поверхность для воспроизведения кадра, конечно, не обязательно, можно переносить содержимое вспомогательного растра прямо на первичную поверхность.

После инициализации DirectDraw подготавливаемся к работе с потоковым видео:

```
procedure TfrmDD.FirstFrame;
var
  wrkDC : HDC;
begin
  AVIFileInit;          // Инициализация библиотеки
  // Открываем AVI-файл для чтения
  AVIStreamOpenFromFile(AviStream, AviName, streamtypeVIDEO,
                        0, OF_READ, nil);
  // Загружаем поток
  Frame := AVIStreamGetFrameOpen(AviStream, nil);
  // Получаем первый кадр видео
  pbmi := AVIStreamGetFrame(Frame, CurrFrame);
  // Получаем указатель на картинку кадра
  bits := Pointer(Integer(pbmi) + SizeOf(TBITMAPINFOHEADER));
  // Получаем контекст для воспроизведения кадра на поверхность
  if FDDSImage.GetDC (wrkDC) = DD_OK then begin
    // Воспроизводим кадр во вспомогательный растр
    TmpBmp.Handle := CreateDIBitmap(
    // Вспомогательным контекстом служит HDC поверхности
      wrkDC,
      pbmi^,               // Адрес размера растра и формата данных
      CBM_INIT,            // Флаг инициализации
```

```
      bits,                // Данные для инициализации
      PBITMAPINFO(pbmi)^,  // Данные о формате цвета
      DIB_RGB_COLORS);     // Флаг цветности растра
    // Переносим картинку из вспомогательного растра на поверхность
    BitBlt (wrkDC, 0, 0, AVIWidth, AVIHeight,
            TmpBmp.Canvas.Handle, 0, 0, SRCCOPY);
    FDDSImage.ReleaseDC (wrkDC);
  end;
  AVIClock := GetTickCount; // Инициализация вспомогательного таймера
end;
```

Действия по воспроизведению очередного кадра аналогичны, но тратить время на получение адресов теперь не нужно:

```
procedure TfrmDD.NextFrame;
var
  wrkDC : HDC;
begin
  // Настало время воспроизвести следующий кадр AVI
  if GetTickCount – AVIClock > AVIDelay then begin
    pbmi := AVIStreamGetFrame(Frame, CurrFrame);
    if FDDSImage.GetDC (wrkDC) = DD_OK then begin
      TmpBmp.Handle := CreateDIBitmap(wrkDC, pbmi^, CBM_INIT,
        bits, PBITMAPINFO(pbmi)^, DIB_RGB_COLORS);
      BitBlt (wrkDC, 0, 0, AVIWidth, AVIHeight,
              TmpBmp.Canvas.Handle, 0, 0, SRCCOPY);
      FDDSImage.ReleaseDC (wrkDC);
    end;
    // Увеличиваем счетчик кадров
    CurrFrame := (CurrFrame + 1) mod AVILength;
    AVIClock := GetTickCount;
  end;
end;
```

В этом примере AVI-файл воспроизводится бесконечно, вслед за последним кадром все повторяется с начала.

Кадр воспроизведен на поверхности `FDDSImage`, блиттинг которой осуществляется тривиальным способом.

По завершении работы добавились ожидаемые действия:

```
AVIStreamRelease(AviStream);  // Закрытие потока
AVIFileExit;                  // Завершение работы с библиотекой
TmpBmp.Free;                  // Удаление вспомогательного растра
```

Итак, теперь вы способны делать видео частью ваших игр, но я должен предупредить, что библиотека может корректно работать не с каждым типом

сжатия. По этой причине я рекомендую заставки игр воспроизводить стандартным для Delphi способом.

Модуль *DirectShow*

Поскольку изложенный в предыдущем разделе способ годится не для каждого видео, нам придется бегло рассмотреть еще один способ воспроизведения видео, основанный на использовании модуля `DirectShow`. Эта библиотека также входит в состав DirectX, включает набор функций для работы с мультимедиа. Подробно рассматривать ее не будем, познакомимся с ее использованием на конкретном примере, проекте каталога Ex02, воспроизводящем AVI-файл на поверхности (рис. 6.2).

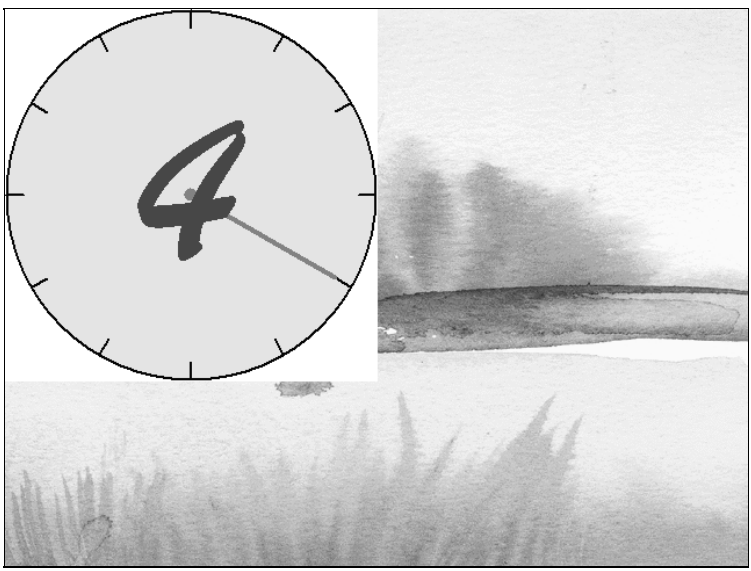


Рис. 6.2. Работа примера на тему использования модуля `DirectShow`

Файл видео для этого примера также взят мною из пакета DirectX SDK.

При воспроизведении файла с помощью модуля `VFW` картинка получается искаженной, поэтому и приходится прибегать к иному способу. Отказаться от первого способа мы также не можем, поскольку `DirectShow` тоже годится не для любого файла. Другая причина, по которой мы не можем усердствовать в изучении упомянутого модуля, состоит в том, что он может применяться лишь с интерфейсами ранних версий.

Модуль использует COM-модель, поэтому здесь мы встретим знакомые понятия главного объекта и дочерних интерфейсов:

```
var
 AMStream : IAMMultiMediaStream;   // Главный объект
 PrimaryVidStream : IMediaStream;  // Дочерний поток, связан с видео
 Sample : IDirectDrawStreamSample; // Интерфейс для вывода на поверхность
```

В процедуру инициализации потока передается имя требуемого файла:

```
procedure TfrmDD.PlayMedia(const FileName: WideString);
var
  hRet : HRESULT;
begin
  // Создание главного объекта ('filter graph')
  AMStream:=IAMMultiMediaStream(CreateComObject
                                (CLSID_AMMultiMediaStream));
  // Инициализация потока для чтения
  hRet := AMStream.Initialize(STREAMTYPE_READ, 0, nil);
  if Failed (hRet) then ErrorOut (hRet, 'Stream Initialize');
  // Добавление потока видео к главному объекту
  hRet := AMStream.AddMediaStream(FDD, MSPID_PrimaryVideo,
                                  0, IMediaStream(nil^));
  if Failed (hRet) then ErrorOut (hRet, 'Add Video Stream');
  // Открытие файла
  hRet := AMStream.OpenFile(PWideChar(FileName), 0);
  if Failed (hRet) then ErrorOut (hRet, 'Open AVI File');
  // Следующие действия предназначены для связывания потока и поверхности
  // Получение дочернего потока
  hRet := (AMStream as IMultiMediaStream).
          GetMediaStream(MSPID_PrimaryVideo, PrimaryVidStream);
  if Failed (hRet) then ErrorOut (hRet, 'GetMediaStream');
  // Преобразование интерфейса в тип Isample
  // и связывание его с поверхностью
  hRet := (PrimaryVidStream as IDirectDrawMediaStream).
  CreateSample(FDDSImage, TRect(nil^), 0, Sample);
  if Failed (hRet) then ErrorOut (hRet, 'CreateSample');
  // Запуск потока
  hRet := (AMStream as IMultiMediaStream).SetState(STREAMSTATE_RUN);
  if Failed (hRet) then ErrorOut (hRet, 'SetState');
end;
```

Обработку возможных ошибок я оставил прежней, но коды ошибок не будут расшифровываться. Модуль DirectDraw не содержит, конечно, пояснения по этим ошибкам.

Перед блиттингом поверхности `FDDSImage` обновляем позицию в видео:

```
if Sample.Update(0, 0, nil, 0) <> S_OK
  then (AMStream as IMultiMediaStream).Seek (0);
```

Неудача свидетельствует о том, что видео прокручено до конца, в этом случае мы заново запускаем его.

Самостоятельно разберитесь, каким образом останавливается ролик при деактивизации окна приложения, и как заново запускается поток при восстановлении.

Запись в видеофайл

Познакомившись с различными способами воспроизведения видеоданных, мы научимся создавать AVI-файлы. Пройдет этот процесс для вас очень легко, поскольку воспользуемся мы готовым модулем `AviMaker`. Он содержит описание класса `TAviMaker`, выполняющего за нас всю изнурительную работу. Нам остается только подготовить набор растров, составляющих последовательность кадров создаваемого фильма, и вызвать метод записи.

Вот скромный набор свойств и методов класса, необходимых нам для работы:

```
Bitmaps   : TList;    // Список объектов класса Bitmap, кадры AVI
Height    : Integer;          // Размеры кадров AVI
Width     : Integer;
FrameTime : Integer;          // Величина паузы между кадрами
Stretch   : BOOL;             // Признак, надо ли масштабировать кадры
FileName  : String;           // Имя файла результата
PixelFormat  : TPixelFormat;  // Разрядность AVI
constructor Create;
destructor Destroy; override;
procedure Write;              // Запись AVI
```

Здесь используется модуль `VFW`, поэтому в опциях проекта указывается путь к файлу ole2.dcu.

В проекте каталога Ex03 формируются кадры, на которых вращается спираль (рис. 6.3).

Для записи фильма используется объект `AviMaker1` класса `TAviMaker`:

```
AviMaker1 := TAviMaker.Create;
with AviMaker1 do begin
  Width := 256;
  Height := 256;
  Stretch := True;          // Кадры будут масштабироваться
  PixelFormat := pf24bit;  // 24-битный формат кадра
  FrameTime := 100;
  FileName := 'test.avi';
end;
```

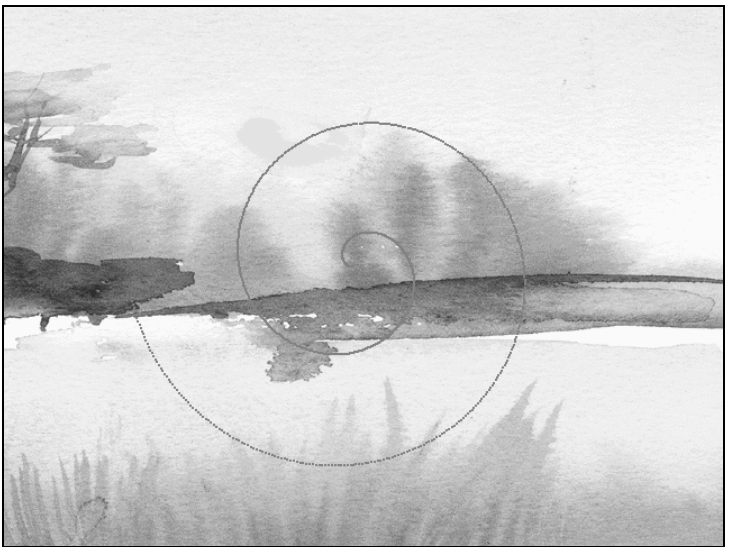


Рис. 6.3. Один из кадров нашего фильма

Создается фильм из 20-ти кадров, продолжительностью 2 секунды:

```
function TfrmDD.UpdateFrame : HRESULT;
const
  step = 2 * Pi / 400;
var
  i : Integer;
  hRet : HRESULT;
  Bitmap : TBitmap;
  DC : HDC;
begin
  ThisTickCount := GetTickCount;
  // В этом примере паузы можно было бы и не делать
  if ThisTickCount - LastTickCount > 30 then begin
    Angle := Angle + 0.25;
    if Angle > 2 * Pi then Angle := Angle - 2 * Pi;
    LastTickCount := GetTickCount;
    // Берем 20 кадров
    Inc (FrameCount);
    if FrameCount > 20 then begin
      FActive := False;
      Result := DD_OK;
      AviMaker1.Write;        // Записываем AVI
      Close;
      Exit;
    end;
```

```
    // Выводим фон
    hRet := FDDSBack.BltFast (0, 0, FDDSBackGround,
                              nil, DDBLTFAST_WAIT);
    if hRet = DDERR_SURFACELOST then begin
      Result := hRet;
      if Failed (RestoreAll) then Exit;
    end;
    // Выводим точки спирали
    for i := 0 to 800 do
      if FDDSBack.BltFast (310 + trunc (cos(Angle + i * step) * i / 4),
                           230 + trunc (sin(Angle + i * step) * i / 4),
                           FDDSImage, nil,
                           DDBLTFAST_WAIT) = DDERR_SURFACELOST then begin
        hRet := RestoreAll;
        if Failed (hRet) then begin
        Result := hRet;
        Exit;
      end;
    end;
    // Формируем кадр ролика
    Bitmap := TBitmap.Create;
    with Bitmap do begin
      Width := ScreenWidth;
      Height := ScreenHeight;
      PixelFormat := pf24bit;       // Важно, иначе устанавливается 8 бит
      FDDSBack.GetDC (DC);
      BitBlt(Canvas.Handle, 0, 0,   // Копируем экран в растр
             ScreenWidth, ScreenHeight,
             DC, 0, 0, SRCCOPY);
      FDDSBack.ReleaseDC (DC);
      AviMaker1.Bitmaps.Add (Bitmap);  // Добавляем кадр в фильм
    end;
  end;
  Result := DD_OK;
end;
```

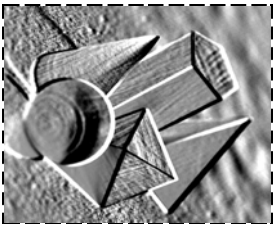
Кадры масштабируются при записи фильма, в качестве упражнения исправьте код так, чтобы их изменение происходило при формировании списка.

Что вы узнали в этой главе

Для работы с видео имеется много способов, и, как всегда, нам пришлось выбирать из них работающие.

Кроме того, вы получили инструмент для создания собственных фильмов.

ГЛАВА 7



Обзор библиотеки Direct3D

С этой главы мы начнем переход к принципиально новой технологии программирования двумерной и трехмерной графики и рассмотрим основные понятия такого подхода.

Примеры к главе располагаются в каталоге \Examples\Ex07.

Модуль *DirectXGraphics*

Итак, приступаем к изучению части DirectX, связанной с трехмерной графикой, хотя наши первые построения будут выполняться на плоскости. Теперь большая часть изученного в предыдущих главах книги непосредственно нами использоваться не будет, но многие подходы, понятия и приемы, с которыми мы уже познакомились, перекликаются с новым материалом. Например, в наших проектах мы встретим знакомые понятия главного объекта и порождаемых его методами вспомогательных объектов, и последовательность выполняемых нами действий будет во многом родственна предыдущим примерам.

Начнем наш путь с простейшего примера, проекта каталога Ex01. Если DirectDraw было удобнее начинать изучать с полноэкранных приложений, то с Direct3D мы познакомимся на примерах оконных приложений. В первом проекте данной главы клиентская часть окна окрашивается синим цветом. Это минимальное приложение, использующее Direct3D. Окно непрерывно перерисовывается, а в его заголовке выводится значение FPS.

Вначале бегло посмотрим код, потом некоторые ключевые моменты обсудим подробнее.

Прежде всего, замечаем, что в списке `uses` модуля дописан модуль `DirectXGraphics`. Это базовый модуль, играющий для наших последующих примеров такую же роль, какую играл ранее модуль DirectDraw. В этом модуле содержится описание базовых интерфейсов, типов и констант.

Имя формы этого и последующих примеров я задал `frmD3D`.

В разделе `private` описания класса формы мною внесены следующие строки:

```
FD3D            : IDIRECT3D8;          // Главный объект
FD3DDevice      : IDIRECT3DDEVICE8;    // Объект устройства
FActive         : BOOL;                // Вспомогательный флаг
ThisTickCount   : DWORD;               // Отсчет времени для подсчета FPS
LastTickCount   : DWORD;
function InitD3D : HRESULT;            // Инициализация системы
function Render : HRESULT;             // Воспроизведение
procedure Cleanup;                     // Удаление объектов
procedure ErrorOut (const Caption : PChar; const hError : HRESULT);
```

Сообщение об ошибке выводится пока в отдельном окне, для оконных приложений здесь не должно возникать проблем:

```
procedure TfrmD3D.ErrorOut (const Caption : PChar;
                            const hError : HRESULT);
begin
  FActive := False;    // Остановить перерисовку окна
  Cleanup;             // Удалить все объекты
  MessageBox (Handle, PChar(DXGErrorString (hError)), Caption, 0);
end;
```

Функция `DXGErrorString` возвращает описание ошибки, код которой передается в качестве аргумента. Эта функция представлена в модуле `DirectXGraphics`.

В процедуре очистки памяти объекты высвобождаются знакомым нам способом:

```
procedure TfrmD3D.Cleanup;
begin
  if Assigned (FD3DDevice) then begin
    FD3DDevice._Release;
    FD3DDevice := nil;
  end;
  if Assigned (FD3D) then begin
    FD3D._Release;
    FD3D := nil;
  end;
end;
```

Данная процедура вызывается при выводе описания аварийной ситуации и при завершении работы приложения, в обработчике `OnDestroy` окна.

Инициализация графической системы включает действия, смысл которых нам интуитивно понятен и без дополнительных комментариев:

```
function TfrmD3D.InitD3D : HRESULT;
var
  d3ddm : TD3DDISPLAYMODE;    // Вспомогательные структуры
  d3dpp : TD3DPRESENT_PARAMETERS;
  hRet : HRESULT;
begin
  FD3D := nil;
  FD3DDevice := nil;
  // Создаем главный объект
  FD3D := Direct3DCreate8(D3D_SDK_VERSION);
  if FD3D = nil then begin
    Result := E_FAIL;
    Exit;
  end;
  // Получаем установки рабочего стола
  hRet := FD3D.GetAdapterDisplayMode(D3DADAPTER_DEFAULT, d3ddm);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Заполняем структуру, задающую параметры работы
  ZeroMemory(@d3dpp, SizeOf(d3dpp));     // Обнуляем поля
  with d3dpp do begin
    Windowed := True;                    // Используется оконный режим
    SwapEffect := D3DSWAPEFFECT_DISCARD; // Режим переключения буферов
    BackBufferFormat := d3ddm.Format;    // Формат заднего буфера
  end;
  // Создаем вспомогательный объект, объект устройства
  Result := FD3D.CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, Handle,
                              D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                              d3dpp, FD3DDevice);
end;
```

Главным интерфейсом является COM-объект класса `IDIRECT3D8`, методы которого позволяют получить доступ к функциям библиотеки. Главный объект создается первым, а уничтожается последним. Создается он с помощью функции `Direct3DCreate8`, единственным аргументом которой является константа, сообщающая системе, какая версия DirectX SDK использовалась при компиляции приложения.

Методы главного объекта позволяют узнать текущие установки видеосистемы, и следующим действием нашей программы служит вызов метода `GetAdapterDisplayMode`. Как правило, обязательным это действие является только для оконных приложений, поскольку такие установки требуются для задания параметров заднего буфера.

У метода GetAdapterDisplayMode два аргумента:

- константа, задающая адаптер, для которого запрашиваются установки;
- указатель на вспомогательную переменную, в которую помещается результат, являющийся описанием характеристик устройства.

Предопределенным значением первого аргумента пока может использоваться только D3DADAPTER_DEFAULT, нулевая константа, соответствующая первичному устройству. Для описания характеристик служит переменная типа TD3DDISPLAYMODE, запись:

```
TD3DDisplayMode = packed record
  Width       : Cardinal;         // Ширина рабочего стола
  Height      : Cardinal;         // Высота рабочего стола
  RefreshRate : Cardinal;         // Частота регенерации
  Format      : TD3DFormat;       // Формат пиксела
end;
```

То есть, чтобы вывести текущую ширину рабочего стола, можно вставить такую строку:

```
ShowMessage (IntToStr (d3ddm.Width));
```

Значением частоты регенерации для основного устройства мы получим ноль.

Последний элемент записи позволяет узнать формат пиксела. Возможные значения этого поля перечислены в модуле DirectXGraphics. Все они начинаются на префикс "D3DFMT_". Констант довольно-таки много, я не стану детально рассматривать их все, только посмотрим, как можно идентифицировать две наиболее распространенных:

```
case d3ddm.Format of
  D3DFMT_X8R8G8B8 : ShowMessage ('Формат пиксела: 32-битный RGB.');
  D3DFMT_R5G6B5 : ShowMessage ('Формат пиксела: 16-битный 5-6-5.');
  else ShowMessage ('Формат пиксела в списке отсутствует.');
end;
```

Примечание

Обратите внимание, что при цветовой палитре рабочего стола, меньшей 16 бит на пиксел, работа Direct3D невозможна.

На следующем шаге инициализации задаются параметры работы, заполняются поля структуры типа TD3DPRESENT_PARAMETERS. В этом примере я выполняю только минимальный набор обязательных действий.

Логическое значение поля Windowed задает режим работы приложения: наше приложение должно работать в оконном режиме. В поле SwapEffect заносится константа, задающая порядок работы с задним буфером. Я использую

константу D3DSWAPEFFECT_DISCARD, соответствующую режиму, при котором DirectX не заботится о сохранности содержимого заднего буфера при циклическом переключении страниц. В поле BackBufferFormat помещается формат пиксела для заднего буфера. Именно здесь необходимы полученные на предыдущем шаге характеристики рабочего стола.

И после этого вызывается метод главного объекта CreateDevice, с помощью которого создается дочерний интерфейс типа IDIRECT3DDEVICE8. Объект такого типа представляет собой непосредственно устройство вывода. Собственно, с помощью его методов и производятся воспроизведение и модификация изображения. У метода CreateDevice шесть аргументов. Первым является устройство вывода, используемая здесь константа нам уже знакома. Вторым аргументом передается константа, задающая тип воспроизведения: использовать или нет аппаратное ускорение. Указание в качестве аргумента константы D3DDEVTYPE_HAL соответствует первому случаю, второму — D3DDEVTYPE_REF. Еще одна возможная константа — D3DDEVTYPE_SW, предназначена для подключения встраиваемых модулей, зарегистрированных в DirectX.

Примечание

На маломощных видеокартах даже в самых простых программах, использующих Direct3D, будет порождаться исключение. Вы можете предусмотреть обработку этого исключения, чтобы попытаться повторно инициализировать систему с параметром D3DDEVTYPE_REF. Тогда скорость работы будет настолько низкой, что вряд ли пользователи останутся удовлетворенными.

Третьим аргументом метода CreateDevice задается идентификатор окна, в котором осуществляется вывод, свойство Handle формы хранит значение этого идентификатора. Следующий параметр задает порядок работы с вершинами: обрабатываются математические операции центральным процессором либо ускорителем. Здесь мы будем использовать константу D3DCREATE_SOFTWARE_VERTEXPROCESSING, чтобы наши программы работали на всех графических картах. Пятый, предпоследний, аргумент метода CreateDevice — переменная типа TD3DPRESENT_PARAMETERS, с помощью которой мы передаем заполненную нами ранее структуру. В ней же будут содержаться скорректированные системой значения устанавливаемого режима. Например, количество задних буферов в примере задается первоначально равным нулю, система скорректирует это значение при создании объекта устройства. Добавьте в код следующую строку:

```
ShowMessage (IntToStr(d3dpp.BackBufferCount));
```

И убедитесь, что наше приложение не осталось без вспомогательного экрана.

Последний аргумент рассматриваемого метода — собственно формируемый объект.

Процедура инициализации вызывается при создании окна, обработка возможных ошибок является, по-прежнему, необходимым элементом наших программ:

```
procedure TfrmD3D.FormCreate(Sender: TObject);
var
  hRet : HRESULT;
begin
  hRet := InitD3D;
  if Failed (hRet) then ErrorOut ('InitD3D', hRet);
end;
```

Основная нагрузка в примере ложится на функцию `Render`, в которой выполняется единственное действие — экран окрашивается синим цветом:

```
function TfrmD3D.Render : HRESULT;
var
  hRet : HRESULT;
begin
  // Инициализация не выполнена, либо произошла серьезная авария
  if FD3DDevice = nil then begin
     Result := E_FAIL;
     Exit;
  end;
  hRet := FD3DDevice.Clear(0, nil, D3DCLEAR_TARGET,
          D3DCOLOR_XRGB(0, 0, 255), 0.0, 0);   // Очистка заднего буфера
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Переключение буферов устройства
  Result := FD3DDevice.Present(nil, nil, 0, nil);
end;
```

Начинается код функции с проверки присутствия объекта устройства. Этот объект может отсутствовать, если инициализация не выполнилась успешно, либо объект потерян. Последняя ситуация может возникнуть, когда, например, по ходу работы приложения меняются установки рабочего стола. Обратите внимание, что при отсутствии объекта устройства наша функция `Render` возвращает значение `E_FAIL`, но функция обработки ошибки `DXGErrorString` в ответ на такую ошибку возвращает строку `'Unrecognized Error'` (Неопознанная ошибка). Вы можете избавиться от неопределенности сообщения, введя собственную константу на случай потери объекта устройства.

Далее в функции `Render` вызывается метод `Clear` объекта устройства. Первый и второй аргументы метода позволяют задавать очищаемую область: первый

аргумент задает количество используемых прямоугольников, вторым аргументом передается указатель на массив, содержащий набор величин типа TRect.

Третьим аргументом метода Clear уточняются параметры очистки. Здесь указывается флаг или комбинация флагов. Константа D3DCLEAR_TARGET используется в ситуации, когда очищается цветовая поверхность устройства. Сам цвет, в который "перекрашивается" устройство, передается следующим параметром. В примере цвет, которым будет окрашено окно, идентифицируем, используя готовую функцию D3DCOLOR_XRGB. Ее аргументом является тройка весов чистых цветов, образующих нужный нам оттенок. Последние два аргумента метода пока оставим без рассмотрения, связаны они со специальными буферами.

Окрасив задний буфер чистым синим цветом, нам остается только переставить буферы — вызываем метод Present объекта устройства. Если третий аргумент метода нулевой, то идентификатором окна, в котором происходит работа, берется значение, установленное ранее, во время инициализации работы системы. Все остальные параметры метода или не используются, или при указанных нулевых значениях задают работу со всей клиентской областью окна, что, как правило, и необходимо.

В состоянии ожидания сообщений беспрерывно вызывается функция Render, если окно приложения не минимизировано:

```
procedure TfrmD3D.ApplicationEvents1Minimize(Sender: TObject);
begin
  FActive := False;   // При минимизации окна приложения флаг опускаем
end;
procedure TfrmD3D.ApplicationEvents1Restore(Sender: TObject);
begin
  FActive := True;    // Окно восстановлено, флаг поднимаем
end;
```

Помимо непрерывной перерисовки окна периодически подсчитывается и выводится в его заголовке значение FPS:

```
procedure TfrmD3D.ApplicationEvents1Idle(Sender: TObject;
  var Done: Boolean);
var
  hRet : HRESULT;
begin
  if FActive then begin     // Только при активном окне
    Inc (Frames);
    hRet := Render;         // Перерисовка окна
    if FAILED(hRet) then begin
      FActive := False;
      ErrorOut ('Render', hRet);
```

```
      Exit;
    end;
    ThisTickCount := GetTickCount;
    if ThisTickCount - LastTickCount > 50 then begin
    // Подсчет и вывод FPS
      Caption := 'FPS = ' + Format('%6.2f',
                 [frames * 1000 / (ThisTickCount - LastTickCount)]);
      Frames := 0;
      LastTickCount := GetTickCount;
    end;
  end;
  Done := False;
end;
```

Минимальное по сложности приложение, использующее Direct3D, мы разобрали, теперь попробуем проверить один момент. В проекте каталога Ex02 левая и правая половины окна окрашиваются в синий и красный цвета соответственно. Клиентская область окна имеет размер 300×300 пикселов. В функции `Render` для задания областей окрашивания используется переменная `wrkRect` типа `TRect`:

```
SetRect (wrkRect, 0, 0, 150, 300);            // Левая область окна
hRet := FD3DDevice.Clear(1, @wrkRect, D3DCLEAR_TARGET,
        D3DCOLOR_XRGB(0, 0, 255), 0.0, 0);    // Первую область
                                              // окрашиваем синим
if FAILED(hRet) then begin
  Result := hRet;
  Exit;
end;
SetRect (wrkRect, 150, 0, 300, 300);          // Правая область
hRet := FD3DDevice.Clear(1, @wrkRect, D3DCLEAR_TARGET,
        D3DCOLOR_XRGB(255, 0, 0), 0.0, 0);    // Вторую область
                                              // окрашиваем красным
if FAILED(hRet) then begin
  Result := hRet;
  Exit;
end;
```

Проект каталога Ex03 вы сможете разобрать и без моей помощи, это несложная модификация предыдущего примера, отличающаяся только тем, что значения цветовых весов со временем меняются. Я же обращу ваше внимание на одну немаловажную особенность последних двух примеров: при изменении размеров окна его половинки окрашиваются так, будто в коде отслеживаются его текущие размеры. Делаем важный вывод: размеры клиентской области окна на момент инициализации Direct3D определяют область вывода на весь сеанс работы с графической системой.

Тип *TColor* и цвет в Direct3D

Цвет в Direct3D задается 32-битным числом, так называемый формат ARGB. Последний байт этого числа задает вес синего цвета (B), предпоследний — зеленого (G), второй — красного (R). Смысл первого байта раскроем попозже, пока же его значение никак не влияет на результат работы программ.

В первом примере для окрашивания окна в чистый синий строку очистки заднего буфера можно записать так:

```
hRet := FD3DDevice.Clear(0, nil, D3DCLEAR_TARGET, $000000FF, 0.0, 0);
```

В типе `TColor`, с которым вам приходилось часто работать в Delphi, также задействованы четыре байта, но последний байт отвечает за красный, а второй — за синий цвета. Потренируемся в переводе цвета из одного формата в другой и обратимся за помощью к проекту каталога Ex04.

На форме появилось два дополнительных объекта: кнопка **Color** и компонент, связанный с диалогом задания цвета. Переменная `DXColor` типа `DWORD` хранит текущее значение цвета, в который окрашивается задний буфер. При нажатии кнопки появляется диалог указания цвета. Выбранный цвет устанавливается значением `DXColor`:

```
procedure TfrmD3D.Button1Click(Sender: TObject);
begin
  if ColorDialog1.Execute then DXColor := ColorToDX (ColorDialog1.Color);
end;
```

В пользовательской функции `ColorToDX` из аргумента "вырезаются" байты цветовых компонентов, затем заново склеиваются в нужном порядке, первый байт остается нулевым:

```
function ColorToDX (C : TColor) : DWORD;
var
  R, G, B : Byte;
begin
  R := C and $FF;                 // Последний байт, красный цвет
  G := (C and $FF00) shr 8;       // Предпоследний байт, зеленый цвет
  B := (C and $FF0000) shr 16;    // Синий цвет
  Result := (R shl 16) or (G shl 8) or B;
end;
```

Протестируйте работу приложения, все должно работать хорошо. Попутно этот несложный пример иллюстрирует, что мы можем без особых ухищрений использовать визуальные компоненты Delphi в проектах на основе Direct3D. Эту хорошую новость я немного подпорчу замечанием, что не все визуальные компоненты хорошо кооперируются с такими проектами. Только те, которые имеют свойство `Handle`, не будут загорожены экраном вос-

произведения. Такие компоненты создают собственное окно, которое не захватывается объектом устройства.

Визуальные компоненты, имеющие свойство `Handle`, вполне подходят для использования их в качестве холста. Посмотрите проект каталога Ex05, который отличается от предыдущего тем, что воспроизведение в нем осуществляется не на канву окна, а на панель, занимающую лишь часть окна (рис. 7.1).

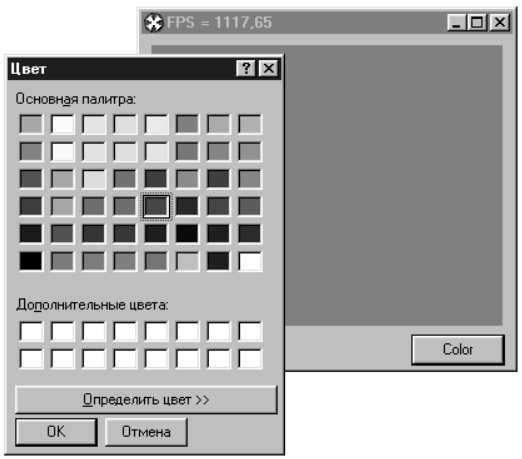


Рис. 7.1. Воспроизведение возможно осуществлять не только на канве окна

Это стоило небольших трудов: третьим аргументом метода `CreateDevice` главного объекта передается идентификатор окна панели:

```
Result := FD3D.CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
                            Panel1.Handle,
                            D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                            d3dpp, FD3DDevice);
```

Наверное, мы уже готовы к тому, чтобы нарисовать что-нибудь на экране.

Примитивы

Рисование в Direct3D осуществляется с помощью примитивов. Под этим термином следует понимать простую фигуру. Базовыми примитивами являются *точка*, *отрезок* и *треугольник*.

Каждый примитив задается набором вершин, характеристиками опорных точек примитива. Для хранения вершин, определяющих примитив, предназначен *буфер вершин* (vertex buffer). Буферы вершин представляют собой области памяти, которыми управляет Direct3D. Данные в буфере вершин

должны иметь строго определенный формат из некоторого набора. Выяснив требуемый формат, клиент должен уведомить об этом графическую систему с помощью набора флагов FVF (Flexible Vertex Format, формат гибких вершин). В FVF-флаге содержится перечисление используемых компонентов формата вершины из определенного набора.

Для манипуляций с вершинами предназначен особый механизм, называемый *вершинным шейдером* (vertex shader). После того как буферы вершин заполнены, объект устройства создает шейдер вершин, заполняемый данными буфера.

Попробуем посмотреть, как все это осуществляется, с помощью простейшего примера, проекта каталога Ex06, в котором посередине окна рисуется точка.

Чтобы отобразить точку на плоскости, нам достаточно задать *две* ее координаты, но мы должны придерживаться правила об использовании форматов вершин из определенного набора. Минимальный набор характеристик вершины включает в себя *три* координаты вершины в пространстве. Хотя наши построения будут выполняться на плоскости, и нет нужды в третьей координате, мы просто обязаны указывать ее. Если же мы хотим опираться в плоскостных построениях на оконные координаты, вершины должны включать дополнительную характеристику, служащую индикатором таких построений, и называемую RHW (Reciprocal Homogeneous W). Нами она также явно не используется, но присутствовать должна.

Формат описания вершины вводится клиентом; все атрибуты должны быть типа `Single`:

```
type
  TCUSTOMVERTEX = packed record
    X, Y, Z, RHW : Single;
  end;
```

Переменная `VPoint` этого типа введена в программе для хранения характеристик нашей точки. Также нам требуется объект буфера вершин:

```
FD3DVB          : IDIRECT3DVERTEXBUFFER8;
```

По окончании работы программы с ним производятся обычные манипуляции, связанные с высвобождением памяти, а создается этот объект вызовом специального метода объекта устройства в отдельной функции инициализации:

```
function TfrmD3D.InitPoint : HRESULT;
var
  pVertices : PByte;
  hRet : HRESULT;
```

```
begin
  // Задаем координаты точки, опираемся на оконные координаты
  with VPoint do begin
    X := 150.0;
    Y := 150.0;
    Z := 0.0;
    RHW := 0.0;
  end;
  // Создание буфера вершин
  hRet := FD3DDevice.CreateVertexBuffer(SizeOf(VPoint),
                                        D3DUSAGE_WRITEONLY, D3DFVF_XYZRHW,
                                        D3DPOOL_DEFAULT, FD3DVB);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Запираем буфер
  hRet := FD3DVB.Lock(0, SizeOf(VPoint), pVertices, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Заполняем данными о вершине
  Move (VPoint, pVertices^, SizeOf(VPoint));
  hRet := FD3DVB.Unlock;        // Отпираем буфер вершин
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Связываем буфер вершин с потоком данных
  hRet := FD3DDevice.SetStreamSource(0, FD3DVB, SizeOf(TCUSTOMVERTEX));
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Устанавливаем вершинный шейдер
  Result := FD3DDevice.SetVertexShader(D3DFVF_XYZRHW);
end;
```

Разберем подробнее действия, выполняемые программой. Как мы уже выяснили, размеры клиентской области окна в момент инициализации графической системы задают область вывода, размеры и координаты. Размеры окна я установил 300×300 пикселов, поэтому координаты точки посередине окна — (150, 150). Координаты построений опираются на левый верхний угол окна: если координату X увеличить на 50, точка сдвинется на 50 пикселов вправо. Значения последних двух полей `VPoint` безразличны.

Структура, содержащая характеристики вершины, заполнена. Однако напрямую она для построений не используется. Этими данными должен заполняться буфер вершин. Сначала буфер вершин должен быть создан методом `CreateVertexBuffer` объекта устройства. Первый аргумент метода — размер буфера вершин, в байтах. Второй аргумент — флаг либо комбинация флагов, задает параметры работы с буфером. Используемый здесь флаг `D3DUSAGE_WRITEONLY` информирует систему, что нам не потребуется чтение содержимого буфера. Такой режим наиболее оптимальный.

Третий параметр — комбинация FVF-флагов, определяет формат вершин. Флаг `D3DFVF_XYZRHW` соответствует используемому нами формату из четырех чисел, координаты опираются на систему координат, связанную с окном.

Четвертый аргумент рассматриваемого метода позволяет задавать месторасположение создаваемого буфера, при использовании константы `D3DPOOL_DEFAULT` буфер будет расположен в видеопамяти, если это возможно.

Последним аргументом передается переменная, в которую будет помещен результат — создаваемый объект типа `IDIRECT3DVERTEXBUFFER8`.

Заполнение буфера конкретными данными производится при его закрытом состоянии, метод `Lock` приводит к запиранию буфера.

Первый аргумент метода — смещение относительно начала буфера. Передаваемый в качестве аргумента ноль приводит к запиранию буфера с самого начала. Второй аргумент метода задает размер запираемой области. Здесь может быть ноль для того, чтобы явно запереть всю память буфера. Третий параметр — возвращаемый методом адрес запираемой области памяти. Последний, четвертый аргумент, обычно задается нулевым, если нет необходимости использовать особые режимы запирания, такие, как режим "только для чтения".

Буфер заперт, по полученному адресу заносятся данные из нашей переменной `VPoint`, используется процедура `Move`. После этого буфер отпирается, вызывается метод `UnLock` буфера.

Далее необходимо связать поток данных, поступающих в объект устройства, используя метод `SetStreamSource`. Поток определен как однородный массив данных, где каждый компонент состоит из единственного элемента. Метод не приводит непосредственно к какому-либо действию, чтение данных из потока будет осуществляться при воспроизведении. Первый аргумент — идентификатор потока, как правило, задается нулевым (присутствует единственный поток). Второй аргумент — буфер вершин, с которым ассоциируется поток. Последний аргумент — размер порции данных.

Завершающее действие, которое следует выполнить в коде инициализации — определиться с вершинным шейдером. Для предопределенных форматов вершин нужно только вызывать метод `SetVertexShader` объекта

устройства. У метода единственный аргумент — FVF-флаг, то же значение, что и использованное при создании буфера вершин (материал книги ограничивается только таким использованием шейдеров).

В инициализации выполнены все необходимые действия, код воспроизведения должен дополниться действиями, связанными с отображением точки на экране. После очистки заднего буфера вызывается связанный с воспроизведением метод объекта устройства:

```
hRet := FD3DDevice.DrawPrimitive(D3DPT_POINTLIST, 0, 1);
```

Первый аргумент — идентификатор нужного примитива. Для вывода точки используется константа `D3DPT_POINTLIST`. Второй и третий аргументы метода задают интервал считываемых из потока примитивов, в примере берется первый и единственный примитив.

Как видим, код непосредственного воспроизведения выглядит очень просто, чего нельзя сказать об инициализации. Именно при вызове метода `DrawPrimitive` происходит обращение к потоку данных, поэтому в коде инициализации вызов метода `SetStreamSource` можно ставить в любом месте после создания буфера. Но, конечно, буфер выбора желательно держать в запертом состоянии максимально короткое время.

В примере я сознательно допустил небольшое упрощение. Процесс непосредственного воспроизведения разработчики рекомендуют обрамлять двумя действиями. Перед первым вызовом метода `DrawPrimitive` необходимо вызывать метод `BeginScene` объекта устройства, после воспроизведения вызывается его метод `EndScene`. У обоих методов отсутствуют параметры. При вызове первого из них программа информирует устройство, что следует подготовиться к воспроизведению. Второй метод сообщает устройству о том, что процесс воспроизведения для текущего кадра закончен. Это парные действия, и если использован один из методов, второй не должен быть пропущен.

Скорее всего, вы не заметите разницы в работе программы, если не станете использовать эти командные скобки. Но я в примерах книги буду неукоснительно следовать рекомендациям разработчиков.

Точки

Примитив точка, соответствующий использованию константы `D3DPT_POINTLIST` в качестве первого аргумента метода `DrawPrimitive`, приводит к тому, что для каждой порции данных, считываемых из потока, на экране вывода ставится точка.

Изучим основательнее этот примитив на примере проекта каталога Ex07, где экран усеивается множеством точек (рис. 7.2).

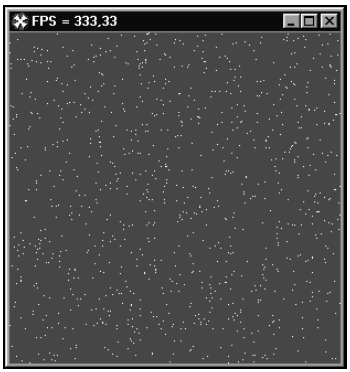


Рис. 7.2. Пример использования множества вершин

Теперь нам требуется массив, хранящий данные о вершинах:

```
const
  MAXPOINTS = 1000;    // Количество точек
var
  VPoints : Array [0..MAXPOINTS — 1] of TCUSTOMVERTEX; // Массив точек
```

Координаты точек берутся случайно, в пределах, обусловленных размерами клиентской области окна приложения:

```
Randomize;       // Инициализируем генератор случайных чисел
for i := 0 to MAXPOINTS — 1 do             // Цикл по точкам
  with VPoints [i] do begin
    X := random (300);
    Y := random (300);
    Z := 0.0;
    RHW := 0.0;
end;
```

Все остальные изменения кода предыдущей программы по большей части связаны только с изменением имени переменной, хранящей вершины. Лишь в функции воспроизведения произошли наиболее важные перемены:

```
hRet := FD3DDevice.BeginScene;   // Информируем устройство о готовности
if FAILED(hRet) then begin       // к воспроизведению
  Result := hRet;
  Exit;
end;
// Последний аргумент — количество используемых точек
hRet := FD3DDevice.DrawPrimitive(D3DPT_POINTLIST, 0, MAXPOINTS);
if FAILED(hRet) then begin
  Result := hRet;
  Exit;
end;
```

```
// Это действие теперь всегда будет завершать код воспроизведения
hRet := FD3DDevice.EndScene;
if FAILED(hRet) then begin
  Result := hRet;
  Exit;
end;
```

Пример совсем несложный, но может помочь уяснить или напомнить многие вещи. Измените строку собственно воспроизведения на следующий лад:

```
hRet := FD3DDevice.DrawPrimitive(D3DPT_POINTLIST, 0, random(MAXPOINTS));
```

Количество примитивов, считываемых из потока, стало теперь случайным. Посмотрите работу программы: одни точки мерцают, другие — нет. Данные, располагающиеся в конце потока, будут реже использоваться при воспроизведении, чем находящиеся ближе к его началу.

Чтобы мерцание получилось равномерным, надо выбирать случайную одну точку для воспроизведения. Например, чтобы в кадре сияло одновременно не более 200 точек, код можно скорректировать таким образом:

```
for i := 0 to 200 do begin
  hRet := FD3DDevice.DrawPrimitive(D3DPT_POINTLIST,
                                   random (MAXPOINTS), 1);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
end;
```

Но все точки располагаются пока неподвижно, попробуем передвигать примитивы.

В проекте каталога Ex08 координаты точек генерируются при каждом обновлении кадра. Перед очередным вызовом метода `DrawPrimitive` происходит обращение к пользовательской функции, заполняющей буфер вершин новыми значениями. Обратите внимание, что код не загромождается действиями, выполненными при инициализации массива точек:

```
function TfrmD3D.GenPoints : HRESULT;
var
  pVertices : PByte;
  hRet : HRESULT;
  i : Integer;
begin
  for i := 0 to MAXPOINTS — 1 do
    with VPoints [i] do begin
      X := random (300);
      Y := random (300);     // Значения остальных полей не меняем
  end;
```

```
  hRet := FD3DVB.Lock(0, SizeOf(VPoints), pVertices, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  Move (VPoints, pVertices^, SizeOf(VPoints));
  Result := FD3DVB.Unlock;
end;
```

Нет необходимости снова выполнять действия по созданию буфера вершин и инициализации шейдера, лишь заполняем буфер вершин новыми значениями.

Это был пример на хаотическое перемещение точек, а в проекте каталога Ex09 по аналогичной схеме рисуется вращающаяся спираль (рис. 7.3).

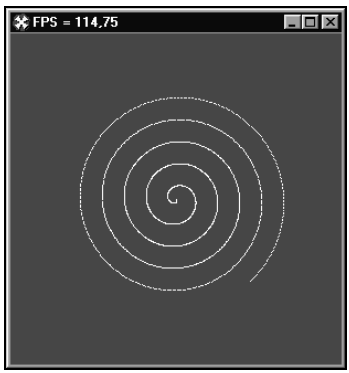


Рис. 7.3. Вращающаяся спираль строится из отдельных точек

Количество точек я взял существенно больше по сравнению с предыдущими примерами. Координаты точек спирали вычисляются при каждой перерисовке экрана:

```
var
  Angle : Single = 0.0;
function TfrmD3D.GenPoints : HRESULT;
var
  pVertices : PByte;
  hRet : HRESULT;
  i : Integer;
const
  Step = 2 * Pi / MAXPOINTS;
begin
  for i := 0 to MAXPOINTS - 1 do
    with VPoints [i] do begin
      X := 150 + cos (Angle + Step * 5 * i) * i / 20;
      Y := 150 + sin (Angle + Step * 5 * i) * i / 20;
  end;
```

```
  Angle := Angle + 0.1;
  if Angle > 2 * Pi then Angle := Angle – 2 * Pi;
  hRet := FD3DVB.Lock(0, SizeOf(VPoints), pVertices, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  Move (VPoints, pVertices^, SizeOf(VPoints));
  Result := FD3DVB.Unlock;
end;
```

Режимы воспроизведения

В этом разделе нам предстоит познакомиться с тем, как можно получать и менять характеристики воспроизведения примитивов.

Методы `GetRenderState` и `SetRenderState` объекта воспроизведения используются для получения и задания текущих режимов воспроизведения, установок, определяющих, в частности, некоторые характеристики рисования примитивов.

Познакомимся с этими методами на конкретной задаче. В проекте каталога Ex10 рисуется знакомая нам по предыдущему примеру вращающаяся спираль, но теперь со временем размеры точек изменяются (рис. 7.4).

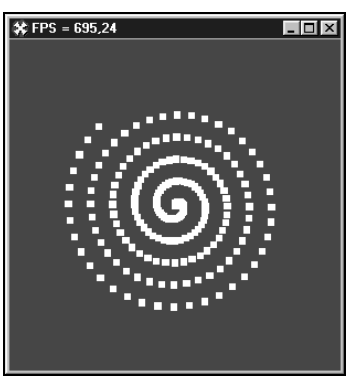


Рис. 7.4. Пример изменения размеров точек

В переменную `StartPointSize` типа `DWORD` при инициализации помещаю значение, соответствующее размеру точек, принятому по умолчанию:

```
FD3DDevice.GetRenderState(D3DRS_POINTSIZE, StartPointSize);
```

Проверку корректности опускаю. Первый аргумент метода — символическая константа, определяющая, какой режим опрашивается. Второй — переменная, в которую помещается результат. Вы можете получить список возмож-

ных режимов из файла справки по DirectX, либо обратиться к содержимому модуля DirectXGraphics.pas. Имена этих констант начинаются с префикса "D3DRS_"; по имени константы обычно становится понятно, о каком режиме идет речь.

Для изменения размеров точек увеличиваю текущее значение на небольшое значение. После того как достигнут некоторый предел, задается первоначальный размер точек:

```
FD3DDevice.GetRenderState(D3DRS_POINTSIZE, PointSize);
FD3DDevice.SetRenderState(D3DRS_POINTSIZE, trunc (1.001 * PointSize));
if PointSize > 1.02 * StartPointSize
  then FD3DDevice.SetRenderState(D3DRS_POINTSIZE, StartPointSize);
```

Первым аргументом метода SetRenderState является точно такая же константа, что и для метода GetRenderState; второй аргумент теперь содержит устанавливаемое значение.

Тип второго аргумента обоих методов должен быть именно DWORD, хотя само значение может трактоваться как булево или вещественное. Например, для размера точки значение по умолчанию устанавливается как 1.0, т. е. вещественное число. Записываемое число тоже интерпретируется как вещественное, но для осмысленной манипуляции с параметром надо выполнять преобразование. Если необходимо увеличить размер точки в два раза по сравнению со значением, принятым по умолчанию, следует объявить отдельную переменную типа Single и преобразовать ее значение в тип DWORD:

```
wrk := 2.0;
FD3DDevice.SetRenderState(D3DRS_POINTSIZE, PDWORD (@wrk)^);
```

Совет

Разработчики рекомендуют методы, изменяющие настройки воспроизведения, такие как SetRenderState, вызывать при установленном состоянии воспроизведения, т. е. после вызова метода BeginScene и до вызова метода EndScene.

Посмотрите проект каталога Ex11: работа его не отличается от предыдущего примера, но код манипуляции с размером точки гораздо понятнее:

```
var
  PointSize : Single = 1.0;
  ...
  PointSize := 1.3 * PointSize;
  if PointSize > 10.0 then PointSize := 1.0;
    FD3DDevice.SetRenderState(D3DRS_POINTSIZE, PDWORD (@PointSize)^);
```

Второй аргумент методов для некоторых режимов служит булевым флагом, включающим или отключающим режим. В этом случае аргумент может записываться так: DWORD (True).

Вы наверняка обратили внимание, что при увеличении размера точки превращаются в квадратики. Этим можно пользоваться, чтобы нарисовать подобные примитивы, но увеличивать размер точки можно лишь до некоторого предела, поэтому таким образом получится изобразить только небольшие квадраты.

Блоки установок

Direct3D позволяет запоминать и именовать наборы установок воспроизведения, при этом необходимый режим назначается вызовом одной команды. В проекте каталога Ex12 экран усеивается хаотически располагающимися точками. Нововведение здесь в том, что точки различаются по размеру: одна половина из них имеет размер, в три раза превышающий принятый по умолчанию, а другая — размер, в четыре раза больше обычного (рис. 5).

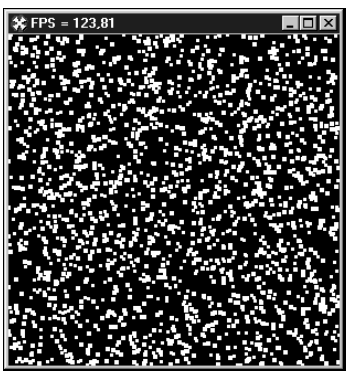


Рис. 7.5. Нововведение в примере: точки различаются по размеру

При инициализации создаем два блока установок для различных размеров точек. Идентификаторами являются переменные типа `DWORD`, значения которых задаются системой:

```
PointSize := 4.0;              // Устанавливаемый размер точек
with FD3DDevice do begin
  BeginStateBlock;             // Начало описания блока установок
  SetRenderState(D3DRS_POINTSIZE, PDWORD (@PointSize)^);
  EndStateBlock (PointSize4); // Конец описания блока установок
end;
PointSize := 3.0;            // Второй блок установок, другой размер точек
with FD3DDevice do begin
  BeginStateBlock;
  SetRenderState(D3DRS_POINTSIZE, PDWORD (@PointSize)^);
  EndStateBlock (PointSize3);
end;
```

Обратите внимание, что само создание блоков установок никак не влияет на режимы воспроизведения, т. е. в данном случае размеры точек пока остаются первоначальными.

Массив вершин примитивов заполняется беспрерывно случайными координатами, а перед воспроизведением первой половины точек вызываем метод `ApplyStateBlock` и устанавливаем режимы воспроизведения в нужный набор состояний. Единственным аргументом метода является имя нужного блока:

```
// Задаем режим 3-кратного размера точек
hRet := FD3DDevice.ApplyStateBlock (PointSize3);
// Рисуем первую половину точек
hRet := FD3DDevice.DrawPrimitive(D3DPT_POINTLIST, 0, MAXPOINTS div 2);
// Устанавливаем режим 4-кратного размера точек
hRet := FD3DDevice.ApplyStateBlock (PointSize4);
// Рисуем вторую половину точек
hRet := FD3DDevice.DrawPrimitive(D3DPT_POINTLIST,
                                 (MAXPOINTS div 2) - 1, MAXPOINTS div 2);
```

Надеюсь, что все понятно, лишь ограничусь небольшим замечанием. Разработчики рекомендуют вызывать метод `ApplyStateBlock`, как и все другие методы, влияющие на режим воспроизведения, при установленном состоянии воспроизведения.

Окрашенные вершины

В предыдущих примерах рисовались точки белого цвета, теперь мы научимся окрашивать наши примитивы. Для этого необходимо задавать цвет каждой вершины примитива и использовать соответствующий FVF-флаг.

В проекте каталога Ex13 экран заполняется хаотически расположенными разноцветными точками (рис. 7.6).

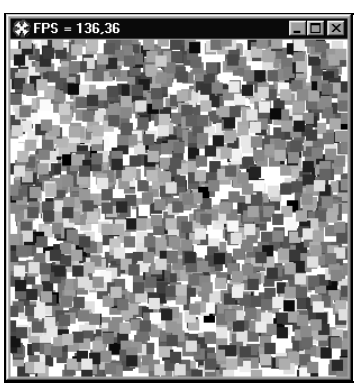


Рис. 7.6. Учимся окрашивать примитивы

Запись формата вершин дополнилась полем, хранящим цвет вершины, тип ее DWORD:

```
type
  TCUSTOMVERTEX = packed record
    X, Y, Z, RHW : Single;
    Color : DWORD;          // Добавлено новое поле
  end;
```

Поскольку FVF-флаг задается в нескольких местах кода, вводим пользовательскую константу, хранящую нужную нам комбинацию:

```
const
  D3DFVF_CUSTOMVERTEX = D3DFVF_XYZRHW or D3DFVF_DIFFUSE;
```

Порядок, в котором перечисляются эти константы, безразличен, но порядок перечисления полей в записи формата вершин предопределен графической системой. При считывании данных из потока будет подразумеваться, что первыми идут координаты вершин, за ними следует цвет (диффузная составляющая).

Итак, теперь прибавляется дополнительная константа, которую будем трактовать пока как включение режима окрашивания вершин примитивов.

При инициализации массива вершин поле цвета заполняется случайным значением:

```
for i := 0 to MAXPOINTS - 1 do
  with VPoints [i] do begin
    Z := 0.0;
    RHW := 0.0;
    Color := D3DCOLOR_XRGB(random (256), random (256), random (256));
  end;
```

В остальном код примера не содержит ничего для нас нового, поэтому разбирать его здесь не будем.

В следующем примере (проект каталога Ex14) окрашивание вершин используется для создания черно-белого изображения. Пример весьма занятный: из облака хаотически располагающихся точек выстраивается упорядоченный образ (рис. 7.7).

Изображение формируют 20 898 отдельных примитивов. Первоначально координаты их задаются хаотически, для каждой точки вычисляется шаг смещения. Текстовый файл содержит координаты окончательного положения точки. За 100 шагов каждая точка должна достичь финишного положения:

```
type
  TStep = packed record       // Тип для хранения скорости точки по осям
    StepX, StepY : Single;
  end;
```

```
var
  Steps : Array [0..MAXPOINTS — 1] of TStep;   // Шаги для каждой точки
function TfrmD3D.InitPoints : HRESULT;
var
  pVertices : PByte;
  hRet : HRESULT;
  i : Integer;
  t : TextFile;
  wrkX, wrkY : Integer;
begin
  AssignFile (t, 'points.txt');
  Reset (t);
  for i := 0 to MAXPOINTS — 1 do begin
    ReadLn (t, wrkX, wrkY);
    with VPoints [i] do begin
      X := random (240);
      Y := random (289);
      // Каждая точка должна достичь своего положения за 100 шагов
      Steps [i].StepX := (wrkX — X) / 100;
      Steps [i].StepY := (wrkY — Y) / 100;
      Z := 0.0;
      RHW := 0.0;
      Color := 0;
    end;
  end;
  CloseFile (t);
  ...
```

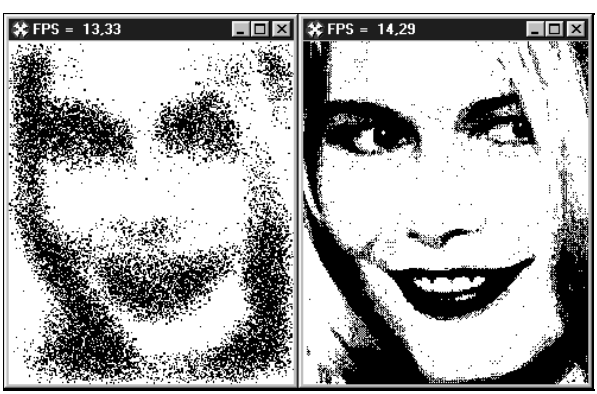


Рис. 7.7. Два момента работы примера движущихся примитивов

Переменная `PointSize` управляет текущим размером точки, первоначально ее значение установлено в 5.0. При перемещении точки размер ее последовательно уменьшается и через 100 шагов должен стать единичным:

```
function TfrmD3D.MovePoints : HRESULT;
var
  pVertices : PByte;
  hRet : HRESULT;
  i : Integer;
begin
  PointSize := PointSize — 0.04;     // Уменьшение размера точки
  FD3DDevice.SetRenderState( D3DRS_POINTSIZE, PDWORD(@PointSize)^);
  for i := 0 to MAXPOINTS — 1 do begin
    with VPoints [i] do begin
      X := X + Steps [i].StepX;     // Перемещение точки
      Y := Y + Steps [i].StepY;
    end;
  end;
```

В цикле ожидания сообщения подсчитывается количество обновлений положения точек. Для их первой сотни вызывается функция `MovePoints`. Шоу можно повторить нажатием пробела:

```
procedure TfrmD3D.FormKeyDown(Sender: TObject; var Key: Word;
                              Shift: TShiftState);
begin
  if Key = VK_ESCAPE then Close else
  if Key = VK_SPACE then begin
    InitPoints;         // Заново разбрасываем точки
    PointSize := 5.0;   // Размер точек снова пятикратный
    Count := 0;         // Очередная сотня кадров
  end;
end;
```

Следующий пример, проект из каталога Ex15, построен по аналогичной схеме, но примитивы на секунду покрывают всю клиентскую часть окна, чтобы затем снова разлететься (рис. 7.8).

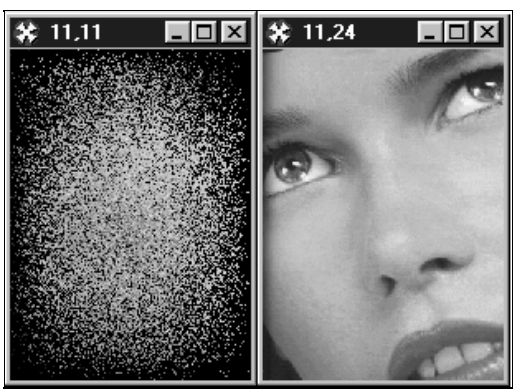


Рис. 7.8. В этом примере для каждого пиксела растра создается отдельный примитив

Для задания образа используется растровое изображение размером 200×146 пикселов, цвет каждого примитива определяется цветом пиксела растра:

```
const
  MAXPOINTS = 200 * 146;
function TfrmD3D.InitPoints : HRESULT;
var
  pVertices : PByte;
  hRet : HRESULT;
  i, j, k : Integer;
  bmp : TBitMap;
  R, G, B : Byte;
begin
  bmp := TBitMap.Create;
  bmp.LoadFromFile ('Claudia.bmp');        // Загрузка растра
  k := 0;
  for i := 0 to 199 do
    for j := 0 to 145 do begin
      with VPoints [k] do begin
        X := random (145);
        Y := random (200);
        Steps [i, j].StepX := (j - X) / 10;
        Steps [i, j].StepY := (i - Y) / 10;
        Z := 0.0;
        // Цветовые веса пиксела растра
        R := GetRValue (bmp.Canvas.Pixels [j, i]);
        G := GetGValue (bmp.Canvas.Pixels [j, i]);
        B := GetBValue (bmp.Canvas.Pixels [j, i]);
        RHW := 0.0;
        Color := D3DCOLOR_XRGB(R, G, B);   // Цвет примитива
      end;
      Inc (k);
  end;
  bmp.Free;
  ...
```

Приращения по координатам задаются так, чтобы за 10 шагов точка добралась до финиша. В этом примере точки, достигнув нужного положения, продолжают двигаться дальше, таким образом, что заветная картинка появляется только на миг. Через каждые 20 кадров направление движения точки меняется на противоположное:

```
var
  Steps : Array [0..199, 0..145] of TStep;
procedure TfrmD3D.ApplicationEvents1Idle(Sender: TObject;
                                         var Done: Boolean);
```

```
var
  hRet : HRESULT;
  i, j : Integer;
begin
  if FActive then begin
    Inc (Frames);
    hRet := Render;
    if FAILED(hRet) then begin
      FActive := False;
      ErrorOut ('Render', hRet);
      Exit;
    end;
    ThisTickCount := GetTickCount;
    if ThisTickCount - LastTickCount > 25 then begin
      Caption := Format('%6.2f',
                 [frames * 1000 / (ThisTickCount - LastTickCount)]);
      Frames := 0;
      Inc (Count);
      // Цикл движения точек в 20 кадров
      if Count <= 20 then MovePoints else begin
        for i := 0 to 199 do
          for j := 0 to 145 do begin
            Steps [i, j].StepX := -Steps [i, j].StepX;
            Steps [i, j].StepY := -Steps [i, j].StepY;
          end;
        Count := 0;
      end;
    end;
    LastTickCount := GetTickCount;
  end;
  Done := False;
end;
```

Обратите внимание, что клиентская область окна в этих двух примерах неквадратная, размеры его подогнаны для конкретных образов примеров.

Итак, мы научились из отдельных точек формировать весьма сложные образы, но такой подход годится только для простых, или тестовых примеров и небольших изображений. На количество используемых примитивов системой наложено ограничение, и работа с десятками тысяч примитивов, как мы видим из этого примера, приводит к существенному снижению FPS.

Отрезки

Для рисования отрезков в Direct3D предусмотрены два типа примитивов: *независимые отрезки* и *связанные отрезки*. Начнем постижение этой темы с первого из этой пары типа примитивов.

Для построения независимых отрезков первым аргументом метода `DrawPrimitive` указывается константа `D3DPT_LINELIST`. По считываемым попарно из потока вершинам строятся отдельные, несвязанные, отрезки прямой.

Несложный пример из каталога Ex16 является иллюстрацией на эту тему. На экране строятся два отрезка красного цвета, параллельные друг другу. Координаты вершин хранятся в четырехэлементном массиве пользовательского типа `TCUSTOMVERTEX`. Массивы заполняются тривиальным образом: значения полей первых двух элементов определяют начало и конец первого отрезка, последние два элемента массива относятся ко второму отрезку.

Обратите внимание, что собственно при построении примитивов последним аргументом передается не количество вершин, а количество примитивов:

```
hRet := FD3DDevice.DrawPrimitive(D3DPT_LINELIST, 0, 2);
```

Примечание

Если данных, поступающих из потока, недостаточно, ошибка генерироваться не станет, поскольку все недостающие данные будут считаться нулевыми.

Константа `D3DPT_LINESTRIP` является признаком другого примитива — группы связанных отрезков. В этом случае вершины, считываемые из потока, задают характеристики вершин, последовательно соединяемых отрезками прямой.

В проекте каталога Ex17 создается пятиугольник (рис. 7.9), в построении которого используется пять связанных отрезков.

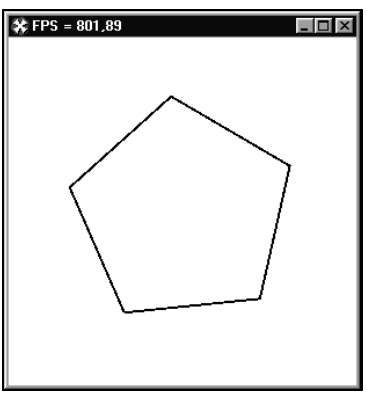


Рис. 7.9. Простой пример использования связанных отрезков

Для получения пятиугольника требуется шесть точек, координаты первой и последней из них совпадают. Чтобы оживить картинку, текущие координаты вершин опираются на увеличивающееся значение переменной `Angle`:

```
for i := 0 to 5 do
  with VPoints [i] do begin
```

```
    X := 150 + cos (Angle + i * 2 * Pi / 5) * Radius;
    Y := 150 + sin (Angle + i * 2 * Pi / 5) * Radius;
  end;
```

Обращаю внимание на параметры метода воспроизведения примитивов:

```
hRet := FD3DDevice.DrawPrimitive(D3DPT_LINESTRIP, 0, 5);
```

Надеюсь, остальной код вопросов у вас не вызывает.

Теперь нам стоит обсудить, как воспроизводить одновременно несколько независимых групп примитивов. Организовать такое воспроизведение можно разными способами: хранить вершины в одном буфере, либо использовать отдельные буферы для каждой группы вершин.

Разберем первый вариант на примере проекта каталога Ex18. На экране вращаются два многоугольника: пятиугольник и квадрат (рис. 7.10).

Массив `VPoints` хранит координаты 11 вершин: первые 6 связаны с пятиугольником, оставшиеся предназначены для построения квадрата.

Квадрат и пентагон[1] вращаются в противоположные стороны с различными скоростями:

```
for i := 0 to 5 do          // Первыми хранятся координаты вершин пентагона
  with VPoints [i] do begin
    X := 150 + cos (Angle + i * 2 * Pi / 5) * Radius;
    Y := 150 + sin (Angle + i * 2 * Pi / 5) * Radius;
  end;
for i := 0 to 4 do          // Координаты вершин квадрата
  with VPoints [6 + i] do begin
    // Скорость вращения квадрата удвоена
    X := 150 + cos (- 2 * Angle - i * Pi / 2) * Radius / 2;
    Y := 150 + sin (- 2 * Angle - i * Pi / 2) * Radius / 2;
  end;
```

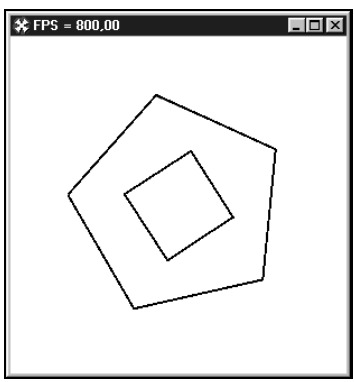


Рис. 7.10. Независимые группы примитивов

[1] То есть пятиугольник. — *Ред.*

Собственно при построении к методу `DrawPrimitive` обращаемся дважды, поскольку строим две независимые фигуры. Обратите внимание на значение второго аргумента метода:

```
hRet := FD3DDevice.DrawPrimitive(D3DPT_LINESTRIP, 0, 5);
if FAILED(hRet) then begin
  Result := hRet;
  Exit;
end;
hRet := FD3DDevice.DrawPrimitive(D3DPT_LINESTRIP, 6, 4);
if FAILED(hRet) then begin
  Result := hRet;
  Exit;
end;
```

В следующем примере, проекте каталога Ex19, осуществляется точно такое же построение, однако координаты вершин многоугольников хранятся в отдельных массивах. Как следствие, по ходу воспроизведения кадра необходимо переключать источники потоков. Чтобы не загромождать страницы книги однообразным кодом, подробно разбирать здесь этот пример не будем и оставим его для вашей самостоятельной работы, он совершенно несложен для этого.

В проекте каталога Ex20 строятся замысловатые построения, создающие иллюзию пространственных поверхностей (рис. 7.11).

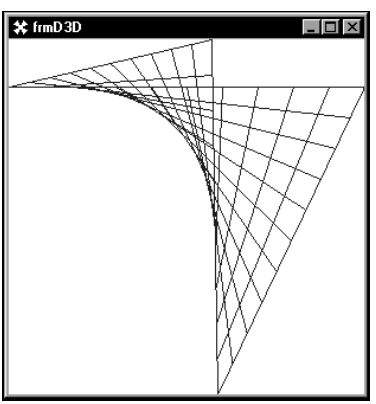


Рис. 7.11. Сетка проекции трехмерной поверхности

Пример построен по очень простому алгоритму: 22 отрезка соединяют узлы сетки, угловые точки которой разбросаны случайно:

```
for i := 0 to 10 do begin              // Первый набор отрезков сетки
  with VPoints [i * 2] do begin        // Начало отрезка
    X := X1 + i * (X2 - X1) / 10;      // Разбиение на 10 точек
    Y := Y1 + i * (Y2 - Y1) / 10;
  end;
```

```
  with VPoints [i * 2 + 1] do begin   // Конец отрезка
    X := X3 + i * (X4 - X3) / 10;
    Y := Y3 + i * (Y4 - Y3) / 10;
  end;
end;
for i := 0 to 10 do begin             // Второй набор отрезков сетки
  with VPoints [i * 2 + 22] do begin
    X := X1 + i * (X3 - X1) / 10;
    Y := Y1 + i * (Y3 - Y1) / 10;
  end;
  with VPoints [i * 2 + 1 + 22] do begin
    X := X2 + i * (X4 - X2) / 10;
    Y := Y2 + i * (Y4 - Y2) / 10;
  end;
end;
```

Угловые точки перемещаются с течением времени, отскакивая от границ области вывода. В примере после нажатия клавиши <Пробел> координаты этих точек заново инициализируются. Стоит сказать, что некоторые комбинации положений порождают очень интересные "поверхности".

Треугольник

Если первым аргументом метода `DrawPrimitive` указана константа `D3DPT_TRIANGLELIST`, то каждая триада вершин, считываемых из потока, задает три вершины независимого треугольника.

Посмотрите проект каталога Ex21, простейший пример на тему построения треугольника. При работе программы на экране вращается треугольник красного цвета. Программа написана по той же схеме, что и предыдущие примеры: задаются координаты трех вершин треугольника, вызывается метод `DrawPrimitive` с соответствующими аргументами:

```
hRet := FD3DDevice.DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
```

Треугольник является базовой фигурой для построений. Именно с его помощью и осуществляется большинство построений в Direct3D. Квадраты, прямоугольники и вообще все остальные фигуры рисуются из треугольников.

Посмотрите проект каталога Ex22, где из отдельных независимых треугольников строится пятиконечная звезда, плавно изменяющаяся в размерах по ходу своего вращения (рис. 7.12).

Код этого примера также не должен вызывать у вас трудностей при изучении, поэтому ограничусь лишь замечанием о том, что первые пять соприкасающихся треугольников образуют внутренний пентагон, вторая половина примитивов создает лучи звезды.

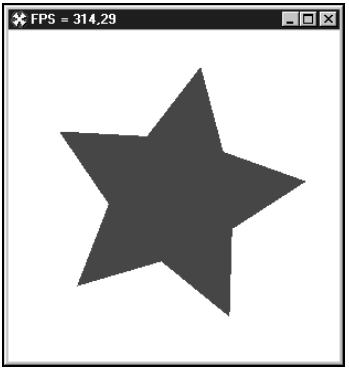


Рис. 7.12. Десять независимых треугольников образуют невыпуклый многоугольник

Попутно с рассмотрением примитивов Direct3D отвлечемся немного на некоторые важные вопросы.

При необходимости, содержимое экрана воспроизведения может быть легко записано в растр или любой другой стандартный графический формат. Точно так же, как мы поступали с приложениями, использующими DirectDraw, для этого потребуется в канву вспомогательного объекта класса `TBitmap` скопировать с помощью функции `BitBlt` содержимое канвы формы и вызвать метод записи в файл.

С созданием видео тоже проблем возникать не должно, поскольку в рассмотренном нами способе кадры создаваемого фильма представляют собой список объектов класса `TBitmap`, а при копировании в его канву содержимого формы, как я только что сказал, в 16-битном и выше режимах проблем не возникает.

Значение пикселов канвы формы согласуется с выводом, производимым DirectX, что можно применять для простейшего выбора объектов, похожего на использованный нами в DirectDraw. Если в этом примере при нажатии кнопки мыши требуется определить, что находится под курсором, то обработчик нужного события можно записать так:

```
procedure TfrmD3D.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
var
  R, G, B : Byte;
begin
  R := GetRValue (Canvas.Pixels [X, Y]);
  G := GetGValue (Canvas.Pixels [X, Y]);
  B := GetBValue (Canvas.Pixels [X, Y]);
  if R = 0
    then ShowMessage ('Под курсором звездочка')
    else ShowMessage ('Под курсором фон')
end;
```

Фон в примере белый, поэтому доля чистого красного (или зеленого) цвета будет нулевой только для пикселов звездочки.

Этот простой прием выбора по цвету можно использовать для более тонкого отделения цветов. Например, нам необходимо рассмотреть вариант, когда пользователь выбирает луч звезды. Окрасим вершины треугольников, образующие лучи, в оттенок синего:

```
Color := D3DCOLOR_XRGB(0, 0, 254);
```

А внутренний пентагон по-прежнему будем заполнять чистым синим цветом. Столь малая разница в оттенках зрителем совершенно не будет ощущаться и позволит нам точнее разделять две группы объектов для выбора:

```
if R = 0 then begin
  if B = 255          // Чистый синий — у лучей звезды
    then ShowMessage ('Под курсором луч')
    else ShowMessage ('Под курсором пентагон')
  end
else ShowMessage ('Под курсором фон');
```

Аналогично, если надо различать выбор для каждого отдельного луча звезды, окрашиваем их в индивидуальные оттенки, по значению которых и ориентируемся в выборе пользователя. Таким образом, можно предлагать для выбора очень много комплексных или одиночных объектов, предел на их количество — чувствительность зрителя.

Позже мы вернемся к теме выбора объектов, а сейчас немного поговорим на тему закрашивания примитивов.

Посмотрите работу примера из каталога Ex23, возвращающего нас к предыдущему проекту с одиночным треугольником. Небольшое отличие в коде данного примера заключается в том, что вершины треугольника окрашены в различные чистые цвета. Интересно же в примере то, что цвета вершин треугольника интерполируются, отчего при окрашивании получается красивый градиентный переход (рис. 7.13).

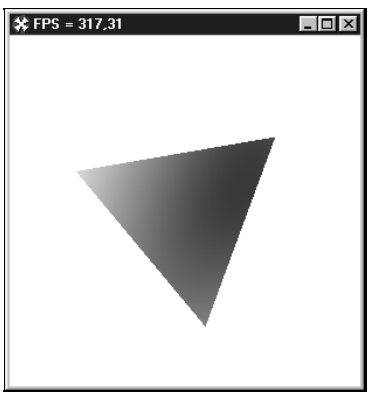


Рис. 7.13. По умолчанию в Direct3D установлена закраска Гуро

Direct3D по умолчанию назначает закраску Гуро — быстрый алгоритм интерполяции цветов вершин треугольника. Также зарезервирована возможность использования закраски Фонга, но пока этот способ системой не поддерживается.

Поменять схему окрашивания или тонирования примитивов возможно с помощью знакомого уже метода задания режимов воспроизведения. Например, чтобы отказаться от интерполяции цветов, надо записать следующую строку:

```
FD3DDevice.SetRenderState(D3DRS_SHADEMODE, D3DSHADE_FLAT);
```

В этом случае цвет первой вершины треугольника будет определять цвет всего примитива.

Вторым аргументом для указанного режима могут использоваться также константы `D3DSHADE_GOURAUD` и `D3DSHADE_PHONG`. Второй случай пока аналогичен отказу от интерполяции.

Еще одним режимом воспроизведения, на который необходимо обязательно обратить внимание, является режим `D3DRS_FILLMODE`. По умолчанию действует твердотельный режим, примитивы выводятся заполненными. Этому режиму соответствует константа `D3DFILL_SOLID`. Для установления проволочного, каркасного режима воспроизведения необходимо вторым аргументом метода `SetRenderState` задавать другую константу:

```
FD3DDevice.SetRenderState(D3DRS_FILLMODE, D3DFILL_WIREFRAME);
```

При проволочном режиме вложенные в треугольник объекты не воспроизводятся, рисуются только отрезки, образующие их контуры. Иллюстрацией применения этого метода служит проект каталога Ex24 — простое продолжение примера со звездой (рис. 7.14).

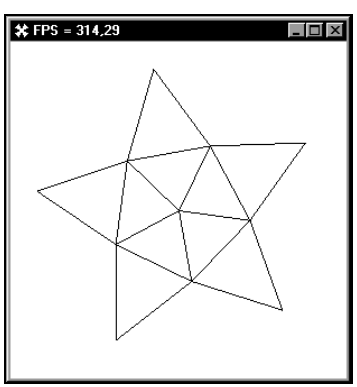


Рис. 7.14. Иллюстрация проволочного режима воспроизведения

Если для этого режима использовать константу `D3DFILL_POINT`, то при воспроизведении станут выводиться только точки вершин примитивов.

Продолжаем изучать примитивы Direct3D. Группе связанных треугольников соответствует флаг `D3DPT_TRIANGLESTRIP`. Первые три вершины задают первый треугольник, вторая, третья и четвертая определяют второй треугольник, третья, четвертая и пятая — третий и т. д. Получается лента соприкасающихся треугольников (рис. 7.15).

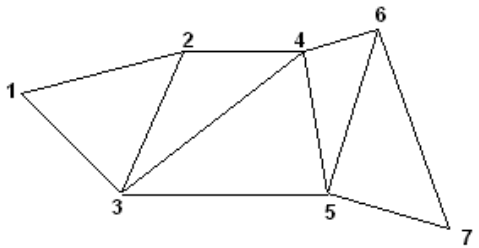


Рис. 7.15. Принцип построения ленты треугольников по данным потока

Использование связанных треугольников — самый экономный и эффективный способ построений. К примеру, если для рисования прямоугольника независимыми треугольниками потребуется задать координаты шести точек, то при использовании связанных треугольников достаточно задать четыре точки.

Для закрепления изученного материала решим следующую задачу: требуется нарисовать диск; значение константы `Level` определяет количество используемых в разбиении треугольников.

Поскольку лента в этой задаче замкнута, вершин потребуется на пару больше, чем значение `Level`:

```
VPoints : Array [0..Level + 1] of TCUSTOMVERTEX;
```

Для построения диска берем попарно точки, лежащие на внутренней и внешней границах диска:

```
i := 0;
repeat
  with VPoints [i] do begin              // Внутренняя граница диска
    X := 150 + cos (Angle + i * 2 * Pi / Level) * Radius / 2;
    Y := 150 + sin (Angle + i * 2 * Pi / Level) * Radius / 2;
    Color := D3DCOLOR_XRGB(255, 0, 0);  // Красного цвета
  end;
  with VPoints [i + 1] do begin          // Внешняя граница диска
    X := 150 + cos (Angle + i * 2 * Pi / Level) * Radius;
    Y := 150 + sin (Angle + i * 2 * Pi / Level) * Radius;
    Color := D3DCOLOR_XRGB(0, 0, 255);  // Синего цвета
  end;
  Inc (i, 2);                // Переходим к следующей паре вершин
until i > Level;
```

Окончательное решение задачи можете посмотреть в каталоге Ex25, результат работы которого в проволочном режиме представлен на рис. 7.16.

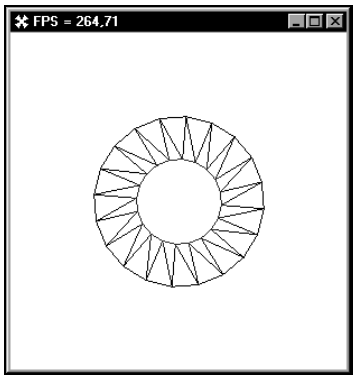


Рис. 7.16. Диск строится лентой треугольников

Обязательно посмотрите работу данного примера с отключенной интерполяцией цветов, в этом случае получается чередование красных и синих треугольников, каждая четная и нечетная вершины, различающиеся цветом, являются первыми для построения очередного треугольника.

Раз мы умеем строить закрашенный прямоугольник, то мы можем попробовать свои силы в решении классической задачи компьютерной графики — рисование пламени. Проект, располагающийся в каталоге Ex26, является решением этой задачи, во время его работы внизу экрана поднимаются языки пламени, в верхней части экрана появляется падающая горящая частица.

Изображение строится по отдельным квадратикам, размеры которых можно варьировать:

```
type
  TRGB = packed record       // Запись цвета
     R, G, B : BYTE;
  end;
const
  Size = 2;   // Размер отдельного квадратика, "пиксела"
  Fade = 4;   // Степень затухания пламени
  NumX = 150; // Количество квадратиков по горизонтали
  NumY = 150; // Количество квадратиков по вертикали
var
  Fire : Array [1..NumX, 1..NumY + 1] of TRGB;  // Цвета узлов сетки
  PreF : Array [1..NumX] of TRGB; // Вспомогательный массив первой строки
  Angle : Single = 0.0;           // для движения падающей точки
  ParticleX : Integer = 0;        // Координаты точки
  ParticleY : Integer = NumY;
```

Следующая пользовательская функция выводит один квадрат, цвета углов которого задаются текущими значениями элементов массива `Fire`:

```
function TfrmD3D.DrawPix(const inX, inY : Integer) : HRESULT;
var
  pVertices : PByte;
  hRet : HRESULT;
begin
  with VPoints [0] do begin    // Левый нижний угол квадрата
    X := inX * Size;
    Y := 300 - inY * Size;     // Переворачиваем ось Y
    Color := D3DCOLOR_XRGB(Fire[inX, inY + 1].R, Fire[inX, inY + 1].G,
                           Fire[inX, inY + 1].B);
  end;
  with VPoints [1] do begin    // Левый верхний угол квадрата
    X := inX * Size;
    Y := 300 - (inY + 1) * Size;
    Color := D3DCOLOR_XRGB(Fire[inX, inY].R, Fire[inX, inY].G,
                           Fire[inX, inY].B);
  end;
  with VPoints [2] do begin    // Правый нижний угол квадрата
    X := (inX + 1) * Size;
    Y := 300 - inY * Size;
    Color := D3DCOLOR_XRGB(Fire[inX + 1, inY + 1].R, Fire[inX + 1,
                           inY + 1].G, Fire[inX + 1, inY + 1].B);
  end;
  with VPoints [3] do begin    // Правый верхний угол квадрата
    X := (inX + 1) * Size;
    Y := 300 - (inY + 1) * Size;
    Color := D3DCOLOR_XRGB(Fire[inX + 1, inY].R, Fire[inX + 1, inY].G,
                           Fire[inX + 1, inY].B);
  end;
  hRet := FD3DVB.Lock(0, SizeOf(VPoints), pVertices, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  Move (VPoints, pVertices^, SizeOf(VPoints));
  hRet := FD3DVB.Unlock;
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  Result := FD3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
end;
```

Для формирования собственно пламени последняя строка инициализируется случайным оттенком красного, в последующих пикселах цвет затухает, а для создания эффекта размытия языков пламени используется простой прием с усреднением цветов соседних точек:

```
procedure TfrmD3D.DrawFire;
var
  i, j : Integer;
  f : Byte;
begin
  // Инициализация последней строки экрана
  for i := 2 to NumX-1 do begin
    f := random(255);
    PreF[i].R := 255;
    PreF[i].G := trunc (f / 1.4);
    PreF[i].B := f div 2;
  end;
  // Заполняем в массиве Fire последнюю строку
  // усредненными значениями соседних элементов PreF
  for i := 2 to NumX — 1 do begin
    Fire[i, 1].R := (PreF[i — 1].R + PreF[i + 1].R + PreF[i].R) div 3;
    Fire[i, 1].G := (PreF[i — 1].G + PreF[i + 1].G + PreF[i].G) div 3;
    Fire[i, 1].B := (PreF[i — 1].B + PreF[i + 1].B + PreF[i].B) div 3;
  end;
  // Смешивание, усреднение значений пикселов по экрану
  for j := NumY — 1 downto 2 do
    for i := 2 to NumX — 1 do begin
      Fire[i,j].R := (Fire[i-1, j].R   + Fire[i+1, j].R + Fire[i,j].R +
                      Fire[i-1, j-1].R + Fire[i+1, j-1].R +
                      Fire[i, j-1].R) div 6;
      Fire[i,j].G := (Fire[i-1, j].G   + Fire[i+1, j].G + Fire[i,j].G +
                      Fire[i-1, j-1].G + Fire[i+1, j-1].G +
                      Fire[i, j-1].G) div 6;
      Fire[i,j].B := (Fire[i-1, j].B   + Fire[i+1, j].B + Fire[i,j].B +
                      Fire[i-1, j-1].B + Fire[i+1, j-1].B +
                      Fire[i, j-1].B) div 6;
  end;
  // Квадратик, соответствующий падающей частице
  for j := ParticleY — 1 to ParticleY  do
    for i := ParticleX — 1 to ParticleX  do begin
      Fire[i, j].R := 255;
      Fire[i, j].G := 0;
      Fire[i, j].B := 0;
    end;
  // Вывод квадратиков содержимого экрана
  for j := 2 to NumY — 1 do
```

```
    for i := 2 to NumX — 1 do
      DrawPix (i — 1, j — 1);
  // Затухание оттенков по мере подъема языков пламени
  for j := NumY downto 2 do
    for i := 1 to NumX do begin
      if Fire[i, j — 1].R >= Fade
        then Fire[i, j].R := Fire[i, j — 1].R — Fade
        else Fire[i, j].R := 0;
      if Fire[i, j — 1].G >= Fade
        then Fire[i, j].G := Fire[i, j — 1].G — Fade
        else Fire[i, j].G := 0;
      if Fire[i, j — 1].B >= Fade
        then Fire[i, j].B := Fire[i, j — 1].B — Fade
        else Fire[i, j].B := 0;
  end;
end;
```

Последний примитив, связанный с флагом `D3DPT_TRIANGLEFAN`, также предназначен для построения группы связанных треугольников, но данные потока здесь трактуются немного иначе, чем в предыдущем случае: вторая, третья и первая вершины определяют первый треугольник, третья, четвертая и первая ассоциированы со вторым треугольником, и т. д. То есть треугольники связаны, подобно раскрою зонтика или веера. Первая вершина потока ассоциирована с центральной точкой (рис. 7.17), а порядок перечисления вершин особенно важен для режима отключенной интерполяции цветов вершин.

По такой схеме удобно строить конусы и пирамиды, а для плоскостных построений — выпуклые многоугольники, эллипсы и окружности. Приведу тривиальный пример на этот случай (проект из каталога Ex27).

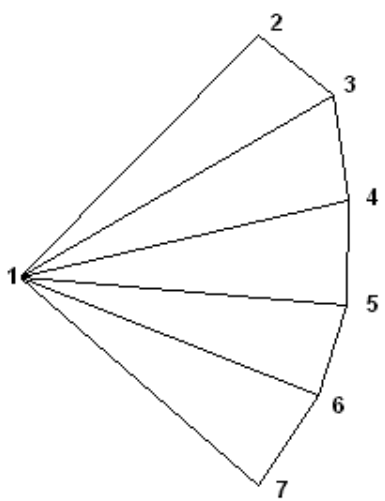


Рис. 7.17. Принцип использования вершин потока для примитива `D3DPT_TRIANGLEFAN`

Значение константы `Level` задает степень разбиения полного круга, количество вершин нам требуется на пару больше этого значения:

```
const
  Level = 255;
var
  VPoints : Array [0..Level + 1] of TCUSTOMVERTEX;
```

Первая вершина массива хранит координаты центральной точки круга, все остальные равномерно располагаются на его границе:

```
const
  Step = 2 * Pi / Level;
  ...
  with VPoints [0] do begin      // Первая точка — центр круга
    X := 150;
    Y := 150;
    Color := D3DCOLOR_XRGB(0, 0, 0);
  end;
  for i := 1 to Level + 1 do     // Точки на краю круга
    with VPoints [i] do begin
      X := 150 + cos (Angle + i * Step) * Radius;
      Y := 150 + sin (Angle + i * Step) * Radius;
      Color := D3DCOLOR_XRGB(0, trunc(i * 255 / Level), 0);
  end;
```

Для каждой вершины последовательно увеличивается вес зеленой составляющей цвета. Для последней точки он принимает максимальное значение при произвольной величине константы `Level`. Градиент зеленого цвета я взял для того, чтобы получить в итоге некое подобие экрана радара (рис. 17.8).

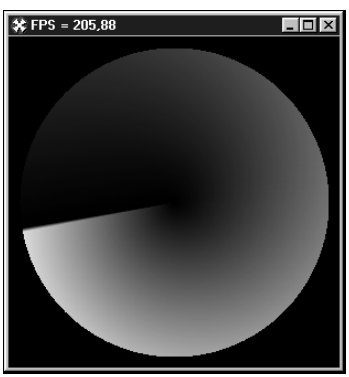


Рис. 7.18. Пример использования примитива `D3DPT_TRIANGLEFAN`

Оконные приложения, использующие Direct3D, безболезненно переживают ситуации потери фокуса и восстановления, но осталась еще одна исключи-

тельная ситуация — спящий режим. Возврат из этого режима гарантированно приведет к потере способности воспроизведения нашим приложением.

Для предупреждения таких исключений можно воспользоваться методом `TestCooperativeLevel` объекта устройства. Метод возвращает значение `D3DERR_DEVICELOST` в ситуации, когда устройство вывода недоступно, например, в спящем состоянии. Другое, кроме успешного, возвращаемое методом значение — `D3DERR_DEVICENOTRESET`, соответствует ситуации, когда устройство, в принципе, готово, но воспроизведение невозможно.

На примере этого проекта рассмотрим, как пользоваться данным методом, чтобы оконные приложения смогли пережить спящий режим.

Код обработчика цикла ожидания приведите к следующему виду:

```
if FActive then begin
  Inc (Frames);
  // Определяем состояние устройства
  hRet := FD3DDevice.TestCooperativeLevel;
  if hRet = D3DERR_DEVICELOST
    // Сейчас устройство не готово, воспроизведение невозможно
    then Exit
    // Выход из спящего режима
    else if Failed(hRet) then begin
    // Заново инициализируем систему
      InitD3D;
      InitPoints;
    end;
    // Воспроизведение осуществляем без проверки исключений
    Render;
   ...
```

То есть при выходе из спящего режима необходимо повторно инициализировать графическую систему и заново подготовить буфер вершин.

Полноэкранный режим

Конечно, для многих ваших приложений потребуется именно полноэкранный режим, поэтому мы изучим нюансы, связанные с использованием Direct3D в таком режиме. Как обычно для этой книги, рассмотрим особенности на конкретном примере (проект каталога Ex28) модифицированного варианта вращающейся звезды. Теперь звезда вращается в полноэкранном режиме.

У формы поменялось значение свойства `BorderStyle`: чтобы окно приложения не просвечивало, реагируя на нахождение курсора вблизи границ, это свойство установлено в значение `bsNone`.

При инициализации графической системы нам требуется определить формат пиксела, вспомогательный массив содержит возможные значения формата, а метод CheckDeviceType главного объекта позволяет определить, какое значение подходит для текущих установок рабочего стола:

```
const // Возможные форматы пиксела
  fmtFullscreenArray : Array [0..4] of DWORD =
       (D3DFMT_R5G6B5,
        D3DFMT_X1R5G5B5,
        D3DFMT_A1R5G5B5,
        D3DFMT_X8R8G8B8,
        D3DFMT_A8R8G8B8);
var
  FD3DfmtFullscreen : DWORD;                              // Формат пиксела
  ScreenWidth, ScreenHeight : Integer;           // Размеры рабочего стола
  HalfScreenWidth, HalfScreenHeight : Integer; // Вспомогательные размеры
  d3dpp : TD3DPRESENT_PARAMETERS;        // Структура, хранящая параметры
function TfrmD3D.InitD3D : HRESULT;
var
  iFmt : Integer;
begin
  if FD3D = nil then FD3D := Direct3DCreate8(D3D_SDK_VERSION);
  if FD3D = nil then begin
    Result := E_FAIL;
    Exit;
  end;
  // Подбираем формат пиксела для текущих установок
  for iFmt := 0 to High(fmtFullscreenArray) do begin
   if SUCCEEDED(FD3D.CheckDeviceType(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL,
     fmtFullscreenArray[iFmt], fmtFullscreenArray[iFmt], FALSE))
    then begin
      FD3DfmtFullscreen := fmtFullscreenArray[iFmt];
      Break;                 // Найден подходящий
    end
  end;
  // Запоминаем размеры рабочего стола
  ScreenWidth := GetSystemMetrics(SM_CXSCREEN);
  ScreenHeight := GetSystemMetrics(SM_CYSCREEN);
  // Координаты центра экрана
  HalfScreenWidth := ScreenWidth div 2;
  HalfScreenHeight := ScreenHeight div 2;
  // Заполняем поля структуры
  ZeroMemory(@d3dpp, SizeOf(d3dpp));
  with d3dpp do begin
    Windowed := False;       // Полноэкранный режим
    SwapEffect := D3DSWAPEFFECT_DISCARD;
```

```
    BackBufferWidth  := ScreenWidth;
    BackBufferHeight := ScreenHeight;
    BackBufferFormat := FD3DfmtFullscreen;
  end;
  Result := FD3D.CreateDevice(D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, Handle,
                              D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                              d3dpp, FD3DDevice);
end;
```

Обратите внимание, что объекты Direct3D обнуляются не в этой функции инициализации, а при создании формы. Перед созданием главного объекта определяем, не хранит ли эта переменная какое-нибудь значение. Делается это постольку, поскольку функция инициализации будет вызываться, возможно, неоднократно, в ситуации, когда главный объект уже существует.

Остальные действия в программе похожи на манипуляции, которые мы проделывали в полноэкранных приложениях, использующих DirectDraw: отслеживаем ситуацию потери активности, когда пользователь переключается, воспроизведение осуществляется только при активном состоянии.

Совершенно новым для нас является в этом примере то, что при восстановлении минимизированного приложения заново выполняется инициализация объекта устройства:

```
procedure TfrmD3D.ApplicationEvents1Restore(Sender: TObject);
begin
  if Assigned (FD3DVB) then begin    // Освобождение объектов
    FD3DVB._Release;
    FD3DVB := nil;
  end;
  WindowState := wsMaximized;        // Распахивание окна
  InitD3D;                           // Повторяем код инициализации
  InitPoints;                        // Инициализация буфера вершин
  FActive := True;
end;
```

В ситуации ухода окна с экрана происходит потеря устройства воспроизведения, подобная потере поверхности в DirectDraw. При восстановлении окна воспроизведения самым безболезненным способом возврата к воспроизведению является повторная инициализация объекта устройства. Чтобы провести эту процедуру, объект устройства необходимо освободить ото всех связанных с ним дочерних объектов, в нашем примере это единственный объект — буфер вершин. Дальше мы повторно вызываем функцию инициализации графической системы. Главный объект заново создавать не нужно, поэтому в код внесены изменения, на которые я выше обращал ваше внимание. Поскольку буфер вершин нами был удален, после инициализации системы вызывается функция, заново создающая этот объект.

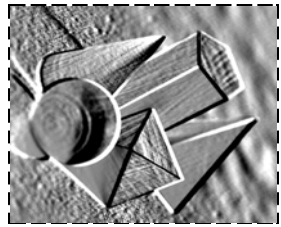
Ситуацию восстановления минимизированного приложения мы, таким образом, обслужили, а для снятия проблем, связанных со спящим режимом, можете воспользоваться рекомендациями, приведенными мною в предыдущем разделе.

Обратите внимание, что в примере не меняются настройки рабочего стола. При построении звездочки опираемся на считанные при инициализации размеры экрана.

Что вы узнали в этой главе

В данной главе мы перешли к принципиально новым для нас построениям, основанным на использовании примитивов — простых фигур, являющихся базовыми для получения объектов любой формы.

Знакомство с подсистемой Direct3D свелось для нас к изучению новых интерфейсов и методов. Главный объект дает возможность создавать дочерний объект устройства, методы которого и позволяют осуществлять воспроизведение примитивов. Буфер вершин заполняется информацией о вершинах, вершинный шейдер управляет процессами загрузки и манипуляциями с вершинами.



ГЛАВА 8

Подробнее о библиотеке Direct3D

Материал данной главы более подробно освещает вопросы, связанные с программированием графики с применением библиотеки Direct3D, и предлагает описание основных приемов ее использования.

Примеры к главе располагаются в каталоге \Examples\Chapter08.

Частичная прозрачность

В этом небольшом разделе нам предстоит подойти к изучению одного из важнейших механизмов Direct3D.

Взгляните на работу примера из каталога Ex01: на фоне звездного неба в разных направлениях вращаются полупрозрачные квадрат и треугольник (рис. 8.1).

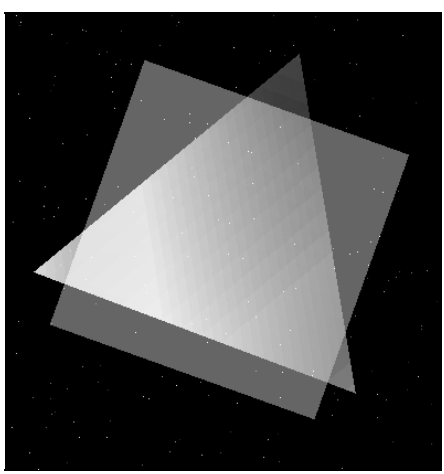


Рис. 8.1. Пример на тему частичной прозрачности примитивов

Нововведение в примере, помимо эффекта полупрозрачности, заключается в присутствии разнородных объектов, примитивов различных типов. Вершины объектов хранятся в трех различных массивах:

```
var
  VPoints : Array [0..500] of TCUSTOMVERTEX;   // 501 точка
  VTriangle : Array [0..2] of TCUSTOMVERTEX;   // Отдельный треугольник
  VQuad : Array [0..3] of TCUSTOMVERTEX;       // Вершины квадрата
```

При инициализации точек цвет вершин задается оттенком серого для имитации различной интенсивности свечения звезд. Буфер вершин для всех примитивов используется один. Чтобы задать его размер, опираемся на самый большой из трех массивов:

```
function TfrmD3D.InitPoints : HRESULT;
var
  hRet : HRESULT;
  i : Integer;
  wrkDWORD : DWORD;
begin
  Randomize;
  // Инициализация массива точек
  for i := 0 to High (VPoints) do begin
    wrkDWORD := random (200) + 40;   // Интенсивность трех весов
    with VPoints [i] do begin
      X := random (ScreenWidth);     // Координаты — случайно по всему
      Y := random (ScreenHeight);    // экрану
      Z := 0.0;
      RHW := 1.0;
      Color := D3DCOLOR_XRGB(wrkDWORD, wrkDWORD, wrkDWORD);
    end;
  end;
  // Инициализация массива вершин треугольника
  for i := 0 to High (VTriangle) do
    with VTriangle [i] do begin
      Z := 0.0;
      RHW := 1.0;
    end;
  // Цвета вершин треугольника
  VTriangle [0].Color := D3DCOLOR_XRGB(255, 0, 0);
  VTriangle [1].Color := D3DCOLOR_XRGB(0, 255, 0);
  VTriangle [2].Color := D3DCOLOR_XRGB(0, 0, 255);
  // Инициализация массива вершин серого квадрата
  for i := 0 to High (VQuad) do
    with VQuad [i] do begin
      Z := 0.0;
```

```
      RHW := 1.0;
      Color := D3DCOLOR_XRGB(100, 100, 100);
  end;
  // Создаем буфер вершин, размер опирается на размер массива точек
  hRet := FD3DDevice.CreateVertexBuffer(SizeOf(VPoints),
                      D3DUSAGE_WRITEONLY, D3DFVF_CUSTOMVERTEX,
                      D3DPOOL_DEFAULT, FD3DVB);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Связываем поток
  hRet := FD3DDevice.SetStreamSource(0, FD3DVB, SizeOf(TCUSTOMVERTEX));
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Задаем шейдер вершин
  Result := FD3DDevice.SetVertexShader(D3DFVF_CUSTOMVERTEX);
end;
```

Следующие три пользовательские функции вызываются собственно для рисования примитивов:

```
function DrawPoints : HRESULT;
function Draw2DTriangle (const inX, inY : Single) : HRESULT;
function TfrmD3D.Draw2DQuad (const inX, inY : Single) : HRESULT;
```

Поскольку точки в этом примере неподвижны, в первой из указанных функций производится только копирование содержимого массива `VPoints` в буфер вершин. Заканчивается она вызовом метода вывода точек:

```
Result:=FD3DDevice.DrawPrimitive(D3DPT_POINTLIST, 0, High (VPoints) + 1);
```

В оставшихся двух функциях перед заполнением буфера вычисляются координаты вершин фигур. Вот код функции, отображающей квадрат:

```
const
  HalfPi = Pi / 2;           // Небольшая оптимизация
function TfrmD3D.Draw2DQuad (const inX, inY : Single) : HRESULT;
var
  pVertices : PByte;
  hRet : HRESULT;
begin
  with VQuad [0] do begin    // Левый нижний угол квадрата
    X := inX + Radius * cos (Angle / 2 - HalfPi);
    Y := inY + Radius * sin (Angle / 2 - HalfPi);
  end;
```

```
  with VQuad [1] do begin    // Левый верхний угол квадрата
    X := inX + Radius * cos (Angle / 2);
    Y := inY + Radius * sin (Angle / 2);
  end;
  with VQuad [2] do begin    // Правый нижний угол квадрата
    X := inX + Radius * cos (Angle / 2 + Pi);
    Y := inY + Radius * sin (Angle / 2 + Pi);
  end;
  with VQuad [3] do begin    // Правый верхний угол квадрата
    X := inX + Radius * cos (Angle / 2 + HalfPi);
    Y := inY + Radius * sin (Angle / 2 + HalfPi);
  end;
  // Заполняем вершинный буфер данными о вершинах квадрата
  hRet := FD3DVB.Lock(0, SizeOf(VQuad), pVertices, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  Move (VQuad, pVertices^, SizeOf(VQuad));
  hRet := FD3DVB.Unlock;
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Квадрат образован двумя связанными треугольниками
  Result := FD3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
end;
```

При перерисовке кадра перед отображением квадрата включаем режим полупрозрачности, а после рисования треугольника его отключаем:

```
function TfrmD3D.Render : HRESULT;
var
  hRet : HRESULT;
begin
  if FD3DDevice = nil then begin
    Result := E_FAIL;
    Exit;
  end;
  // Очистка экрана и окрашивание его в черный цвет
  hRet := FD3DDevice.Clear(0, nil, D3DCLEAR_TARGET, 0, 0.0, 0);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
```

```
  hRet := FD3DDevice.BeginScene;
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Первыми рисуются точки фона
  hRet := DrawPoints;
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Включаем режим полупрозрачности
  with FD3DDevice do begin
    SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD(True));
    SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
    SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
  end;
  // Полупрозрачный квадрат
  hRet := Draw2DQuad (HalfScreenWidth, HalfScreenHeight);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Полупрозрачный треугольник
  hRet := Draw2DTriangle (HalfScreenWidth, HalfScreenHeight);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Режим полупрозрачности больше не нужен
  FD3DDevice.SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD(False));
  hRet := FD3DDevice.EndScene;
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  Result := FD3DDevice.Present(nil, nil, 0, nil);
end;
```

Чтобы примитив оказался полупрозрачным, необходимо, во-первых, включить режим `D3DRS_ALPHABLENDENABLE`. Вторым аргументом для этого режима задается преобразованная в `DWORD` (или `Cardinal`) булева константа. Для простоты можно использовать просто нуль или единицу. И во-вторых, следует задать параметры источника и приемника. Для простейшего случая, когда все примитивы одинаково полупрозрачны, этим параметром назначается константа `D3DBLEND_ONE`.

Совет

В случае одинаковой полупрозрачности примитивов достаточно задавать параметр приемника, значение для `D3DRS_SRCBLEND` по умолчанию установлено в `D3DBLEND_ONE`.

При включенном режиме `D3DRS_ALPHABLENDENABLE` примитивы как бы рисуются на кальке: в местах наложения листов кальки плотность цветовых компонентов увеличивается.

Для оптимизации сразу же после того, как нарисованы все полупрозрачные примитивы, режим полупрозрачности необходимо отключить.

Альфа-составляющая цвета

Рассмотренный в предыдущем разделе пример полупрозрачности является частным случаем работы с четвертой составляющей цвета, так называемым *альфа-компонентом* или *альфа-составляющей цвета*. Этот компонент позволяет регулировать степень прозрачности. По умолчанию установлено значение 255, т. е. примитивы совершенно не прозрачны. Нулевое значение имитирует полную прозрачность. Все промежуточные значения соответствуют градации прозрачности.

Рассмотрим простой пример на эту тему (проект каталога Ex02), где на фоне непрозрачного красного квадрата вращается сложная геометрическая композиция зеленого цвета, которую для простоты назовем звездой (рис. 8.2).

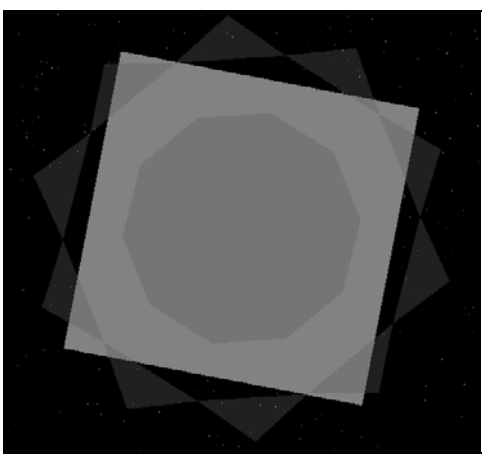


Рис. 8.2. В этом примере степень прозрачности примитива меняется со временем

Звезда сначала абсолютно прозрачна, но со временем становится плотнее. Цвета квадрата и звезды накладываются точно, образуя оттенки желтого, пока звезда не станет совершенно непрозрачной.

Вспомогательная переменная Alpha последовательно принимает значения от 0 до 255 и используется в функции рисования звезды, при задании цветовой составляющей вершин:

```
for i := 0 to High (VStar) do
  VStar [i].Color := D3DCOLOR_ARGB(Alpha, 0, 255, 0);
```

Обратите внимание, что теперь для задания цвета вершины используется функция D3DCOLOR_ARGB, аргументом которой является четверка чисел. Первый аргумент — значение альфа-компонента.

При перерисовке кадра режим альфа-смешения устанавливается только на время работы функции рисования звезды из соображений оптимальности:

```
with FD3DDevice do begin
  SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD(True));
  SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
  SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
end;
hRet := DrawStar;
if FAILED(hRet) then begin
  Result := hRet;
  Exit;
end;
FD3DDevice.SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD (False));
```

Для режима регулируемой прозрачности значение степени прозрачности источника определяется текущим значением альфа-составляющей цвета вершин. Режиму соответствует константа D3DBLEND_SRCALPHA. В данном режиме каждый чистый цвет источника имеет коэффициент прозрачности, равный значению альфа-составляющей. Для режима смешения цветов режим приемника задается константой D3DBLEND_INVSRCALPHA так, что суммарно получается единица — итоговый пиксел совершенно непрозрачный:

- D3DBLEND_SRCALPHA: коэффициент смешения (A_S, A_S, A_S, A_S);
- D3DBLEND_INVSRCALPHA: коэффициент смешения ($1-A_S$, $1-A_S$, $1-A_S$, $1-A_S$).

Не путайте с весом цвета, речь идет о прозрачности каждого цветового компонента.

Совсем необязательно, чтобы все вершины, образующие примитив, имели одинаковое значение альфа-компонента. Варьируя альфа-составляющую, можно добиваться эффекта градиентной прозрачности примитива. В качестве примера использования такого эффекта я приготовил проект каталога Ex03 — полноэкранное приложение, в котором при каждом щелчке кнопки мыши на экране расплываются красивые полупрозрачные пятна.

Для манипуляций с пятнами использую концепцию ООП:

```
type
  TRGB = packed record      // Запись тройки цветов
    R, G, B : Byte;
  end;
  TDrip = class             // Класс отдельного пятна
     PosX, PosY : Integer;  // Координаты на экране
     Ring_Color : TRGB;     // Цвет пятна
     Ring_Radius : Integer; // Текущий радиус пятна
  end;
const
  Level = 36;               // Уровень разбиения круга
  Max_Drips = 120;          // Максимум присутствия пятен в окне
  Max_Ring_Radius = 100.0;  // Максимальный радиус пятна
var
  VCircle : Array [0..Level + 1] of TCUSTOMVERTEX;  // Вершины круга
  First_Drip, New_Drip : Integer;                   // Счетчики пятен
  Drips : Array [0..Max_Drips - 1] of TDrip;        // Массив пятен
```

При каждом нажатии кнопки мыши создается новое пятно. Координатами его центра выступают текущие координаты курсора:

```
procedure Create_Drip(const inX, inY : Integer; const R, G, B : Byte);
begin
  // Создание нового пятна
  Drips[New_Drip] := TDrip.Create;
  with Drips[New_Drip] do begin
    Ring_Color.R := R;
    Ring_Color.G := G;
    Ring_Color.B := B;
    Ring_Radius := 0;
    PosX := inX;
    PosY := inY;
  end;
  // Увеличение счетчиков
  New_Drip := (New_Drip + 1) mod Max_Drips;
  // Достигнут предел по количеству пятен на экране
  if New_Drip = First_Drip
    then First_Drip := (First_Drip + 1) mod Max_Drips;
end;
procedure TfrmD3D.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Create_Drip(X,Y,random(150)+105,random(150)+105,random(150)+105);
end;
```

Вес чистых цветов выбирается из некоторого диапазона. При градиентной прозрачности примитивов для больших треугольников или при чересчур низком весе цветов образуются полоски, портящие картинку (возможно, при использовании не каждой видеокарты). Ограничение для размеров кругов установлено также с целью предотвращения появления таких узоров.

Для рисования каждого пятна вызывается функция `DrawCircle`, круг рисуется группой связанных треугольников:

```
function TfrmD3D.DrawCircle (const inX, inY, inRadius : Integer;
                             const Ring_Color : TRGB) : HRESULT;
const
  Step = 2 * Pi / Level;
var
  pVertices : PByte;
  hRet : HRESULT;
  i : Integer;
begin
  // Первая точка — центр круга
  with VCircle [0] do begin
    X := inX;
    Y := inY;
  // Точка центра совершенно непрозрачна
  Color := D3DCOLOR_ARGB(255, Ring_Color.R, Ring_Color.G, Ring_Color.B);
  end;
  // Точки края круга абсолютно прозрачны
  for i := 1 to Level + 1 do
    with VCircle [i] do begin
     X := VCircle [0].X + cos (i * Step) * inRadius;
     Y := VCircle [0].Y + sin (i * Step) * inRadius;
     Color := D3DCOLOR_ARGB(0, Ring_Color.R, Ring_Color.G, Ring_Color.B);
  end;
  hRet := FD3DVB.Lock(0, SizeOf(VCircle), pVertices, 0);
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  Move (VCircle, pVertices^, SizeOf(VCircle));
  hRet := FD3DVB.Unlock;
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  Result := FD3DDevice.DrawPrimitive(D3DPT_TRIANGLEFAN, 0, Level);
end;
```

При перерисовке кадра отображается круг для каждого существующего пятна:

```
i := First_Drip;
// Цикл по всем присутствующим на экране пятнам
while i <> New_Drip do begin
  // Каждое пятно увеличивается в размерах
  Drips[i].Ring_Radius := Drips[i].Ring_Radius + 1;
  DrawCircle (Drips[i].PosX,
              Drips[i].PosY,
              Drips[i].Ring_Radius,
              Drips[i].Ring_Color);
  // Пятно достигло максимального размера, поэтому исчезает с экрана
  if Drips[i].Ring_Radius > Max_Ring_Radius
     then First_Drip := (First_Drip + 1) mod Max_Drips;
  i := (i+1) mod Max_Drips;
end;
```

Посмотрите работу данного примера для случаев, когда альфа-компоненты для центра и краев круга одинаковы и равны небольшой константе, или когда для точки центра таким значением берется 255, а для точек края — значение радиуса круга.

Размытие при движении

Манипулированием альфа-смешением можно легко получить эффект размытия при движении, когда быстро движущиеся объекты оставляют за собой след. Обычно такой эффект называется *motion blur*. Для получения эффекта в местах предыдущего положения объекта рисуют его полупрозрачную копию. Более простой способ создания эффекта — манипулирование цветовой интенсивностью.

Оба способа используются в проекте каталога Ex04, где на экране быстро вращаются два круга: желтый оставляет полупрозрачный след, а за красным тянется постепенно угасающий след (рис. 8.3).

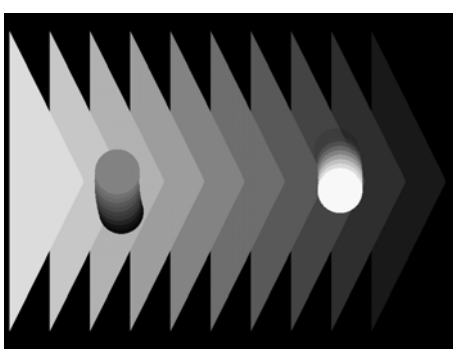


Рис. 8.3. Эффект размытия при движении

Рисование каждого круга сводится к выводу ряда близко расположенных примитивов:

```
wrkGreen := 5;
for i := 1 to 10 do begin            // 10 зеленых треугольников фона
  wrkGreen := wrkGreen + 25;
  hRet := DrawTriangle (ScreenWidth – (i + 1) * (ScreenWidth div 11),
                        ScreenWidth div 6, wrkGreen);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
end;
wrkAngle := Angle;
wrkAlpha := 5;       // Круги рисуются, начиная с самого прозрачного
with FD3DDevice do begin
  SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD(True));
  SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
  SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
end;
for i := 1 to 10 do begin         // 10 желтых кругов различной
  wrkAngle := wrkAngle + 0.04;    // прозрачности
  wrkAlpha := wrkAlpha + 25;
  hRet := DrawYellowCircle (wrkAngle, wrkAlpha);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
end;
FD3DDevice.SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD(False));
wrkAngle := Angle + Pi;
wrkRed := 5;                      // Степень насыщенности красного
for i := 1 to 10 do begin         // 10 красных кругов
  wrkAngle := wrkAngle + 0.04;
  wrkRed := wrkRed + 25;
  hRet := DrawRedCircle (wrkAngle, wrkRed);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
end;
```

Конечно, способ, базирующийся на прозрачности, дает более эффектный результат, но и требует больше времени для воспроизведения. Ко второму же способу можно безболезненно прибегнуть, если объект будет двигаться только на черном фоне, иначе проступают явные следы движения объекта.

Работа с переменным числом вершин

Мы уже хорошо освоились в построениях фигур с помощью Direct3D и в этом небольшом разделе попробуем развить наши навыки и узнать некоторые новые для нас вещи.

Как выяснилось из многочисленных предыдущих примеров, при использовании FVF-флага `D3DFVF_XYZRHW` в своих построениях мы опираемся на систему координат, ассоциированную с окном. Теперь нам предстоит постичь смысл еще одного флага: `D3DFVF_XYZ`. При его применении система координат экрана воспроизведения выглядит так: центру окна, независимо от его размеров, соответствует точка с координатами (0, 0), правому верхнему углу окна — (1, 1), левому нижнему углу — (−1, −1).

Пока мы ограничимся такой трактовкой этого флага, а позже узнаем о его особенностях кое-что дополнительно. Сейчас же для нас важно, что при использовании этого FVF-флага мы не сможем окрашивать вершины так, как привыкли это делать, и временно можем использовать только черно-белые картинки.

На примере проекта каталога Ex05 попробуем закрепить знания об этом флаге и попутно решим еще одну задачу: научимся работать с переменным числом вершин.

Самое простое решение задачи состоит, конечно, в том, чтобы задавать размер буфера вершин максимальным. Но такой прием приводит к неэффективному расходу памяти.

Во время работы программы на экране появляются узоры, образуемые отрезками, соединяющими равномерно расположенные точки на окружности. Число узлов меняется с течением времени случайно (рис. 8.4).

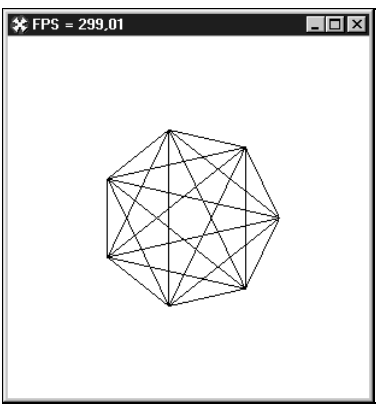


Рис. 8.4. Пример с произвольным числом вершин

Текущее значение переменной `numPoints` хранит число узлов. В периодически вызываемой функции `InitVB` не используется массив вершин, а при-

меняется единственная переменная — указатель на структуру `TCustomVertex`. В этой структуре, в отличие от предыдущих примеров, отсутствует поле `Color`, а описание формата данных вершины сократилось до одной константы:

```
const
  D3DFVF_CUSTOMVERTEX = D3DFVF_XYZ;
```

Поскольку размер буфера постоянно изменяется, при каждом обращении к этой функции повторяются все действия инициализации буфера вершин:

```
function TfrmD3D.InitVB : HRESULT;
const
  Pi2 = 2 * Pi;                // Для сокращения числа операций
var
  Vertices : ^TCustomVertex;   // Указатель на запись вершины
  i, j, k : Byte;
  hRet : HRESULT;
begin
  numPoints := random (7) + 3;  // Генерация количества узлов
  k := 0;                // Подсчет количества отрезков, образующих узор
  for i := 1 to numPoints do
    for j := i + 1 to numPoints do begin
    Inc(k);
  end;
  numLines := k;         // Используется в DrawPrimitive
  // Создание буфера вершин нужного размера
  hRet := FD3DDevice.CreateVertexBuffer(2 * k * SizeOf(TCustomVertex), 0,
                          D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT, FD3DVB);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Заполнение буфера
hRet := FD3DVB.Lock(0,2 * k * SizeOf(TCustomVertex), PByte(Vertices), 0);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Перебор точек узлов
  for i := 1 to numPoints do
    for j := i + 1 to numPoints do begin
      // Начало отрезка, точка на окружности радиусом 0.5
      Vertices.X := 0.5 * cos(Pi2 * i / numPoints);
      Vertices.Y := 0.5 * sin(Pi2 * i / numPoints);
      Vertices.Z := 0;
      Inc(Vertices);               // Сдвигаем указатель
```

```
      // Конец отрезка
      Vertices.X := 0.5 * cos(Pi2 * j / numPoints);
      Vertices.Y := 0.5 * sin(Pi2 * j / numPoints);
      Vertices.Z := 0;
      Inc(Vertices);
  end;
  hRet := FD3DVB.Unlock;
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Заново устанавливаем поток
  hRet := FD3DDevice.SetStreamSource(0, FD3DVB, SizeOf(TCUSTOMVERTEX));
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Задаем вершинный шейдер
  Result := FD3DDevice.SetVertexShader(D3DFVF_CUSTOMVERTEX);
end;
```

Последнее действие можно вынести из кода функции, чтобы выполнить его один раз, в начале работы приложения.

Беспрерывным созданием буфера вершин лучше не злоупотреблять и использовать только в случае крайней необходимости, иначе работа приложения может оказаться неустойчивой.

Теперь мы можем выяснить одну, очень важную для нас особенность использования флага `D3DFVF_XYZ`. Приведите код функции `InitVB` к следующему виду:

```
function TfrmD3D.InitVB : HRESULT;
var
  Vertices : ^TCustomVertex;
  hRet : HRESULT;
begin
  hRet := FD3DDevice.CreateVertexBuffer(3 * SizeOf(TCustomVertex), 0,
                  D3DFVF_CUSTOMVERTEX,D3DPOOL_DEFAULT, FD3DVB);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := FD3DVB.Lock(0, 3 * SizeOf(TCustomVertex), PByte(Vertices), 0);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
```

```
  Vertices.X := 0.0;
  Vertices.Y := 0.0;
  Vertices.Z := 0;
  Inc(Vertices);
  Vertices.X := 0.0;
  Vertices.Y := 0.5;
  Vertices.Z := 0;
  Inc(Vertices);
  Vertices.X := 0.5;
  Vertices.Y := 0.5;
  Vertices.Z := 0;
  hRet := FD3DVB.Unlock;
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := FD3DDevice.SetStreamSource(0, FD3DVB, SizeOf(TCUSTOMVERTEX));
  if Failed (hRet) then begin
    Result := hRet;
    Exit;
  end;
  Result := FD3DDevice.SetVertexShader(D3DFVF_CUSTOMVERTEX);
end;
```

Таким образом, буфер вершин всегда заполняется данными о трех вершинах. Построим один треугольник. Для этого подправьте аргументы метода воспроизведения примитивов:

```
hRet := FD3DDevice.DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
```

Запустите программу и посмотрите результат: выводится один треугольник. Ничего особенного, но теперь поменяйте координаты первой и второй вершины треугольника и снова запустите программу. Экран станет чистым, ничего теперь воспроизводиться не будет. Ошибок нет. Просто для этого режима очень важен порядок перечисления вершин треугольников. Он задает сторону примитива, которую мы наблюдаем: лицевую или изнаночную. Для лицевой стороны вершины перечисляются по часовой стрелке. Поскольку в первом случае вершины треугольника задавались именно в таком порядке, зрителю видна передняя сторона треугольника. Когда мы переставили вершины, треугольник повернулся к нам своей тыльной стороной, а задние стороны треугольников по умолчанию не воспроизводятся. Поэтому мы не получили никакого результата.

Чтобы отключить режим отсечения задних сторон треугольников, можете вставить в код функции `Render` следующую строку:

```
FD3DDevice.SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
```

То есть мы выключаем таким образом режим отсечения. Если вторым параметром использовать константу `D3DCULL_CW`, будут отсекаться примитивы, вершины которых перечисляются в поле зрения по часовой стрелке, а при значении, равным `D3DCULL_CCW` — против часовой стрелки. Именно это значение и установлено по умолчанию. В плоскостных построениях мы не станем менять установки этого режима, а будем следить за порядком перечисления вершин треугольников.

Текстура

Теперь нам предстоит изучить одну из важнейших тем — использование растровых образов. В Direct3D имеется несколько типов (стилей) текстур. Мы изучим текстуру, подобную наклеиваемым обоям.

Как всегда, для изучения нового понятия нам потребуется познакомиться с новыми типами объектов и интерфейсов. И как обычно для этой книги, знакомство осуществим на конкретном примере. Сейчас им послужит проект каталога Ex06. Работа примера очень проста: на экране выводится содержимое растрового файла — картинка с изображением дискеты (рис. 8.5).

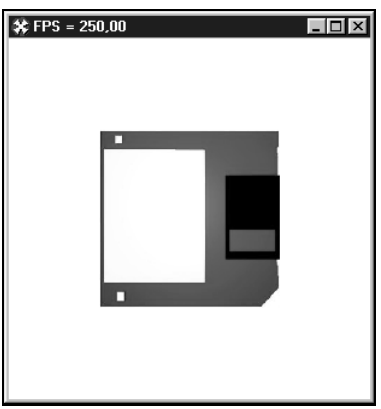


Рис. 8.5. Простейший пример использования текстуры

Кратко смысл программы можно описать так: на два связанных треугольника, образующих квадрат, накладывается квадратная текстура.

В списке переменных добавилась еще одна, связанная с используемым COM-объектом:

```
FD3Texture      : IDIRECT3DTEXTURE8;
```

В начале работы ее значением устанавливается `nil`, а при завершении работы перед окончательным освобождением памяти вызывается метод `_Release` этого объекта.

Формат данных вершины, помимо пространственных координат, содержит еще две, связанные с наложением текстуры:

```
type
  TCUSTOMVERTEX = packed record
    X, Y, Z : Single;
    U, V : Single;     // Новая пара координат в формате вершины
  end;
const
  D3DFVF_CUSTOMVERTEX = D3DFVF_XYZ or D3DFVF_TEX1;   // Новая константа
```

Итак, для наложения текстуры на объект для каждой вершины должны указываться текстурные координаты. Сейчас мы используем двумерную текстуру. Она представляет собой прямоугольный массив данных. Для такой текстуры в формате вершин необходимо задавать две координаты, обычно называемые U и V. Первая из этих координат ассоциирована с горизонтальной осью текстуры, вторая, V-координата — с вертикальной. То есть для вершины, связываемой с левым нижним углом текстуры, оба эти значения должны быть нулевыми, а для вершины, к которой приклеивается правый верхний угол текстуры, оба эти значения должны быть единичными. Обращаю внимание, что эти координаты никак не связаны с пространственными координатами вершин и примитива, сам примитив не обязан иметь единичные размеры.

Константа `D3DFVF_TEX1` является указанием на то, что координаты текстуры задаются именно парой чисел, максимальным может быть девять координат.

При инициализации буфера вершин используем тот же прием, что и в предыдущем примере, т. е. обходимся без вспомогательного массива. Но инициализация буфера производится один раз, в самом начале работы приложения.

Квадрат располагаем в центре экрана:

```
function TfrmD3D.InitVB : HRESULT;
var
  Vertices : ^TCustomVertex;
  hRet : HRESULT;
begin
  // Буфер вершин на четыре вершины квадрата
  hRet := FD3DDevice.CreateVertexBuffer(4 * SizeOf(TCustomVertex), 0,
                                        D3DFVF_CUSTOMVERTEX,
                                        D3DPOOL_DEFAULT, FD3DVB);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
```

```
  // Устанавливаем поток
  hRet := FD3DDevice.SetStreamSource(0, FD3DVB, SizeOf(TCustomVertex));
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Задаем шейдер вершин
  hRet := FD3DDevice.SetVertexShader(D3DFVF_CUSTOMVERTEX);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Заполняем буфер данными
  hRet := FD3DVB.Lock(0, 4 * SizeOf(TCustomVertex), PByte(Vertices), 0);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Левый нижний угол квадрата
  Vertices.X := -0.5;     // Координата на листе
  Vertices.Y := -0.5;
  Vertices.Z := 0;        // Левый нижний угол текстуры
  Vertices.U := 0;
  Vertices.V := 0;
  Inc(Vertices);          // Переходим к следующей вершине
  Vertices.X := -0.5;     // Левый верхний угол квадрата
  Vertices.Y := 0.5;
  Vertices.Z := 0;
  Vertices.U := 0;
  Vertices.V := 1;
  Inc(Vertices);
  Vertices.X := 0.5;      // Правый нижний угол квадрата
  Vertices.Y := -0.5;
  Vertices.Z := 0;
  Vertices.U := 1;
  Vertices.V := 0;
  Inc(Vertices);
  Vertices.X := 0.5;      // Правый верхний угол квадрата
  Vertices.Y := 0.5;
  Vertices.Z := 0;
  Vertices.U := 1;
  Vertices.V := 1;
  Result := FD3DVB.Unlock;
end;
```

Текстура создается с помощью отдельной функции, единственным аргументом которой является имя файла-прототипа:

```
function TfrmD3D.InitTexture (const FileName : String) : HRESULT;
var
  hRet : HRESULT;
  d3dlr : TD3DLOCKED_RECT;   // Вспомогательная запись
  dwDstPitch : DWORD;        // Шаг поверхности текстуры
  X, Y : DWORD;
  Bmp : tBitmap;
  R, G, B : Byte;
begin
  Bmp := TBitmap.Create;
  Bmp.LoadFromfile (FileName);
  // Создание объекта текстуры
  hRet := FD3DDevice.CreateTexture (Bmp.Width, Bmp.Height, 0, 0,
                                    D3DFMT_A8R8G8B8,
                                    D3DPOOL_MANAGED, FD3Texture);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Запираем поверхность текстуры
  FD3Texture.LockRect(0, d3dlr, nil, 0);
  dwDstPitch := d3dlr.Pitch;  // Запоминаем шаг поверхности
  // Заполняем поверхность данными из растра
  for Y := 0 to Bmp.Height - 1 do
    for X := 0 to Bmp.Width - 1 do begin
      R := GetRValue (Bmp.Canvas.Pixels [X, DWORD (Bmp.Height - 1) - Y]);
      G := GetGValue (Bmp.Canvas.Pixels [X, DWORD (Bmp.Height - 1) - Y]);
      B := GetBValue (Bmp.Canvas.Pixels [X, DWORD (Bmp.Height - 1) - Y]);
      PDWORD (DWORD(d3dlr.pBits)+Y*dwDstPitch + X * 4)^:=
              D3DCOLOR_XRGB(R,G,B);
  end;
  Bmp.Free;
  // Отпираем поверхность текстуры
  Result := FD3Texture.UnlockRect(0);
end;
```

Первые два аргумента метода `CreateTexture` объекта устройства — ширина и высота создаваемой текстуры. Каждое из этих чисел должно быть степенью двойки. Это очень важное правило, не пропустите его. Растр может быть любого размера, но поверхность текстуры произвольные размеры иметь не может. Если необходимо использовать растр, размеры которого не равны степени двойки, его следует масштабировать, используя те же приемы,

которые мы рассмотрели в нескольких примерах на тему применения DirectDraw.

Следующие два параметра метода для нас не важны, а вот на пятый аргумент, формат пиксела, надо обратить внимание. Выбор формата поверхности текстуры оставлен за разработчиком, который сам должен решить, какое значение из предлагаемого набора для него наиболее всего подходит. Для текстур, представляющих собой обычные растры, самым удобным является 32-битный формат, в котором на каждый пиксел приходится четверка чисел. Константа, соответствующая такому формату — `D3DFMT_A8R8G8B8`. Конечно, можно использовать и другие форматы, например "5—6—5", но при манипуляции с пикселами поверхности необходимо учитывать сделанный выбор.

Последними аргументами метода `CreateTexture` являются константа, отражающая пожелание разработчика о месте расположения поверхности текстуры, и собственно имя создаваемого объекта.

После того как объект текстуры создан, необходимо заполнить его поверхность. Как видим из кода, порядок действий здесь похож на манипуляции, производимые с поверхностями в DirectDraw: поверхность вначале запирается, и мы получаем информацию о ее шаге и адресе, по которому она располагается. После того как поверхность заполнена, она должна быть разблокирована.

Вторым аргументом метода `LockRect`, запирающего поверхность текстуры, должна передаваться величина типа `TD3DLOCKED_RECT`, вспомогательная запись из двух полей: шаг, ширина поверхности и адрес поверхности в памяти.

Заполняем поверхность текстуры тривиальным образом, сообразно с пикселами загруженного растра. Ось Y при этом переворачиваем, присутствующее здесь преобразование типа в `DWORD` совсем не обязательно, его я осуществляю только для того, чтобы предотвратить ненужные предупреждения компилятора.

Адресация ячейки пиксела текстуры аналогична тому, что мы производили в DirectDraw: опираемся на шаг поверхности, который не обязательно равен ее ширине. Значение X умножается на 4, т. е. на размер одной ячейки. Число это обусловлено выбранным форматом пиксела.

После того как поверхность текстуры заполнена и разблокирована, необходимо задать параметры ее использования и назначить текущей. Из соображений оптимизации рекомендуется устанавливать текстуру только на время непосредственного ее использования:

```
with FD3DDevice do begin
  SetTexture(0, FD3Texture);       // Задаем текущую текстуру
  SetTextureStageState(0, D3DTSS_COLOROP, D3DTA_TEXTURE);
end;
```

```
// Квадрат, покрытый текстурой
hRet := FD3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
if FAILED(hRet) then begin
  Result := hRet;
  Exit;
end;
// Делаем текстуру недействительной
FD3DDevice.SetTexture(0, nil);
```

Чтобы задать текущую текстуру, необходимо вызвать метод `SetTexture` объекта устройства, вторым аргументом передается нужный объект текстуры или `nil` в случае, если текстура больше не используется. Представленное за этим действие следует понимать как задание правил операций с цветом при работе с текстурой, значение цвета для каждого пиксела определяется содержимым поверхности текстуры.

Методы `SetTexture` и `SetTextureStageState` должны вызываться в установленном состоянии воспроизведения, после вызова метода `BeginScene`. Также помните о том, что блоки установок могут содержать вызовы этих методов.

Итак, текстура является приклеенным к примитиву образом, который масштабируется и поворачивается вслед за ним. В проекте каталога Ex07 квадрат, покрытый текстурой, вращается, а нажатием клавиш <Insert> и <Delete> можно манипулировать его размерами (рис. 8.6).

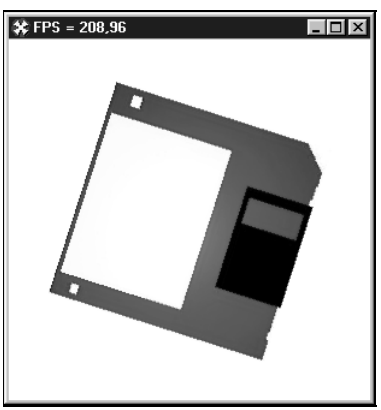


Рис. 8.6. Текстура поворачивается и растягивается вслед за положением вершин примитива

Поворачивается квадрат тривиальным образом, с течением времени меняем пространственные координаты вершин треугольников, образующих квадрат. Текстурные же координаты вершин неизменны, они не зависят от текущего положения квадрата, этими кнопками мы прикрепляем углы образа к нашему квадрату.

Текстурные координаты

Быстро выводить растровое изображение с помощью DirectDraw мы уже давно научились. Теперь же должны посмотреть все возможности, которые предоставляются нам Direct3D, и то, что сделать раньше мы могли, только затрачивая титанические усилия.

Например, если в предыдущих примерах единичные значения текстурных координат заменить на 3, образ будет повторяться 9 раз. А если и нулевые значения изменить на −3, мы получим 36 образов, уменьшенных в размерах.

Теперь посмотрите проект каталога Ex08. Текстура здесь накладывается на квадрат, образованный четырьмя десятками независимых треугольников: полный круг разделен на четыре четверти, в пределах каждой из которой строится десять независимых треугольников. Первая вершина каждого треугольника — центр итогового квадрата. Остальные вершины лежат на его границе.

Вот часть кода, посвященная верхней четверти квадрата:

```
for i := 10 downto 1 do begin
  Vertices.X := 0;                    // Центр экрана
  Vertices.Y := 0;
  Vertices.Z := 0;
  Vertices.U := 0.5;                  // Центр текстуры
  Vertices.V := 0.5;
  Inc(Vertices);
  // Вершины перечисляем по часовой стрелке,
  // движемся с левого верхнего угла квадрата
  Vertices.X := 0.5 - i / 10;
  Vertices.Y := 0.5;          // Верхний край, значение Y не меняется
  Vertices.Z := 0;
  Vertices.U := 1 - i / 10;           // X-координата текстуры
  Vertices.V := 1.0;                  // Y-координата текстуры
  Inc(Vertices);
  Vertices.X := 0.5 - (i - 1) / 10;   // По часовой стрелке,
  Vertices.Y := 0.5;                  // точка слева
  Vertices.Z := 0;
  Vertices.U := 1 - (i - 1) / 10;
  Vertices.V := 1.0;
  Inc(Vertices);
end;
```

Таким образом, на каждом из сорока треугольников хранится кусочек целого образа, и сложенные рядом, они складывают картинку исходного растра. Включите проволочный режим воспроизведения, чтобы уяснить, как разбивается растр. Кстати, это нам позволит убедиться также в том, что текстуру можно накладывать и на отрезки.

Зачем так сложно сделано, вам станет ясно после знакомства со следующим примером, проектом каталога Ex09, где по нажатии клавиши <Пробел> треугольники разлетаются в разные стороны (рис. 8.7).

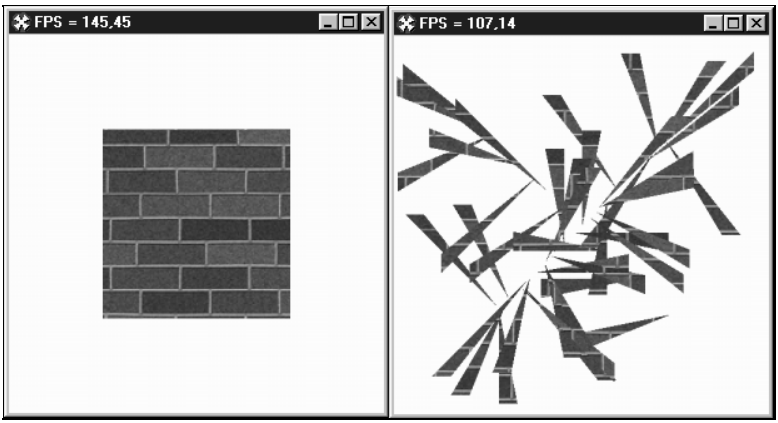


Рис. 8.7. Этапы разрушения стены

Координаты первой вершины всех треугольников, точки разлома картинки, выбираются случайным образом, а на каждый треугольник генерируется свое направление движения:

```
for i := 10 downto 1 do begin
  Vertices.X := CenterX + Radius * W1 [i];   // Точка разлома картинки
  Vertices.Y := CenterY + Radius * W1 [i];
  Vertices.Z := 0;
  Vertices.U := CenterX + 0.5;   // CenterX находится в точке [-0.5; 0.5]
  Vertices.V := CenterY + 0.5;
  Inc(Vertices);
  // Точки, расположенные на границе квадрата
  Vertices.X := 0.5 - i / 10 + Radius * W1 [i];
  Vertices.Y := 0.5 + Radius * W1 [i];
  Vertices.Z := 0;
  Vertices.U := 1 - i / 10;
  Vertices.V := 1.0;
  Inc(Vertices);
  Vertices.X := 0.5 - (i - 1) / 10 + Radius * W1 [i];
  Vertices.Y := 0.5 + Radius * W1 [i];
  Vertices.Z := 0;
  Vertices.U := 1 - (i - 1) / 10;
  Vertices.V := 1.0;
  Inc(Vertices);
end;
```

В программе предусмотрен режим пошагового разрушения, а по нажатии клавиши <Enter> картинка собирается заново:

```
procedure TfrmD3D.FormKeyDown(Sender: TObject; var Key: Word;
                              Shift: TShiftState);
var
  i : Integer;
begin
  if Key = VK_ESCAPE then Close else
  // Пошаговое разрушение
  if Key = VK_INSERT then Radius := Radius + 0.05 else
  // Пошаговое движение в обратном направлении
  if Key = VK_DELETE then Radius := Radius - 0.05 else
  // Пробел - быстрое разрушение
  if Key = VK_SPACE then Moving := True else
  // Ввод - картинка собирается заново
  if Key = VK_RETURN then begin
     Moving := False;              // Прекратить движение
     Radius := 0;                  // Картинка собирается
     CenterX := random - 0.5;      // Координаты точки разлома
     CenterY := random - 0.5;
     for i := 1 to 10 do begin     // Коэффициенты скорости движения
         repeat                    // треугольников, все ненулевые
         W1 [i] := random - 0.5;
         until W1 [i] <> 0.0;
         repeat
         W2 [i] := random - 0.5;
         until W2 [i] <> 0.0;
         repeat
         W3 [i] := random - 0.5;
         until W3 [i] <> 0.0;
         repeat
         W4 [i] := random - 0.5;
         until W4 [i] <> 0.0;
     end;
  end;
end;
```

Немного повозившись, вы можете добиться разрушения стены по отдельным кирпичикам или другим способам разлома.

У вершин треугольников текстурные координаты могут совпадать. С помощью такого трюка можно добиться интересных эффектов, например, как в проекте каталога Ex10, где исходный растр выводится мозаично (рис. 8.8).

В массиве некоторого предопределенного размера хранятся точки, разбросанные в пределах области вывода.

```
type
  TXY = packed record    // Координаты точки на плоскости
    X, Y : Single;
  end;
const
  SIDES = 20;                          // Уровень детализации круга
  SIZE = 5500;                         // Количество точек
var
  Points : Array [0..SIZE-1] of TXY;   // Массив точек
  Radius : Single = 0.03;              // Размер отдельной точки
```

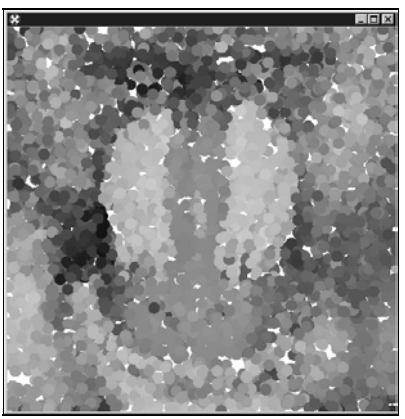


Рис. 8.8. Эту технику живописи отличают крупные мазки

Массив заполняется в начале работы значениями из интервала [−1.0; 1.0]:

```
procedure TfrmD3D.FormCreate(Sender: TObject);
var
  hRet : HRESULT;
  i : Integer;
begin
  Randomize;
  for i := 0 to SIZE — 1 do begin      // Заполнение массива точек
    Points[i].X := random * 2 — 1.0;
    Points[i].Y := random * 2 — 1.0;
  end;
  hRet := InitD3D;
  if Failed (hRet) then ErrorOut ('InitD3D', hRet);
  hRet := InitVB;             // Буферы вершин под (SIDES + 1) вершину
  if Failed (hRet) then ErrorOut ('InitVB', hRet);
  hRet := InitTexture ('../Mandrill.bmp');
  if Failed (hRet) then ErrorOut ('InitTexture', hRet);
end;
```

При рисовании отдельного мазка текстурные координаты всех вершин одинаковы и связаны с координатами точки:

```
function TfrmD3D.DrawCircle (const inX, inY : Single) : HRESULT;
const
  Step = 2 * Pi / SIDES;
var
  Vertices : ^TCustomVertex;
  hRet : HRESULT;
  i : Integer;
begin
  hRet := FD3DVB.Lock(0, (SIDES + 1) * SizeOf(TCustomVertex),
                      PByte(Vertices), 0);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Первая точка, точка центра мазка
  Vertices.X := inX;
  Vertices.Y := inY;
  Vertices.Z := 0.0;
  Vertices.U := (inX + 1.0) / 2;
  Vertices.V := (inY + 1.0) / 2;
  Inc(Vertices);
  // Точки, лежащие на краю круга
  for i := 0 to SIDES do begin
    Vertices.X := inX + sin(i * Step) * Radius; // По часовой стрелке
    Vertices.Y := inY + cos(i * Step) * Radius;
    Vertices.Z := 0;
    Vertices.U := (inX + 1.0) / 2;
    Vertices.V := (inY + 1.0) / 2;
    Inc(Vertices);
  end;
  hRet := FD3DVB.Unlock;
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Связанные треугольники выстраиваются в полный круг
  Result := FD3DDevice.DrawPrimitive(D3DPT_TRIANGLEFAN, 0, SIDES);
end;
```

Пользуемся мы этой функцией отдельно для каждого элемента массива:

```
for i := 0 to SIZE – 1 do begin
  hRet := DrawCircle (Points [i].X, Points [i].Y);
```

```
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
end;
```

Размер мазков регулируется. Можно также получить новый набор точек, чтобы подобрать самую эффектную картинку:

```
procedure TfrmD3D.FormKeyDown(Sender: TObject; var Key: Word;
                              Shift: TShiftState);
var
  i : Integer;
begin
  if Key = VK_ESCAPE then Close else
  if Key = VK_INSERT then Radius := Radius + 0.005 else
  if Key = VK_DELETE then Radius := Radius - 0.005 else
  if Key = VK_SPACE then begin       // Заново генерируем набор точек
    for i := 0 to SIZE - 1 do begin
      Points[i].X := random * 2 - 1.0;
      Points[i].Y := random * 2 - 1.0;
    end;
  end;
end;
```

Надо приложить совсем немного усилий, и вы можете попробовать себя в качестве художника. Достаточно запустить откомпилированный модуль из каталога Ex11. Работа его похожа на предыдущий, но точки массива теперь генерируются не в пределах всего окна, а в области расположения курсора, при нажатой кнопке мыши:

```
procedure TfrmD3D.FormMouseMove(Sender: TObject; Shift: TShiftState;
                                X, Y: Integer);
var
  i : Integer;
begin
  if Down then begin             // Нажата ли кнопка мыши
    // Сравниваем с предыдущим расположением курсора
    if (X <> LastX) and (Y <> LastY) then begin
    for i := 1 to 20 do begin    // Берется 20 точек облачка
                                 // вокруг курсора
    NumPoints := (NumPoints + 1) mod SIZE;
    // Масштабируем точки для системы координат D3DFVF_XYZ
    Points[NumPoints].X := ((X + random (7) - 3)/ ClientWidth) * 2 - 1.0;
    Points[NumPoints].Y := ((ClientHeight -
                         (Y + random (7) - 3)) / ClientHeight) * 2 - 1.0;
```

```
      LastX := X;
      LastY := Y;
      end;
      end;
  end;
end;
```

Для каждого мазка строится набор из 21 треугольника. Как следствие, имеем низкое значение FPS, уменьшающееся с каждым мазком. При маленьком размере кругов подобный подход неэффективен, и нам стоит поискать альтернативное решение такой задачи.

Альфа-составляющая текстуры

Формат текстуры `D3DFMT_A8R8G8B8` позволяет для каждого пиксела образа задавать индивидуальное значение альфа-составляющей, чем можно воспользоваться для получения массы интересных эффектов. Так, проект каталога Ex12 решает задачу, сходную задаче предыдущего примера: курсор при своем движении по поверхности окна оставляет след постепенно проступающего образа (рис. 8.9).

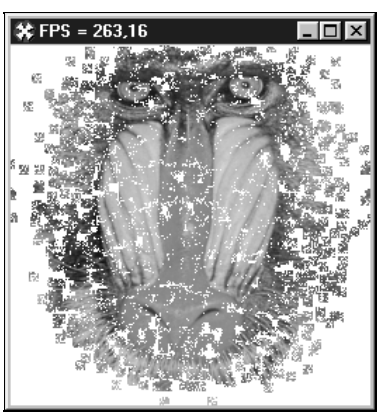


Рис. 8.9. Постепенно проступающий образ: пример использования альфа-составляющей текстуры

Но теперь образ проступает точка за точкой. Мы не используем множество примитивов, и работает пример гораздо быстрее предыдущего.

При заполнении прямоугольника текстуры значение альфа-составляющей для каждого пиксела задается явно, нулевым значением:

```
PDWORD (DWORD(d3dlr.pBits) + Y * dwDstPitch + X * 4)^ :=
        D3DCOLOR_ARGB(0, R, G, B);
```

То есть все точки образа текстуры первоначально задаются совершенно прозрачными.

При воспроизведении квадрата с наложенной текстурой разрешаем работу с альфа-составляющей привычным для нас способом. Помимо этого необходимо отметить, что значение альфа для каждого пиксела текстуры определяется содержимым ее образа:

```
with FD3DDevice do begin
  SetTexture(0, FD3Texture);    // Устанавливаем текстуру
  // Операции с цветом пикселов текстуры
  SetTextureStageState(0, D3DTSS_COLOROP, D3DTA_TEXTURE);
  // Операции с альфа-компонентом пикселов текстуры
  SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTA_TEXTURE);
  // Разрешаем работу с альфа-составляющей
  SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD (True));
  // Параметры альфа-смешения
  SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
  SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
end;
// Квадрат, покрытый текстурой
hRet := FD3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
if FAILED(hRet) then begin
  Result := hRet;
  Exit;
end;
// Выключаем текстуру и альфа-смешение
with FD3DDevice do begin
  SetTexture(0, nil);
  SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD (False));
end;
```

При движении курсора увеличиваем значение альфа-составляющей для пикселов растра текстуры так, чтобы они стали непрозрачными:

```
procedure TfrmD3D.FormMouseMove(Sender: TObject; Shift: TShiftState;
                                X, Y: Integer);
var
  d3dlr : TD3DLOCKED_RECT;
  dwDstPitch : DWORD;
  i : Integer;
  wrkX, wrkY : DWORD;
begin
  if Down then begin             // Нажата ли кнопка мыши
    FD3Texture.LockRect(0, d3dlr, nil, 0);
    dwDstPitch := d3dlr.Pitch;
    for i := 1 to 50 do begin    // 50 точек в районе курсора
    repeat                       // Генерируем точку в пределах окна
      wrkX := DWORD (X + random (7) - 3);
      wrkY := DWORD (ClientHeight - Y + random (7) - 3);
```

```
    until (wrkX < DWORD (ClientWidth)) and (wrkY < DWORD (ClientHeight))
                                       and (wrkX > 0) and (wrkY > 0);
    PDWORD (DWORD(d3dlr.pBits) + wrkY * dwDstPitch + wrkX * 4)^ :=
    // Альфа-составляющую для точек задаем равной 255
    PDWORD (DWORD(d3dlr.pBits) + wrkY * dwDstPitch + wrkX * 4)^ +
           $FF000000;
   end;
   FD3Texture.UnlockRect(0);
  end;
end;
```

Несколько небольших замечаний:

- размер окна я установил равным размеру образа, поэтому нет необходимости соотносить координаты курсора и положение пикселов в растре;
- преобразование типа используется только для того, чтобы не появлялось замечание компилятора;
- нельзя допускать попытки записи за пределы прямоугольной области текстуры;
- при движении курсора по непрозрачным пикселам полагаемся на то, что исключение, связанное с переполнением, возникать не будет.

Проверка границ генерируемых значений точек необходима еще для того, чтобы при нахождении курсора вблизи границ окна не появлялся след на его противоположном краю. Однако эта проверка может оказать плохую службу нашему приложению: если удерживать кнопку мыши и переместить курсор за пределы окна, приложение зацикливается на безуспешной попытке сгенерировать точки в указанных пределах. Чтобы вы убедились, как важно проследить за подобными вещами, в этом примере я оставлю брешь, а в следующем устраню ее: достаточно добавить проверку нахождения курсора в пределах клиентской области окна.

Мультитекстурирование

Для помещения на объект нескольких текстур одновременно можно воспользоваться простейшим способом альфа-смешения: воспроизводить несколько раз один и тот же полупрозрачный объект с наложением различных текстур.

Сначала попробуем добиться того, чтобы на экране просто присутствовали несколько текстур одновременно. Это сделано в проекте каталога Ex13, где экран покрыт одним образом, а курсор оставляет за собой след, в котором просвечивает другой образ (рис. 8.10).

В коде появилось два объекта, связанных с текстурами, `FD3Texture1` и `FD3Texture2`, второй заполняется согласно содержимому загружаемого рас-

трового изображения, а первый — с помощью отдельной функции черно-белой клеткой:

```
function TfrmD3D.InitTexture1 : HRESULT;
var
  hRet : HRESULT;
  d3dlr : TD3DLOCKED_RECT;
  dwDstPitch : DWORD;
  X, Y : DWORD;
begin
  hRet := FD3DDevice.CreateTexture (256, 256, 0, 0,
                                    D3DFMT_A8R8G8B8,
                                    D3DPOOL_MANAGED, FD3Texture1);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := FD3Texture1.LockRect(0, d3dlr, nil, 0);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  dwDstPitch := d3dlr.Pitch;
  for X := 0 to 255 do
    for Y := 0 to 255 do
      // Клетка 16x16
      if ((X shr 4) and 1) xor ((Y shr 4) and 1) = 0
 then PDWORD (DWORD(d3dlr.pBits) + Y * dwDstPitch + X * 4)^ := $FF000000
 else PDWORD (DWORD(d3dlr.pBits) + Y * dwDstPitch + X * 4)^ := $FFFFFFFF;
  Result := FD3Texture1.UnlockRect(0);
end;
```

Рис. 8.10. Симпатичный пример на тему присутствия двух текстур на экране

Для обоих растров значение альфа-составляющей цвета всех пикселов равно максимуму, оба образа первоначально непрозрачны.

Текущее значение булевой переменной `First` задает, какая из текстур отображается первой и будет потом перекрываться вторым образом:

```
with FD3DDevice do begin
  if First
    then SetTexture(0, FD3Texture2)    // Картинка внизу, фон
    else SetTexture(0, FD3Texture1);   // Внизу клетки
      SetTextureStageState(0, D3DTSS_COLOROP, D3DTA_TEXTURE);
      SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTA_TEXTURE);
      SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD (True));
      SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
      SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
end;
// Квадрат, закрытый первым растром, будет фоном
hRet := FD3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
if FAILED(hRet) then begin
  Result := hRet;
  Exit;
end;
if First    // Накладываем вторую текстуру из нашей пары
  then FD3DDevice.SetTexture(0, FD3Texture1)
  else FD3DDevice.SetTexture(0, FD3Texture2);
  hRet := FD3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
if FAILED(hRet) then begin
  Result := hRet;
  Exit;
end;
with FD3DDevice do begin
  SetTexture(0, nil);
  SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD (False));
end;
```

Нажимая на цифровые клавиши, можно менять порядок, в котором накладываются текстуры:

```
procedure TfrmD3D.FormKeyDown(Sender: TObject; var Key: Word;
                              Shift: TShiftState);
begin
  if Key = VK_ESCAPE then Close else
  if Key = Ord ('1') then First := True else   // Клетки сверху
  if Key = Ord ('2') then First := False;      // Клетки снизу
end;
```

При движении указателя мыши располагающиеся в районе курсора пикселы текстуры, находящейся сверху, делаем прозрачными.

```
procedure TfrmD3D.FormMouseMove(Sender: TObject; Shift: TShiftState;
                                X, Y: Integer);
var
  d3dlr : TD3DLOCKED_RECT;
  dwDstPitch : DWORD;
  i : Integer;
  wrkX, wrkY : DWORD;
begin
  // Добавилась проверка положения курсора
  if Down and (X > 0) and (X < ClientWidth)
          and (Y > 0) and (Y < ClientHeight) then begin
    if First     // Определяемся, в какой текстуре вносятся изменения
       then FD3Texture1.LockRect(0, d3dlr, nil, 0)
       else FD3Texture2.LockRect(0, d3dlr, nil, 0);
    dwDstPitch := d3dlr.Pitch;
    for i := 1 to 50 do begin
      repeat
        wrkX := DWORD (X + random (7) - 3);
        wrkY := DWORD (ClientHeight - Y + random (7) - 3);
      until (wrkX < DWORD (ClientWidth)) and
            (wrkY < DWORD (ClientHeight));
    // Значение альфа-составляющей пикселов сбрасываем в ноль
    PDWORD (DWORD(d3dlr.pBits) + wrkY * dwDstPitch + wrkX * 4)^ :=
      PDWORD (DWORD(d3dlr.pBits) + wrkY * dwDstPitch + wrkX * 4)^ -
              $FF000000;
  end;
  if First
    then FD3Texture1.UnlockRect(0)
    else FD3Texture2.UnlockRect(0);
  end;
end;
```

Обратите внимание на дополнительную проверку положения курсора, о ней я говорил при описании предыдущего примера. Данный пример безболезненно воспринимает "уход" курсора с клиентской области окна приложения.

Поскольку изменения вносятся в текстуру, воспроизводимую второй текстурой, то узор остается только на одной из них, и при переключении порядка их воспроизведения нарисованный узор как-будто не проявляется. Но в точках пересечения двух независимых узоров, нанесенных на разные текстуры, пикселы обеих текстур становятся прозрачными, и в этих местах проступает белый цвет, которым закрашивается задний буфер (рис. 8.11).

Думаю, мы достаточно подробно разобрали пример использования нескольких текстур, и мне остается только предложить вам посмотреть работу приложения, когда значение альфа-составляющей для всех пикселов обеих тек-

стур с самого начала равно 127. В этом случае на экране проступают два образа одновременно, а картинка, выводимая последней, выглядит чуть "плотнее" первой.

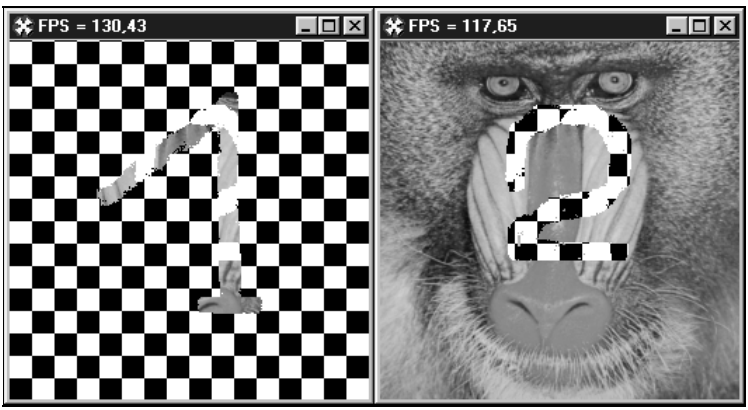


Рис. 8.11. Места пересечения узоров становятся совершенно прозрачными

Цветовой ключ текстур

С помощью манипулирования значения альфа-составляющей пикселов текстуры можно добиться прозрачности ее участков, окрашенных в цвет заднего плана, и применить цветовой ключ.

В очередном примере, проекте каталога Ex14, рисуется дерево на фоне кирпичной стены, участки растра с изображением дерева, имеющие чистый белый цвет, прозрачны (рис. 8.12).

Рис. 8.12. Наложение текстуры с использованием цветового ключа

Аналогично предыдущим примерам, используются два объекта текстур:

```
FD3TextBrick    : IDIRECT3DTEXTURE8;    // Кирпичная кладка
FD3TextBmp      : IDIRECT3DTEXTURE8;    // Дерево
```

Размеры обоих растров установлены 128×128 пикселов. Координаты вершин квадрата, на который накладываются последовательно обе текстуры, установлены в единицы, чтобы закрыть весь экран.

Функция инициализации текстуры из растра отличается тем, что в нее передается значение цветового ключа. При заполнении текстуры пикселы, имеющие такой цвет, делаются полностью прозрачными:

```
function TfrmD3D.InitTexture (const FileName : String;
                              const keyR, keyG, keyB : Byte) : HRESULT;
var
  hRet : HRESULT;
  d3dlr : TD3DLOCKED_RECT;
  dwDstPitch : DWORD;
  X, Y : DWORD;
  Bmp : tBitmap;
  R, G, B : Byte;
begin
  Bmp := TBitmap.Create;
  Bmp.LoadFromfile (FileName);
  hRet := FD3DDevice.CreateTexture (Bmp.Width, Bmp.Height, 0, 0,
                                    D3DFMT_A8R8G8B8,
                                    D3DPOOL_MANAGED, FD3TextBmp);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := FD3TextBmp.LockRect(0, d3dlr, nil, 0);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  dwDstPitch := d3dlr.Pitch;
  for Y := 0 to Bmp.Height - 1 do
    for X := 0 to Bmp.Width - 1 do begin
      R := GetRValue (Bmp.Canvas.Pixels [X, DWORD (Bmp.Height - 1) - Y]);
      G := GetGValue (Bmp.Canvas.Pixels [X, DWORD (Bmp.Height - 1) - Y]);
      B := GetBValue (Bmp.Canvas.Pixels [X, DWORD (Bmp.Height - 1) - Y]);
      // Сравнение цвета пиксела с цветовым ключом
      if (R = keyR) and (G = keyG) and (B = keyB)
        then PDWORD (DWORD(d3dlr.pBits) + Y * dwDstPitch + X * 4)^:=
           D3DCOLOR_ARGB(0, R, G, B)      // Такой пиксел
                                          // должен стать прозрачным
```

```
        else PDWORD (DWORD(d3dlr.pBits) + Y * dwDstPitch + X * 4)^ :=
           D3DCOLOR_ARGB(255, R, G, B);   // Все остальные пикселы
                                          // непрозрачны
  end;
  Bmp.Free;
  Result := FD3TextBmp.UnlockRect(0);
end;
```

Кирпичная кладка создается "вручную", дополнительные растры не используются:

```
function TfrmD3D.MakeBrick : HRESULT;
var
  hRet : HRESULT;
  d3dlr : TD3DLOCKED_RECT;
  dwDstPitch : DWORD;
  X, Y, wrkStep : DWORD;
begin
  wrkStep := 0;
  hRet := FD3DDevice.CreateTexture (128, 128, 0, 0,
                                    D3DFMT_A8R8G8B8,
                                    D3DPOOL_MANAGED, FD3TextBrick);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := FD3TextBrick.LockRect(0, d3dlr, nil, 0);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  dwDstPitch := d3dlr.Pitch;
  for Y := 0 to 127 do
    for X := 0 to 127 do
    // Горизонтальные полоски — через каждые 10 пикселов
      if Y mod 10 = 0 then begin
      // Полоски сероватого цвета
        PDWORD (DWORD(d3dlr.pBits) + Y * dwDstPitch + X * 4)^ :=
                D3DCOLOR_XRGB(200 + random (30), 200+ random (30),
                200+ random (30));
        Inc (wrkStep);    // Сдвиг для вертикальных полосок
      end else
      // Вертикальные полоски сдвигаются через каждый ряд кладки
      if (X + wrkStep) mod 20 = 0
        then PDWORD (DWORD(d3dlr.pBits) + Y * dwDstPitch + X * 4)^ :=
             D3DCOLOR_XRGB(200 + random (30), 200+ random (30),
                           200+ random (30))
```

```
      // Собственно кирпичи
      else PDWORD (DWORD(d3dlr.pBits) + Y * dwDstPitch + X * 4)^ :=
           D3DCOLOR_XRGB(150 + Random(100), 80, 10);
   Result := FD3TextBrick.UnlockRect(0);
end;
```

Квадрат воспроизводится дважды. При наложении второй текстуры включаем альфа-смешение. Все сопутствующие действия аналогичны уже рассмотренным примерам.

Спрайты в Direct3D

Итак, мы знаем все, чтобы познакомиться с альтернативным DirectDraw способом создания настоящих 2D-приложений, являющихся частным случаем 3D-графики. Спрайты теперь представляются в виде примитивов, на которые накладывается текстура.

В проекте каталога Ex15 реализована несложная иллюстрация такого подхода, во время работы ее по экрану меланхолично проплывают рыбки (рис. 8.13).

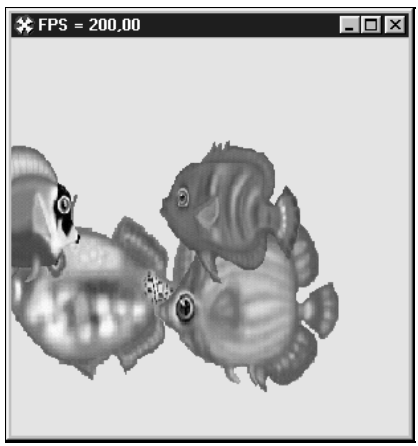


Рис. 8.13. Пример создания спрайтов в Direct3D

Как обычно для многих примеров этой книги, массив объектов предопределенного размера содержит объекты отдельных изображений:

```
type
  TFish = class                    // Отдельная рыбка
    private
      FD3Texture : IDIRECT3DTEXTURE8;
    public
      PosX, PosY, StepX : Single;  // Позиция на экране и шаг перемещения
      Scale : Single;              // Масштабный множитель
      function RotateTexture : HRESULT;      // Поворот текстуры
```

```
      function Draw : HRESULT;     // Собственно отображение на экране
      procedure Move;              // Движение по экрану
constructor Create (const FileName : String; const inR, inG, inB : Byte);
      destructor Destroy; override;
  end;
const
  NumFish = 10;                     // Количество рисуемых рыбок
var
  Fishes : Array [0..NumFish-1] of TFish; // Массив объектов
```

В конструктор объекта передается имя файла-растра, а также информация о цвете, который необходимо использовать в качестве цветового ключа:

```
constructor TFish.Create (const FileName : String;
                          const inR, inG, inB : Byte);
var
  hRet : HRESULT;
  d3dlr : TD3DLOCKED_RECT;
  dwDstPitch : DWORD;
  X, Y : DWORD;
  Bmp, wrkBmp : TBitmap;
  R, G, B : Byte;
begin
  Bmp := TBitmap.Create;
  Bmp.LoadFromfile (FileName);
  wrkBmp := TBitmap.Create;
  wrkBmp.Width := 128;
  wrkBmp.Height := 128;
  // Масштабирование исходного растра
  wrkBmp.Canvas.StretchDraw (Rect (0, 0, 128, 128), Bmp);
  hRet := frmD3D.FD3DDevice.CreateTexture (wrkBmp.Width, wrkBmp.Height,
          0, 0, D3DFMT_A8R8G8B8, D3DPOOL_MANAGED, FD3Texture);
  if FAILED(hRet) then begin
    if Failed (hRet) then frmD3D.ErrorOut ('InitTexture', hRet);
    Exit;
  end;
  hRet := FD3Texture.LockRect(0, d3dlr, nil, 0);
  if FAILED(hRet) then begin
    if Failed (hRet) then frmD3D.ErrorOut ('InitTexture', hRet);
    Exit;
  end;
  dwDstPitch := d3dlr.Pitch;
  for Y := 0 to wrkBmp.Height - 1 do
    for X := 0 to wrkBmp.Width - 1 do begin
      R := GetRValue(wrkBmp.Canvas.Pixels[X, DWORD (wrkBmp.Height-1)-Y]);
      G := GetGValue(wrkBmp.Canvas.Pixels[X, DWORD (wrkBmp.Height-1)-Y]);
      B := GetBValue(wrkBmp.Canvas.Pixels[X, DWORD (wrkBmp.Height-1)-Y]);
```

```
      // Пикселы предопределенного цвета делаются прозрачными
      if (R = inR) and (G = inG) and (B = inB)
         then PDWORD (DWORD(d3dlr.pBits) + Y * dwDstPitch + X * 4)^ :=
              D3DCOLOR_ARGB(0, R, G, B)
         else PDWORD (DWORD(d3dlr.pBits) + Y * dwDstPitch + X * 4)^ :=
              D3DCOLOR_ARGB(255, R, G, B);
    end;
  hRet := FD3Texture.UnlockRect(0);
  if FAILED(hRet) then begin
    if Failed (hRet) then frmD3D.ErrorOut ('InitTexture', hRet);
    Exit;
  end;
  Bmp.Free;
  wrkBmp.Free;
end;
```

Итак, все растровые изображения масштабируются под размер 128×128 пикселов, и поскольку исходные картинки прямоугольные, некоторые из них окажутся немного искаженными в своих пропорциях.

В начале работы приложения случайным образом выбираются картинки для загрузки, одна из них нарисована на красном фоне, остальные — на зеленом:

```
for i := 0 to NumFish - 1 do begin      // Инициализация массива объектов
  case random (4) of
    0 : Fishes [i] := TFish.Create ('Fish1.bmp', 0, 255, 0);
    1 : Fishes [i] := TFish.Create ('Fish2.bmp', 255, 0, 0);
    2 : Fishes [i] := TFish.Create ('Fish3.bmp', 0, 255, 0);
    3 : Fishes [i] := TFish.Create ('Fish4.bmp', 0, 255, 0);
  end;
  with Fishes [i] do begin
    PosX := random - 0.5;
    PosY := (random (60) - 30) / 100;
    StepX := (random - 0.5) / 10;
    if StepX < 0 then RotateTexture;    // Требуется поворот
    Scale := (random (60) + 40) / 100;
  end;
end;
```

В исходных образах рыбки нарисованы плывущими слева направо, при обратном движении содержимое прямоугольника переворачивается:

```
function TFish.RotateTexture : HRESULT;
var
  d3dlr : TD3DLOCKED_RECT;
  dwDstPitch : DWORD;
  pDst, pDst1 : PDWORD;
```

```
  X, Y : DWORD;
  wrkDW : DWORD;
begin
  FD3Texture.LockRect(0, d3dlr, nil, 0);
  dwDstPitch := d3dlr.Pitch;
  for Y := 0 to 127 do
    for X := 0 to 63 do begin     // До половины ширины образа
    // Переставляем содержимое двух пикселов
      pDst := PDWORD (DWORD(d3dlr.pBits) + Y * dwDstPitch + X * 4);
      pDst1 := PDWORD (DWORD(d3dlr.pBits) + Y * dwDstPitch +
                (127 - X) * 4);
      wrkDW := pDst^;
      pDst^ := pDst1^;
      pDst1^ := wrkDW;
  end;

  Result := FD3Texture.UnlockRect(0);
end;
```

Буфер вершин инициализируется с размером под четыре вершины, поскольку для изображения рыбки этот буфер заполняется координатами четырех сторон квадрата. Размер стороны квадрата — `Scale`:

```
function TFish.Draw : HRESULT;
var
  Vertices : ^TCustomVertex;
  hRet : HRESULT;
begin
  hRet := frmD3D.FD3DVB.Lock(0, 4 * SizeOf(TCustomVertex),
          PByte(Vertices), 0);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  Vertices.X := -0.5 + PosX;          // Левый нижний угол квадрата
  Vertices.Y := -0.5 + PosY;
  Vertices.Z := 0;
  Vertices.U := 0;
  Vertices.V := 0;
  Inc(Vertices);
  Vertices.X := -0.5 + PosX;          // Левый верхний угол квадрата
  Vertices.Y := -0.5 + Scale + PosY;
  Vertices.Z := 0;
  Vertices.U := 0;
  Vertices.V := 1;
  Inc(Vertices);
  Vertices.X := -0.5 + Scale + PosX;  // Правый нижний угол квадрата
  Vertices.Y := -0.5 + PosY;
```

```
  Vertices.Z := 0;
  Vertices.U := 1;
  Vertices.V := 0;
  Inc(Vertices);
  Vertices.X := -0.5 + Scale + PosX;  // Правый верхний угол квадрата
  Vertices.Y := -0.5 + Scale + PosY;
  Vertices.Z := 0;
  Vertices.U := 1;
  Vertices.V := 1;
  frmD3D.FD3DVB.Unlock;
  with frmD3D.FD3DDevice do begin
    SetTexture(0, FD3Texture);
    SetTextureStageState(0, D3DTSS_COLOROP, D3DTA_TEXTURE);
    SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTA_TEXTURE);
    SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD (True));
    SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
    SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
    DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
    SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD (False));
  end;
  Result := frmD3D.FD3DDevice.SetTexture(0, nil);
end;
```

Через некоторый промежуток времени для каждого объекта вызывается метод, связанный с перемещением:

```
procedure TFish.Move;
begin
  PosX := PosX + StepX;
  if (PosX < -1.5) or (PosX > 1.5) then begin  // Уход за границу экрана
    RotateTexture;                             // Переворачиваем образ
    StepX := -StepX;            // Меняем направление на противоположное
  end;
end;
```

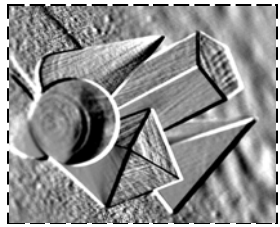
Как видим, код значительно сократился, и программировать стало гораздо удобнее, но видеокарты, поддерживающие только 2D-акселерацию, не смогут ускорить работу таких приложений.

Что вы узнали в этой главе

Примеры данной главы позволили нам совершенно освоиться с ключевыми методами Direct3D. Мы научились работать с альфа-составляющей цвета примитивов, познакомились с важнейшим понятием текстуры.

Игры и многие эффекты программируются теперь гораздо легче, чем при использовании DirectDraw.

ГЛАВА 9



Трехмерные построения

Предлагаемый материал во многом основан на сведениях предыдущей главы, но имеет принципиальное отличие: в ней мы переходим к трехмерной графике.

Такой переход потребует небольшого экскурса в линейную алгебру. В остальном же знакомство с новой темой потребует изучения методов объектов, связанных с 3D графикой.

Примеры, рассматриваемые в данной главе, располагаются в каталоге \Examples\Chapter09.

Матричный подход

Прежде, чем мы приступим к рисованию в пространстве, нам предстоит поговорить о некоторых важных вещах, обойти которые невозможно, хотя они напрямую, казалось бы, и не связаны с программированием.

Вкратце повторим подходы, используемые нами в предыдущих главах, посвященных Direct3D. Буфер вершин заполняется данными некоторого формата об опорных вершинах, образующих примитивы. Если примитивы должны перемещаться по экрану, буфер вершин заполняется новыми данными. Для поворота объекта надо запереть буфер, чтобы получить доступ к его содержимому, и заполнить буфер новыми данными.

В трехмерных построениях мы будем избегать такого подхода. Использованные нами ранее форматы данных о вершинах содержат три пространственные координаты, и нетрудно догадаться, что для перехода к трехмерной графике надо для начала задействовать Z-координату, ранее нами игнорируемую. Конечно, потребуются еще некоторые действия, но интуиция подсказывает, что для рисования, например, кубика, надо построить треугольники, образующие стороны куба, манипулируя значением третьей коорди-

наты. А для того, чтобы нарисовать вращающийся кубик, следует периодически обновлять содержимое буфера вершин. Но мы сразу же должны оговориться, что было бы лучше, если бы мы один раз заполняли буфер данными о кубике, а воспроизводили его каждый раз немного повернутым относительно предыдущего положения. Конечно, это оптимально: заполнить буфер один раз массивом данных об объектах сцены, а при воспроизведении каждого объекта выполнять менее требовательные к ресурсам операции, указывая его текущее положение в пространстве. К такому порядку действий мы и будем стремиться. Не использовал я такого подхода раньше только потому, что боялся нагрузить вас обилием материала (этого я боюсь и сейчас), и хотел бы, чтобы мы двигались шаг за шагом. Но, к сожалению, сейчас нам придется сделать очень большой скачок, и для того, чтобы не споткнуться, следует утроить внимание. Начнем.

При описании объекта, заполнении буфера вершин опираемся на мировую систему координат. Иными словами, указываем координаты вершин объектов так, как будто все они находятся в точке начала глобальной системы координат.

Объекты трехмерной сцены наделяются системой координат, первоначально совпадающей с мировой системой. Каждая трансформация системы координат, связанной с объектом, приведет к трансформации объекта. Если перед воспроизведением объекта сместить его систему координат, то объект будет рисоваться на новом месте, т. е. относительно смещенной по одной или нескольким осям системы координат. Для осуществления поворота объекта поворачиваем систему координат, связанную с ним, вокруг одной из осей. Если на сцене присутствует несколько объектов, то перед рисованием каждого из них трансформируем систему координат, ассоциированную с этим объектом.

Надеюсь, пока все понятно и просто, и мы можем поговорить о том, как собственно осуществлять манипуляции с системой координат объекта. Самыми популярными математическими методами для описания таких преобразований служат векторный и матричный. Трехмерная графика базируется, как правило, на матричном подходе, заключающемся в том, что операции с системой координат основываются на матричном представлении. Базовым элементом матричного метода является матрица (таблица чисел) размером 4×4. Я знаю первый вопрос, который возникает всегда и у всех, кто впервые слышит о матричном методе: почему размер матрицы именно такой. В математике для описания точки в пространстве используется четыре числа, вспомогательной характеристике можно придать любой смысл, это может быть, например, заряд частицы или материальная масса. В графике четвертый компонент координаты точки называется W-координатой и предназначен для осуществления проекции точки на плоскость экрана. Это весовой фактор, на который умножаются координаты точки при ее проецировании. Его значение задается единичным.

Основной операцией, к которой прибегают при манипуляции с матрицами, является перемножение матриц, осуществляемое по формуле:

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

Количество строк перемножаемых матриц должно быть одинаковым.

При умножении матрицы на вектор первым множителем слагаемых суммы берутся последовательно элементы единственного столбца вектора.

Единичная матрица, т. е. матрица, по главной диагонали которой располагаются единицы, а все остальные элементы равны нулю, соответствует мировой системе координат. Другое название такой матрицы — *матрица идентичности*, после умножения ее на вектор получается исходный вектор.

Матрицы сдвига по осям X, Y и Z выглядят так:

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & 0 & 0 & 1 \end{matrix} \quad \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & b & 0 & 1 \end{matrix} \quad \begin{matrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & c & 1 \end{matrix}$$

Если умножить вектор *(X, Y, Z, W)* на матрицу сдвига по оси X, в результате получится вектор *(X + W · a, Y, Z, W)*. Умножение вектора координат всех точек объектов на матрицу сдвига приводит к перемещению объекта по нужной оси.

Три матрицы сдвига можно объединить в одну, дающую возможность осуществлять сдвиг одновременно по нескольким осям. Последняя строка такой матрицы имеет ненулевые значения в столбцах, соответствующих нужным осям.

Возвращаясь в Direct3D, поясню: у объекта устройства есть метод, позволяющий задать матрицу, на которую будут умножаться векторы координат вершин непосредственно перед отображением в пространстве. И пока в качестве такой матрицы указана матрица сдвига, все воспроизводимые объекты будут сдвигаться в пространстве.

Аналогично сдвигу, операции поворота описываются матрицами. Для поворота на угол α вокруг оси X вектор координат вершины надо умножить на такую матрицу:

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Если же надо повернуть на угол β вокруг оси Y, то пользуются такой матрицей:

$$\begin{matrix} \cos\beta & 0 & -\sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

И последняя ситуация с поворотом: угол γ, поворот вокруг оси Z:

$$\begin{matrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Чтобы осуществить одновременный поворот по нескольким осям либо скомбинировать поворот и сдвиг, надо применить в качестве трансформаций произведение нужных матриц. При этом важен порядок, в котором мы перемножаем матрицы, он определяет последовательность трансформаций системы координат.

Операции с объектами осуществляются в трехмерном пространстве, описываемом матрицей, которую будем называть *мировой матрицей*. Помимо мировой матрицы требуется указать *видовую матрицу*, соответствующую позиции глаза наблюдателя и направлению, в котором он смотрит. В принципе, ее можно задавать точно так же, как и мировую, используя матрицы сдвига и поворота.

Последняя матрица, которая нужна для получения проекции трехмерной сцены на экране, так и называется — *матрицей проекции*. Значения элементов этой матрицы задают правила, согласно которым будет осуществляться проецирование: положение задней и передней отсекающих плоскостей, искажения, имитирующие перспективу (рис. 9.1).

Объекты, или части объектов, располагающиеся за пределами области видимости, на экран проецироваться не будут, и мы их не увидим.

Итак, мы бросили беглый взгляд на сухой теоретический материал, из которого вынесли тяжелое подозрение, что впереди нас ожидает бурелом кодирования математических формул. Отчасти это правда. Direct3D оставил программисту тяготы перемножения матриц, ожидая от него три результирующие матрицы трансформаций. Однако мы воспользуемся модулем `DXGUtils`, который содержит набор полезных функций. Автор переноса на Delphi кода этих функций указан в заголовке модуля.

В списке подключаемых модулей первого примера этой главы, проекте каталога Ex01, как раз и добавлен указанный модуль. Пример очень простой,

в пространстве вращаются два объекта: разноцветный треугольник и желтый квадрат (рис. 9.2).

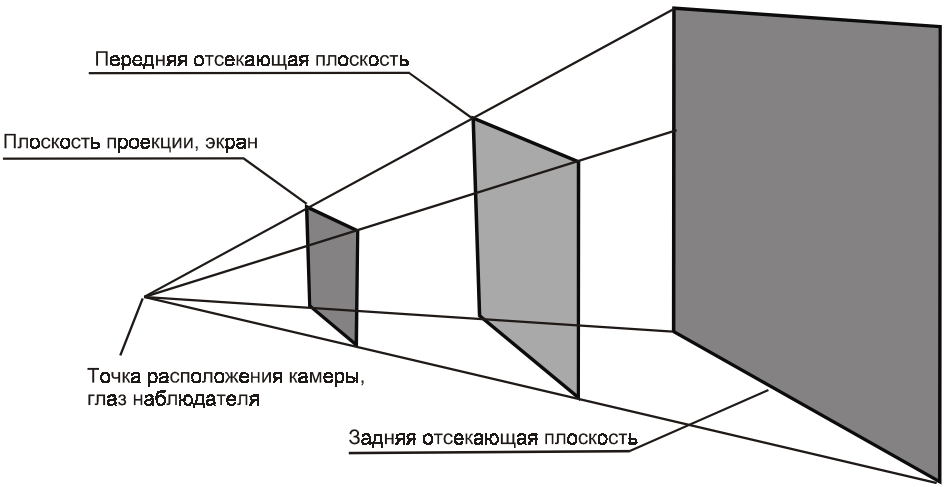


Рис. 9.1. Область видимости задается положением двух отсекающих плоскостей

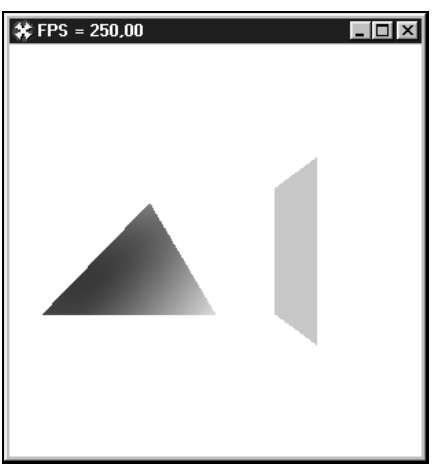


Рис. 9.2. Простейший пример трехмерного построения фигур

Чтобы при вращении примитивов мы могли видеть обе их стороны, режим отсечения отключается, а для использования окрашенных примитивов запрещена работа с источником света:

```
with FD3DDevice do begin
  SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
  SetRenderState(D3DRS_LIGHTING, DWORD (False));
end;
```

Буфер вершин запирается один раз. Семь вершин содержат координаты треугольника и квадрата. Если бы они выводились не трансформируемыми, то накладывались бы друг на друга:

```
Vertices.X := 0.0;              // Первая вершина треугольника
Vertices.Y := 1.0;
Vertices.Z := 0.0;
Vertices.Color := $00FF0000;
Inc(Vertices);
Vertices.X := 1.0;              // Вторая вершина треугольника
Vertices.Y := -1.0;
Vertices.Z := 0.0;
Vertices.Color := $0000FF00;
Inc(Vertices);
Vertices.X := -1.0;             // Третья вершина треугольника
Vertices.Y := -1.0;
Vertices.Z := 0.0;
Vertices.Color := $000000FF;
Inc(Vertices);
Vertices.X := -1.0;             // Первая вершина квадрата
Vertices.Y := -1.0;
Vertices.Z := 0.0;
Vertices.Color := $00FFFF00;
Inc(Vertices);
Vertices.X := -1.0;             // Вторая вершина квадрата
Vertices.Y := 1.0;
Vertices.Z := 0.0;
Vertices.Color := $00FFFF00;
Inc(Vertices);
Vertices.X := 1.0;              // Третья вершина квадрата
Vertices.Y := -1.0;
Vertices.Z := 0.0;
Vertices.Color := $00FFFF00;
Inc(Vertices);
Vertices.X := 1.0;              // Четвертая вершина квадрата
Vertices.Y := 1.0;
Vertices.Z := 0.0;
Vertices.Color := $00FFFF00;
```

При каждой перерисовке кадра вызывается процедура:

```
procedure TfrmD3D.DrawScene;
var
  matView, matProj : TD3DMatrix;        // Матрицы 4x4
  matRotate, matTranslate : TD3DMatrix;
```

```
begin
  // Получить матрицу поворота вокруг оси X
  SetRotateXMatrix(matRotate, Angle);
  // Матрица сдвига по оси X, на единицу влево
  SetTranslateMatrix(matTranslate, -1.0, 0.0, 0.0);
  // Устанавливаем мировую матрицу трансформаций
  FD3DDevice.SetTransform(D3DTS_WORLD,
                          MatrixMul(matRotate, matTranslate));
  // Выводится треугольник
  FD3DDevice.DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
  // Квадрат вращается по оси Y в 2 раза быстрее треугольника
  SetRotateYMatrix(matRotate, 2 * Angle);
  // Квадрат сдвигается на единицу вправо
  SetTranslateMatrix(matTranslate, 1.0, 0.0, 0.0);
  // Матрица трансформаций для квадрата
  FD3DDevice.SetTransform(D3DTS_WORLD,
                          MatrixMul(matTranslate, matRotate));
  // Вывод квадрата
  FD3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 3, 2);
  // Задаем видовую матрицу
  SetViewMatrix(matView, D3DVector(0, 0, -5),
                D3DVector(0, 0, 0), D3DVector(0, 1, 0));
  // Устанавливаем видовую матрицу
  FD3DDevice.SetTransform(D3DTS_VIEW, matView);
  // Задаем матрицу проекций
  SetProjectionMatrix(matProj, 1, 1, 1, 10);
  // Устанавливаем матрицу проекций
  FD3DDevice.SetTransform(D3DTS_PROJECTION, matProj);
end;
```

Тип `TD3DMatrix`, массив 4×4 вещественных чисел, определен в модуле `DirectXGraphics`, а все функции операций с матрицами — в модуле `DXGUtils`. Эти функции возвращают величину типа `HRESULT`, значение которой мы, для простоты, анализировать не будем.

Функция `D3DVector` этого же модуля возвращает сформированный по трем аргументам вектор, тройку вещественных чисел, величину типа `TD3DVector`.

Функция `SetRotateXMatrix` первым аргументом получает переменную, в которую помещается результат, матрицу поворота вокруг оси X. Второй аргумент — угол, в радианах, на который осуществляется поворот. Функция `SetTranslateMatrix` первым аргументом получает переменную, в которую помещается заполненная матрица сдвига. Одновременно можно сдвинуть по нескольким осям.

Метод `SetTransform` объекта устройства позволяет установить матрицу трансформаций. Первый аргумент — константа, определяющая, для какой

матрицы устанавливается трансформация. Второй аргумент — собственно матрица трансформаций. Здесь мы передаем результирующую матрицу, полученную умножением матрицы поворота и матрицы сдвига, но не обязательно, чтобы в трансформации участвовало несколько матриц. Функция `MatrixMul` позволяет умножить две матрицы, передаваемые в качестве параметров.

Напоминаю, что порядок перечисления этих матриц очень важен. В данном случае разноцветный треугольник поворачивается вокруг оси X, затем сдвигается на единицу влево, по этой же оси.

Квадрат в этом примере вначале сдвигается вправо, затем поворачивается вокруг оси Y (собственной оси, а не мировой). Измените порядок перемножения матриц, чтобы убедиться, что результат будет отличаться от предыдущего.

Функция `SetViewMatrix` подготавливает видовую матрицу. Параметры функции следующие: матрица, в которую помещается результат, вектор, определяющий точку, где располагается голова наблюдателя, опорная точка, определяющая середину видимой области, и вектор, задающий направление взгляда.

Функция `SetProjectionMatrix` предназначена для удобного определения матрицы проекции. Второй аргумент функции задает угол обзора камеры по оси Y, третий аргумент — отношение, определяющее угол обзора по оси X, последние два аргумента — расстояния от глаза наблюдателя до ближней и дальней плоскостей отсечения.

Подозреваю, что последние две функции вызовут много вопросов, поэтому чуть позже мы подробно разберем их смысл. Пока же мы должны только помнить, что смотрим на сцену со стороны оси Z, и находимся от точки отсчета системы координат на расстоянии 5 единиц.

Реалистичные изображения

Для получения реалистичных изображений необходимо выполнить три условия:

- при описании примитивов задать нормали;
- определить свойство материала;
- включить источник света.

Нормали помогают системе рассчитать освещенность примитива при различном его положении относительно источника света. В самом простом использовании нормаль представляет собой вектор, перпендикулярный воспроизводимому треугольнику. Этот вектор задается для каждой вершины, образующей примитив, и из требований оптимизации должен быть нормализован, т. е. иметь единичную длину.

Формат вершин теперь помимо пространственных координат обязан включать вектор нормали (тройку вещественных чисел), а FVF-флаг должен дополниться константой D3DFVF_NORMAL. Это первое новшество в модуле нашего следующего примера, проекта каталога Ex02, где рисуется красивый желтый кубик (рис. 9.3).

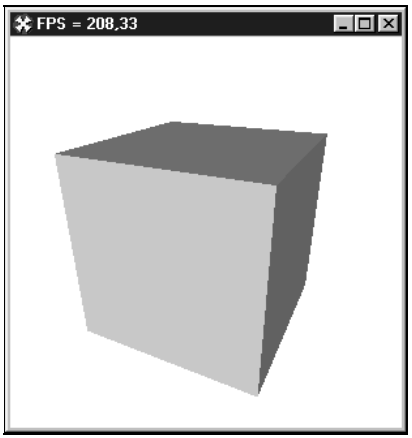


Рис. 9.3. Наше первое реалистичное изображение

Итак, запись описания вершины дополнилась тремя полями:

```
type
  TCUSTOMVERTEX = packed record
    X, Y, Z : Single;
    nX, nY, nZ : Single;        // Вектор нормали
  end;
const
  D3DFVF_CUSTOMVERTEX = D3DFVF_XYZ or D3DFVF_NORMAL;
```

Буфер содержит 36 вершин, предназначенных для построения куба. Они образуют 12 независимых треугольников, по 2 соприкасающихся треугольника на каждую сторону куба. Все треугольники описываются по часовой стрелке, чтобы при воспроизведении мы могли, для экономии времени, отключить воспроизведение примитивов, перечисляемых в поле зрения против часовой стрелки. То есть стороны куба, повернутые к нам задней стороной, воспроизводить не будем, это обычный прием, применяемый к замкнутым трехмерным фигурам. Нормали для всех вершин, относящихся к одной стороне, задаются одинаковыми, исходя из того, какая сторона куба описывается. Так выглядит описание первого треугольника:

```
Vertices.X := -0.5;
Vertices.Y := -0.5;
Vertices.Z := -0.5;
Vertices.nX := -1.0;
```

```
Vertices.nY := 0.0;
Vertices.nZ := 0.0;
Inc(Vertices);
Vertices.X := -0.5;
Vertices.Y := -0.5;
Vertices.Z :=  0.5;
Vertices.nX := -1.0;
Vertices.nY := 0.0;
Vertices.nZ := 0.0;
Inc(Vertices);
Vertices.X := -0.5;
Vertices.Y :=  0.5;
Vertices.Z := -0.5;
Vertices.nX := -1.0;
Vertices.nY := 0.0;
Vertices.nZ := 0.0;
Inc(Vertices);
```

При инициализации графической системы вызывается процедура, задающая свойства материала и включающая источник света:

```
procedure TfrmD3D.SetupLights;
var
  Material  : TD3DMaterial8;
  Light     : TD3DLight8;
begin
  // Инициализация материала, желтый цвет
  Material := InitMaterial(1, 1, 0, 0);
  // Устанавливаем материал в объекте устройства
  FD3DDevice.SetMaterial(Material);
  // Инициализация направленного источника, белый свет
  Light := InitDirectionalLight(D3DVector(0, 0, 1), 1, 1, 1, 0);
  // Устанавливаем источник света
  FD3DDevice.SetLight(0, Light);
  // Включаем источник света
  FD3DDevice.LightEnable(0, True);
end;
```

Материал и источник света являются записями (не COM-объекты) и имеют тип `TD3DMaterial8` и `TD3DLight8` соответственно. Пользовательская функция `InitMaterial` заполняет поля структуры материала и получает в качестве аргументов значения ARGB. Отличает эти параметры от привычного их использования, помимо порядка, в котором они перечисляются, то, что это вещественные числа, единица соответствует максимальному значению аргумента.

В примере материал задается желтым, для того, чтобы установить его. При этом используется метод `SetMaterial` объекта устройства.

Функция `InitDirectionalLight` заполняет поля структуры, описывающей направленный источник света. Первым аргументом передается вектор, задающий направление лучей света. Напоминаю, что мы наблюдаем сцену с отрицательной стороны оси Z. Чтобы лучи света были параллельны нашему взору, вектор направления задается (0, 0, 1). Следующие три аргумента описывают цветовой фильтр, накладываемый на источник света, обычно источник задается белым. Эти числа также вещественны. Значение последнего аргумента для направленного источника безразлично.

Метод `SetLight` объекта устройства устанавливает источник света на сцене. Первый аргумент, целое число, основанное на нуле, является индексом, идентификатором источника света. Метод только задает источник света, включается же он с помощью отдельного метода, `LightEnable`, первый аргумент которого — индекс нужного источника, второй аргумент — булево выражение.

Как я уже говорил, отключается воспроизведение задних сторон треугольников, т. е. тех, чьи вершины перечисляются против часовой стрелки:

```
SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
```

Совсем не обязательно, чтобы вершины примитива перечислялись именно по часовой стрелке, можно использовать и противоположное направление. Просто желательно, чтобы существовал какой-нибудь определенный порядок перечисления, чтобы можно было отсекать воспроизведение задних сторон. Повторюсь, внутренние стороны кубика нам не видны в любом случае, поэтому и незачем тратить время на их воспроизведение. Также обращаю ваше внимание на то, что связанные треугольники приспособлены для перечисления вершин именно по часовой стрелке.

Есть и еще один важный аспект, который нам необходимо учитывать: Direct3D не может окрашивать примитивы с двух сторон. Замените последний аргумент метода `DrawPrimitive` на 2 и установите значение для режима `D3DRS_CULLMODE` в `D3DCULL_NONE`. Теперь будет выводиться только одна сторона куба, отсечение задней стороны примитивов не производится. Обратите внимание, что когда квадрат поворачивается к зрителю задней стороной, он выводится черным, т. е. совершенно не окрашиваемым.

Кубик в нашем примере вращается вокруг двух осей одновременно:

```
SetRotateXMatrix(matRotateX, Angle);
SetRotateYMatrix(matRotateY, Angle);
FD3DDevice.SetTransform(D3DTS_WORLD, MatrixMul(matRotateX, matRotateY));
FD3DDevice.DrawPrimitive(D3DPT_TRIANGLELIST, 0, 12);
```

На рисунке куб получился крупнее, чем при работе приложения. Для того, чтобы увеличить изображение, можно просто "приблизить" глаз наблюдателя:

```
SetViewMatrix(matView, D3DVector(0, 0, -2),
              D3DVector(0, 0, 0), D3DVector(0, 1, 0));
```

Есть и другой способ: действительно увеличить объект. Для этого в матрицу трансформаций надо добавить матрицу масштабирования, по главной диагонали которой стоят числа, отличные от единицы и равные масштабным множителям по трем осям отдельно. Попробуйте сейчас увеличить кубик в два раза:

```
procedure TfrmD3D.DrawScene;
var
  matView, matProj : TD3DMatrix;
  matRotateX, matRotateY : TD3DMatrix;
  matScale : TD3DMatrix;          // Добавилась матрица масштабирования
begin
  SetRotateXMatrix(matRotateX, Angle);
  SetRotateYMatrix(matRotateY, Angle);
  SetScaleMatrix(matScale, 2.0, 2.0, 2.0); // Увеличиваем в 2 раза
  // Добавляем матрицу масштабирования
  FD3DDevice.SetTransform(D3DTS_WORLD, MatrixMul(matScale,
  MatrixMul(matRotateX, matRotateY)));
  ...
```

Обязательно это сделайте, чтобы увидеть, что куб действительно увеличился. Однако освещение его тоже изменилось. Связано это с тем, что векторы нормалей к вершинам вслед за масштабированием стали увеличенными, и требуется их нормализация. В таких случаях необходимо включить режим автоматической нормализации этих векторов:

```
SetRenderState(D3DRS_NORMALIZENORMALS, DWORD (True));
```

Буфер глубины

Продолжим рассмотрение нашего примера с вращающимся кубом. В нем еще остались некоторые новые для нас вещи. Рисуемые примитивы накладываются друг на друга в том порядке, в котором они воспроизводятся: нарисованные позже лежат поверх созданных ранее. Это хорошо для двумерных построений, но при переходе в 3D-пространство нам приходится беспокоиться о том, чтобы положения объектов передавались правильно: более удаленные от глаза наблюдателя объекты могут заслонять воспроизведенные позже, но располагающиеся ближе к камере. Графическая система предлагает решение в виде использования *буфера глубины* — вспомогательного экрана, предназначенного только для сортировки объектов, располагающихся в пространстве. При подключении этого буфера воспроизведение осуществляется дважды: первый раз в буфер записывается информация о значении расстояния от камеры до точки, второй раз в буфер кадра помещаются данные только о точках, действительно видимых и не заслоняемых другими точками.

Примечание

Другое название буфера глубины — *Z-буфер*.

При инициализации Direct3D надо указать, что будет использоваться буфер глубины, и задать его формат. Обычно используется 16-битный формат:

```
with d3dpp do begin
  Windowed := True;
  SwapEffect := D3DSWAPEFFECT_DISCARD;
  BackBufferFormat := d3ddm.Format;
  // Будет использоваться буфер глубины
  EnableAutoDepthStencil := True;
  AutoDepthStencilFormat := D3DFMT_D16;       // 16-битный формат
end;
```

Размеры буфера глубины будут автоматически определяться системой при каждом изменении размеров окна.

При очередной перерисовке кадра теперь должен очищаться не только буфер кадра, но и подключенный буфер глубины. Предпоследний параметр метода `Clear` объекта устройства — значение, которым заполняется буфер глубины. Этим значением должна быть единица, фон экрана бесконечно удален в пространстве:

```
FD3DDevice.Clear(0, nil, D3DCLEAR_TARGET or D3DCLEAR_ZBUFFER,
                 $00FFFFFF, 1.0, 0);
```

Для разрешения работы с буфером глубины надо также задать положительный флаг для соответствующего режима:

```
SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
```

Флагом для этого состояния может быть и обычная булева константа.

Сейчас нам необходимо перейти к следующему примеру, проекту каталога Ex03, после его запуска на экране появляется вращающийся чайник и стрелки осей координат (рис. 9.4).

Буфер вершин заполняется данными для трех трехмерных объектов: цилиндра, конуса и чайника:

```
function TfrmD3D.InitVB : HRESULT;
const
  radius = 0.1;       // Радиус цилиндра
var
  Vertices : ^TCustomVertex;
  hRet : HRESULT;
  theta : Single;
  i : Integer;
  t : TextFile;      // Данные модели хранятся в текстовом файле
  wX, wY, wZ : Single;
```

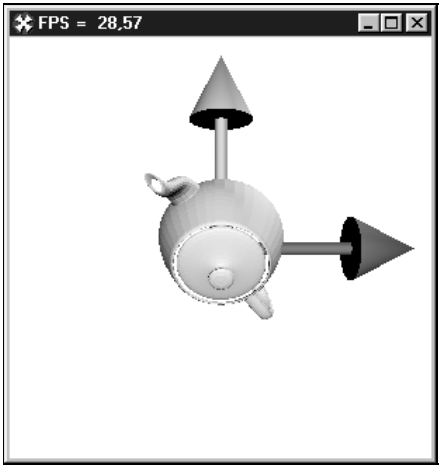


Рис. 9.4. Этот пример поможет досконально разобраться с матрицами

```
begin
  hRet := FD3DDevice.CreateVertexBuffer((100 + 51 * 2 + 6322 * 3) *
          SizeOf(TCustomVertex), 0, D3DFVF_CUSTOMVERTEX, D3DPOOL_DEFAULT,
          FD3DVB);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := FD3DDevice.SetStreamSource(0, FD3DVB, SizeOf(TCustomVertex));
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := FD3DDevice.SetVertexShader(D3DFVF_CUSTOMVERTEX);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := FD3DVB.Lock(0, (100 + 51 * 2 + 6322 * 3)*
          SizeOf(TCustomVertex), PByte(Vertices), 0);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // 100 вершин цилиндра, по часовой стрелке
  for i := 49 downto 0 do begin
    theta := 2 * Pi * i / 49;
    Vertices.X := sin(theta) * radius;
```

```
  Vertices.Y := -1;
  Vertices.Z := cos(theta) * radius;
  Vertices.nX := sin(theta);
  Vertices.nY := 0;
  Vertices.nZ := cos(theta);
  Inc(Vertices);
  Vertices.X := sin(theta) * radius;
  Vertices.Y := 1;
  Vertices.Z := cos(theta) * radius;
  Vertices.nX := sin(theta);
  Vertices.nY := 0;
  Vertices.nZ := cos(theta);
  Inc(Vertices);
end;
// Вершина конуса
Vertices.X := 0.0;
Vertices.Y := 0.0;
Vertices.Z := 1.0;
Vertices.nX := 0.0;
Vertices.nY := 0.0;
Vertices.nZ := 1.0;
Inc(Vertices);
// Треугольники, образующие конус
for i := 0 to 49 do begin
  theta := 2 * Pi * i / 49;
  Vertices.X := cos(theta);
  Vertices.Y := sin(theta);
  Vertices.Z := 0.0;
  Vertices.nX := cos(theta);
  Vertices.nY := sin(theta);
  Vertices.nZ := 1.0;
  Inc(Vertices);
end;
// Центр донышка конуса
Vertices.X := 0.0;
Vertices.Y := 0.0;
Vertices.Z := 0.0;
Vertices.nX := 0.0;
Vertices.nY := 0.0;
Vertices.nZ := -1.0;
Inc(Vertices);
// Круг, закрывающий конус
for i := 0 to 49 do begin
  theta := 2 * Pi * i / 49;
```

```
    Vertices.X := sin(theta);
    Vertices.Y := cos(theta);
    Vertices.Z := 0.0;
    Vertices.nX := 0.0;
    Vertices.nY := 0.0;
    Vertices.nZ := -1.0;
    Inc(Vertices);
  end;
  // Считываем данные модели из файла
  AssignFile (t, 'teapot.txt');
  Reset (t);
  while not EOF(t) do begin
    Readln (t, wX);           // Нормаль к треугольнику
    Readln (t, wY);
    Readln (t, wZ);
    Readln (t, Vertices.X);   // Первая вершина треугольника
    Readln (t, Vertices.Y);
    Readln (t, Vertices.Z);
    Vertices.nX := wX;
    Vertices.nY := wY;
    Vertices.nZ := wZ;
    Inc (Vertices);
    Readln (t, Vertices.X);   // Вторая вершина треугольника
    Readln (t, Vertices.Y);
    Readln (t, Vertices.Z);
    Vertices.nX := wX;
    Vertices.nY := wY;
    Vertices.nZ := wZ;
    Inc (Vertices);
    Readln (t, Vertices.X);   // Последняя вершина треугольника
    Readln (t, Vertices.Y);
    Readln (t, Vertices.Z);
    Vertices.nX := wX;
    Vertices.nY := wY;
    Vertices.nZ := wZ;
    Inc (Vertices);
  end;
  CloseFile (t);
  Result := FD3DVB.Unlock;
end;
```

Цилиндр радиуса 0.1 и высотой 2 строится вокруг оси Y, а конус единичной высоты — вокруг оси Z. О том, как получены точки модели, мы поговорим чуть позже, сейчас же я должен сообщить, что вершины треугольников модели перечисляются против часовой стрелки.

Текущие параметры матриц вида и проекций хранятся в следующих переменных:

```
FromX, FromY, FromZ : Single;
AtX, AtY, AtZ : Single;
WorldUpX, WorldUpY, WorldUpZ : Single;
fFOV, fAspect, fNearPlane, fFarPlane : Single;
```

Инициализируются эти переменные значениями, такими же, как в предыдущих примерах, лишь точка зрения отодвинута на единицу:

```
procedure TfrmD3D.FormCreate(Sender: TObject);
var
  hRet : HRESULT;
begin
  hRet := InitD3D;
  if Failed (hRet) then ErrorOut ('InitD3D', hRet);
  hRet := InitVB;
  if Failed (hRet) then ErrorOut ('InitVertex', hRet);
  // Включаем источники света и инициализируем материалы
  SetupLights;
  MaterialRed := InitMaterial(1, 0, 0, 1);
  MaterialBlue := InitMaterial(0, 0, 1, 1);
  MaterialGreen := InitMaterial(0, 1, 0, 1);
  MaterialYellow  := InitMaterial(1, 1, 0, 1);
  FromX := 0.0;       // Вектор "From"
  FromY := 0.0;
  FromZ := -6.0;
  AtX := 0.0;         // Вектор "At"
  AtY := 0.0;
  AtZ := 0.0;
  WorldUpX := 0.0;    // Вектор "WorldUp"
  WorldUpY := 1.0;
  WorldUpZ := 0.0;
  fFOV := 1.0;        // Угол обзора по оси Y
  fAspect := 1.0;     // Угол обзора по оси X
  fNearPlane := 1.0; // Передняя плоскость отсечения
  fFarPlane := 20;    // Задняя плоскость отсечения
end;
```

Для повышения красочности на сцене присутствует два источника света:

```
procedure TfrmD3D.SetupLights;
var
  Light0     : TD3DLight8;
  Light1     : TD3DLight8;
```

```
begin
  Light0 := InitDirectionalLight(D3DVector(-1, -1, -1), 1, 1, 1, 0);
  FD3DDevice.SetLight(0, Light0);
  Light1 := InitDirectionalLight(D3DVector(0, 0, 1), 1, 1, 1, 0);
  FD3DDevice.SetLight(1, Light1);
  FD3DDevice.LightEnable(0, True);
  FD3DDevice.LightEnable(1, True);
end;
```

При воспроизведении объектов сцены параметры матриц вида и проекций опираются на текущие значения управляющих переменных:

```
procedure TfrmD3D.DrawScene;
var
  matView, matProj : TD3DMatrix;
  matRotate, matTranslate : TD3DMatrix;
  matRotateX, matRotateY : TD3DMatrix;
  matScale : TD3DMatrix;
begin
  // Цилиндр по оси X
  SetRotateZMatrix(matRotate, Pi / 2);
  SetTranslateMatrix(matTranslate, 1.0, 0.0, 0.0);
  with FD3DDevice do begin
    SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
    SetTransform(D3DTS_WORLD, MatrixMul(matTranslate, matRotate));
    SetMaterial(MaterialRed);      // Красного цвета
    DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 50 * 2 — 2);
  end;
  // Конус стрелки по оси Z
  SetRotateYMatrix(matRotate, Pi / 2);
  SetTranslateMatrix(matTranslate, 2.0, 0.0, 0.0);
  SetScaleMatrix(matScale, 1.0, 0.5, 0.5);
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, MatrixMul(matScale,
                 MatrixMul(matTranslate, matRotate)));
    DrawPrimitive(D3DPT_TRIANGLEFAN, 100, 49);   // Сам конус
    DrawPrimitive(D3DPT_TRIANGLEFAN, 151, 50);   // Донышко конуса
  end;
  // Цилиндр по оси Y
  SetTranslateMatrix(matTranslate, 0.0, 1.0, 0.0);
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, matTranslate);
    SetMaterial(MaterialGreen);    // Цвет — зеленый
    DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 50 * 2 — 2);
  end;
  // Конус стрелки по оси Y
  SetRotateXMatrix(matRotate, -Pi / 2);
```

```
  SetTranslateMatrix(matTranslate, 0.0, 2.0, 0.0);
  SetScaleMatrix(matScale, 0.5, 1.0, 0.5);
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, MatrixMul(matScale,
                 MatrixMul(matTranslate, matRotate)));
    DrawPrimitive(D3DPT_TRIANGLEFAN, 100, 49);
    DrawPrimitive(D3DPT_TRIANGLEFAN, 151, 50);
  end;
  // Цилиндр по оси Z
  SetRotateXMatrix(matRotate, Pi / 2);
  SetTranslateMatrix(matTranslate, 0.0, 0.0, 1.0);
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, MatrixMul(matTranslate, matRotate));
    SetMaterial(MaterialBlue);     // Синего цвета
    DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 50 * 2 — 2);
  end;
  // Конус стрелки по оси Z
  SetTranslateMatrix(matTranslate, 0.0, 0.0, 2.0);
  SetScaleMatrix(matScale, 0.5, 0.5, 1.0);
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, MatrixMul(matScale, matTranslate));
    DrawPrimitive(D3DPT_TRIANGLEFAN, 100, 49);
    DrawPrimitive(D3DPT_TRIANGLEFAN, 151, 50);
  end;
  // Чайник, вращающийся вокруг осей X и Y
  SetRotateXMatrix(matRotateX, Angle);
  SetRotateYMatrix(matRotateY, Angle);
  SetTranslateMatrix(matTranslate, 0.0, -1.5, 0.0);
  SetScaleMatrix(matScale, 0.5, 0.5, 0.5); // Уменьшаем в два раза
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, MatrixMul(matRotateX,
                 MatrixMul(matRotateY,
                 MatrixMul(matScale, matTranslate))));
    SetMaterial(MaterialYellow);
    // Вершины модели перечисляются против часовой стрелки
    SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
    DrawPrimitive(D3DPT_TRIANGLELIST, 100 + 51 * 2, 6322);
  end;
  // Матрица вида
  SetViewMatrix(matView, D3DVector(FromX, FromY, FromZ),
                         D3DVector(AtX, AtY, AtZ),
                         D3DVector(WorldUpX, WorldUpY, WorldUpZ));
  FD3DDevice.SetTransform(D3DTS_VIEW, matView);
  // Матрица проекций
  SetProjectionMatrix(matProj, fFOV, fAspect, fNearPlane, fFarPlane);
  FD3DDevice.SetTransform(D3DTS_PROJECTION, matProj);
end;
```

Поначалу, наверняка, вам будет тяжело разбирать последовательности манипуляций с матрицами при воспроизведении нескольких объектов. Для приобретения опыта попробуйте решить простейшие задачи, например удлините цилиндры и конусы осей.

Но главное предназначение этого примера — разрешить все возможные вопросы об установках матриц вида и проекций. По нажатии клавиши <Пробел> появляется вспомогательное окно, в полях редактирования которого выводится текущее значение управляющих переменных:

```
procedure TfrmD3D.FormKeyDown(Sender: TObject; var Key: Word;
                              Shift: TShiftState);
begin
  if Key = VK_ESCAPE then Close else
  if Key = VK_SPACE then with Form2 do begin
    edtFromX.Text := FloatToStr (FromX);
    edtFromY.Text := FloatToStr (FromY);
    edtFromZ.Text := FloatToStr (FromZ);
    edtAtX.Text := FloatToStr (AtX);
    edtAtY.Text := FloatToStr (AtY);
    edtAtZ.Text := FloatToStr (AtZ);
    edtWorldUpX.Text := FloatToStr (WorldUpX);
    edtWorldUpY.Text := FloatToStr (WorldUpY);
    edtWorldUpZ.Text := FloatToStr (WorldUpZ);
    edtFOV.Text := FloatToStr (fFOV);
    edtAspect.Text := FloatToStr (fAspect);
    edtNearPlane.Text := FloatToStr (fNearPlane);
    edtFarPlane.Text := FloatToStr (fFarPlane);
    Show;
  end;
end;
```

Устанавливая в полях другие значения, можно посмотреть, какова зависимость картинки от этих параметров (рис. 9.5).

Первоначально мы видим только две оси: стрелка оси Z закрыта вращающейся моделью. Меняя значения координат вектора "From", мы передвигаем точку обзора — координаты той точки в пространстве, где находится глаз наблюдателя. Вектор "At" определяет точку, находящуюся в середине сцены. Если здесь задавать отличные друг от друга значения, то наша композиция будет перемещаться по плоскости экрана, т. е. этот вектор соответствует направлению взгляда наблюдателя. Вектор "WorldUp" указывает направление и величину поворота головы. Если менять значения его составляющих, оси нашей сцены начнут "меняться местами".

Значение **FOV** задает величину производимого увеличения в радианах. Чем меньше это число, тем крупнее выглядит наша картинка. Обратите внима-

ние, что сами объекты при этом не перемещаются, мы как будто просто вращаем колесико настройки бинокля. Значение величины **Aspect** определяет степень сжатия картинки по горизонтали: чем больше это число, тем сильнее растягивается изображение. Обычно здесь передается отношение ширины окна к его высоте.

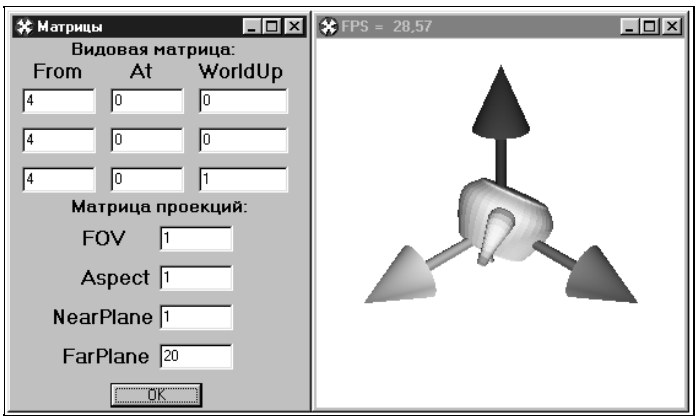


Рис. 9.5. Значения параметров матриц вида и проекций для получения привычного вида объекта в сцене

Расстояния до передней и задней плоскостей отсечения задают видимую область пространства. Расстояния отмеряются от глаза наблюдателя. Все точки, выходящие за пределы этой области, не воспроизводятся. Из соображений оптимизации плоскости сечения располагаются максимально близко друг к другу, чтобы сократить время вычислений. Обратите внимание, это очень важно: нельзя устанавливать нулевым значение расстояния до передней плоскости отсечения. Такое значение равносильно отказу от использования буфера глубины.

Надеюсь, неспешная работа с этим примером позволит вам хорошо разобраться с матрицами, определяющими вид картинки.

Подготовка моделей

Конусы и цилиндры, сфера и правильные многогранники — все подобные геометрические фигуры легко описываются и могут украсить вашу программу. Пример — проект каталога Ex04, в котором рисуется икосаэдр (рис. 9.6).

Этот многогранник описан двадцатью независимыми треугольниками. Координаты вершин и нормали я предварительно вычислил и использую в программе конкретные значения.

Знаний, которые мы уже получили, достаточно, чтобы создать более-менее интересные построения. Пример — проект каталога Ex05, где рисуется про-

стая модель человечка. Используются две геометрические формы: цилиндр и икосаэдр. С помощью клавиш перемещения курсора конечностями человечка можно управлять, заставляя его поднимать и опускать руки и ноги, но нельзя заставить поднять обе ноги одновременно (рис. 9.7).

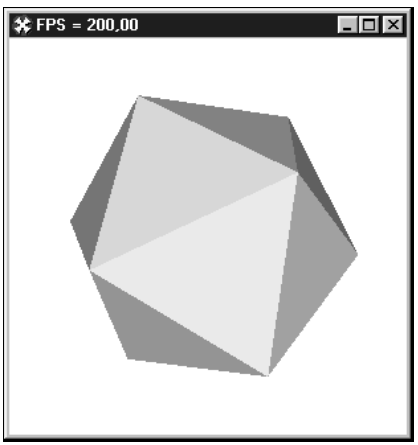


Рис. 9.6. Многогранники для демонстрационных программ всегда выглядят выигрышно

Рис. 9.7. Пример построения комплексных объектов

Для построения ног применяются цилиндры единичной длины, руки строятся цилиндрами длиной 0.75 единиц. Движения конечностей осуществляются плавно:

```
const
  INCR = 0.05; // Приращение для углов, задает темп вращения цилиндров
var
  Down : BOOL = False;   // Флаг, указывающий, нажата ли кнопка мыши
  oX : Integer;          // Используются для навигации в пространстве
  oY : Integer;
  Angle : Single = 0;
  sHeight : Single = 0;
  // Левая/правая стороны — с точки обзора зрителя
  R_hand_up_angle,   // Текущий угол поворота верхней части правой руки
  R_hand_down_angle, // Текущий угол поворота нижней части правой руки
  L_hand_up_angle,   // Углы для частей левой руки
  L_hand_down_angle,
  R_foot_up_angle,   // Углы для частей правой ноги
  R_foot_down_angle,
  L_foot_up_angle,   // Углы поворотов левой ноги
  L_foot_down_angle : Single;
```

```
  R_hand_move,       // Флаги перемещений конечностей
  L_hand_move,
  R_foot_move,
  L_foot_move : BOOL;
```

Данный пример важен тем, что учит, как строить комплексные объекты: части конечностей человечка прикреплены друг к другу. Для построения подобных частей необходимо осуществлять относительные трансформации, поэтому во вспомогательной матрице запоминаем текущую трансформацию системы координат:

```
procedure TfrmD3D.DrawScene;
var
  matRotateX, matRotateZ : TD3DMatrix;
  matScale, matTranslate : TD3DMatrix;
  matWrk : TD3DMatrix; // Вспомогательная матрица текущей трансформации
begin
  Timer;        // Пересчет текущих значений углов поворота конечностей
  // Икосаэдр головы
  SetTranslateMatrix(matTranslate, 0.0, -3.0, 0.0);
  // Масштабируем единичный многогранник
  SetScaleMatrix (matScale, 0.5, 0.5, 0.5);
  matWrk := MatrixMul(matScale, matTranslate);
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, matWrk);
    SetMaterial(MaterialYellow);     // Желтого цвета
    DrawPrimitive(D3DPT_TRIANGLELIST, 0, 20);
  end;
  // Цилиндры левой ноги
  SetTranslateMatrix(matTranslate, -0.2, 0.0, 0.0);
  SetRotateXMatrix(matRotateX, L_foot_up_angle);
  // Запоминаем положение верхней части
  matWrk := MatrixMul(matTranslate, matRotateX);
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, matWrk);
    SetMaterial(MaterialBlue);        // Ноги — синего цвета
    // Цилиндр единичной длины
    DrawPrimitive(D3DPT_TRIANGLESTRIP, 60, 98);
  end;
  // Перемещаемся к концу цилиндра единичной длины
  SetTranslateMatrix(matTranslate, 0.0, 1.0, 0.0);
  // Поворот нижней части конечности
  SetRotateXMatrix(matRotateX, L_foot_down_angle);
  // Трансформации осуществляются относительно предыдущего состояния
  // системы координат
  matWrk := MatrixMul(matWrk, MatrixMul(matTranslate, matRotateX));
```

```
with FD3DDevice do begin
  SetTransform(D3DTS_WORLD, matWrk);
  DrawPrimitive(D3DPT_TRIANGLESTRIP, 60, 98);
end;
// Правая нога
SetTranslateMatrix(matTranslate, 0.2, 0.0, 0.0);
SetRotateXMatrix(matRotateX, R_foot_up_angle);
// Запоминаем текущее положение верхней части правой ноги
matWrk := MatrixMul(matTranslate, matRotateX);
with FD3DDevice do begin
  SetTransform(D3DTS_WORLD, matWrk);
  DrawPrimitive(D3DPT_TRIANGLESTRIP, 60, 98);
end;
// Трансформации в новой системе координат
SetTranslateMatrix(matTranslate, 0.0, 1.0, 0.0);
SetRotateXMatrix(matRotateX, R_foot_down_angle);
// Поворот и сдвиг — относительно текущей трансформации
matWrk := MatrixMul(matWrk, MatrixMul(matTranslate, matRotateX));
with FD3DDevice do begin
  SetTransform(D3DTS_WORLD, matWrk);
  DrawPrimitive(D3DPT_TRIANGLESTRIP, 60, 98);
end;
// Туловище
// Цилиндр с левой стороны туловища
SetTranslateMatrix(matTranslate, -0.2, 0.0, 0.0);
SetRotateZMatrix(matRotateZ, 5 * Pi / 6);
with FD3DDevice do begin
  SetTransform(D3DTS_WORLD, MatrixMul(matTranslate, matRotateZ));
  SetMaterial(MaterialGreen);              // Текущий цвет — зеленый
  DrawPrimitive(D3DPT_TRIANGLESTRIP, 60, 98);
end;
// Цилиндр правой части туловища
SetTranslateMatrix(matTranslate, 0.2, 0.0, 0.0);
SetRotateZMatrix(matRotateZ, -5 * Pi / 6);
FD3DDevice.SetTransform(D3DTS_WORLD,
                        MatrixMul(matTranslate, matRotateZ));
FD3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 60, 98);
// Цилиндр верхней части туловища
SetTranslateMatrix(matTranslate, -1.0, -1.0, 0.0);
SetScaleMatrix (matScale, 1.0, 2.0, 1.0);  // Растягиваем цилиндр
SetRotateZMatrix(matRotateZ, Pi / 2);
FD3DDevice.SetTransform(D3DTS_WORLD, MatrixMul(matRotateZ,
                        MatrixMul(matTranslate, matScale)));
FD3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 60, 98);
```

```
  // Цилиндр нижней части туловища
  SetTranslateMatrix(matTranslate, 0.0, -0.25, 0.0);
  SetScaleMatrix (matScale, 1.0, 0.5, 1.0);  // Уменьшаем цилиндр
  SetRotateZMatrix(matRotateZ, Pi / 2);
  FD3DDevice.SetTransform(D3DTS_WORLD, MatrixMul(matRotateZ,
                          MatrixMul(matTranslate, matScale)));
  FD3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 60, 98);
  // Левая рука
  // Верхняя часть
  SetTranslateMatrix(matTranslate, -1.0, -1.0, 0.0);
  SetRotateZMatrix(matRotateZ, R_hand_up_angle);
  matWrk := MatrixMul(matTranslate, matRotateZ);
  with FD3DDevice do begin
       SetTransform(D3DTS_WORLD, matWrk);
       SetMaterial(MaterialRed);             // Текущий цвет — красный
  // Цилиндр длиной 0.75
       DrawPrimitive(D3DPT_TRIANGLESTRIP, 160, 98);
  end;
  // Сдвигаемся к концу цилиндра
  SetTranslateMatrix(matTranslate, 0.0, 0.75, 0.0);
  SetRotateZMatrix(matRotateZ, R_hand_down_angle);
  matWrk := MatrixMul(matWrk, MatrixMul(matTranslate, matRotateZ));
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, matWrk);
    DrawPrimitive(D3DPT_TRIANGLESTRIP, 160, 98);
  end;
  // Правая рука
  SetTranslateMatrix(matTranslate, 1.0, -1.0, 0.0);
  SetRotateZMatrix(matRotateZ, L_hand_up_angle);
  matWrk := MatrixMul(matTranslate, matRotateZ);
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, matWrk);
    DrawPrimitive(D3DPT_TRIANGLESTRIP, 160, 98);
  end;
  SetTranslateMatrix(matTranslate, 0.0, 0.75, 0.0);
  SetRotateZMatrix(matRotateZ, L_hand_down_angle);
  matWrk := MatrixMul(matWrk, MatrixMul(matTranslate, matRotateZ));
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, matWrk);
    DrawPrimitive(D3DPT_TRIANGLESTRIP, 160, 98);
  end;
end;
```

При нажатии клавиш управления курсором меняются значения флагов, и наблюдаем мы сцену зеркально.

```
procedure TfrmD3D.FormKeyDown(Sender: TObject; var Key: Word;
                              Shift: TShiftState);
begin
  if Key = VK_ESCAPE then Close else
    // Клавиша "влево" — правая рука
    if Key = VK_LEFT then R_hand_move := not R_hand_move else
      // Клавиша "вправо" — левая рука
      if Key = VK_RIGHT then L_hand_move := not L_hand_move else
        // Клавиша "вверх" — правая нога
        if Key = VK_UP then begin
          // Двигается, если не поднята левая нога
          if L_foot_up_angle < 1.0 then R_foot_move := not R_foot_move;
        end else
        // Клавиша "вниз" — левая нога
        if Key = VK_DOWN then begin
          // Двигается, если не поднята правая нога
          if R_foot_up_angle < 1.0 then L_foot_move := not L_foot_move;
         end;
  end;
```

При установленных флагах значения углов поворотов увеличиваются на величину `INCR`:

```
procedure TfrmD3D.Timer;
begin
  if R_hand_move then begin                  // Правая рука поднимается
    if R_hand_up_angle < Pi / 2 then begin       // Не достигнут предел
      R_hand_up_angle := R_hand_up_angle + INCR;  // Верхняя часть руки
      R_hand_down_angle := R_hand_down_angle — INCR;    // Нижняя часть
    end          // Предел достигнут, движется только нижняя часть руки
    else if (R_hand_up_angle >= Pi / 2) and (R_hand_down_angle < 0.0)
      then R_hand_down_angle := R_hand_down_angle + INCR;
    end else                    // Правая рука опускается или уже опущена
    if R_hand_up_angle > 0.0 then begin
      R_hand_up_angle := R_hand_up_angle — INCR;
      if R_hand_down_angle < 0.0
      then R_hand_down_angle := R_hand_down_angle + INCR;
  end;
  if L_hand_move then begin                   // Левая рука поднимается
    if L_hand_up_angle > -Pi / 2 then begin
      L_hand_up_angle := L_hand_up_angle — INCR;
      L_hand_down_angle := L_hand_down_angle + INCR;
    end else if (L_hand_up_angle <= Pi / 2) and (L_hand_down_angle > 0.0)
      then  L_hand_down_angle := L_hand_down_angle — INCR;
    end else
    if L_hand_up_angle < 0.0 then begin
      L_hand_up_angle := L_hand_up_angle + INCR;
```

```
      if L_hand_down_angle > 0.0
        then L_hand_down_angle := L_hand_down_angle - INCR;
    end;
    if R_foot_move then begin                // Правая нога поднимается
      if R_foot_up_angle < Pi / 2 then begin
        R_foot_up_angle := R_foot_up_angle + INCR;
        R_foot_down_angle := R_foot_down_angle - INCR;
      end else
      if (R_foot_up_angle >= Pi / 2) and (R_foot_down_angle < 0.0)
        then R_foot_down_angle := R_foot_down_angle + INCR;
    end else
    if R_foot_up_angle > 0.0 then begin
      R_foot_up_angle := R_foot_up_angle - INCR;
      if R_foot_down_angle < 0.0
        then R_foot_down_angle := R_foot_down_angle + INCR;
    end;
    if L_foot_move then begin                    // Движение левой ноги
      if L_foot_up_angle < Pi / 2 then begin
        L_foot_up_angle := L_foot_up_angle + INCR;
        L_foot_down_angle := L_foot_down_angle - INCR;
      end   else
        if (L_foot_up_angle >= Pi / 2) and (L_foot_down_angle < 0.0)
          then L_foot_down_angle := L_foot_down_angle + INCR;
    end else
      if L_foot_up_angle > 0.0 then begin
        L_foot_up_angle := L_foot_up_angle - INCR;
        if L_foot_down_angle < 0.0
          then L_foot_down_angle := L_foot_down_angle + INCR;
      end;
  end;
```

Из этого примера мы также можем вынести для себя механизм удобной навигации в пространстве. Матрица проекций задается один раз, при инициализации:

```
procedure TfrmD3D.FormCreate(Sender: TObject);
var
  hRet : HRESULT;
  matView, matProj : TD3DMatrix;
begin
  hRet := InitD3D;
  if Failed (hRet) then ErrorOut ('InitD3D', hRet);
  hRet := InitVB;
  if Failed (hRet) then ErrorOut ('InitVertex', hRet);
  SetupLights;
  MaterialRed := InitMaterial(1, 0, 0, 1);
```

```
  MaterialBlue := InitMaterial(0, 0, 1, 1);
  MaterialGreen := InitMaterial(0, 1, 0, 1);
  MaterialYellow  := InitMaterial(1, 1, 0, 1);
  // Первоначальная установка видовой матрицы
  SetViewMatrix(matView, D3DVector(2, 1, 5),
                D3DVector(0, 0, 0), D3DVector(0, -1, 0));
  FD3DDevice.SetTransform(D3DTS_VIEW, matView);
  // Матрица проекций задается один раз
  SetProjectionMatrix(matProj, 1, 1, 1, 20);
  FD3DDevice.SetTransform(D3DTS_PROJECTION, matProj);
end;
```

Положение точки зрения и параметры матриц я подобрал с помощью пройденного нами примера с осями координат.

При движении курсора мыши с нажатой кнопкой видовая матрица пересчитывается в соответствии с положением курсора:

```
procedure TfrmD3D.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Down := True;      // Кнопка нажата, флаг устанавливается
  oX := X;           // Запомнили положение курсора
  oY := Y;
end;
procedure TfrmD3D.FormMouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Down := False;     // Кнопка отпущена, флаг сбрасывается
end;
procedure TfrmD3D.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
var
  eyeX, eyeZ : Single;
  matView : TD3DMatrix;
begin
  if Down then begin  // При нажатой кнопке мыши
    // Величина перемещения курсора по горизонтали
    // задает перемещения точки обзора в пространстве по осям X и Z
    Angle := Angle + (X - oX) / 50.0;
    // Перемещение курсора по вертикали задает сдвиг по оси Y
    sHeight := sHeight + (Y - oY) / 15.0;
    eyeX := cos(Angle) * 5;
    eyeZ := sin(Angle) * 5;
    // Устанавливаем новую видовую матрицу
    SetViewMatrix(matView, D3DVector(eyeX, sHeight, eyeZ),
                  D3DVector(0, 0, 0), D3DVector(0, -1, 0));
    FD3DDevice.SetTransform(D3DTS_VIEW, matView);
```

```
    // Запомнили новое положение курсора
    oX := X;
    oY := Y;
  end;
end;
```

В качестве упражнения "обуйте" человечка в башмаки, для чего постройте параллелепипед, масштабируя куб.

Итак, с помощью цилиндров и кубиков мы можем получить занятные построения, но наверняка трудно удовлетвориться только такими фигурами. Вы уже видели в одном из предыдущих примеров модель чайника и справедливо полагаете, что она создана с использованием редактора, а опорные точки треугольников извлечены мною с помощью каких-то дополнительных средств. Конечно, для масштабных проектов требуются подобные вспомогательные средства, облегчающие процесс разработки будущих элементов сцены. Большинство трехмерных редакторов и программ моделирования объектов позволяют записывать в открытом формате или применять собственные форматы с помощью встраиваемых модулей. Так, к примеру, вы можете использовать распространенный DXF-формат, поддерживаемый большинством трехмерных редакторов, а из файла такого формата легко извлекаются вершины треугольников, образующих модель. В каталоге Ex06 располагается проект, с помощью которого я получил из файла такого формата текстовый файл, содержащий данные о нормалях и треугольниках модели чайника. При запуске приложения запрашиваются имена DXF-файла и файла-результата.

Списки, переменные типа `TList`, `Model` и `Normals` содержат данные о считанных вершинах и вычисленных нормалях:

```
// Блокировать предупреждения компилятора
// о возможно пропущенной инициализации переменных
{$WARNINGS OFF}
procedure TForm1.LoadDXF (const FileName : String);
var
  f : TextFile;
  wrkString : String;
  group, err : Integer;
  x1, x2, y1, y2, z1, z2, x3, y3, z3 : Single;
// Процедура, дополняющая список вектором
procedure AddToList (const X, Y, Z : Single);
var
  pwrkVector : ^TD3DVector;
begin
  New (pwrkVector);
  pwrkVector^ := D3DVector (X, Y, Z);
  Model.Add (pwrkVector);
end;
```

```
begin
  AssignFile(f, FileName);
  Reset(f);
  // Считываем данные из DXF-файла до секции ENTITIES
  repeat
    ReadLn(f, wrkString);
  until (wrkString = 'ENTITIES') or eof(f);
  while not eof (f) do begin
    ReadLn (f, group); // Нулевая группа содержит вершины треугольника
    ReadLn (f, wrkString);            // Идентификатор либо координата
    case group of
      0: begin
           AddToList (x3, y3, z3);       // Добавляем вершины в список
           AddToList (x2, y2, z2);
           AddToList (x1, y1, z1);
         end;
      10: val(wrkString, x1, err);
      20: val(wrkString, y1, err);
      30: val(wrkString, z1, err);
      11: val(wrkString, x2, err);
      21: val(wrkString, y2, err);
      31: val(wrkString, z2, err);
      12: val(wrkString, x3, err);
      22: val(wrkString, y3, err);
      32: val(wrkString, z3, err);
    end;
  end;
  CloseFile(f);
end;
{$WARNINGS ON}
// Процедура вычисления нормалей к треугольникам списка Model
procedure TForm1.CalcNormals;
var
  i : Integer;
  wrki, vx1, vy1, vz1, vx2, vy2, vz2 : Single;
  nx, ny, nz : Single;
  wrkVector : TD3DVector;
  pwrkVector : ^TD3DVector;
  wrkVector1, wrkVector2, wrkVector3 : TD3DVector;
  pwrkVector1, pwrkVector2, pwrkVector3 : ^TD3DVector;
begin
  for i := 0 to Model.Count div 3 - 1 do begin
    pwrkVector1 := Model [i * 3 + 1];
    wrkVector1 := pwrkVector1^;
    pwrkVector2 := Model [i * 3];
    wrkVector2 := pwrkVector2^;
```

```
    pwrkVector3 := Model [i * 3 + 2];
    wrkVector3 := pwrkVector3^;
    // Приращения по координатам
    vx1 := wrkVector1.X — wrkVector2.X;
    vy1 := wrkVector1.Y — wrkVector2.Y;
    vz1 := wrkVector1.Z — wrkVector2.Z;
    vx2 := wrkVector2.X — wrkVector3.X;
    vy2 := wrkVector2.Y — wrkVector3.Y;
    vz2 := wrkVector2.Z — wrkVector3.Z;
    // Вектор, перпендикулярный центру треугольника
    nx := vy1 * vz2 — vz1 * vy2;
    ny := vz1 * vx2 — vx1 * vz2;
    nz := vx1 * vy2 — vy1 * vx2;
    // Получаем вектор единичной длины
    wrki := sqrt (nx * nx + ny * ny + nz * nz);
    if wrki = 0 then wrki := 1;  // Для предотвращения деления на ноль
    wrkVector.X := nx / wrki;
    wrkVector.Y := ny / wrki;
    wrkVector.Z := nz / wrki;
    New (pwrkVector);
    pwrkVector^ := wrkVector;
    Normals.Add (pwrkVector);
  end;
end;
procedure TForm1.FormCreate(Sender: TObject);
var
  i : Integer;
  t : TextFile;
  p : ^TD3DVector;
  n : ^TD3DVector;
begin
  if OpenDialog1.Execute then begin
  if SaveDialog1.Execute then begin
  Model := TList.Create;
  Normals := TList.Create;
  LoadDxf (OpenDialog1.FileName);
  CalcNormals;
  Caption := 'Треугольников — ' + IntToStr(Normals.Count);
  AssignFile (t, SaveDialog1.FileName);
  ReWrite (t);
  // Запись в текстовый файл результатов
  for i := 0 to Normals.Count — 1 do begin
    n := Normals.Items [i];
    // Первым выводится вектор нормали к треугольнику
    WriteLn (t, n.X);
    WriteLn (t, n.Y);
    WriteLn (t, n.Z);
```

```
    // Координаты вершин треугольников
    p := Model.Items [i * 3];
    WriteLn (t, p.X);
    WriteLn (t, p.Y);
    WriteLn (t, p.Z);
    p := Model.Items [i * 3 + 1];
    WriteLn (t, p.X);
    WriteLn (t, p.Y);
    WriteLn (t, p.Z);
    p := Model.Items [i * 3 + 2];
    WriteLn (t, p.X);
    WriteLn (t, p.Y);
    WriteLn (t, p.Z);
  end;
  CloseFile (t);
  Model.Free;
  Normals.Free;
  end;
  end;
end;
```

В заголовке окна выводится количество считанных треугольников, ведь эта информация потребуется для дальнейшего кодирования. Результирующий файл не обязательно должен быть текстовым, вы можете закодировать данные. Также с помощью небольших манипуляций вы можете масштабировать модель, чтобы потом, при ее воспроизведении, не тратить на это драгоценное время.

В Internet существует масса ресурсов, содержащих свободно распространяемые модели. Например, на сайте **http://www.3dcafe.com** находятся сотни DXF-файлов моделей самой разнообразной тематики, и некоторыми из этих моделей я воспользовался при подготовке примеров настоящей книги. Если же нужная вам модель записана в другом формате, вы можете воспользоваться импортирующей программой.

Таких программ существует множество, я могу порекомендовать 3D Exploration, разработанную компанией X Dimension Software. Эта программа поддерживает огромный набор форматов и поэтому заслуживает вашего внимания.

С любезного разрешения авторов программы, я поместил на компакт-диск, прилагаемый к книге, демонстрационную версию продукта, которую скачал с сайта **http://www.xdsoft.com/explorer**.

Должен предупредить все ваши возможные вопросы о проблемах с конкретными DXF-файлами. Прежде всего, редактор или импортирующая программа, в которых создан файл, должны уметь разбивать поверхность модели на

отдельные треугольники (программа, которую я вам рекомендовал, делает это успешно). В таких файлах самая громоздкая секция следует после фразы ENTITIES, за которой обязан идти большой массив данных, а не короткие фразы.

Вершины треугольников должны перечисляться либо по часовой стрелке, либо против нее. Если по полученному текстовому файлу построить модель у вас не выходит, попробуйте начать разрешение вопросов с того, что отключите отсечение. Если вы уверены, что вершины всех треугольников перечисляются в одинаковом порядке, а вместо модели выводится черный контур, то проблема состоит в вычислении нормали. Обратите внимание, что в коде этого примера при вычислении нормали я поменял порядок перечисления вершин, подобрав такую последовательность, при которой нормали перпендикулярны треугольникам. Поскольку вершин только три, вы не потеряете много времени на поиски подходящего для конкретного DXF-файла порядка. И последнее, что, возможно, вам придется сделать, если вы получаете только черную тень модели, — это поменять направление нормали на противоположное:

```
wrkVector.X := – nx / wrki;
wrkVector.Y := – ny / wrki;
wrkVector.Z := – nz / wrki;
```

Мне пришлось столкнуться с тем, что многие корреспонденты обращаются с возникающими проблемами при обработке DXF-файлов, поэтому должен четко ограничить круг действительно серьезных проблем лишь двумя: либо файл не содержит разбиение модели по треугольникам, либо вершины треугольников перечисляются в различном порядке. Во всех остальных случаях решение получить нетрудно.

Достоинствами использования DXF-файлов является то, что это открытый формат, и поддерживается он практически всеми моделирующими программами. Недостатки таковы:

- модель строится по отдельным, независимым треугольникам, что приводит к избыточности данных;
- это текстовый формат, поэтому для больших моделей получаются файлы гигантских размеров.

Однако в наших примерах эти файлы нужны только при подготовке кода, непосредственно при работе приложения они не применяются, и нет необходимости распространять их вместе с приложением. Наши примеры загружают данные модели из текстового файла. Полученные текстовые файлы, конечно, тоже имеют большие размеры, но помните, что вы не обязаны использовать именно текстовые файлы. Записывайте данные в виде вещественных значений, и объем файла сразу же значительно уменьшится.

А теперь перейдем к следующему примеру, проекту каталога Ex07 — несложной заготовке увлекательной игры. Вам остается только развить про-

грамму, чтобы получить законченное произведение, со стрельбой и коварными противниками, а пока что сюжет игры совсем прост: космический корабль мчится в пространстве, наполненном сферами (рис. 9.8).

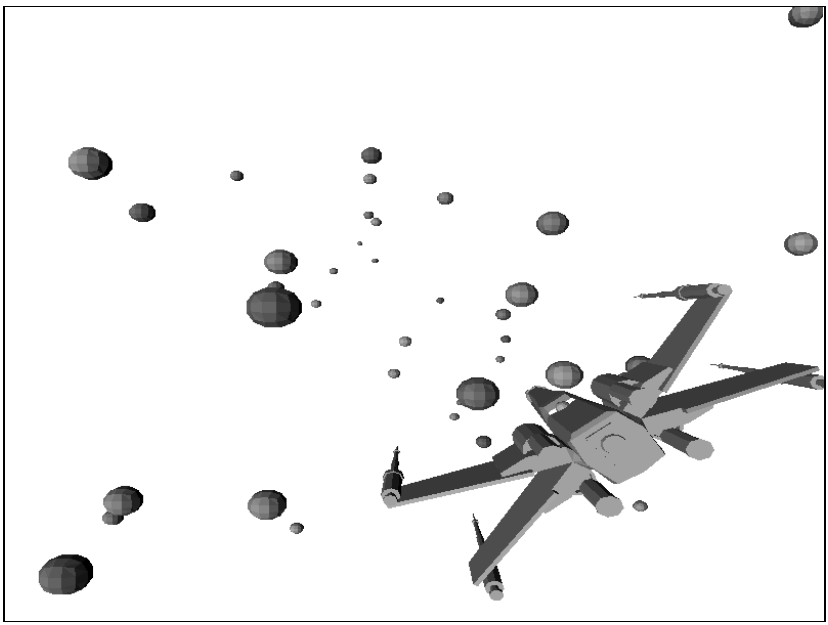


Рис. 9.8. Этот пример легко развить для получения качественной игры

Уже сейчас пример во многом напоминает профессиональный продукт: существует полноэкранный режим. Для управления положением космического корабля используется библиотека DirectInput.

Модели сферы и космического корабля загружаются из текстовых файлов, их я получил с помощью изученных утилит.

Поскольку операции с матрицами осуществляются центральным процессором, а именно нашим приложением, оптимизации их выполнения необходимо уделять максимум внимания. Для каждой движущейся сферы запоминаем текущую матрицу трансформаций, чтобы не тратить время на ее инициализацию при очередном обновлении кадра:

```
type
  TSPHERE = packed record       // Запись, относящаяся к отдельной сфере
    Z : Single;                 // Текущая координата по оси Z
    Radius : Single;            // Радиус
    MaterialSphere : TD3DMaterial8;    // Индивидуальный материал
    matSphere : TD3DMatrix;     // Текущая матрица трансформаций сферы
  end;
```

```
const
  NumSpheres = 60;              // Количество сфер
var
  Spheres : Array [0..NumSpheres — 1] of TSPHERE;    // Массив сфер
  MaterialXWing  : TD3DMaterial8;  // Материал космического корабля
  matXWing : TD3DMatrix;           // Матрица трансформаций корабля
```

Заполняются целиком матрицы трансформаций сфер и корабля один раз, при инициализации:

```
procedure TfrmD3D.FormCreate(Sender: TObject);
var
  hRet : HRESULT;
  matView, matProj : TD3DMatrix;
  i : Integer;
  matWrk : TD3DMatrix;
begin
  ShowCursor (False);
  hRet := OnCreateDevice;     // Инициализация библиотеки DirectInput
  if Failed (hRet) then ErrorOut ('InitDirectInput', hRet);
  hRet := InitD3D;
  if Failed (hRet) then ErrorOut ('InitD3D', hRet);
  hRet := InitVB;
  if Failed (hRet) then ErrorOut ('InitVertex', hRet);
  SetupLights;
  // Инициализация массива сфер
  for i := 0 to NumSpheres — 1 do
    with Spheres [i] do begin
      // Положение по оси Z, расстояние до космического корабля
      Z := random * 80 — 40;
      Radius := random * 0.1 + 0.1;             // Размер сферы
      SetScaleMatrix(matSphere, Radius, Radius, Radius);
      // Вспомогательная матрица трансформаций
      SetTranslateMatrix (matWrk, random * 20 — 10, random * 20 — 10, Z);
      // Окончательная инициализация матрицы трансформаций сферы
      matSphere := MatrixMul (matSphere, matWrk);
      // Инициализация материала сферы
      MaterialSphere := InitMaterial(random * 0.5+0.5, random * 0.5+0.5,
                                     random * 0.5 + 0.5, 0);
  end;
  // Космический корабль — золотистого цвета
  MaterialXWing := InitMaterial(1.0, 1.0, 0.0, 0);
  // Поворот модели по оси X
  SetRotateXMatrix(matXWing, -Pi / 2);
  // Видовая матрица и матрица проекций устанавливается один раз
  SetViewMatrix(matView, D3DVector(0, 0, -5), D3DVector(0, 0, 0),
                D3DVector(0, 1, 0));
```

```
  FD3DDevice.SetTransform(D3DTS_VIEW, matView);
  SetProjectionMatrix(matProj, 1, 1, 1, 100);
  FD3DDevice.SetTransform(D3DTS_PROJECTION, matProj);
end;
```

При движении сферы в пространстве и при генерации нового ее положения не обращаемся к операциям перемножения матриц, а изменяем значение только элементов четвертой строки матрицы трансформации сферы:

```
procedure TfrmD3D.DrawScene;
var
  i : Integer;
begin
  // Рисуем космический корабль
  with FD3DDevice do begin
    SetMaterial(MaterialXWing);     // Устанавливаем материал
    // Матрица трансформаций рассчитана раньше
    SetTransform(D3DTS_WORLD, matXWing);
    // Модель корабля нарисована по часовой стрелке
    SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
    // Вывод треугольников модели
    DrawPrimitive(D3DPT_TRIANGLELIST, 0, 2498);
    // Сфера нарисована против часовой стрелки
    SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
  end;
  // Вывод массива сфер
  for i := 0 to NumSpheres - 1 do begin
    with FD3DDevice do begin
      SetMaterial(Spheres [i].MaterialSphere);
      SetTransform(D3DTS_WORLD, Spheres [i].matSphere);
      DrawPrimitive(D3DPT_TRIANGLELIST, 7494, 110);
    end;
    with Spheres [i] do begin
      // Движение сферы в пространстве
      Z := Z - 0.3;
      // Не перемножаем матрицы, меняем значение только одного элемента
      Spheres [i].matSphere._43 := Z;
      // Сфера улетела за пределы экрана
      if Z < -20 then begin
        // Генерируем новое значение координаты X сферы
        matSphere._41 := random * 20 - 10;
        // Генерируем новое значение координаты Y сферы
        matSphere._42 := random * 20 - 10;
        Z := 50 + random (10);      // Новое значение координаты Z
        matSphere._43 := Z;
```

```
        // Генерируем новый материал сферы
        MaterialSphere := InitMaterial(random * 0.5 + 0.5,
                          random * 0.5 + 0.5, random * 0.5 + 0.5, 0);
      end;
    end;
  end;
end;
```

Для максимального ускорения работы приложения из кода удалены все проверки успешности выполнения операций для каждого обновления кадра. Обратите внимание, что на сцене присутствует три источника света для освещения корабля с различных сторон.

При перемещении курсора мыши меняются значения элементов матрицы трансформаций космического корабля, при нажатых кнопках мыши происходит поворот корабля вокруг оси Y:

```
function TfrmD3D.ReadImmediateData : HRESULT;
var
  hRet : HRESULT;
  dims2 : TDIMOUSESTATE2;
  matRotateY : TD3DMatrix;
begin
  ZeroMemory(@dims2, SizeOf(dims2));
  hRet := DIMouse.GetDeviceState(SizeOf(TDIMOUSESTATE2), @dims2);
  if Failed (hRet) then begin
    hRet := DIMouse.Acquire;
    while hRet = DIERR_INPUTLOST do hRet := DIMouse.Acquire;
  end;
  // Нажата левая кнопка мыши, вращение корабля
  if dims2.rgbButtons[0] = 128 then begin
    SetRotateYMatrix(matRotateY, 0.1);
    matXWing := MatrixMul (matXWing, matRotateY);
  end;
  // Правая кнопка мыши, вращение в противоположную сторону
  if dims2.rgbButtons[1] = 128 then begin
    SetRotateYMatrix(matRotateY, -0.1);
    matXWing := MatrixMul (matXWing, matRotateY);
  end;
  // Движение курсора мыши, перемещение корабля по осям X и Y
  matXWing._41:= matXWing._41 + 0.01 * dims2.lX;
  matXWing._42 := matXWing._42 – 0.01 * dims2.lY;
  Result := DI_OK;
end;
```

Одной из классических задач компьютерной графики является генерация ландшафтов, следующий наш пример, проект из каталога Ex08, является

иллюстрацией именно на эту тему. Здесь на фоне горного ландшафта летит пассажирский лайнер, терпящий, по-видимому, катастрофу, поскольку летит с выпущенными шасси и вращается вокруг своей оси (рис. 9.9).

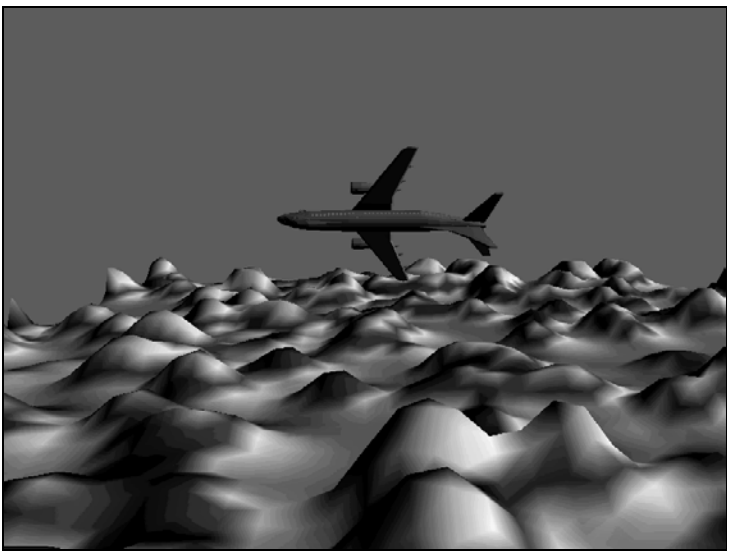


Рис. 9.9. Работа простого примера на тему создание ландшафта

Формат вершин включает в себя координату, вектор нормали и цвет, порядок их следования строго определен. Тройку чисел нормали я объединил в вектор только из соображений оптимизации:

```
type
  TCUSTOMVERTEX = packed record
    X, Y, Z : Single;
    normVector : TD3DVector;    // Нормаль должна предшествовать цвету
    Color : DWORD;
  end;
const
  D3DFVF_CUSTOMVERTEX = D3DFVF_XYZ or D3DFVF_NORMAL or D3DFVF_DIFFUSE;
type
  LandParam = packed record     // Описание опорных точек сетки ландшафта
    Color : DWORD;              // Цвет точки
    h : Single;                 // Высота
    VecNormal : TD3DVector;     // Нормаль к вершине
  end;
const
  RandPoint = 400;   // Количество холмов и гор ландшафта
  FlatLand = 3;      // Степень сглаживания возвышенностей
```

```
  Numx = 77;          // Размер ландшафта по оси X
  NumZ = 60;          // Количество точек по оси Z
  Step = 0.2;         // Масштабный множитель для одной площадки
var
  matAirplan : TD3DMatrix;      // Матрица трансформаций для самолета
  Land : array [1..NumX,1..NumZ] of LandParam;    // Массив ландшафта
```

Для генерации ландшафта произвольные вершины равномерной сетки приподнимаются на произвольную высоту. Затем, в несколько проходов, значения высот всех узлов сетки усредняются. Таким образом, вокруг пиков образуются плавные возвышенности. Для имитации горного пейзажа цвет вершин задается в зависимости от ее высоты.

При генерации ландшафта используется функция, вычисляющая нормаль к треугольнику так же, как в одном из предыдущих примеров:

```
procedure GenLand;
var
  i, j, k : Integer;
  x, z : Integer;
begin
  // Генерируем вершины возвышенностей
  for i := 1 to RandPoint do begin
    x := random(NumX - 3) + 1;
    z := random(NumZ - 3) + 1;
    Land[x,z].h := random(500);
  end;
  // Усредняем высоты соседних точек, чтобы получить плавные холмы
  for k := 1 to FlatLand do
    for i:= 2 to NumX do
      for j := 2 to NumZ do
        Land[i,j].h := (Land[i,j].h +
                        Land[(i + 1) mod NumX,j].h +
                        Land[i - 1, j].h +
                        Land[i,(j + 1) mod NumZ].h +
                        Land[i, j - 1].h) / 5;
  // Приводим данные к удобному виду, задаем цвет вершин
  for i := 1 to NumX do
    for j := 1 to NumZ do
      with Land[i,j] do begin
        h := h / 100;
        if h > 0.85 then h := 0.85;
          if h > 0.4    // Высокие вершины окрашиваем белым цветом
            then Land[i,j].Color := $00FFFFFF else
          if h > 0.2    // Точки чуть ниже - коричневым
            then Land[i,j].Color := $00804000 else
```

```
          if h > 0.1    // Вершины еще ниже — желтые
            then Land[i,j].Color := $00FFFF00
            // Точки на равнине — зеленые
          else Land[i,j].Color := $0000FF00;
      end;
  // Рассчитываем нормали к вершинам
  for i := 1 to NumX — 1 do
     for j := 1 to NumZ do
     CalcNormals (D3DVector (i * Step, Land[i, j — 1].h, (j — 1) * Step),
                  D3DVector (i * Step, Land[i, j].h, j * Step),
                  D3DVector ((i + 1) * Step, Land[i + 1, j — 1].h,
                             (j — 1) * Step), Land[i, j].VecNormal);
end;
```

Данные модели считываются из текстового файла. В буфере вершин первые четыре вершины отводятся для построения отдельного квадрата ландшафта:

```
function TfrmD3D.InitVB : HRESULT;
var
  Vertices : ^TCustomVertex;
  hRet : HRESULT;
  t : TextFile;
  wrkVec : TD3DVector;
begin
  FD3DDevice.CreateVertexBuffer(20665 * SizeOf(TCustomVertex), 0,
                                      D3DFVF_CUSTOMVERTEX,
                                      D3DPOOL_DEFAULT, FD3DVB);
  FD3DDevice.SetStreamSource(0, FD3DVB, SizeOf(TCustomVertex));
  FD3DVB.Lock(0, 20665 * SizeOf(TCustomVertex), PByte(Vertices), 0);

  Inc (Vertices);  // Первые четыре вершины отводятся для построения
  Inc (Vertices);  // отдельного квадрата ландшафта
  Inc (Vertices);
  Inc (Vertices);
  AssignFile (t, 'Boeing.txt');
  Reset (t);
  while not EOF(t) do begin
    Readln (t, wrkVec.X); // Считываем вектор нормали
    Readln (t, wrkVec.Y);
    Readln (t, wrkVec.Z);
    // Считываем вершины очередного треугольника
    Readln (t, Vertices.X);
    Readln (t, Vertices.Y);
    Readln (t, Vertices.Z);
    // Исходные данные модели масштабируются
    Vertices.X := Vertices.X / 3;
    Vertices.Y := Vertices.Y / 3;
```

```
    Vertices.Z := Vertices.Z / 3;
    Vertices.normVector := wrkVec;
    Vertices.Color := $00808080; // Цвет — серебристый
    Inc (Vertices);
    Readln (t, Vertices.X);
    Readln (t, Vertices.Y);
    Readln (t, Vertices.Z);
    Vertices.X := Vertices.X / 3;
    Vertices.Y := Vertices.Y / 3;
    Vertices.Z := Vertices.Z / 3;
    Vertices.normVector := wrkVec;
    Vertices.Color := $00808080;
    Inc (Vertices);
    Readln (t, Vertices.X);
    Readln (t, Vertices.Y);
    Readln (t, Vertices.Z);
    Vertices.X := Vertices.X / 3;
    Vertices.Y := Vertices.Y / 3;
    Vertices.Z := Vertices.Z / 3;
    Vertices.normVector := wrkVec;
    Vertices.Color := $00808080;
    Inc (Vertices);
  end;
  CloseFile (t);
  FD3DVB.Unlock;
  Result := FD3DDevice.SetVertexShader(D3DFVF_CUSTOMVERTEX);
end;
```

После считывания данных модели поворачиваем ее вокруг собственных осей:

```
procedure TfrmD3D.FormCreate(Sender: TObject);
var
  hRet : HRESULT;
  matView, matProj : TD3DMatrix;
  matWrk1, matWrk2 : TD3DMatrix;
begin
  Randomize;            // Ландшафт генерируется каждый раз по-новому
  ShowCursor (False);   // Устанавливаем полноэкранный режим
  hRet := InitD3D;
  if Failed (hRet) then ErrorOut ('InitD3D', hRet);
  hRet := InitVB;
  if Failed (hRet) then ErrorOut ('InitVertex', hRet);
  SetupLights;
  // Поворачиваем самолет
  SetRotateXMatrix(matWrk1, Pi / 2);
```

```
  SetRotateZMatrix(matWrk2, Pi);
  SetTranslateMatrix (matAirplan, 7.0, 2.0, 5.0);
  // Первоначальная матрица трансформаций для самолета
  matAirplan := MatrixMul (matAirplan, MatrixMul (matWrk2, matWrk1));
  GenLand;              // Генерируем ландшафт пейзажа
  SetViewMatrix(matView, D3DVector(16, 2.5, 5),
                D3DVector(0, 0, 5), D3DVector(0, 1, 0));
  FD3DDevice.SetTransform(D3DTS_VIEW, matView);
  SetProjectionMatrix(matProj, 1, 1, 1, 15);
  FD3DDevice.SetTransform(D3DTS_PROJECTION, matProj);
end;
```

Ландшафт рисуется на основе данных массива, по отдельным квадратикам:

```
procedure TfrmD3D.DrawArea(const x, y : Integer);
var
  Vertices : ^TCustomVertex;
begin
  FD3DVB.Lock(0, 4 * SizeOf(TCustomVertex), PByte(Vertices), 0);
  Vertices.X := x * Step;
  Vertices.Y := Land[x, y - 1].h;
  Vertices.Z := (y - 1) * Step;
  Vertices.normVector := Land[x, y - 1].VecNormal;
  Vertices.Color := Land[x, y - 1].Color;
  Inc (Vertices);
  Vertices.X := x * Step;
  Vertices.Y := Land[x, y].h;
  Vertices.Z := y * Step;
  Vertices.normVector := Land[x, y].VecNormal;
  Vertices.Color := Land[x, y].Color;
  Inc (Vertices);
  Vertices.X := (x + 1) * Step;
  Vertices.Y := Land[x + 1, y - 1].h;
  Vertices.Z := (y - 1) * Step;
  Vertices.normVector := Land[x + 1, y - 1].VecNormal;
  Vertices.Color := Land[x + 1, y - 1].Color;
  Inc (Vertices);
  Vertices.X := (x + 1) * Step;
  Vertices.Y := Land[x + 1, y].h;
  Vertices.Z := y * Step;
  Vertices.normVector := Land[x + 1, y].VecNormal;
  Vertices.Color := Land[x + 1, y].Color;
  FD3DVB.Unlock;
  FD3DDevice.DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
end;
```

```
function TfrmD3D.Render : HRESULT;
var
  hRet : HRESULT;
  i, j :Integer;
begin
  // Экран окрашивается голубоватым цветом
  FD3DDevice.Clear(0, nil, D3DCLEAR_TARGET or D3DCLEAR_ZBUFFER,
                   $00000FFF, 1.0, 0);
  FD3DDevice.BeginScene;
  with FD3DDevice do begin
    SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
    // Треугольники ландшафта перечисляются по часовой стрелке
    SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
    // Вершины ландшафта сгенерированы в мировой системе координат
    SetTransform(D3DTS_WORLD, IdentityMatrix);
  end;
  // Выводим квадратики ландшафта
  for j := 2 to NumZ - 1 do
    for i := 1 to NumX - 5 do DrawArea(i,j);
  with FD3DDevice do begin
    // Устанавливается матрица трансформаций самолета
    SetTransform(D3DTS_WORLD, matAirplan);
    // Вершины модели перечисляются против часовой стрелки
    SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
    // Данные располагаются, начиная с четвертой вершины
    DrawPrimitive(D3DPT_TRIANGLELIST, 4, 20661 div 3);
  end;
  FD3DDevice.EndScene;
  Result := FD3DDevice.Present(nil, nil, 0, nil);
end;
```

После того, как нарисован самолет, текущей трансформацией мировой матрицы остается матрица трансформаций нашей модели, поэтому перед рисованием ландшафта задаем здесь матрицу идентичности.

При каждой перерисовке кадра ландшафт циклически сдвигается, а самолет поворачивается вокруг своей оси на небольшой угол:

```
procedure MoveLand;                    // Циклическое движение пейзажа
var
  i, j : Integer;
  TempLand : array [1..NumX] of LandParam;   // Вспомогательный массив
begin
  // Запомнили строку массива ландшафта
  for i := 1 to NumX do TempLand[i] := Land[i,NumZ];
  // Сдвигаем ландшафт
  for j := NumZ downto 2 do
```

```
    for i := 1 to NumX do Land[i,j] := Land[i,j-1];
    // Круговое появление последней строки массива
  for i := 1 to NumX do Land[i,1] := TempLand[i];
end;
procedure TfrmD3D.ApplicationEvents1Idle(Sender: TObject;
  var Done: Boolean);
var
  matWrk : TD3DMatrix;
begin
  if FActive then begin
    Render;               // Нарисовали кадр
    MoveLand;             // Передвинули ландшафт
    SetRotateYMatrix(matWrk, 0.1); // Матрица для небольшого поворота
    matAirplan := MatrixMul (matAirplan, matWrk); // Поворот самолета
  end;
  Done := False;
end;
```

Для оптимизации в коде программы я матрицу поворота вычисляю один раз.

Обратите внимание на то, что в программе используется два направленных источника света, и, самое главное, на то, что формат вершин с указанием нормали и цвета позволяет воспроизводить объекты без дополнительных ухищрений. В самом деле, в программе отсутствуют материалы, и такой способ окрашивания примитивов является самым простым и быстрым.

Однако в этом примере мы сильно перерасходуем память, ведь две тысячи треугольников модели окрашиваются одним цветом, а для каждой вершины модели мы вынуждены задавать цвет. При использовании же материала память сильно экономится, но мы не получим тогда сглаживание цветов для треугольников ландшафта.

Пример проекта каталога Ex09 подсказывает возможное решение. Здесь на фоне того же ландшафта, что и в предыдущем примере, гордо парит орел (рис. 9.10).

Вкратце суть решения можно передать следующими словами: в программе применяются два буфера вершин различных форматов, один из которых не имеет цветового компонента:

```
type
  TCUSTOMVERTEXLand = packed record
    X, Y, Z : Single;
    normVector : TD3DVector;
    Color : DWORD;
  end;
  TCUSTOMVERTEXEagle = packed record
    X, Y, Z : Single;
```

```
    normVector : TD3DVector;
  end;
const
  D3DFVF_CUSTOMVERTEXLand =  D3DFVF_XYZ or D3DFVF_NORMAL or
                             D3DFVF_DIFFUSE;
  D3DFVF_CUSTOMVERTEXEagle = D3DFVF_XYZ or D3DFVF_NORMAL;
```

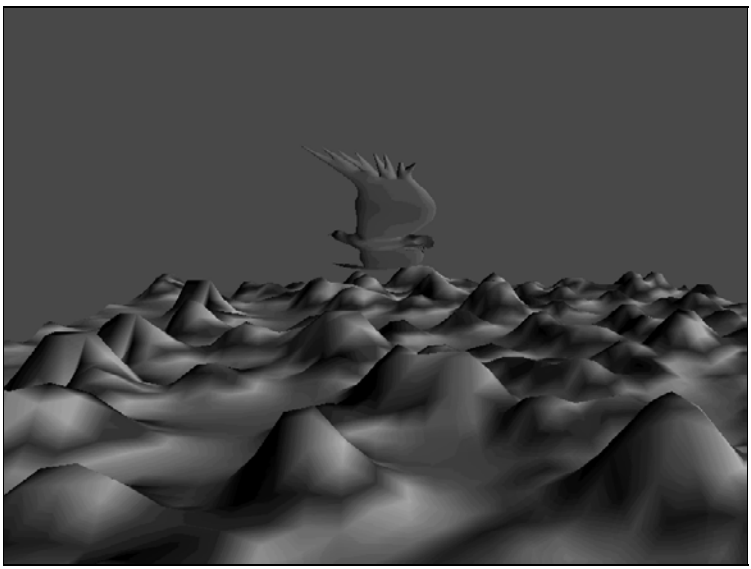


Рис. 9.10. Пример сбалансированного подхода к окрашиванию примитивов

При воспроизведении переключаем потоки источников, задавая в качестве таковых буферы, содержащие вершины различных форматов. При подобном подходе существенно экономится память.

Особое внимание мы должны обратить на то, как в этом примере заполняется буфер вершин модели. Рекомендованная мною импортирующая программа в качестве одного из форматов позволяет использовать код на языке C++. Для подготовки этого примера я результирующий файл преобразовал в код на языке Pascal. Это совершенно не сложно, поскольку большая его часть представляет собой массивы данных. Только имя массива, содержащего данные вершин, пришлось изменить на `Avertices`, чтобы не появилось конфликтов с переменной `Vertices`.

Первые 13 строк такого файла необходимо удалить. Также удаляются последние строки кода, начиная со строки `GLint Gen3DObjectList()`. В оставшемся файле убираются все символы `f`, предшествующие запятой и фигурной скобке. Далее все фигурные скобки заменяются на обычные.

Последнее, что необходимо сделать — изменить синтаксис описания массивов. Например, такая строка

```
static GLint face_indicies[1200][9]
```

заменяется следующей:

```
face_indicies : array [0..1199, 0..8] of integer
```

Тип `GLfloat` заменяется типом `Single`, остальные типы соответствуют целому.

Полученный файл с директивой `Include` подключается к головному модулю проекта (в секцию `const`), а код функции инициализации буфера становится практически универсальным, в зависимости от модели меняется только число, задающее размер буфера. Впрочем, и это число можно заменить выражением, опирающемся на размер массива `normals`. Также, возможно, потребуется исправить и масштабный множитель:

```
function TfrmD3D.InitVBEagle : HRESULT;
var
  Vertices : ^TCustomVertexEagle;
  hRet : HRESULT;
  i, j : Integer;
  vi : Integer;     // Индекс вершин треугольников
  ni : Integer;     // Индекс нормалей треугольников
begin
  hRet := FD3DDevice.CreateVertexBuffer(10500 *
          SizeOf(TCustomVertexEagle), 0, D3DFVF_CUSTOMVERTEXEagle,
          D3DPOOL_DEFAULT, FD3DVBEagle);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := FD3DVBEagle.Lock(0, 10500 * SizeOf(TCustomVertexEagle),
                           PByte(Vertices), 0);
  if Failed(hRet) then begin
    Result := hRet;
    Exit;
  end;
  // Цикл заполнения буфера данными из массивов
  for i := 0 to sizeof(face_indicies) div sizeof(face_indicies[0]) - 1 do
    for j := 0 to 2 do begin
      vi := face_indicies[i][j];      // Индекс фасета
      ni := face_indicies[i][j+3];    // Индекс нормали фасета
      // Исходные данные масштабируем, умножая на 5
      Vertices.X := Avertices[vi][0] * 5;
      Vertices.Y := Avertices[vi][1] * 5;
      Vertices.Z := Avertices[vi][2] * 5;
```

```
      Vertices.normVector.X := normals[ni][0];
      Vertices.normVector.Y := normals[ni][1];
      Vertices.normVector.Z := normals[ni][2];
      Inc(Vertices);
    end;
  Result := FD3DVBEagle.Unlock;
end;
```

При инициализации работы один раз устанавливается материал, а при воспроизведении необходимо указывать, окрашивание производится исходя из цветовой составляющей вершины, либо используется установленный материал:

```
with FD3DDevice do begin
  SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
  SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
  // Для ландшафта цвет примитивов задается цветовой составляющей вершин
  SetRenderState(D3DRS_DIFFUSEMATERIALSOURCE, D3DMCS_COLOR1);
  SetTransform(D3DTS_WORLD, IdentityMatrix);
  // Выключаем третий источник,
  // предназначенный для освещения только модели
  LightEnable(2, False);
  SetStreamSource(0, FD3DVBLand, SizeOf(TCustomVertexLand));
  SetVertexShader(D3DFVF_CUSTOMVERTEXLand);
end;
// Вывод треугольников ландшафта
for j := 2 to NumZ - 1 do
  for i := 1 to NumX - 5 do DrawArea(i, j);
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, matEagle);
    LightEnable(2, True);    // Включаем дополнительный источник
    SetStreamSource(0, FD3DVBEagle, SizeOf(TCustomVertexEagle));
    SetVertexShader(D3DFVF_CUSTOMVERTEXEagle);
    // Окрашивание осуществляется исходя из свойств материала
    SetRenderState(D3DRS_DIFFUSEMATERIALSOURCE, D3DMCS_MATERIAL);
    DrawPrimitive(D3DPT_TRIANGLELIST, 0, 10500 div 3);
  end;
```

По умолчанию для режима `D3DRS_DIFFUSEMATERIALSOURCE` устанавливается значение `D3DMCS_COLOR1`. Здесь же мы восстанавливаем это значение, потерянное после воспроизведения модели орла.

Закончу главу небольшими замечаниями по поводу моделей. Конечно, совсем не обязательно, чтобы используемые вами модели были однотонными, как в моих примерах. Импортирующая программа, рекомендованная мной, позволяет записывать в DXF-файлах (или в другом формате) отдельные части моделей. Вы можете разбить модель на части, считывать данные на них

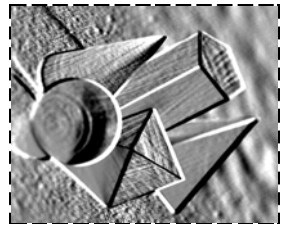
по отдельности и окрашивать фрагменты в различные цвета, меняя текущий материал, или задавать нужный цвет вершин.

Если данные модели заполняются так же, как в последнем примере, в виде массивов констант, и без расчета нормалей, то массивы могут храниться в отдельных файлах внутреннего формата или загружаться из библиотек. В этом случае размер главного модуля станет меньше. Также мне необходимо уточнить, что модель строится группой несвязанных треугольников.

Что вы узнали в этой главе

Глава посвятила нас в премудрости матричных операций, что позволило нам перенести построения в пространство. Мы узнали, как с помощью несложных средств можно создавать составные объекты. Хотя примеры главы крайне просты, усердные читатели смогут легко развить их до совершенных и серьезных программ.

ГЛАВА 10



Визуальные эффекты

Последняя глава в основном посвящена рассмотрению вопросов повышения реалистичности создаваемых построений.

Примеры располагаются в каталоге \Examples\Chapter10.

Источник света и свойства материала

Изучив предыдущие примеры, вы получили представление о направленном источнике света и материале объектов. Теперь нам предстоит разобраться с этими вещами основательнее.

Направленный источник располагается в бесконечности. Вектор, задаваемый при его инициализации, определяет направление потока испускаемых лучей. Лучи света параллельны. Интенсивность источника постоянна для каждой точки пространства. Данный источник света можно считать моделью солнечного освещения.

При такой модели освещения если для всех вершин квадрата задать одну и ту же нормаль, то при любом его положении все точки имеют один и тот же цвет. Цвет этот определяется комбинацией цвета материала и источника света. Если квадрат материала желтого цвета освещать белым светом, результат будет точно таким же, как и при освещении квадрата белого материала источником света с наложенным желтым светофильтром.

Для получения действительно реалистичных изображений направленный источник не годится в принципе, например, стены комнаты будут иметь ровный оттенок. Для таких целей предусмотрен *точечный источник света*, отличающийся от направленного именно тем, что при его использовании учитывается реальное положение источника в пространстве. Точечный источник света похож на лампочку или свечу, лучи света испускаются из какой-то точки во всех направлениях.

Помимо положения, параметрами такого источника являются его интенсивность и ослабление. *Интенсивность точечного источника* — это его изначальная яркость, мощность. Явно она не задается, ее определяют значения цветовых составляющих поля `Diffuse`. *Ослабление* складывается из нескольких составляющих: область действия источника и коэффициенты, задающие закон ослабления освещенности. Область действия определяется линейной характеристикой, расстоянием. Все точки, расположенные от источника дальше этого расстояния, никак им не освещаются. Коэффициенты закона ослабления (их три) задают, как падает освещенность в пространстве. Первый коэффициент соответствует неизменному, постоянному освещению. Если установить такое правило, то, независимо от расстояния до источника света, все точки, попадающие в область освещения, освещаются одинаково. Второй коэффициент соответствует линейному затуханию. По мере удаления от источника света интенсивность освещения падает по линейному закону так, что на границе области его интенсивность становится нулевой. Последний коэффициент определяет квадратичное убывание интенсивности, степень падения освещенности — квадрат расстояния.

Коэффициенты задаются вещественными, обычно их значения нулевые или единичные. Самой распространенной схемой является линейный закон убывания, но вы можете строить и собственный, сложный закон освещенности, а не использовать определенную схему (если задать единичными все три коэффициента, интенсивность падает по полиномиальному закону).

Давайте закрепим пройденное, познакомившись с проектом каталога Ex01, в котором на экране рисуется тор. Во внутренней области тора перемещается точечный источник света, в точке его текущего положения рисуется сфера (рис. 10.1).

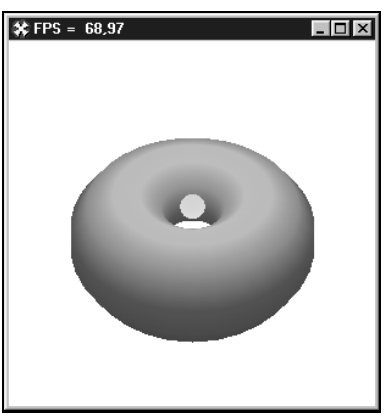


Рис. 10.1. Пример использования точечного источника света

При инициализации такого источника нам необходимо самим заполнить все поля структуры `TD3DLight8`.

```
procedure TfrmD3D.SetupLights;
var
  Material  : TD3DMaterial8;
begin
  Material := InitMaterial(1, 1, 0, 0);      // Материал желтого цвета
  FD3DDevice.SetMaterial(Material);
  ZeroMemory(@Light, SizeOf(Light));
  with Light do begin
    _Type := D3DLIGHT_POINT;                 // Тип источника — точечный
     Diffuse.R := 1.0;                       // Цвет источника
     Diffuse.G := 1.0;
     Diffuse.B := 1.0;
     Specular := Diffuse;                    // Дополнительные параметры
     Ambient := Diffuse;
     Position := D3DVector(0.0, 0.0, 0.0);  // Позиция в пространстве
     Attenuation0 := 1.0;           // Коэффициенты закона ослабления
     Attenuation1 := 1.0;
     Attenuation2 := 1.0;
     Range := 2.5;       // Расстояние, задающее область освещенности
  end;
  FD3DDevice.SetLight(0, Light);
  FD3DDevice.LightEnable(0, True);
end;
```

Первое поле записи содержит константу, задающую тип источника. Структура `Diffuse` определяет цветовой фильтр, накладываемый на источник. Позиция источника света будет устанавливаться в текущей системе координат, ее значение остается действующим до следующего вызова метода `SetLight` (не обязательно заново инициализировать все поля структуры). Чтобы сфера освещалась так, как будто источник света находится внутри нее, необходимо переместить источник света в ее систему координат:

```
procedure TfrmD3D.DrawScene;
var
  matTranslate, matScale : TD3DMatrix;
begin
  // Вычисляем текущее положение источника
  Light.Position := D3DVector(0.0, cos (Angle) * 2, 0.0);
  with FD3DDevice do begin
    // Возвращаем мировую систему координат
    SetTransform(D3DTS_WORLD, IdentityMatrix);
    // Устанавливаем источник света в новом положении
    SetLight(0, Light);
    DrawPrimitive(D3DPT_TRIANGLELIST, 0, 864); // Вывод тора
  end;
  // Источник света будет внутри сферы
  Light.Position := D3DVector(0.0, 0.0, 0.0);
```

```
  // Матрица трансформаций для сферы
  SetTranslateMatrix(matTranslate, 0.0, cos (Angle) * 2, 0.0);
  SetScaleMatrix(matScale, 0.1, 0.1, 0.1);
  with FD3DDevice do begin
    SetTransform(D3DTS_WORLD, MatrixMul(matScale, matTranslate));
    SetLight(0, Light);
    DrawPrimitive(D3DPT_TRIANGLELIST, 864 * 3, 1200);
  end;
end;
```

Позже мы подробнее поговорим о полях структуры, связанной с источником света, а сейчас попробуем построить модель комнаты, чтобы убедиться, что использование точечного источника света значительно повышает реализм изображений. В проекте каталога Ex02 рисуется комната, в ее центре находится конус, вокруг которого вращается сфера (рис. 10.2).

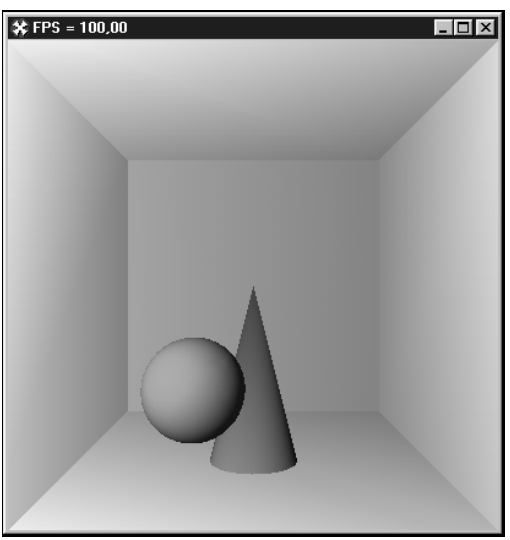


Рис. 10.2. Эту композицию сделаем тестовой для дальнейших иллюстраций

Матрицы трансформаций полностью заполняются один раз, в начале работы приложения:

```
procedure TfrmD3D.FormCreate(Sender: TObject);
var
  hRet : HRESULT;
  matView, matProj : TD3DMatrix;
  matRotate, matTranslate, matScale : TD3DMatrix;
begin
  hRet := InitD3D;
```

```
  if Failed (hRet) then ErrorOut ('InitD3D', hRet);
  hRet := InitVB;
  if Failed (hRet) then ErrorOut ('InitVertex', hRet);
  // Голубоватый материал конуса
  MaterialConus := InitMaterial(0, 0.5, 1, 0);
  // Белый материал стен комнаты
  MaterialWhite := InitMaterial(1, 1, 1, 0);
  // Светло-коричневый материал сферы
  MaterialSphere := InitMaterial(1, 0.5, 0, 0);
  // Точка зрения задается один раз
  SetViewMatrix(matView, D3DVector(0, 0, 2.577), D3DVector(0, 0, -5),
                         D3DVector(0, 1, 0));
  FD3DDevice.SetTransform(D3DTS_VIEW, matView);
  // Матрица проекций
  SetProjectionMatrix(matProj, 1, 1, 1, 10);
  FD3DDevice.SetTransform(D3DTS_PROJECTION, matProj);
  // Инициализация источников света
  SetupLights;
  // Поворот конуса вокруг оси X
  SetRotateXMatrix(matRotate, -Pi / 2);
  // Переносим конус, его вершина в центре сцены
  SetTranslateMatrix(matTranslate, 0.0, -1.0, 0.0);
  // Масштабируем конус
  SetScaleMatrix(matScale, 0.25, 1.0, 0.2);
  // Матрица трансформаций конуса вычисляется один раз
  matCone := MatrixMul(matScale, MatrixMul(matTranslate, matRotate));
  // Инициализация матрицы трансформаций сферы
  matSphere := IdentityMatrix;
  // Переносим сферу по оси Y
  matSphere._42 := -0.5;
end;
```

Я ввел в сцену четыре источника света. Три точечных источника предназначены для освещения стен комнаты, конус и сфера освещаются направленным источником света:

```
procedure TfrmD3D.SetupLights;
var
  Light0     : TD3DLight8;
  Light1     : TD3DLight8;
  Light2     : TD3DLight8;
  Light3     : TD3DLight8;
begin
  ZeroMemory(@Light0, SizeOf(Light0));
  with Light0 do begin
    _Type := D3DLIGHT_POINT;
```

```
    Diffuse.r := 0.4;    // Поскольку присутствует три источника,
    Diffuse.g := 0.4;    // их яркость задается небольшой
    Diffuse.b := 0.4;
    Specular := Diffuse;
    Ambient := Diffuse;
    Position := D3DVector(0.5, 0.75, 1.5);
    Attenuation0 := 1.0;
    Attenuation1 := 1.0;
    Attenuation2 := 0.0;
    Range := 2.56;
  end;
  ZeroMemory(@Light1, SizeOf(Light1));
  with Light1 do begin
    _Type := D3DLIGHT_POINT;
    Diffuse.r := 0.4;
    Diffuse.g := 0.4;
    Diffuse.b := 0.4;
    Specular := Diffuse;
    Ambient := Diffuse;
    Position := D3DVector(0.5, 0.3, 0.3);
    Attenuation0 := 1.0;
    Attenuation1 := 1.0;
    Attenuation2 := 0.0;
    Range := 2.5;
  end;
  ZeroMemory(@Light2, SizeOf(Light1));
  with Light2 do begin
    _Type := D3DLIGHT_POINT;
    Diffuse.r := 0.4;
    Diffuse.g := 0.4;
    Diffuse.b := 0.4;
    Specular := Diffuse;
    Ambient := Diffuse;
    Position := D3DVector(0.5, -0.3, 0.3);
    Attenuation0 := 1.0;
    Attenuation1 := 1.0;
    Attenuation2 := 0.0;
    Range := 2.5;
  end;
  // Один направленный источник света
  Light3:=InitDirectionalLight(D3DVector(-0.5, -0.5, -1),
                               1.0, 1.0, 1.0, 0);
  // Источники только инициализируются, но пока не включаются
  with FD3DDevice do begin
    SetLight(0, Light0);
```

```
    SetLight(1, Light1);
    SetLight(2, Light2);
    SetLight(3, Light3);
  end;
end;
```

При рисовании объектов включаем только определенные источники света:

```
procedure TfrmD3D.DrawScene;
begin
  // Стены комнаты — 10 независимых треугольников
  with FD3DDevice do begin
    // Матрица идентичности возвращает в мировую систему координат
    SetTransform(D3DTS_WORLD, IdentityMatrix);
    SetMaterial(MaterialWhite);  // Стены из белого материала
    LightEnable(0, True);        // Работают только точечные источники
    LightEnable(1, True);
    LightEnable(2, True);
    LightEnable(3, False);       // Направленный источник выключаем
    DrawPrimitive(D3DPT_TRIANGLELIST, 0, 10);
  end;
  // Конус и сфера освещаются только направленным источником
  with FD3DDevice do begin
    LightEnable(0, False);
    LightEnable(1, False);
    LightEnable(2, False);
    LightEnable(3, True);
    SetMaterial(MaterialConus);            // Синий материал конуса
    SetTransform(D3DTS_WORLD, matCone);    // Неизменное положение конуса
    DrawPrimitive(D3DPT_TRIANGLEFAN, 30, 49);  // Сам конус
    DrawPrimitive(D3DPT_TRIANGLEFAN, 81, 49);  // Основание конуса
  end;
  // Перемещаем сферу в новое положение
  matSphere._41 := cos (Angle) / 2;     // Меняем только два элемента
  matSphere._43 := sin (Angle) / 2;     // текущей матрицы трансформаций
  // Вывод сферы; источник света — текущий, направленный
  with FD3DDevice do begin
    // Переносим систему координат
    SetTransform(D3DTS_WORLD, matSphere);
    SetMaterial(MaterialSphere);
    DrawPrimitive(D3DPT_TRIANGLELIST, 30 + 51 + 51, 1200);
  end;
end;
```

Обратите внимание, что среди задаваемых режимов воспроизведения появилось что-то новое для нас.

```
with FD3DDevice do begin
  // Все вершины примитивов перечисляются по часовой стрелке
  SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
  SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
  SetRenderState(D3DRS_AMBIENT, $00202020);
  SetRenderState(D3DRS_LIGHTING, Dword (True));
  // Конус масштабируется, поэтому включаем пересчет нормалей
  SetRenderState(D3DRS_NORMALIZENORMALS, DWORD (True));
end;
```

Включение режима `D3DRS_AMBIENT` равносильно включению дополнительного источника света, эмулирующего окружающую среду. Свет этого рассеянного источника излучается из всех направлений. Предназначен такой источник для передачи присутствия на сцене, в данном случае, воздуха, в котором лучи света рассеиваются во всех направлениях.

Записи, определяющие источник света и материал, содержат поля `Diffuse`, `Ambient` и `Specular`. Первая структура соответствует диффузным свойствам объекта: для источника света это светофильтр, накладываемый на него; для материала это непосредственно цвет материала, та составляющая падающего света, которая не поглощается поверхностью. Это самая весомая составляющая получающегося цвета. Вторая, рассеянная составляющая проявляется в областях, примыкающих к области, на которую непосредственно падает свет. Используется она в комбинации с третьей, зеркальной составляющей для передачи таких свойств, как гладкость или матовость. Комбинируя значения этих составляющих, можно получать яркие или тусклые блики на поверхности объекта.

Разницы в том, задаются оптические свойства материала или источника, нет, но вы можете комбинировать свойства источника и материала для того, чтобы передать, что на сцене присутствуют, например, светящиеся объекты и объекты с обычными свойствами.

Проект из каталога Ex03 наглядно демонстрирует смысл атрибутов свойств материала и источника света. Это развитие примера с тором. Теперь мы можем произвольно задавать значения всех параметров (рис. 10.3).

При нажатии кнопок вызывается стандартный диалог задания цвета. Выбранный пользователем цвет устанавливается в качестве параметров источника света или материала. Обратите внимание, как подготавливается диалог:

```
procedure TfrmD3D.Button2Click(Sender: TObject);
begin
  // Предоставляем пользователю увидеть установленный диффузный цвет
  ColorDialog1.Color :=
  Round(MaterialTorus.Diffuse.R * 255) +
        Round(MaterialTorus.Diffuse.G * 255 * $100) +
        Round(MaterialTorus.Diffuse.B * 255 * $10000);
```

```
  if ColorDialog1.Execute then
    with MaterialTorus.Diffuse do begin
      R := (ColorDialog1.Color and $FF) / 255;
      G := ((ColorDialog1.Color and $FF00) shr 8) / 255;
      B := ((ColorDialog1.Color and $FF0000) shr 16) / 255;
  end;
end;
```

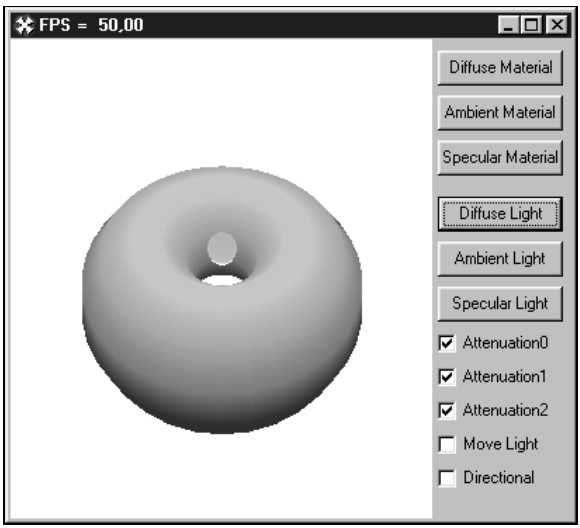


Рис. 10.3. Пример с заданием цветовых параметров

По умолчанию зеркальная составляющая в расчет не принимается, блики на поверхностях объектов не появляются. Чтобы учесть ее, надо включить режим `D3DRS_SPECULARENABLE`.

Я советую вам внимательно поработать с этим примером. Для начала по отдельности включите одну из трех составляющих, чтобы увидеть, как они проявляются на поверхности объектов. Назначьте ей белый цвет, а всем остальным — черный, и посмотрите результат.

Этот пример может стать очень полезным в моменты, когда вам потребуется подобрать материал для построений. Ведь наверняка далеко не у каждого из вас под рукой окажется справочник оптических свойств материалов.

После того как вы хорошенько поработаете с этим примером, я хочу обсудить с вами важную проблему, напрямую не относящуюся к основной теме главы. Поговорим с вами на тему выбора объектов. Выбор по цвету, предлагаемый мною в предыдущих примерах, напрямую использовать очень сложно. Если мы вернемся к тестовой сцене с конусом и сферой и внимательно посмотрим на получающуюся картинку, то увидим, что значение пиксела экрана никак не поможет решить задачу выбора: оба объекта имеют

участки черного или очень темного цвета. Даже в таком случае, когда цвета объектов различаются кардинально, их очень тяжело отличать. Например, на поверхности объектов могут появляться блики одинакового цвета. А если на объекты накладывается текстура, или объекты покрашены одинаковым цветом, задача выбора по цвету становится неразрешимой. В случае DirectDraw мы решали подобную проблему использованием вспомогательной поверхности, на которой объекты раскрашивались по произвольной схеме, аналогичный метод можно применять и в Direct3D. Мы можем на вспомогательном, невидимом зрителю экране, повторить построения сцены, окрашивая объекты так, как нам удобно для их идентификации, и ориентироваться при выборе по значению пиксела в определенной точке этого экрана.

Вспомним, что нам системой предоставлены два экрана, передний и задний буферы, причем второй экран скрыт от зрителя до тех пор, пока не вызывается метод `Present` объекта устройства. Поэтому данным экраном мы можем воспользоваться для наших целей, осуществляя в него построения по нужной схеме, и не выкладывать его содержимое на передний экран. Система предоставляет нам доступ к содержимому заднего буфера, с помощью метода `GetBackBuffer` объекта устройства, результат помещается в объект типа `IDirect3DSurface8`.

Чтобы окрашивать объекты в чистые цвета, можно в формат вершин включить диффузный компонент, аналогично нашим первым смоделированным объектам, и отключать при построениях в заднем буфере источники света, запретив работу с освещением. Таким образом, мы добьемся, что все пикселы, занимаемые объектом, примут одинаковый, сплошной цвет.

Переходим к иллюстрации — проекту из каталога Ex04, где рисуется знакомая тестовая сцена, при щелчке кнопки мыши сообщается, какой объект находится под курсором (рис. 10.4).

Первым делом обращаю ваше внимание на то, что при инициализации графической системы необходимо указать возможность запирания поверхности заднего буфера, для чего в поле `Flags` структуры `TD3DPRESENT_PARAMETERS` необходимо занести соответствующую константу:

```
ZeroMemory(@d3dpp, SizeOf(d3dpp));
with d3dpp do begin
  Windowed := True;
  SwapEffect := D3DSWAPEFFECT_DISCARD;
  // Разрешаем запирание поверхности заднего буфера
  Flags := D3DPRESENTFLAG_LOCKABLE_BACKBUFFER;
  BackBufferFormat := d3ddm.Format;
  EnableAutoDepthStencil := True;
  AutoDepthStencilFormat := D3DFMT_D16;
end;
```

Это очень важный момент, не упустите его.

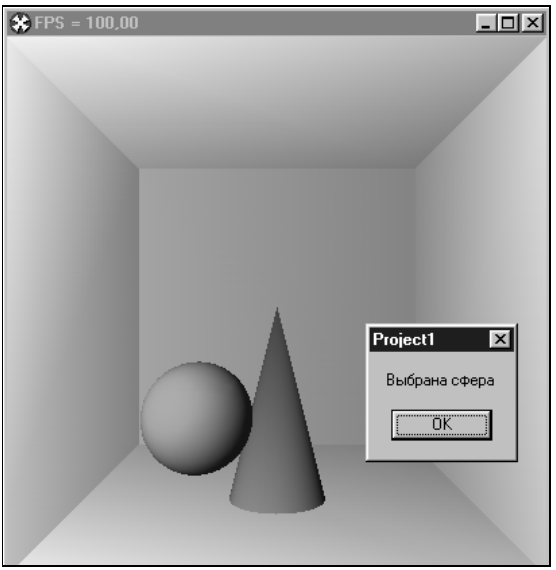


Рис. 10.4. В примере осуществляется выбор пространственных объектов

Формат вершин включает в себя координаты, нормаль и цветовую составляющую:

```
D3DFVF_CUSTOMVERTEX = D3DFVF_XYZ or D3DFVF_NORMAL or D3DFVF_DIFFUSE;
```

При заполнении буфера вершин цветовая составляющая заполняется только для треугольников сферы и конуса. Для треугольников, образующих комнату, значение диффузной составляющей вершин остается нулевым. Вы можете оптимизировать подобные моменты и использовать отдельные форматы вершин.

Материалы для стен, конуса и сферы инициализируются точно так же, как в первоначальном примере, но при обычном воспроизведении необходимо обязательно указать, что окрашивание треугольников производится с учетом текущего установленного материала, а не значения диффузной составляющей их вершин:

```
with FD3DDevice do begin
  SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
  SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
  SetRenderState(D3DRS_AMBIENT, $00202020);
  SetRenderState(D3DRS_LIGHTING, Dword (True));
  SetRenderState(D3DRS_NORMALIZENORMALS, DWORD (True));
  // Явно указываем использование материала
  SetRenderState(D3DRS_DIFFUSEMATERIALSOURCE, D3DMCS_MATERIAL);
  SetRenderState(D3DRS_SPECULARMATERIALSOURCE, D3DMCS_MATERIAL);
  SetRenderState(D3DRS_AMBIENTMATERIALSOURCE, D3DMCS_MATERIAL);
end;
```

При движении курсора мыши по поверхности окна отслеживаются его координаты:

```
var
  OX, OY : DWORD;
procedure TfrmD3D.FormMouseMove(Sender: TObject; Shift: TShiftState;
                                X, Y: Integer);
begin
  OX := X;
  OY := Y;
end;
```

Вы можете оптимизировать часть кода, связанную с определением позиции, ведь для получения положение курсора в любой момент времени можно использовать функцию `GetCursorPos`.

Помимо функции `Render`, я ввел функцию укороченного воспроизведения, которая отображает сцену с измененными установками и не заканчивается переключением буферов:

```
function TfrmD3D.Draw : HRESULT;
var
  hRet : HRESULT;
begin
  if FD3DDevice = nil then begin
    Result := E_FAIL;
    Exit;
  end;
  // Очищаем только Z-буфер
  hRet := FD3DDevice.Clear(0, nil, D3DCLEAR_ZBUFFER, 0, 1.0, 0);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := FD3DDevice.BeginScene;
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  with FD3DDevice do begin
    SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
    SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
    // Работа с освещением запрещена
    SetRenderState(D3DRS_LIGHTING, Dword (False));
  end;
  DrawScene;        // Рисуем комнату
  Result := FD3DDevice.EndScene;
end;
```

При отключенном освещении стены комнаты будут выглядеть черными. Поэтому нам незачем тратить время для очистки цветового буфера. Здесь также можно оптимизировать код, воспроизводить только те объекты, между которыми будет осуществляться выбор, и не тратить время на воспроизведение объектов фона. В таком случае потребуется, конечно, очищать цветовой буфер.

Чтобы увидеть, каким остается содержимое заднего буфера после работы этой функции, можете дополнить ее строкой переключения буферов. После щелчка кнопки мыши вы увидите такую же картинку, как на рис. 10.5.

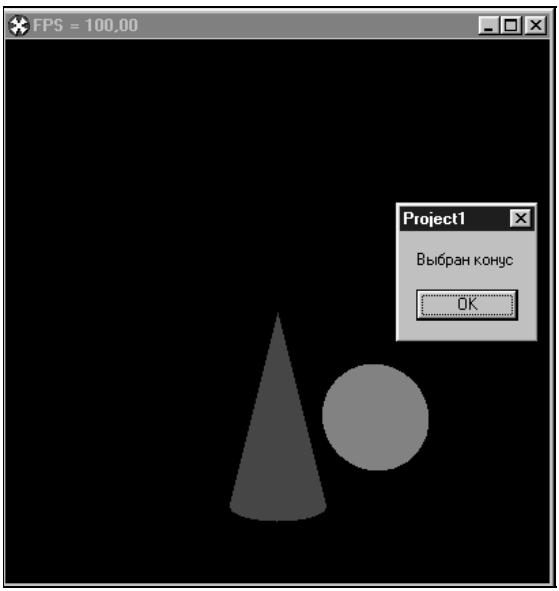


Рис. 10.5. Содержимое заднего буфера в момент выбора

При щелчке кнопки мыши получаем доступ к заднему буферу, запираем полученную поверхность и анализируем содержимое нужного пиксела:

```
procedure TfrmD3D.FormClick(Sender: TObject);
var
  Back : IDirect3DSurface8;     // Поверхность заднего буфера
  d3dlr : TD3DLOCKED_RECT;
  dwDstPitch : DWORD;
  hRet : HRESULT;
  DWColor : DWORD;
  R, G, B : Byte;
begin
  R := 0;  // Инициализация для предотвращения предупреждений компилятора
  G := 0;
```

```
  B := 0;
  FActive := False;    // Перерисовку кадра временно отменяем
  Back := nil;
  hRet := Draw;        // Рисуем упрощенный вариант сцены, в задний буфер
  if Failed (hret) then ErrorOut ('Draw', hRet);
  // Получаем доступ к заднему буферу
  hRet := FD3DDevice.GetBackBuffer (0, D3DBACKBUFFER_TYPE_MONO, Back);
  if Failed (hret) then ErrorOut ('GetBackBuffer', hRet);
  // Обнуляем поля вспомогательной структуры
  ZeroMemory (@d3dlr, SizeOf (d3dlr));
  // Поверхность заднего буфера запирается
  hRet := Back.LockRect (d3dlr, nil, D3DLOCK_READONLY);
  if Failed (hret) then ErrorOut ('LockRect', hRet);
  // Значение смещения при выравнивании поверхности
  dwDstPitch := d3dlr.Pitch;
  case d3ddm.Format of            // Текущий формат рабочего стола
    D3DFMT_X8R8G8B8 : begin       // 32-битный RGB
      // Пиксел, соответствующий позиции курсора
      DWColor := PDWORD (DWORD(d3dlr.pBits) + OY *
                 dwDstPitch + OX * 4)^;
      // Цветовые веса пиксела
      R := (DWColor shr 23) and $1f;
      G := (DWColor shr 7)  and $1f;
      B := DWColor and $1f;
    end;
    D3DFMT_R5G6B5 : begin         // 16-битный 5-6-5
      DWColor := PDWORD (DWORD(d3dlr.pBits) + OY *
                 dwDstPitch + OX * 2)^;
      R := (DWColor shr 11) and $1f;
      G := (DWColor shr 5)  and $3f;
      B :=  DWColor and $1f;
    end;
  end;
  Back.UnLockRect;      // Возможное исключение не обрабатывается
  if Assigned (Back) then begin   // Удаляем поверхность
    Back._Release;
    Back := nil;
  end;
  // Интерпретация результата
  if B <> 0 then ShowMessage ('Выбран конус') else
  if R <> 0 then ShowMessage ('Выбрана сфера') else
  if G <> 0 then ShowMessage ('Выбран объект зеленого цвета') else
                 ShowMessage ('Ничего не выбрано');
  Factive := True;
end;
```

Первый аргумент метода `GetBackBuffer` указывает номер присоединенного буфера, основан на нуле[1]. Вторым аргументом является константа. В момент написания книги здесь можно использовать единственно возможное значение, `D3DBACKBUFFER_TYPE_MONO`. Последний аргумент метода — переменная типа `IDirect3DSurface8`, в которую помещается результат. Поверхности в Direct3D очень похожи на знакомые нам по DirectDraw, на время доступа к их содержимому они должны запираться.

При анализе содержимого пиксела я предусмотрел поддержку только двух, наиболее распространенных, форматов пиксела, и этот код, возможно, вам придется дополнить.

Зеленую составляющую пиксела мы в этом примере никак не используем, но я оставил рассмотрение ее значения для предотвращения замечаний компилятора. Удалять этот код я не стал, вам он может понадобиться для выбора из трех объектов.

Выбор по цвету, разобранный в данном примере, вы можете использовать для идентификации сотен объектов. Ведь объекты могут различаться оттенками, и совсем не обязательно, чтобы они окрашивались именно в чистые цвета: вы можете использовать смеси всех трех цветов.

Туман

Простейшим средством передачи глубины пространства является включение дымки. Объекты сцены в таком режиме при удалении от наблюдателя становятся менее различимыми, погружаются в туман.

Работа с туманом в Direct3D очень простая. Достаточно включить указанный режим и задать несколько параметров. При воспроизведении графическая система будет учитывать эти установки, и никаких изменений в коде воспроизведения объектов сцены не требуется.

Параметры тумана таковы:

- формула, задающая закон эффекта (линейный или экспоненциальный);
- плотность дымки, указываемая для нелинейных законов;
- интервал, на протяжении которого эффект действует, используется для линейного закона;
- цвет тумана.

При линейном законе плотность дымки равномерно увеличивается по мере удаления от глаза наблюдателя. Дымка действует в пределах интервала от передней до задней плоскостей отсечения. Этот интервал можно сузить, задавая значение параметров `D3DRS_FOGSTART` и `D3DRS_FOGEND`. Есть две схемы

[1] То есть нумерация начинается с 0. — *Ред.*

расчета тумана: пикселная и вершинная. Если задана первая схема, значения связанных с расстоянием параметров лежат в пределах от нуля до единицы и задают расстояния относительно текущих видовых параметров. Минимальное значение соответствует расстоянию до передней плоскости отсечения, максимальное соотносится с задней плоскостью. Во второй, вершинной схеме тумана значения параметров указывают на действительное расстояние в мировом пространстве. Для большей определенности я буду применять только одну, первую схему. Ей соответствует режим `D3DRS_FOGTABLEMODE`. Для использования вершинной схемы необходимо менять установки состояния `D3DRS_FOGVERTEXMODE`. В обеих схемах объекты, располагающиеся дальше границы действия тумана, становятся совершенно неразличимыми.

Нелинейных законов два: оба опираются на экспоненциальную зависимость, но в одном из них используется экспонента квадрата. Аргументом экспоненты в обоих случаях является произведение расстояния и весового фактора, называемого плотностью. Этот параметр должен быть вещественным и не превышать 1.

Проект каталога Ex05 поможет вам глубже постичь все вышесказанное. Тестовая композиция воспроизводится на панели, рядом с которой располагаются элементы, позволяющие менять текущие параметры тумана (рис. 10.6).

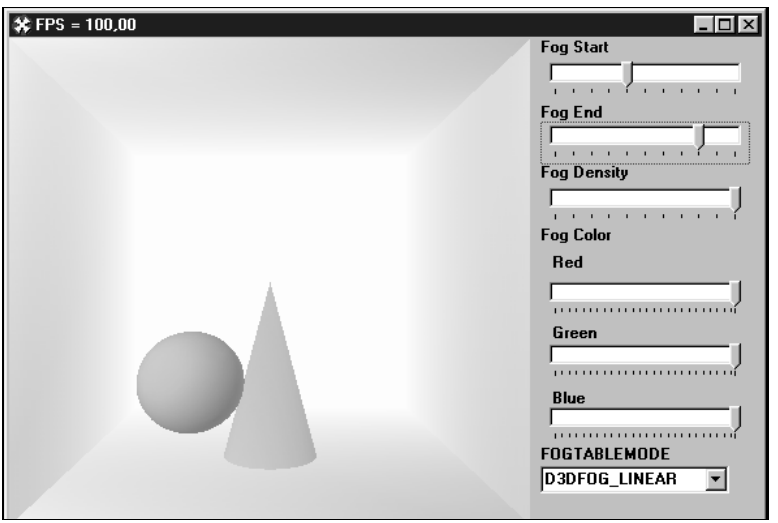


Рис. 10.6. Пример для изучения смысла параметров дымки

Для возможности динамической смены параметров их значения хранятся в переменных:

```
var
  FogDensity : Single = 1.0;     // Плотность
  FogStart : Single = 0.4;       // Расстояние, с которого туман действует
```

```
  FogEnd : Single = 1.0;        // Граничное расстояние действия тумана
  FogColor : DWORD = $00FFFFFF; // Цвет тумана, первоначально — белый
  FOGTABLEMODE : DWORD = D3DFOG_LINEAR; // Закон тумана
...
  with FD3DDevice do begin
    // Включаем режим использования дымки
    SetRenderState(D3DRS_FOGENABLE, DWORD (True));
    // Используем пикселную схему расчета тумана
    SetRenderState(D3DRS_FOGTABLEMODE, FOGTABLEMODE);
    // Устанавливаем текущие параметры тумана
    SetRenderState(D3DRS_FOGCOLOR, FogColor);
    SetRenderState(D3DRS_FOGDENSITY, PDWORD (@FogDensity)^);
    SetRenderState(D3DRS_FOGSTART, PDWORD (@FogStart)^);
    SetRenderState(D3DRS_FOGEND, PDWORD (@FogEnd)^);
  end;
```

При изменении пользователем состояний интерфейсных элементов меняются значения соответствующих переменных:

```
procedure TfrmD3D.tbStartChange(Sender: TObject); // Ползунок "Fog Start"
begin
  FogStart := tbStart.Position / 10;
end;

procedure TfrmD3D.tbEndChange(Sender: TObject);   // Ползунок "Fog End"
begin
  FogEnd := tbEnd.Position / 10;
end;

procedure TfrmD3D.tbDensityChange(Sender: TObject); // Ползунок "Density"
begin
  FogDensity := tbDensity.Position / 10;
end;

// Ползунки, связанные с цветовыми весами тумана
procedure TfrmD3D.tbRedChange(Sender: TObject);
begin
  FogColor := tbBlue.Position + (tbGreen.Position shl 8) +
                                (tbRed.Position shl (4 * 4));
end;

// Закон тумана
procedure TfrmD3D.cmbxFOGTABLEMODEChange(Sender: TObject);
begin
  case cmbxFOGTABLEMODE.ItemIndex of
       0 : FOGTABLEMODE := D3DFOG_NONE;
       1 : FOGTABLEMODE := D3DFOG_EXP;
```

```
      2 : FOGTABLEMODE := D3DFOG_EXP2;
      3 : FOGTABLEMODE := D3DFOG_LINEAR;
  end;
end;
```

Эффект дымки часто служит для усиления передачи глубины пространства, как в проекте каталога Ex06, где рисуется вращающийся додекаэдр (рис. 10.7).

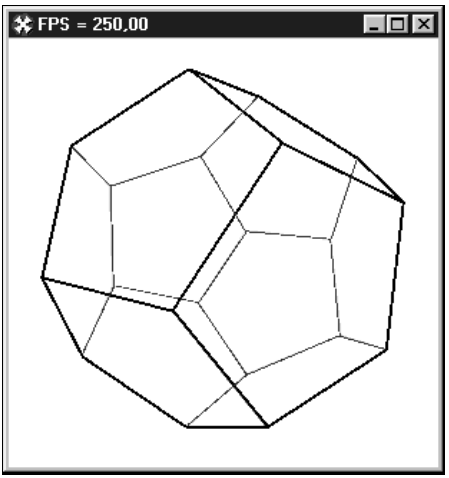


Рис. 10.7. Туман можно использовать для усиления эффекта пространства

При каркасном режиме зритель часто теряется в пространстве, гадая, как линии располагаются в пространстве, и включение режима тумана значительно улучшит восприятие таких картинок.

Двусторонние поверхности

Я обращал ваше внимание на то, что Direct3D умеет окрашивать примитивы только с одной стороны. В этом небольшом разделе, на примере проекта каталога Ex07 мы разберем принципы построения двусторонних поверхностей. Работа примера очень простая: на экране вращается квадрат, с одной стороны окрашенный в синий цвет, с другой — в красный. Цвета разные только для наглядности, чтобы мы могли различать стороны площадки. Используется два материала, но вы можете получать таким же способом примитивы, выглядящие одинаково независимо от точки обзора.

Метод очень прост: примитивы фигуры описываются дважды, с одинаковыми координатами, но противоположным направлением нормалей. В моем примере первые четыре вершины описывают связанные треугольники, образующие квадрат. Нормаль к вершинам задается из расчета, что описывается передняя сторона квадрата. Затем буфер наполняется четверкой вершин,

с противоположным направлением нормали. Считаем, что это соответствует задней стороне квадрата. В обоих случаях вершины перечисляются по часовой стрелке.

При воспроизведении выводим переднюю сторону квадрата, отсекая примитивы, вершины которых перечисляются в поле зрения против часовой стрелки. Затем выводим заднюю сторону квадрата, меняя правило отсечения на противоположное:

```
with FD3DDevice do begin
  SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
  SetMaterial(MaterialRed);
  DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
  SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
  SetMaterial(MaterialBlue);
  DrawPrimitive(D3DPT_TRIANGLESTRIP, 4, 2);
end;
```

Теперь передняя сторона квадрата не будет отображаться, если он повернут к нам обратной стороной, и наоборот, задняя сторона воспроизводится только тогда, когда квадрат развернулся к нам обратной стороной.

Соприкасающиеся поверхности

Обращаю ваше внимание еще на одну проблему, с которой вы можете столкнуться. Наверняка в ваших построениях рано или поздно потребуется использовать соприкасающиеся поверхности, и здесь вы можете обнаружить, что на таких поверхностях появляется паразитный узор.

Посмотрим на данный эффект, запустив проект из каталога Ex08, где рисуются две частично перекрывающиеся разноцветные площадки. В местах их соприкосновения возникает картинка, которую мы не рисовали (рис. 10.8).

Связано появление таких узоров с использованием буфера глубины. При его заполнении одинаковыми значениями из-за погрешностей некоторые участки примитивов выводятся перепутанными. Проявляется эффект только после смены матрицы трансформаций, как в этом примере:

```
procedure TfrmD3D.DrawScene;
var
  matRotateY, matTranslate : TD3DMatrix;
begin
  // Сдвиг и поворот первого квадрата
  SetTranslateMatrix (matTranslate, -0.5, -0.5, 0);
  SetRotateYMatrix(matRotateY, Angle);
  with FD3DDevice do begin
       SetTransform(D3DTS_WORLD, MatrixMul(matRotateY, matTranslate));
```

```
        SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
        SetMaterial(MaterialRed);
        DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
        // Сдвиг второго квадрата
        SetTranslateMatrix (matTranslate, -0.4, -0.4, 0);
        SetTransform(D3DTS_WORLD, MatrixMul(matRotateY, matTranslate));
        SetMaterial(MaterialBlue);
        DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
  end;
end;
```

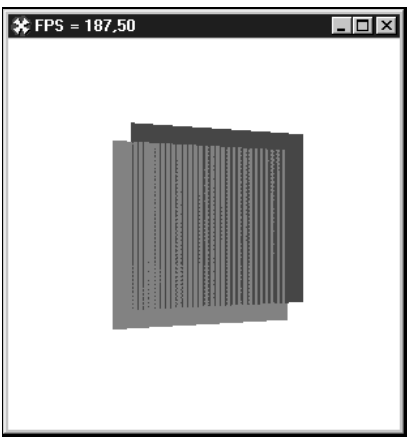


Рис. 10.8. Соприкасающиеся поверхности порождают нежелательные узоры

Если второй квадрат воспроизводить сразу же после первого, т. е. перед его воспроизведением не изменять матрицу трансформаций, ошибок возникать не будет. В таких случаях примитив, нарисованный последним, перекроет предыдущий без проступающих узоров.

Решение проблемы состоит в том, чтобы на время воспроизведения соприкасающихся поверхностей запретить работу с буфером глубины. Так и делается в проекте из каталога Ex09, где рисуется аналогичная сцена, но во время воспроизведения второго квадрата работа с Z-буфером приостанавливается:

```
SetRenderState(D3DRS_ZENABLE, D3DZB_FALSE);
DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
```

Конечно, в этом конкретном примере можно и не включать буфер глубины вообще, но если на сцене присутствует множество объектов, то без использования Z-буфера положения их будут передаваться неправильно. Поэтому обычно такое действие выполняют только на время воспроизведения одного из примитивов, имеющих большие участки одинаковой координаты.

Частичная прозрачность объемных фигур

Предыдущие разделы подготовили нас к тому, чтобы вплотную заняться выводом объемных полупрозрачных объектов. Для начала рассмотрим вывод одной единственной фигуры на сцене (проект каталога Ex10), в которой сфера стала наполовину прозрачной (рис. 10.9).

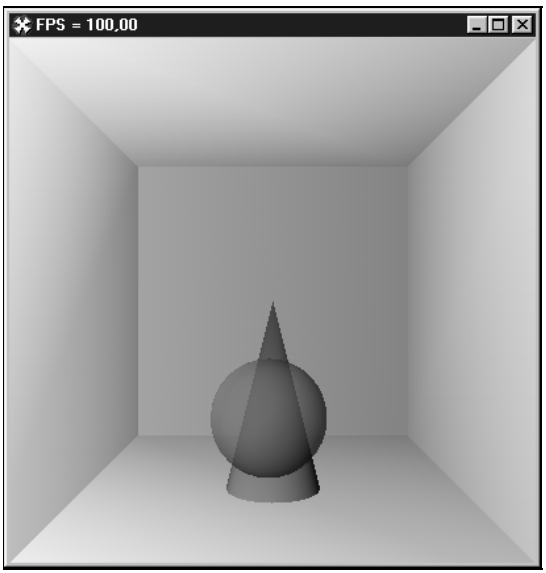


Рис. 10.9. Первый пример частичной прозрачности объемных фигур

При инициализации материала сферы четвертый компонент цвета равен теперь 0.5, чтобы сфера стала наполовину прозрачной. Обратите внимание, что нулевое значение этого параметра соответствует полной прозрачности материала, прямо противоположно тому, что мы имели при работе с диффузной составляющей формата вершин.

Помимо этого, нам нужно позаботиться, чтобы сфера имела двустороннюю поверхность. Данные о сфере заносятся теперь дважды. Во втором случае координаты вершин повторяются, направление нормалей меняем на прямо противоположное.

Последнее, что нам необходимо учесть — операция с буфером глубины применяется раньше, чем с буфером цвета. Поэтому первой следует вывести внутреннюю сторону сферы, она загорожена лицевой стороной сферы, и при обычном порядке воспроизведения двусторонних поверхностей альфа-смешения не произойдет:

```
with FD3DDevice do begin
  SetTransform(D3DTS_WORLD, matSphere);
```

```
  // Устанавливаем полупрозрачный материал
  SetMaterial(MaterialSphere);
  // Включаем режим смешения
  SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD(True));
  // Первой выводится внутренняя сторона сферы
  SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
  DrawPrimitive(D3DPT_TRIANGLELIST, 30 + 51 + 51 + 960, 960);
  // Внешняя сторона сферы
  SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
  DrawPrimitive(D3DPT_TRIANGLELIST, 30 + 51 + 51, 960);
  SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD(False));
end;
```

Обязательно посмотрите, как работает пример с переставленным порядком воспроизведения и убедитесь, что в этом случае эффект получается точно таким же, как и при отсутствии воспроизведения внутренней стороны сферы.

Теперь мы попытаемся внести в сцену еще небольшое изменение — сделать полупрозрачным конус. Конечно, мы помним, что помимо изменения свойств материала для конуса требуется также добавить дублированное описание, с перевернутыми нормалями его внутренней стороны. Но для этой фигуры нашей композиции есть еще одна тонкость: конус стоит на полу комнаты, его дно соприкасается с квадратом пола. Следовательно, на время воспроизведения этой части фигуры надо отключать работу с Z-буфером, иначе при включении полупрозрачности нам станут видны паразитные узоры. В коде примера из каталога Ex11 я и делаю именно так:

```
// Первой воспроизводится внутренняя поверхность конуса
  SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);
  DrawPrimitive(D3DPT_TRIANGLEFAN, 81 + 51, 49);        // Сам конус
// Дно конуса рисуется с отключенной работой Z-буфера
  SetRenderState(D3DRS_ZENABLE, D3DZB_FALSE);
  DrawPrimitive(D3DPT_TRIANGLEFAN, 81 + 51 + 51, 49);   // Дно
  SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
// Второй воспроизводится внешняя поверхность конуса
  SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
  SetRenderState(D3DRS_ZENABLE, D3DZB_FALSE);
  DrawPrimitive(D3DPT_TRIANGLEFAN, 30, 49);
  SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
  DrawPrimitive(D3DPT_TRIANGLEFAN, 81, 49);
```

Конус теперь рисуется замечательно, и мы видим его внутреннюю часть, на которой нет никаких непрошеных узоров. Однако работа примера не может удовлетворять нас полностью, поскольку при прохождении сферы за конусом она становится невидна, как будто конус непрозрачный (рис. 10.10).

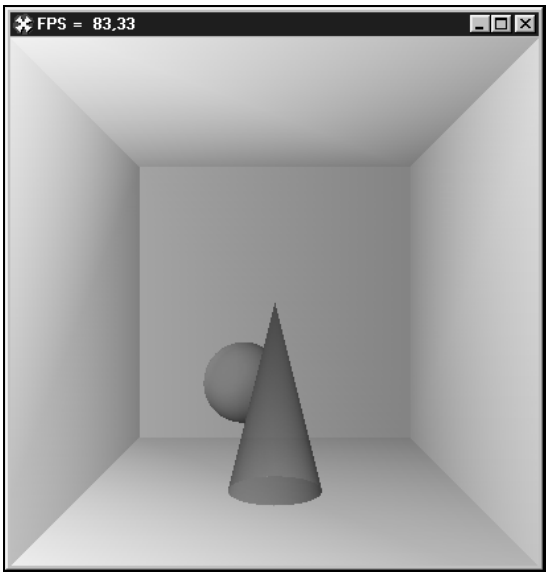


Рис. 10.10. При присутствии на сцене нескольких полупрозрачных объектов требуется сортировка вывода

Мы знаем, почему так происходит и что необходимо сделать, чтобы все выводилось правильно. Объекты следует воспроизводить отсортированными в поле зрения так, чтобы первыми выводились более удаленные объекты, и заполнять кадр полупрозрачным цветом до того, как будет применена операция отсечения по содержимому буфера глубины.

Сейчас посмотрите работу проекта каталога Ex12, в котором оба объекта выводятся полупрозрачными. В зависимости от текущего значения переменной `Angle`, определяющего положение сферы в пространстве, задаем порядок вывода фигур сцены: первой воспроизводится фигура, располагающаяся в текущий момент дальше от глаза наблюдателя:

```
if Angle < Pi
then   // Конус расположен дальше, чем сфера; первым выводится конус
else   // Конус загораживает сферу; первой выводится сфера
```

Наложение текстуры на трехмерные объекты

Для использования трехмерных объектов, покрытых текстурой, необходимо, конечно, описание их вершин дополнить текстурными координатами. Помимо инициализации текстуры, это является минимальным действием, резко усиливающим зрелищность наших построений.

Проект каталога Ex13 представляет собой первый пример на данную тему. Это вариация одного из наших примеров с вращающимся кубиком. Формат вершин включает в себя нормали и текстурные координаты. Нормали, правда, в примере не применяются и оставлены мною "про запас". Не используются они постольку, поскольку на сцене отсутствуют источники света. Так что их удаление из описания вершин не приведет к каким-либо изменениям в работе данного примера. Работа с текстурой в рассматриваемом примере ничем не отличается от наших плоских построений, и запомните, что задание режимов текстуры в привычное для нас значение приводит к тому, что работа с освещенностью не осуществляется:

```
SetTextureStageState(0, D3DTSS_COLOROP, D3DTA_TEXTURE);
```

Файл текстуры для этого примера я взял на сайте **nehe.gamedev.net**, она мне показалась очень подходящей для наложения на кубик (рис. 10.11).

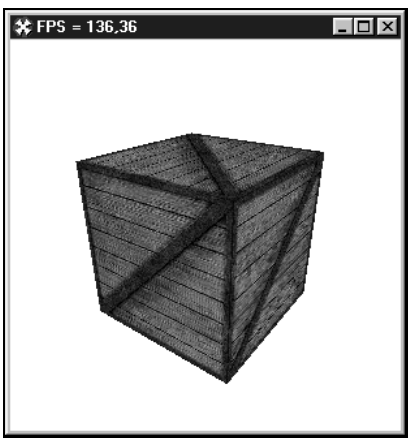


Рис. 10.11. Простой, но эффектный пример использования текстуры в пространственных построениях

Конечно, большая часть того, что мы наблюдаем в играх, представляет собой наложение текстур. В проекте из каталога Ex14 вы можете увидеть, как наложение текстур на стороны куба позволяет создать окружение игрока. Здесь глаз наблюдателя помещается внутрь куба, на каждую сторону которого наложена текстура, имитирующая картину, наблюдаемую зрителем при повороте головы. Запомните, что поверхность покрывается текстурой с обеих сторон.

Растры для примера взяты мною с сайта **gamedeveloper.org/delphi3d**.

Если необходимо модулировать, т. е. накладывать освещенность на поверхность, покрытую текстурой, то параметры следует задавать так:

```
SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
```

В проекте из каталога Ex15 формат вершин включает в себя пространственные координаты, нормаль, диффузную составляющую и текстурные координаты.

```
type
  TCUSTOMVERTEX = packed record
    X, Y, Z : Single;
    nX, nY, nZ : Single;
    DColor : DWORD;
    U, V : Single;
  end;
const
D3DFVF_CUSTOMVERTEX = D3DFVF_XYZ or D3DFVF_NORMAL or
                      D3DFVF_DIFFUSE or D3DFVF_TEX1;
```

Буфер вершин заполняется данными о сфере. В этом примере пространственные и текстурные координаты, а также нормали вершин, рассчитываются, и строится последовательность связанных треугольников.

На сферу накладывается растр с изображением поверхности Земли, который я позаимствовал из набора файлов, поставляемых в составе DirectX SDK.

В примере диффузная составляющая вершин сферы задается белым цветом. Поскольку по умолчанию в Direct3D при разрешенной работе с освещенностью используется диффузный компонент вершины, в примере просто включается направленный источник света, материалы не используются. Так как в этом примере включена модуляция, участки глобуса отличаются по своей яркости (рис. 10.12).

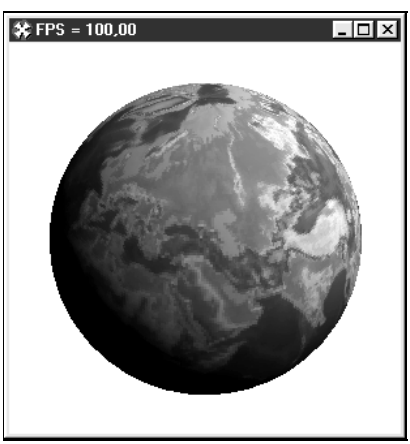


Рис. 10.12. Пример использования текстуры совместно с источником света

Сейчас в качестве упражнений выполните следующие задания:

- задайте диффузную составляющую вершин сферы, отличную от белого цвета, и посмотрите результат;
- запретите работу с освещением объектов и посмотрите результат;
- рассмотрите работу примера с запрещенной модуляцией;

- в примере на тему выбора объектов найдите значения состояний для включения материала; в рассмотренном выше примере задайте материал для сферы и добейтесь того, чтобы глобус окрашивался с учетом текущего материала.

Точечный источник света также подходит для освещения объектов, покрытых текстурой. Так, в примере проекта каталога Ex16 рисуется тестовая сцена комнаты с конусом и сферой. На стены комнаты здесь накладываются разнообразные текстуры.

В предыдущих примерах текстура накладывалась на простые объекты, но вам, наверняка, захочется узнать, возможно ли использование текстуры с объектами сложной формы. Если у вас возникли вопросы по этому поводу, отсылаю вас к следующему примеру, проектам каталога Ex17, в одном из которых выводится модель игрока из игры Quake, а во втором — вращающаяся голова. Программа импорта, которой я пользовался для подготовки примеров этой книги, позволяет записывать в результирующем файле и текстурные координаты моделей. Мне оставалось только отметить такую опцию при записи результирующего файла.

Механизм трехмерной игры

Этот раздел я закончу примером, который можно считать заготовкой трехмерной игры. Но прежде, чем мы перейдем непосредственно к этому проекту, посмотрим решение двух связанных с ним задач: вывод текста в пространстве и раскрашивание модели.

Вам, наверняка, пригодится моя простая программа из каталога Ex18, с помощью которой создается файл, содержащий координаты вершин треугольников, образующих нужный символ установленного для формы шрифта. Программа основана на материале моей книги по OpenGL, подробно рассматривать ее здесь не буду, ограничусь лишь небольшими замечаниями по поводу ее использования.

Требуемый символ должен устанавливаться аргументом процедуры `OutText`, вызываемой в коде два раза: первый раз — для получения координат вершин треугольников, второй раз — для контрольного отображения на экране. В текстовый файл выводятся построчно две координаты очередной вершины треугольника, по оси X и по оси Y. Количество треугольников заранее неизвестно и зависит от базового символа. Выводимые в файл координаты вершин соответствуют оконным, поэтому при дальнейшем использовании должны быть масштабированы. Как правило, вершины треугольников перечисляются по часовой стрелке, но возможны исключения.

Еще один проект (из каталога Ex19) строит средствами Direct3D символ, используя файл, полученный по результатам работы предыдущей программы. Количество считываемых треугольников необходимо установить равным

константе `NumTriangles`. Считываемые координаты вершин масштабируются при заполнении буфера вершин.

Замечу также, что оба примера могут использоваться и для вывода фраз целиком, а не только отдельных символов.

Сейчас перейдем к очередному примеру (проекту из каталога Ex20), во время работы которого на экране воспроизводится симпатичная модель человечка из детского конструктора (рис. 10.13).

Рис. 10.13. Во время работы примера человечек шевелит конечностями

Подходящую модель я нашел по Internet-адресу **http://www.people.zeelandnet.nl/nihil/download/legoman.zip**. Автор модели, Kortekaas, любезно предоставил разрешение на использование ее в этой книге.

Эта модель также конвертирована мною с помощью программы импорта 3D Exploration, а код был преобразован из программы на языке C++. При импортировании комплексных моделей, состоящих, как в данном примере, из нескольких частей, в код вставляются метки-имена составляющих элементов. По этим меткам можно ориентироваться для получения данных о том, сколько треугольников потрачено на описание отдельной части, чтобы идентифицировать каждый элемент:

```
procedure TfrmD3D.DrawScene;
begin
  with FD3DDevice do begin
    // Ноги покрашены материалом серого цвета
    SetMaterial(MaterialGray);
    SetTransform(D3DTS_WORLD, matLeftFoot);
    // Левая нога
    DrawPrimitive(D3DPT_TRIANGLELIST, 0, 112);
    // Правая нога
    SetTransform(D3DTS_WORLD, matRightFoot);
    DrawPrimitive(D3DPT_TRIANGLELIST, (112 + 204) * 3, 112);
```

```
    // Руки покрашены красным цветом
    SetMaterial(MaterialRed);
    // Левая рука
    SetTransform(D3DTS_WORLD, matLeftHand);
    DrawPrimitive(D3DPT_TRIANGLELIST,(112+204 + 112 + 620 + 614)*3, 612);
    // Кисти — желтого цвета
    SetMaterial(MaterialYellow);
    // Левая кисть
    DrawPrimitive(D3DPT_TRIANGLELIST,(112+204+112+620+614+612)*3, 324);
    SetMaterial(MaterialRed);
    SetTransform(D3DTS_WORLD, matRightHand);
    // Правая рука
    DrawPrimitive(D3DPT_TRIANGLELIST, (112 + 204 + 112 + 620) * 3, 614);
    // Правая кисть
    SetMaterial(MaterialYellow);
   DrawPrimitive(D3DPT_TRIANGLELIST,
                (112+204+112+620+614+612+324)*3, 324);
    // Голова
    SetTransform(D3DTS_WORLD, matRot);
    DrawPrimitive(D3DPT_TRIANGLELIST, (112 + 204 + 112) * 3, 620);
    // Туловище, красного цвета
    SetMaterial(MaterialRed);
    DrawPrimitive(D3DPT_TRIANGLELIST, 112 * 3, 204);
  end;
end;
```

Буфер вершин заполняется данными на всю модель целиком, а при воспроизведении отдельных частей из него последовательно выбираются соответствующие треугольники. Перед воспроизведением каждого элемента устанавливается предварительно рассчитанная матрица трансформаций, поэтому изначально монолитная модель пришла в движение. Для каждого элемента модели задается индивидуальный материал, поэтому модель стала разноцветной. Фигурирующие числа получены следующим образом: я подсчитал количество отдельных фасетов между метками, расставленными программой моделирования трехмерных объектов в описании массива `face_indicies`.

Матрицы, связанные с поворотом конечностей, из соображений оптимизации вычисляются не при каждой перерисовке кадра, а только при изменении значений управляющих переменных. Обратите внимание, что поворот конечностей в точках крепления осуществляется следующим образом: система координат перемещается в точку крепления, выполняется поворот, а затем система координат возвращается в первоначальное положение:

```
procedure TfrmD3D.MoveMan;
begin
  // Поворот глобальной системы координат,
  // вращение всей модели вокруг своей оси
  SetRotateZMatrix (matRot, Angle);
```

```
  // Переменная, задающая вращение конечностей
  AngleFoot := AngleFoot + StepFoot;
  if (AngleFoot > Pi / 4) or (AngleFoot < -Pi / 4)
     then StepFoot := -StepFoot;
  // Ноги вращаются в противофазе
  SetRotateXMatrix (rotLeftFoot, AngleFoot);
  SetRotateXMatrix (rotRightFoot, -AngleFoot);
  // Поворот левой ноги, в три этапа
  matLeftFoot := MatrixMul(matRot,
                 MatrixMul(transFoot2,
                 MatrixMul(rotLeftFoot, transFoot1)));
  // Поворот правой ноги
  matRightFoot := MatrixMul(matRot,
                  MatrixMul(transFoot2,
                  MatrixMul(rotRightFoot, transFoot1)));
  // Поворот левой руки
  matLeftHand := MatrixMul(matRot,
                 MatrixMul(transHand2,
                 MatrixMul(rotRightFoot, transHand1)));
  // Поворот правой руки
  matRightHand := MatrixMul(matRot,
                  MatrixMul(transHand2,
                  MatrixMul(rotLeftFoot, transHand1)));
end;
```

Рабочие матрицы, связанные с перемещениями в точки крепления конечностей, инициализируются один раз, в начале работы приложения:

```
SetTranslateMatrix(transFoot1, 0, 0, 0.25);
SetTranslateMatrix(transFoot2, 0, 0, -0.25);
SetTranslateMatrix(transHand1, 0.25, 0.0, -0.23);
SetTranslateMatrix(transHand2, -0.25, 0.0, 0.23);
```

Этот пример я подготовил для использования в дальнейшем в расчете на то, что человечком можно будет легко управлять, перемещая его в пространстве. Но если на сцене присутствует только одна модель, для оптимизации можно сократить количество операций с матрицами. В самом деле, в этом примере матрица `matRot`, связанная с глобальной системой координат, может вообще не использоваться: модель можно не вращать и оставить неподвижной, а перемещать точку зрения наблюдателя. Эффект вращения модели останется, а количество операций существенно уменьшится.

И теперь мы можем перейти к разбору заключительного примера — проекта каталога Ex21. Как я уже говорил, это заготовка трехмерной игры: игрок попадает внутрь комнаты, населенной движущимися человечками (рис. 10.14).

Окружение игрока построено из текстур, накладываемых на треугольники, описание окружающего мира загружается из текстового файла.

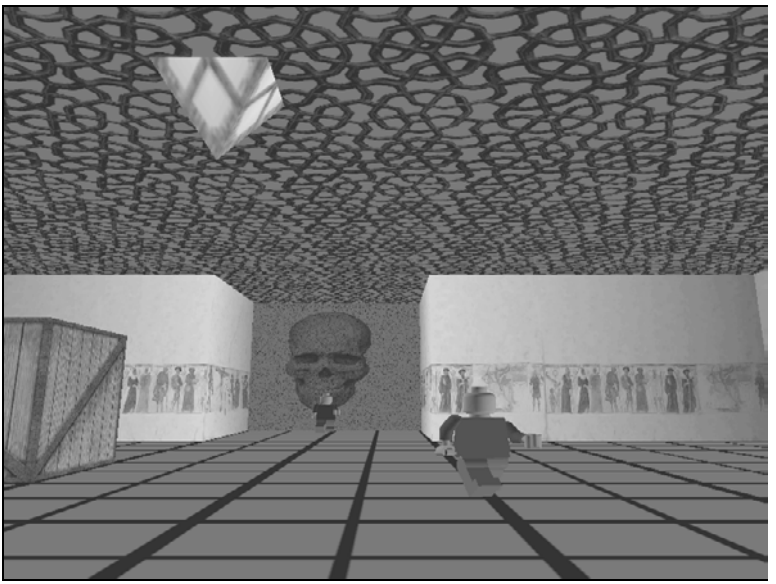


Рис. 10.14. Фрагмент работы игрового движка

```
type
  // Формат вершин для треугольников окружения
  TNormDiffTextVertex = packed record
    X, Y, Z : Single;
    nX, nY, nZ : Single;
    DColor : DWORD;
    U, V : Single;
  end;
  // Формат вершин для треугольников человечков
  TNormVertex = packed record
    X, Y, Z : Single;
    nX, nY, nZ : Single;
  end;
  // Отдельный треугольник описания окружения
  TTriangle = record
    NumTexture : Integer;   // Номер текстуры
    DIFFUSE : DWORD;        // Диффузная составляющая треугольника
  end;
const
  // FVF-флаг для треугольников окружения
  D3DFVF_NORMDIFFTEXTVERTEX = D3DFVF_XYZ or D3DFVF_NORMAL or
                              D3DFVF_DIFFUSE or D3DFVF_TEX1;
  // FVF-флаг для треугольников человечков
  D3DFVF_NORMVERTEX = D3DFVF_XYZ or D3DFVF_NORMAL;
```

```
  // Имя файла с описанием мира
  WorldFile = 'Data/World.txt';
  // Имя файла с треугольниками символов, для вывода FPS
  NumbersFile = 'Data/Numbers.txt';
  // Количество треугольников в описании окружения
  NumTriangles = 58;
{$I legoman.pas}           // Данные модели
var
  frmD3D: TfrmD3D;
  Frames : Integer = 0;    // Счетчик кадров
  FpsOut : String = '';    // Значение FPS
  // Вспомогательная матрица, для вывода символов FPS
  LetTrans : TD3DMatrix;
  // Используется как вспомогательный массив для хранения образа текстуры
  TexPointer : Pointer;
  // Характеристики образа текстуры
  wrkTexWidth, wrkTexHeight : Integer;
  // Флаг, выводить ли FPS
  flgFPS : BOOL = True;
  // Угол зрения по вертикали
  Lookupdown : Single = 0.0;
  // Вспомогательный вектор для оптимизации
  ZVector : TD3DVector;
  // Угол зрения по горизонтали и положение игрока
  RotY, XPos, ZPos : Single;
  // Массив описания мира
  World : Array [0..NumTriangles  - 1] of TTriangle;
  // Переменные для обработки устройств ввода
  DInput : IDIRECTINPUT8 = nil;
  DIMouse : IDIRECTINPUTDEVICE8 = nil;
  DIKeyboard : IDirectInputDevice8;
  KeyBuffer : TDIKeyboardState;
  // Угол поворота красного человечка
  Angle : Single = 0.0;
  // Угол поворота конечностей человечков
  AngleFoot : Single = 0.0;
  StepFoot : Single = 0.1;
  // Тестовая точка для определения столкновений с препятствиями
  TestPointX, TestPointY : DWORD;
```

В файле описания окружения данных идут в следующем порядке:

- строка комментария;
- номер текстуры;
- цвет треугольника;

- три строки описания вершин треугольника, которые включают координаты в пространстве;
- нормаль и текстовые координаты каждой вершины треугольника.

Вот что записано в текстовом файле для первого треугольника:

```
// Потолок
4
$00FF0000
 -3.0  1.0  3.0 0.0 -1.0  0.0  0.0  0.0
 -3.0  1.0 -3.0 0.0 -1.0  0.0  0.0  12.0
1.0  3.0 0.0 -1.0  0.0  12.0  0.0
```

Пол и потолок комнаты представляют собой квадраты с координатами точек углов по диагонали (−3; −3) и (3; 3). Координата Y для всех вершин пола нулевая, для вершин потолка — единичная. При считывании данных предусматриваем обработку исключений на случай отсутствия файла данных или присутствия ошибки при описании треугольников:

```
procedure TfrmD3D.SetupWorld;
var
  t : TextFile;
  i, j : Integer;
  Vertices : ^TNormDiffTextVertex;
  wrkStr : String;
begin
  if FileExists(WorldFile) then begin
    AssignFile(t, WorldFile);
    try
    Reset(t);
    FD3DVB.Lock(0, NumTriangles * 3 * SizeOf(TNormDiffTextVertex),
                PByte(Vertices), 0);
    for i := 0 to NumTriangles  - 1 do begin
      // Строка комментария, в программе не используется
      ReadLn (t, wrkStr);
      ReadLn (t, World[i].NumTexture);   // Текстура треугольника
      ReadLn (t, World[i].DIFFUSE);      // Цвет вершин треугольника
      for j := 0 to 2 do begin           // Три вершины треугольника
        ReadLn (t, Vertices.X, Vertices.Y, Vertices.Z,
                   Vertices.nX, Vertices.nY, Vertices.nZ,
                   Vertices.U, Vertices.V);
        Vertices.DColor := World[i].DIFFUSE;
        Inc(Vertices);
     end;
  end;
```

```
  FD3DVB.Unlock;
  except     // Данные на треугольник заданы неверно
    raise EAbort.Create ('Can''t read file: ' + WorldFile);
  end;
  CloseFile(t);
  end else raise EAbort.Create ('Can''t read file: ' + WorldFile);
end;
```

При возникновении исключений программа завершается, описание ошибки выводится в текстовый файл.

Помимо треугольников, образующих стены комнаты, на сцене присутствуют треугольники стоящего в комнате ящика и пирамиды источника света, прикрепленного к потолку. Обратите внимание, что треугольники пола и потолка окрашены красным цветом, а треугольники препятствий, стен и ящика — синим. Позже я поясню смысл этого окрашивания.

Координаты игрока задаются значениями переменных `Xpos` и `Zpos`, переменная `RotY` определяет угол поворота головы наблюдателя вокруг своей оси, а переменная `Lookupdown` — наклон головы по вертикали. Сразу после запуска игрок "располагается" в точке (0, 0, 0), направление взгляда параллельно оси X.

Текстуры треугольников задаются обычным образом, но текстуры, накладываемые на квадраты выходов из сектора, инициализируются отдельной функцией:

```
procedure TfrmD3D.FormCreate(Sender: TObject);
var
  hRet : HRESULT;
  matView, matProj : TD3DMatrix;
  wrkMat : TD3DMatrix;   // Вспомогательная матрица разворота человечков
begin
  // Приложение полноэкранное, курсор отключаем
  ShowCursor (False);
  Randomize;
  hRet := InitD3D;
  if Failed (hRet) then ErrorOut ('InitD3D', hRet);
  hRet := InitVB;
  if Failed (hRet) then ErrorOut ('InitVertex', hRet);
  try
    InitVBLetter;        // Считываются треугольники цифр
  except                 // Возможно, файл удален
    on E : EAbort do ErrorOut (PChar(E.Message), S_FALSE);
  end;
  InitMan;               // Инициализация буфера вершин человечков
  try
    SetupWorld;          // Считываем данные мира
```

```
except
  on E : EAbort do ErrorOut (PChar(E.Message), S_FALSE);
end;
// Инициализация используемых в программе материалов
MaterialWhite   := InitMaterial (1.0, 1.0, 1.0, 0.0);
MaterialYellow  := InitMaterial(1, 1, 0, 0);
MaterialRed     := InitMaterial(1, 0, 0, 0);
MaterialGray    := InitMaterial(0.5, 0.5, 0.5, 0);
MaterialBlue    := InitMaterial(0, 0, 1.0, 0);
MaterialGreen   := InitMaterial(0, 1, 0, 0);
// Вспомогательный вектор для видовой трансформации
ZVector := D3DVector(0, 1, 0);
// Матрица перемещений букв при выводе FPS
LetTrans := IdentityMatrix;
LetTrans._42 := 0.5;
LetTrans._43 := 0.9;
// Первоначальные положения человечков
transMan1 := IdentityMatrix;
transMan2 := IdentityMatrix;
transMan2._41 := 3.1;   // Синий человечек перемещается по оси X
transMan3 := IdentityMatrix;
// Зеленый человечек устанавливается в первоначальное положение
transMan3._41:= Man3PosX;
transMan3._43 := Man3PosZ;
// Разворот модели человечков
SetRotateYMatrix (wrkMat, -Pi / 2);
SetRotateXMatrix (matWrk1, -Pi / 2);
matWrk2 := MatrixMul (wrkMat, Matwrk1);
matWrk3 := matWrk2;
// Вспомогательные матрицы для поворота конечностей
SetTranslateMatrix(transFoot1, 0, 0, -0.1);
SetTranslateMatrix(transFoot2, 0, 0, 0.1);
SetTranslateMatrix(transHand1, 0.25, 0.0, -0.2);
SetTranslateMatrix(transHand2, -0.25, 0.0, 0.2);
SetupLights;
// Первоначальные установки, в дальнейшем переопределяются
SetViewMatrix(matView, D3DVector(0, 0, 0), D3DVector(0, 0, 1),
              ZVector);
FD3DDevice.SetTransform(D3DTS_VIEW, matView);
SetProjectionMatrix(matProj, 1, 1, 0.01, 6);
FD3DDevice.SetTransform(D3DTS_PROJECTION, matProj);
// Инициализация текстур
try
  InitTexture (FD3DTextures [0], 'data/0.bmp');
  InitTexture (FD3DTextures [1], 'data/1.bmp');
```

```
    InitTexture (FD3DTextures [2], 'data/2.bmp');
    InitTexture (FD3DTextures [3], 'data/3.bmp');
    InitTexture (FD3DTextures [4], 'data/4.bmp');
    InitTexture (FD3DTextures [5], 'data/5.bmp');
    BukupTexture (FD3DTextures [6], 'data/6.bmp');
  except
    on E : EAbort do ErrorOut (PChar(E.Message), S_FALSE);
  end;
  OnCreateDevice;   // Инициализация устройств ввода
end;
```

Всего предусмотрено три источника света: два направленных и один точечный, располагающийся под потолком в центре комнаты:

```
procedure TfrmD3D.SetupLights;
var
  Light0 : TD3DLight8;
  Light1 : TD3DLight8;
  Light2 : TD3DLight8;
begin
// Направленные источники светят во взаимно противоположных направлениях
  Light0 := InitDirectionalLight(D3DVector(-0.5, -0.5, -1), 0.5,
                                 0.5, 0.5, 0);
  Light1 := InitDirectionalLight(VectorNormalize(D3DVector(0.5, 0.5, 1)),
                                 0.5, 0.5, 0.5, 0);
  // Точечный источник
  ZeroMemory(@Light2, SizeOf(Light2));
  with Light2 do begin
    _Type := D3DLIGHT_POINT;
    Diffuse.r := 0.5;
    Diffuse.g := 0.5;
    Diffuse.b := 0.5;
    Specular := Diffuse;
    Ambient := Diffuse;
    Position := D3DVector(0.0, 1.0, 0.0);
    Attenuation0 := 1.0;
    Attenuation1 := 0.0;
    Attenuation2 := 0.0;
    Range := 2.5;
  end;
  with FD3DDevice do begin
    SetLight(0, Light0);
    SetLight(1, Light1);
    SetLight(2, Light2);
    LightEnable(0, True);
```

```
    LightEnable(1, True);
    LightEnable(2, True);
  end;
end;
```

Все объекты сцены, за исключением человечков, освещаются тремя источниками света. При воспроизведении человечков точечный источник выключается.

При воспроизведении сцены голову наблюдателя "помещаем" в точку, соответствующую его текущему положению в пространстве, и поворачиваем ее в направлении `RotY`:

```
procedure TfrmD3D.DrawScene;
var
  i : Integer;
  matView : TD3DMatrix;
begin
  // Видовая матрица, в соответствии с текущими параметрами игрока
  SetViewMatrix(matView, D3DVector(XPos, 0.25, ZPos),
                         D3DVector(XPos + cos(RotY), 0.25 + Lookupdown,
                                   ZPos  — sin (RotY)), ZVector);
  with FD3DDevice do begin
    SetTransform(D3DTS_VIEW, matView);
    SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
    SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
    SetRenderState(D3DRS_LIGHTING, DWORD (True));
  end;
  // При необходимости выводим значение FPS
  if flgFPS then DrawLetters;
  // Рисуем человечков
  DrawMan1;    // Красный
  DrawMan2;    // Синий
  DrawMan3;    // Зеленый
  // Подготовка к рисованию стен
  with FD3DDevice do begin
    // Учитывать освещение
    SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
    // Проводить интерполяцию текстур
    SetTextureStageState(0,D3DTSS_MAGFILTER, D3DTEXF_LINEAR);
    SetTextureStageState(0,D3DTSS_MINFILTER, D3DTEXF_LINEAR);
    // Не учитывать диффузию треугольников окружения
    SetRenderState(D3DRS_DIFFUSEMATERIALSOURCE, D3DMCS_MATERIAL);
    SetRenderState(D3DRS_AMBIENT, $000F0F0F);
    // Задаем белый материал
    SetMaterial(MaterialWhite);
```

```
    // Направляем потоки на буфер вершин описания мира
    SetStreamSource(0, FD3DVB, SizeOf(TNormDiffTextVertex));
    SetVertexShader(D3DFVF_NORMDIFFTEXTVERTEX);
    // Координаты треугольников заданы в глобальной системе координат
    SetTransform(D3DTS_WORLD, IdentityMatrix);
  end;
  // Цикл вывода треугольников окружения
  for i := 0 to NumTriangles - 1 do with FD3DDevice do begin
    // Устанавливаем нужную текстуру в соответствии с описанием
    SetTexture(0, FD3DTextures[World [i].NumTexture]);
    DrawPrimitive(D3DPT_TRIANGLELIST, i * 3, 1); // Вывод треугольника
  end;
  FD3Ddevice.SetTexture(0, nil);    // Текстура больше не используется
end;
```

Обратите внимание, что в этом примере задано интерполирование текстур, так называемая *билинейная фильтрация*. Сделано это для того, чтобы при приближении к ящику и стенам не проявлялась блочность текстур, а изображение не становилось бы крупнозернистым (рис. 10.15).

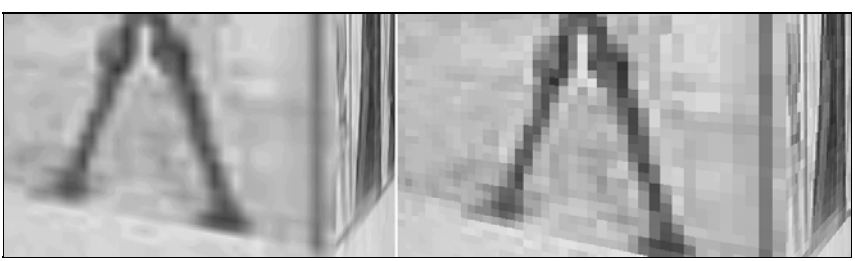


Рис. 10.15. Слева — работа приложения с включенной фильтрацией, справа — без нее

Учтите, что использование интерполяции существенно снижает работу приложения, поэтому обычно ее включают только при приближении к поверхности, покрытой текстурой. Другой способ достижения мелкой зернистости — использование чередующихся текстур. В зависимости от расстояния до объекта на него накладываются текстуры различной детализации.

Также я должен напомнить, что для оптимизации работы приложения следует применять запомненные блоки состояний.

Три человечка, присутствующие на сцене, перемещаются по различным законам. Первый, одетый в красную футболку, беспрерывно кружит вокруг центра комнаты:

```
procedure TfrmD3D.MoveMan1;
begin
  // Поворот вокруг вертикальной оси
  SetRotateYMatrix (rotMan1, Angle + Pi);
```

```
  // Перемещение по кругу
  transMan1._41 := cos (-Angle) / 2;
  transMan1._43 := sin(-Angle) / 2;
  // Опорная трансформация первого человечка
  matMan1 := MatrixMul(transMan1, MatrixMul(rotMan1, matWrk1));
...
```

Второй человечек пересекает комнату, появляясь из одной стены и исчезая в противоположной:

```
procedure TfrmD3D.MoveMan2;
begin
  // Изменение X-координаты
  transMan2._41 := transMan2._41  - 0.01;
  // При прохождении комнаты процесс начинается сначала
  if transMan2._41 < -3.1 then transMan2._41 := 3.1;
  matMan2 := MatrixMul(transMan2, matWrk2);
...
```

Третий человечек назойливо преследует игрока, перемещается в направлении к наблюдателю, всегда разворачиваясь к нему лицом:

```
procedure TfrmD3D.MoveMan3;
var
  wrkAngle : Single;
  distX, distZ : Single;
begin
  // Расстояния до игрока
  distX := XPos  - Man3PosX;
  distZ := ZPos  - Man3PosZ;
  // Вычисляем угол поворота человечка
  if distZ < 0
     then wrkAngle := arctan (distX / distZ)  - Pi / 2
     else wrkAngle := arctan (distX / distZ) + Pi / 2;
  // Разворот человечка лицом к игроку
  SetRotateYMatrix (rotMan3, wrkAngle);
  // Если человечек удален от зрителя, то двигается в его направлении
  if (abs(distX) > 0.02) and (abs (distZ) > 0.02) then begin
    Man3PosX := Man3PosX + distX / 20;   // Новое положение человечка
    Man3PosZ := Man3PosZ + distZ / 20;
    transMan3._41 := Man3PosX;
    transMan3._43 := Man3PosZ;
  end;
  // Опорная матрица третьего человечка
  matMan3 := MatrixMul(transMan3, MatrixMul(rotMan3, matWrk2));
...
```

Для упрощения вычислений я позволяю человечкам проходить сквозь препятствия и друг через друга. Код предотвращения этих ситуаций очень прост, и вы можете самостоятельно дополнить его отслеживанием таких ситуаций.

Для вывода значения FPS треугольники символов цифр и точки объединены мною в один файл numbers.txt. Процедура `PlaceLetter` определяет в потоке положение и количество треугольников для нужного символа:

```
procedure TfrmD3D.DrawLetters;
var
  i : Integer;
  nS, nW : Integer;
begin
  with FD3DDevice do begin
    // Некоторые треугольники построены против часовой
    SetRenderState(D3DRS_CULLMODE, D3DCULL_NONE);
    // Не тратить время на освещение
    SetRenderState(D3DRS_LIGHTING, DWORD (False));
    // Направляем поток на буфер символов
    SetStreamSource(0, FD3DVBLetter, SizeOf(TNormVertex));
    SetVertexShader(D3DFVF_NORMVERTEX);
  end;
  // Цикл вывода в пространстве символов FPS
  for i := 1 to Length(FpsOut) do begin
    // Получаем положение треугольников символа
    PlaceLetter (FpsOut[i], nS, nW);
    // Сдвигаемся в пространстве для вывода очередного символа
    LetTrans._41 := i * 0.1;
    FD3DDevice.SetTransform(D3DTS_WORLD, LetTrans);
    FD3DDevice.DrawPrimitive(D3DPT_TRIANGLELIST, nS, nW);
  end;
  // Возвращаем обычные установки
  with FD3DDevice do begin
    SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);
    SetRenderState(D3DRS_LIGHTING, DWORD (True));
  end;
end;
```

Символы выводятся "подвешенными" в воздухе, что выглядит красиво и загадочно (рис. 10.16).

Приложение я тестировал на машине с очень скромными ресурсами. Возможно, вы получите более впечатляющую цифру. Наиболее весомый удар по скорости работы данного примера наносится фильтрацией текстуры, а усложнение игрового мира не приведет к сильному падению этого значения, до некоторой степени. Например, удаление человечков практически не

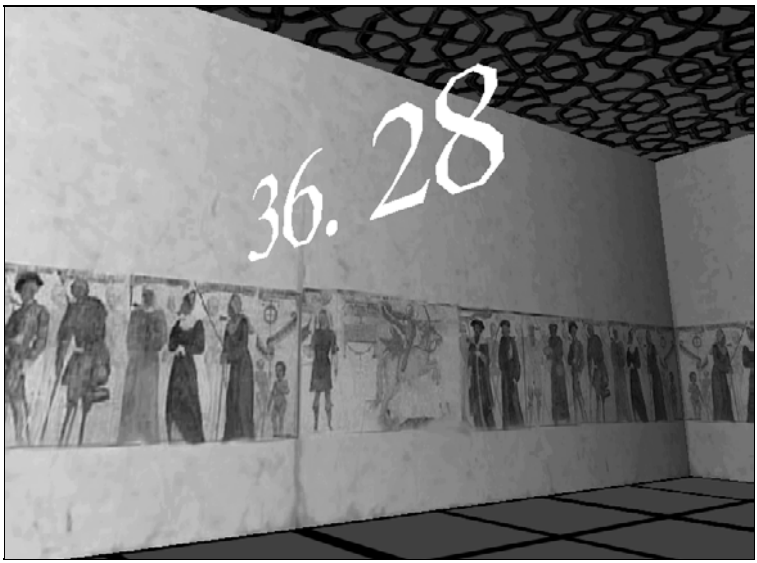


Рис. 10.16. Для красивого вывода FPS не требуется особых усилий, но все-таки скрадывается немного драгоценного времени

сказывается на скорости работы программы. Также динамическая смена текстуры, используемая мною для стен, символизирующих выходы из сектора, не привела к заметному замедлению:

```
function TfrmD3D.BukupTexture (var FD3TextBMP : IDIRECT3DTEXTURE8;
                               const FileName : String) : HRESULT;
var
  hRet : HRESULT;
  d3dlr : TD3DLOCKED_RECT;
  dwDstPitch : DWORD;
  X, Y : DWORD;
  Bmp : TBitmap;
  R, G, B : Byte;
begin
  Bmp := TBitmap.Create;
  try
    Bmp.LoadFromfile (FileName);
  except
    raise EAbort.Create ('Can''t open file: ' + FileName);
    Result := S_FALSE;
    Exit;
  end;
  hRet := FD3DDevice.CreateTexture (Bmp.Width, Bmp.Height, 0, 0,
                                    D3DFMT_A8R8G8B8,
                                    D3DPOOL_MANAGED, FD3TextBMP);
```

```
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
  hRet := FD3TextBMP.LockRect(0, d3dlr, nil, 0);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;

  dwDstPitch := d3dlr.Pitch;
  for Y := 0 to Bmp.Height  - 1 do
    for X := 0 to Bmp.Width  - 1 do begin
      R := GetRValue (Bmp.Canvas.Pixels [X,
                      DWORD (Bmp.Height - 1) - Y]);
      G := GetGValue (Bmp.Canvas.Pixels [X,
                      DWORD (Bmp.Height - 1) - Y]);
      B := GetBValue (Bmp.Canvas.Pixels [X,
                      DWORD (Bmp.Height - 1) - Y]);
      PDWORD(DWORD(d3dlr.pBits)+Y*dwDstPitch+X*4)^ :=
            D3DCOLOR_XRGB(R, G, B);
    end;
  // Резервируем место для копии первоначального растра
  GetMem (TexPointer, 4 * Bmp.Width * Bmp.Height);
  // Запоминаем первоначальный растр
  CopyMemory (TexPointer, d3dlr.pBits, 4 * Bmp.Width * Bmp.Height);
  wrkTexWidth := Bmp.Width;
  wrkTexHeight := Bmp.Height;
  Bmp.Free;
  Result := FD3TextBMP.UnlockRect(0);
end;

// Покрытие снегом текстуры
function TfrmD3D.SnowTexture (var FD3TextBMP : IDIRECT3DTEXTURE8) :
                              HRESULT;
var
  hRet : HRESULT;
  d3dlr : TD3DLOCKED_RECT;
  i : Integer;
  dwDstPitch : DWORD;
begin
  // Запираем прямоугольник текстуры
  hRet := FD3TextBMP.LockRect(0, d3dlr, nil, 0);
  if FAILED(hRet) then begin
    Result := hRet;
    Exit;
  end;
```

```
  // Копируем в него первоначальный растр
  CopyMemory (d3dlr.pBits, TexPointer, 4 * wrkTexWidth * wrkTexHeight);
  dwDstPitch := d3dlr.Pitch;
  // Произвольные точки текстуры закрашиваем черным
  for i := 1 to 10000 do
  PDWORD (DWORD(d3dlr.pBits) + DWORD(random(wrkTexHeight)) * dwDstPitch +
          DWORD(random(wrkTexWidth)) * 4)^ := 0;
  Result := FD3TextBMP.UnlockRect(0);
end;
```

Одно из самых важных мест кода — управление игроком. Перемещения мыши изменяют его положение и угол поворота головы по горизонтали, клавиши управления курсором отвечают за положение игрока в пространстве, клавиши <Page Up> и <Page Down> ответственны за угол поворота головы по вертикали:

```
function TfrmD3D.ReadImmediateData : HRESULT;
var
  hRet : HRESULT;
  dims2 : TDIMOUSESTATE2;
  NewXPos, NewZPos : Single;
begin
  ZeroMemory(@dims2, SizeOf(dims2));
  hRet := DIMouse.GetDeviceState(SizeOf(TDIMOUSESTATE2), @dims2);
  if Failed (hRet) then begin
    hRet := DIMouse.Acquire;
    while hRet = DIERR_INPUTLOST do hRet := DIMouse.Acquire;
  end;
  // Перемещение курсора мыши влево-вправо
  if dims2.lX <> 0
    // Меняем угол поворота головы по горизонтали
    then RotY := RotY + 0.01 * dims2.lX;
  // Перемещение курсора мыши вперед-назад
  if dims2.lY > 0 then begin        // Движение игрока назад
    // Вычисляем новое положение
    NewXPos := XPos + sin(RotY  - Pi / 2) * 0.05;
    NewZPos := ZPos + cos(RotY  - Pi / 2) * 0.05;
    // Нет ли препятствий к движению назад, голову разворачиваем
    if TestRender (NewXPos, NewZPos, RotY  - Pi) then begin
      XPos := NewXPos;         // Препятствий нет, перемещаем игрока
      ZPos := NewZPos;
    end
  end else
  if dims2.lY < 0 then begin   // Движение вперед
    NewXPos := XPos + sin(RotY + Pi / 2) * 0.05;
    NewZPos := ZPos + cos(RotY + Pi / 2) * 0.05;
```

```
    // Есть ли препятствия к движению
    if TestRender (NewXPos, NewZPos, RotY) then begin
      XPos := NewXPos;
      ZPos := NewZPos;
    end;
  end;
  // Обработка клавиатуры
  Result := DIKeyboard.GetDeviceState(SizeOf(KeyBuffer), @KeyBuffer);
  if KeyBuffer[DIK_ESCAPE] and $80 <> 0 then begin             // Esc
     Close;
     Exit;
  end;
  // Нажата клавиша "вправо", вычисляем новое положение в пространстве
  if KeyBuffer[DIK_RIGHT] and $80 <> 0 then begin
     XPos := XPos  - sin(RotY) * 0.05;
     ZPos := ZPos  - cos(RotY) * 0.05;
  end;
  // Нажата клавиша "влево"
  if KeyBuffer[DIK_LEFT] and $80 <> 0 then begin
     XPos := XPos + sin(RotY) * 0.05;
     ZPos := ZPos + cos(RotY) * 0.05;
  end;
  // Нажата клавиша "вниз"
  if KeyBuffer[DIK_DOWN] and $80 <> 0 then begin
     XPos := XPos + sin(RotY  - Pi / 2) * 0.05;
     ZPos := ZPos + cos(RotY  - Pi / 2) * 0.05;
  end;
  // Нажата клавиша "вверх"
  if KeyBuffer[DIK_UP] and $80 <> 0 then begin
     XPos := XPos + sin(RotY + Pi / 2) * 0.05;
     ZPos := ZPos + cos(RotY + Pi / 2) * 0.05;
  end;
  // Нажата клавиша "F", показывать ли значение FPS
  if KeyBuffer[DIK_F] and $80 <> 0 then begin
     flgFPS := not flgFPS;    // Обращение значения флага
     Sleep (50);              // Маленькая пауза
  end;
  // Клавиша <Page Up>, голову задираем вверх
  if KeyBuffer[DIK_PRIOR] and $80 <> 0 then begin
     Lookupdown := Lookupdown + 0.05;
     if Lookupdown > 1 then Lookupdown := 1;
  end;
  // Клавиша <Page Down>, голову опускаем вниз
  if KeyBuffer[DIK_NEXT] and $80 <> 0 then begin
     Lookupdown := Lookupdown - 0.05;
```

```
    if Lookupdown < -1 then Lookupdown := -1;
  end;
end;
```

Обратите внимание, что при перемещении с помощью мыши осуществляется проверка, нет ли на пути движения препятствия, стены комнаты или ящика. При нажатии клавиш такую проверку не произвожу, и игрок свободно проходит через все препятствия. Опускаю я проверку, чтобы определить, сильно ли она замедляет работу программы.

Для проверки того, свободен ли путь, я применяю самый простой метод: в заднем буфере сцена воспроизводится в новой точке, взгляд наблюдателя при этом повернут в направлении движения. Глаз наблюдателя опускаем ближе к полу, и выясняем цвет точки, расположенной прямо по ходу движения. Поскольку пол окрашивается красным, а препятствия и фон — синим, то синий цвет контрольной точки означает, что игрок вплотную подошел к непреодолимому препятствию или выходит за границу сектора:

```
function TfrmD3D.TestRender (const XPos, ZPos, RotY : Single) : BOOL;
var
  i : Integer;
  matView : TD3DMatrix;
  d3dlr : TD3DLOCKED_RECT;
  dwDstPitch : DWORD;
  DWColor : DWORD;
  B : Byte;       // Доля синего пиксела контрольной точки
begin
  B := 0;         // Предотвращение замечаний компилятора
  // Смотрим на сцену из новой точки, по вертикали — ближе к полу
  SetViewMatrix(matView, D3DVector(XPos, 0.1, ZPos),
                         D3DVector(XPos + cos(RotY), 0.1,
                                   ZPos  — sin(RotY)), ZVector);
  // Упрощенное воспроизведение сцены
  with FD3DDevice do begin
    Clear(0, nil, D3DCLEAR_TARGET or D3DCLEAR_ZBUFFER,
          $000000FF, 1.0, 0);
    BeginScene;
    // Отключаем источники света
    SetRenderState(D3DRS_LIGHTING, DWORD (False));
    // Использовать диффузный компонент описания вершин
    SetRenderState(D3DRS_DIFFUSEMATERIALSOURCE, D3DMCS_COLOR1);
    SetTransform(D3DTS_VIEW, matView);
    SetStreamSource(0, FD3DVB, SizeOf(TNormDiffTextVertex));
    SetVertexShader(D3DFVF_NORMDIFFTEXTVERTEX);
    SetTransform(D3DTS_WORLD, IdentityMatrix);
  end;
```

```
  // Рисуем только комнату
  for i := 0 to NumTriangles  - 1 do with FD3DDevice do
    DrawPrimitive(D3DPT_TRIANGLELIST, i * 3, 1);
  with FD3DDevice do begin
    EndScene;
    // Получаем доступ к заднему буферу
    GetBackBuffer (0, D3DBACKBUFFER_TYPE_MONO, FD3SurfBack);
    SetRenderState(D3DRS_LIGHTING, DWORD (True));
  end;
  // Запираем задний буфер
  FD3SurfBack.LockRect (d3dlr, nil, D3DLOCK_READONLY);
  dwDstPitch := d3dlr.Pitch;
  // Определяем долю синего в контрольной точке
  case FD3DfmtFullscreen of
    D3DFMT_X8R8G8B8 : begin
      DWColor := PDWORD (DWORD(d3dlr.pBits) + TestPointY * dwDstPitch +
                         TestPointX * 4)^;
      B := DWColor and $1f;
    end;
    D3DFMT_R5G6B5 : begin
      DWColor := PDWORD (DWORD(d3dlr.pBits) + TestPointY * dwDstPitch +
                         TestPointX * 2)^;
      B := DWColor and $1f;
    end;
  end;
  FD3SurfBack.UnLockRect;
  // Нет синего цвета, значит можно пройти
  Result := not (B <> 0);
end;
```

Синий цвет взят мною в качестве контрольного, поскольку для его вырезки требуется минимум операций. Замечу, что код этой функции можно дополнительно оптимизировать, например, вполне можно обойтись без использования промежуточной переменной `DWColor`.

Если установлен 24-битный режим, соответствующий формату `D3DFMT_R8G8B8`, то X-координату контрольной точки надо умножить на 3, именно столько байт отводится для одного пиксела в этом режиме.

Контрольная точка для определения столкновения с препятствиями берется одна — посередине экрана по горизонтали, на 10 пикселов выше нижней границы экрана:

```
ScreenWidth := GetSystemMetrics(SM_CXSCREEN);
ScreenHeight := GetSystemMetrics(SM_CYSCREEN);
TestPointX := ScreenWidth div 2;
TestPointY := DWORD(ScreenHeight - 10);
```

Используемый здесь алгоритм — самый простой, но, конечно, не самый совершенный. На его основе мы можем построить и более быстрые алгоритмы. Например, можно определенным цветом прорисовать только области, доступные для нахождения, и тогда достаточно легко определять цвет точки "под ногами". В этом случае описание мира усложняется, поскольку он описывается дважды, и требуется отдельный буфер. Вознаграждением будет то, что перемещения игрока при использовании такого алгоритма будет легко сделать пространственными, различая по оттенкам высоту препятствий. Поскольку положение игрока в пространстве является дискретным, можно воспользоваться массивом, значения элементов которого содержат данные о возможности нахождения игрока в определенной точке пространства, сведения о включении для этой точки фильтрации текстур, а также данные о занимаемой высоте.

Пример упрощен во многих отношениях, и, конечно, далек по своему качеству от профессиональных творений. Мастера должны с большой иронией смотреть на наши первые шаги в захватывающем мире программирования трехмерных игр, но ведь нам, например, не требуется кулинарного образования для того, чтобы приготовить себе обед, ведь так? И то, что профессиональным кулинарам может не понравиться наше кушанье, не означает, что мы должны оставаться голодными. На этом простом примере мы должны убедиться, что можем написать игру в принципе, дальше же нам предстоит совершенствоваться. Но эта тема иной книги, а данная книга на этих словах заканчивается.

Что вы узнали в этой главе

Direct3D располагает массой средств, позволяющих добиться высококачественных изображений, и в настоящей главе рассмотрена только небольшая их доля, например туман и источник света.

В заключительной главе мы познакомились с важными примерами, иллюстрирующими использование текстуры в пространственных построениях.

Закончилась глава примером простого движка трехмерной игры.

Заключение

Надеюсь, книга принесла вам пользу и удовольствие, но, возможно, у вас появились и замечания, которыми обязательно поделитесь со мной: **softgl@chat.ru**.

С данной книгой вы начали знакомство с DirectX, но это только вводное руководство, которое охватывает лишь небольшую толику огромной темы. В ней, например, нет ни слова о библиотеке Direct3DX, а это очень важная тема. Отказался я от ее рассмотрения постольку, поскольку для программ, написанных на Delphi, доступна она лишь в виде дополнительных и нестандартных файлов. Тем не менее после прочтения книги вам легко будет разобраться с этой библиотекой, существенно упрощающей многие действия, такие, например, как загрузка текстур.

Книга закончилась, но тема ее не закрыта вместе с ней, и, наверняка, вы пожелаете узнать нечто большее. Я хочу указать вам места, где вы найдете еще массу информации. Помимо адресов, указанных во введении, приведу еще несколько полезных ссылок.

- Конференция, посвященная DirectX, в которой вы можете найти ответы на ваши вопросы

  http://chat.mos.ru/fidow/fido.dll/_-0?Browse?hDpZs.&832&-1&9

- Ресурс, посвященный графике в Delphi

  http://turbo.gamedev.net/

- Сайты, предоставляющие сведения о программировании игр

  http://www.gamedev.net/

  http://www.flipcode.com/

  http://www.gamasutra.com/

  Здесь вы сможете узнать немало подробностей о секретах профессиональных разработчиков игр, найдете много статей с подсказками решений.

ПРИЛОЖЕНИЕ 1

Глоссарий

2D Graphics
Двумерная графика. Действие в такой графике происходит на одной плоскости.

3D Graphics
Трехмерная графика. Графическое отображение на дисплее трехмерной сцены.

Alpha
Альфа. Коэффициент прозрачности. В описание цвета может входить специальная составляющая, называемая альфа-каналом.

Alpha blending
Альфа-смешение. Смешивание значений цветов исходного и результирующего пикселов для достижения эффекта прозрачности и просвечивания.

Alpha channel
Альфа-канал. Массив значений, определяющих способ объединения пикселов изображения-источника с изображением-приемником. Альфа-буфер может использоваться для реализации прозрачности, размытия границ и создания тумана.

Ambient
Окружающая среда. Источник света, который светит одинаково во всех направлениях, все объекты освещаются с равной интенсивностью.

API (Application Programming Interface)
Интерфейс прикладного программирования. Спецификация набора функций, которой должны придерживаться разработчики программного обеспечения для совместимости своих программ с соответствующей операционной системой.

Back buffer
Вторичный буфер. Видеобуфер для подготовки следующего кадра анимационной последовательности, в которой первоначально осуществляется воспроизведение. Готовый вторичный буфер заменяет первичный (Front buffer) и, таким образом, выводится на экран.

BitBLT (Bit BLock Transfer)
Графическая операция, при которой прямоугольная область пикселов копируется между различными участками памяти с учетом требований графической памяти.

Bitmap
Прямоугольный битовый массив. Чаще всего применяется для хранения образа растра.

Brightness
Яркость. Характеристика цвета, определяющая интенсивность цвета.

Buffer
Буфер. Область адресуемой памяти центрального процессора системы или часть периферийного устройства, используемая для согласования различий между скоростями обмена, размерами блоков данных и моментами возникновения событий при обмене данными.

Channel
Канал. Компьютерная форма отображения каждой составляющей цветовой модели.

Computer graphics
Компьютерная графика. Общее направление, описывающее ввод, обработку и вывод графических изображений с помощью компьютера.

Contrast
Контраст. Степень тонового различия между областями изображения. Максимальный контраст реализуют белое и черное без всяких переходов, низкий контраст возникает при сближенных тонах без резких переходов.

Convex
Выпуклый многоугольник. Многоугольник, в котором никакие две вершины не могут быть соединены отрезком прямой, выходящим за пределы многоугольника.

Diffuse
Диффузное отражение. Световой поток, рассеиваемый объектом. Цвет потока в основном совпадает с естественным цветом объекта.

Directional
Направленный источник света. Источник света, освещающий все объекты сцены одинаково, в определенном направлении из бесконечности.

Double buffering
Двойная буферизация. Технология, при которой два или более буфера используются для создания анимационного изображения без эффекта мерцания. Новые данные записываются в буфер, который не отображается на экране (Back buffer), в то время, когда отображается содержимое другого буфера (Front buffer). Затем буферы переключаются (меняются местами), чтобы вывести новое изображение на экран.

Face cutting
Удаление задних либо передних граней. Для повышения скорости воспроизведения из расчетов исключаются задние части поверхности, если объект непрозрачный, и его обратную часть все равно не видно.

Flat shading (Flat)
Постоянное затенение. Поверхность объекта, построенного с использованием этого метода, получается наиболее низкого качества, и изображение выглядит блочным. Такой метод затенения дает худший результат, чем Gourad, но работает значительно быстрее.

Fogging
Туман. Комбинирование смешанных компьютерных цветовых пикселов с цветом тумана (Fog) под управлением функции, определяющей глубину дымки.

FPS (Frames Per Second)
Частота смены кадров. Величина, используемая для оценки быстродействия системы трехмерной визуализации, число кадров в секунду, которое система способна отобразить.

Frame buffer
Буфер кадра. Видеобуфер, содержащий текущее изображение на экране, делится на передний и задний буферы. Передний буфер — это то, что видит пользователь в данный момент. При воспроизведении передний буфер остается неизменным до формирования нового кадра полностью. При этом вся работа ведется с невидимым обратным буфером, который заметит пользователь через долю секунды. Такой механизм называется двойной буферизацией.

Front buffer
Первичный буфер. Область памяти, из которой происходит вывод кадра на экран. В приложениях, работающих в оконном режиме, первичный буфер используется совместно с другими приложениями.

Gamma
Гамма. Коэффициент контраста в средних тонах изображения. При низком общем уровне напряжения малое изменение напряжения приводит к изменению уровня яркости.

Gamma correction
Гамма-коррекция. Перед выводом на дисплей линейные данные RGB должны быть скорректированы для компенсации нелинейной составляющей дисплея.

Geometric primitive
Примитив. Точка, отрезок прямой или многоугольник.

Gouraud shading (Smooth shading)
Цветовая интерполяция. Наиболее популярный алгоритм затенения, обеспечивающий прорисовку плавных теней вокруг объекта, позволяющий отображать трехмерные объекты на плоском экране. Метод назван по имени его разработ-

чика, француза Генри Гуро. Цветовая информация интерполируется по поверхности многоугольника для определения цветов в каждом пикселе.

Gradient
Градиент. Плавный переход между двумя или несколькими цветами.

Image
Образ. Прямоугольный массив пикселов, располагающийся во вспомогательной памяти или в буфере кадра.

Interpolation
Интерполяция. Математический способ восстановления отсутствующей информации по некоторым заданным значениям.

Lighting
Освещение. Метод реалистичного отображения трехмерных объектов на плоском экране. При отображении объекта для придания ему объема используются разные уровни яркости.

Matrix
Матрица. Двумерный массив значений. В компьютерной графике используются, как правило, матрицы размером 4×4.

Motion blur
Размытие при движении. Технология имитации эффекта, возникающего при съемке быстро движущегося объекта, при котором его контуры выглядят размытыми.

Normal
Вектор нормали. Перпендикуляр к плоскости.

Parallel point
Параллельный источник света. Все объекты освещаются равномерно параллельным пучком света.

Perspective projection
Перспективная проекция. Тип проекции, создающий иллюзию глубины изображения. Грани объекта, находящиеся в отдалении от наблюдателя, кажутся меньше расположенных вблизи. Метод создания правдоподобного изображения трехмерного объекта на плоскости.

Pixel
Пиксел. Минимальный элемент изображения на экране монитора.

Plug-in
Дополнительный модуль. Программное обеспечение, разработанное сторонними компаниями для использования с программой.

Point
Точечный источник света. Светит одинаково во всех направлениях из одной точки.

Point graphics
Точечная графика. Изображение, состоящее из совокупности точек (пикселов). Каждый пиксел имеет атрибут цвета, кодируемый от 1 бита (черно-белый штрих) до 24 бит (цветное изображение с 16,7 млн оттенков) и выше.

Polygon
Многоугольник. Поверхность, ограниченная краями, заданными точками, вершинами.

Projection
Проецирование. Процесс преобразования видимой части трехмерного объекта для отображения на плоском дисплее.

Projection matrix
Матрица проекций. Матрица размером 4×4, использующаяся для преобразования позиций примитивов из видовых координат в координаты плоскости экрана.

Rasterization
Растеризация. Преобразование спроецированной точки, линии, многоугольника или точек растра на фрагменты, каждый из которых связывается с буфером кадра.

Rendering
Воспроизведение. Преобразование примитивов, заданных в объектных координатах, в образ в буфере кадра.

Resolution
Разрешение. Количество пикселов на единицу длины.

RGB
Конечный цвет пиксела получается за счет смешения с различной интенсивностью трех основных цветов: красного (Red), зеленого (Green) и синего (Blue).

Specular highlights
Зеркальное отражение. Световой поток, отражаемый блестящим объектом. Цвет потока обычно совпадает не с цветом самого объекта, а с цветом источника света.

Spot
Разновидность точечного источника света. Освещает объекты, попадающие в некоторый конус.

Texture
Текстура. Основной метод моделирования поверхностей наложением на них изображений. Точки текстуры называются текселами.

Transformation
Изменение координат. Последовательность математических операций над графическими примитивами, включающая сдвиг, поворот и масштабирование, для преобразования их координат из рассчитанных в системные.

Transparency
Прозрачность. Эффект, достигаемый использованием дополнительного компонента цвета, альфа-составляющей. Коэффициент альфа используется в качестве величины, отвечающей за степень прозрачности.

Triangle strip and fans
Группы связанных треугольников. В последовательности треугольников, описывающей поверхность фигуры, для треугольника задается лишь одна вершина.

Tri-linear filtering (Tri-linear MIP Mapping)
Трилинейная фильтрация. Текстуры, накладываемые на поверхность, изменяют свой вид в зависимости от изменения расстояния от объекта до положения глаза зрителя. При уменьшении объекта размер карты текстур тоже уменьшается.

Vector graphics
Векторная графика. Способ предоставления графической информации с помощью совокупных кривых, описываемых математическими формулами. Обеспечивается возможность трансформаций изображений без потери качества.

Vertex
Вершина. Точка в трехмерном пространстве.

Vertex buffer
Вершинный буфер. Область памяти, содержащая данные о вершинах.

Vertex shaders
Вершинные шейдеры. Последовательность операций, применяемых к исходным данным. Вершинные шейдеры определяют операции, проводимые над геометрическими данными вершин.

View matrix
Видовая матрица. Матрица трансформаций примитивов из координат объектов в координаты наблюдателя, системные координаты.

Wireframe
Проволочный (каркасный) режим. Представление объекта, состоящее исключительно из отрезков прямых, обычно указывающих края многоугольника.

World matrix
Мировая матрица. Матрица трансформаций, задающая преобразования в мировом пространстве трехмерной сцены.

Z-buffer
Буфер глубины. Область памяти, в которой хранятся значения Z-координаты пикселов, расстояния от него до точки наблюдения.

Z-buffering
Z-буферизация. Процесс удаления скрытых точек на основе значения глубины. Для каждого записываемого пиксела значение глубины сравнивается со значением, хранящимся в буфере, и пиксел записывается в буфер кадра, только если величина глубины меньше сохраненного значения.

ПРИЛОЖЕНИЕ 2

Описание компакт-диска и требования к компьютеру

Компакт-диск, прилагаемый к книге, содержит три каталога:

- directx8 — дистрибутив ядра DirectX 8.0a;
- 3DExplor — дистрибутив демо-версии программы импорта 3D Exploration, использованной при подготовке примеров книги *глав 9* и *10*;
- Examples — проекты примеров к книге и заголовочные файлы, необходимые для компиляции проектов.

Примеры книги предоставляются на условиях "как есть", без дополнительных гарантий, и сгруппированы по папкам, соответствующим главам книги. Большинство откомпилированных проектов можно запускать прямо с диска, некоторые же из них для своей работы потребуют копирования на устройство, предусматривающее запись. Также следует учесть, что в случае появления исключений многие примеры пытаются записать в текущий каталог текстовый файл с описанием исключения.

Помимо заголовочных файлов, в папке \Examples\DUnits содержатся файлы библиотеки D3DX, их компоновка осуществлена Алексеем Барковым (**http://clootie.narod.ru/**).

Большинство примеров двумерной графики, использующие модуль DirectDraw, нетребовательны к оборудованию и работают с удовлетворительной скоростью на компьютерах, оснащенных видеокартой с размером видеопамяти 1 Мбайт.

Для запуска примеров, использующих Direct3D, на картах, имеющих размер видеопамяти, меньший 8 Мбайт, и не поддерживающих аппаратное ускорение Direct3D, потребуется перекомпиляция их проектов, в тексте книги содержатся указания необходимых действий. Однако на видеокартах, не поддерживающих аппаратное ускорение Direct3D, удовлетворительную скорость работы можно получить лишь для самых простейших примеров.

Список литературы

1. Краснов М. В. OpenGL. Графика в проектах Delphi. — СПб.: БХВ—Санкт-Петербург, 2000. — 352 с.: ил.
2. Трухильо С. Графика для Windows: библиотека программиста (+CD). — СПб.: Питер Ком, 1998. — 320 с.: ил.
3. Томпсон Н. Секреты программирования трехмерной графики для Windows 95: Пер. с англ. — СПб.: Питер, 1997. — 352 с.: ил.

Предметный указатель

E

F

G