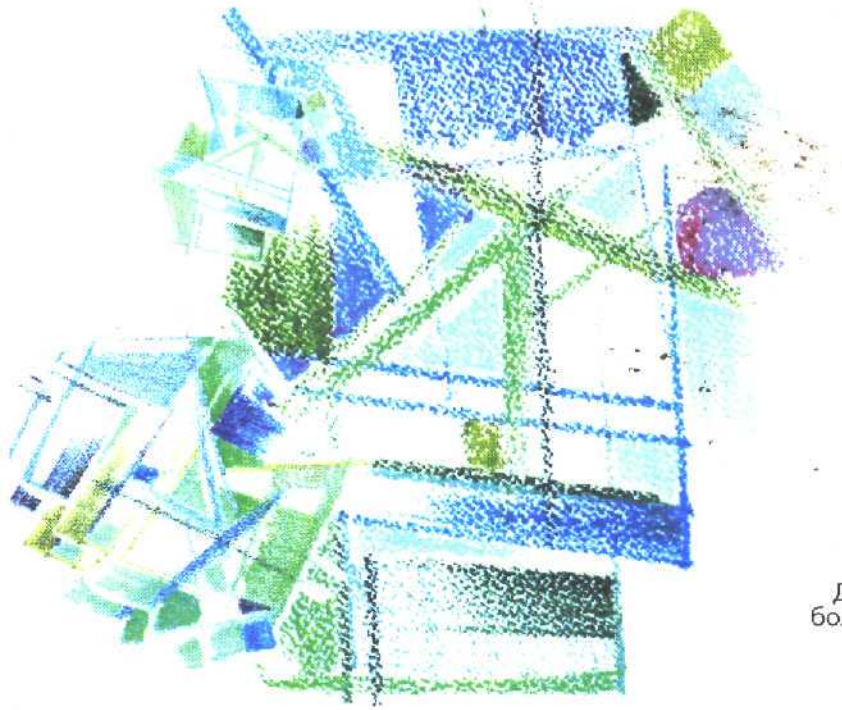
М. Краснов

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

OpenGL

ГРАФИКА В ПРОЕКТАХ DELPHI

- Полный курс по использованию библиотеки OpenGL
- Применение недокументированных команд
- Полезные советы, подсказки, приемы оптимизации



MASTER

ПРАКТИЧЕСКОЕ РУКОВОДСТВО

Дискета содержит более 200 проектов

Михаил Краснов

OpenGL

ГРАФИКА В ПРОЕКТАХ *DELPHI*

Дюссельдорф • Киев • Москва • Санкт-Петербург

УДК 681.3.06

Книги посвящена использованию стандартной графической библиотеки OpenGL в проектах Delphi. Начиная с самой минимальной программы, последовательно и подробно рассматриваются все основные принципы программирования компьютерной графики: двумерные и трехмерные построения, анимация, работа с текстурой, визуальные эффекты и др. Большое внимание уделяется вопросам оптимизации и ускорения приложений. Изложение построено на многочисленных примерах, среди которых есть и такие сложные, как многофункциональный графический редактор и CAD-система визуализации работы робототехнической установки, что облегчает усвоение материала и прививает хороший стиль программирования.

Для широкого круга программистов, интересующихся графикой

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зав. редакцией	*Наталья Таркова*
Редактор	*Ирина Агафонова*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Ангелины Лужиной*
Зав. производством	*Николай Тверских*

Краснов М. В.

OpenGL. Графика в проектах Delphi. — СПб.: БХВ-Петербург, 2002. - 352 с: ил.

ISBN 5-8206-0099-1

© М. В. Краснов, 2000

© Оформление, издательство "БХВ — Санкт-Петербург", 2000

OpenGL are registered trademarks of Silicon Graphics, Inc.; Delphi are registered trademarks of Inprise, Inc.; Windows are registered trademarks of Microsoft, Inc.
OpenGL является зарегистрированным товарным знаком Silicon Graphics; Delphi является зарегистрированным товарным знаком Inprise; Windows является зарегистрированным товарным знаком Microsoft.

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 23.11.01.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 28,38
Доп. тираж 5000 экз. Заказ 1317
"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 линия, 12

Содержание

Введение ... 7

Глава]. Подключение OpenGL ... **11**

Событие, сообщение, ссылка ... 11
Почему приложения Delphi имеют большой размер ... 15
Программирование на Delphi без VCL ... 16
Минимальная Windows-программа ... 16
Вывод с использованием функций GDI ... 20
Перехват сообщений ... 22
Работа с таймером ... 25
Работа с мышью и клавиатурой ... 26
DLL ... 27
Контекст устройства и контекст воспроизведения ... 29
Минимальная программа OpenGL ... 30
Формат пиксела ... 33
Решение проблем ... 37
Вывод на компоненты Delphi средствами OpenGL ... 38
Стили окна и вывод OpenGL ... 39
Полноэкранные приложения ... 41
Типы OpenGL ... 43
Тип *TColor* и цвет в OpenGL ... 46
Подробнее о заголовочном файле opengl.pas ... 47

Глава 2. Двумерные построения ... **50**

Точка ... 50
Команда *glScissor* ... 56
Совместный вывод посредством функций GDI и OpenGL ... 56
Отрезок ... 57
Треугольник ... 60
Многоугольник ... 65
Команда *glEdgeFlag* ... 69
Массивы вершин ... 69
Прямое обращение к пикселам экрана ... 73

Команда *glGetString* - 77
Обработка ошибок 78
Масштабирование 79
Попорот 80
Перенос 82
Сохранение и восстановление текущего положения 83
Первые шаги в пространстве 85

Глава 3. Построения в пространстве 87

Параметры вида 87
Матрицы OpenGL 98
Буфер глубины 105
Источник света 108
Объемные объекты 111
Надстройки над OpenGL 112
Quadric-объекты библиотеки glu 115
Сплайны и поверхности Безье 125
NURBS-поверхности 132
Дисплейные списки 138
Tess-объекты 144
Таймеры и потоки 149

Глава 4. Визуальные эффекты 162

Подробнее об источнике света 162
Свойства материала 165
Вывод на палитру в 256 цветов 178
Подробнее о поверхностях произвольной формы 180
Использование патчей 185
Буфер трафарета 189
Смешение цветов и прозрачность 202
Подробнее о пикселных операциях 212
Буфер накопления 218
Туман 225
Тень и отражение 228
Шаблон многоугольников 246
Текстура 250

Глава 5. Пример CAD-системы: визуализация работы робота 277

Постановка задачи 277
Структура программы 280
Модули приложения 288
Обмен данными с DLL 290
Дополнительные замечании 293

Глава 6. Создаем свой редактор 297

Выбор элементов 297
Буфер выбора 299

Вывод текста 308
Связь экранных координат с пространственными 314
Режим обратной связи 318
Трансформация объектов 323
Постановка задачи 325
Структура программы 331
Несколько советов 332

Заключение 334

Приложение 1. OpenGL в Интернете 335

Приложение 2. Содержимое прилагаемой дискеты и требования к компьютеру 337

Список литературы 339

Предметный указатель 340

Введение

Эта книга посвящена компьютерной графике, а именно тому, как использовать **OpenGL** в Delphi.

OpenGL — это стандартная библиотека для всех 32-разрядных операционных систем, в том числе и для операционной системы Windows.

OpenGL — не отдельная программа, а часть операционной системы. Это означает, что откомпилированное приложение, использующее OpenGL, не нуждается ни в каких дополнительных программах и модулях, кроме стандартных, содержащихся на любом компьютере с установленной операционной системой Windows 95 версии OSR2 и выше.

Вообще говоря, в этой книге идет речь о программировании приложений, использующих графический акселератор, однако все приводимые программы будут работать и на компьютере, не оснащенном ускорителем.

Для программистов, использующих язык C, существует множество источников, из которых можно почерпнуть сведения о том, как использовать библиотеку OpenGL, для программистов же, работающих с Delphi, таких источников крайне мало. Данная книга призвана восполнить этот недостаток информации.

В состав стандартной поставки Delphi (начиная с третьей версии) входит заголовочный файл, позволяющий строить приложения с использованием OpenGL, а также справочный файл по командам этой библиотеки. Однако инсталляция Delphi не снабжается ни одним примером по использованию OpenGL, а из файла справок новичку трудно понять, как это сделать. Поэтому основная цель книги — помочь программистам, в том числе и опытным, разобраться в этой теме.

В свое время, когда я сам учился использовать OpenGL в проектах Delphi, никаких источников, кроме набора текстов программ на языке C из файла оперативной помощи, у меня не было, и начинал я с того, что просто переносил эти программы на Delphi. Такая практика оказалась весьма полезной. Поэтому многие примеры в книге представляют собой "перевод" свободно распространяемых программ, изначально написанных на C. В текстах модулей этих примеров я оставил указание на авторов исходных версий.

В книге вы также встретите множество оригинальных программ. Кроме того, я по мере возможностей старался приводить и рекомендации профес-

сионалов, содержащиеся в учебных программах пакета OpenGL SDK (Software Design Kit) и других учебных курсов, поскольку многие читатели не имеют возможности самостоятельно перенести эти программы на Delphi и, думаю, нуждаются в некоторой помощи.

Книга задумывалась как учебник, которого мне когда-то не хватало, и именно такой я ее и написал, в соответствии со своим опытом и пристрастиями.

Хотелось бы отметить следующие особенности книги.

1. Отсутствует **описание математического аппарата компьютерной графики.** Мне не стоило бы большого труда в очередной раз переписать главы учебника по линейной алгебре, как это делается многими авторами книг по компьютерной графике. Подозреваю, что многие читатели просто пролистывают страницы книг, испещренные формулами и схемами. По моему убеждению, OpenGL позволяет нарисовать все, что угодно даже тем программистам, которые не помнят наизусть формулу транспонирования матрицы. Я не умаляю значения этих знаний, рано или поздно они вам понадобятся, но литературы по этой теме уже имеется предостаточно.

2. **Эта книга не заменит документации по OpenGL или Delphi.** Есть довольно много книг, представляющих собой просто перевод содержимого файлов оперативной помощи. Может быть, кому-то такие переводы и нужны, но только не программистам, для которых знание английского языка является необходимым условием профпригодности. Как правило, я ограничиваюсь краткими замечаниями, позволяющими понять суть излагаемой темы, а подробности всегда можно найти в справочных файлах.

3. **Главный упор делается на практические примеры.** Все проекты я предоставляю в виде исходных файлов. Примеров в книге более двухсот, и остановился я тогда, когда исчерпал объем дискеты. В среде программистов бытует мнение, что документацию следует читать только тогда, когда что-то не получается, и многие из них знакомство с новым средством разработки начинают сразу с попытки написания программы. Именно для таких специалистов книга подойдет как нельзя кстати. (При таком обилии примеров читателю, наверное, будет непросто восстановить в памяти, в каком из них содержится необходимый кусок кода. Поэтому главное назначение иллюстраций в книге — помочь в навигации по примерам.)

4. **Это учебник, а не справочник.** Материал книги я построил в расчете на то, что читатель будет знакомиться с ним последовательно. Хотя материал и разбит на тематические разделы, ко многим темам я обращаюсь многократно. Если первоначальное знакомство с какой-либо темой оказалось трудным, у читателя будет возможность разобраться с ней позднее.

5. **Книга рассчитана на новичка в области машинной графики, но не новичка в программировании на Delphi.** При изложении материала подразумевается, что читатель имеет навыки работы в Delphi, и чем увереннее он чувствует себя здесь, тем больше пользы сможет извлечь из этой книги. Не-

которые разделы, например функции API, могут показаться поначалу трудными. Однако в подавляющей части примеров особо сложные приемы программирования не используются, и они достаточно легко поддаются освоению.

В главе 1 книги описываются базовые механизмы операционной системы. Знание этих механизмов необходимо для понимания того, как построить минимальное приложение, использующее OpenGL.

Умудренные опытом программисты со стажем, пришедшие к Delphi после изрядной практики в Turbo Pascal, вряд ли найдут в этой главе что-то новое. за исключением разделов, непосредственно относящихся к OpenGL. Поэтому эту главу я рекомендую внимательно прочитать тем, кто пока не имеет достаточного опыта.

Сегодня студенты в большинстве учебных заведений начинают знакомство с программированием непосредственно с изучения Delphi, не повторяя весь путь, пройденный предыдущим поколением программистов.

Можно сказать, что система программирования Delphi явилась подлинной революцией, полностью изменившей взгляд на программирование и, в частности, на программирование для Windows. За прошедшие годы ряды программистов пополнились армией людей, способных быстро, подчас виртуозно, создать масштабное приложение, не имея особого понятия ни об архитектуре Windows, ни об основополагающих принципах работы приложения и его взаимодействия с операционной системой. В принципе, этих знаний и не требуется, чтобы создать приложение типа калькулятора или программы расчета напряжения в трубе. Однако при использовании OpenGL даже для построения минимальной программы необходимо иметь представление о базовых понятиях операционной системы.

Глава 2 посвящена примитивам OpenGL — базовым фигурам, из которых строятся объекты сцены. Собственно с этой главы и начинается рисование. Все примеры в ней плоскостные, однако пропускать ее не стоит, поскольку весь остальной материал предполагает наличие знаний и навыков, полученных при ее изучении. Материала главы достаточно для того, чтобы читатель смог построить график функции или чертеж автомобиля.

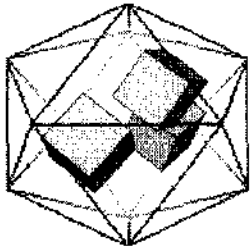
Глава 3 продолжает вводный курс по построениям в OpenGL — здесь читатель получит навыки трехмерной графики. Заканчивается глава разбором методов создания анимации. После изучения этой главы читатель сможет создавать уже довольно сложные модели, например, нарисовать автомобиль или самолет.

Глава 4 знакомит с тем, как приблизить качество изображения к фотореалистическому и как использовать OpenGL для создания специальных эффектов. Это самая важная глава книги. После усвоения ее материала читатель сможет нарисовать, например, модель Вселенной со всеми ее компонентами.

Глава 5 содержит пример построения сравнительно масштабного приложения, визуализирующего работу робототехнической установки. Здесь читатель может получить представление о том, как создавать подобные приложения и как можно использовать OpenGL для "серьезных" целей. Здесь же можно углубить знания по важнейшим понятиям операционной системы.

Глава 6 освещает некоторые дополнительные темы использования OpenGL, такие как вывод текста и выбор. Здесь же содержится еще один пример сравнительно большой программы — модельера, позволяющего из набора базовых объектов создавать сложные системы и автоматизировать подготовку кода для таких систем.

ГЛАВА 1



Подключение OpenGL

В этой главе дается представление о том, как в действительности работает Windows-приложение. Для понимания действий, требуемых для подключения OpenGL, необходимо иметь представление о важнейших понятиях операционной системы Windows, завуалированных в Delphi и напрямую обычно не используемых программистом. В качестве примера подробно разбирается минимальная программа, использующая OpenGL.

Весь последующий материал книги основывается на содержании этой главы, поэтому я рекомендую прочесть ее достаточно внимательно.

Проекты примеров записаны на дискете в каталоге Chapter1.

Событие, сообщение, ссылка

С понятием "событие" знаком каждый программист, использующий Delphi. Термин "сообщение" напрямую в концепции Delphi не используется.

Очень часто это синонимы одного и того же термина операционной системы, общающейся с приложениями (окнами) посредством посылки сигналов, называемых *сообщениями*.

Код, написанный в проекте Delphi как обработчик события `OnCreate`, выполняется при получении приложением сообщения `WM_CREATE`, сообщению `WM_PAINT` соответствует событие `OnPaint` И Т. Д.

Такие события — аналоги сообщений операционной системы — используют мнемонику, сходную с мнемоникой сообщений, т. е. сообщения начинаются с префикса "`WM_`" (Windows Message), а аналогичные события начинаются с префикса "on".

Для того чтобы операционная система могла различать окна для осуществления диалога с ними, все окна при своем создании регистрируются в опе-

рационной системе и получают уникальный идентификатор, называемый *"ссылка на окно"*. Тип этой величины в Delphi — HWND (Handle WiNDow). Синонимом термина "ссылка" является *дескриптор*.

Ссылка на окно может использоваться не только операционной системой, но и приложениями для идентификации окна, с которым необходимо производить манипуляции.

Попробуем проиллюстрировать смысл ссылки на окно на несложном примере.

Откомпилируйте минимальное приложение Delphi и начните новый проект. Форму назовите Form.2, разместите на ней кнопку. Обработчик события нажатия кнопки onclick приведите к следующему виду (готовый проект располагается на дискете в подкаталоге Ex01 каталога Chapter1):

```
procedure TForm2.Button1Click(Sender: TObject);
var
 H : HWND;                // ссылка на окно
begin
 H := FindWindow ('TForm1', 'Form1');        // ищем окно
 If H <> 0 then ShowMessage ('Есть Form1!') // окно найдено
           else ShowMessage ('Нет Form1!')   // окно не найдено
end;
```

Теперь при нажатии кнопки выдается сообщение, открыто ли окно класса, зарегистрированного в операционной системе как 'TForm1', имеющее заголовок 'Form1'. Если одновременно запустить обе наши программы, то при нажатии кнопки будет выдано одно сообщение, а если окно с заголовком 'Form1' закрыть, то другое.

Здесь мы используем функцию Findwindow, возвращающую величину типа HWND — ссылку на найденное окно либо ноль, если такое окно не найдено. Аргументы функции — класс окна и его заголовок. Если заголовок искомого окна безразличен, вторым аргументом нужно задать nil.

Итак, ссылка на окно однозначно определяет окно. Свойство Hand: е формы и есть эта ссылка, а тип THandle в точности соответствует типу HWND, так что в предыдущем примере переменную H можно описать как переменную типа THandle.

Рассмотрим подробнее некоторые выводы. Класс окна минимального приложения, созданного в Delphi, имеет значение 'TForm1', что полностью соответствует названию класса формы в проекте. Следовательно, то, как мы называем формы в проектах Delphi, имеет значение не только в период проектирования приложения, но и во время его работы. Начните новый проект, назовите форму каким-нибудь очень длинным именем и откомпилируйте проект. Сравните размер откомпилированного модуля с размером самого

первого проекта, и убедитесь, что он увеличился — вырос только за счет длинного имени класса.

Также очень важно уяснить, что, если вы собираетесь распространять какие-либо приложения, необходимо взять за правило называть формы отлично от значения, задаваемого Delphi по умолчанию. Лучше, если эти названия будут связаны по смыслу с работой вашего приложения. Так, например, головную форму в примерах этой книги я буду называть, как правило, frmGL.

Имея ссылку на окно, операционная система общается с ним путем посылки сообщений — сигналов о том, что произошло какое-либо событие, имеющее отношение именно к данному окну. Если окно имеет намерение отреагировать на событие, операционная система совместно с окном осуществляет эту реакцию.

Окно может и, не имея фокус, получать сообщения и реагировать на них. Проиллюстрируем это на примере.

Обработчик события OnMouseMove формы приведите к следующему виду (проект находится в подкаталоге ExO2):

```
procedure TForm2.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
   Y: Integer);
begin
Caption := 'x=' + IntTcStr (X; ' ', y=' + IntToStr (Y) // X, Y -
                                                    // координаты курсора
end;
```

При движении курсора мыши в заголовке формы выводятся его координаты.

Запустите два экземпляра программы и обратите внимание, что окно, не имеющее фокус, т. е. неактивное, тоже реагирует на перемещение указателя по своей поверхности и выводит в заголовке текущие координаты курсора в своей системе координат.

Имея ссылку на окно, приложение может производить с ним любые (почти) действия путем посылки ему сообщений.

Изменим код обработки щелчка кнопки (проект из подкаталога ExO3):

```
procedure TForm2.Button1Click(Sender: TObject);
var
 H : HWND;
begin
 H := FindWindow ('TForm1', 'Form1');
 If H <> 0 then SendMessage (H, WM_CLOSE, 0, 0}//закрыть найденное окно
end;
```

Если имеется окно класса 'TForm1' с заголовком 'Form1', наше приложение посылает ему сообщение WM_CLOSE — пытается закрыть окно. Для посылки

сообщения используем функцию операционной системы (функцию API) SendMessage. Функция PostMessage имеет сходное назначение, но отличается тем, что не дожидается, пока посланное сообщение будет отработано. У этих функций четыре аргумента — ссылка на окно, которому посылаем сообщение, константа, соответствующая посылаемому сообщению, и два параметра сообщения, смысл которых определяется в каждом конкретном сообщении по-своему. Параметры сообщения называются wParam и lParam. При обработке сообщения WM_CLOSE эти значения никак не используются, поэтому здесь их можно задавать произвольно.

Заметим, что одновременно могут быть зарегистрированы несколько окон класса 'TForm1', и необходимо закрыть их все. Пока наше приложение закрывает окна поодиночке при каждом нажатии на кнопку. Автоматизировать процесс можно разными способами, простейший из них используется в проекте подкаталога Ex04 и заключается в том, что вызов FindWindow заключен в цикл, работающий до тех пор, пока значение переменной H не станет равным нулю:

```
procedure TForm2.Button1Click(Sender: TObject);
var
   H : HWND;
begin
   Repeat
     H := FindWindow ('TForm1', 'Form1');
     If H <> 0 then SendMessage (H, WM_CLOSE, 0, 0)
   Until H = 0;
end;
```

Ну а как работать с приложениями, класс окна которых не известен, поскольку у нас нет (и не может быть) их исходного кода?

Для решения подобных проблем служит утилита Ws32, поставляемая с Delphi.

Например, с помощью этой утилиты я выяснил, что класс окна главного окна среды Delphi имеет значение 'TAppBuilder'. Узнав это, я смог написать проект, где делается попытка закрыть именно это окно (находится в подкаталоге Ex05).

Каждый раз я говорю именно о попытке закрыть окно, потому что приложение, получающее сообщение WM_CLOSE, может и не закрыться сразу же. Например, среда Delphi или текстовый процессор перед закрытием переспрашивают пользователя о необходимости сохранения данных. Но ведет себя приложение точно так же, как если бы команда поступала от пользователя.

В качестве следующего упражнения рассмотрите проект, располагающийся в подкаталоге Ex06, где по нажатию кнопки минимизируется окно, соответствующее минимальному проекту Delphi.

Для того чтобы минимизировать окно, ему посылается сообщение WM_SYSCOMMAND, соответствующее действию пользователя "выбор системного меню окна". Третий параметр функции sendMessage для минимизации окна необходимо установить в значение SC_MINIMIZE.

Работа с функциями API, сообщения Windows — темы весьма объемные. Пока мы рассмотрели только самые простейшие действия — закрыть и минимизировать окно.

В заключение раздела необходимо сказать, что ссылки, в зависимости от версии Delphi, соответствуют типам integer или Lcngword и описываются в модуле windows.pas.

Почему приложения Delphi имеют большой размер

Этот вопрос часто задают начинающие программисты при сравнении приложений, созданных в различных средах программирования. Действительно, минимальное приложение, созданное в различных версиях Delphi, может достигать от 170 до 290 Кбайт. Это очень большая цифра для операционной среды Windows, в компиляторах C++ она составляет порядка 40 Кбайт. Конечно, это не катастрофическая проблема, когда емкости накопителей измеряются гигабайтами, и средний пользователь, как правило, не обращает внимания на размер файла. Неудобства возникают, например, при распространении приложений по сети.

Использование пакетов значительно снимает остроту проблемы для масштабных проектов, но суммарный вес приложения и используемых пакетов все равно значителен.

Краткий ответ на вопрос, поставленный в заголовке раздела, состоит в том, что большой размер откомпилированных приложений является платой за невероятное удобство проектирования, предоставляемое Delphi. Архитектура среды программирования, RTTI, компонентный подход — все это превращает Delphi в поразительно мощный инструмент. С помощью Delphi легко написать приложения, в которых, например, динамически создаются интерфейсные элементы любого типа (класса).

Однако приложения среднего уровня не используют и не нуждаются в этих мощных возможностях. Часто ли вам встречались приложения, предлагающие пользователю перед вводом/выводом данных определиться, с помощью каких интерфейсных элементов будет осуществляться ввод или вывод, а затем разместить эти элементы на окне в удобных местах? И пользователи, и разработчики в таких средствах, как правило, не испытывают необходимости.

Однако откомпилированный модуль содержит в себе весь тот код, благодаря которому в Delphi так легко производить манипуляции со свойствами и ме-

годами объектов. К примеру, если просмотреть содержимое откомпилированного модуля, то мы встретим в нем фразы, имеющие к собственно операционной системе косвенное отношение, например, "OnKeyDown" или другие термины Delphi.

Дело здесь не в несовершенстве компилятора, компилятор Delphi оптимизирует код превосходно, дело в самой идеологии Delphi.

Очень часто после выяснения этого факта начинающие программисты задают вопрос, как избавиться от RTTI, от включения "ненужного" кода в исполняемые модули.

К сожалению, это сделать невозможно. Кардинально проблема решается только через отказ от использования библиотеки классов Delphi, т. е. программирование без VCL.

Программирование на Delphi без VCL

После того как мы прикоснулись к основополагающим терминам и понятиям операционной системы Windows "сообщение" и "ссылка на окно", мы сможем опуститься ниже уровня объектно-ориентированного программирования, VCL и RAD-технологий. Требуется это по четырем причинам.

Во-первых, приложения, активно использующие графику, чаще всего не нуждаются и не используют богатство библиотеки классов Delphi. Таким приложениям, как правило, достаточно окна в качестве холста, таймера и обработчиков мыши и клавиатуры.

Во-вторых, при программировании, основанном только на использовании функций API, получаются миниатюрные приложения. Откомпилированный модуль не отягощается кодом описания компонентов и кодом, связанным с концепциями ООП.

В-третьих, для понимания приемов, используемых для увеличения скорости воспроизведения, нужно иметь представление о подлинном устройстве Windows-программы. Например, чтобы команды перерисовки окна выполнялись быстрее, МЫ будем избегать ИСПОЛЬЗОВаниЯ МеТОДОВ Refresh И Paint формы.

В-четвертых, это необходимо для понимания действий, производимых для подключения OpenGL. Эта библиотека создавалась в эпоху становления ООП, и ее пока не коснулись последующие нововведения в технологии программирования.

Минимальная Windows-программа

Посмотрите на проект из подкаталога Ex07 — код минимальной программы Windows. Минимальной она является в том смысле, что в результате получа-

ется просто пустое окно. Также ее можно назвать минимальной программой потому, что откомпилированный модуль занимает всего около 16 Кбайт.

Приложение меньшего размера, имеющее собственное окно, получить уже никак не удастся, хотя могут быть и программы еще короче и меньше, например, такая:

```
program p;
uses Windows;
begin
 MessageBeep(mb_ok)
end.
```

Единственное, что делает эта программа, — подача звукового сигнала.

Однако вернемся к коду проекта из подкаталога Ex07. Первое, на что необходимо обратить внимание: в списке uses указаны только два модуля — windows и Messages. Это означает, что в программе используются исключительно функции API, и как следствие — длинный С-подобный код. И действительно, перенести эту и подобные ей программы на С потребует немного усилий.

Данная программа для нас крайне важна, поскольку она станет шаблоном для некоторых других примеров.

Программу условно можно разделить на две части — описание оконной функции и собственно головная программа.

В оконной функции задается реакция приложения на сообщения Windows. Именно оконную функцию необходимо дополнять кодом обработчиков сообщений для расширения функциональности приложения. Нечто подобное мы имеем в событийно-ориентированном программировании, но, конечно, в совершенно ином качестве.

В минимальной программе задана реакция на единственное сообщение `wm_Destroy`. На все остальные сообщения вызывается функция ядра операционной системы `DefWindowProc`, осуществляющая стандартную реакцию окна. Полученное окно ведет себя обычно, его можно изменять в размерах, минимизировать, максимизировать. Приложение реагирует также привычным образом, однако необходимости кодировать все эти действия нет.

В принципе, можно удалить и обработку сообщения `wm_Destroy`, но в этом случае приложение после завершения работы оставит след в памяти, съедающий ресурсы операционной системы.

Значение переменной-результата обнуляется в начале описания оконной функции для предотвращения замечания компилятора о возможной неинициализации переменной.

Головная программа начинается с того, что определяются атрибуты окна. Термин "структура", перешедший в Delphi из языка С, соответствует терми-

ну "запись". Термин "класс окна" имеет к терминологии объектно-ориентированного программирования скорее приближенное, чем непосредственное отношение.

Значения, задаваемые полям структуры, определяют свойства окна. В этой программе я задал значения всем полям, что, в принципе, делать не обязательно, мы обязаны указать адрес оконной функции, а все остальные значения можно брать по умолчанию. Однако в этом случае окно будет выглядеть или вести себя необычно. Например, при запуске любого приложения операционная система задает курсор для него в виде песочных часов, и если мы не станем явно задавать вид курсора в классе окна, курсор окна приложения так и останется в виде песочных часов.

После заполнения полей класса окна его необходимо зарегистрировать в операционной системе.

В примере я анализирую результат, возвращаемый функцией `RegisterClass`. Это также делать не обязательно, невозможность регистрации класса окна — ситуация крайне редкая при условии корректного заполнения его полей.

Следующие строки можно интерпретировать как "создание конкретного экземпляра на базе зарегистрированного класса". Очень похоже на ООП, но схожесть эта весьма приблизительная и объясняется тем. что первая версия Windows создавалась в эпоху первоначального становления концепции объектно-ориентированного программирования.

При создании окна мы уточняем его некоторые дополнительные свойства — заголовок, положение, размеры и прочее. Значения этих свойств задаются аргументами функции `CreateWindow`, возвращающей внимание, величину типа `HWND` — ту самую ссылку на окно, что в Delphi называется `Handle`.

После создания окна его можно отобразить — вызываем функцию `ShowWindow`. Как правило, окно сразу после этого перерисовывают вызовом функции `UpdateWindow`— действие тоже необязательное, но для корректной работы приложения удалять эту строку нежелательно.

Далее следует цикл обработки сообщений, наиважнейшее место в программе, фактически это и есть собственно работа приложения. В нем происходит диалог приложения с операционной системой: извлечение очередного сообщения из очереди и передача его для обработки в окопную функцию.

Как уже говорилось, функции API и сообщения — темы очень обширные, и я не ставлю целью исчерпывающе осветить эти темы. В разумных объемах я смогу изложить только самое необходимое, а более подробную информацию можно получить в оперативной помощи Delphi.

К сожалению, версии Delphi 3 и 4 поставляются с системой помощи, не настроенной должным образом для получения информации по функциям API и командам OpenGL. Если судить по содержанию помощи, то может сложиться впечатление, что эти разделы в ней вообще отсутствуют.

Можно либо настраивать справочную систему самостоятельно, либо, что я и предлагаю, пользоваться контекстной подсказкой — для получения сведений по любой функции API достаточно поставить курсор на соответствующую строку и нажать клавишу <F1>.

Замечание

В пятой версии Delphi система помощи настроена вполне удовлетворительно, а сами файлы помощи обновлены.

Кстати, обращаю внимание, что описания функций приводятся из файлов фирмы Microsoft, предназначенных главным образом для программистов, использующих язык C, поэтому полученную информацию необходимо интерпретировать в контекст Delphi.

Код минимальной программы я подробно прокомментировал, так что надеюсь, что все возникшие вопросы вы сможете разрешить с помощью моих комментариев.

Замечание

Итак, в приложениях Windows на самом деле управление постоянно находится в цикле обработки сообщений, событийно-ориентированная схема как таковая отсутствует. При получении окном очередного сообщения управление передается оконной функции, в которой задается реакция на него, либо вызывается функция API `DefWindowProc` для реакции, принятой по умолчанию.

Приведенный пример лишь отдаленно напоминает то, что мы имеем в Delphi — событийно-ориентированное программирование, основанное на объектах. Конечно, совершить путь, обратный исторически пройденному, нелегко. Отказаться от библиотеки VCL при написании программ на Delphi для многих оказывается непосильным. Вознаграждением здесь может стать миниатюрность полученных программ: как мы видим, минимальная программа уменьшилась минимум в десять раз, быстрее загружается, выполняется и быстрее выгружается из памяти.

Такой размер нелегко, а порой и невозможно получить в любом другом компиляторе, кроме как в Delphi. К тому же проекты, в списке uses которых стоят только windows и Messages, компилируются еще стремительнее, несмотря на устрашающую массивность кода.

А сейчас посмотрите проект из подкаталога Ex08, где окно дополнено кнопкой и меткой. В коде появились новые строки, а простейшие манипуляции, например, изменение шрифта метки, еще потребуют дополнительных строк.

Подобный обширный код обычно обескураживает новичков, поэтому я не буду злоупотреблять такими примерами и ограничусь только самыми необходимыми для нас темами — как создать обработчик мыши, клавиатуры и таймера.

Я не буду заставлять вас писать все программы таким изнурительным способом, нам просто нужно иметь представление о работе базовых механиз-

мов, чтобы лучше понимать, что делает за нас Delphi и что необходимо сделать, чтобы подключить OpenGL к нашим проектам.

Вывод с использованием функций GDI

В первом разделе мы рассмотрели, как, получив ссылку на чужое окно, можно производить с ним некоторые действия, например, закрыть его.

Точно так же, если необходимо нарисовать что-либо на поверхности чужого окна, первым делом нужно получить ссылку на него.

Для начала попробуем рисовать на поверхности родного окна.

Разместите еще одну кнопку, обработку щелчка которой приведите к следующему виду (проект из подкаталога Ex09):

```
procedure TForm2.Button2Click(Sender: TObject);
var
   dc : HDC;                          // ссылка на контекст устройства
begin
   dc := GetDC (Handle);              // задаем значение ссылки
   Rectangle (dc, 10, 10, 110, 110); // рисуем прямоугольник
   ReleaseDC (Handle, dc);   // освобождение ссылки
   DeleteDC (dc);           // удаление ссылки, освобождение памяти
end;
```

Запустите приложение. После щелчка на добавленной кнопке на поверхности окна рисуется квадрат.

Для рисования в этом примере используем низкоуровневые функции вывода Windows, так называемые функции GDI (Graphic Device Interface, интерфейс графического устройства). Эти функции требуют в качестве одного из своих аргументов ссылку на контекст устройства.

Тип такой величины — `HDC` (Handle Device Context, ссылка на контекст устройства), значение ее можно получить вызовом функции API `GetDC` с аргументом-ссылкой на устройство вывода. В нашем примере в качестве аргумента указана ссылка на окно.

После получения ссылки на контекст устройства обращаемся собственно к функции, строящей прямоугольник. Первым аргументом этой функции является ссылка на контекст устройства.

После использования ссылки ее необходимо освободить, а в конце работы приложения — удалить для освобождения памяти.

Поставленная "клякса" будет оставаться на окне формы при изменении размеров окна и исчезнет только при его перерисовке, для чего можно, например, минимизировать окно формы, а затем вновь его развернуть. Исчезновение квадрата после такой операции объясняется тем, что за перерисовку окна отвечает его собственный обработчик.

Теперь попробуем порисовать на поверхности чужого окна, для чего изменим только что написанный кол (готовый проект находится в подкаталоге Ex10):

```
procedure TForm2 .Button2Click(Sender: TObject);
var
 dc : hDC;
 Window : HWND;
begin
 Window := FindWindow ('TForm1', 'Form1');
 If Window <> 0 then begin                    // окно найдено
    dc := GetDC (Window);                     // ссылка на найденное окно
    Rectangle (dc, 10, 10, 110, 110.) ;       // квадрат на чужом окне
    ReleaseDC (Window, dc);                   // освобождение ссылки
    DeleteDC (dc);                            // удаление ссылки
 end;
end;
```

Теперь при щелчке на кнопке, если в системе зарегистрировано хотя бы одно окно класса 'TForm1' с заголовком 'Form1', вывод (рисование квадрата) будет осуществляться на него.

Запустите параллельно откомпилированные модули минимального и только что созданного приложений. При щелчке на кнопке квадрат рисуется на поверхности чужого окна.

Замечу, что если закрыть Project1.exe и загрузить в Delphi соответствующий ему проект, то при щелчке на кнопке прямоугольник будет рисоваться на поверхности окна формы, что будет выглядеть необычно.

Этот эксперимент показывает, что окна, создаваемые Delphi во время проектирования, такие же равноправные окна, как и любые другие, т. е. они регистрируются в операционной системе, идентифицируются, и любое приложение может иметь к ним доступ. Если попытаться минимизировать окно класса 'TForm1', окно формы будет отрабатывать эту команду точно так же, как полученную от пользователя.

Замечание

Следует обратить внимание на то, что мы не можем рисовать на поверхности вообще любого окна. Например, не получится, поменяв имя класса окна на 'TAppBuilder', поставить "кляксу" на главном окне среды Delphi.

Окно со значением ссылки, равным нулю, соответствует окну рабочего стола. В примере Ex11 я воспользовался этим, чтобы нарисовать квадратик на рабочем столе:

```
procedure TForm2.Button2Click(Sender: TObject);
var
   dc : HDC;
```

```
begin
   dc := GetDC (0); // получаю ссылку на рабочий стол
   Rectangle (dc, 10, 10, 110, 110);
   ReleaseDC (Handle, dc);
   DeleteDC (DC);
end;
```

Во всех примерах этого раздела я для вывода использовал функции GDI потому, что если для вывода на родном окне Delphi и предоставляет удобное средство — методы свойства формы canvas, то для вывода на чужом окне мы этими методами воспользоваться не сможем в принципе.

Разберем основные детали вывода с помощью функций GDI. В проекте из подкаталога Ex12 оконная функция минимальной программы дополнилась обработкой сообщения WM_PAINT. ВЫВОД заключен между строками с вызовом функций BeginPaint и EndPaint, первая из которых возвращает ссылку на контекст устройства, т. е. величину типа нос, требуемую для функций вывода **GDI.** Еще раз повторю, что после использования ссылки ее необходимо освободить и удалить по окончании работы — это необходимо для корректной работы приложения.

Смысл ссылки на контекст устройства мы подробно обсудим немного позже.

Остальные строки кода подробно прокомментированы, так что, надеюсь, особых вопросов не вызовут, разве только начинающих может в очередной раз поразить обширность кода, возникающая при отказе от удобств, предоставляемых библиотекой классов Delphi (которая выполняет за программиста всю черновую работу и превращает разработку программы в сплошное удовольствие).

Перехват сообщений

Большинство событий формы и компонентов являются аналогами соответствующих сообщений операционной системы.

Конечно, не все сообщения имеют такие аналоги, поскольку их (сообщений) очень много, несколько сотен. С каждой новой версией Delphi у формы появляются все новые и новые свойства и события, благодаря чему программировать становится все удобнее, по зато размеры откомпилированного приложения все растут и растут.

Замечание

В угоду программистам, до сих пор использующим третью версию Delphi для получения сравнительно небольших по объему исполняемых модулей, все примеры данной книги я разрабатывал именно в этой версии, но все они прекрасно компилируются и в более старших версиях.

У программистов всегда будет возникать потребность обрабатывать сообщения, не имеющие аналогов в списке событий, либо самостоятельно перехватывать сообщения, для которых есть аналоги среди событий формы и компонентов. Как увидим ниже, сделать это несложно.

Для начала обратимся к проекту из подкаталога Ex13, где мы опять будем программировать без VCL. Задача состоит в том, чтобы при двойном щелчке левой кнопкой мыши выводились текущие координаты указателя. Прежде всего обратите внимание, что в стиль окна добавилась константа `cs_DblClks`, чтобы окно могло реагировать на двойной щелчок, а оконная функция дополнилась обработкой сообщения `wm_LButtonDblClk`, в которой выводятся координаты курсора.

Теперь создадим обработчик этого же сообщения в проекте Delphi обычного типа (проект располагается в подкаталоге Ex14).

Описание класса формы я дополнил разделом protected, в который поместил forward-описание соответствующей процедуры:

```
procedure MesDblClick (var MyMessage : TWMMouse); message
                                                  wm_LButtonDblClk;
```

Замечание

Как правило, перехватчики сообщений для повышения надежности работы приложения описываются в блоке `protected`.

Имя процедуры я задал таким, чтобы не появлялось предупреждение компилятора о том, что я перекрываю соответствующее событие формы.

Служебное слово `message` указывает на то, что процедура будет перехватывать сообщение, мнемонику которого указывают за этим словом. Тип аргумента процедуры-перехватчика индивидуален для каждого сообщения. Имя аргумента произвольно, но, конечно, нельзя брать в качестве имени служебное слово message.

Пожалуй, самым сложным в процессе описания перехвата сообщений является определение типа аргумента процедуры, здесь оперативная помощь оказывается малополезной. В четвертой и пятой версиях Delphi инспектор кода облегчает задачу, но незначительно.

Чтобы решить эту задачу для сообщения `wm_LButtonDblClk`, я просмотрел все вхождения фразы "LButtonDblClk" в файле messages.pas и обнаружил строку, подсказавшую решение:

```
TWMLButtonDblClk = TWMMouse;
```

В этом же файле я нашел описание структуры `TWMMouse`, чем и воспользовался при кодировании процедуры `MesDblClick` для получения координат

курсора. Обратите внимание, что здесь не пришлось самостоятельно разбивать по словам значение параметра, как в предыдущем проекте.

Итак, в рассматриваемом примере перехватывается сообщение "двойной щелчок левой кнопки мыши". Событие `DblClick` формы наступает точно в такой же ситуации. Выясним, какая из двух процедур, перехватчик сообщения или обработчик события, имеет преимущество или же они равноправны. Создайте обработчик события `OnDblClick` формы — вывод любого тестового сообщения (можете воспользоваться готовым проектом из подкаталога Ex15). Запустите проект, дважды щелкните на форме. Процедура-перехватчик среагирует первой и единственной, до обработчика события очередь не дойдет.

Замечание

Перехватчики сообщений приходится писать в тех случаях, когда в списке событий нет аналога нужного нам сообщения, а также тогда, когда важна скорость работы приложения. Обработка сообщений происходит быстрее обработки событий, поэтому именно этим способом мы будем пользоваться в приложениях, особенно требовательных к скорости работы.

В проекте из подкаталога Ex16 создан обработчик сообщения `wm_paint` — перерисовка окна:

```
protected
    procedure WMPaint(var Msg: TWMPaint); message WM_PAINT;
...
procedure TForm1.WMPaint(var Msg: TWMPaint);
var
  ps : TPaintStruct;
begin
  BeginPaint(Handle, ps);
  Rectangle (Canvas.Handle, 10, 10, 100, 100);
  EndPaint(Handle, ps);
end;
```

Строки BeginPaint и EndPaint присутствуют для более корректной работы приложения, при их удалении появляется неприятное мерцание при изменении размеров окна. Обратите внимание на функцию построения прямоугольника: я воспользовался тем, что свойство `Canvas.Handle` и есть ссылка на контекст устройства, соответствующая окну формы.

Точно так же, как перехватчики сообщений предпочтительнее обработчиков событий, использование непосредственно ссылок на окно и ссылок на контекст устройства предпочтительнее использования их аналогов из мира ООП.

Запомните этот пример, таким приемом мы будем пользоваться очень часто.

Работа с таймером

В этом разделе мы разберем, как использовать таймер, основываясь только на функциях **API.** Поскольку вы наверняка умеете работать с компонентом класса TTimer, вам легко будет уяснить, как это делается на уровне функций API.

Посмотрите простой пример, располагающийся в подкаталоге Ex17, где с течением времени меняется цвет нарисованного кружочка. Первым делом замечаем, что блок описания констант дополнился описанием идентификатора таймера, в качестве которого можно взять любое целое число:

```
const
  AppName = 'WinPaint';
  id_Timer = 100; // идентификатор таймера
```

Идентифицировать таймер необходимо потому, что у приложения их может быть несколько. Для включения таймера (то, что в привычном антураже соответствует Timerl. Enabled := True) вызывается функция API SetTimer, где задается требуемый интервал таймера:

```
SetTimer (Window, id_Timer, 200, nil) ; // установка таймера
```

Сделал я это перед входом в цикл обработки сообщений, но можно и при обработке сообщения WM_CREATE. Кстати, самое время сказать, что это сообщение обрабатывается в обход цикла обработки сообщений, поэтому таймер, включенный в обработчике WM_CREATE, начнет работать раньше. Оконная функция дополнилась обработкой сообщения, соответствующего такту таймера:

```
wm_Timer :   InvalidateRect (Window, nil, False);
```

Если в приложении используется несколько таймеров, необходимо отделять их по значению идентификатора, передаваемому в wParam.

В моем примере каждые 200 миллисекунд окно перерисовывается вызовом функции **API** InvalidateRect. Запомните этот прием, потом мы не раз будем его использовать. Изменение цвета кружочка достигается тем, что при каждой перерисовке объект "кисть" принимает новое значение:

```
Brush:=CreateSolidBrush (RGB (random (255) , random (255) , r andom (255) ) ) ;
```

Как всегда в Windows, созданные объекты должны по окончании работы удаляться, дабы не поглощали ресурсы. Для удаления таймера вызываем функцию KillTimer в обработчике сообщения wm_Destroy:

```
KillTimer (Window, id_Timer);
```

Как видим, работать с таймером, используя только функции API, совсем не сложно. Компонент Delphi TTimer основывается на функциях и сообщениях, которые мы только что рассмотрели.

Работа с мышью и клавиатурой

Как обработать двойной щелчок левой кнопки мыши, опираясь на сообщения, мы рассмотрели выше в разделе "Перехват сообщений" данной главы.

Проект из подкаталога Ex18 является примером на обработку остальных сообщений, связанных с мышью. При нажатой левой кнопки мыши за указателем остается след. Оконная функция дополнилась обработчиками сообщений wm_LButtonDown, wm_LButtonUp И wm_MouseMove. Для определения координат курсора пользуемся тем, что поле lParam подобных сообщений содержит эти самые координаты.

```
wm_Create  :  Down :- False;
wm_LButtonDown, wm_LButtonUp : Down : = not Down;
wm_MouseMove : begin
                 If Down then begin
                 xpos :- LoWord f LParam) ;
                 ypos := HiWord ( LParam);
                 InvalidateRect(Window, nil, False) ;
                 end;
                 end;
wm_Paint:    begin
             If Down then begin
             dc := BeginPaint (Window, MyPaint);
             Ellipse (dc, xPos, yPos, xPos + 2, yPos - 2);
             EndPaint (Window, MyPaint);
             ReleaseDC (Window, dc) ;
             end;
             end;
```

Обратите внимание, что здесь при движении мыши с удерживаемой кнопкой окно перерисовывается точно так же, как и в предыдущем примере с таймером.

Последнее, что мы рассмотрим в данном разделе и что обязательно потребуется в дальнейшем — это обработка клавиатуры.

Как обычно, обратимся к несложной иллюстрации — проекту из подкаталога Ex19. Оконная функция дополнилась обработчиком соответствующего сообщения:

```
wm_Char:  // анализ нажатой клавиши
 case wParam of
 $58, $78 : If HiWord (GetKeyState (vk_Shift)) = 0 { Shift \
    then MessageBox(Window, 'X', 'Нажата клавиша', MB_OK]
    else MessageBox (Window, 'X вместе с Shift', 'Нажата клавиша', MB    _OK);
 end; // wm_char
```

При нажатии клавиши 'X' выводится сообщение, в котором указано, нажата ли одновременно клавиша <Shift>. Я использовал шестнадцатеричное представление кода клавиши, но, конечно, можно использовать и десятичное. Надеюсь, здесь не требуются особые пояснения, и мы сможем использовать этот код в качестве шаблона в будущих проектах.

DLL

Файлы DLL (Dynamic Link Library, библиотека динамической компоновки) являются основой программной архитектуры Windows и отличаются от исполняемых файлов фактически только заголовком.

Замечание

Но это не означает, что если переименовать DLL-файл, то он станет исполняемым: имеется в виду заголовочная информация файла.

Для загрузки операционной системы необходимо запустить файл win.com, имеющий размер всего 25 Кбайт. Как легко догадаться, в файл такого размера невозможно поместить код, реализующий всю ту гигантскую работу, которая производится по ходу выполнения любого приложения. Этот файл является загрузчиком ядра операционной системы, физически размещенным в нескольких DLL-файлах.

Помимо кода, DLL-файлы могут хранить данные и ресурсы. Например, при изменении значка (ярлыка) пользователю предлагается на выбор набор значков из файла SHELL32.DLL.

Как мы уже знаем, для создания любой программы Windows, имеющей собственное окно, в проекте необходимо подключать как минимум два модуля: windows и Messages. Первый из этих файлов содержит прототипы функций API и GDI, Посмотрим на прототип одной из них:

```
function CreateDC; external gdi32 name 'CreateDCA';
```

Здесь величина gdi32 — константа, описанная в этом же модуле:

```
const
  gdi32----'gdi32.dll';
```

Таким образом, функция CreateDC физически размещена в файле gdi32.dll и каждое приложение, использующее функции GDI, обращается к этой библиотеке.

Приблизительно так же выглядят прототипы остальных функций и процедур, но указываемые в прототипе имена библиотек индивидуальны для каждой из них.

Обратите внимание, что в этом же файле находится описание константы opengl32.

Использование DLL, в частности, позволяет операционной системе экономить память.

Приложение не умеет ни создавать окно, ни выводить в него информацию и не содержит кода для этих действий. Все запущенные приложения *(клиенты)* передают соответствующие команды файлу gdi32.dll *(серверу),* который отрабатывает их, осуществляя вывод на устройство, ссылку на контекст которого передается в качестве первого аргумента функции. При этом клиентов обслуживает единственная копия сервера в памяти.

Помимо важности DLL как основы программной архитектуры операционной системы, представление о динамических библиотеках необходимо иметь каждому разработчику программных систем. Многие программные системы строятся по следующему принципу: основной код и данные размещаются в динамических библиотеках, а исполняемому файлу отводится роль загрузчика. Подробнее о такой организации мы поговорим в главе 5.

Библиотека OpenGL физически также размещена в виде двух DLL-файлов: opengl23.dll и glu32.dll. Первый из этих файлов и есть собственно библиотека OpenGL. Назначение его — осуществление взаимодействия с акселератором или программная эмуляция ускорителя за счет центрального процессора. Поддержка 3D-акселерации осуществляется с помощью *полного (устанавливаемого) клиентского драйвера* (Installable Client Driver, ICD) и *мини-драйвера* (Mini-Client Driver, MCD).

Библиотека OpenGL реализована по клиент-серверной схеме, т. е. ее одновременно может использовать несколько приложений при единственной копии сервера в памяти или вообще при удаленном расположении сервера (сервер в принципе может располагаться и не на компьютере клиента).

Иногда программисту, как, например, в случае с OpenGL, бывает необходимо просмотреть список функций и процедур, размещенных в конкретном файле DLL. Для этого можно воспользоваться либо утилитой быстрого просмотра, поставляемой в составе операционной системы, либо утилитой tdump.exe, поставляемой в составе Delphi.

Для вызова утилиты быстрого просмотра установите курсор мыши на значок нужного файла и нажмите правую кнопку. В появившемся меню должен присутствовать пункт "Быстрый просмотр", если этого пункта нет, го требуется установить компонент операционной системы "Быстрый просмотр".

Для использования утилиты tdump скопируйте ее и необходимый dll-файл в отдельный каталог. Запустите его с параметрами <имя анализируемого файла> и <имя файла-результата>, например:

```
TDUMP.EXE opengl32.dll opengl.txt
```

В файле opengl.txt будет выдана информация, извлеченная утилитой из заголовка библиотеки, аналогичная той, что выводится утилитой быстрого просмотра. Информация группируется по секциям, среди которых наиболее

часто программисту требуется секция экспортируемых функций для уточнения содержимого библиотеки.

Итак, чаше всего DLL представляет собой набор функций и процедур. Как говорится в справке Delphi по DLL, "динамические библиотеки являются идеалом для многоязыковых проектов". Это действительно так: при использовании OpenGL совершенно безразлично, в какой среде созданы сама библиотека и вызывающие ее модули.

Контекст устройства и контекст воспроизведения

Мы уже знаем, что ссылка на контекст устройства — это величина типа нос. Для ее получения можно вызвать функцию `GetDC`, аргументом которой является ссылка на нужное окно.

Ссылке на контекст устройства соответствует свойство `Canvas.Handle` формы, принтера и некоторых компонентов Delphi.

Каков же все-таки смысл контекста устройства, если он и так связан с однозначно определенным объектом — окном, областью памяти или принтером, и зачем передавать дополнительно какую-то информацию об однозначно определенном объекте?

Для ответа на эти вопросы обратим внимание на замечательное свойство вывода в Windows, состоящее в том, что одними и теми же функциями осуществляется вывод на различные устройства.

Строки программы

```
Form1.Canvas.Ellipse(0, 0, 100, 100);
```

И

```
Printer.BeginDoc;
Printer.Canvas.Ellipse(0, 0, 100, 100);
Printer.EndDoc;
```

рисуют один и тот же круг как на поверхности формы, так и в распечатываемом документе, т. е. на различных устройствах, причем если мы будем выводить разноцветную картинку на монохромный принтер, он справится с этой задачей, передавая цвета оттенками серого.

Даже если мы рисуем только на поле формы, мы имеем дело с различными устройствами — нам неизвестно, какова графическая плата компьютера и каковы характеристики текущей установки настроек экрана. Например, имея в своем распоряжении более 16 миллионов цветов, приложение не заботится об отображении этой богатой палитры на экране, располагающем всего 256 цветами. Такие вопросы приложение перекладывает на плечи опе-

рационной системы, решающей их посредством использования драйверов устройств.

Для того чтобы воспользоваться функциями воспроизведения Windows, приложению необходимо только указать ссылку на контекст устройства, содержащий средства и характеристики устройства вывода.

Справочный файл Win32 Programmer's Reference фирмы Microsoft, поставляемый в составе Delphi, о контексте устройства сообщает следующее:

"Контекст устройства является структурой, которая определяет комплект графических объектов и связанных с ними атрибутов и графические режимы, влияющие на вывод. Графический объект включает в себя карандаш для изображения линии, кисть для закраски и заполнения, растр для копирования или прокрутки частей экрана, палитру для определения комплекта доступных цветов, области для отсечения и других операций, маршрут для операций рисования".

В OpenGL имеется аналогичное ссылке на контекст устройства понятие *ссылка на контекст воспроизведения.*

Графическая система OpenGL, как и любое другое приложение Windows (хоть и размещенное в DLL), также нуждается в ссылке на устройство, на которое будет осуществляться вывод. Это специальная ссылка на контекст воспроизведения — величина типа HGLRC (Handle openGL Rendering Context, ссылка на контекст воспроизведения OpenGL).

Замечание

Контекст устройства Windows содержит информацию, относящуюся к графическим компонентам GDI, а контекст воспроизведения содержит информацию, относящуюся к OpenGL, т. е. играет такую же роль, что и контекст устройства для GDI.

В частности, упомянутые контексты являются хранилищами состояния системы, например, хранят информацию о текущем цвете карандаша.

Минимальная программа OpenGL

Рассмотрев основные вопросы функционирования приложения и его взаимодействия с операционной системой, мы можем перейти к изучению собственно OpenGL. Заглянем в подкаталог Ex20. содержащий проект минимальной программы, использующей OpenGL. В программе с помощью команд OpenGL окно формы окрашивается в голубоватый цвет.

Во-первых, обратите внимание на то, что список uses дополнен модулем OpenGL — это программист должен сделать сам.

Раздел private описания класса формы содержит строку

```
hrc: HGLRC;  // ссылка на контекст воспроизведения
```

Смысл этой величины мы рассмотрели в предыдущем разделе.

Обработчик события OnCreate формы содержит следующие строки:

```
SetDCPixelFormat(Canvas.Handle);          //задаем формат пиксела
hrc := wglCreateContext(Canvas.Handle);//создаем контекст воспроизведения
```

Первая строка — обращение к описанной в этом же модуле пользовательской процедуре, задающей формат пиксела:

```
procedure SetDCPixelFormat (hdc : HDC);
var
 pfd : TPixelFormatDescriptor;
 nPixelFormat : Integer;
begin
 FillChar (pfd, SizeOf (pfd), 0);
 nPixelFormat := ChoosePixelFormat (hdc, @pfd);
 SetPixelFormat (hdc, nPixelFormat, @pfd);
end;
```

По поводу формата пиксела мы подробнее поговорим в следующем разделе.

Во второй строке обработчика OnCreate, как ясно из комментария, задается величина типа HGLRC, т. е. создается контекст воспроизведения. Аргументом функции wglCreateContext является ссылка на контекст устройства, на который будет осуществляться вывод. Сейчас устройством вывода служит окно формы. Для получения этого контекста OpenGL необходима величина типа HDC. Здесь, как и во многих других примерах, мы используем факт, что Canvas.Handle и есть ссылка на контекст устройства, связанная с окном формы.

Поскольку это первая функция, имеющая непосредственно отношение к OpenGL, то я немного отвлекусь на некоторые общие пояснения.

Как уже говорилось, только начиная с пятой версии Delphi поставляется с системой помощи, удовлетворительно настроенной с точки зрения получения справок по командам OpenGL и функциям API, в предыдущих версиях ее вроде как и нет. Однако на самом деле такая помощь доступна и в ранних версиях, и самый простой способ ее получения — контекстная подсказка. По командам OpenGL справки мы будем получать точно так же, как и по функциям API, т. е. если вам необходима более подробная информация, например, о функции wglCreateContext, то установите курсор на строку с этой функцией и нажмите клавишу <F1>.

Функция wglCreateContext физически размещается в файле opengl32.dll, а прототип ее находится в файле windows.pas. В этот файл также помещены прототипы всех функций и процедур, имеющих отношение к реализации OpenGL под Windows, а прототипы собственно команд OpenGL расположены в файле opengl.pas.

Функции и процедуры, имеющие отношение только к Windows-версии OpenGL, обычно имеют приставку `wgl`, как, например, `wglCreateContext`, но могут и не иметь такой приставки, как, например, `SwapBuffers`. Собственно команды OpenGL имеют приставки `gl` или `glu` в зависимости от размещения в библиотеках opengl32.dll и glu32.dll, соответственно.

Итак, контекст воспроизведения создан, и теперь можно осуществлять вывод командами OpenGL. Обработка события `OnPaint` выглядит следующим образом:

```
wglMakeCurrent (Canvas.Handle, hrc); // установить контекст
glClearColor (0.5, 0.5, 0.75, 1.0);  // цвет фона
glClear (GL_COLOR_BUFFER_BIT);       // очистка буфера цвета
wglMakeCurrent (0,0);                // освободить контекст
```

Первая строка делает контекст воспроизведения текущим, т. е. занимает его для последующего вывода. Далее задаем цвет фона. Следующую строку будем понимать как очистку экрана и окрашивание его заданным цветом. После работы освобождаем контекст.

Замечание

Согласно справке, для освобождения контекста воспроизведения оба параметра должны быть установлены в NULL, НО ХОТЯ компилятор и пропустит такие значения, во время выполнения получим ошибку "Invalid variant type conversion", так что будем всегда для освобождения контекста задавать эти значения нулевыми.

Обработка события OnDestroy формы состоит из одной строки:

```
wglDeleteContext(hrc);
```

Тем самым мы по завершении работы приложения удаляем контекст воспроизведения, освобождая память.

Замечание

Очень важно запомнить, что процедуры и функции, имена которых начинаются на `gl` или `glu`, т. е. команды OpenGL, имеют *какой-либо* результат только при установленном контексте воспроизведения.

Вернемся к команде `glClearColor`, определяющей цвет фона. У нее четыре аргумента, вещественные числа, первые три из которых задают долю красного, зеленого и синего в результирующем цвете. О четвертом аргументе мы поговорим в четвертой главе, здесь я его значение задал равным единице. Можете варьировать это значение произвольно, в данном примере это никак не скажется, так что пока можете просто не обращать внимания на этот аргумент. Согласно справке, все четыре аргумента функции `glClearColor` имеют тип `GLclampf`, соответствующий вещественным числам в пределах от нуля до единицы. О типах OpenGL подробнее поговорим ниже.

Теперь обратитесь к проекту из подкаталога Ex21, представляющему собой еще один вариант минимальной программы Delphi, использующей OpenGL. но без VCL. Надеюсь, читатель, вы уже можете ориентироваться в таких программах. Отличие от предыдущего примера состоит в том. что приходится самому описывать все те действия, которые при обычном подходе выполняет за нас Delphi, т. е. в данном случае "вручную" создавать ссылку па контекст устройства, устанавливать, освобождать и удалять ее.

Если вы используете Delphi версии три или четыре, вы, возможно, столкнетесь с одной небольшой проблемой. Если запускать проекты, использующие OpenGL, под управлением среды Delphi, программа может случайным образом аварийно завершаться. Оборот "случайным образом" здесь я употребил постольку, поскольку один и тот же проект может привести к аварийному завершению, а может и работать вполне успешно.

Я сталкивался с этой проблемой на компьютерах с различной конфигурацией и с различными версиями операционной системы, и, по-видимому, она связана с некорректным взаимодействием среды Delphi с драйверами. Если подобная проблема возникла и у нас, я рекомендую просто не запускать под управлением среды проекты, использующие OpenGL, а запускать собственно откомпилированные модули. В пятой версии Delphi такая ситуация не возникала, так что, по-видимому, этот недостаток разработчиками выявлен и устранен.

Формат пиксела

Напомню, ссылка на контекст устройства содержит характеристики устройства и средства отображения. Упрощенно говоря, получив ссылку на контекст устройства, мы берем в руки простой либо цветной карандаш или кисть с палитрой в миллионы оттенков.

Сервер OpenGL, прежде чем приступать к работе, также должен определиться, на каком оборудовании ему придется работать. Это может быть скромная персоналка, а может быть и мощная графическая станция.

Прежде чем получить контекст воспроизведения, сервер OpenGL должен получить детальные характеристики используемого оборудования. Эти характеристики хранятся в специальной структуре, тип которой — `TPixelFormatDescriptor` (описание формата пиксела). Формат пиксела определяет конфигурацию буфера цвета и вспомогательных буферов.

Наберите в тексте модуля фразу "PixelFormatDescriptor", нажмите клавишу <F1>, и вы получите подробную информацию об этом типе. Обращаю внимание: мы видим раздел справки Microsoft, рассчитанной на программистов, использующих C или C++, поэтому описание содержит термины и стилистику именно этих языков. По традиции Delphi имена типов начинаются с префикса T. НО нам не удастся НАЙТИ ПОМОЩЬ ПО ТерМИНу `TPixelFormatDescriptor`.

К сожалению, это не единственное неудобство, которое придется испытать. Например, если мы заглянем в файл windows.pas и найдем описание записи TPixelFormatDescriptor, то обнаружим, что в файле помощи не указаны некоторые константы, имеющие отношение к этому типу, а именно:

PFD_SWAP_LAYER_BUFFERS,
PFD_GENERIC_ACCELERATED и PFD_DEPTH_DONTCARE.

А константа, названная PFD_DOUBLE_BUFFER_DONTCARE, по-видимому, соответствует константе, описанной в модуле windows.pas как PFD_DOUBLEBUFFER_DONTCARE.

Итак, смысл структуры PixelFormatDescriptor — детальное описание графической системы, на которой происходит работа. Вас может озадачить дотошность этого описания, но, уверяю, особое внимание из всего этого описания требуют совсем немногие вещи.

В проекте Я Привел описание всех ПОЛей структуры TPixelFormatDescriptor на русском языке (в момент их первоначального заполнения). Делается это в процедуре SetDCPixelFormat, вызываемой между получением ссылки на контекст устройства и созданием ссылки на контекст воспроизведения OpenGL.

Посмотрим подробнее, что там делается. Полям структуры присваиваются желаемые значения, затем вызовом функции ChoosePixelFormat осуществляется запрос системе, поддерживается ли на данном рабочем месте выбранный формат пиксела, и, наконец, вызовом функции SetPixelFormat устанавливается формат пиксела в контексте устройства.

Функция ChoosePixelFormat возвращает индекс формата пиксела, который нам нужен в качестве аргумента функции SetPixelFormat.

Заполнив поля структуры TPixelFormatDescriptor, мы определяемся со своими пожеланиями к графической системе, на которой будет происходить работа приложения, OpenGL подбирает наиболее подходящий к нашим пожеланиям формат и устанавливает уже его в качестве формата пиксела для последующей работы. Наши пожелания корректируются сервером OpenGL применительно к реальным характеристикам системы.

То, что OpenGL не позволит нам установить нереальный для конкретного рабочего места формат пиксела, значительно облегчает нашу задачу. Предполагая, что разработанное приложение будет работать на машинах разного класса, можно запросить "всего побольше", а уж OpenGL разберется в каждом конкретном случае, каковы параметры и возможности оборудования, на котором в данный момент выполняется приложение.

На этом можно было бы и закончить разговор о формате пиксела, если бы мы могли полностью довериться выбору OpenGL.

Обратим внимание на поле структуры "битовые флаги", dwFlags. То, как мы зададим значение флагов, может существенно сказаться на работе нашего

приложения, и наобум задавать эти значения не стоит. Тем более что некоторые флаги совместно "не уживаются", а некоторые присутствуют только в паре с определенными флагами.

В рассматриваемом примере я присвоил флагам значение PFD_DRAW_TO_WINDOW or PFD_SUPPORT_OPENGL, сообщив тем самым системе, что собираюсь осуществлять вывод в окно и что моя система в принципе поддерживает OpenGL (рис. 1.1). Я ограничился всего двумя константами из обширного списка, приведенного в модуле windows.pas. (В файле справки почти для каждой из них имеется детальное описание.)

Так, константа PFD_DOUBLEBUFFER включает режим двойной буферизации, когда вывод осуществляется не на экран, а в память, затем содержимое буфера выводится на экран. Это очень полезный режим: если в любом примере на анимацию убрать режим двойной буферизации и все связанные с этим режимом команды, то при выводе кадра будет заметно мерцание. Во всех последующих примерах, начиная со второй главы, я буду использовать этот режим, кроме некоторых специально оговариваемых случаев.

Замечание

Кадр, содержимое которого мы непосредственно видим на экране, называется передним буфером кадра, вспомогательный раздел памяти, в котором подготавливается изображение, называется задним буфером кадра.

Константу PFD_GENERIC_ACCELERATED имеет смысл устанавливать только в случае, если компьютер оснащен графическим акселератором.

Флаги, заканчивающиеся на "DONTCARE", сообщают системе, что соответствующий режим может иметь оба значения, например, при установке флага PFD_DOUBLE_BUFFER_DONTCARE запрашиваемый формат пиксела допускает оба режима — как одинарной, так и двойной буферизации.

Со всеми остальными полями и константами я предоставляю вам возможность разобраться самостоятельно. Отмечу, что поле iLayerType, описанное в windows.pas как имеющее тип Byte, может, согласно справке, иметь три значения: PFD_MAIN_PLANE, PFD_OVERLAY_PLANE И PFD_UNDERLAY_PLANE, однако константа PFD_UNDERLAY_PLANE имеет значение — 1, так что установить такое значение для величины типа Byte не удастся.

OpenGL позволяет узнать, какой же формат пиксела он собирается использовать. Для ЭТОГО необходимо ИСПОЛЬЗОВАТЬ функцию DescribePixelFormat, заполняющую величину типа TPixelFormatDescriptor установленным форматом пиксела.

Построим несложное приложение на основе использования этой функции, которое позволит детальнее разобраться с форматом пиксела и подобрать формат для конкретного рабочего места (проект из подкаталога Ex22).

В примере битовым флагам задаем все возможные значения одновременно, числовым полям задаем заведомо нереальное значение 64, и смотрим на вы-

бор формата пиксела, сделанный OpenGL. Результат, который вы получите для выбранного OpenGL формата пиксела, я предсказать не могу: он индивидуален для каждой конкретной конфигурации компьютера и текущих настроек. Скорее всего окажется, что режим двойной буферизации не будет установлен. (Напоминаю: многие флаги устанавливаются только в определенных комбинациях с другими.)

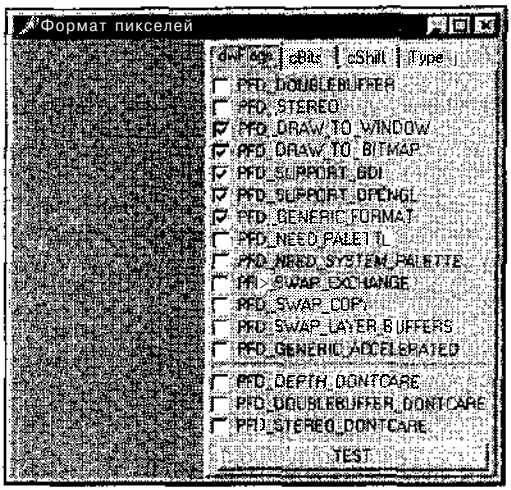


Рис. 1.1. По умолчанию режим двойной буферизации не установлен

Наше приложение позволяет менять параметры формата пиксела и устанавливать его заново, а чтобы видеть воспроизведение, небольшая площадка на экране при каждом тестировании окрашивается случайным цветом, используя функции OpenGL. Поэкспериментируйте с этим приложением, например, определите комбинацию флагов для установления режима двойной буферизации. Посмотрите значение числовых полей формата при различной палитре экрана: 16 бит, 24 бита, 32 бита, если у вас есть такой режим, но не в палитре с 256 цветами. О выводе OpenGL при палитре экрана в 256 цветов у пас будет отдельный разговор.

Это приложение, в частности, дает ответ на вопрос, как определить, оснащен ли компьютер графическим акселератором. Сделать это можно после вызова функции `DescribePixelFormat` следующим образом:

```
var
  i, j : Integer;
...
i :=   pfd.dwFlags and PFD_GENERIC_ACCELERATED;
j := pfd.dwFlags and PFD_GENERIC_FORMAT;
If (i = 0) and (j = 0)
 then // полноценный ICD-драйвер г. функциями ускорения
 else If (i = 1) and (j = 1)
```

```
 then  // MCD-драйвер, аппаратно реализуется
       // только часть функций ускорения
else   // режим программной эмуляции, всю работу выполняет центральный
       // процессор
```

В следующей главе мы узнаем еще один способ определения наличия акселератора.

С помощью рассмотренного проекта вы найдете ответ на вопрос, на который я вам ответить не смогу, а именно — как заполнить структуру TPixelFormatDescriptor для вашего компьютера.

Решение проблем

Обратите внимание, что в коде проекта TestPFD я установил несколько проверок на отсутствие ссылки на контекст воспроизведения, который может быть потерян по ходу работы любого приложения, использующего OpenGL — редкая, но возможная ситуация в штатном режиме работы системы и очень вероятная ситуация, если, например, по ходу работы приложения менять настройки экрана. Если значение соответствующей переменной равно нулю, вывод OpenGL оказывается невозможным:

```
If hrc=0 then ShowMessage('Отсутствует контекст воспроизведения OpenGL');
```

Обратите также внимание на следующую важную вещь. В самой первой программе, использующей OpenGL, как и в подавляющем большинстве последующих примеров, процедура установки формата пиксела записана мною в самом коротком варианте:

```
FillChar (pfd, SizeOf (pfd), 0);
nPixelFormat := ChoosePixelFormat (hdc, @pfd);
Set Pixel Format (hdc,nPixelFormat, @pfd) ;
```

То есть ни одно из полей pfd я не задаю явно, отдавая все на откуп OpenGL. В некоторых же примерах я ограничиваюсь только заданием необходимых значений для полей битовых флагов.

Я не встречал ситуаций, когда бы такой подход не срабатывал, не считая случаев с использованием других, нежели фирмы Microsoft, версий OpenGL, но поручиться за то, что он будет работать для всех графических карт, не могу. Возможно, проблемы возникнут также из-за некорректной работы драйверов (стандартная отговорка, не правда ли?).

Если примеры с прилагаемой дискеты у вас не работают, выдавая просто черный экран, начните поиск причины с определения значения hrc сразу же после создания ссылки на контекст воспроизведения. Если это значение равно нулю, в процедуре установки формата пиксела задайте всем полям значения согласно полученным с помощью приложения проекта TestPFD.

Скорее всего, вам рано или поздно потребуется разрешать подобные проблемы, связанные с неверным форматом пиксела или подобными системными ошибками. Сделать это в проектах, где не используется библиотека классов Delphi, оказывается сложным для новичков. Помощью может стать пример из подкаталога Ex23, где я демонстрирую, как выдать информацию о последней системной ошибке. В программе намеренно сделана ошибка путем превращения строки с получением ссылки на контекст устройства в комментарий.

Функция API FormatMessage позволяет преобразовать сообщение об ошибке в формат, пригодный для вывода:

```
lpMsgBuf: PChar;
...
FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER or
FORMAT_MESSAGE_FROM_SYSTEM,
nil, GetLastError() , LANG_NEUTRAL, @lpMsgBuf, 0, nil);
MessageBox(Window, lpMsgBuf, 'GetLastError', MB_OK);
```

Сообщение выводится в осмысленной форме и на языке, соответствующем локализации операционной системы. В данном случае выводится фраза "Неверный дескриптор".

Если убрать знаки комментария со строки с получением ссылки, а закомментировать строку с вызовом функции SetPixelFormat, сообщение об ошибке будет выглядеть как "Неправильный формат точки" (подразумевается "Неверный формат пиксела").

Полный список системных ошибок, связанных с использованием OpenGL, можно посмотреть в файле windows.pas, в разделе "OpenGL Error Code".

Учтите, что в этом примере выводится информация о последней системной ошибке, а она могла произойти задолго до работы приложения, так что следует использовать такую диагностику ошибок только при отладке приложений. Первый аргумент функции API FormatMessage позволяет определять дополнительные детали вывода сообщения.

Замечание

Во второй главе мы познакомимся с еще одним способом диагностирования ошибок, стандартным для OpenGL.

Вывод на компоненты Delphi средствами OpenGL

Теоретически с помощью функций OpenGL можно осуществлять вывод не только на поверхность формы, но и на поверхность любого компонента, если у него имеется свойство Canvas.Handle, для чего при получении ссыл-

ки на контекст воспроизведения необходимо указывать ссылку на контекст устройства, ассоциированную с нужным компонентом, например, Image1.Canvas.Handle. Однако чаще всего это приводит к неустойчивой работе, вывод то есть, то нет, хотя контекст воспроизведения присутствует и не теряется.

OpenGL прекрасно уживается с визуальными компонентами, как видно из примера TeslPFD, так что чаще всего нет необходимости осуществлять вывод на поле не формы, а компонента Delphi.

Если для ваших задач необходимо ограничить размер области вывода, то для этого есть стандартные методы, которые мы обсудим во второй главе.

Подкаталог Ex24 содержит проект, в котором вывод осуществляется на поверхность панели — компонента, вообще не имеющего свойства Canvas. Для этого мы пользуемся тем, что панель имеет отдельное окно:

```
dc := GetDC (Panel1.Handle);
SetDCPixelFormat(dc);
hrc := wglCreateContext(dc);
```

Аналогичным образом можно организовать вывод на поверхность любого компонента, имеющего свойство Handle (т. е. имеющего самостоятельное окно), например, на поверхность обычной кнопки. Обязательно попробуйте сделать это.

Для вывода на компонент класса TImage можете записать:

```
dc := Image1.Canvas.Handle;
```

И удалить строки BeginPaint и EndPaint, поскольку класс TImage не имеет свойства Handle, т. е. не создает отдельного окна.

Однако вывод на компоненты, подобные компонентам класса Timage, т. е. не имеющие свойства Handle, отличается полной неустойчивостью, так что я не гарантирую вам надежного положительного результата.

Почему это происходит, выясним в следующем разделе.

Стили окна и вывод OpenGL

В проекте из подкаталога Ex25 я немного модифицировал пример минимальной программы OpenGL таким образом, что получилось простое MDI-приложение, в котором каждое дочернее окно окрашивается случайным образом с использованием команд OpenGL (рис. 1.2).

Проект из подкаталога Ex26 отличается от предыдущего только тем, что получается SDI-приложение (рис. 1.3).

Из этих примеров видно, что приложение может иметь сколько угодно контекстов воспроизведения. Однако следует иметь в виду, что каждое окно

может иметь только один контекст воспроизведения, чем. в частности, и объясняется неустойчивый вывод на компоненты, не имеющих собственного окна.

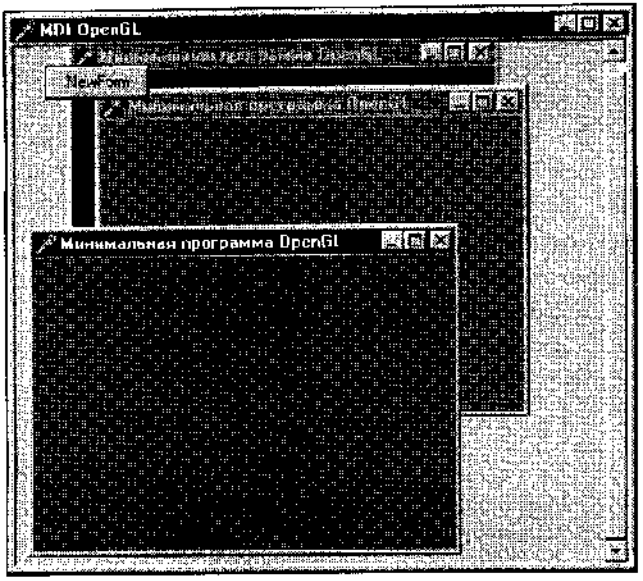


Рис. 1.2. Приложение для вывода командами OpenGL может иметь сколько угодно окон

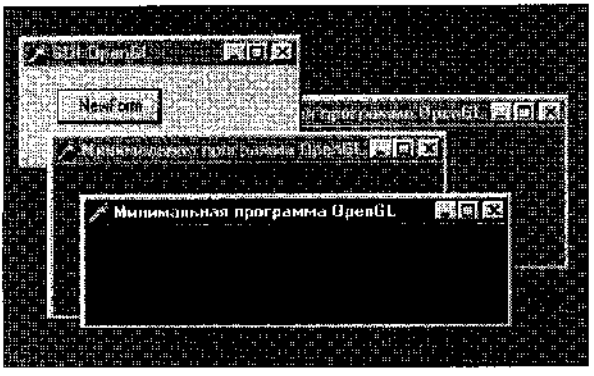


Рис. 1.3. Для вывода командами OpenGL можно использовать любой интерфейс приложений

В справке по функции `SetPixelFormat` говорится о том, что вывод средствами OpenGL не может быть осуществлен на окно совершенно произвольного стиля, класс окна должен включать стили "`WS_CLIPCHILDREN and WS_CLIPSIBLINGS`". Это соответствует обычным окнам Delphi— и родительским, и дочерним. Также говорится и о том, что атрибуты класса окна не могут включать стиль "`CS_PARENTDC`", ЧТО является полным ответом на вопрос о неустойчивости вывода средствами OpenGL на поверхность компонентов типа класса `TImage`.

В примере из подкаталога Ex27 по сравнению с проектом минимальной программы OpenGL я поменял значение свойства BorderStyle окна формы на bsNone, т. е. осуществил вывод на окно без рамки.

Я не встречал ситуации, когда бы подобные проекты с выводом на окно без рамки или MDI-приложения с выводом средствами OpenGL на дочерние окна работали некорректно или неустойчиво, независимо от текущих установок или используемого акселератора, если свойство WindowState имеет значение wsNormal.

Более вероятно возникновение проблем в случае полноэкранных приложений.

Полноэкранные приложения

Такие приложения достойны отдельного разговора в силу их особой значимости. Занять всю область экрана вам может потребоваться для того, чтобы повысить эффектность и зрелищность вашего проекта, особенно если на экране присутствует много объектов.

Прежде всего, необходимо сказать о том, что некоторые графические акселераторы поддерживают ускорение только в полноэкранном режиме и при определенных установках экрана, например, только при разрешении экрана 640x480 и цветовой палитре 16 бит на пиксел.

К сожалению, мне придется сообщить о наличии здесь некоторых проблем, впрочем, мы их успешно разрешим. Вернитесь к последнему примеру — проекту из подкаталога Ex27, где вывод средствами OpenGL осуществляется на окно без рамки и области заголовка. Поменяйте свойство WindowState формы на wsMaximized, чтобы окно после запуска раскрывалось на весь экран. Запустите проект или откомпилированный модуль. Что у вас получится, я предсказать не могу, он зависит от конкретной конфигурации машины. Если на вашем компьютере все происходит в ожидаемом режиме, т. е. весь экран занят окном голубоватого цвета, вы избавлены от множества проблем, если только не собираетесь распространять свои приложения. Дело в том, что я тестировал подобные проекты на компьютерах самой различной конфигурации и обнаружил, что чаще всего результат получается обескураживающий и неприятный: окно раскрывается некорректно, не полностью занимая экран. Причем неприятности появляются именно в связи с использованием OpenGL, простые проекты вполне справляются с задачей раскрывания на весь экран. Возможно, корень проблемы в несогласованности работы драйверов (как всегда!) или ошибках операционной системы.

Однако решение проблемы оказывается совсем простым.

Посмотрите пример из подкаталога Ex28. Это тот же пример, что и предыдущий, только немного измененный. Свойство WindowState окна формы установлено в wsNormal, а обработчик события OnCreate дополнился строкой:

```
WindowState := wsMaximized;
```

Теперь все работает превосходно, окно действительно раскрывается на весь экран.

Свойство `FormStyle` окна можно задать как `fsStayOnTop`, и тогда приложение будет вести себя так, как многие профессиональные игры, не позволяя переключиться на другие приложения.

Другое решение проблемы очень похоже на предыдущее. Посмотрите пример Ex29 — модифицированный пример вывода на поверхность панели. Панель занимает всю клиентскую область окна (свойство `Align` имеет значение `alClient`), а окно формы максимизировано и не имеет рамки.

Надеюсь, у вас теперь не будет неприятностей при попытке захватить всю область экрана, хотя нелегко объяснить, почему обычный метод не работает, а такой метод, по сути, ничем от него не отличающийся, работает.

Итак, полный экран отвоевывать мы научились. Теперь необходимо научиться менять программно, т. е. без перезагрузки, разрешение экрана: приложению может потребоваться другое, нежели установленное пользователем разрешение, к которому он привык в своей повседневной работе.

Проект FullScr из подкаталога Ex30 является упрощенным вариантом такой программы. После запуска приложения пользователю из списка предлагается выбрать желаемое разрешение экрана, которое устанавливается на время работы основного модуля — минимальной программы OpenGL. После окончания работы модуля возвращается первоначальное разрешение экрана.

Пример ПОСТроен на использовании функции API `ChangeDisplaySettings`, первый аргумент которой — структура, описывающая требуемый режим. Второй аргумент — битовая комбинация констант, одна из которых задает тестирование режима без его установки.

Массив `LowResModes` заполняем перечислением всех возможных режимов, тестируем последовательно каждый из них и отмечаем те, тестирование для которых оказалось успешным. После пользовательского выбора действительно устанавливаем выбранный режим, а по завершению работы приложения возвращаем запомненный первоначальный режим.

Протестировав программу в различных режимах, вы можете выяснить, что не во всех из них возможно использование OpenGL, в некоторых режимах контекст воспроизведения не может быть получен.

В целом такой способ создания полноэкранного приложения я нахожу вполне удовлетворительным. При тестировании на компьютере без акселератора приложение в проблемных режимах выдавало сообщение о невозможности получения контекста, а при подключении акселераторов сообщение не появлялось, но в некоторых режимах окно и не окрашивалось. Акселераторы первых моделей могут искаженно передавать картинку в некоторых режимах.

Приведу еще один пример на полноэкранный режим работы (проект из подкаталога Ex31). Здесь для переключения в различные режимы используется DirectDraw, все необходимые модули для его использования находятся также в этом подкаталоге.

Параметры режима — ширина, высота и разрядность пиксела — задаются в виде трех чисел в командной строке.

С помощью этого приложения вы можете выяснить, что на самом деле не *все* режимы доступны для использования в принципе, чем и объясняется то, что в предыдущем примере не во всех режимах можно было получить контекст воспроизведения. В проблематичных режимах на экране хорошо заметно искажение изображения.

Я думаю, что описанный способ создания полноэкранного приложения вполне можно считать универсальным и достаточно надежным.

Есть еще один, очень простой, способ создания полноэкранного приложения: рисование прямо на поверхности рабочего стола. Во второй главе я приведу соответствующий пример, хотя вы уже сейчас знаете, что для этого необходимо сделать. Но этот способ может не работать с вашей картой.

Типы OpenGL

Библиотека OpenGL является переносимой по отношению к платформам, операционным системам и средам программирования.

Для обеспечения этой независимости в ней, в частности, определены собственные типы. Их префикс — "GL", например, GLint.

В каждой среде программирования в заголовочных файлах эти типы переопределяются согласно собственным типам среды. Разберем, как это делается в Delphi.

Заголовочный файл Delphi opengl.pas начинается с определения знакомого нам типа HGLRC:

```
type
  HGLRC = THandle;
```

Далее следует описание всех остальных типов OpenGL, например, наиболее "ходовой" тип GLfloat соответствует типу Single:

```
GLfloat = Single;
```

Поначалу многие испытывают затруднение, когда приходится использовать "неродные" для Delphi типы. По мере накопления опыта эта неловкость быстро проходит, и я рекомендую использовать с самого начала знакомства именно типы библиотеки OpenGL, даже если вы наизусть знаете их родные для Delphi аналоги. Наверняка вам рано или поздно придется разбираться

в чужих программах или переносить свои программы в другую среду программирования или даже в другую операционную систему. В атмосфере беспрерывной смены технологий, в которой мы находимся все последние годы, нельзя быть уверенным в том, что какая-либо операционная система (и/или среда программирования) на долгие годы станет надежным средством воплощения наших идей. Вряд ли кто-то может поручиться, что его любимая операционная система проживет еще хотя бы десяток лет и не выйдет внезапно из моды, сменившись другой, о которой вы и не слышали пару месяцев назад.

Однако вернемся к типам OpenGL. Не все из них удается точно перевести. Например, `GLclampf` — вещественное число в пределах от пуля до единицы — в Delphi определен просто как single. Поэтому обычно в программах устанавливают "ручную" проверку на вхождение величины такого типа в требуемый диапазон.

Будьте внимательны с целыми числами: помимо типа GLint имеется тип `GLUint` — целое без знака, соответствующее типу Cardinal.

В ряду типов OpenGL особо надо сказать о типе

```
GLboolean = BYTEBOOL;
```

Соответственно, определены две константы:

```
GL_FALSE  = 0;
GL_TRUE   = 1;
```

Константы эти имеют непосредственное отношение к типу `GLboolean`, однако их значения, как вы понимаете, не соответствуют типу BYTEBCOL. Из-за ошибки в описании типа (или определении констант) не удастся использовать стандартный для OpenGL код, поэтому вместо констант `GL_FALSE` и `GL_TRUE` будем ИСПОЛЬЗОВаТЬ `False` И `True`, соответственно.

Конечно, можно самому скорректировать описание типа, например, так:

```
GLboolean = 0. . 1;
```

После этой корректировки не придется отходить от стандарта кода графической библиотеки, но модифицировать стандартные модули Delphi нежелательно, иначе ваши проекты будут успешно компилироваться только на вашем рабочем месте.

Помимо основных типов, стандартных для OpenGL и вводимых в любой среде программирования, в заголовочном файле введены также типы, специфические только для Delphi, например, для наиболее часто употребляемых в OpenGL массивов введены специальные типы:

```
TGLArrayf4 = array [0..3] of GLFloat;
TGLArrayf3 = array [0..2] of GLFloat;
TGLArrayi4 = array [0..3] of GLint;
```

Это сделано, по-видимому, для удобства кодирования и повышения читабельности кода, поскольку нигде больше в этом модуле указанные типы не встречаются.

Разработчикам также пришлось ввести специальные типы для указателей, например:

```
PGLfloat = ^GLFloat;
```

Такого типа нет в стандартном наборе типов OpenGL: библиотека изначально создавалась на языке C, синтаксис которого хорошо приспособлен к использованию указателей, поэтому во введении особого типа для них просто не было необходимости.

Вообще, должен сказать, что OpenGL наиболее приспособлен для программирования на C, поэтому некоторые моменты будут вызывать некоторые неудобства при использовании Delphi (уже упоминавшаяся система справок лишь одно звено в этой цепи). Тем не менее, это не помешает нам успешно освоить программировании на Delphi с использованием этой библиотеки.

Система Delphi имеет, конечно, слабые места. Чересчур большой размер откомпилированных модулей — не самое значительное из них. Для графических приложений крайне важна скорость работы, и здесь пальма первенства тоже не за Delphi. Если приложение интенсивно использует массивы и указатели, операции с памятью и проводит много вычислительных операций, то падение скорости при использовании Delphi вместо C/C++ оказывается значительным. По некоторым тестам, лучшие компиляторы C++ создают код, работающий в два раза быстрее.

Однако это не должно отпугнуть вас от дальнейшего изучения использования OpenGL в проектах Delphi, поскольку здесь как раз тот случай, когда скорость работы самого приложения не так уж и важна. Основную долю работы берет на себя сервер OpenGL, а приложению достается роль транслятора команд серверу, поэтому нет особых потерь производительности, если вместо языка C++ мы используем Pascal и Delphi.

Конечно, для сокращения потерь производительности желательно использовать приемы объектно-ориентированного программирования, хотя я бы не сказал, что эти приемы во всех случаях приведут к заметному на глаз ускорению работы приложения.

Delphi имеет свои неоспоримые достоинства — прежде всего это несравнимая ни с каким другим средством скорость разработки и компиляции. Именно поэтому, а также из-за "скрытого обаяния" Delphi (вы понимаете, о чем я говорю) мы и выбрали эту замечательную систему в качестве основы для изучения OpenGL.

Тип *TColor* и цвет в OpenGL

Разберем еще одну версию нашей первой программы, использующей OpenGL — пример из подкаталога Ex32. Здесь на форму помещена кнопка, при нажатии которой появляется стандартный диалог Windows выбора цвета. После выбора окно окрашивается в выбранный цвет, для чего используются команды OpenGL. Поскольку такой прием мы активно будем применять в будущем, разберем подробно, как это делается.

Цвет, возвращаемый диалогом, хранится в свойстве `Color` компонента класса `TColorDialog`. Согласно справке, значение `$00FFFFFF` ЭТОГО свойства соответствует белому цвету, `$00FF0000` — синему, `$0COOFFOO` — зеленому, `$OOOOOOFF` — красному. То есть для выделения красной составляющей цвета необходимо вырезать первый слева байт, второй байт даст долю зеленого, третий — синего. Максимальное значение байта — 255, минимальное — ноль. Цвета же OpenGL располагаются в интервале от нуля до единицы.

В нашем примере я ввел пользовательскую процедуру, определяющую тройку составляющих цветов для OpenGL по заданному аргументу типа `TColor`:

```
procedure TfrmGL.ColorToGL (c : TColor; var R, G, B : GLFloat) ;
begin
 R := (c mod S100) / 255;
 G := ((c div $100) mod $100) / 255;
 B := (c div $10000) / 255;
end;
```

Из аргумента вырезаются нужные байты и масштабируются в интервал [0; 1].

Замечание

Те же действия можно сделать и другим, более "продвинутым" способом:

```
R := (c and $FF) / 255;
G := ((c and $FF00) shr 8) / 255;
B := {(c and $FF0000) shr 16) / 255.
```

Эта процедура используется в обработчике нажатия кнопки:

```
If ColorDialog1.Execute then begin
   ColorToGL (ColorDialog1.Color, R, G, B) ;
   Refresh;
end;
```

В примере для простоты окно перекрашивается обычным для Delphi способом — через вызов метода Refresh формы.

Подробнее о заголовочном файле opengl.pas

Вместе с Delphi версии три и выше поставляется заголовочный файл, позволяющий подключать библиотеку OpenGL к проектам Delphi. Этот файл содержит только прототипы используемых функций и процедур, сами функции и процедуры размещены в соответствующих файлах DLL.

Например, в секции interface заголовочного файла содержится следующее forward-описание использованной нами во всех предыдущих примерах процедуры:

```
procedure glClearColor (red, green, blue, alpha: GLclampf); stdcall;
```

В секции implementation модуля собственно описание процедуры выглядит так:

```
procedure glClearColor; external opengl32;
```

Служебное слово stdcall, указанное для всех процедур и функций в этом модуле, означает стандартный вызов функции или процедуры и определяет некоторые правила обмена данными между приложением и библиотекой: как передаются параметры (через регистры или стек), в каком порядке перечисляются параметры и кто, приложение или библиотека, очищает области после их использования.

Служебное слово external указывается для функций, подключаемых из библиотек. После него следует указание имени подключаемой библиотеки. Здесь opengl32 — константа, определяемая, как я отмечал раньше, в другом модуле — в windows.pas:

```
opengl32   = 'opengl32.dll';
```

Константа, соответствующая второй используемой библиотеке, содержится в модуле opengl.pas:

```
const
  glu32 = 'glu32.dll';
```

Содержательная часть модуля opengl, соответствующая его инициализации, содержит единственную строку:

```
    Set8087CW($133F);
```

В справке по этой процедуре можно прочитать, что она служит для включения/выключения исключений при проведении операций с плавающей точкой. Здесь же отмечается, что для OpenGL рекомендуется эти исключения отключать.

Разброс описаний констант и некоторых функций и процедур по разным модулям объясняется тем, напомню, что информация, относящаяся к реали-

зации OpenGL под Windows, помешена в заголовочный файл windows.pas. Это логично и объяснимо, но в некоторых случаях может вызвать дополнительные проблемы, например, при использовании альтернативных заголовочных файлов или библиотек.

Далее мы встретимся с ситуациями, когда выяснится, что в стандартном модуле, поставляемом с Delphi, не окажется прототипов многих полезных команд. Там же попробуем найти объяснение этому факту.

Альтернативные заголовочные файлы, разработанные некоторыми сторонними организациями или энтузиастами, оказываются более полными в этой части и поэтому полезны в некоторых ситуациях, но из-за того, что модуль windows.pas уже содержит описание некоторых процедур, связанных с OpenGL, могут возникнуть накладки.

Следующая возможная ситуация — использование других. нежели производства фирмы Microsoft, библиотек. например, версии OpenGL фирмы SGI, которую отличают более высокие скоростные характеристики по некоторым показателям. В приложении "OpenGL в Интернете" я указан адрес, по которому вы можете найти ее дистрибутив. Правда, эту версию OpenGL имеет смысл использовать только на компьютерах, не оснащенных акселератором, поскольку она не может самостоятельно использовать драйверы ICD и MCD, а переадресует все вызовы в Microsoft OpenGL, чем сводятся на нет все ее достоинства. В свое время SGI обещала в следующей версии устранить этот недостаток, однако планы этой корпорации по поддержке операционной системы Windows и сотрудничеству с Microsoft, по-видимому, изменились, так что эта работа, возможно, не будет завершена.

Если вам потребуется модифицировать заголовочные файлы для подключения другой библиотеки, то придется учитывать несколько нюансов, связанных с версиями используемой Delphi.

Здесь я смогу только обозначить эти тонкости. Хотя я и нашел решение проблемы подключения других версий OpenGL, но не имею возможности поделиться им — файл windows.pas лаже в сжатом виде имеет очень большой размер, поэтому па одной дискете модифицированные версии этого файла не разместить {для всех версий Delphi).

Первое, с чего надо начать, — это изменить значения констант opengl32 и glu32 в заголовочных файлах и установить имена соответствующих файлов библиотек. Если вас не страшит то, что модифицируются стандартные модули Deiphi, можете сразу компилировать проекты. Если же вы модифицируете копии стандартных модулей, то придется учитывать, что почти каждый из стандартных модулей Delphi в списке uses подключает модуль windows. и, возможно, придется переставлять порядок модулей в списке uses.

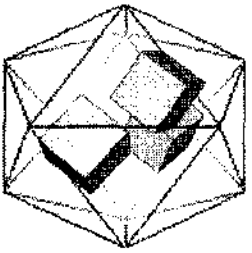
Обязательно посмотрите с помощью утилит быстрого просмотра или tdump заголовочную информацию откомпилированного приложения для того, что-

бы убедиться, что оно использует только те библиотеки, которые подразумевались. При наличии нескольких библиотек невозможна ситуация, когда функции берутся вперемешку из разных библиотек.

Если используется только одна определенная библиотека, но контекст воспроизведения оказывается невозможным получить, попробуйте явным образом заполнять формат пиксела, указывая требуемые значения для каждого поля — это может помочь.

В приложении "OpenGL в Интернете" я указал адреса, по которым вы можете получить свободно распространяемые заголовочные файлы независимых разработчиков, более полные, чем стандартный модуль. Ими можно воспользоваться и в том случае, если у вас не получится самостоятельно подключать другие версии OpenGL.

ГЛАВА 2



Двумерные построения

В этой главе на самых простых примерах мы разберем азы построений объектов. Пока рассмотрим рисование только на плоскости, однако полученные знания постоянно будут использоваться при переходе к рисованию в пространстве.

OpenGL является низкоуровневой библиотекой. В частности это означает, что рисование объемных фигур сводится к последовательному рисованию в пространстве плоских фигур, образующих нужную объемную. Поэтому, даже если вас интересует только использование 3D возможностей библиотеки, пропускать эту главу не рекомендуется.

Примеры располагаются на дискете в каталоге Chapter2.

Точка

Для начала скажем, что в OpenGL левый нижний угол области вывода имеет координаты [-1; —1], правый верхний — [I, 1].

Начнем наше знакомство с примитивами OpenGL с самого простого из них — точки. Нарисуем на экране пять точек, четыре по углам окна и одну в центре (проект располагается в подкаталоге Ex01).

Обработчик события onPair.t формы дополнился следующими строками:

```
glViewPort (0, 0, ClientWidth, ClientHeight); // область вывода
glPointSize (20);                             // размер точек
glColor3f (1.0,1.0,1.0);                      // цвет примитивов
glBegin (GL_POINTS);              // открываем командную скобку
   glVertex2f (-1,-1) ;           // левый нижний угол
   glVertex2f (-1, 1);            // левый верхний угол
```

```
   glVertex2f (0, 0);                    // центр окна
   glVertex2f (1, -1);                   // правый верхний угол
   glVertex2f (1, 1);                    // правый нижний угол
glEnd;                                   // закрываем командную скобку
SwapBuffers(Canvas.Handle);              // содержимое буфера — на экран
```

Разберем каждую из строк программы подробно.

Первая строка задает область вывода указанием координат левого нижнего и правого верхнего углов (в пикселах, в оконных координатах). Я взял в качестве области вывода всю клиентскую часть окна. Если третий параметр процедуры `glViewPort` записать как round (ClientWidth / 2), то картинка вдвое сузится, но окрашено будет все окно (проект из подкаталога ExO2).

Следующие две строки программы определяют параметры выводимых точек, размер и цвет.

На примере команды `glColor3f` разберем синтаксис команд OpenGL.

Из справки по этой команде вы узнаете, что она принадлежит к целому набору команд `glColor` с различными окончаниями: зь, `4i` и прочие. Цифра в окончании соответствует количеству требуемых аргументов, а следующая цифрой буква показывает требуемый тип аргументов. То есть `glColor3f` требует в качестве аргументов тройку вещественных (float) чисел, а glColor3i — тройку целых (int.) чисел.

Аналогичный синтаксис мы встретим у многих других команд OpenGL.

Здесь же, в справке, выясняем, что при записи функции в вещественной форме аргументы лежат в интервале [0; 1], а в целочисленной форме — линейно отображаются на этот интервал, т. е. для задания белого цвета целочисленная форма команды будет выглядеть так:

```
glColor3i (2147483647, 2147483647, 2147483647);     // цвет примитивов
```

где максимальное 8-битное целое без знака соответствует предельному значению интервала.

Почти всегда предпочтительно использовать команду в вещественной форме, поскольку OpenGL хранит данные именно в вещественном формате. Исключения оговариваются в файле справки.

Замечание

Если имя команды заканчивается на v (векторная форма), то аргументом ее служит указатель на структуру, содержащую данные, например, массив. То есть, например, если последние три символа в имени команды 3fv, то ее аргумент — адрес массива трех вещественных чисел. Использование такой формы команды является самым оптимальным по скоростным характеристикам.

Далее в программе следуют функции (командные скобки) `glBegin` и `glEnd`, между которыми заключены собственно процедуры рисования.

Разберемся подробнее с функциями glBegin и glEnd ввиду их особой важности: большинство графических построений связано с использованием именно этой пары функций.

Во-первых, необходимо уяснить, что это командные скобки библиотеки OpenGL, не заменяющие операторные скобки языка Pascal и не имеющие к ним никакого отношения. Ошибка при использовании командных скобок не распознается компилятором. Если в программе написана неправильная вложенность командных скобок OpenGL, то ошибка проявится только в процессе диалога приложения с сервером.

Во-вторых, внутри этих скобок могут находиться любые операторы языка Pascal и почти любые функции OpenGL (вернее, очень многие). Включенные в скобки команды OpenGL отрабатываются так же, как и за пределами этих скобок. Главное назначение командных скобок — это задание режима (примитива) для команд glVertex (вершина), определяющих координаты вершин для рисования примитивов OpenGL.

В рассмотренном примере аргументом функции glBegin я взял символическую константу GL_POINTS. Из файла справки выясняем, что в этом случае все встретившиеся до закрывающей скобки glEnd вершины (аргументы glVertex) задают координаты очередной отдельной точки. Команду givertex я взял в форме с двумя вещественными аргументами. Получив справку по givertex, вы можете убедиться, что и эта команда имеет целый ворох разновидностей.

Мы собирались нарисовать четыре точки по углам окна и одну в центре, поэтому между командными скобками располагаются пять строк с вызовом givertex, аргументы которых соответствуют положениям точек в системе координат области вывода библиотеки OpenGL.

Сейчас обратите внимание на то, что вместе с изменением размеров окна рисуемое изображение также изменяется: точки всегда рисуются на своих местах относительно границ окна. Следующее замечание тоже очень важно: обработчик события OnResize формы тот же, что и у события OnPaint.

Приведу еще несколько простых примеров на рисование точек.

Если надо нарисовать десять точек по диагонали, то можно написать так (пример располагается в подкаталоге Ex03):

```
glBegin (GL_POINTS);
    For i := 0 to 9 do
      glVertex2f (i / 5 - 1, i / 5 - 1);
glEnd;
```

Ну а следующий код нарисует сотню точек со случайными координатами и цветами (пример из подкаталога Ex04):

```
glBegin (GL_POINTS);
   For i := 1 to 100 do begin
```

```
      glColor3f (random, random, random);
      glVertex2f (random * 2 - 1, random * 2 - 1 ) ;
   end;
glEnd;
```

Из этого примера мы видим, что команда, задающая цвет примитивов, может присутствовать внутри командных скобок OpenGL. Вызов же команды `glPointSize` между командными скобками безрезультатен и будет генерировать внутреннюю ошибку OpenGL. Так что если мы хотим получать точки случайных размеров, цикл надо переписать так (проект из подкаталога Ex05):

```
For i := 1 to 100 do begin
   glColor3f (random, random, random);
   glPointSize (random (20) ) ; // обязательно за пределами скобок
   glBegin (GL_POINTS);
     glVertex2f (random * 2 - 1, random * 2 - 1 ) ;
   glEnd;
end;
```

Скорее всего, вас удивило то, что точки рисуются в виде квадратиков. Чтобы получить точки в виде кружочков, перед `glBegin` вставьте строку:

```
glEnable (GL_POINT_SMOOTH) ;  // включаем режим сглаживания точек
```

Пара команд `glEnable` и `glDisable` играет в OpenGL очень важную роль, включая и отключая режимы работы следующих за ними команд. Нам придется еще не раз обращаться к возможным аргументам этих функций, задающим, какой конкретно режим включается или отключается.

Заканчивается обработчик события `OnPaint` вызовом `SwapBuffers`. Эта команда используется в режиме двойной буферизации для вывода на экран содержимого заднего буфера.

Замечание

Вспомним: в режиме двойной буферизации все рисование осуществляется в задний буфер кадра, он является текущим на всем процессе воспроизведения. По команде SwapBuffers текущее содержимое переднего буфера подменяется содержимым заднего буфера кадра, но текущим буфером после этого все равно остается задний.

Команда `glClear` с аргументом `GL_COLOR_BUFFER_BIT` очищает текущий буфер вывода.

Иногда в некоторых программах вы можете встретить, что серия команд рисования очередного кадра заканчивается командой `glFinish`, ожидающей, пока все предыдущие команды OpenGL выполнятся, или же командой `glFlush`, ускоряющей выполнение предыдущих команд.

Замечание

В большинстве примеров этой книги я не использую вызов glFinis:: или glFlush, ускорение от них получается микроскопическим. Но в программах, не применяющих двойную буферизацию, эти команды должны вызываться обязательно, иначе на экране может оказаться "недорисованная" картинка.

Надеюсь, теперь каждая строка рассмотренной программы вам ясна.

Приведу еще одну иллюстрацию — занятный пример на использование полученных знаний. В разделе private опишите две переменные:

```
xpos : GLfloat; // координаты курсора в системе координат OpenGL
ypos : GLfloat;
```

Создайте следующий обработчик события MouseMove формы:

```
xpos := 2 * X / ClientWidth - 1;
ypos := 2 * (ClientHeight - Y) / ClientHeight - I;
Refresh; // перерисовываем окно
```

В обработчике события Paint опишите локальную целочисленную переменную i и содержательную часть кода приведите к виду:

```
For i := 1 to 40 do begin // сорок точек
     glColcr3f (random, random, random); // случайного цвета
     glPointSize (random (10)); // случайного размера
     glBegin (GL_POINTS) ; //со случайными координатами вокруг курсора
       glVertex2f (xpos + 0.2 * random * sin (random (360)),
                   ypos + 0.2 * random * cos (random (360)));
     glEnd;
end;
```

Если все сделано правильно, при движении курсора по поверхности формы в районе курсора появляется облачко разноцветных точек (готовый проект я поместил в подкаталог Ex06). Разберитесь в этом примере с масштабированием значения переменных xpos и ypos и обратите внимание, что обработчик движения мыши заканчивается вызовом Refresh — принудительной перерисовкой окна при каждом движении курсора.

Предлагаю также разобрать простой пример из подкаталога Ex07 — построение синусоиды средствами OpenGL:

```
procedure TfrmGL.FormPaint(Sender: TObject);
const
 a = -Pi;     // начало интервала
 b = Pi;      // конец интервала
 num = 200;   // количество точек на интервале
var
 x : GLfloat;
```

```
 i : GLint;
begin
 wglMakeCurrent(Canvas.Handle, hrc);

 glViewPort (0, 0, ClientWidth, ClientHeight);

 glClearColor (0.5, 0.5, 0.75, 1.0);
 glClear (GL_COLOR_BUFFER_BIT);

 glEnable (GL_POINT_SMOOTH);
 glColor3f (1.0, 0.0, 0.5);

 glBegin (GL_POINTS);
  For i := 0 to num do begin
   x := a + i * (b - a) / num;
   glVertex2f (2 * (x - a) / (b - a) - 1.0, sin(x) * 0.75);
  end;
 glEnd;

 SwapBuffers(Canvas.Handle);
 wglMakeCurrent(0, 0);
end;
```

Пример из подкаталога Ex08 демонстрирует вывод средствами OpenGL прямо на поверхность рабочего стола. Напоминаю, окно с нулевым значением дескриптора соответствует поверхности рабочего стола, чем мы и пользуемся в этом примере для получения ссылки на контекст устройства:

```
dc := GetDC (0);
```

Для завершения работы приложения нажмите клавиши <Alt>+<F4> или <Esc>. Обратите внимание, что по завершении работы приложения перерисовываем рабочий стол:

```
InvalidateRect (0, nil, False);
```

Перерисовка здесь необходима для восстановления нормального вида рабочего стола.

Замечание

Подобный прием для создания полноэкранного приложения прекрасно работает на компьютерах без акселератора, чего не скажу о компьютерах, оснащенныхускорителем.

В заключение разговора о точках приведу одно маленькое, но важное замечание: помимо константы GL_POINTS в OpenGL имеется символическая константа GL_POINT, использующаяся в других, нежели glBegin, командах. Но компилятор Delphi, конечно, не сможет распознать ошибку, если вместо одной константы OpenGL указать другую, когда типы констант совпадают.

В этом случае ошибка приведет только к тому, что не будет построено ожидаемое изображение, аварийное завершение работы приложения не произойдет.

То же самое справедливо для всех рассматриваемых далее констант — необходимо внимательно следить за синтаксисом используемых команд.

Команда *glScissor*

Вернемся к проекту из подкаталога ExO2, в котором область вывода задается на половину экрана. Возможно, вас этот пример не удовлетворил: хотя картинка и выводится на половину экрана, окрашивается все-таки весь экран, а иногда это нежелательно и необходимо осуществлять вывод именно в пределах некоторой части окна.

Решение может заключаться в использовании функции вырезки glScissor, определяющей прямоугольник в окне приложения, т. е. область вырезания. После того как область вырезки задана, дальнейшие команды воспроизведения могут модифицировать только пикселы, лежащие внутри области (формулировка взята из файла справки).

Для использования этой функции необходимо включить режим учета вырезки:

```
glEnable(GL_SCISSOR_TEST);
```

После использования вырезки этот режим необходимо отключить парной командой glDisable.

Разберите проект из подкаталога ExO9 (второй пример этой главы), дополненный строками включения режима вырезки и командой, задающей область вырезки:

```
glEnable(GL_SCISSOR_TEST); // включаем режим использования вырезки
glScissor(0, 0, round(ClientWidth/2), ClientHeight); // область вырезки
```

Функция glScissor не заменяет команды glViewPort, задающей область вывода: если в этом примере область вывода распространить на весь экран, то на экране будет рисоваться половина картинки, т. е. только то, что попадает в область вырезки.

Совместный вывод посредством функций GDI и OpenGL

Возможно, вы заинтересовались вопросом, можно ли перемежать функции вывода GDI и OpenGL. Вопрос этот, конечно, носит скорее академический, чем практический характер.

Совместное использование поверхности окна возможно при условии, что канва будет доступна для вывода, т. е. в этом случае надо обязательно освобождать контексты.

Замечание

Многое также зависит от графической карты компьютера.

Посмотрите несложный пример из подкаталога Ex10, где код перерисовки окна первого примера данной главы дополнен строками:

```
Canvas.Brush.Color := clGreen;
Canvas.Ellipse (10, 10, 50, 50);
```

Обратите внимание, что эти строки располагаются после строки, освобождающей контекст воспроизведения. Если поставить вывод средствами GDI перед этой строкой, вывода не произойдет, а если поставить до строки, устанавливающей контекст воспроизведения, то картинка, выдаваемая OpenGL, закроет нарисованное изображение.

Отрезок

От точек перейдем к линиям. Разберем следующий возможный аргумент команды `glBegin` — константу `GL_LINES`, задающий примитив "независимый отрезок".

Для этого примитива следующие в командных скобках вершины (т. е. функции `glVertex`) задают попарно координаты начала и конца каждого отрезка прямой. Снова вернемся к первому примеру с точками и подправим код рисования следующим образом (готовый проект можете найти в подкаталоге Ex11):

```
glBegin (GL_LINES);
   glVertex2f (-1, 1);
   glVertex2f (1,-1);
   glVertex2f (-1,-1);
   glVertex2f (1, 1);
glEnd;
```

Рисуются два отрезка, соединяющие углы окна по диагоналям.

Для увеличения толщины отрезков перед командными скобками укажите ширину линии:

```
glLineWidth (2.5);
```

Эта функция также должна выноситься за командные скобки.

Как и у точек, у линий можно устранять ступенчатость. Исправьте код следующим образом (подкаталог Ex12):

```
glLineWidth (15) ;
glEnable (GL_LINE_SMOOTH);
glBegin (GL_LINES);
   glVertex2f (-0.7, 0.7);
   ...
```

и посмотрите результаты работы программы с вызовом и без вызова glEnable (GL_LINE_SMOOTH).

Итак, константа GL_LINES задает примитив отдельных отрезков, определенных указанием пар вершин. Понятно, что количество вершин должно быть четным.

Следующая константа — GL_LINE_STRIP — определяет примитив, когда перечисляемые вершины последовательно соединяются одна за другой. Приводимый код поясняет отличие этого примитива от предыдущего.

```
glBegin (GL_LINE_STRIP);
   glVertex2f (-1, -1) ;
   glVertex2f (-1, 1) ;
   glVertex2f (1, 1) ;
   glVertex2f (1,-1) ;
glEnd;
```

Результат — буква П по границе окна (проект из подкаталога Ex13).

В примитиве, задаваемом константой GL_LINE_LOOP, также последовательно соединяются перечисляемые вершины, однако последняя вершина замыкается с самой первой. Если в предыдущем примере использовать GL_LINE_LOOP, будет построен квадрат по границе окна (подкаталог Ex14).

В примерах на отрезки мы пока использовали непрерывную линию. Для рисования пунктирной линией перед командными скобками добавьте следующие строки (проект из подкаталога Ex15):

```
glLineStipple (1, $F0F0);
glEnable (GL_LINE_STIPPLE);
```

У функции glLineStipple первый аргумент — масштабный множитель, второй аргумент задает шаблон штриховки (побитовым способом).

Разберем проект из подкаталога Ex16 — еще один пример на использование штриховки (рис. 2.1).

Пользовательская процедура drawOneLine вызывается для воспроизведения каждого отдельного отрезка:

```
procedure TfrmGL.drawOneLine(x1,y1,x2,y2 : GLfloat);
begin
 glBegin(GL_LINES);
```

```
  glVertex2f (2 * x1 / ClientWidth - 1.0, y1 / ClientHeight - 0.5);
  glVertex2f (2 * x2 / ClientWidth - 1.0, y2 / ClientHeight - 0.5);
 glEnd;
end;
```

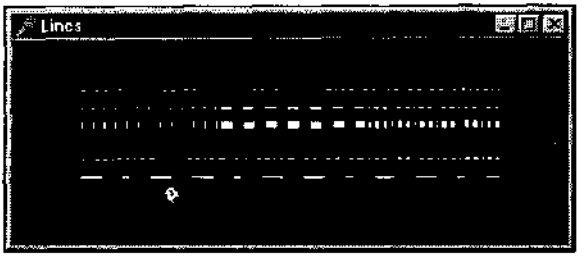


Рис. 2.1. Несколько готовых шаблонов штриховых линий

Содержательная часть кода перерисовки окна выглядит так:

```
    glColor3f (1.0, 1.0, 1.0); // все отрезки рисуются белым
    // вторая строка: рисуется 3 отрезка, все с различной штриховкой
    glEnable (GL_LINE_STIPPLE);
    glLineStipple (1, $0101);  // точечный
    drawOneLine (50.0, 125.0, 150.0, 125.0);
    glLineStipple (1, $00FF);  // штрихи
    drawOneLine (150.0, 125.0, 250.0, 125.0);
    glLineStipple (1, $1C47);  // штрихпунктир
    drawOneLine (250.0, 125.0, 350.0, 125.0);

    // третья строка: рисуется три широких отрезка с той же штриховкой
    glLineWidth (5.0); // задаем ширину линии
    glLineStipple (1, $0101);
    drawOneLine (50.0, 100.0, 150.0, 100.0);
    glLineStipple (1, $00FF);
    drawOneLine (150.0, 100.0, 250.0, 100.0);
    glLineStipple (1, $1C47);
    drawOneLine (250.0, 100.0, 350.0, 100.0);
    glLineWidth (1.0);
// В первой строке рисуется 6 отрезков, шаблон "пунктир/точка/пунктир",
// как части одного длинного отрезка, без вызова процедуры drawOneLine
    glLineStipple (1, $1C47);
    glBegin (GL_LINE_STRIP);
      for i := 0 to 6 do
        glVertex2f ( 2 * (50.0 + (i * 50.0)) / ClientWidth - 1.0,
        75.0 / ClientHeight);
    glEnd;
// четвертая строка — аналогичный результат, но 6 отдельных отрезков
    for i := 0 to 5 do
        drawOneLine (50.0 + i * 50.0, 50,0, 50.0 + (i+1) *50.0, 50.0);
```

```
// пятая строка — рисуется один штрихпунктирный отрезок, множитель = 4
    glLineStipple (5, $1C47);
    drawOneLine (50.0, 25.0, 350.0, 25.0);
```

В заключение разговора по поводу линий посмотрите проект из подкаталога Ex17 — модифицированный пример с отслеживанием позиции курсора. Теперь картинка напоминает бенгальский огонь — рисуются отрезки случайного цвета, длины, штриховки:

```
glEnable (GL_LINE_STIPPLE);
For i := 1 to 100 do begin
    glColor3f (random, random, random);
    glLineStipple (random (5), random ($FFFF));
    glBegin (GL_LINES);
      glVertex2f (xpos, ypos);
      glVertex2f (xpos + 0.5 * random * sin (random (360)),
                  ypos + 0.5 * random * cos (random (360) ) ;';
    glEnd;
end;
```

Треугольник

Закончив с линиями, перейдем к треугольникам — примитиву, задаваемому константой GL_TRIANGLES. В этом примитиве последующие вершины берутся триплетами, тройками, по которым строится каждый отдельный треугольник.

Следующий код служит иллюстрацией рисования одного треугольника (проект из подкаталога Ex18).

```
glBegin (GL_TRIANGLES);
   glVertex2f (-1, -1);
   glVertex2f (-1, 1);
   glVertex2f (1, 0);
glEnd;
```

Для рисования правильного шестиугольника из отдельных треугольников код должен выглядеть так (готовую программу можете найти в подкаталоге Ex19):

```
glBegin (GL_TRIANGLES);
   For i := 0 to 5 > do begin
       glVertex2f (0, 0);
       glVertex2f (0.5 * cos  (2 *  Pi * i ,'6) ,
                   0.5 * sin  (2 *  Pi * i ,'6) );
```

```
        glVertex2f (0.5 * cos (2 * Pi * (i + 1) / 6),
                    0.5 * sin (2 * Pi * (i + 1) / 6));
    end;
glEnd;
```

В качестве опорных точек выбраны шесть точек на окружности.

Надеюсь, здесь не требуется дополнительных пояснений, и мы можем перейти к примитиву, задаваемому константой GL_TRIANGLE_STRIP: связанная группа треугольников. Первые три вершины образуют первый треугольник, вершины со второй по четвертую — второй треугольник, с третьей по пятую — третий и т. д.

Проект из подкаталога Ex20 нарисует флажок, образованный наложением двух треугольников (рис. 2.2):

```
glBegin (GL_TRIANGLE_STRIP);
   glVertex2f (1,1);
   glVertex2f (-1, 1);
   glVertex2f (-1, -1);
   glVertex2f (1, -1);
glEnd;
```

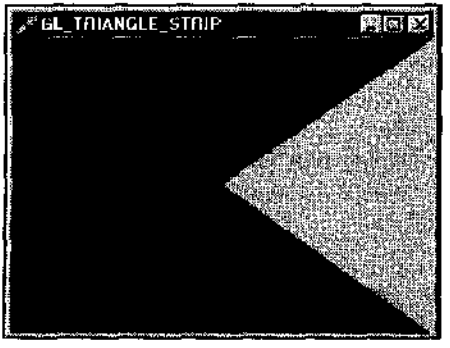


Рис. 2.2. Флаг получается наложением двух отдельных треугольников

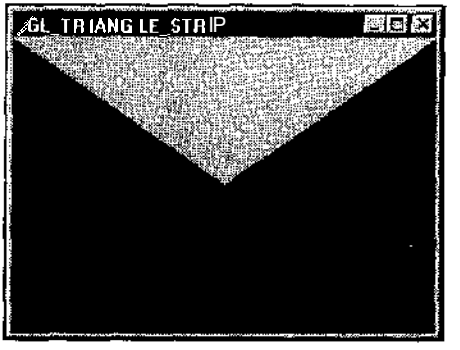


Рис. 2.3. Эту картинку попробуйте нарисовать самостоятельно

А сейчас для тренировки попробуйте изменить код так, чтобы была нарисована такая же картинка, как на рис. 2.3.

Попробуем поэкспериментировать с нашей программой: будем рисовать треугольники разного цвета (проект из подкаталога Ex21):

```
glBegin (GL_TRIANGLE_STRIP);
   glColor3f (0.0, 0.0, 1.0);
   glVertex2f (1, 1);
   glVertex2f (-1, 1);
```

```
   glColor3f (1.0, 0.0, 0.0);
   glVertox2f (-1, -1) ;
   glVertex2f (1, -1) ;
glEnd;
```

Результат окажется неожиданным и живописным: у фигуры возникнет плавный переход синего цвета в красный (рис. 2.4).

Вызов перед командными скобками функции glShadeModel(GL_FLAT), "задающей правило тонирования, избавит от градиентного заполнения фигуры, но результат все равно будет неожиданным — оба треугольника станут красными, т. е. цвета второго из них (проект находится в подкаталоге Ex22). Ознакомившись со справкой по этой команде, мы обнаружим, что для связанных треугольников наложение цветов происходит именно по правилу старшинства цвета второго примитива. Здесь же узнаем, что по умолчанию тонирование задается плавным, как и получилось в предыдущей программе.

В продолжение экспериментов код рисования приведем к следующему виду:

```
glBegin (GL_TRIANGLE_STRIP);
   glVertex2f (random * 2 - 1, random * 2 - 1) ;
   For i := 0 to 9 do begin
       glColor3f (random, random, random);
       glVertex2f (random * 2 - 1, random * 2 - 1) ;
       glVertex2f (random * 2 - 1, random * 2 - 1) ;
   end;
glEnd;
```

Результат получается также интересный: на экране рисуются калейдоскопические картинки, некоторые из них вполне могут порадовать глаз. Пример одной из таких картинок приведен на рис. 2.5.

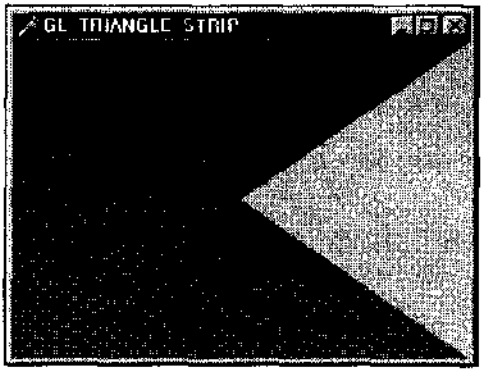


Рис. 2.4. Плавный переход цвета

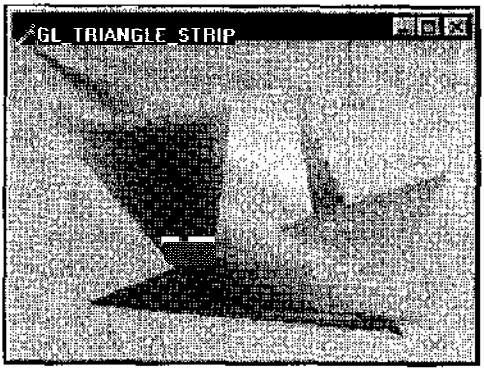


Рис. 2.5. Результат работы программы из каталога Chapter2\Ex23

Предлагаю вам создать обработчик нажатия клавиши, включающий в себя вызов функции Refresh, тогда по нажатию любой клавиши картинка будет меняться (в подкаталоге Ex23 можете взять готовый проект).

Теперь добавьте в программу вызов glShadeModel с аргументом GL_FLAT И обратите внимание, что треугольники окрашиваются попарно одинаковым цветом. Позже мы поговорим о том, по какому правилу отображаемые примитивы накладываются друг на друга, а пока просто понаблюдайте, как рисуется подобие смятой бумажной змейки (треугольники последовательно накладываются друг на друга).

Рисование шестиугольника путем наложения треугольников может быть реализовано с помощью следующего кода (пример располагается в подкаталоге Ex24):

```
glBegin (GL_TRIANGLE_STRIP);
   For i := 0 to 6 do begin
       glColor3f (random, random, random) ;
       glVertex2f (0, 0) ;
       glVertex2f (0.5 * cos (2 * Pi * i / 6},
                   0.5 * sin (2 * Pi * i / 6) ) ;
   end;
glEnd;
```

Обязательно посмотрите результат работы этой программы, а также потренируйтесь в выборе различных моделей тонирования.

Проект из подкаталога Ex25 тоже советую не пропустить: здесь находится небольшая модификация предыдущего примера. Увеличение количества опорных точек привело к симпатичному результату: рисуется окружность с подобием интерференционной картинки на поверхности компакт-диска. Картинка меняется при нажатии клавиши и при изменении размеров окна.

Следующий примитив, определяемый константой GL_TRIANGLE_FAN, также задает последовательно связанные треугольники, однако фигура строится по другому принципу: первая вершина является общей для всех остальных треугольников, задаваемых перечислением вершин, т. е. треугольники связываются наподобие веера.

Для построения шестиугольника с использованием такого примитива цикл надо переписать так (проект находится в подкаталоге Ex26):

```
glBegin (GL_TRIANGLE_FAN);
   glVertex2f (0, 0} ;        // вершина, общая для всех треугольников
   For i := 0 to 6 do begin
       glColor3f (random, random, random);
       glVertex2f (0.5 *cos (2 * Pi * i / 6},
                   0.5 * sin (2 * Pi * i / 6)};
   end;
glEnd;
```

Теперь поговорим о режимах вывода многоугольников.

Для устранения ступенчатости многоугольников используется команда glEnable С аргументом GL_POLYGON_SMOOTH.

Если в примеры на треугольники перед командными скобками поместить строку

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
```

то треугольники будут рисоваться контурно — только линии границ (проект находится в подкаталоге Ex27).

Замечание

Запомните, что команда glPolygonMode задает режим воспроизведения для всех типов многоугольников.

Ширину линий контура можно варьировать с помощью glLineWidth, пунктирные ЛИНИИ КОНТура задаются командой glLineStipple.

Команда giPolygonMode позволяет выводить вместо заполненных и контурных многоугольников только их вершины, если ее вторым аргументом взять константу GL_POINT (не путать с GL_POINTS!). Размеры точек вершии и наличие сглаживания у них можно задавать так же, как и для обычных точек. По умолчанию многоугольники строятся заполненными (включен режим GL_FILL).

Команда glPolygonMode заставляет обратить внимание на порядок перечисления вершин, задающий лицевую и обратную сторону рисуемых фигур. Этот порядок для рассматриваемых плоскостных построений задает пока только то, какую сторону рисуемой фигуры мы видим, что в данном случае не особо существенно, но для будущего важно хорошо разобраться в этом вопросе.

В программе из подкаталога Ex28 рисуется все тот же шестиугольник, но вершины перечислены в обратном порядке.

Контурный режим, включенный для лицевой стороны вызовом

```
glPolygonMode(GL_FRONT, GL_LINE);
```

не приводит ни к каким изменениям в рисунке, поскольку мы видим не лицевую, а изнаночную сторону объекта, режим рисования которой мы не меняли, следовательно, он остался принятым по умолчанию, т. е. сплошной заливкой.

Сейчас самое время поэкспериментировать с режимами воспроизведения многоугольников. В последней программе задайте различные режимы и посмотрите, к каким изменениям это приведет.

Мы изучили примитивы "точка", "линия", "треугольник". В принципе, этих примитивов вполне достаточно, чтобы нарисовать все что угодно, пусть

подчас и чересчур громоздким способом. Даже более того, остальные примитивы фактически являются усовершенствованными треугольниками и строятся из треугольников, чем и вызваны их некоторые ограничения. Построения на основе треугольников являются оптимальными по своим скоростным показателям: треугольники строятся наиболее быстро, и именно этот формат чаще всего предлагается для аппаратного ускорения.

Замечание

По возможности старайтесь использовать связанные треугольники.

Но наш разговор о примитивах OpenGL был бы, конечно, не полным, если бы мы остановились на данной этапе и оставили без рассмотрения оставшиеся примитивы-многоугольники.

Многоугольник

Для рисования прямоугольника на плоскости можно воспользоваться командой `glRectf`. Это одна из версий команды `glRect`. Ее аргументом являются координаты двух точек — противоположных углов рисуемого прямоугольника. Посмотрите проект, располагающийся в подкаталоге Ex29 — простой пример на построение прямоугольника с использованием этой команды.

Замечание

При использовании `glRect` необходимо помнить, что координата по оси Z в текущей системе координат для всех вершин равна нулю.

Константа `GL_QUADS` задает примитив, когда перечисляемые вершины берутся по четыре и по ним строятся независимые четырехугольники.

Следующий код — иллюстрация использования этого примитива: строятся два независимых четырехугольника (взято из проекта, располагающегося в подкаталоге Ex30).

```
glBegin (GL_QUADS);
   glColor3f (random, random, random) ;
   glVertex2f (-0.6, 0.2);
   glVertex2f (-0.7, 0.7);
   glVertex2f (0.1, 0.65);
   glVertex2f (0.25, -0.78);
   glColor3f (random, random, random) ;
   glVertex2f (0.3, -0.6);
   glVertex2f  (0.45,  0.7);
   glVertex2f (0.8, 0.65);
   glVertex2f  (0.9, -0.8);
glEnd;
```

Результат работы программы иллюстрирует рис. 2.6.

Примитив, задаваемый константой GL_QUAD_STRIP, состоит из связанных четырехугольников. Первый четырехугольник формируется из вершин номер один, два, три и четыре. Второй четырехугольник в качестве опорных берет вторую, третью, пятую и четвертую вершины. Ну и так далее.

Если в предыдущем примере поменять константу на GL_QUAD_STRIP, как это сделано в проекте из подкаталога Ex31, то изображение в окне получится таким, как на рис. 2.7.

Для рисования выпуклого многоугольника используется примитив GL_POLYGON. Многоугольник строится из связанных треугольников с общей вершиной, в качестве которой берется первая среди перечисляемых в командных скобках. Код для рисования шестиугольника может выглядеть так:

```
glBegin (GL_POLYGON);
   For i := 0 to 6 do
     glVertex2f (0.5 * cos (2 * Pi * i / 6), 0.5 * sin (2 * Pi * i / 6));
glEnd;
```

Обратите внимание, что в отличие от предыдущих реализаций этой задачи вершины шестиугольника последовательно соединяются не с центром окна, а с крайней правой вершиной, указанной самой первой. Это становится хорошо заметным, если менять цвет для каждой вершины, как это сделано в проекте из подкаталога Ex32.

Замечание

Для воспроизведения треугольников и четырехугольников лучше не использовать **примитив** GL_POLYGON, **в таких случаях оптимальным будет использование примитивов, специально предназначенных для этих фигур.**

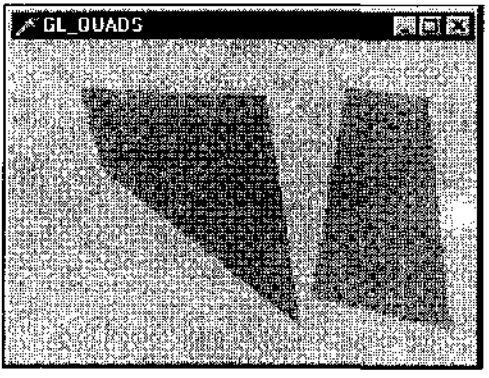


Рис. 2.6. Для построения независимых четырехугольников используется примитив, задаваемый константой GL_QUADS

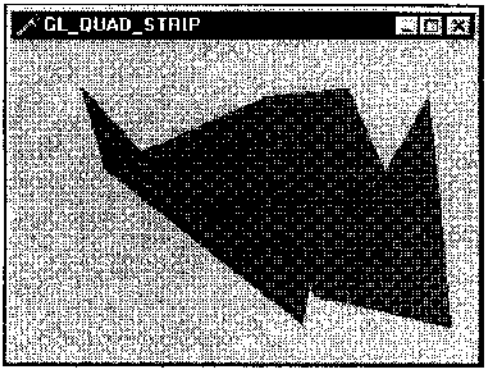


Рис. 2.7. То же, что и рис. 2.6, но константа GL_QUAD_STRIP

Попробуем чуть усложнить наши построения: зададимся целью нарисовать фигуру, как на рис. 2.8.

Поскольку примитив GL_POLYGON позволяет строить только выпуклые многоугольники, построение всей фигуры разбиваем на две части, каждая из которых представляет собой выпуклый многоугольник. Соответствующий проект располагается в подкаталоге Ex33, а на рис. 2.9 я пометил эти части разными цветами.

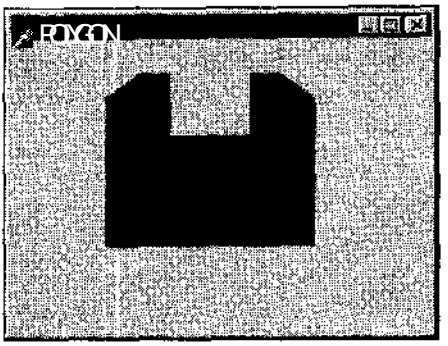


Рис. 2.8. Невыпуклый многоугольник

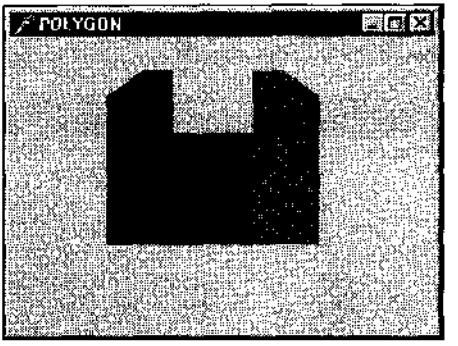


Рис. 2.9. Фигуру для построения разбиваем на несколько частей

Если же попытаться нарисовать эту фигуру "за один присест", то картинка может получиться, например, такой, как на рис. 2.10.

Подкаталог Ex34 содержит проект, в котором эта же фигура строится с использованием связанных четырехугольников и только одной пары командных скобок. При этом построении также имеются потери эффективности: некоторые четырехугольники местами строятся на уже закрашенных местах. Для проверки достаточно включить контурный режим рисования многоугольников, причем при включении такого режима выборочно для лицевой или изнаночной стороны можно заметить, что одни многоугольники мы видим с лицевой стороны, а другие — с изнаночной.

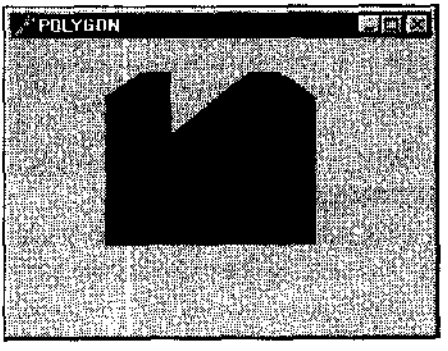


Рис. 2.10. Так выглядит попытка построения фигуры с использованием одной пары командных скобок

Замечание

Важно запомнить: базовые команды OpenGL предназначены для построения только выпуклых фигур, поэтому сложные фигуры чаще всего рисуются этапами, по частям.

Возможно, вы уже задались вопросом, как нарисовать фигуру с внутренним отверстием. Подкаталог Ex35 содержит проект для рисования диска, а Ex36 содержит проект, в котором внутри квадрата рисуется круглое отверстие. Всю фигуру разбиваем на четыре четверти, каждая из которых — группа связанных четырехугольников. Если включить режим контурного рисования многоугольников, картинка получится такой, как на рис. 2.11.

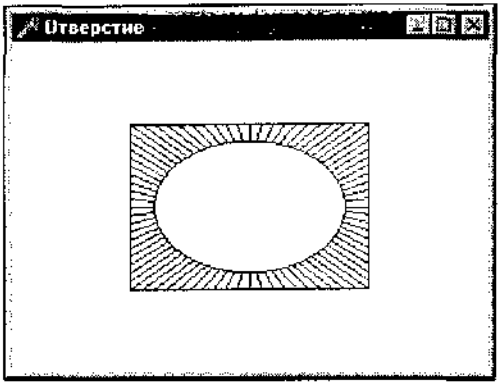


Рис. 2.11. Так разбивается фигура, если требуется нарисовать внутреннее отверстие

Если вы имеете опыт работы в графических редакторах, такой подход, возможно, вас несколько разочаровал. Конечно, было бы гораздо удобнее, если бы имелась, например, функция закраски замкнутой области, как это принято в большинстве графических редакторов. Но мы имеем дело с низкоуровневой библиотекой, и она не предоставляет подобных готовых функций.

Замечание

Как мы увидим дальше, есть несколько способов решения задачи построения сложных объектов — невыпуклых многоугольников или объектов, содержащих отверстия, — однако самый быстрый (в смысле скорости воспроизведения, а не кодирования) способ состоит в том, чтобы вы сами полностью расписали алгоритм построения на основе многоугольников, а в идеале — с использованием только треугольников.

Утешением может стать то, что аппаратные возможности растут стремительно и подобные рекомендации теряют свою актуальность.

Сейчас самое время обратить ваше внимание на очень важный факт. Если в примере на отверстие включить режим сглаживания многоугольников:

```
gl.Enable(GL_POLYGON_SMOOTH);
```

то построение фигуры замедляется, что хорошо заметно при изменении размеров окна, когда происходит его перерисовка. Использование режимов, их включение и отключение, может сильно сказаться на скорости воспроизведения, о чем необходимо постоянно помнить.

Команда *glEdgeFlag*

Режим вывода полигонов (так мы будем иногда называть многоугольники) позволяет рисовать контуры фигур или только точки в опорных вершинах фигуры. Когда сложная фигура разбивается на части, контурный режим может испортить картинку: станет заметным поэтапное построение фигуры. В такой ситуации решение может состоять в исключении некоторых вершин из построения границы фигуры, что осуществляется вызовом команды glEdgeFlag. Аргумент этой команды имеет тип Boolean, если точнее — `GLboolean`, и мы в главе 1 говорили о небольшой проблеме с этим типом OpenGL. Как оговаривается в справке, команда дает эффект только в режиме контурного или поточечного вывода многоугольников. Также специально оговаривается возможность использования этой команды внутри командных скобок.

Смысл команды следующий: вершины, указываемые после вызова команды с аргументом False, при построении границы многоугольника не учитываются, как если бы мы рисовали контур в этом месте прозрачным цветом.

Посмотрите пример, располагающийся в подкаталоге Ex37, в котором наша тестовая фигура рисуется в двух режимах: полная заливка и контурно. Код при этом выполняется один и тот же, но для того, чтобы скрыть от наблюдателя секторное разбиение фигуры, некоторые вершины заключены между строками:

```
glEdgeFlag  (FALSE);
...
glEdgeFlag  (TRUE);
```

Поэтому при контурном режиме эти вершины просто пропускаются.

Режим вывода многоугольников меняется при нажатии пробела, после чего окно перерисовывается.

В качестве упражнения я бы посоветовал удалить строки, в которых вызывается команда `glEdgeFlag`, и посмотреть на получающийся результат.

Массивы вершин

Мы рассмотрели все десять примитивов, имеющихся в нашем распоряжении. Код практических построений, включающих сотни и тысячи отдельных

примитивов, подчас чересчур громоздок, большая часть его в таких случаях — СОТНИ И ТЫСЯЧИ строк С ВЫЗОВОМ команды glVertex.

Библиотека OpenGL располагает средством сокращении кода, базирующимся на использовании массивов вершин. В массиве вершин, т. е. массиве вещественных чисел, задаются координаты опорных вершин, по которым вызовом ОДНОЙ КОМанДЫ gIDrawArrays СТрОИТСЯ последовательность примитивов заданного типа.

Поскольку эта команда не входит в стандарт OpenGL и является его расширением (extension), для получения справки по ней необходимо вызвать КОНтекстную ПОМОЩЬ на СЛОВО glDrawArraysEXT.

У команды gIDrawArrays три аргумента: тип примитива и характеристики используемого массива.

Для использования этой функции надо, как минимум, задать массив вершин, по которым будет строиться множество примитивов. Ссылка на массив вершин создается командой glVertexPointer, также являющейся расширением стандарта. Помимо массива вершин, для украшения картинки будем также использовать массив цвета вершин, ссылка на который задается командой glColorPointer, тоже не входящей в стандарт. Прибавив окончание EXT к именам этих команд, можно получить контекстную подсказку, по которой легко разобраться с их аргументами. Но, к сожалению, для использования массива вершин полученной информации недостаточно, необходимо еще использовать, как минимум, команду glEnableClientState, справку по которой уже невозможно получить никакими ухищрениями. Эта команда аналогична glEnable, но применяется только в контексте массивов вершин. У нее имеется парная команда — glDisableClientState, отключающая использование массива вершин.

В заголовочном файле opengl.pas, поставляемом с Delphi, отсутствуют прототипы команд, не входящих в стандарт OpenGL, а также отсутствует описание констант, используемых такими командами, что немного затруднит наше изучение этой библиотеки.

Обратимся к проекту из подкаталога Ex38.

Массив с именем Vertex содержит координаты четырех точек — углов квадрата, а в массиве цветов colors содержатся соответствующие вершинам значения RGB:

```
Vertex : Array [0..3, 0..1] of GLFloat;
Colors : Array [0..3, 0..2] of GLFloat;
```

Код рисования выглядит так:

```
glVertexPointer(2, GL_FLOAT, 0, @Vertex); // указатель на массив вершин
glColorPointer(3, GL_FLOAT, 0, @Colors);  // указатель на массив цветов
```

```
glEnableClientState(GL_VERTEX_ARRAY);    // массив версии - включаем режим
glEnableClientState(GL_COLOR_ARRAY);     // массив цветов - включаем режим
glDrawArrays(GL_POLYGON, 0, 4);          // рисование множества полигонов
glDisableClientState(GL_COLOR_ARRAY);    // выключаем режимы (Б ЭТОМ
glDisableClientState(GL_VERTEX_ARRAY);   // примере  не обязательно)
```

Значение первого аргумента команды `glVertexPointer` равно двум, поскольку вершины заданы указанием двух координат. То есть этот аргумент задает, по сколько вещественных чисел считывать для каждой точки.
Результат работы программы показан на рис. 2.12.

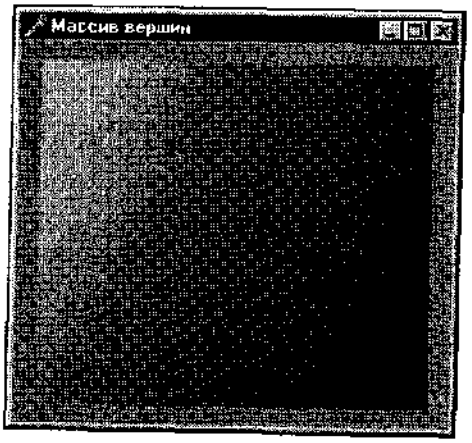


Рис. 2.12. Иллюстрация к примеру на использование массива вершин

Для использования команд-расширений программу пришлось дополнить строками с описанием прототипов процедур:

```
procedure glVertexPointer (size: GLint; atype: GLenum;
          stride: GLsizei; data: pointer); stdcall; external OpenGL32;
procedure glColorPointer (size: GLint; atype: GLenum; stride: GLsizei;
          data: pointer); stdcall; external OpenGL32;
procedure glDrawArrays (mode: GLenum; first: GLint; count: GLsizei);
          stdcall; external OpenGL32;
procedure glEnableClientState (aarray: GLenum); stdcall; external OpenGL32;
procedure glDisableClientState (aarray: GLenum); stdcall; external OpenGL32;
```

Здесь `OpenGL32` — константа, определяемая в модуле opengl.pas.

Потребовалось также задать значение констант, используемых этими процедурами:

```
const
 GL_VERTEX_ARRAY  = $8074;
 GL_COLOR_ARRAY   = $8076;
```

Значения констант и информацию о типах аргументов процедур я почерпнул из заголовочных файлов сторонних разработчиков и перепроверил по содержимому файла gl.h, поставляемым с Visual C++. Информация о команде glEnableClientState взята из описания OpenGL фирмы SGI.

Первым аргументом glDrawArrays может быть любая константа, которую допускается использовать в glBegin. Чтобы лучше разобраться с принципом построения, рекомендую посмотреть результат работы программы с использованием ОТЛИЧНЫХ ОТ GL_POLYGON КОНСТАНТ.

В примере по массиву вершин строится множество полигонов вызовом команды

```
glDrawArrays(GL_POLYGON, 0, 4);  // рисование множества полигонов
```

Эта команда является сокращением следующей последовательности команд (пример из подкаталога Ex39):

```
glBegin (GL_POLYGON);
  glArrayElement(0);
  glArrayElement(1);
  glArrayElement(2);
  glArrayElement(3);
glEnd;
```

Используемая здесь функция glArrayElement (также расширение стандарта) берет в качестве вершины примитива элемент массива с заданным индексом. Индекс, как видно из примера, начинается с нуля.

В продолжение этой темы разберите также проект из подкаталога Ex40 — мою адаптацию и трансляцию под Delphi программы Polygons из книги |![, результат работы которой иллюстрирует рис. 2.13.

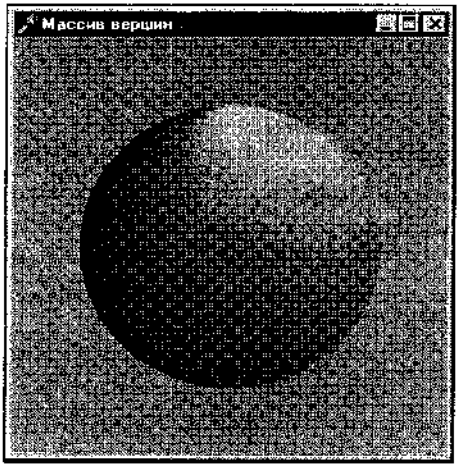


Рис. 2.13. Уже сейчас вы умеете создавать высококачественные изображения

Вы, возможно, задаетесь вопросом, почему в заголовочном файле, поставляемом с Delphi, отсутствуют прототипы процедур, хоть и не входящих в стандарт OpenGL, но документированных разработчиком библиотеки. Ответ на этот вопрос заключается, по-видимому, в том, что этот файл является трансляцией заголовочных файлов gl.h и glu.h, опубликованных SGI, и в него вошло только то, что документировано в этих файлах. Как явствует из заголовка файла, за основу положены файлы версии 1993 года.

Помимо массивов вершин и массива цветов вершин, имеется массив границ — аналог команды glEdgeFlag применительно к массивам вершин. В этом случае необходимо использовать команду glEdgeFlagPointer. Прибавив суффикс EXT, вы можете получить справку по этой команде. В ней. в частности, говорится, что включение режима использования массива границ осуществляется так:

```
glEnable (GL_EDGE_FLAG_ARRAY_EXT);
```

Однако это указание, по-видимому, является ошибочным. *Я* смог воспользоваться массивом флагов границ только с использованием следующего кода:

```
glEnableClientState(GL_EDGE_FLAG_ARRAY);
```

То же самое я могу сказать по поводу аргументов команды.

Посмотрите проект из подкаталога Ex41, где для иллюстрации используется все та же тестовая фигура, изображенная на рис. 2.8.

В ЭТОМ примере фигура ВЫВОДИТСЯ С ИСПОЛЬЗОВанием команды glDrawArrays в двух режимах — сплошной заливкой и контурно. При втором режиме сказывается действие подключения массива границ.

В КОДе ПрОГрамМЫ ДОбавИЛСЯ ПрОТОТИП процедуры glEdgeFlagPointer. Обратите внимание на расхождение в описании прототипа с документацией: у прототипа два аргумента вместо трех. Если попытаться в точности следовать документации, в результате либо не используется массив границ, либо возникает сообщение об ошибке.

Прямое обращение к пикселам экрана

Библиотека OpenGL располагает командами, позволяющими осуществить непосредственный доступ к пикселам экрана. Разберем проект из подкаталога Ex42 — простой пример на вывод блока пикселов на экран вызовом двух команд:

```
glRaster?os2f (-0.25, -0.25);
glDrawPixels(ImageWidth, ImageHeight, GL_RGB, GL_UNSIGNED_BYTE, @ Image);
```

Первая команда задает базовую точку для выводимого блока, вторая осуществляет собственно вывод массива пикселов. Высота при выводе может и

отличаться от реального размера массива, т. е. быть меньше, ширину же для корректного вывода желательно не брать меньше действительного размера массива.

В этом простом примере выводимый массив заполняется значениями RGB так, что в результате получается шахматная доска с сине-красными клетками:

```
const
  ImageWidth = 64;
  ImageHeight = 64;
  ...
  Image: Array [0..ImageHeight-1, 0..ImageWidth-1, 0..2] of GLUbyte;
  ...
{=============================================================}
Создание образа шахматной доски}
procedure TfrmGL.MakeImage;
var
 i, j : Integer;
begin
 For i := 0 to ImageHeight - 1 do
     For j := 0 to ImageWidth - 1 do begin
         If ((i and 8) = 0) xor ((j and 8) = 0)
         then begin
         Image[i][j][0] := 0;    // красный
         Image[i][j][1] := 0;    // зеленый
         Image[i][j][2] := 255;  // синий
         end
         else begin
         Image[i][j][0] := 255;  // красный
         Image[i][j][1] := 0;    // зеленый
         Image[i][j][2] := 0;    // синий
         end;
 end;
end;
```

В развитие темы разберем более интересный пример (подкаталог Ex43), где выводимый массив заполняется следующим образом: создается объект класса TBitmap, в который загружается растр из bmp-файла, последовательно считываются пикселы объекта, и в соответствии со значениями цвета каждого пиксела заполняются элементы массива. Все просто, только надо обратить внимание, что ось Y битовой карты направлена вниз, поэтому элементы массива заполняются в обратном порядке:

```
procedure TfrmGL.MakeImage;
var
 i, j : Integer;
 PixCol : TColor;
```

```
 Bitmap : TBitmap;
begin
 Bitmap := TBitmap.Create;
 Bitmap.LoadFromFile ('Claudia.bmp');
 For i := 0 to ImageHeight - 1 do
 For j := 0 to ImageWidth - 1 do begin
     PixCoi := Bitmap.Canvas.Pixels [j, i];
     // перевод цвета из TColor в цвет для команд OpenGL
     Image[ImageHeight - i - 1][j][0] := PixCol and SFF;
     Image[ImageHeight - i - 1][j][1] := (PixCoi and $FF00) shr 8;
     Image[ImageHeight - i - 1][j][2] := (PixCoi and $FF0000) shr 16;
 end;
 Bitmap.Free;
end;
```

Обратите внимание на строку, задающую выравнивание пикселов — без нее массив пикселов будет выводиться некорректно:

```
glPixelStorei(GL_UNPACK_ALIGNMENT,1);
```

Возможно, при знакомстве с файлом справки вы обратили внимание на команду `glBitMap`, специально предназначенную для вывода битовых массивов. Растр в таком случае может быть монохромным, а не 24-битным. Обычно эта команда используется для вывода текста, мы рассмотрим ее в главе 6.

Выводимые массивы пикселов легко масштабируются вызовом команды `glPixelZoom`, аргументы которой — масштабные множители по осям X и Y.

Для иллюстрации посмотрите проект из подкаталога Ex44, где нажатием клавиш 'X' и 'Y' можно управлять масштабом по соответствующим осям. Обратите внимание, что при отрицательном значении множителей изображение переворачивается по соответствующей оси.

Этот пример демонстрирует также еще одно возможное использование команды glPixelStorei, позволяющей, в частности, прокручивать изображение по вертикали или горизонтали, что в нашей программе осуществляется нажатием клавиш 'R' и 'P'.

Замечание

Учтите, что при прокрутке изображения на величину, превышающую половину растра, происходит аварийное завершение программы.

Команда `glCopyPixels` позволяет копировать часть экрана в текущей позиции, задаваемой `glRasterPos`.

В проекте из подкаталога Ex45 я воспользовался этой командой следующим образом: при нажатой кнопке мыши вслед за указателем мыши перемешает-

ся копия левого нижнего угла экрана. Изображение восстанавливается при каждой перерисовке экрана.

Здесь надо обратить внимание, что, в отличие от всех остальных примеров данной главы, флаги окна не включают двойную буферизацию и что контекст воспроизведения занимается приложением сразу по началу работы и не освобождается до завершения работы. Обработка движения мыши заканчивается не перерисовкой окна, а командой giFiush.

Команда `glPixelTransfer` позволяет задавать режимы вывода пикселов. в частности, задавать цветовые фильтры. В примере, располагающемся в подкаталоге Ex46, нажатием клавиш 'R', 'G', 'B' можно изменить составляющую долю соответствующего цвета.

Замечание

Надо отметить, что команды непосредственного доступа к пикселам экрана работают сравнительно медленно, поэтому используются только в исключительных случаях.

Команда `glReadPixels` позволяет считывать содержимое всего экрана или части его. Это одна из самых важных для нас команд, мы еще неоднократно будем к ней обращаться.

Для иллюстрации ее работы я подготовил проект (подкаталог E.\47), выводящий на экран простейшие стереокартинки. Одна из них показана на рис. 2.14.

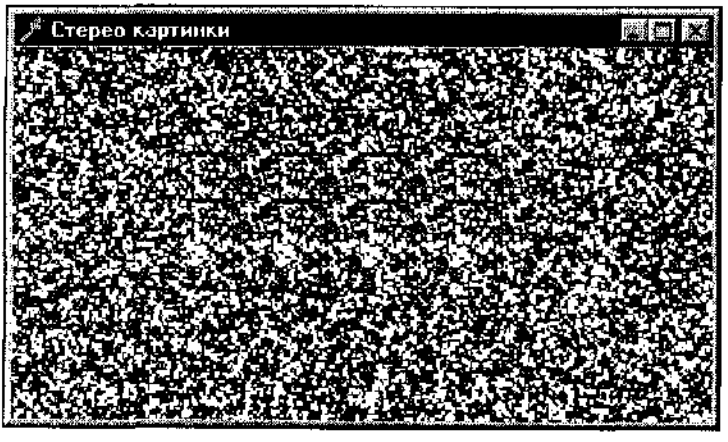


Рис. 2.14. Если вы умеете смотреть на стереокартинки, то увидите здесь парящий квадрат. Картинка рассчитана на размеры монитора, не ручаюсь, что в книге эффект повторится

В примере реализован следующий алгоритм: экран заполняется множеством маленьких черно-белых квадратиков, создающих хаос. Затем небольшая об-

ласть экрана размером 50x50 пикселов считывается в массив и последовательно копируется восемь раз, в две строки по четыре квадрата впритык.

Вот фрагмент кода для первой копии:

```
Pixel : Array [0..50, 0..50, 0..2] of GLUbyte;
...
// считываем в массив часть образа экрана вблизи центра
glReadPixels(round(ClientWidth / 2),round(ClientHeight / 2), 50, 50,
             GL_RGB, GL_UNSIGNED_BYTE, @Pixel);
glRasterPos2f (-0.5, 0.0); // устанавливаем точку вывода
glDrawPixels(50, 50, GL_RGB, GL_UNSIGNED_BYTE, @Pixel); // вывод копии
```

В принципе, достаточно однократного копирования, но тогда требуется особое мастерство для того, чтобы увидеть объемное изображение.

Чтобы скрыть уловку, можно еще набросать на экране немного точек или линий, но разглядеть после этого спрятанный квадрат становится труднее.

При нажатии клавиши <пробел> экран перерисовывается, так что у вас есть возможность подобрать наиболее удачную картинку.

Команда *glGetString*

Вы, наверное, обратили внимание, изучая справки по командам-расширениям, что приложение может получить информацию об имеющихся расширениях OpenGL с помощью команды `glGetString`. Из соответствующей справки выясняем, что результат работы команды — строка типа `PChar`, а аргументом может быть одна из четырех констант.

В зависимости от используемой константы строка содержит информацию о фирме-производителе или способе воспроизведения, версии OpenGL и имеющихся расширениях стандарта.

На основе использования этой функции построен пример, располагающийся в подкаталоге Ex48. Результат работы программы представлен на рис. 2.15.

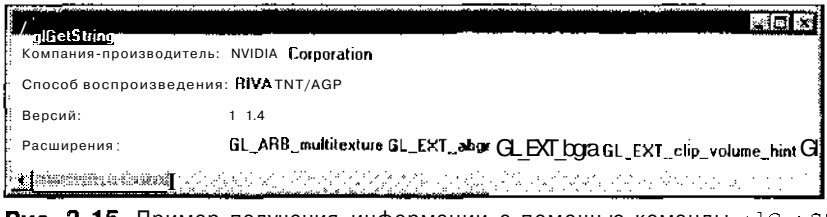


Рис. 2.15. Пример получения информации с помощью команды `glGetString`

При разборе программы обратите внимание на две вещи:

- Использовать команду `glGetString`, точно так же, как и любую другую функцию OpenGL, мы можем только тогда, когда установлен контекст

воспроизведения, хоть в данном случае не осуществляется никакого воспроизведения вообще.

- Результат работы этой функции — длинная строка, заканчивающаяся нулем, поэтому для использования в модуле Delphi конвертируем ее с помощью функции strPas.

Этот пример дает еще одну подсказку, как распознать наличие графического акселератора: при отсутствии такового glGetString с аргументом GL_RENDERER возвращает строку 'GDI Generic'.

В файле справки можно также прочитать, что glGetString с аргументом GL_EXTENSIONS возвращает строку, где через пробел перечисляются поддерживаемые текущим соединением расширения.

В частности, согласно документации, командой glDrawArrays можно пользоваться, только если в расширениях присутствует GL_EXT_vertex_array. Это одно из спорных мест, потому что я с успехом применяю расширение массива вершин, хотя в указанном списке ссылка на его наличие отсутствует. Поэтому я бы не советовал полностью доверяться возвращаемому списку. Не стоит и в программах сразу же прекращать выполнение, если вы опираетесь на расширение стандарта OpenGL, которое не определяется. По-моему, в такой ситуации достаточно просто предупредить пользователя о возможных сбоях в работе программы.

Обработка ошибок

Использующее OpenGL приложение по ходу работы может выполнять некорректные действия, приводящие к ошибкам в работе. Часть ошибок носит фатальный характер, ведущий к аварийному завершению работы приложения. Другие могут и не приводить к такому финалу, но код воспроизведения, содержащий ошибку, не выполняется. Пример такой ошибки, который я уже приводил — неверная константа в качестве аргумента функции glBegin или неверное количество вершин для воспроизведения примитива. Конечно, компилятор не может и не должен распознавать подобные ошибки, но и OpenGL не может отработать такие указания клиента. В документации по OpenGL к каждой команде прикладывается раздел "Errors", содержащий указания, какие ошибки могут возникнуть при использовании этой команды.

Команда OpenGL glGetError возвращает информацию об ошибке в виде одной из семи констант. Обратите внимание, что в свою очередь эта команда сама может генерировать ошибку, если вызывается внутри командных скобок OpenGL.

На использовании этой функции построен пример, располагающийся в подкаталоге Ex49. В программе из-за заведомо неверного аргумента функции glBegin генерируется ошибка библиотеки OpenGL типа GL_INVALID_ENUM.

Соответствующее сообщение выводится в заголовке окна, использовать диалоговые окна в такой ситуации нежелательно, поскольку это приведет к перерисовке окна, т. е. снова будет сгенерирована ошибка.

В программе также введена пользовательская функция, возвращающая строку с описанием ошибки:

```
function GetError : String;
begin
 Case glGetError of
    GL_INVALID_ENUM  : Result := 'Неверный аргумент!';
    GL_INVALID_VALUE : Result := 'Неверное значение аргумента!';
    GL_INVALID_OPERATION : Result := 'Неверная операция!';
    GL_STACK_OVERFLOW : Result := 'Переполнение стека!';
    GL_STACK_UNDERFLOW : Result := 'Потеря значимости стека!';
    GL_OUT_OF_MEMORY : Result := 'Не хватает памяти!';
    GL_NO_ERROR : Result := 'Нет ошибок.';
 end;
end;
```

Масштабирование

Мы уже знаем, что границы области вывода лежат в пределах от —1 до 1. Это может привести к неудобству при подготовке изображений. К счастью. OpenGL предоставляет удобное средство на этот случай — масштабирование.

Разберем его на примере программы построения фигуры, показанной на рис. 2.8. Для изменения масштаба используется команда `glScalef` с тремя аргументами, представляющими собой масштабные множители для каждой из осей.

Например, если перед командными скобками вставим строку:

```
glScalef (0.5, 0.5, 1.0!;,
```

то будет нарисована уменьшенная в два раза фигура (готовый проект располагается в подкаталоге Ex50).

После команд рисования необходимо восстановить нормальный масштаб, т. е. в данном случае добавить строку:

```
glScalef (2.0, 2.0, 1.0};.
```

Есть и другой способ запоминания/восстановления текущего масштаба, но о нем мы поговорим позднее.

Восстанавливать масштаб необходимо для того, чтобы каждое последующее обращение к обработчику перерисовки экрана не приводило бы к последовательному уменьшению/увеличению изображения. В принципе, можно ис-

пользовать и флаги для того, чтобы обратиться к строке единственный раз в ходе работы приложения.

Масштабные множители могут иметь отрицательные значения, при этом изображение переворачивается по соответствующей оси. Иллюстрирующий это свойство проект находится в подкаталоге Ex51.

При двумерных построениях значение коэффициента по оси Z безразлично, единица взята без особых соображений.

Поворот

Для поворота используется команда glRotatef с четырьмя аргументами: угол поворота, в градусах, и вектор поворота, три вещественных числа.

Для двумерных построений наиболее нагляден поворот по оси Z, чем я и пользуюсь в приводимых примерах.

В предыдущем примере перед командными скобками вставьте строку:

```
glRotatef (5, 0.0, 0.0, 1.0);
```

и создайте обработчик события KeyPress с единственной командой Refresh. Теперь при нажатии любой клавиши окно перерисовывается, при этом каждый раз фигура поворачивается на пять градусов по оси Z (проект из подкаталога Ex52).

Здесь надо обратить внимание на две вещи: на то, что при положительном значении компоненты вектора поворот осуществляется против часовой стрелки и то, что важно не само значение компоненты, а ее знак и равенство/неравенство ее нулю.

Хотя мы пока ограничиваемся плоскостными построениями, поворот по любой из осей сказывается на воспроизводимой картинке. Проверьте: при повороте по осям X и Y мы получаем правильную картинку в проекции с учетом поворота по осям.

Поворот часто используется при построениях, поэтому важно разобраться в нем досконально. Точно так же, как было с масштабом, поворот действует на все последующие команды воспроизведения, так что при необходимости текущее состояние восстанавливается обратным поворотом.

Например, если надо нарисовать повернутый на 45 градусов квадрат, т. с. ромб, то код должен выглядеть так (готовый проект можете взять в подкаталоге Ex53):

```
glRotatef (45, 0.0, 0.0, 1.0);
glBegin (GL_POLYGON);
   glVertex2f (-0.6, -0.1);
   giVertex2f (-0.6, 0.4);
```

```
   glVertex2f (-0.1, 0.4);
   glVertex2f (-0.1,-G.1);
glEnd;
glRotatef (-45, 0.0, 0.0, 1.0);
```

Этот пример очень занимателен и вот почему. Удалим восстановление угла поворота и запустим приложение. Увидим не ромб, а квадрат. При внимательном рассмотрении обнаружим, что квадрат был повернут дважды. Произошло это потому, что сразу после появления окна па экране (функция API ShowWindow) происходит его перерисовка (функция API UpdateWindow). Если вы были внимательны, то могли заметить, что такой же эффект наблюдался и в предыдущем примере.

При выполнении операции поворота можно спросить: поворачивается изображение или точка зрения? Мы будем для ясности считать, что поворачивается именно изображение. Следующий пример пояснит это.

Замечание

Точный ответ такой: все объекты в OpenGL рисуются в точке отсчета системы координат, а команда glRotate осуществляет поворот системы координат.

Нарисуем две фигуры: квадрат и ромб, причем ромб получим путем поворота квадрата. Текст программы будет такой (проект из подкаталога Ex54):

```
glRotatef (45, 0.0, 0.0, 1.0);
glBegin (GL_POLYGON);
   glVertex2f (-0.6, -0.1};
   glVertex2f (-0.6, 0.4);
   glVertex2f (-0.1, 0.4);
   glVertex2f (-0.1, -0.1);
glEnd;
glRotatef (-45, 0.0, 0.0, 1.0);
glBegin (GL_POLYGON);
   glVertex2f (0.1, -0.1);
   glVertex2f (0.1, 0.4);
   glVertex2f (0.6, 0.4);
   glVertex2f (0.6, -0.1);
glEnd;
```

Точно так же нам придется поступать всегда, когда на экране присутствует несколько объектов, повернутых относительно друг друга: перед рисованием очередного объекта осуществлять поворот, а после рисования — возвращать точку зрения или осуществлять следующий поворот с учетом текущего положения точки зрения.

Пожалуйста, будьте внимательны! Начинающие пользователи OpenGL постоянно задают вопрос, как повернуть примитив, не поворачивая остальные примитивы. Еще один раз перечитайте предыдущий абзац.

В заключение разговора о повороте рассмотрите проект (подкаталог Ex55). основанный на примере с диском. При нажатой кнопке или движении курсора происходит перерисовка окна и поворот диска на 60 градусов.

Чтобы вы могли оценить преимущества использования "низкоуровневых" приемов, окно перерисовывается в этих случаях по-разному:

```
procedure TfrmGL.FormKeyPress(Sender: TObject; var Key: Char);
begin
 Refresh
end;
procedure TfrmGL.FormMouseMove (Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  InvalidateRect (Handle, nil, False);
end;
```

При нажатии кнопки хорошо видно мерцание на поверхности окна, которого не появляется при движении указателя мыши по его поверхности.

Перенос

Перенос точки зрения (системы координат) осуществляется командой `glTranslatef` с тремя аргументами — величинами переноса для каждой из осей.

Все сказанное по поводу восстановления точки зрения справедливо и в отношении переноса.

Рассмотрите пример из подкаталога Ex56, иллюстрирующий использование переноса системы координат. Думаю, особых пояснений здесь не потребуется, поэтому перейдем к вопросу о совместном использовании поворота и переноса — это чаще всего и используется при построениях.

Программа из подкаталога Ex57 строит узор, показанный па рис. 2.16.

Стоит разобрать этот пример подробнее. В цикле шесть раз происходит перенос и поворот системы координат:

```
glTranslatef (-0.3, 0.3, 0.01;
glRotatef (60, 0, 0, 1);
```

Кружочки рисуются в текущей точке отсчета системы координат, относительно которой происходят каждый раз преобразования. По завершении цикла точка зрения возвращается точно в начальное положение, поэтому дополнительных манипуляций с системой координат не требуется.

Перед циклом делаем перенос для выравнивания картинки на экране:

```
glTranslatef (0.4, 0.1, 0.0);
```

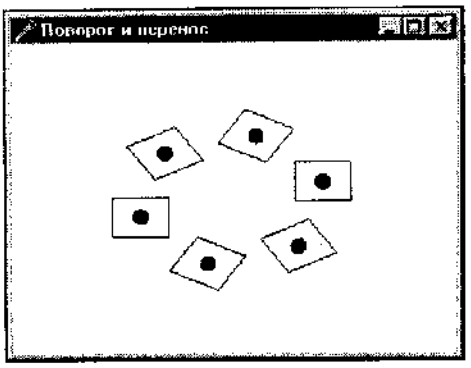


Рис. 2.16. Пример на комбинацию поворота и переноса

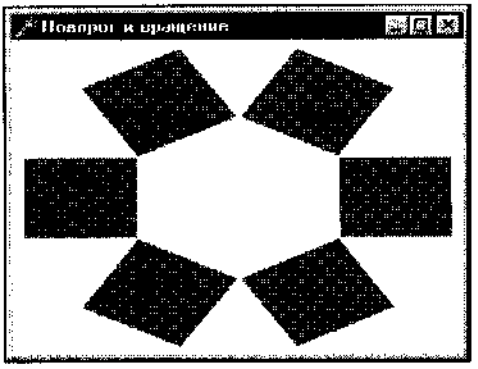


Рис. 2.17. Еще один пример на поворот и перенос

После цикла, конечно, требуется восстановить первоначальное положение системы координат:

```
glTranslatef (-0.4, -0.1, 0.0);
```

Разобравшись с этим примером, перейдите к примеру, располагающемуся в следующем подкаталоге, Ex58. Здесь строятся шесть квадратов по кругу, как показано на рис. 2.17.

Как и все предыдущие, этот пример тоже не отличается особой сложностью. В цикле перед рисованием очередного квадрата смещаем и поворачиваем систему координат:

```
glTranslatef (0.7 * cos (Pi * i / 3), 0.7 * sin (Pi * i / 3), 0.0);
glRotatef (60 * i, 0, 0, 1);
```

а после рисования очередного квадрата делаем обратные действия:

```
glRotatef (60 * i, 0, 0, 1);
glTranslatef (0.7 * cos (Pi * i / 3), -0.7 * sin (Pi * i / 3), 0.0);
```

Все, надеюсь, просто и понятно. Здесь только надо хорошенько уяснить, что порядок манипуляций с системой координат поменять нельзя: вначале перенос, затем поворот, по окончании рисования — в обратном порядке: поворот, затем перенос. Если поменять порядок в любой из пар этих действий либо в обеих парах, будет рисоваться другая картинка — обязательно проверьте это самостоятельно.

Сохранение и восстановление текущего положения

По предыдущим примерам вы должны были очень хорошо уяснить, насколько важно восстанавливать текущую систему координат. При каждой

следующей перерисовке окна она сдвигается и поворачивается, поэтому нам приходится постоянно следить, чтобы после рисования система координат вернулась в прежнее место. Для этого мы пользовались обратными поворотами и перемещениями. Такой подход неэффективен и редко встречается в примерах и профессиональных программах, т. к. приводит к потерям скорости воспроизведения. Если вы прочитаете справку по функциям `glRotatef` и `glTranslatef`, то узнаете, что эти функции реализуются как перемножение текущей матрицы на матрицы попорота и переноса, соответственно.

Обычно в состоянии, к которому потребуется затем вернуться, запоминают значение текущей матрицы. Вместо того чтобы заново перемножать матрицы и тратить время на операции, связанные с математическими расчетами, возвращаемся сразу к запомненному значению.

Команды `glPushMatrix` и `glPopMatrix` позволяют запомнить и восстановить текущую матрицу. Эти команды оперируют со стеком, то есть, как всегда в подобных случаях, можно запоминать (проталкивать) несколько величин, а при каждом восстановлении (выталкивании) содержимое стека поднимается вверх на единицу данных.

Использование этих функций делает код более читабельным, а воспроизведение — более быстрым, поэтому во всех оставшихся примерах мы. как правило, не будем производить малоэффективные действия по обратному перемножению матриц, а будем опираться на использование рассмотренных команд.

Для большей ясности рассмотрите пример из подкаталога Ex59 — модификацию примера на шесть квадратов. В цикле перед операциями поворота и переноса запоминаем текущую систему координат (обращаемся к `glPushMatrix`), а после рисования очередного квадрата — восстанавливаем ее (вызываем `glPopMatrix`). Надеюсь, понятно, что в стек помещается и извлекается из пего пять раз одна и та же матрица, причем в этой программе в стеке содержится всегда не больше одной матрицы.

Библиотека OpenGL позволяет запоминать и возвращать не только текущую систему координат, но и остальные параметры воспроизведения.

Посмотрите документацию по функции `glGet`, а конкретно, список модификаций этой функции, различающихся но типу аргумента в зависимости от запоминаемого параметра. Обратите внимание, что второй аргумент функции — всегда указатель на переменную соответствующего типа, поэтому все функции заканчиваются на "v". Первый аргумент — символическая константа, указывающая, какой параметр запоминаем.

Закрепим изученный материал разбором проекта из подкаталога Ex60.

Рисуем три точки, первую и третью одинаковым цветом. Код следующий:

```
glColor3f  (1.0,  0.0, 0.0);
glGetFloatv (GL_CURRENT_COLOR, @Colox);
```

```
glBegin (GL_POINTS);
    glVertex2f (-0.25, -0.25);
    glColor3f (0.0, O.G, 1.0);
    glVertex2f (-0.25, 0.25);
    glCclor3f (Color [0], Color [1], Color [2]);
    glVertex2f (0.25, 0.25);
glEnd;
```

Для сохранения текущего цвета используется переменная Color типа TGLArrayf3 (массив трех вещественных чисел). В качестве пояснения напомню, что вторым аргументом glGet должен быть указатель на переменную, в которой сохраняется значение параметра.

Первые шаги в пространстве

Во всех предыдущих примерах для задания вершин мы использовали версию команды glVertex с указанием двух координат вершин. Третья координата, по оси Z, устанавливалась нулевой.

Для дальнейшего знакомства с OpenGL нам необходимо выяснить одну важную вещь, для чего попробуем рисовать примитивы с заданием третьей координаты, не равной нулю.

Взгляните па пример из подкаталога Ex61. Здесь рисуется треугольник со значением координаты вершин по Z равным текущему значению переменной h. При нажатии клавиш <пробел>+<Shift> значение этой переменной увеличивается на единицу, если нажимать просто <пробел>, то значение переменной на единицу уменьшается. Значение переменной выводится в заголовке окна. После этого картинка перерисовывается.

Поработав с этим примером, вы можете обнаружить, что пока координата по оси Z не превышает по модулю единицу, треугольник виден наблюдателю. Вы можете уменьшить шаг изменения переменной h, чтобы повысить достоверность вывода, но результат окажется таким же: как только прпмпгпв выходит за пределы воображаемого куба с координатами вершин, имеющими по модулю единичное значение, примитив перестает быть пилимым.

Точнее, нам становится не видна часть примитива, выходящая за пределы этого куба. В примере из подкаталога Ex62 мы манипулируем координатой только одной из вершин треугольника, остальные не перемешаются и располагаются точно на плоскости. При движении третьей вершины в пространстве мы видим только ту часть треугольника, которая помещается в куб.

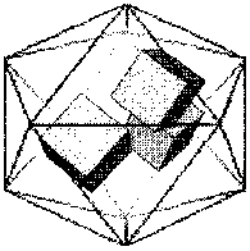
Этот пример иллюстрирует еще одну важную вещь. Помимо треугольник;"!, здесь строится квадрат. Первым строится квадрат, вторым — треугольник. Треугольник частично перекрывает квадрат так. как если бы ом был нарисован сверху. Если поменять порядок рисования примитивов, квадрат рисуется "выше" треугольника.

Замечание

Итак, OpenGL воспроизводит только те части примитивов, координаты которых не превышают по модулю единицу. Примитивы с одинаковыми координатами рисуются по принципу: "каждый последующий рисуется поверх предыдущего". Если вы не теряли внимание по ходу наших уроков, то должны были убедиться в правильности второго вывода, еще разбираясь с примером на связанные четырехугольники.

Последний пример этой главы достоин того, чтобы разобраться с ним подробнее. Обратите внимание, что при отрицательном значении переменной h треугольник строится все равно "выше" квадрата, т. е. пока мы не можем получить действительно пространственных построений, учитывающих взаимное положение примитивов в пространстве.

ГЛАВА 3



Построения в пространстве

Главной темой этой главы является трехмерная графика. Начнем мы с обзора методов, позволяющих достичь объемности получаемых образов, а в конце главы будут описаны методы построения анимационных программ.

Глава завершает базовый курс изучения основ OpenGL, и после ее изучения читатель сможет создавать высококачественные трехмерные изображения.

Примеры к главе помещены на дискете в каталоге Chapter3.

Параметры вида

В предыдущей главе мы убедились, что объем сцены ограничен кубом с координатами точек вершин по диагоналям $\{-1, -1, -1)$ и (1, 1, 1). Начнем дальнейшее изучение с того, что увеличим объем этого пространства. Проект из подкаталога Ex01 нам поможет. На сцене присутствует все тот же треугольник, одну из вершин которого можно перемещать по оси Z нажатием клавиши <пробел>, значение координаты вершины выводится в заголовке окна. Теперь мы видим треугольник целиком в пределах большего, чем раньше, объема.

Код перерисовки окна выглядит так:

```
wglMakeCurrent(Canvas.Handle,hrc);
glViewport(0, 0, ClientWidth, ClientHeight);

glPushMatrix;
glFrustum (-1, 1, -1, 1, 3, 10);     // задаем перспективу
glTranslatef(0.0, 0.0, -5.0);        // перенос объекта по оси Z

glClearColor (0.5, 0.5, 0.75, 1.0);
glClear (GL_COLOR_BUFFER_BIT);
```

```
glColor3f (1.0, 0.0, 0.5);
glBegin (GL_TRIANGLES);
  glVertex3f (-1, -1, 0);
  glVertex3f (-1, 1, 0);
  glVertex3f (1, 0, h);
glEnd;
glPopMatrix;

SwapBuffers (Canvas.Handle);
wglMakeCurrent (0, 0);
```

Основная последовательность действий заключена между командами glPushMatrix и glPopMatrix. Если этого не делать, то при каждой перерисовке окна, например, при изменении его размеров, сцена будет уменьшаться в размерах.

Здесь встречается новая для нас команда — glFrustum, задающая параметры вида, в частности, определяющие область воспроизведения в пространстве. Все, что выходит за пределы этой области, будет отсекаться при воспроизведении. Первые два аргумента задают координаты плоскостей отсечении слева и справа, третий и четвертый параметры определяют координаты плоскостей отсечения снизу и сверху. Последние аргументы задают расстояния до ближней и дальней плоскостей отсечения, значения этих двух параметров должны быть положительными — это не координаты плоскостей, а расстояния от глаза наблюдателя до плоскостей отсечения.

Замечание

Старайтесь переднюю и заднюю плоскости отсечения располагать таким образом, чтобы расстояние между ними было минимально возможным: чем меньший объем ограничен этими плоскостями, тем меньше вычислений приходится производить OpenGL.

Теперь все, что рисуется с нулевым значением координаты Z, не будет видно наблюдателю, поскольку ближнюю плоскость отсечения мы расположили на расстоянии трех единиц от глаза наблюдателя, располагающегося в точке (0, 0. 0). Поэтому перед воспроизведением треугольника смещаем систему координат на пять единиц вниз.

Треугольник удалился от наблюдателя, и его вершины располагаются уже не на границе окна, а сместились вглубь экрана.

Замечание

В главе 6 мы узнаем, как соотнести пространственные и оконные координаты, если видовые параметры заданы с помощью команды glFrustum.

Переходим к следующему примеру — проекту из подкаталога Ex02. Отличие от первого примера состоит в том, что команды glPushMatrix и glPopMatrix

удалены, а перед вызовом команды glFrustum стоит вызов команды glLoadIdentity. Будем понимать это как действие "вернуться в исходное состояние". При каждой перерисовке экрана перед заданием видовых параметров это следует проделывать, иначе объем смены будет последовательно отсекаться из предыдущего.

Замечание

Устанавливать видовые параметры не обязательно при каждой перерисовке экрана, достаточно делать это лишь при изменении размеров окна.

Это несложное соображение предваряет следующий пример — проект из подкаталога Ex03. Для повышения надежности работы приложения пользуемся явно получаемой ссылкой на контекст устройства, а не значением свойства Canvas.Handle. Сразу же после получения контекста воспроизведения делаем его текущим в обработчике события Create формы, а непосредственно перед удалением освобождаем контекст в обработчике Destroy.

Теперь такие параметры OpenGL, как цвет фона и цвет примитивов, можно задавать единственный раз — при создании формы, а не выполнять это действие каждый раз при перерисовке экрана. В отличие от всех предыдущих проектов, в данном появляется отдельный обработчик события, связанного с изменением размеров окна. Напомню, раньше при этом событии выполнялся тот же код, что и при перерисовке окна.

Во всех оставшихся примерах, как правило, будет присутствовать отдельный обработчик события, связанного с изменением размеров окна. В этом обработчике задается область вывода и устанавливаются параметры вида, после чего окно необходимо перерисовать:

```
procedure TfrmGL.FormResize(Sender: TObject);
begin
 glViewport (0, 0, ClientWidth, ClientHeight);
 glLoadIdentity;
 glFrustum (-1, 1, -1, 1, 3, 10);     // видовые параметры
 glTranslatef (0.0, 0.0, -5.0);       // начальный сдвиг системы координат
 InvalidateRect(Handle, nil, False);
end;
```

Код, связанный с перерисовкой окна, теперь сокращается и становится более читабельным:

```
procedure TfrmGL.FormPaint(Sender: TObject);
begin
 glClear (GL_COLOR_BUFFER_BIT);

 glBegin (GL_TRIANGLES);
   glVertex3f (-1., -1, 0);
```

```
    glVertex3f (-1, 1, 0);
    glVertex3f (1, 0, h);
  glEnd;

  SwapBuffers(DC);
end;
```

Хоть мы уже и рисуем пространственные картинки, однако почувствовать трехмерность пока не могли. Попробуем нарисовать что-нибудь действительно объемное, например, куб (проект из подкаталога Ex04). Результат работы программы — на рис. 3.1.

Рис. 3.1. В будущем мы получим более красивые картинки, начинаем же с самых простых

Для придания трехмерности сцепе поворачиваем ее по осям:

```
procedure TfrmGL.FormResize(Sender: TObject);
begin
  glViewport (0, 0, ClientWidth, ClientHeight) ;
  glLoadIdentity;
  glFrustum (-1, 1, -1, 1, 3, 10) ;    // задаем перспективу
  // этот фрагмент нужен для придания трехмерности
  glTranslatef (0.0, 0.0, -8.0);       // перенос объекта - ось Z
  glRotatef (30.0, 1.0, 0.0, 0.0);     // поворот объекта - ось X
  glRotatef (70.0, 0.0, 1.0, 0.0);     // поворот объекта - ось Y

  InvalidateRect(Handle, nil, False);
end;
```

Построение куба сводится к построению шести квадратов, отдельно для каждой стороны фигуры:

```
glBegin (GL_QUADS);
  glVertex3f (1.0, 1.0, 1.0);
  glVertex3f (-1.0, 1.0, 1.0);
  glVertex3f (-1.0, -1.0, 1.0);
  glVertex3f (1.0, -1.0, 1.0);
glEnd;
```

```
glBegin (GL_QUADS);
  glVertex3i (1.0, 1.0, -1.0);
  glVertex3f (1.0, -1.0, -1.0);
  glVertex3f (-1.0, -1.0, -1.0);
  glVertex3f (-1.0, 1.0, -1.0);
glEnd;

glBegin (GL_QUADS);
  glVertex3f (-1.0, 1.0, 1.0);
  glVertex3f (-1.0, 1.0, -1.0);
  glVertex3f (-1.0, -1.0, -1.0);
  glVertex3f (-1.0, -1.0, 1.0};
glEnd;

glBegin (GL_QUADS);
  glVertex3f (1.0, 1.0, 1.0);
  glVertex3f (1.0, -1.0, 1.0);
  glVertex3f (1.0, -1.0, -1.0);
  glVertex3f (1.0, 1.0, -1.0);
glEnd;

glBegin (GL_QUADS);
  glVertex3f (-1.0, 1.0, -1.0);
  glVertex3f (-1.0, 1.0, 1.0);
  glVertex3f (1.0, 1.0, 1.0);
  glVertex3f (1.0, 1.0, -1.0);
glEnd;

glBegin(GL_QUADS);
  glVertex3f (-1.0, -1.0, --1.0);
  glVertex3f (1.0, -1.0, -1.0};
  glVertex3f (1.0, -1.0, 1.0);
  glVertex3f (-1.0, -1.0, 1.0);
glEnd;
```

Код получается громоздким, согласен.

Замечание

Для некоторых базовых фигур мы сможем в будущем найти возможность сократить код, но в общем случае объемные объекты строятся именно так, т. е. из отдельных плоских примитивов.

Я хочу предостеречь вас на случай, если вы желаете прямо сейчас потренироваться и нарисовать что-нибудь интересное. Пока что наши примеры сильно упрощены, могут быть только учебной иллюстрацией и не годятся в качестве шаблона или основы для построения более серьезных программ.

Получившаяся картинка действительно трехмерная, но пространственность здесь только угадывается: куб залит монотонным цветом, из-за чего плохо попятно, что же нарисовано. Сейчас это не очень важно, мы только учимся задавать видовые параметры. Для большей определенности в ближайших примерах будем рисовать каркас куба, как, например, в следующем примере — проекте из подкаталога Ex05 (рис. 3.2).

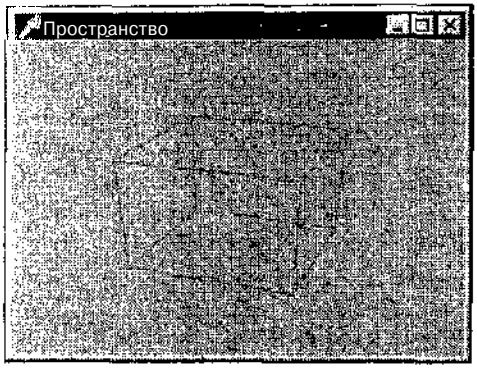


Рис. 3.2. Для ориентировки в пространстве будем рисовать каркасную модель куба

Чтобы выводить только ребра куба, после установления контекста воспроизведения задаем нужный режим воспроизведения полигонов:

```
gl.PolygonMode (GL_FRONT_AND_BACK, GL_LINE);
```

Полученная картинка иллюстрирует важную вещь: использование команды glFrustum приводит к созданию *перспективной проекции.* Хорошо видно, что ребра куба не параллельны друг другу и имеют точку схода где-то на горизонте.

Чтобы помочь вам лучше разобраться с одной из важнейших команд библиотеки OpenGL, я написал пример, расположенный в подкаталоге Ex06. В этом примере аргументы команды — переменные:

```
glFrustum (vLeft, vRight, vBottom, vTop, vNear, vFar);
```

Нажатием на пробел можно приблизить точку зрения к сцене, если при этом удерживать клавишу <Shift>, точка зрения будет удаляться.

Чтобы приблизить точку зрения, варьируем с помощью первых четырех аргументов параметры перспективы, уменьшая объем сцены, для удаления и пространстве проводим обратную операцию.

Затем заново обращаемся к обработчику Resize формы, чтобы установить параметры перспективы в обновленные значения и перерисовать окно:

```
If Key = VK_SPACE then begin;
    If (ssShift in Shift) then begin // нажат Shift, удаляемся
      vLeft := vLeft - 0.1;
```

```
        vRight := vRight + 0.1;
        vBottom := vBottom - 0.1;
        vTop := vTop + 0.1;
        end
        else begin  // приближаемся
        vLeft := vLeft + 0.1;
        vRight := vRight - 0.1;
        vBottom := vBottom - 0.1;
        vTop := vTop - 0.1;
    end;
 FormResize(nil);
 end;
```

Аналогично клавишами управления курсором можно манипулировать значениями каждой из этих четырех переменных в отдельности, а клавиши <Insert> и <Delete> отвечают за координаты ближней и дальней плоскостей отсечения.

Так, если чересчур близко к наблюдателю расположить залнюю плоскость отсечения, дальние ребра куба станут обрезаться (рис. 3.3).

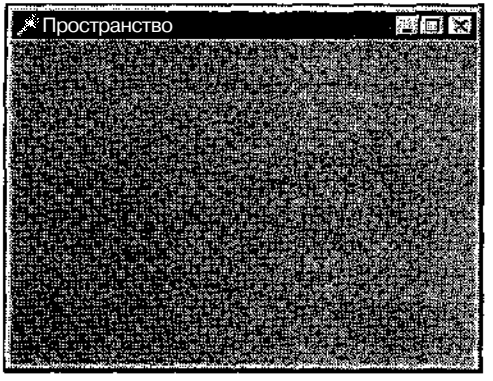


Рис. 3.3. Вот что происходит с кубом, если заднюю плоскость приближать слишком близко

Еще один способ приблизить к сцене точку зрения — уменьшать значение координаты передней плоскости отсечения. Возможно, это более практичный способ, поскольку у первого способа есть один недостаток: если приближаться к сцене чересчур близко, картинка переворачивается и при дальнейшем "приближении" удаляется, оставаясь перевернутой.

Замечание

Это в принципе два разных способа навигации в пространстве, и, если быть точным, оба они не перемещают точку зрения. Можно для этих нужд не изменять параметры вида, а действительно смещать в пространстве систему координат.

Теперь мы перейдем к другой проекции — *ортографической*, параллельной. Посмотрим проект из подкаталога Ex07, результат работы которого показан на рис. 3.4.

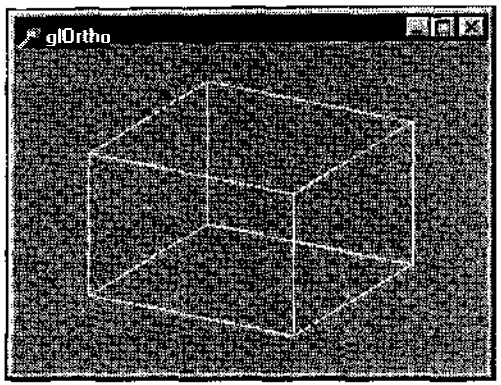


Рис. 3.4. Великий Леонардо нашел бы эту картинку нелепой, но чертежник должен быть довольным

Отличается эта проекция от перспективной именно отсутствием перспективы. В обработчике изменения размеров окна видовые параметры задаются с помощью команды glOrtho:

```
procedure TfrmGL.FormResize(Sender: TObject);
begin
 glViewport(0, 0, ClientWidth, ClientHeight);
 glLoadIdentity;
 glOrtho (-2, 2, -2, 2, 0, 15.0);   // видовые параметры
 glTranslatef (0.0, 0.0, -10.0);    // перенос системы координат по оси Z
 giRotatef (30.0, 1.0, 0.0, 0.0);   // поворот системы координат по оси X
 giRotatef (60.0, CO, 1.0, 0.0);    // поворот системы координат го оси Y

 InvalidateRect (Handle, nil, False;;
end;
```

Аргументы КОМаНДЫ glOrtho **ИМеЮТ ТОЧНО ТаКОЙ же СМЫСЛ, ЧТО И у** glFrustum, но последние два аргумента могут иметь отрицательное значение.

Помимо этих двух команд, OpenGL предоставляет еще несколько возможностей установки видовых параметров, например, библиотека glu содержит команду gluOrtho2D. Эта команда имеет четыре аргумента, смысл которых такой же, как и у glOrtho. По своему действию она эквивалентна вызову giortho с указанием значения расстояния до ближней плоскости отсечения равным минус единице, и расстоянием до дальней плоскости отсечения равным единице..

Как при такой проекции выглядит куб из предыдущих примеров, показано на рис. 3.5, а проект находится в подкаталоге Ex08.

Рис. 3.5. Командой `gluOrtho2D` следует пользоваться с осторожностью, но с точки зрения скорости воспроизведения она является оптимальной

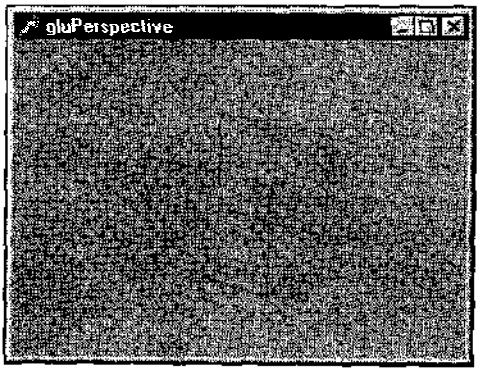


Рис. 3.6. Так работает команда `gluPerspective`

Обратите внимание, что здесь отсутствует начальный сдвиг по оси Z:

```
procedure TfrmGL.FormResize(Sender: TObject);
begin
 glViewport (0, 0, ClientWidth, ClientHeight);
 glLoadIdentity;
 gluOrtho2D (-2, 2, -2, 2);          // задаем перспективу
 glRotatef (30.0, 1.0, 0.0, 0.0);  // поворот объекта -- ось X
 glRotatef (60.0, 0.0, 1.0, 0.0);  // поворот объекта - ось Y

 InvalidateRect(Handle, nil, False);
end;
```

Куб рисуется вокруг глаза наблюдателя и проецируется на плоскость экрана. Согласно установкам этой команды передняя и задняя части нашего куба частично обрезаются.

Следующая команда, которую мы рассмотрим, пожалуй, наиболее популярна в плане использования для первоначального задания видовых параметров. Команда `gluPerspective`, как ясно из ее названия, также находится в библиотеке glu. Проект примера содержится в подкаталоге Ex09, а получающаяся в результате работы программы картинка показана на рис. 3.6.

Смысл аргументов команды поясняется в комментариях:

```
procedure TfrmGL.FormResize(Sender: TObject);
begin
 glViewport(0, 0, ClientWidth, ClientHeight);
 glLoadIdentity;
```

```
// Задаем перспективу
gluPerspective ; 30 . 0, // угол видимости в направлении оси Y
   ClientWidth / ClientHeight, // угол  видимости в направлении оси X
    1.0,   // расстояние от наблюдателя до ближней плоскости отсечения
    15.0); // расстояние от наблюдателя до дальней плоскости отсечения
glTranslatef (0.0, 0.0, -10.0);   // перенос - ось Z
glRotatef (30,0, 1.0, 0.0, 0.0) ;  // поворот — ось X
glRotatef (60.0, 0.C,  1.0,  0.0);   // поворот — ось Y

 InvalidateRect (Handle, nil, False) ;
end;
```

Замечание

В главе 4 мы узнаем, как соотносятся аргументы команд `gluPerspective` и `glFrustum`.

С перспективами, конечно, надо попрактиковаться. Настоятельно рекомендую разобраться с примером из подкаталога ExЮ. Клавиши управления курсором позволяют манипулировать значениями первых двух аргументов команды `gluPerspective`. При уменьшении первого аргумента происходит приближение глаза наблюдателя к сцене, уменьшение второго аргумента приводит к тому, что сцена растягивается в поперечном направлении (рис. 3.7).

Замечание

Как правило, в качестве значения второго аргумента команды `gluPerspective`, так называемого аспекта, задают отношение ширины и высоты области вывода.

В проекте из подкаталога Ex11 я объединил все примеры на изученные команды установки видовых параметров (рис. 3.8). Вы имеете возможность еще раз уяснить различия между этими способами подготовки сцены.

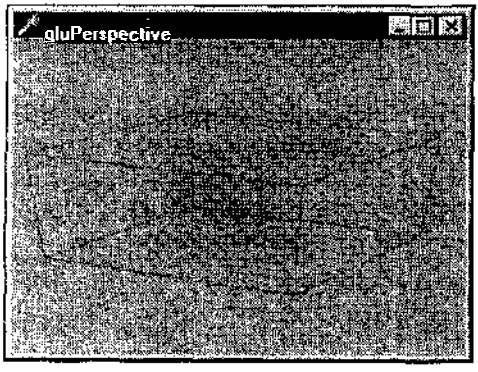


Рис. 3.7. Изменения в видовых установках приводят к трансформации объектов на экране

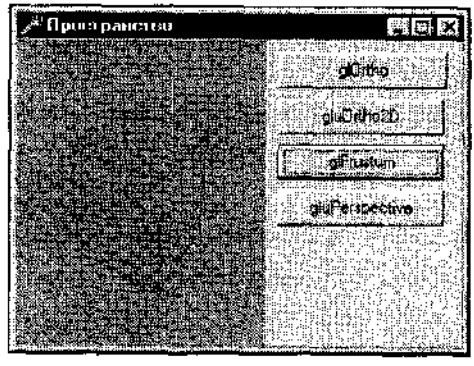


Рис. 3.8. Команды задания видовых параметров

Библиотека glu располагает еще одной вспомогательной командой, имеющей отношение к рассматриваемой теме — gluLookAt. У нее девять аргументов: координаты позиции глаза наблюдателя в пространстве, координаты точки, располагающейся в центре экрана, и направление вектора, задающего попорот сцены (вектор "up").

При использовании этой команды можно обойтись без начальных операций со сдвигом и поворотом системы координат. Ее работу демонстрирует проект из подкаталога Ex12 (рис. 3.9).

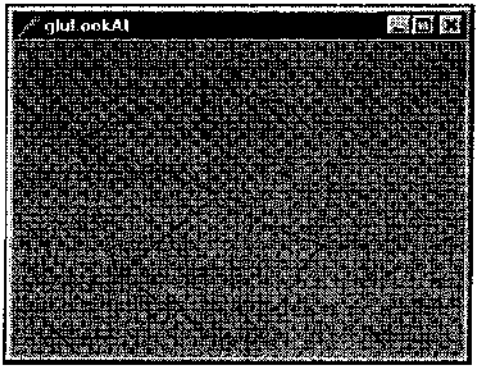


Рис. 3.9. Командой gluLookAt: удобно пользоваться при перемещениях точки зрения в пространстве

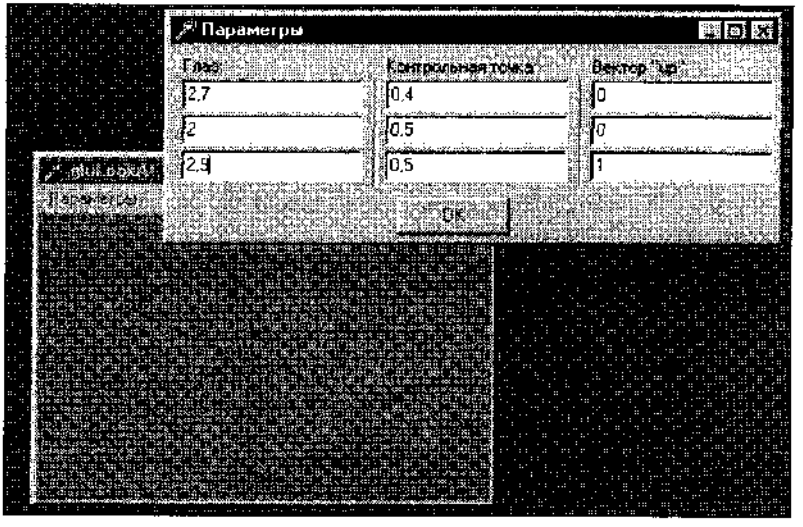


Рис. 3.10. Команду gluLookAt нужно изучить основательно

При задании параметров вида ограничиваемся минимумом команд:

```
glLoadIdentity;
gluPerspective (50.0, ClientWidth / ClientHeight, 2.0, 10.0);
```

```
gluLookAt (2.7, 2, 2.5, 0.4, 0.5, 0.5, 0, 0, 1);
InvalidateRect(Handle, nil, False);
```

На рис. 3.10 показаны экранные формы, выдаваемые программой из подкаталога Ex13.

В них можно менять значения всех аргументов команды gluLookAt, и поскольку этих аргументов сравнительно много, я создал вспомогательную форму, в полях редактирования которой выводятся и задаются пользователем координаты всех девяти аргументов. Рекомендую обязательно попрактиковаться с этим примером, чтобы "почувствовать" работу параметров. Обратите внимание, что, как подчеркивается в документации, вектор "up" не должен быть параллельным линии, соединяющей точку зрения и контрольную точку.

Матрицы OpenGL

При обсуждении команд glRotatef и giTranslatef мы уже обращали внимание на то, что их действие объясняется с позиций линейной алгебры. Например, про команду giTranslatef в соответствующем файле справки говорится, что эта команда умножает текущую матрицу на матрицу переноса. Аналогично glRotatef сводится к операции перемножения текущей матрицы на матрицу поворота. Многие изученные нами команды, действие которых я объяснял по их функциональному смыслу, в действительности описываются с помощью базовых понятий раздела математики, изучающего вопросы, связанные с геометрическими операциями, например, такими, как проекция.

Надеюсь, отсутствие ссылок на линейную алгебру нисколько не помешало вам при изучении предыдущего материала. Если я говорю, что команда glScale является командой масштабирования (и умалчиваю о том. что она сводится к операции перемножения матриц), то рассчитываю на то, что читателю данной информации будет вполне достаточно для успешного использования этой команды на практике.

Я сознательно не включил в книгу изложение математических основ для всех подобных команд, поскольку замыслил ее прежде всего как практическое руководство по компьютерной графике. Мне кажется, что многие читатели не очень хотели бы особо углубляться в дебри формул. Это отнюдь не значит, что я пытаюсь принизить значение математической грамотности. Просто мой опыт преподавания подсказывает, что очень многие программисты предпочитают руководства, делающие упор на практике.

К счастью, существует масса литературы по компьютерной графике, уделяющей много внимания математике, где подробно рассказывается об операциях перемножения матриц и прочих подобных вопросах. Если вы все-таки испытаете необходимость в более основательной теоретической подготовке, найти подходящее пособие не составит большого труда.

Однако и в нашей практической книжке без представления о некоторых специальных терминах не обойтись. Ограничимся самым необходимым.

В OpenGL имеется несколько важных матриц. *Матрица модели* ("modelview matrix") связана с координатами объектов. Она используется для того, чтобы в пространстве построить картину как бы видимую глазу наблюдателя. Другая матрица, *матрица проекций* ("projection matrix"), связана с построением проекций пространственных объектов на плоскость.

Матрица проекций, имея координаты точки зрения, строит усеченные ("clip") координаты, по которым после операций, связанных с перспективой, вычисляются нормализованные координаты в системе координат устройства ("normalized device coordinates"). После трансформаций, связанных с областью вывода, получаются оконные координаты.

Координаты вершин в пространстве и на плоскости проекций четырехмерные, помимо обычных координат есть еще w-координата. Поэтому матрица модели и матрица проекций имеют размерность 4x4.

Перенос и поворот системы координат сводится к операции перемножения матриц, связанных с текущей системой координат, и матриц переноса и поворота. Библиотека OpenGL располагает набором команд, связанных с этими операциями, а также имеет универсальную команду для перемножения матриц: `glMultMatrix`. При желании и наличии математической подготовки этой командой можно заменить команды переноса, поворота и ряд других. Посмотрим, как это делается.

В проекте из подкаталога Ex14 команда начального переноса заменена командой `glMultMatrix`. Для этого введен массив 4x4, хранящий матрицу переноса:

```
mt : Array [0..3, 0..3] of GLfloat;
```

Матрица переноса заполняется нулями, кроме элементов главной диагонали, которые должны быть единицами, и последней строки, содержащей вектор переноса. В примере перемещение осуществляется только по оси 2:

```
mt [0, 0] := 1;
mt [1, 1] := 1;
mt [2, 2] := 1;
mt [3, 3] := 1;
mt [3, 2] := -8;
```

Стартовый сдвиг теперь осуществляется так:

```
glMultMatrixf (@mt) ;
```

Если надо вернуться к исходной позиции, текущая матрица заменяется матрицей с единицами по диагонали и равными нулю всеми остальными элементами, *единичной матрицей*. Это и есть действие команды giLcadidentity. Она является частным случаем более универсальной команды `glLoadMatrix`,

предназначенной для замены текущей матрицы на заданную, ссылка на которую передается в качестве аргумента.

Без примера эти команды не усвоить, поэтому загляните в подкаталог Ex15.

Массив mt аналогичен единичной матрице. Команда glLoadIdentity отсутствует, вместо ее вызова используется явная загрузка единичной матрицы:

```
glLoadMatrixf (@mt);
```

После знакомства с матричными операциями становится яснее технический смысл команд glPushMatrix и glPopMatrix, запоминающих в стеке текущую матрицу и извлекающих ее оттуда. Такая последовательность манипуляций выполняется быстрее, чем вызов glLoadMatrix. То же самое можно сказать и по поводу glLoadIdentity, т. е. единичная матрица загружается быстрее командой glLoadIdentity, чем glLoadMatrix.

Замечание

Чем реже вы пользуетесь командами манипулирования матрицами, тем быстрее будет работать программа. Но учтите, что если на сцене присутствует не больше пары десятков объектов, все манипуляции с матрицами съедают один кадр из тысячи.

Узнать, какая сейчас установлена матрица, можно с помощью команды glGet. Во всех наших предыдущих примерах мы задавали параметры вида применительно к матрице, установленной по умолчанию. Определим, какая *это* матрица.

Подкаталог Ex16 содержит проект, который нам в этом поможет. Обработчик Resize формы дополнен выводом в заголовке окна имени текущей матрицы:

```
var
 wrk : GLUInt;
begin
 glGetIntegerv (GL_MATRIX_MODE, @wrk);
 case wrk of
   GL_MODELVIEW : Caption := 'GL_MODELVIEW';
   GL_PROJECTION : Caption := 'GL_PROJECTION';
 end;
```

Запустив проект, выясняем, что по умолчанию в OpenGL установлена матрица модели.

Итак, во всех предыдущих примерах операции производились над матрицей модели.

В любой момент мы можем выяснить значение этой матрицы. Проиллюстрируем примером — проектом из подкаталога Ex17. При создании формы массив 4x4 заполняется матрицей модели:

```
glGetFloatv (GL_MODELVIEW_MATRIX, @mt);
```

При выборе пункта меню появляется вспомогательное окно, в котором выводится текущая матрица модели (рис. 3.11).

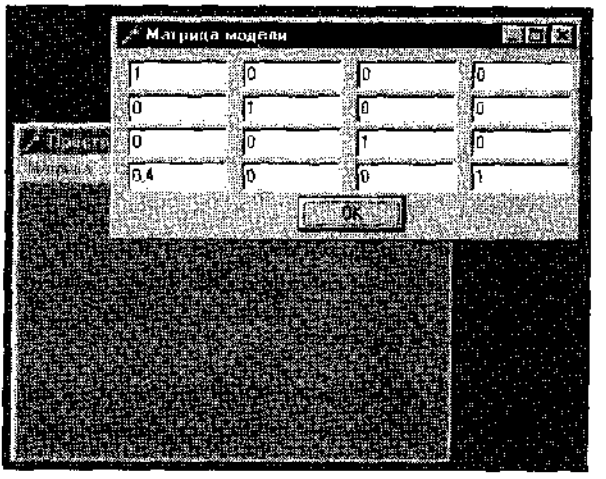


Рис. 3.11. Вывод содержимого матрицы модели

При первом появлении окна эта матрица совпадает с единичной матрицей: по главной диагонали единицы, все остальные элементы равны нулю. Команда `glLoadIdentity` в этой программе также заменена явной загрузкой матрицы.

В проекте имеется возможность варьировать загружаемую матрицу, по нажатию на кнопку "OK" она устанавливается в модели.

Сейчас у вас имеется прекрасная возможность попрактиковаться в матричных операциях. Например, числа на главной диагонали являются масштабными множителями по осям. Поменяйте первые из этих двух чисел произвольно для масштабирования по осям X и Y.

Замечание

Для одного из примеров следующей главы нам важно уяснить, что нулевое значение первого или второго из диагональных элементов приведет к проекции сцены на плоскость.

Если вы внимательно прочитали этот раздел, то полученных знаний о матрицах теперь достаточно для того, чтобы нарисовать все что угодно. Осталось узнать еще одну команду — `glMatrixMode`. Она позволяет установить текущую матрицу.

Проекты всех предыдущих примеров подходят только для простейших пространственных построений, таких как рисование каркаса кубика. Для более сложных программ они не годятся из-за одного упрощения.

Для корректного вывода пространственных фигур параметры вида задаются при установленной матрице проекции, после чего необходимо переключить-

ся в пространство модели. То есть обычная последовательность здесь, например, такая:

```
glViewport(0, 0, ClientWidth, ClientHeight);
glMatrixMode (GL_PROJECTION);
glLoadIdentity;
glFrustum (-1, 1, -1, 1, 3, 10);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity;
```

Замечание

На простейших примерах мы не почувствуем никакой разницы и не поймем, что же изменилось. Позже я попробую показать, с чем связана необходимость выполнения именно такой последовательности действий.

Напоминаю, что параметры вида обычно помещаются в обработчике изменения размеров окна. Стартовые сдвиги и повороты обычно располагаются здесь же, а код воспроизведения сцены заключается между командами `glPushMatrix` и `glPopMatrix`.

Иногда поступают иначе: код воспроизведения начинается с `glLoadIdentity`, а далее идут стартовые трансформации и собственно код сцены.

Замечание

Поскольку программы профессионалов, как правило, основываются на первом подходе, именно его я буду использовать в большинстве примеров этой книги.

Сейчас мы рассмотрим пару примеров на темы предыдущей главы, которые мы не могли рассмотреть раньше из-за того, что в них используются команды задания вида. Начнем с проекта из подкаталога Ex18 — модификации классической программы, поставляемой в составе OpenGL SDK. Программа рисует точку и оси системы координат (рис. 3.12).

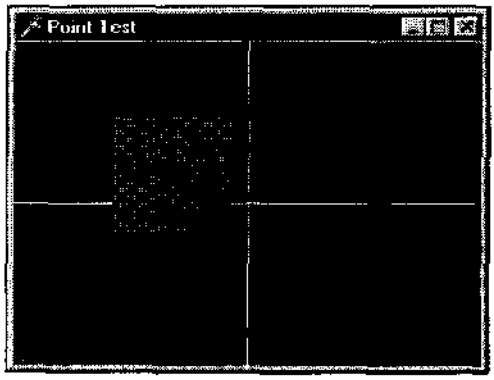


Рис. 3.12. Результат работы проекта Point Test

Клавишами управления курсором можно передвигать точку, нажатием клавиши 'W размер точки можно уменьшать, если же нажимать эту клавишу вместе с <Shift>, то размер точки увеличивается. Первая цифровая клавиша задает режим сглаживания точки. Обратите внимание, что при установленном режиме сглаживания размер получающегося кружочка ограничен некоторыми пределами, для точек больших размеров он не вписывается точно в квадрат точки. Если точнее, то в примере рисуется две точки — вторая рисуется зеленым цветом посередине первой. Вторая точка не изменяется в размерах и всегда выводится несглаженной, но с единичным размером.

Пример можно использовать в качестве простого теста, скорость передвижения большой по размеру точки позволяет оценить возможности компьютера.

Посмотрите внимательно, как в этом примере задаются видовые параметры. Здесь интересно то, что плоскости отсечения привязаны к текущим размерам окна:

```
glViewport(0, 0, ClientWidth, ClientHeight);
glMatrixMode(GL_PROJECTION);
glLoadIdentity;
gluOrtho2D(-ClientWidth/2, ClientWidth/2, -ClientHeight/2, ClientHeight/2);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity;
```

Теперь при изменении размеров окна картинка не масштабируется, а изменяются размеры видимой части пространства. Если точку поставить вблизи границы окна, при уменьшении его размеров точка пропадает, выходит за пределы видимости. В примерах предыдущей главы в такой ситуации точка перемешалась внутрь окна пропорционально его размерам.

Координаты точки хранятся в массиве трех вещественных чисел, вершина воспроизведения задается командой с окончанием V. Аргументом в этом случае является указатель на массив:

```
glBegin(GL_POINTS);
  glVertex3fv (@point);
glEnd;
```

Замечание

Напоминаю, что эта — векторная — форма команд является оптимальной по скоростным показателям.

Следующий пример, проект из подкаталога Ex19, тоже может использоваться в качестве простого теста скоростных характеристик компьютера.

Здесь рисуется серия отрезков прямых, наподобие разметки циферблата часов. В начале и в конце каждого отрезка рисуются точки.

В примере видовые параметры опираются на конкретные числа:

```
glViewport (0, 0, ClientWidth, ClientHeight);
glMatrixMode(GL_PROJECTION);
glLoadIdentity;
gluOrtho2D (-175, 175, -175, 175);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity;
```

Поэтому при изменении размеров окна картинка масштабируется так, что вся сцена всегда вписывается в экран.

Клавиша 'W все так же позволяет регулировать ширину линий, первые две цифровые клавиши отведены для управления двумя режимами воспроизведения: первый режим включает штриховку линий, второй задаст сглаживание отрезков.

Управляя этими тремя параметрами, можно получать разнообразные картинки, например такую, как приведена на рис. 3.13.

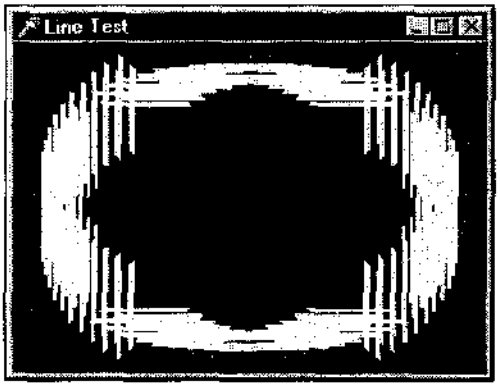


Рис. 3.13. Получить такие узоры на самом деле легко

Поработав с данным примером, вы должны уяснить для себя, что и для сглаженных отрезков есть ограничение — по ширине.

Также полезно разобраться, как в этом примере рисуется система отрезков. Два массива залают координаты начала и конца единственного отрезка, этот отрезок рисуется в цикле, и каждый раз происходит поворот на пять градусов:

```
glLineWidth (size) ; // ширина отрезка
If mode1            // режим, задающий штриховку отрезков
   then glEnable(GL_LINE_STIPPLE) /,' использовать штриховку
   else glDisable(GL_LINE_STIPPLE); // не использовать штриховку
If mode2    // режим, задающий сглаженность отрезков
   then glEnable (GL_LINE_SMOOTH)    /7 сглаживать отрезки
   else glDisable (GL_LINE_SMOOTH);  // не использовать сглаженность
glPushMatrix;                        // запоминаем систему координат
```

```
For i := 0 to 71 do begin   // цикл рисования 72 отрезков
   glRotatef(5.0, 0, 0, 1); // поворот на пять градусов

   glColor3f(1.0, 1.0, 0.0); // цвет отрезков -- желтый
   glBegin(GL_LINE_STRIP);   // примитив - отрезок
     glVertex3fv(@pntA);     // указатель на начало отрезка
     glVertex3fv(@pntB);     // указатель на конец отрезка
   glEnd;

   glColor3f(0.0, 1.0, 0.0); // цвет точек — зеленый
   glBegin(GL_POINTS);       // примитив -- точка
     glVertex3fv(@pntA);     // точка в начале отрезка
     glVertex3fv(@pntB);     // точка в конце отрезка
   glEnd;
end;
glPopMatrix; // возвращаемся в первоначальную систему координат
```

Буфер глубины

При создании контекста воспроизведения в число параметров формата пикселов входят размеры разделов памяти, предоставляемой для нужд OpenGL, или буферов. Помимо буферов кадра, в OpenGL присутствуют еще три буфера: буфер глубины, буфер трафарета и вспомогательный буфер.

Для специальных нужд могут использоваться еще буфер выбора и буфер обратной связи, они подготавливаются пользователем по мере надобности.

Работа с буферами будет нами подробно изучена в соответствующих разделах книги. В этом разделе мы познакомимся с буфером глубины.

Как ясно из его названия, он используется для передачи пространству. При воспроизведении каждого пиксела в этот буфер записывается информация о значении координаты Z пиксела, так называемая оконная Z. Если на пиксел приходится несколько точек, на экран выводится точка с наименьшим значением этой координаты.

При пространственных построениях отказ от использования буфера глубины приводит к неверной передаче пространства. Посмотрим, в чем тут дело.

Для удобства отладки я написал процедуру, строящую оси координат и помечающую оси буквами X, Y и Z:

```
procedure Axes;
var
  Color ; Array [1..4] of GLFloat;
begin
  glPushMatrix;
```

```
  glGetFloatv (GL_CURRENT_COLOR, @Color);

  glScalef (0.5, 0.5, 0.5);

  glColor3f (0, 1, 0);

  glBegin (GL_LINES);
    glVertex3f (0,  0, 0);
    glVertex3f (3,  0, 0);
    glVertex3f (0,  0, 0);
    glVertex3f (0,  3, 0);
    glVertex3f (0,  0, 0);
    glVertex3f (0,  0, 3);
  glEnd;

  // буква X
  glBegin (GL_LINES);
    glVertex3f (3.1, -0.2, 0.5);
    glVertex3f (3.1,  0.2, 0.1);
    glVertex3f (3.1, -0.2, 0.1);
    glVertex3f (3.1,  0.2, 0.5);
  glEnd;

  // буква Y
  glBegin (GL_LINES);
    glVertex3f (0.0, 3.1, 0.0);
    glVertex3f (0.0, 3.1, -0.1);
    glVertex3f (0.0, 3.1, 0.0);
    glVertex3f (0.1, 3.1, 0.1);
    glVertex3f (0.0, 3.1, 0.0);
    glVertex3f (-0.1, 3.1, 0.1);
  glEnd;

  // буква Z
  glBegin (GL_LINES);
    glVertex3f (0.1, -0.1, 3.1);
    glVertex3f (-0.1, -0.1, 3.1);
    glVertex3f (0.1, 0.1, 3.1);
    glVertex3f (-0.1, 0.1, 3.1);
    glVertex3f (-0.1, -0.1, 3.1);
    glVertex3f (0.1, 0.1, 3.1);
  glEnd;

  // Восстанавливаем значение текущего цвета
  glColor3f (Color [1], Color [2], Color [3]);

  glPopMatrix;
end;
```

Обратите внимание, что оси рисуются зеленым цветом, а цветовые установки запоминаются во вспомогательном массиве, по которому они восстанавливаются в конце работы процедуры.

Оси выходят из точки (0. 0, 0) и визуализируются лишь в положительном направлении. Проект находится в подкаталоге Ex20. В качестве основы взят пример `glFrustum`, в который добавлен вызов вышеописанной процедуры и убрано контурное отображение куба. Результат работы программы приведен на рис. 3.14. Видно, что пространство передается некорректно, куб полностью загораживает оси координат, хотя оси должны протыкать грани куба.

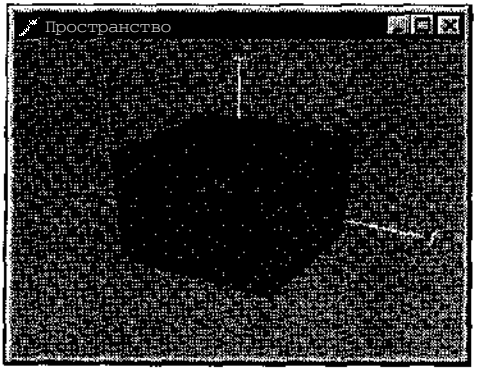


Рис. 3.14. Без использования буфера глубины пространство сцены передается некорректно

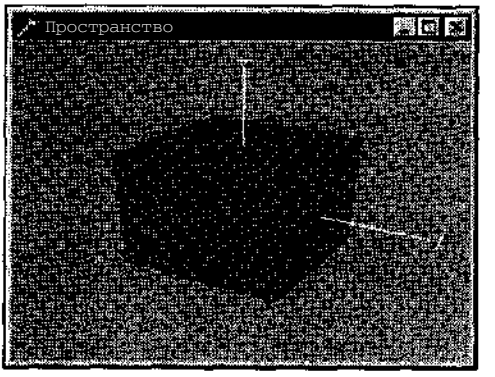


Рис. 3.15. Теперь правильно: оси протыкают грани куба

В следующем примере, проекте из подкаталога Ex21, та же сцена выводится верно (рис. 3.15).

После получения контекста воспроизведения сообщаем системе OpenGL о том, что необходимо корректировать построения в соответствии с глубиной:

```
glEnable (GL_DEPTH_TEST); // включаем режим тестирования глубины
```

Код сцены начинается с очистки двух буферов: буфера кадра и буфера глубины:

```
glClear (GL_COLOR_BUFFER_BIT or GL_DEPTH_BUFFER_BIT); // + буфер глубины
```

Точно так же, как перед очередным построением необходимо очистить поверхность рисования, для корректного воспроизведения требуется очистить буфер пространства. Эти действия, которые мы впервые выполнили в этом примере. будут использоваться в большинстве последующих примеров.

Замечание

О содержимом буфера глубины мы будем говорить еще неоднократно.

С буфером глубины связаны две команды: glDepthFunc и glDepthRange. Хоть они применяются довольно редко, представление о них иметь не помешает.

Первая из этих команд задает правило, по которому происходит сравнение значения оконного Z перед выводом пиксела. По умолчанию установлено значение GL_LESS — выводить на экран точки с минимальным значением оконной Z. Остальные значения приводят чаще всего к тому, что вообще ничего не будет выведено.

Вторая команда задает распределение оконной координаты Z при переводе из нормализованных координат в оконные. На рис. 3.16 приведен результат работы программы (проект из подкаталога Ex22), где такое распределение установлено в обратное принятому по умолчанию:

```
glDepthRange (1, 0);
```

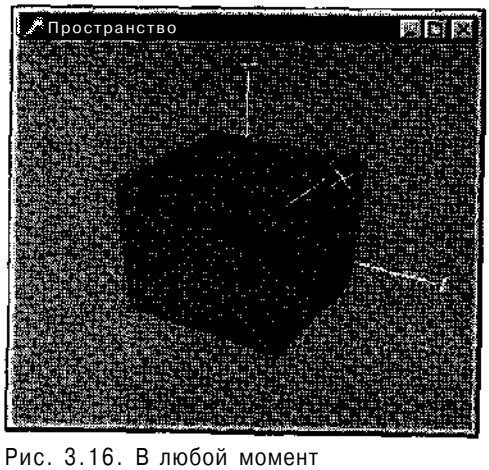


Рис. 3.16. В любой момент можно увидеть то, что скрыто от глаза наблюдателя

Рис. 3.17. На сцене появился источник света

Источниксвета

Предыдущие примеры вряд ли могут удовлетворить кого-либо в силу своей невыразительности. Рисуемый кубик скорее угадывается, все грани покрыты монотонным цветом, за которым теряется пространство. Теперь мы подошли к тому, чтобы увеличить реализм получаемых построений.

Вот в следующем примере кубик рисуется более реалистично — рис. 3.17, проект из подкаталога Ex23.

При создании окна включается источник света:

```
glEnable (GL_LIGHTING); // разрешаем работу с освещенностью
glEnable (GL_LIGHT0);   // включаем источник света
```

Это минимальные действия для включения источника света. Теперь в сцепе присутствует один источник спета с именем 0.

При необходимости можно "установить" несколько источников, для этого точно так же используется команда glEnabie, например:

```
glEnabie (GL_LIGHT1);   // включаем источник света 1
```

Пока нет смысла использовать дополнительные источники света, это никак не повлияет на получаемые картинки, поскольку все добавляемые источники света используют установки, принятые по умолчанию, и ничем не отличаются друг от друга.

При рисовании каждой стороны куба задается вектор нормали, используемый для расчета цветовых параметров каждого пиксела. Для сокращения кода из шести сторон куба я оставил три непосредственно видимые наблюдателю.

```
glBegin(GL_QUADS);
  glNormal3f(0.0, 0.0, 1.0);
  glVertex3f(1.0, 1.0, 1.0);
  glVertex3f(-1.0, 1.0, 1.0);
  glVertex3f(-1.0, -1.0, 1.0);
  glVertex3f(1.0, -1.0, 1.0);
glEnd;

glBegin(GL_QUADS);
  glNormal3f(-1.0, 0.0, 0.0);
  glVertex3f(-1.0, 1.0, 1.0);
  glVertex3f(-1.0, 1.0, -1.0);
  glVertex3f(-1.0, -1.0, -1.0);
  glVertex3f(-1.0, -1.0, 1.0);
glEnd;

glBegin(GL_QUADS);
  glNormal3f(0.0, 1.0, 0.0);
  glVertex3f(-1.0, 1.0, -1.0);
  glVertex3f(-1.0, 1.0, 1.0);
  glVertex3f(1.0, 1.0, 1.0);
  glVertex3f(1.0, 1.0, -1.0);
glEnd;
```

Теперь поговорим о некоторых деталях.

Во-первых, выясним, какое максимальное число источников света мы можем использовать. Проект из подкаталога Ex24 в заголовке окна выводит это число, получаемое при создании окна, с помощью команды `glGet`:

```
glGetIntegerv (GL_MAX_LIGHTS, @wrk) ;
Caption := IntToStr (wrk);
```

Вектора нормалей строятся перпендикулярно каждой стороне куба. В силу того, что наш кубик строится вокруг точки (0, 0, 0), аргументы glNormal3f в данном случае совпадают с точкой пересечения диагоналей каждой грани куба. Чтобы уяснить, насколько важно верно задавать вектор нормали, посмотрите пример, располагающийся в подкаталоге Ex25.

Здесь рисуется только передняя грань кубика, тремя клавишами управления курсором можно менять координаты вектора нормали. При этом меняется вид картинки: освещенность площадки передается в соответствии с текущим значением вектора нормали.

Вектор нормали не обязательно должен исходить именно из середины площадки, достаточно того, чтобы он был параллелен действительному вектору нормали к площадке. Это иллюстрирует проект из подкаталога Ex26, где теми же клавишами можно передвигать площадку в пространстве, а вектор нормали задается единожды при создании окна:

```
glNormal3f(-1.0, 0.0, 0.0);
```

Где бы ни располагалась в пространстве площадка, она освещается единообразно.

Замечание

По умолчанию источник света располагается где-то в бесконечности, поэтому освещенность площадки не меняется вместе с ее перемещением.

В примере из подкаталога Ex27 клавишей <курсор влево> задается поворот площадки в пространстве вокруг оси Y, чтобы можно было взглянуть на площадку с разных точек зрения. Если мы смотрим на заднюю сторону площадки, то видим, что она окрашивается черным цветом. В некоторых ситуациях необходимо, чтобы при таком положении точки зрения наблюдателя примитив не отображался вообще, например, при воспроизведении объектов, образующих замкнутый объем, нет необходимости тратить время на воспроизведение примитивов, заведомо нам не видимых, раз они повернуты к нам задней стороной. Запомните, как это делается, разобрав проект из подкаталога Ex28.

Здесь площадка не рисуется, если повернута к наблюдателю задней стороной. Для этого необходимо включить отсечения задних сторон многоугольников:

```
glEnable (GL_CULL_FACE);
```

Команда glCullFace позволяет задавать, какие стороны при этом подвергаются отсечению, передние или задние. Понятно, что по умолчанию предлагается отсекать задние стороны. Противоположное правило отсечения можно установить так:

```
glCullFace (GL_FRONT);
```

Объемные объекты

Подчеркну еще раз, что для объемных построений используются все те же десять изученных нами во второй главе примитивов.

В следующем примере строится объемный объект на базе тестовой фигуры, знакомой нам по первой главе (рис. 3.18). Проект расположен в подкаталоге Ex29.

Рис. 3.18. Теперь деталь стала объемной

Шестнадцать отдельных многоугольников образуют то, что наблюдателю представляется единым объемным объектом.

В проекте введена переменная, ответственная за режим воспроизведения детали:

```
mode : (POINT, LIKE, FILL) = LINE;
```

В зависимости от значения этой переменной устанавливается режим рисования многоугольников:

```
case mode of
 POINT : glPolygonMode (GL_FRONT_AND_BACK, GL_POINT);
 LINE  : glPolygonMode (GL_FRONT_AND_BACK, GL_LINE) ;
 FILL  : glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);
end;
```

Режим можно менять, нажимая на первые три цифровые клавиши, а клавиши управления курсором позволяют изменять положение точки зрения в пространстве.

В этом примере деталь получается разноцветной. При включенном источнике света для того, чтобы включить учет текущего цвета примитивов, необходимо вызвать команду `glEnable` с соответствующим аргументом:

```
glEnable (GL_COLOR_MATERIAL);
```

Попутно обращу ваше внимание на то, что если объект приближается чересчур близко к глазу наблюдателя и пересекает ближнюю плоскость отсечения, в нем появляется дырка, сквозь которую можно заглянуть внутрь объекта.

Замечание

Необходимо сказать, что при рисовании очень больших объектов происходят сильные искажения, связанные с перспективой, поэтому рекомендуется объекты масштабировать таким образом, чтобы их линейные характеристики лежали в пределах 100.

Надстройки над OpenGL

Существует несколько надстроек над OpenGL, представляющих собой набор готовых команд для упрощения кодирования. Стандартной надстройкой, поставляемой вместе с OpenGL, является библиотека glu, физически располагающаяся в файле glu32.dll. Мы уже изучили несколько команд этой библиотеки, и в дальнейшем продолжим ее изучение. Помимо этой стандартной надстройки наиболее популярной является библиотека glut. Для программистов, пишущих на языке C, эта библиотека особенно привлекательна, поскольку является независимой от операционной системы. Ее применение значительно упрощает кодирование программ, поскольку вся черновая работа по созданию окна, заданию формата пиксела, получению контекстов и пр. выполняется с помощью вызовов библиотечных функций. Вот пример начальной части программы, где с использованием функций библиотеки glut создается окно с заданными параметрами:

```
glutInitDisplayMode(GLUT_DOUBLE j GLUT_RGB | GLUT_DEPTH);
glutInitWindowSize(400, 400);
glutInitWindowPosition(50, 50) ;
glutCreateWindow(argv[0]);
```

Программисты, работающие с Delphi, также могут пользоваться этой библиотекой. В приложении я привожу адрес, по которому можно получить заголовочный файл для подключения библиотеки. Однако при программировании на Delphi мы не получим независимости от операционной системы, а из команд библиотеки чаще всего программистов интересует только набор функций для построения некоторых объемных фигур.

Поэтому вместо нестандартной библиотеки я предпочитаю использовать модуль DGLUT.pas — перенос на Delphi исходных файлов библиотеки glut. Во многих последующих примерах будут встречаться обращения к этому модулю, соответствующие команды начинаются с префикса glut. Например, для рисования куба с единичной длиной ребра вместо двух десятков строк теперь достаточно одной:

```
glutSolidCube (1.0);
```

Помимо куба, модуль (и библиотека) содержит команды воспроизведения сферы, тора, конуса и некоторых правильных многогранников, таких как тетраэдр и додекаэдр. Есть здесь и команда для рисования классического объекта для тестовых программ машинной графики — чайника.

Правильные многогранники строятся как совокупность многоугольников. о том, как создаются остальные объекты этой библиотеки, мы поговорим позднее.

Для получения представления о модуле DGLUT разберите несложный пример, проект из подкаталога Ex3O.

По выбору пользователя можно построить любой объект модуля в одном из режимов: точками, линиями или сплошной поверхностью. Для задания типа объекта и режима воспроизведения введены переменные типа "перечисление":

```
mode : (POINT, LINE, FILL) = FILL;
glutobj : (CUBE, SPHERE, CONE, TORUS, DODECAHEDRON,
           ICOSAHEDRON, TETRAHEDRON, TEAPOT) = CUBE;
```

По умолчанию заданы значения, соответствующие кубу со сплошными гранями. При воспроизведении сцены устанавливается режим и производится обращение к нужной команде:

```
case mode of
 POINT : glPolygonMode (GL_FRONT_AND_BACK, GL_POINT);
 LINE  : glPolygonMode (GL_FRONT_AND_BACK, GL_LINE);
 FILL  : glPolygonMode (GL_FRONT_AND_BACK, GL_FILL);
end;

case glutobj of
  TEAPOT : glutSolidTeapot (1.5);
  CUBE : glutSolidCube (.1.5);
  SPHERE : glutSoiidSphere (1.5, 20, 20);
  CONE : glutSolidCone (0.5, 1.5, 20, 20);
  TORUS : glutSolidTcrus (0.5, 1.5, 20, 20);
  DODECAHEDRON : glutSolidDodecahedron;
  ICOSAHEDRON : glutSolidIcosahedron;
  TETRAHEDRON : glutSolidTetrahedron;
end;
```

Нажимая на первые три цифровые клавиши, можно задавать режим, четвертая клавиша позволяет переключать тип объекта:

```
If Key = 52 then begin
   Inc (glutobj); // установить следующее значение
   If glutobj > High (glutobj) then glutobj := Low (glutobj);
   InvalidateRect(Handle, nil, False);
end;
```

Параметр команды, рисующей чайник, имеет такой же смысл, что и для куба, — размер объекта. Для сферы необходимо указать радиус и количество линий разбиения по широте и по долготе.

Замечание

Чем больше эти числа, тем более гладкими получаются объекты, но тем больше времени требуется для их воспроизведения. Если все предыдущие рекомендации по оптимизации скоростных характеристик дают малоощутимый эффект, то детализация поверхностей является одним из самых важных факторов, влияющих на скорость воспроизведения.

Для конуса задаются радиус основания, высота и пара чисел, задающих гладкость построений.

У тора параметры следующие: внутренний и внешний радиусы и все те же два числа, задающих, насколько плавной будет поверхность рисуемой фигуры.

В этом примере режим задается способом, знакомым нам по предыдущим примерам — командой `glPolygonMode`. Для каркасного изображения объектов модуль располагает серией команд, аналогичных тем, что мы используем в этом примере, но с приставкой glutWire вместо `glutSolid`.

Следующий пример показывает, как можно манипулировать объектами исходного набора, чтобы получить другие объемные фигуры. В подкаталоге Ex31 содержится проект, представляющий модификацию классической программы из SDK. В программе моделируется рука робота в виде двух параллелепипедов (рис. 3.19).

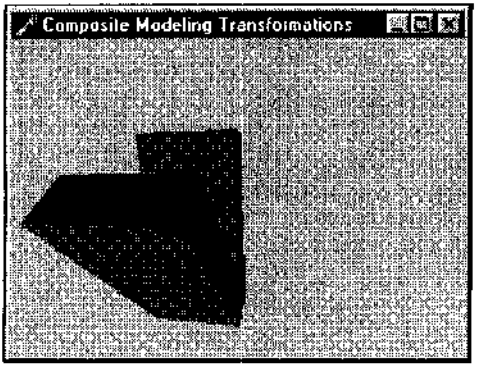


Рис. 3.19. В программе рукой манипулятора можно управлять

Я не стал расписывать каждый параллелепипед по шести граням, а просто задал масштаб по трем осям перед воспроизведением куба так, чтобы вытянуть его в нужную фигуру:

```
glTranslatef (-1.0,  0.0,  0.0);
glRotatef (shoulder,  0.0,  0.0,  1.0);
glTranslatef (1.0,  0.0,  0.0);
```

```
glPushMatrix; // запомнить текущий масштаб
glScaief (2.0, 0.4, 1.0);   // для вытягивания куба в параллелепипед
glutSolidCube(1.0);         // в действительности — параллелепипед
glPopMatrix;                // вернуть обычный масштаб
// вторая часть руки робота
glTranslatef (1.0, 0.0, 0.0);
glRotatef (elbow, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
glPushMatrix;
glScaief (2.0, 0.4, 1.0);
glutSolidCube(1.0);
glPopMatrix;
```

В программе клавишами <Home>, <End>, <Insert> и <Delete> можно задавать относительное положение "суставов" руки, эмулируя управление рукой робота. Первой цифровой клавишей можно менять режим воспроизведения, клавишами управления курсором можно менять положение точки зрения наблюдателя.

Quadric-объекты библиотеки glu

Библиотека glu предоставляет набор команд, без которых, в принципе, можно обойтись, но с их использованием решение многих задач сильно упрощается.

Для упрощения построений некоторых поверхностей второго порядка вводится серия команд, основой которых являются квадратичные (quadric) объекты — собственный тип этой библиотеки.

Как и раньше, познакомимся с новой темой на примерах, для чего откройте проект gluDisk.dpr в подкаталоге Ex32. То, что получается в результате работы программы, изображено на рис. 3.20. Пример этот является переложением на Delphi классической программы из набора примеров OpenGL SDK.

Прежде всего обратите внимание, что видовые параметры задаются в зависимости от пропорций окна, чтобы при любых значениях габаритов окна сцена не выходила за его пределы:

```
If ClientWidth <= ClientHeight
   then glOrtho (0,0, 50.0, 0.0, 50.0 * ClientHeight / ClientWidth,
                                                        -1.0, 1.0)
   else glOrtho (0.0, 50.0 * ClientWidth / ClientHeight, 0.0, 50.0,
                                                        -1.0, 1.0);
```

Для работы с командами библиотеки glu вводится переменная специального типа:

```
quadObj : GLUquadricObj;
```

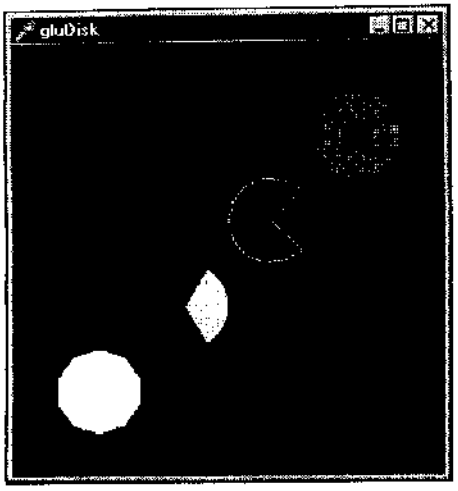


Рис. 3.20. Проект иллюстрирует использование quadric-объектов

При создании окна вызываем команду, создающую quadric-объект, без этого действия обращение к объекту приведет к исключениям:

```
quadObj := gluNewQuadric;
```

Режим воспроизведения объекта задается командой `gluQuadricDrawStyle`, первым аргументом команды указывается имя quadric-объекта. По умолчанию стиль задается сплошным, так что чаще всего нет надобности вызывать эту команду. Она также является аналогом команды `glPolygonMode` и всегда может быть ею заменена, за исключением случая использования с аргументом `GLU_SILHOUETTE`. При установлении такого режима рисуется только граничный контур фигуры, отдельные сектора не рисуются, как это сделано при рисовании третьего диска рассматриваемого примера.

Диск строится с помощью команды `gluDisk`. Помимо имени объекта се аргументами задаются внутренний и внешний радиусы, затем два числа, задающих число разбиений диска по оси Z и число концентрических окружностей при разбиении диска. Смысл этих величин хорошо виден при каркасном режиме воспроизведения, когда в получающейся паутинке хорошо видны отдельные сектора диска.

В этом примере показано, как нарисовать дугу диска, сектор, "кусок пирога". Это делается с помощью команды `gluPartialDisk`, первые пять параметров которой полностью аналогичны параметрам `gluDisk`, а остальные задают начальный угол и угол развертки. Углы задаются в градусах.

По окончании работы память, используемую quadric-объектами, необходимо освобождать. Сделать это нужно до освобождения контекста воспроизведения:

```
gluDeleteQuadric (quadObj); // удаление объекта
wglMakeCurrent (0, 0);
wglDeleteContext fhrc) ;
```

Замечание

Согласно файлу справки, обращение к удаленному объекту невозможно. Может быть, вы столкнетесь здесь с тем же неожиданным явлением, что и я: поставьте вызов этой команды сразу же за его созданием и запустите программу. У меня не возникло никаких исключений, что меня удивило и озадачило. Тем не менее, я все же порекомендую не забывать удалять quadric-объекты.

Библиотека glu имеет средства работы с возможными исключениями, для этого предназначена команда `gluQuadricCallback`. Первый аргумент, как обычно, имя quadric-объекта, вторым аргументом может быть только константа `GLU_ERROR`, третий аргумент — адрес функции, вызываемой при исключении.

В заголовочном файле отсутствует описание константы `GLU_ERROR`, вместо нее можно использовать присутствующую там константу `GLU_TESS_ERROR`, либо самому определить такую константу.

Замечание

Точнее, описание константы не отсутствует, а закомментировано.

Без примера, конечно, не обойтись, им станет проект из подкаталога Ex3O.

В примере описаны процедура, которая будет вызываться в случае ошибки, и нужная константа:

```
procedure FNGLUError;
begin
 ShowMessage {'Ошибка при работе с quadric-объектом!');
end;
const
 GLU_ERROR = GLU_TESS_ERROR;
```

Замечание

Процедура, используемая в таких ситуациях, не может присутствовать в описании класса. При описании подобных процедур рекомендую использовать ключевое слово `stdcall`.

Сразу после создания quadric-объекта задается функция, вызываемая при исключениях; здесь передаем адрес пользовательской процедуры:

```
gluQuadricCallback(quadObj, GLU_ERROR, @FNGLUError) ;
```

Для того чтобы протестировать программу, создается исключение указанием заведомо неверной константы в команде установки режима:

```
gluQuadricDrawStyle (quadObj, GL_LINE};
```

Теперь при запуске программы работает наша тестовая процедура (рис. 3.21).

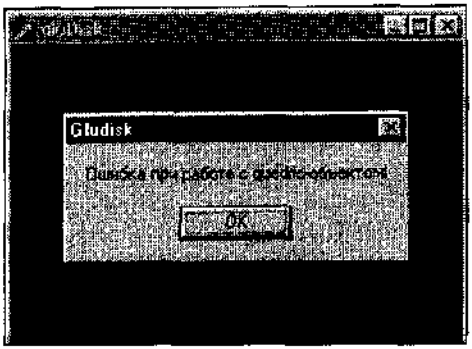


Рис. **3.21.** Пример на использование команды gluQuadricCallback

Замечание

Исключение при этом не снимается, так что после этого сообщения появляется сообщение операционной системы об аварийной ситуации. Можно использовать защищенный режим, чтобы такого сообщения не появлялось,

Если при обращении к команде gluQuadricCallback в качестве адреса функции задать значение, равное nil, возникающие исключения будут сниматься, а выполнение программы будет продолжаться со строки, следующей за строкой, породившей исключение.

Помимо диска, библиотека располагает командами рисования еще некоторых объектов, например, команда gluCylinder предназначена для построения цилиндров и конусов. Параметры команды следующие: имя объекта, радиусы основания и верхней части и два числа, задающих частоту разбиения.

Команда gluSphere, как ясно из ее имени, упрощает построение сферы. У нее четыре аргумента, второй аргумент является радиусом сферы, остальные параметры традиционны для этой серии команд.

Все объекты библиотеки glu я собрал в одном проекте, располагающемся в подкаталоге Ex34. Этот пример аналогичен примеру на объекты библиотеки glut: по выбору пользователя рисуется объект заданного типа.

Здесь также демонстрируется действие команды gluQuadricOrientation, задающей направление нормали к поверхности объекта, внутрь или наружу. Еще можно выяснить, как сказывается на виде объекта работа команды gluQuadricNormals, определяющей, строятся ли нормали для каждой вершины, для всего сегмента либо вообще не строятся.

Нажимая на первые четыре цифровые клавиши, можно задавать значения параметров: режим, тип объекта, ориентацию нормалей и правило их построения.

В проекте введены соответствующие перечисления:

```
mode : (POINT, LINE, FILL, SILHOUETTE) = FILL;
gluobj ; (SPHERE, CONE, CYLINDER, DISK) = SPHERE;
```

```
orientation : (OUTSIDE, INSIDE) = OUTSIDE;
normals : (NONE, FLAT, SMOOTH) = SMOOTH;
```

Первым аргументом всех команд является имя quadric-объекта, все возможные константы перечисляются в конструкциях case:

```
case mode of     // режим воспроизведения
  POINT : gluQuadricDrawStyle (quadObj, GLU_POINT);       // точки
  LINE  : gluQuadricDrawStyle (quadObj, GLU_LINE);        //линии
  FILL  : gluQuadricDrawStyle (quadObj, GLU_FILL);        // сплошным
  SILHOUETTE : gluQuadricDrawStyle (quadObj, GLU_SILHOUETTE); // контур
end;

case orientation of // направление нормалей
  INSIDE : gluQuadricOrientation (quadObj, GLU_INSIDE) ;   // внутрь
  OUTSIDE : gluQuadricOrientation (quadObj, GLU_OUTSIDE); // наружу
end;

case normals of // правило построения нормалей
  NONE : gluQuadricNormals (quadObj, GLU_NONE); // не строить
  FLAT : gluQuadricNormals (quadObj, GLU_FLAT) ; // для сегмента
  SMOOTH : gluQuadricNormals (quadObj, GLU_SMOOTH);// для каждой вершины
end;

case gluobj of// тип объекта
  SPHERE : gluSphere (quadObj, 1.5, 10, 10); // сфера
  CONE : gluCylinder (quadObj, 0.0, 1.0, 1.5, 10, 10);     // конус
  CYLINDER : gluCylinder (quadObj, 1.0, 1.0, 1.5, 10, 10); // цилиндр
  DISK : gluDisk (quadObj, O.O, 1.5, 10, 5);               // диск
end;
```

На рис. 3.22 приводится одна из возможных картинок, получаемых с помощью этого проекта.

Следующие примеры этого раздела также являются переложением на Delphi классических примеров из SDK.

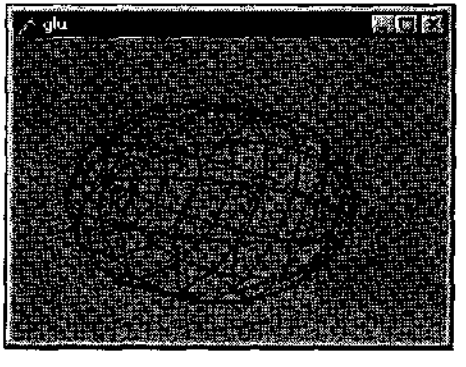


Рис. 3.22. С помощью этого примера вы можете познакомиться со всеми quadric-объектами

Первым делом посмотрите проект из подкаталога Ex35 — вступительный пример к нескольким следующим проектам и одновременно простейший пример на использование объектов библиотеки glu.

Здесь просто рисуется сфера, модель звезды, наблюдатель находится над полюсом (рис. 3,23).

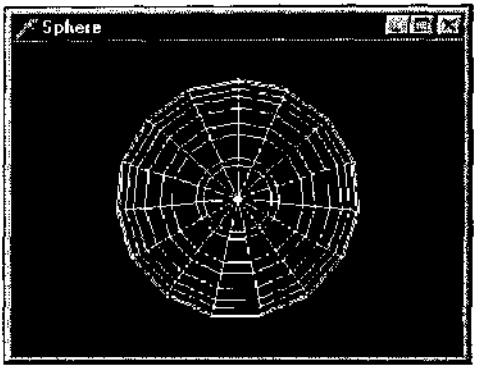


Рис. 3.23. Упрощенная модель звезды

Проект следующего примера располагается в подкаталоге Ex32, а экранная форма приложении приведена на рис. 3.24.

Здесь моделируется уже планетарная система, клавишами управления курсором можно задавать положение планеты относительно звезды и угол полорота ее вокруг собственной оси.

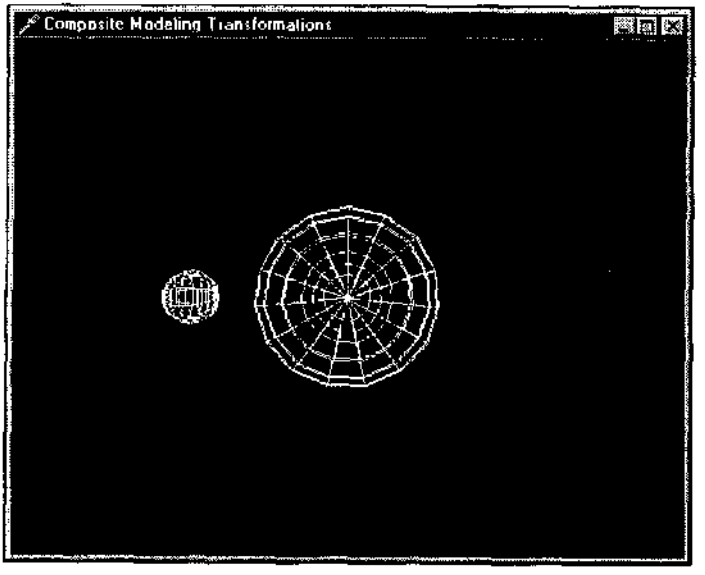


Рис. 3.24. Добавилась планета

Замечание

Начинающим я рекомендую хорошенько разобраться с этим примером, здесь впервые для нас в пространстве присутствует несколько объектов, располагающихся независимо друг от друга.

На базе одного quadric-объекта можно строить сколько угодно фигур; не обязательно для каждой из них создавать собственный объект, если параметры всех их идентичны.

В этом примере на базе одного объекта рисуется две сферы:

```
// рисуем солнце
gluSphere (quadObj, 1.0, 15, 10);
// рисуем маленькую планету
glRotatef (year, 0.0, 1.0, 0.0);
glTranslatef (2.0, 0.0, 0.0);
glRotatef (day, 0.0, 1.0, 0.0);
gluSphere (quadObj, 0.2, 10, 10);
```

Следующий пример, проект из подкаталога Ex37, является небольшой модификацией предыдущего. Положение точки зрения изменилось так, что модель наблюдается под другим углом — рис. 3.25.

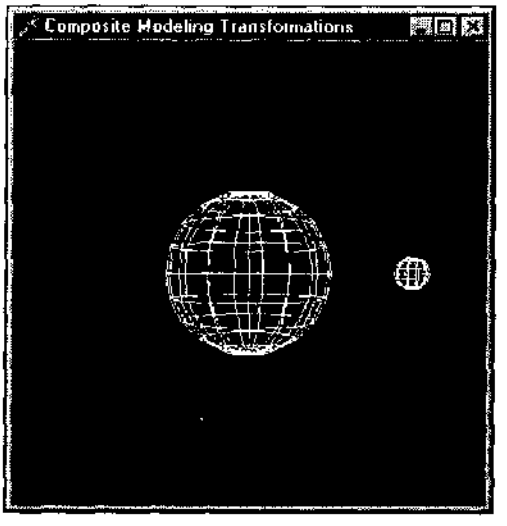


Рис. 3.25. В этом примере смотрим на систему с другой точки зрения

Этот пример служит хорошей иллюстрацией на использование команд glPushMatrix и glPopMatrix: солнце и планета поворачиваются по отдельности относительно базовой системы координат:

```
glPushMatrix;

// рисуем солнце
glPushMatrix;
```

```
glRotatef (90.0, 1.0, 0.0, 0.0);  // поворачиваем прямо
gluSphere (quadObj, 1.0, 15, 10);
glPopMatrix;
// рисуем маленькую планету
glRotatef (year, 0.0, 1.0, 0.0);
glTranslatef (2.0, 0.0, 0.0);
glRotatef (day, 0.0, 1.0, 0.0);
glRotatef (90.0, 1.0, 0.0, 0.0);  // поворачиваем прямо
gluSphere (quadObj, 0.2, 10, 10);
glPopMatrix;
```

Рано или поздно вам потребуется узнать, как в OpenGL можно получить вырезку пространственных фигур, например, полусферу. Следующий пример (подкаталог Ex38) поможет узнать, как это делается. В нем рисуется четверть сферы — рис. 3.26.

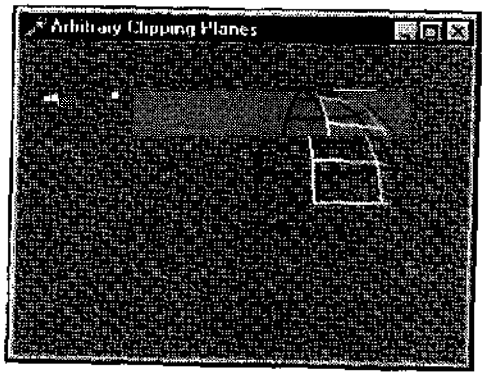


Рис. 3.26. Совсем несложно получить полусферу или четверть сферы

Для вырезки части пространства используется новая для нас команда glClipPlane. Для вырезки можно использовать несколько плоскостей, эта команда идентифицирует используемые плоскости. Первый аргумент — символическое имя плоскости вырезки, второй — адрес массива, задающего эту плоскость. Символические имена начинаются с GL_CLIP_PLANE, дальше следует цифра, нумерация начинается с нуля.

При каждой вырезке отсекается полупространство, массив задает вектор, определяющий остающуюся часть. Вектор не должен быть обязательно перпендикулярен осям, как в нашем примере.

Для получения четверти сферы проделываем две вырезки: сначала обрезаем нижнее полупространство, удаляя все вершины с отрицательным значением координаты Y, затем отсекаем левое полупространство, т. е. удаляются вершины с отрицательным значением координаты X:

```
const
  eqn : Array [0..3] of GLdouble = (0.0, 0.0, 0.0, 0.0);
```

```
eqn2 : Array [0..3] of GLdouble = (1.0, 0.0, 0.0, 0.0);
...
// удаление нижней половины, для y < 0
giClipPlane (GL_CLIP_PLANE0, @eqn);// идентифицируем плоскость отсечения
glEnable (GL_CLIP_PLANE0);        // включаем первую плоскость отсечения
// удаление левой половины, для x < 0
glClipPlane (GL_CLIP_PLANE1, @eqn2);
glEnatle (GL_CLIP_PLANE1);        // включаем вторую плоскость отсечения
```

Если теперь вернуться к библиотеке glut, то можно заметить, что сфера и конус в ней строятся на базе объектов библиотеки glu. Например, процедура для построения каркаса сферы выглядит так:

```
procedure glutWireSphere(
  Radius : GLdouble;
  Slices : GLint;
  Stacks : GLint);
begin ( glutWireSphere }
  if quadObj = nil then
    quadObj := gluNewQuadric;
  gluQuadricDrawStyle(quadObj, GLU_LINE);
  gluQuadricNormals(quadObj, GLU_SMOOTH);
  gluSphere(quadObj, Radius, Slices, Stacks);
end; { glutWireSphere }
```

Здесь можно подсмотреть несложный прием для определения того, надо ли создавать quadric-объект. Прием основан на том, что тип GLUquadricObj является указателем и его nil-значение соответствует тому, что объект пока еще не создан. Кстати, можете сейчас заглянуть в заголовочный файл opengl.pas, чтобы убедиться, что этот тип является указателем, указателем на пустую запись:

```
type
  _GLUquadricObj = record end;
  GLUquadricObj = ^_GLUquadricObj;
```

Заключительным примером раздела станет проект из подкаталога Ex39 — модель автомобиля (рис. 3.27).

Клавишами управления курсором можно вращать модель по осям, клавиши <Insert> и <Delete> служат для приближения/удаления модели. Пробелом и клавишей ввода можно задавать различные режимы воспроизведения.

В примере используются объекты библиотеки glu и модуля DGLUT. Нет смысла его подробно разбирать, остановимся только на некоторых моментах.

Напоминаю, что для воспроизведения объектов с различным пространственным положением перед собственно воспроизведением каждого из них система координат перемещается в нужную точку в пространстве. Для удобства ориентирования в пространстве перед каждой трансформацией системы

координат текущую матрицу (систему координат) запоминаем вызовом команды `glPushMatrix`, затем возвращаемся в эту систему координат вызовом `glPopMatrix`. Это будет выполняться быстрее, чем обратные переносы. но было бы лучше вычислять координаты следующего объекта в текущей системе координат.

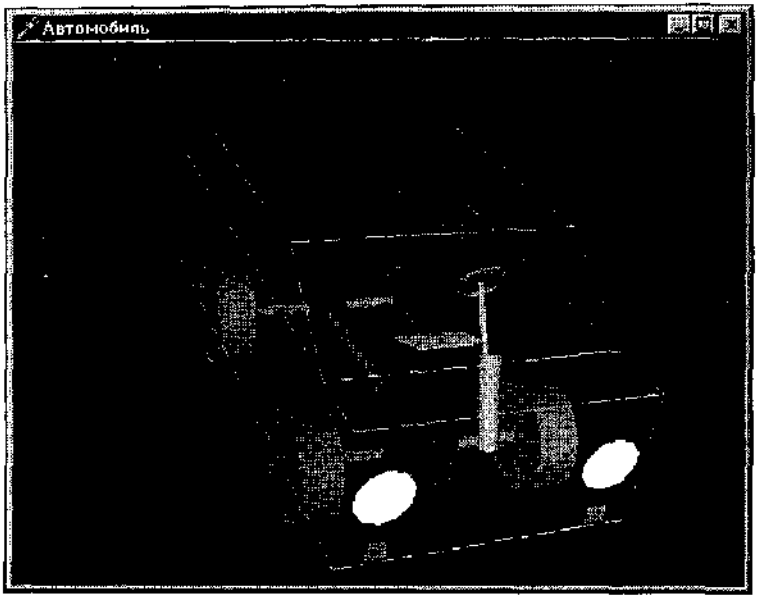


Рис. 3.27. Мы можем строить модели и посложнее двух сфер

Для разнообразия передние квадратные фары автомобиля строятся с использованием команды `gluDisk`: если последние два аргумента команды задать равными четырем, строится ромб, перед построением система координат поворачивается:

```
glRotatef(45, 0.0, 0.0, 1.0);
gluDisk (quadObj, 0.0, 0.1, 4, 4);
```

Квадраты задних фар строятся с помощью команды `glRect`, мы изучали эту команду в предыдущей главе.

Для того чтобы нормаль квадрата была направлена в нужную сторону, переворачиваем систему координат:

```
glRotatef(180, 1.0, 0.0, 0.0);
glRectf (0.1, 0.1, -0.1, -0.1);
```

Вместо этого можно было бы просто задать нужный вектор нормали:

```
glNormal3f (0, 0,-1);
```

Эту программу вы можете взять в качестве шаблона, чтобы поупражняться в построениях.

Проектировать системы из множества объектов без использования редакторов может оказаться трудным делом для новичков. Могу посоветовать воспользоваться приведенной выше процедурой построения осей текущей системы координат: обращайтесь к ней каждый раз, когда необходимо выяснить, "где я сейчас нахожусь".

Сплайны и поверхности Безье

Мы теперь можем без труда рисовать кубики, сферы, цилиндры и многие другие аналогичные фигуры, но, конечно, этого недостаточно, и рано или поздно возникнет потребность нарисовать поверхность произвольной формы. В этом разделе мы узнаем, как это сделать.

Подход, предлагаемый библиотекой OpenGL для изображения криволинейных поверхностей, традиционен для компьютерной графики: задаются координаты небольшого числа опорных точек, определяющих вид искомой поверхности. В зависимости от способа расчета опорные точки могут лежать на получаемой поверхности, а могут и не располагаться на ней.

Сглаживающие поверхности называются *сплайнами*. Есть много способов построения сплайнов, из наиболее распространенных нас будут интересовать только два: кривые Безье (Bezier curves, в некоторых книгах называются "сплайны Бежье") и B-сплайны (base-splines, базовые сплайны).

Изучение этой темы начнем с простейшего, с двумерных кривых.

Операционная система располагает функциями GDI, позволяющими строить кривые Безье.

В примере — проекте из подкаталога Ex40 — модуль OpenGL не используется, это пример как раз на использование функций GDI. Он является модификацией одного из проектов первой главы, но теперь рисуется не квадрат и круг, а кривая Безье по четырем точкам — рис. 3.28.

Рис. 3.28. Функции GDI позволяют строить кривые Безье

Опорные вершины заданы в массиве четырех величин типа `TPoint`, этот тип вводится в модуле windows.pas:

```
Const
  Points : Array [0..3] of TPoint =
    ((x:5; y:5), (x:20; y:70), (x:80; y:15), (x:100;y:90));
```

Собственно рисование кривой состоит в вызове функции GDI PolyBezier, первый аргумент которой — ссылка на контекст устройства, затем указывается массив опорных точек, последний аргумент — количество используемых точек:

```
PolyBezier (dc, Points, 4) ;
```

Построение для большей наглядности дополнено циклом, в котором окружностями визуализируются опорные точки:

```
For i := G to 3 do
       Ellipse (dc, Points [i].x - 3, Points [i].y - 3,
                    Points [i] .x + 3, Points [i] .y + 3);
```

В случае четырех опорных точек кривая всегда будет начинаться точно в первой точке и приходить в последнюю точку. Если их переставить местами, вид кривой не изменится, что не произойдет и если переставить местами вторую и третью опорные точки. Опорные точки могут располагаться в пространстве произвольно, т. е. не требуется, чтобы они, например, равномерно располагались по интервалу. Если необходимо продолжить кривую, добавляется по три точки для каждого последующего сегмента.

Теперь посмотрим, как нарисовать кривую Безье, используя команды OpenGL. Откройте проект из подкаталога Ex41. Получаемая кривая показана на рис. 3.29.

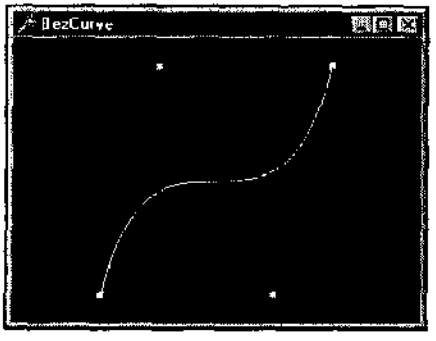


Рис. 3.29. Кривая Безье, построенная с помощью функций библиотеки OpenGL

В обработчике создания формы задаются параметры так называемого *одномерного вычислителя,* и включается этот самый вычислитель:

```
glMap1f (GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, @ctrlpoints);
glEnable (GL_MAP1_VERTEX_3);
```

Первый параметр команды glMap1 — символическая константа, значение GL_MAP1_VERTEX_3 соответствует случаю, когда каждая контрольная точка

представляет собой набор трех вещественных чисел одинарной точности, т. е. координаты точки. Значения второго и третьего аргументов команды определяют конечные точки интервала предварительного образа рассчитываемой кривой. Величины ноль и один для них являются обычно используемыми, подробнее мы рассмотрим эти аргументы чуть ниже.

Четвертый параметр команды, "большой шаг", задает, сколько чисел содержится в считываемой порции данных. Как говорится в документации, контрольные точки могут содержаться в произвольных структурах данных, лишь бы их значения располагались в памяти друг за другом.

Последние два параметра команды — число опорных точек и указатель па массив опорных точек.

Для построения кривой можно использовать точки или отрезки: вместо команды, задающей вершину, вызывается команда glEval coord, возвращающая координаты рассчитанной кривой:

```
glBegin(GL_LINE_STRIP);
  For i := 0 to 30 do
    glEvalCoord1f (i / 30.0);
glEnd;
```

Аргумент команды — значение координаты и. В данном примере соединяются отрезками тридцать точек, равномерно расположенных на кривой.

Теперь выясним правила отображения интервала. Если в этом примере третий параметр команды glMap1f задать равным двум, то на экране получим половину первоначальной кривой. Для получения полной кривой надо при воспроизведении взять интервал в два раза больший:

```
glBegin(GL_LINE_STRIP);
  For i := 0 to 60 do
    glEvalCoord1f(i / 30.0);
glEnd;
```

Если же этот параметр задать равным 0.5, то при отображении интервала с u в пределах от нуля до единицы получаемая кривая не будет останавливаться на последней опорной точке, а экстраполироваться дальше. Если в этом нет необходимости, конечное значение параметра цикла надо взять равным 15.

Чтобы действительно разобраться в подобных вопросах, надо обязательно попрактиковаться, посмотреть, какие кривые строятся для различных наборов опорных точек. Здесь вам поможет проект из подкаталога Ex42, где среди четырех опорных точек имеется одна выделенная. Клавишами управления курсором можно менять положение выделенной точки, при нажатии на пробел выделенной становится следующая точка набора. Выделенная точка рисуется красным.

Обратите внимание, что простой перерисовки окна при изменении в массиве опорных точек недостаточно, необходимо заново обратиться к командам, "заряжающим" вычислитель, чтобы пересчитать кривую:

```
If Key = VK_SPACE then begin
   // выделенной становится следующая точка набора
   selpoint := selpoint + 1;
   If selpoint > High (selpoint) then selpoint := Low (selpoint);
   InvalidateRect(Handle, nil, False);
end;
If Key = VK_LEFT then begin
   // сдвигаем выделенную точку влево
   ctrlpoints [selpoint, 0] := ctrlpoints [selpoint, 0] - 0.1;
   // пересчитываем кривую по измененному массиву опорных точек
   glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, @ctrlpoints);
   glEnable(GL_MAP1_VERTEX_3);
   InvalidateRect(Handle, nil, False);     // перерисовка окна
end;
```

С помощью этого примитивного редактора можно построить замысловатые фигуры и заодно получить представление о кривых Безье.

Замечание

Не получится расположить все четыре точки на кривой, если только это не линейная функция.

С помощью команды `glGetMapfv` в любой момент можно получить полную информацию о текущих параметрах вычислителя. Не думаю, что вы часто будете обращаться к этой команде, но на всякий случай приведу пример на ее использование (проект из подкаталога Ex43). Клавишами <Insert> и <Delete> можно менять текущее значение параметра вычислителя u2, в заголовке окна выводятся значения u1 и u2. Эти значения приложение получает от OpenGL:

```
 wrk : Array [0..1] of GLfloat;
begin
 giGetMapfv (GL_MAP1_VERTEX_3, GL_DOMAIN, @wrk) ;
 Caption := FloatToStr (wrk[0]) + ' , ' + FloatToStr(wrk[1]);
```

Из файла справки вы можете узнать, как получить значения всех остальных параметров вычислителя.

Построить кривую можно и другим способом. Посмотрим пример из подкаталога Ex44, отличающийся от предыдущего примера на кривые Безье следующим: сразу после включения вычислителя вызывается команда, строящая одномерную сетку, т. е. рассчитывающая координаты набора точек на интервале:

```
glMapGrid1f (30, 0, 1);
```

Первый аргумент — количество подинтервалов, далее задается интервал по координате и. После того как сетка построена, вывод ее осуществляется одной командой:

```
glEvalMesh1 (GL_LINE, 0, 30) ;
```

Первый аргумент— режим воспроизведения, отрезками или точками, остальные аргументы задают номера первой и последней точек рассчитанной сетки.

Получив представление о кривых, мы можем перейти к поверхностям Безье. Соответствующий пример располагается в подкаталоге Ex45, а результат работы программы показан на рис. 3.30.

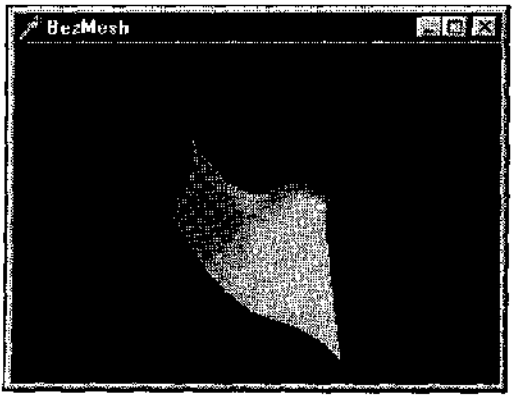


Рис. 3.30. Классический пример на построение поверхности Безье

Массив опорных точек содержит координаты шестнадцати вершин. Работа программы начинается с установки параметров вычислителя:

```
glMap2f (GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, &ctrl_points);
glEnable (GL_MAP2_VERTEX_3);
glMapGrid2f (20, 0.0, 1.0, 20, 0.0, 1.0);
```

У команды `glMap2f` аргументов больше, чем у `glMap1f`, но теперь нам легко понять их смысл. Первый аргумент — константа, определяющая тип рассчитываемых величин, в данном случае это координаты точек поверхности. Последний аргумент — указатель на массив контрольных точек поверхности.

Второй и третий параметры задают преобразования по координате и поверхности. Четвертый аргумент, как и в предыдущем случае, задает, сколько вещественных чисел содержится в порции данных — здесь мы сообщаем, что каждая точка задана тремя координатами. Пятый аргумент — количество точек в строке структуры, хранящей данные.

Следующие четыре аргумента имеют смысл, аналогичный предыдущим четырем, но залают параметры для второй координаты поверхности, координаты v. Значение восьмого аргумента стало равно двенадцати путем пере-

множения количества чисел, задающих координаты одной вершины (3), на количество точек в строке массива (4).

После задания параметров вычислителя он включается вызовом команды glEnable, после чего вызывается одна из пары команд, позволяющих построить поверхность — команда glMapGrid2f, рассчитывающая двумерную сетку. Первый и четвертый аргументы этой команды определяют количество разбиений по каждой из двух координат поверхности, остальные параметры имеют отношение к отображению интервалов.

Собственно изображение поверхности, двумерной сетки, осуществляется вызовом второй команды из тандема:

```
glEvalMesh2 (GL_FILL, O, 2G, 0, 20);
```

Первый аргумент команды задает режим воспроизведения, следующие две пары чисел задают количество подинтервалов разбиения по каждой координате поверхности. В примере мы берем по 20 разбиений, столько же, сколько задано в команде glMapGrid2f, чтобы не выходить за пределы интервалов, но это не обязательно, можно брать и больше.

Если смысл параметров, связанных с отображением интервалов, вам кажется не совсем ясным, рекомендую вернуться к примеру по кривой Безье и еще раз его разобрать.

Замечу, что режим воспроизведения можно варьировать также с помощью команды glPolygonMode.

Обратите внимание, что в рассматриваемом примере используется режим автоматического расчета нормалей к поверхности:

```
glEnable (GL_AUTO_NORMAL);
```

Без этого режима поверхность выглядит невыразительно, но его использование допускается только в случае поверхностей, рассчитываемых вычислителем,

В проекте введены два режима: один управляет тем, как строится поверхность — сплошной или линиями. Второй режим задает, надо ли выводить опорные точки. Нажимая на клавиши ввода и пробела, можно менять текущие значения этих режимов.

При нажатой кнопке мыши при движении курсора поверхность вращается по двум осям, что позволяет хорошо рассмотреть ее с разных позиций. Для осуществления этого режима введен флаг, булевская переменная Down, которая принимает истинное значение при удерживаемой кнопке мыши, и две вспомогательные переменные, связанные с экранными координатами указателя.

В момент нажатия кнопки запоминаются координаты курсора; при движении курсора сцена поворачивается по двум осям на угол, величина которого зависит от разницы текущей и предыдущей координат курсора:

```
procedure TfrmGL.FormMouseMove(Sender: TObject; Shift : TShiftState; X,
  Y: Integer);
begin
  If Down then begin // кнопка мыши нажата
     glRotatef (X - wrkX, 0.0, 1.0, 0.0);// поворот по горизонтали экрана
     glRotatef (Y - wrkY, 1.0, 0.0, 0.0);// поворот по вертикали экрана
     InvalidateRect(Handle, nil, False!; // перерисовать экран
     wrkX := X;       // запоминаем координаты курсора
     wrkY := Y;
  end;
end;
```

Но при изменении размеров окна система координат возвращается в первоначальное положение.

Точно так же, как и в случае с кривой Безье, для воспроизведения поверхности можно воспользоваться командой `glEvalCoord2f`.

На рис. 3.31 приведен результат работы примера из подкаталога Ex46. где на основе все той же поверхности строится множество отрезков, наложение которых на экране приводит к получению разнообразных узоров, называемых узорами муара.

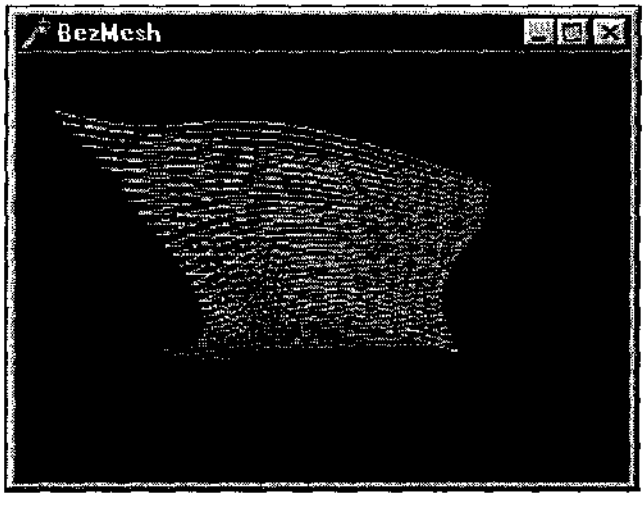


Рис. 3.31. Узор получается наложением отрезков, лежащих на поверхности Безье

Отрезки строятся по вершинам, предоставленным вычислителем:

```
glBegin(GL_LINE_STRIP);
 For i := 0 to 30 do
   For j := 0 to 30 do
      glEvalCoord2f (i / 30, j / 30);
glEnd;
```

Мы уже изучили множество параметров команды glEnable, задающей режимы воспроизведения и, в частности, позволяющей использовать вычислители. Я должен обязательно привести пример на команду, позволяющую определить, включен сейчас какой-либо режим или вычислитель — команду glIsEnabled. Ее аргумент — константа, символ определяемого режима, а результат работы, согласно документации, имеет тип GLboolean. Мы знаем о небольшой проблеме Delphi, связанной с этим типом, так что для вас не должно стать откровением то, что обрабатывать возвращаемое значение мы будем как величину булевского типа.

Приведу здесь простой пример, в котором до и после включения режима окрашивания поверхностей выводится сообщение о том, доступен ли этот режим:

```
If glIsEnabled (GL_COLOR_MATERIAL) = TRUE
   then ShowMessage ('COLOR_MATERIAL is enabled')
   else ShowMessage ('COLOR_MATERIAL is disabled');
```

Соответствующий проект располагается в подкаталоге Ex47.

В этом разделе мы рассмотрели, как в OpenGL строятся кривые и поверхности Безье, позже рассмотрим еще несколько примеров на эту тему.

NURBS-поверхности

Один из классов B-сплайнов, рациональные B-сплайны, задаваемые на неравномерной сетке (Non-Uniform Rational B-Spline, NURBS), является стандартным для компьютерной графики способом определения параметрических кривых и поверхностей.

Библиотека glu предоставляет набор команд, позволяющих использовать этот класс поверхностей. Будем знакомиться с соответствующими командами непосредственно на примерах и начнем с проекта из подкаталога Ex48, где строится NURBS-кривая по тем же опорным точкам, что и в первом примере на кривую Безье. Вид получающейся кривой тоже ничем не отличается от кривой, приведенной на рис. 3.29.

Для работы с NURBS-поверхностями в библиотеке glu имеются переменные специального типа, используемые для идентификации объектов:

```
theNurb  : gluNurbsObj;
```

При создании окна объект, как обычно, создается:

```
theNurb := gluNewNurbsRenderer;
```

А в конце работы приложения память, занимаемая объектом, высвобождается:

```
gluDeleteNurbsRenderer (theNurb);
```

Замечание

В отличие от quadric-объектов, удаленные NURBS-объекты действительно более недоступны для использования.

Для манипулирования свойствами таких объектов предусмотрена специальная команда библиотеки, в примере она используется для задания гладкости поверхности и режима воспроизведения. Гладкость кривой или поверхности задается допуском дискретизации: чем меньше это число, тем поверхность получается более гладкой:

```
gluNurbsProperty (theNurb, GLU_SAMPLING_TOLERANCE, 25.0);
```

В отличие от других объектов, NURBS-поверхности рассчитываются каждый раз заново при каждом построении. В этом легко убедиться, например, следующим образом: увеличьте допуск дискретизации раз в десять и изменяйте размер окна. При каждой перерисовке кривая получается разной.

Собственно построение кривой осуществляется следующей командой:

```
gluNurbsCurve (theNurb, 8, @curveKnots, 3, @ctrlpoints, 4,
               GL_MAP1_VERTEX_3);
```

Первый аргумент— имя NURBS-объекта, вторым аргументом задается количество параметрических узлов кривой, третий аргумент — указатель на массив, хранящий значения этих узлов. Следующий параметр — смещение, т. е. сколько вещественных чисел содержится в порции данных, далее следует указатель на массив опорных точек. Последний аргумент равен порядку (степени) кривой плюс единица.

В документации указывается, что количество узлов должно равняться количеству опорных точек плюс порядок кривой. Количество опорных точек достаточно взять на единицу больше степени кривой, квадратичная кривая однозначно определяется тремя точками, кубическая — четырьмя и т. д. Так что для построения квадратичной кривой необходимо задавать шесть узлов:

```
gluNurbsCurve (theNurb, 6, @curveKnots, 3,
                        @CtripointS, 3, GL_MAP1_VERTEX_3) ;
```

Значения параметрических узлов в массиве или другой структуре, хранящей данные, должны быть упорядочены по неубыванию, т. с. каждое значение не может быть меньше предыдущего. Обычно значения первой половины узлов берутся равными нулю, значения второй половины задаются единичными, что соответствует неискаженной кривой, строящейся на всем интервале от первой до последней опорной точки.

Прежде чем мы сможем перейти к NURBS-поверхности, рекомендую самостоятельно поработать с этим примером, посмотреть на вид кривой при различных значениях параметров. Не забывайте о упорядоченности этих значе-

ний, и приготовьтесь к тому, что их набор не может быть совершенно произвольным: в некоторых ситуациях кривая строиться не будет.

Теперь мы можем перейти к поверхностям, и начнем с проекта из подкаталога Ex49 — модификации классического примера на эту тему, в котором строится холмообразная NURBS-поверхность (рис. 3.32).

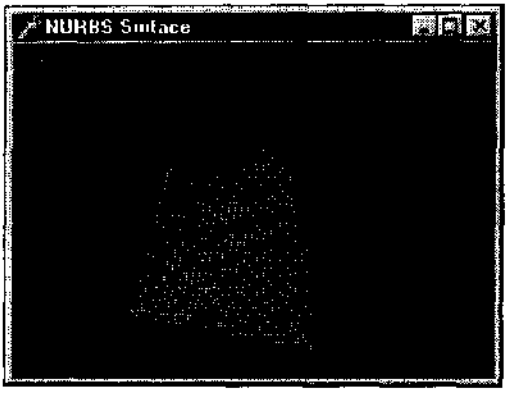


Рис. 3.32. Классический пример на использование NURBS-поверхности

Первым делом замечу, что здесь появился новый для нас режим:

```
glEnable (GL_NORMALIZE);
```

Сделано это из-за того, что поверхность при построении масштабируется, и чтобы автоматически рассчитанные нормали "не уплыли", и используется этот режим.

Режим воспроизведения меняется по нажатию клавиши ввода, для его установки используется та же команда gluNurbsProperty:

```
If solid
   then gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_FILL)
   else gluNurbsProperty(theNurb, GLU_DISPLAY_MODE, GLU_OUTLINE_POLYGON);
```

Команда собственно построения заключена в специальные командные скобки:

```
gluBeginSurface (theNurb);
   gluNurbsSurface (theNurb,
                    S, @knots,
                    8, @knots,
                    4 * 3,
                    3,
                    @ctrlpoints,
                    4, 4,
                    GL_MAP2_VERTEX_3);
gluEndSurface (theNurb),-
```

В данном примере эти скобки не обязательны, поскольку они обрамляют единственную команду.

Если вы внимательно разобрали примеры предыдущего раздела, то большинство параметров команды `gluNurbsSurface` не должны *вызывать* вопросов, они аналогичны параметрам команд для построения поверхности Безье.

Так, шестой и седьмой параметры задают "большой шаг" по каждой координате, ассоциированной с поверхностью, т. е. сколько вещественных чисел содержится в строке структуры данных и сколько вещественных чисел задают отдельную точку. Восьмой параметр — адрес массива контрольных точек, а последним параметром задается символическая константа, определяющая тип возвращаемых значений; в данном случае ими являются трехмерные координаты вершин.

В примере задано шестнадцать контрольных точек, располагающихся равномерно по координатам X и Y в пределах квадрата, третья координата для точек, лежащих на границе квадрата, равна —3, для внутренних опорных точек эта координата равна 7. Таким способом массив заполняется для получения холмообразной поверхности. Если по заданным опорным точкам построить поверхность Безье, то увидим точно такой же холмик, как и в рассматриваемом примере.

Отличает NURBS-поверхности то, что параметризируемы. Так, предпоследние два параметра задают степень (порядок) поверхности по координатам и и v. Задаваемое число, как сказано в документации, должно быть на единицу больше требуемой степени. Для поверхности, кубической по этим координатам, число должно равняться 4, как в нашем примере. Порядок нельзя задавать совершенно произвольным, ниже мы разберем имеющиеся ограничения.

Второй параметр команды — количество узлов в параметризации по u-направлению, третьим параметром задается адрес массива, хранящего значения узлов. Третий и четвертый параметры команды имеют аналогичный смысл, но для второго направления.

Массивы узлов должны заполняться значениями, упорядоченными по неубыванию.

Как сказано в файле справки, при заданных *uknot_count* и *vknot_count* количествах узлов, *uorder* и *vorder* порядках количество опорных точек должно равняться *(uknot_count — uorder)* x *(vknot_count — vorder)*. Так что при изменении порядка по координатам необходимо подбирать и все остальные параметры поверхности.

Если вы хотите подробнее узнать о параметризации и ее параметрах, то обратитесь к литературе с подробным изложением теории NURBS-поверхностей.

В данном примере используется один массив для задания узлов по обоим направлениям, а в проекте из подкаталога Ex50 используется два отдельных

массива — для каждой координаты задаются свои значении узлов. Поверхность строится не на всем интервале, а на части его, т. с. происходит подобие отсечения.

Чаще всего используются "кубические" NURBS-поверхности. Для иллюстрации построения поверхностей других порядков предназначен проект из подкаталога Ex51, где берется "квадратичная" поверхность.

Библиотека glu предоставляет также набор команд для вырезки кусков NURBS-поверхностей. Примером служит проект из подкаталога Ex52. Опорные точки поверхности располагаются в пространстве случайно, а затем из поверхности вырезается звездочка — рис. 3.33.

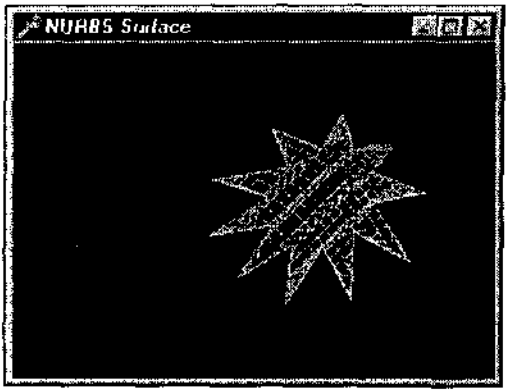


Рис. 3.33. Команды библиотеки glu позволяют строить невыпуклые многоугольники

Для хранения точек на границе вырезаемой области — звездочки — введен массив:

```
trim: Array [0..20, 0..1] of GLfloat;
```

Массив заполняется координатами точек, лежащих поочередно на двух вложенных окружностях:

```
procedure InitTrim;
var
  i: Integer;
begin
  For i := 0 to 20 do
    If Odd(i) then begin // нечетные точки — на внешней окружности
      trim [i, 0] := 0.5 * cos (i * Pi / 10) + 0.5;
      trim [i, 1] := 0.5 * sin (i * Pi / 10) + 0.5;
      end
      else begin          // четные точки — на внутренней окружности
      trim [i, 0] := 0.25 * cos (i * Pi / 10) + 0.5;
      trim [i, i] := 0.25 * sin (i * pi / 10) + 0.5;
  end;
end;
```

Внутри операторных скобок построения NURBS-поверхности вслед за командой `gluNurbsSurface` задаем область вырезки:

```
gluBeginTrim (theNurb);
  giuPwlCurve (theNurb, 21, @trim, 2, GLU_MAP1_TRIM_2);
gluEndTrim (theNurb);
```

При задании области вырезки используются опять-таки специальные командные скобки, собственно область вырезки задается командой `gluPwlCurve`. Команда задает замкнутую кусочно-линейную кривую; часть NURBS-поверхности, не вошедшая внутрь этой области, не воспроизводится. Второй аргумент — количество точек границы, третий — адрес массива этих точек, четвертым параметром является символьная константа, задающая тип вырезки.

В примере при каждом нажатии пробела вызывается процедура инициализации поверхности, так что вид звездочки каждый раз получается разным, случайным.

Следующий пример (подкаталог Ex53) также иллюстрирует вырезку NURBS-поверхности (рис. 3.34).

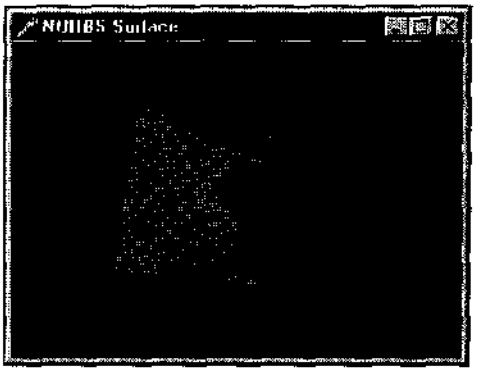


Рис. 3.34. Вырезка внутри NURBS-поверхности

Здесь вырезается внутренняя область поверхности, для чего задаются две линии, связанные с вырезкой:

```
   gluBeginTrim (theNurb);
      giuPwlCurve (theNurb, 5, @edgePt, 2, GLU_MAP1_TRIM_2) ;
   gluEndTrim (theNurb);

   gluBeginTrim (theNurb);
      gluNurbsCurve (theNurb, 8, @curveKnots, 2,
                     @curvePt, 4, GLU_MAP1_TRIM_2);
      giuPwlCurve (theNurb, 3, @pwlPt, 2, GLU_MAP1_TRIM_2);
gluEndTrim (theNurb) ;
```

Первая линия охватывает весь интервал по обеим координатам поверхности, в массиве указаны точки границы по часовой стрелке:

```
edgePt : Array [0..4, 0..1] of GLfloat = ((0.0, 0.0), (1.0, 0.0),
(1.0, 1.0), (0.0, 1.0), (0.0, 0.0));
```

Теперь следующая область вырезки ограничивает внутреннюю вырезаемую область. В примере эта область состоит из двух кривых; если это усложняет понимание программы, строку с вызовом `gluNurbsCurve` можете удалить, а массив граничных точек дополните замыкающей точкой:

```
pwlPt : Array [0..3, 0..1] of GLfloat = ((0.75, 0.5), (0.5, 0.25),
(0.25, 0.5), (0.75, 0.5));
```

Вторым параметром `gluPwlCurve` теперь надо указать 4. После этих манипуляций внутри поверхности вырезается треугольник — рис. 3.35.

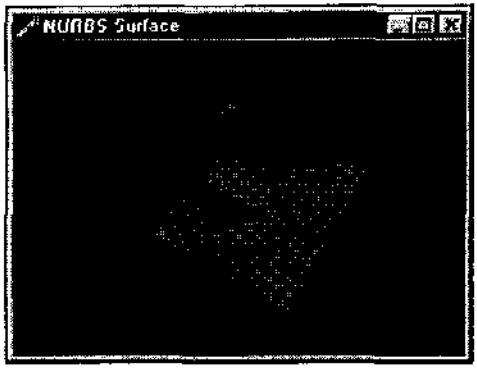


Рис. 3.35. Если предыдущий пример показался трудным, начните с более простого примера

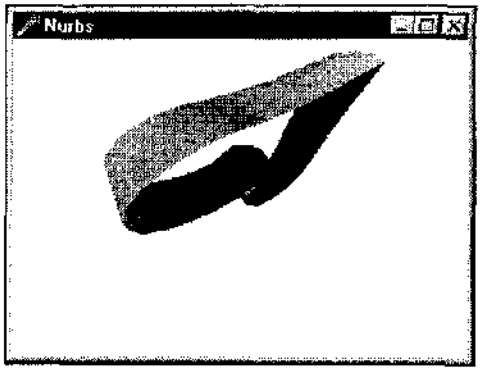


Рис. 3.36. "She eyes me like a piercer when I am weak..."

Последний пример этого раздела также из ряда классических примеров. Проект из подкаталога Ex54 после компиляции и запуска рисует на экране поверхность в форме карточного сердца — рис. 3.36.

Как и во многих предыдущих примерах, по нажатию пробела визуализируются опорные точки поверхности. Этот пример прекрасно иллюстрирует, как можно манипулировать параметрами NURBS-поверхности для выявления преимуществ параметризации. Обратите внимание, что опорные точки заданы только для одной половины воспроизводимой симметричной фигуры.

Дисплейные списки

В этом разделе мы познакомимся с одним из важнейших понятий OpenGL. Дисплейные списки представляют собой последовательность команд, запо-

минаемых для последующего вызова. Они подобны подпрограммам и являются удобным средством для кодирования больших систем со сложной иерархией.

Знакомство начнем с проекта из подкаталога Ex55, классического примера на эту тему. Работа приложения особо не впечатляет: рисуется десять треугольников и отрезок (рис. 3.37).

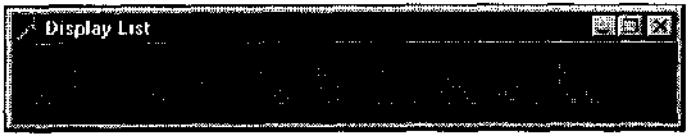


Рис. 3.37. Первый пример на использование дисплейных списков

В начале работы вызывается процедура инициализации, где создается (описывается) дисплейный список:

```
const
 listNane : GLUint = 1; // идентификатор списка
procedure init;
begin
 glNewList (listName, GL_COMPILE); /7 начало описания списка
   glCoior3f (1.0, 0.0, 0.0);
   glBegin (GL_TRIANGLES);
    glvertex2f (0.0, 0.0) ;
    glVertex2f (1.0, 0.0);
    glVertex2f (0.0, 1.0) ;
   glEnd;
   glTranslatef (1.5, 0.0, 0.0);
 glEndList;                          // конец описания списка
end;
```

Описание списка начинается с вызова команды glNewList, первым аргументом которой является целочисленный идентификатор — имя списка (в примере именем служит константа). Вторым аргументом команды является символическая константа; здесь мы указываем системе OpenGL на то, что список компилируется для возможного последующего использования. Все следующие до glEndList команды OpenGL и образуют дисплейный список, единую последовательность.

В данном примере в списке задается красный цвет примитивов, далее следуют команды для построения одного треугольника, заканчивается список переносом системы координат по оси X.

В коде воспроизведения кадра цвет примитивов задастся зеленым, затем десять раз вызывается описанный список:

```
glColor3f (0.0, 1.0, 0.0); // текущий цвет - зеленый
gl PushMatrix;             // запомнили систему координат
For i := 0 to 9 do         // десять раз вызывается список с именем 1
    glCailList (listName);
drawLine;                  // построить отрезок
glPopMatrix;               // вернулись в запомненную систему координат
```

Списки вызываются командой `glCallList`, единственным аргументом которой является идентификатор вызываемого списка.

В примере после каждого вызова списка номер один система координат смешается, поэтому на экране получается ряд треугольников, а отрезок рисуется вслед за последним треугольником. Хотя цвет примитивов задается зеленым, в списке он переустанавливается в красный, поэтому отрезок рисуется именно таким цветом.

Новички часто путаются, считая дисплейные списки полной аналогией подпрограмм. Отличает списки от подпрограмм то, что списки не имеют параметров, команды при вызове списка действуют точно так же, как и при создании списка. Если при описании списка используются переменные, то при вызове списка значения этих переменных никак не используются. Впрочем, ряд команд, помещенных в список, при вызове будут отрабатываться каждый раз заново, в документации приведен список таких команд.

Замечание

Я много раз встречал утверждение, что использование списков эффективно по причине того, что компиляция их приводит к более быстрому выполнению — в списке не запоминаются промежуточные команды. Если это и так, то ощутить "на глаз" ускорение работы приложения при использовании дисплейных списков как-то не получается. Тем не менее, я настоятельно рекомендую их использовать, это приводит к повышению читабельности и упрощению кода.

По окончании работы память, запятую списками, необходимо освобождать, как это делается в этом примере:

```
glDe: eteLists (listName, 1);
```

Первый аргумент команды — имя первого удаляемого *списка*, второй параметр — количество удаляемых списков. Удаляются все списки, следующие за указанным. Стоит напомнить, что удалять списки необходимо до освобождения контекста воспроизведения.

В продолжение темы рассмотрим проект из подкаталога Ex56. Результат работы приложения приведен на рис. 3.38.

Пример иллюстрирует несколько аспектов, связанных с дисплейными списками.

Во-первых, это использование команды `glGenLists`, генерирующей имя для списка. В примере идентификатором используемого списка является пере-

менная, а не константа. Перед описанием списка значение этой переменной подбирается системой OpenGL среди незанятых на текущий момент значений:

```
listName := glGenLists (1);
```

Рис. 3.38. В этом примере цветовые настройки при использовании списка не сбиваются

То есть вы можете и не затрачивать усилий на то, чтобы самому следить за именами используемых списков, а полностью положиться в этом вопросе па OpenGL. Это особенно удобно при интенсинной работе со списками, формируемыми в программе динамически.

Во-вторых, обратите внимание, что каждый вызов списка не портит текущие цветовые настройки. Корректность работы обеспечивается использованием пары новых для нас команд: glPushAttrib и glPopAttrib:

```
glNewList (listName,GL_COMPILE) ;
  glPushAttrib (GL_CURRENT_BIT); // запомнили текущие атрибуты цвета
  glColor3fv (@color_vector);    // установили нужный цвет
  glBegin (GL_TRIANGLES);           // отдельный треугольник
    glVertex2f (0.0, 0.0);
    glVertex2f (1.0, 0.0);
    glVertex2f (0.0, 1.0);
  giEnd;
  glTranslatef (1.5, 0.0, 0.0);
  glPopAttrib;       // восстановили запомненные настройки
glEndList;
```

Аналогично командам сохранения и восстановления системы координат, эта пара команд позволяет запоминать в отдельном стеке текущие настройки и возвращаться к ним. Аргумент команды glPushAttrib — символическая константа — задает, какие атрибуты будут запоминаться (значение в примере соответствует запоминанию цветовых настроек). Из документации можно узнать, что существует целое множество возможных констант для этой команды: если вам покажется сложным разбираться со всеми ними, то пользуйтесь универсальной константой GL_ALL_ATTRIB_BITS, В ЭТОМ случае в стеке атрибутов запомнятся разом все текущие установки.

При перерисовке окна текущая система координат запоминается перед вызовом списков и восстанавливается перед прорисовкой отрезка:

```
glPushMatrix;
For i := 0 to 9 do
   glCallList (listName);
glPopMatrix;
drawLine;
```

Поэтому в примере отрезок рисуется поверх ряда треугольников.

Следующий пример, проект из подкаталога Ex57, является очень простой иллюстрацией на использование команды glIsList, позволяющей узнать, существует ли в данный момент список с заданным именем. Аргумент команды — имя искомого списка, возвращает команда величину типа GLbooiean (как всегда в таких случаях, в Delphi обрабатываем результат как булевскую переменную):

```
If glIsList (listName)
   then ShowMessage ('Список с именем listName существует')
   else ShowMessage ('Списка с именем listName не существует');
```

Теперь перейдем к проекту из подкаталога Ex58. Результат работы программы отличается от первого примера этого раздела только тем. что рисуется восемь треугольников вместо десяти, но суть примера не в этом: здесь иллюстрируется то, что при описании списка можно использовать вызов других списков. В примере описано три списка:

```
const // идентификаторы используемых списков
 listNamei : GLUint = 1;
 listName2 : GLUint = 2;
 listName3 : GLUint = 3;

procedure init;
var
 i : GLuint;
begin
 glNewList (listNamei, GL_COMPILE); // список I - отдельный треугольник
   glColor3f (1.0, 0.0, 0.0);
   glBegin (GL_TRIANGLES);
    glVertex2f (0.0, СП) ;
    glVertex2f (1.0, 0.0);
    glVcrtex2f (0.0, 1.0);
   glEnd;
   glTranslatef (1.5, 0.0, 0.0);
 glEndList;

 glNewList (listName2, GL_COMPILE) ; // список 2 — четыре треугольника
   For i := 0 to 3 do
     glCallList (listNamei) ; //' вызывается прежде описанный список ]
 glEndList;
```

```
 glNewList (listName3, GL_COMPILE); // список 3 - четыре треугольника
   glCallList (listName2); // вызывается прежде описанный список 2
 glEndList;
end;
```

Второй и третий списки используют прежде описанные списки, но glCallList относится к тем командам, которые, будучи помещенными в список, при выполнении его выполняются независимо от дисплейного режима, т. е. не компилируются. Так что в этом примере результат не изменится, если описания второго и третьего списков поменять местами.

В примере введен массив, содержащий имена двух списков:

```
const
 list : Array [0..1] of GLUint = (2, 3) ;
```

Этот массив используется при воспроизведении кадра в команде, позволяющей вызвать сразу несколько дисплейных списков:

```
glCallLists (2, GL_INT, @list);
```

Первый аргумент команды — количество вызываемых списков, второй аргумент указывает смещение, третий аргумент — указатель на структуру, содержащую имена вызываемых списков. Смещение задано согласно описанию используемого массива из указанного в файле справки списка возможных констант.

В примере вначале вызывается список с именем 2 и строятся первые четыре треугольника. Затем вызывается список с именем 3, который состоит во вторичном вызове списка номер 2.

Замечание

Подобный прием может показаться малополезным, но важно хорошо разобраться в этом и следующем примерах, поскольку используемые в них команды будут применяться в дальнейшем для вывода текста.

Последний пример этого раздела, проект из подкаталога Ex59, по своему функциональному действию ничем не отличается от предыдущего, однако для нас в нем интересна новая команда — glListBase. Смысл ее очень прост — она задает базовое смещение для имен вызываемых списков.

В примере это смещение задано единичным:

```
glListBase (1);
```

А массив имен вызываемых списков содержит значения на единицу меньше, чем в предыдущем примере:

```
list : Array [0..1] of GLUint = (1, 2);
```

При вызове списков командой glCallLists к их именам прибавляется заданное смещение.

Tess-объекты

Мозаичные (tesselated — мозаичный) объекты являются последним нововведением библиотеки glu, предназначены они для упрощения построений невыпуклых многоугольников.

После того как мы изучили основные объекты библиотеки glu, нам будет несложно освоиться с собственно tess-объектами.

Работа с ними в Delphi сопряжена с трудностями, возникающими из-за того, что раздел заголовочного файла, посвященный этим объектам, содержит несколько ошибок.

Рис. 3.39 демонстрирует работу примера — проекта, расположенного в подкаталоге Ex6O.

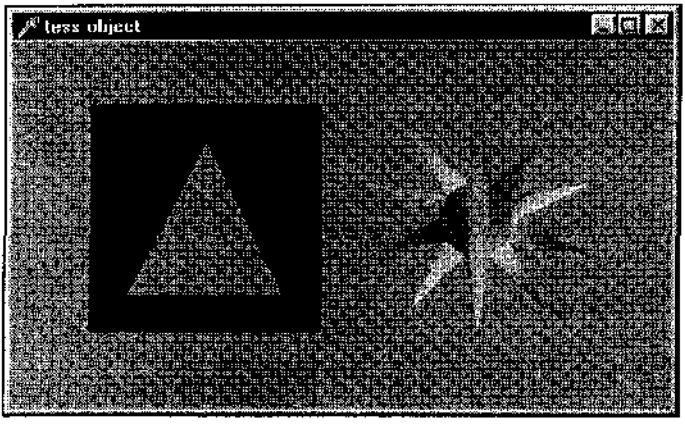


Рис. 3.39. Tess-объекты можно использовать для тех же целей, что и NURBS-поверхности

В программе определен особый тип для хранения координат одной точки:

```
type
  TVector = Array [0..2] of GLdouble;
```

Для уверенной работы команд данного раздела следует использовать именно тип удвоенной точности.

В процедуре инициализации описана переменная специального типа, введенного для работы с мозаичными объектами. Она инициализируется приблизительно так же, как другие объекты библиотеки glu, но, конечно, другой командой:

```
var
   tobj : gluTesselator;
...
   tobj := gluNewTess;
```

Теперь посмотрим, как подготавливается список для левой фигуры, квадрата с треугольным отверстием внутри.

С помощью команды giuTessCallback задаются адреса процедур, вызываемых на различных этапах рисования tess-объекта, например:

```
gluTessCallback(tobj, GLU_TESS_BEGIN, @glBegin); // начало рисования
```

При начале рисования объекта мы не планируем особых манипуляций, поэтому просто передаем адрес процедуры glBegin.

Если же нам по сценарию потребуется выполнить какие-то дополнительные действия, необходимо описать пользовательскую несвязанную (не являющуюся частью описания класса) процедуру, обязательно указав ключевое слово stdcall, и передавать адрес этой процедуры.

Синтаксис описания подобных процедур описан в справке по команде giuTessCallback. Например, если мы хотим, чтобы перед воспроизведением примитива подавался бы звуковой сигнал, необходимо вызвать следующую процедуру:

```
procedure beginCallback(which : GLenum);stdcall;
begin
   MessageBeep (MB_OK);
   glBegin(which);
end;
...
gluTessCallback(tobj, GLU_TESS_BEGIN, @ beginCailback);
```

Имя процедуры безразлично, но аргументы ее должны задаваться строго по указаниям, содержащимся в документации.

Внимательно посмотрите на следующие две строки кода:

```
gluTessCallback(tobj, GLU_TESS_VERTEX, @glVertex3dv); // вершина
giuTessCallback (tobj, GLU_TESS_END, @glEnd) ;  // конец рисования
```

То есть при обработке отдельной вершины и в конце рисования примитивов также не будет выполняться чего-то необычного.

Для диагностики ошибок, возникающих при работе с tess-объектами, используем пользовательскую процедуру:

```
procedure errorCallback (errorCode : GLenum) ; stdcall;
begin
```

```
  ShowMessage (gluErrorString(errorCode));
end;
...
   gluTessCallback(tobj, GLU_TESS_ERROR, @errorCallback); // ошибка
```

Команда gluErrorString возвращает строку с описанием возникшей ошибки.

Описание ошибки выдается на русском языке (на языке локализации операционной системы), так что с диагностикой проблем не будет.

Координаты вершин квадрата и треугольника вырезки хранятся в структурах, единицей данных которых должна быть тройка вещественных чисел:

```
const
   rect : Array [0..3] of TVector = ((50.0, 50.0, 0.0),
                           (200.0, 50.0, 0.0),
                           (200.0, 200.0, 0.0),
                           (50.0, 200.0, 0.0));
   tri : Array[0..2] of TVector = ((75.0, 75.0, 0.0),
                          (125.0, 175.0, 0.0),
                          (175.0, 75.0, 0.0));
```

Наша фигура строится приблизительно по таким же принципам, что и в примере на вырезку в NURBS-поверхности:

```
glNewList(1, GL_COMPILE);
glColor3f(0.0, 0.0, 1.0); // цвет - синий
gluTessBeginPolygon (tobj, nil); // начался tess-многоугольник
   gluTessBeginContour(tobj); // экешний контур — квадрат
      gluTessVertex(tobj, @rect(0], @rect[0]); // вершины квадрата
      gluTessVertex(tobj, @rect[1], @rect [1]);
      gluTessVertex(tobj, @rect[2], @rect[2]);
      gluTessVertex(tobj, @rect[3], @rect[3]);
   gluTessEndContour(tobj);
   gluTessBeginContour(tobj); // следующие контуры задают вырезки
      gluTessVertex(tobj, @tri[0], @tri [0]:; // треугольник
      gluTessVertex (tobj, @tri[1], @tri [1] );
      gluTessVertex(tobj, @tri[2], @tri[2]);
   gluTessEndContour(tobj);
gluTessEndPolygon(tobj); // закончили с tess-многоугольником
glEndList;
```

При перерисовке окна просто вызывается список с именем 1.

После того как список описан, tess-объект можно удалить, это делается в конце процедуры инициализации:

```
gluDeleteTess(tobj);
```

Замечание

Обратите внимание: при вызове списка сами объекты библиотеки glu уже не используются. Точно так же вы можете удалять quadric-объекты сразу после описания всех списков, использующих их.

Надеюсь, с первой фигурой вам все понятно, и мы можем перейти ко второй фигуре, звездочке.

Здесь для наглядности перед вызовом каждой вершины текущий цвет меняется, для чего описана специальная процедура, адрес которой задается вызовом Команды gluTessCallback:

```
procedure vertexCallback (vertex : Pointer);stdcall;
begin
  glColor3f (random, random, random);
  glVertex3dv (vertex);
end;
...
gluTessCallback(tobj, GLU_TESS_VERTEX, @vertexcallback) ;
```

Массив, хранящий координаты вершин звездочки, заполняется приблизительно так же, как в одном из предыдущих примеров на NURBS-поверхность. Многоугольник второго списка состоит из единственного контура. Перечисляем вершины, хранящиеся в массиве:

```
glNewList(2, GL_COMPILE);
gluTessBeginPolygon (tobj, nil);
   gluTessBeginContour(tobj);
     For i := 0 to 20 do
      gluTessVertex(tobj,@star[i],@star[ij:;
   gluTessEndContour(tobj);
gluTessEndPolygon(tobj);
glEndList;
```

Прототип одной из используемых команд мне пришлось переписать:

```
procedure gluTessBeginPolygon (tess: GLUtesselator; polygon_data:
Pointer) ; stdcall; external GLU32;
```

То, что записано в стандартном заголовочном файле, расходится с документацией и приводит к ошибке.

Мы рассмотрели простейший пример на использование tess-объектов, и надеюсь, вы смогли оценить, как удобно теперь становится рисовать невыпуклые многоугольники.

В качестве исследования можете задать контурный режим воспроизведения многоугольников, чтобы увидеть, как строятся получающиеся фигуры.

Приведу еще несколько примеров на мозаичные объекты.

Подкаталог Ex61 содержит проект, где строится объект в виде звездочки (рис. 3.40).

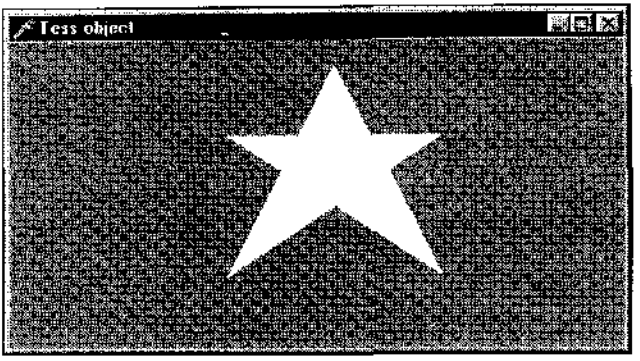


Рис. 3.40. Звездочка построена по координатам пяти вершин

На первый взгляд вам может показаться, что этот пример проще предыдущего, однако это не так: звездочка задана координатами пяти вершин, перечисляемых в том порядке, в котором обычно дети рисуют пятиконечные звезды. Мне пришлось изрядно потрудиться, чтобы этот пример заработал.

Если попытаться в предыдущем примере задать вершины звезды именно по такому принципу, мы получим сообщение об ошибке "требуется составной ответный вызов" (на русском). То есть серверу надо подсказать, как обрабатывать ситуации с пересечениями границ контура.

Почерпнув всю требуемую информацию из файла справки, *я* написал следующую процедуру и задал ее адрес для вызова обработчика пересечения:

```
gluTessCallback(tobj, GLU_TESS_COMBINE, @combineCallback);
```

Для того чтобы заполнить внутренности фигуры, я использовал команду, позволяющую определять свойства мозаичного объекта:

```
gluTessProperty(tobj, GLU_TESS_WINDING_RULE, GLU_TESS_WINDING_POSITIVE);
```

Обратите внимание, что значение первой символической константы в программе переопределено, в файле opengl.pas это значение задано неверно.

Предоставляю вам еще один пример на эту тему, проект из подкаталога Ex62. Не стану разбирать этот пример, чтобы не испугать начинающих. Если он вам покажется трудным, можете отложить его подробное изучение на потом — до того момента, когда вы будете чувствовать себя с OpenGL совсем уверенно.

По ходу подготовки этого примера я обнаружил массу ошибок все в том же заголовочном файле, так что мне самому он дался очень тяжело.

Таймеры и потоки

В этом разделе мы познакомимся с различными способами создания анимации. При этом нам придется сосредоточиться на вопросах, больше связанных с операционной системой- чем с OpenGL.

Для самых простейших задач вполне подходит использование обычного таймера Delphi.

Посмотрим проект из подкаталога Ex63. Это немного модифицированный пример второй главы, в котором вокруг курсора рисовалось облачко отрезков. Теперь это облачко постоянно меняется так, что картинка еще более напоминает бенгальский огонь.

Первое изменение в проекте заключается в том, что на форме появился таймер, обработчик которого заключается в том, что десять раз в секунду окно перерисовывается. При каждой перерисовке окна вокруг курсора рисуется множество отрезков со случайными координатами конца.

Важно обратить внимание на действия, позволяющие ускорить работу приложения. Вместо традиционного для проектов Delphi вызова метода Refresh окно перерисовывается вызовом функции **API** (мы уже убедились, насколько это значимо):

```
procedure TfrmGL.Timer1Timer(Sender: TObject);
begin
  InvalidateRect(Handle, nil, False);
end;
```

Цвет окна формы я намеренно задал ярко-синим, чтобы проиллюстрировать, как важно в таких приложениях бороться за каждую миллисекунду. Если в обработчике таймера поставить Refresh, то при каждой перерисовке окно мерцает, по нему пробегают синие полосы. Впрочем, может случиться и так, что на вашем компьютере такие эффекты не возникают, тут многое зависит от характеристик "железа".

Также для ускорения работы в этом примере вместо `Canvas.Handle` используется явно полученная ссылка на контекст устройства.

Код перерисовки окна максимально сокращен, в нем оставлено только то, что не переносится в другие обработчики. Включение режима штриховки и задание области вывода перемешены в обработчики oncreate и onResize формы, соответственно.

В таких приложениях также желательно использовать перехватчик сообщения `WM_PAINT` вместо обработчика `OnPaint`.

Это сделано в следующем примере, проекте из подкаталога Ex64. в котором на экране двигаются по кругу шесть кубиков (рис. 3.41).

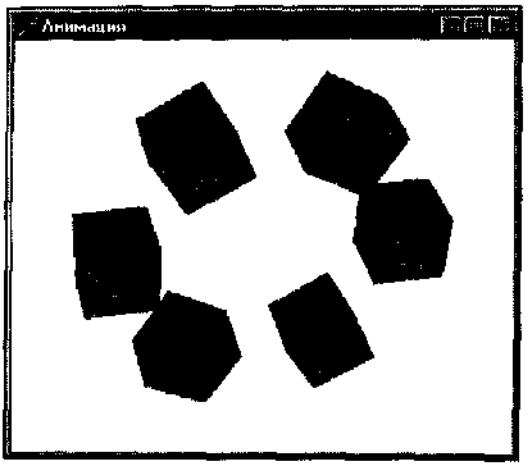


Рис. 3.41. При работе программы кубики вращаются по кругу

Положения центров кубиков хранятся во вспомогательных массивах, заполняемых при начале работы приложения синусами и косинусами:

```
For i := 0 to 5 do begin
     wrkX [i] := sin (Pi / 3 * i);
     wrkY [i] := cos (Pi / 3 * i);
end;
```

Поворот всей системы с течением времени обеспечивается тем, что в обработчике таймера значение переменной, связанной с углом поворота, увеличивается, после чего экран перерисовывается:

```
Angle := Angle + 1; // значение угла изменяется каждый "тик"
If Angle >= 60.0 then Angle := 0.0;
InvalidateRect(Handle, nil, False);
```

Ядро кода воспроизведения кадра выглядит так:

```
glPushMatrix;                  // запомнили начальную систему координат
glRotatef(Angle, 0.0, 0.0, 1.0);// поворот системы на угол Angle по Z
{Цикл рисования шести кубиков}
For i := 0 to 5 do begin
 glPushMatrix;                              // запомнили систему координат
 glTranslatef(wrkX [i], wrkY [i], 0.0); // перенос системы координат
 glRotatef(-60 * i, 0.0, 0.0, 1.0};     // поворот кубика
 glutSolidCube (0.5);                       // рисуем кубик
 glPopMatrix;                               // вернулись в точку
 end;
glPopMatrix;    // вернулись в первоначальную систему координат
```

Занятный результат получается, если поменять местами первые две строки этого кода, обязательно посмотрите, к чему это приведет.

Замечание

Приведет это к тому, что кубики будут вращаться все быстрее и быстрее: теперь они при каждом тике таймера поворачиваются на все более увеличивающийся угол Angle относительно предыдущего положения. Можно использовать более оптимальный прием в подобных примерах: не использовать управляющую переменную (здесь это Angle), не использовать команды glPushMatrix и glPopMatrix, а код кадра начинать с поворота на угол, константу. С точки зрения скорости это оптимально, но может нарушать сценарий кадра: ведь при изменении размеров окна мы принудительно возвращаем объекты сцены в первоначальную систему координат, и кубики резко дергаются.

Использование системного таймера является самым простым решением задачи, но имеет очевидные недостатки. На маломощных компьютерах уже этот пример выводит кадры рывками, а если количество объектов перевалит за два десятка, то удовлетворительную скорость воспроизведения можно будет получить только па очень хороших машинах.

Следующий пример, проект из подкаталога Ex65, является продолжением предыдущего, здесь рисуется пятьдесят параллелепипедов (рис. 3.42).

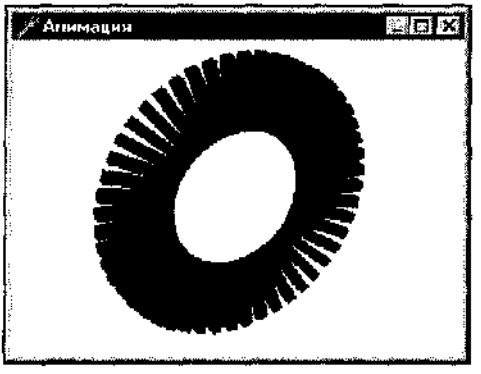


Рис. 3.42. Эту систему мы возьмем в качестве тестовой для сравнения методов создания анимации

Вся система вращается по двум осям, по оси Y вращение происходит с удвоенной скоростью:

```
glRotatef(2 * Angle, 0.0, 1.0, 0.0); // поворот по оси Y
glRotatef(Angle, 0.0, 0.0, 1.0);     // поворот по оси Z
```

Интернат таймера я задал равным 50 миллисекунд, т. е. экран должен обновляться двадцать раз в секунду. Попробуем выяснить, сколько кадров в секунду выводится в действительности.

Это делается в проекте из подкаталога Ex66.

Введены переменные, счетчики кадров и количество кадров в секунду:

```
newCount, frameCount, lastCount : LongInt;
fpsRate : GLfloat;
```

При запуске приложения инициализируем значения:

```
lastCount := GetTickCount;
frameCount := 0;
```

Функция API GetTickCount возвращает количество миллисекунд, прошедших с начала сеанса работы операционной системы.

При воспроизведении кадра определяем, прошла ли очередная секунда, и вычисляем количество кадров, выведенных за эту секунду:

```
newCount := GetTickCount; // текущее условнее время
Inc(frameCount);      // увеличиваем счетчик кадров
If (newCount - lastCount) > 1000 then begin // прошла секунда
   // определяем количество выведенных кадров
   fpsRate := frameCount * 1000 / (newCount - lastCount);
   // выводим в заголовке количество кадров
   Caption := 'FPS - ' + FloatToStr (fpsRate);
   lastCount := newCount; // запоминаем текущее время
   frameCount := 0; // обнуляем счетчик кадров
end;
```

Получающееся количество воспроизведенных кадров в секунду зависит от многих факторов, в первую очередь, конечно, от характеристик компьютера, и я не могу предсказать, какую цифру получите именно вы. Будет совсем неплохо, если эта цифра будет в районе двадцати.

Но если задаться целью увеличить скорость работы этого примера, то выяснится, что сделать это будет невозможно, независимо от характеристик компьютера. Сколь бы малым не задавать интервал таймера, выдаваемая частота воспроизведения не изменится, системный таймер в принципе не способен обрабатывать тики с интервалом менее пятидесяти миллисекунд. Еще один недостаток такого подхода состоит в том, что если обработчик тика таймера не успевает отработать все действия за положенный интервал времени, то последующие вызовы этого обработчика становятся в очередь. Это приводит к тому, что приложение работает с разной скоростью на различных компьютерах.

Мы не будем опускать руки, а поищем другие способы анимации, благо их существует несколько. Рассмотрим еще один из этих способов (я его нахожу привлекательным), состоящий в использовании модуля MMSystem (Multimedia System). Мультимедийный таймер позволяет обрабатывать события с любой частотой, настолько часто, насколько это позволяют сделать ресурсы компьютера.

Посмотрим на соответствующий пример — проект из подкаталога Ex67. Здесь рисуются все те же пятьдесят параллелепипедов, но частота смены кадров существенно выше.

Список подключаемых модулей в секции implementation дополнился модулем MMSystem:

```
uses DGLUT, MMSystem;
```

Мультимедийный таймер также нуждается в идентификаторе, как и обычный системный, описываем его в разделе private класса формы:

```
TimerId : uint;
```

А вот процедура, обрабатывающая тик таймера, не может являться частью класса:

```
procedure TimeProc(uTimerID, uMessage: UINT;dwUser, dw1, dw2: DWORD;
stdcall;
begin
 // значение угла изменяется каждый "тик"
 With frmGL do begin
      Angle := Angle + 0.1;
      If Angle >= 360.0 then Angle := 0.0;
      InvalidateRect(Handle, nil, False);
 end;
end;
```

При создании окна таймер запускается специальной функцией API:

```
TimerID := timeSetEvent (2, 0, @TimeProc, 0, TIME_PERIODIC) ;
```

По окончании работы приложения таймер необходимо остановить; если это не сделать, то работа операционной системы будет заметно замедляться:

```
timeKillEvent(TimerID);
```

Самым важным в этой цепочке действий является, конечно, команда установки таймера, timeSetEvent. Первый аргумент команды — интервал таймера в миллисекундах. Второй аргумент — разрешение таймера, т. е. количество миллисекунд, ограничивающее время на отработку каждого тика таймера. Если это число задано нулем, как в нашем примере, то обработка таймера должна происходить с максимальной точностью.

Замечание

В документации рекомендуется задавать ненулевое значение для уменьшения системных потерь.

Следующий параметр — адрес функции, ответственной за обработку каждого тика. Четвертый параметр редко используется, им являются задаваемые пользователем данные возврата. Последним параметром является симболи-

ческам константа, при этом значение TIME_PERIODIC соответствует обычному поведению таймера.

Итак, в примере каждые две миллисекунды наращивается угол поворота системы и перерисовывается экран.

Конечно, за две миллисекунды экран не будет перерисован, однако те кадры, которые компьютер не успевает воспроизвести, не будут накапливаться.

При использовании мультимедийного таймера сравнительно легко планировать поведение приложения во времени: на любом компьютере и в любой ситуации система будет вращаться с приблизительно одинаковой скоростью, просто на маломощных компьютерах повороты будут рывками.

В продолжение темы посмотрите проект из подкаталога E.\68. где все тот же мультфильм рисуется на поверхности рабочего стола, подобно экранным заставкам. Во второй главе мы уже рассматривали похожий пример, здесь добавлена анимация, а команда glViewPort удалена, чтобы не ограничивать область вывода размерами окна приложения.

Замечание

Напоминаю, что такой подход срабатывает не на каждой карте.

Наиболее распространенным способом построения анимационны,\ приложений является использование фоновой обработки, альтернативы таймерам.

Разберем, как это делается.

В Delphi событие OnIdle объекта Application соответствует режиму ожидания приложением сообщений. Все, что мы поместим в обработчике этого события, будет выполняться приложением беспрерывно, пока оно находится в режиме ожидания.

Переходим к примеру, проекту из подкаталога Ex69. Сценарий приложения не меняем, чтобы можно было сравнить различные способы. Отличает пример то, что в нем отсутствуют какие-либо таймеры; код, связанный с анимацией, перешел в пользовательскую процедуру:

```
procedure TfrmGL. Idle (Sender:TObject;varDone:boolean) ;
beg ir.
 With frmGL do begin
      Angle := Angle + 0.1;
      If Angle >= 360.0 then Ancle :-•• 0.0;
      Done := False; // обработка завершена
      InvalidateRect(Handle, nil, False);
 end;
end;
```

Второй параметр Done используется для того, чтобы сообщить системе, требуется ли дальнейшая обработка в состоянии простоя, или алгоритм завершен. Обычно задается False, ЧТОбы не вызывать функцию WaitMessage.

При создании окна устанавливаем обработчик события OnIdle объекта Application:

```
Application.OnIdle := Idle;
```

Вот и все необходимые действия, можете запустить приложение и сравнить этот метод с предыдущим.

Замечание

При тестировании на компьютерах, оснащенных ускорителем, этот способ дал наивысший показатель при условии обычной загруженности системы. На компьютере без акселератора цифры должны получиться одинаковые, однако примеры с мультимедийным таймером выглядят поживее.

При использовании последнего способа у нас нет никакого контроля над выполнением кода, есть только весьма приблизительное представление о том, сколько времени будет затрачено на его выполнение. Скорость работы приложения в этом случае полностью зависит от загруженности системы, другие приложения могут с легкостью отнимать ресурсы, и тогда работа нашего приложения замедлится. Запустите одновременно приложения, соответствующие мультимедийному таймеру и фоновой обработке. Активное приложение будет генерировать большую частоту кадров, но приложение, построенное на таймере, менее болезненно реагирует на потерю фокуса.

В таких случаях можно повышать приоритет процесса, этот прием мы рассмотрим в главе 5.

Замечу, что при запуске приложения или изменении его размеров требуется несколько секунд, чтобы работа вошла в нормальный режим, поэтому первые цифры, выдаваемые в качестве частоты воспроизведения, не являются особо надежными.

Теперь посмотрим, как реализовать фоновый режим в проектах, основанных только на функциях API. Это иллюстрирует проект из подкаталога Ex70.

Пользовательская функция Idle содержит код, связанный с изменениями кадра.

Для отслеживания состояния активности окна заведен флаг AppActive, а оконная функция дополнилась обработчиком сообщения, связанного с активацией окна:

```
WM_ACTIVATEAPP:
      If (wParam = WA_ACTIVE) or (wParam = WA_CLICKACTIVE)
         then AppActive := True
         else AppActive := False;
```

Кардинальные изменения коснулись цикла обработки сообщений, вместо которого появился вот такой вечный цикл:

```
While True do begin
  // проверяем очередь на наличие сообщения
  If PeekMessage(Message,0, 0, 0, pm_NoRenove) then begin
       // в очереди присутствует какое-то сообщение
     If not GetMessage(Message, 0, 0, 0)
        then Break // сообщение WM_QUIT, прервать вечный цикл
        else begin // обрабатываем сообщение
          TranslateMessage(Message);
          DispatchMessage(Message);
        end;
      end
      else // очередь сообщений пуста
      If AppActive
         then Idle // приложение активно, рисуем очередной кадр
         else WaitMessage; // приложение не активно, ничего не делаем
end;
```

Надеюсь, все понятно по комментариям, приведу только небольшие пояснения.

Функция PeekMessage с такими параметрами, как в этом примере, не удаляет сообщение из очереди, чтобы обработать его в дальнейшем традиционным способом.

Функция Idle в этом примере вызывается при пустой очереди только в случае активности приложения.

Код можно немного сократить, если не акцентироваться па том, активно ли приложение; в данном же случае минимизированное приложение "засыпает", не внося изменений в кадр.

Рассмотрим еще один прием, использующийся для построения анимационных приложений и заключающийся в зацикливании программы. Для начала приведу самый простой способ зацикливания программы (проект из подкаталога Ex71).

Сценарий не изменился, рисуется все то же подобие шестерни. Никаких таймеров и прочих приемов, просто обработчик перерисовки окна заканчивается приращением управляющей переменной и командой перерисовки региона (окна):

```
Angle := Angle + 0.1;
If Angle >= 360.0 then Angle := 0.0;
InvalidateRect(Handle, nil, False);
```

Все просто: воспроизведя очередной кадр, подаем команду на воспроизведение следующего.

В этом методе, как, впрочем, и в предыдущем, при уменьшении размеров окна частота кадров увеличивается, но и вращение происходит быстрее, здесь так же отсутствует контроль за поведением системы.

Приложение "замирает", будучи минимизированным.

Если присмотреться внимательнее к поведению этого примера, то можно заметить некоторые необычные вещи, например, при наведении курсора на системные кнопки в заголовке окна подсказки не появляются. А при попытке активизации системного меню окна появляется явная накладка в работе приложения, область меню требуется перерисовать, двигая курсор в пределах его границ. Короче, зацикливание программы приводит к тому, что ожидающие сообщения могут и не обрабатываться. Может, это и не страшно, но ведь у нас нет гарантии, что мы обнаружили все странности работы приложения.

Решение проблемы состоит в использовании функции `ProcessMessages` объекта `Application`, приостанавливающей работу приложения, чтобы система могла обрабатывать сообщения из очереди.

Посмотрим на проект из подкаталога Ex72. Перед перерисовкой окна обрабатываем все сообщения очереди, вызвав функцию ProcessMessages, однако этого добавления не достаточно, иначе приложение невозможно будет закрыть. В примере введен вспомогательный флаг closed, принимающий истинное значение при попытке пользователя или системы закрыть приложение:

```
procedure TfrmGL.FormCloseQuery(Sender: TObject; var CanClose: Boolean);
begin
  Closed := True
end;
```

Теперь следующий кадр воспроизводится только в том случае, если не поступало сигнала о том, что приложение необходимо закрыть:

```
If not Closed then begin
   Angle := Angle + 0.1;
   If Angle >= 360.0 then Angle := 0.0;
   Application.ProcessMessages;
   InvalidateRect(Handle, nil, False);
end;
```

Этот вариант лишен недостатков предыдущего и реагирует на все поступающие сообщения, но приобрел новую особенность: если, например, активировать системное меню окна приложения, то вращение системы останавливается, пока меню не исчезнет.

Чтобы вы не забыли о том, что можно перемежать вывод командами OpenGL и обычными средствами операционной системы, предлагаю в этом примере вывод частоты кадров осуществлять не в заголовке окна, а на поверхности формы, заменив соответствующую строку кода на такую:

```
Canvas.TextOut (0, 0, 'FPS - ' + FloatToStr (fpsRate));
```

При выводе на поверхность окна с помощью функций GDI не должно возникать проблем ни с одной картой, а вот если попытаться текст выводить на метку, то проблемы, скорее всего, возникнут со всеми картами: метка не будет видна.

Следующий пример, проект из подкаталога Ex73, является очередным моим переводом на Delphi классической программы, изначально написанной на C профессиональными программистами корпорации Silicon Graphics. Экран заполнен движущимися точками так, что у наблюдателя может появиться ощущение полета в космосе среди звезд (рис. 3.43).

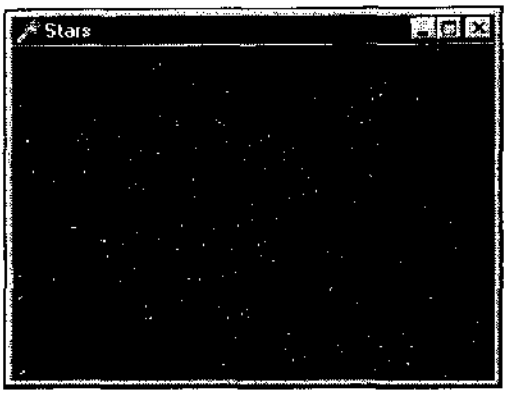


Рис. 3.43. Проект Stars создает иллюзию полета в космосе

Предусмотрены два режима работы программы, управление которыми осуществляется нажатием пробела и клавиши 'T. После нажатия пробела некоторые звезды летят по "неправильной" траектории, после нажатия второй управляющей клавиши происходит "ускорение" полета на некоторое время.

Последний прием, который мы разберем в этом разделе, основан на использовании потоков. Проект из подкаталога Ex74 иллюстрирует этот прием на примере хорошо нам знакомой по предыдущим упражнениям вращающейся шестерни.

В программе введен тип. ответственный за используемый поток:

```
type
  TGLThread = class (TThread)
    protected
    procedure Execute; override; // метод обязательно переопределяется
    procedure Paint; // пользовательский метод потока
  end;
```

Два метода потока описаны в традиционном стиле:

```
procedure TGLThread.Paint:;
begin
```

```
  With frmGL do begin
     Angle := Angle + 0.1;
     If Angle >= 360.0 then Angle := 0.0;
     InvalidateRect(Handle, nil, False);
  end;
end;

procedure TGLThread.Execute;
begin
  repeat
     Synchronize (Paint); // синхронизация потоков
  until Terminated;
end;
```

После создания окна поток инициализируется и запускается:

```
GLThread := TGLThread.Create (False);
```

По окончании работы приложения выполняются стандартные действия:

```
GLThread.Suspend; // приостановить поток
GLThread.Free;    // удалить поток
```

В этом разделе мы рассмотрели несколько способов организации анимационных программ. У каждого из этих методов есть свои достоинства и свои недостатки, и вы вольны самостоятельно решать, какой из этих способов является для вас наиболее подходящим. Если при уменьшении размеров окна частота воспроизведения увеличивается, это является положительной стороной метода, но если в данном методе невозможно предугадать поведение программы на разных машинах, то это можно отнести к его недостаткам. Повторю, что обработка ожидания является самым распространенным способом, и при обычной нагрузке системы он должен показать максимальную частоту воспроизведения.

В оставшихся примерах книги вы можете встретить самые разные из этих способов, я не стану придерживаться какого-либо одного.

Приведу еще один пример на анимацию, проект из подкаталога Ex75, где используется обычный системный таймер. В примере рисуется фонтан из двух тысяч точек (рис. 3.44).

Как и в примере на звезды, в отдельных массивах хранится информация для каждого объекта о текущем положении и направлении движения.

На смену каждой "упавшей точки" струя фонтана дополняется новой:

```
procedure UpdatePOINT(i: Word); // процедура перемещения капли
begin
  points[i][0] := points[i][0] + motion[i][0]; // изменение координат
```

```
  points fi] [1] :=points [i] [11 •motion • i ( •i ';
  points[i] [2] := points [i] [2]+motion[i] [2];
  If points [i] [1] < -0.75 then begin // капля фонтана упала на землю
    points[i][0] := 0.0; // новая капля вырывается из фонтана
    points[i][1] := -0.5;
    points[i]/2.] := 0.0;
    motion[i] [0] := (Random -- 0.5) / 20;
    motion[i][1] := Random / 7 + 0.01;
    mot:on[i] "2 ; := (Random - 0.5) / 20;
    end
    else motion[i ; {1] := motion[i] [1] - 0.01; // условная сила тяготения
end;
```

Меняя значение силы тяготения, можно регулировать высоту фонтана.

Последним примером главы станет проект из подкаталога Ex76, один из получающихся кадров работы программы приведен на рис. 3.45.

Если вы собираетесь распространять приложение, то позаботьтесь о том, чтобы ваше приложение на компьютерах сторонних пользователей не стало работать так быстро, что зритель ничего не сможет разглядеть на экране.

Разберем, как это сделать.

Предположим, управляющая переменная описана следующим образом:

```
Angle : GLinL = 0;
```

Вы пользуетесь не таймером, а скажем, зацикливаете программу. Шаг увеличения переменной вы задаете так, что на вашем компьютере движение объектов происходит ровно и с удовлетворительной скоростью:

```
Angle := (Angle + 2) mod 360;
```

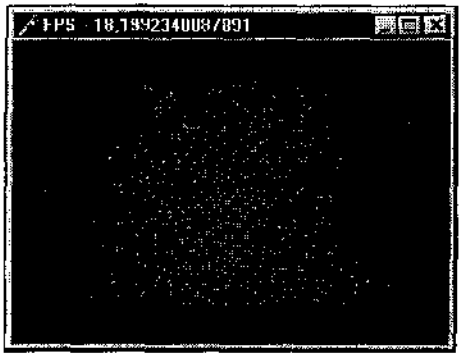


Рис. 3.44. Проект Fontain, две тысячи капель

Рис. 3.45. Теперь вы умеете рисовать даже такие "художественные произведения"

Может случиться, что на компьютере пользователя скорость воспроизведения окажется *р* пять раз выше, и тогда объекты на сцене будут перемещаться в пять раз быстрее.

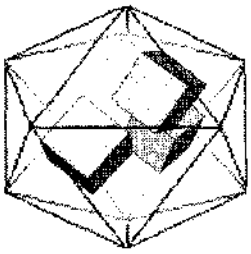
В таких случаях лучше опираться на системное время. Например, введите переменную для хранения текущего времени, а управляющую переменную задайте вещественной:

```
Angle : GLfloat = 0;
time : LongInt;
```

Если по сценарию полный оборот должен произойти за десять секунд, то код должен быть таким:

```
Angle := Angle + 0.1 * (GetTickCount - time) * 360 / 1000;
If Angle >= 360.0 then Angle := 0.0;
time := GetTickCount;
```

ГЛАВА 4



Визуальные эффекты

Эта глава посвящена тому, как повысить качество получаемых образов и как получить некоторые специальные эффекты. Приступать к ней стоит только после того, как материал предыдущих глав основательно изучен.

Примеры к главе помещены на дискете в каталоге Chapter4.

Подробнее об источнике света

Начнем с позиции источника света в пространстве. Раньше мы пользовались источником света со всеми характеристиками, задаваемыми по умолчанию. Источник света по умолчанию располагается в пространстве в точке с координатами (0, 0, 1).

Поучимся менять позицию на примере проекта из подкаталога Ex01. На экране присутствует тор, вокруг которого вращается источник света. Позиция источника света визуализируется каркасным кубиком. Игра теней на поверхности тора меняется в зависимости от текущего положении источника света.

Замечание

Напомню: если режим GL_COLOR_MATERIAL не включен, то текущие цветовые установки не влияют на цвет поверхности тора, поэтому он выглядит серым, хотя текущий цвет задан зеленоватым.

Переменная spin задает угол поворота системы координат, связанной с источником света по оси X. Положение источника света в этой системе координат определено в массиве position:

```
const
  position : Array [0..3] of GLfloat = (0.0, 0.0, 1.5, 1.0);
```

Перерисовка кадра выглядит так:

```
glPushMatrix; // запомнили мировую систему координат
glRotated (spin, 1.0, 0.0, 0.0); // поворот системы координат
glLightfv (GL_LIGHT0, GL_POSITION, @position); // задаем новую позицию
                                                // источника света
glTranslated (0.0, 0.0, 1.5); // перемещаемся в точку, где
                              // располагается источник света
glDisable (GL_LIGHTING); // отключаем источник света
glutWireCube (0..1);     // визуализируем источник света
glEnable (GL_LIGHTING); // включаем источник света
glPopMatrix;        // возвращаемся в мировую систему координат

glutSolidTorus (0.275, 0.85, 8, 15); // рисуем тор
```

Требуются некоторые пояснения. Кубик рисуется с отключенным источником света для того, чтобы он не получился таким же, как тор, серым.

Большинство параметров источника света задается с помощью команды glLightfv. Первый параметр команды — идентификатор источника, второй аргумент — символическая константа, указывающая, какой атрибут устанавливается, последним аргументом задается ссылка на структуру, содержащую задаваемые значения.

Как видно из кода, для задания позиции символическая константа должна быть GL_POSITION, указываемый массив определяет позицию источника света в текущей системе координат. Источник света, подобно невидимому объекту, располагается в заданной точке и не перемещается вслед за системой координат. После того как его позиция зафиксирована, трансформации системы координат не приведут к изменению положения источника.

В этом примере значения элементов массива не изменяются, а система координат перемешается с течением времени. Можно и наоборот — изменять значения элементов массива, а систему координат не трогать. Это сделано в следующем примере, проекте из подкаталога Ex02, где положение источника света задается значениями элемента массива LightPos, изменяющимися с течением времени:

```
With frmGL do begin
   LightPos[0] := LightPos[0] + Delta;
   If LightPos[0] > 15.0
      then Delta := -1.0
      else If (LightPosfO] < -15.0) then
           Delta := 1.0;
   InvalidateRect(Handle, nil, False;;
end;
```

Перерисовка кадра начинается с того, что задается текущее положение источника света:

```
glLightfv(GL_LIGHT0, GL_POSITION, @LightPos);
glCallList(Sphere);
```

i3 примере источник света колеблется над поверхностью сферы.

Теперь настала пора разобрать параметры источника света, связанные с оптическими характеристиками окружающей среды. Эти характеристики складываются из трех отражений: фоновое, диффузное и зеркальное.

Символьные константы GL_AMBIENT, GL_DIFFUSE и GL_SPECULAR, указанные в качестве второго аргумента команды glLight, позволяют задавать требуемые свойства окружающей среды.

Наиболее важными для нас пока являются первые две характеристики. Я приведу их упрощенное толкование, которого вам будет вполне достаточно для успешного использования соответствующих команд. Конечно, эти понятия имеют физическое обоснование, и неплохо было бы разобраться в нем, но пока можно ограничиться и поверхностным представлением.

Вес цвета в диффузной составляющей задает, насколько сильно этот цвет отражается поверхностью при ее освещении. И наоборот, вес цвета в фоновой составляющей задает, насколько сильно этот цвет поглощается поверхностью.

По умолчанию поверхность ничего не поглощает и все отражает.

Следующий пример, проект из подкаталога ExO3, является развитием предыдущего, в нем добавлена возможность задания значений для характеристик источника цвета.

Окно приложения снабжено всплывающим меню. По выбору пункта меню появляется диалог задания цвета; выбранный пользователем цвет устанавливается значением нужной характеристики источника света.

Здесь я воспользовался разобранной *во* второй главе процедурой, которая переводит системный цвет в диапазон, обычный для OpenGL:

```
procedure TfrmGL.Ambient2Click(Sender: TObject);
begin
  If ColorDialog1.Execute thc-n
   ColorToGL (ColorDialog1.Color, Ambient[0], Ambient [1], Ambient [2]);
end;
```

С помощью этого проекта можно подбирать свойства источника света для дальнейшего использования в других проектах. При выборе пункта Info появляется окно, в котором выводятся текущие характеристики источника света (рис. 4.1).

Пункт меню Reset позволяет вернуться к первоначальным установкам, соответствующим принятым в OpenGL по умолчанию.

Проведем простые эксперименты. Задайте фоновое отражение каким-либо чистым цветом, например красным. После этого наименее освещенные час-

ти поверхности сферы окрашиваются этим цветом. Если значения цветового фильтра, RGB, тоже установить красными, вся сфера окрасится равномерно.

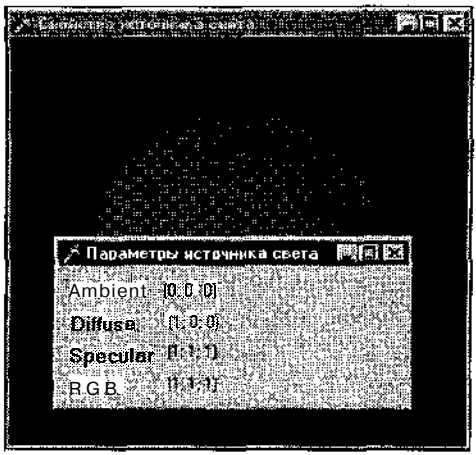


Рис. 4.1. В примере можно менять текущие установки источника света

Верните значения установок в первоначальные и установите значения диффузного отражения в тот же чистый цвет. Теперь в этот цвет окрашиваются участки поверхности сферы, наиболее сильно освещенные источником света. Слабо освещенные участки окрашены в оттенки серого. Если RGB установить в этот же цвет, насыщенность им увеличивается по всей поверхности.

Если обе характеристики задать равными, поверхность равномерно окрашивается ровным светлым оттенком.

Проект этот простой, но весьма интересный. Самые красивые результаты получаются при цветовом фильтре из смеси базовых цветов и отражениях, заданных в чистые цвета.

Свойства материала

Помимо свойств материала в этом разделе мы также продолжим изучение свойств источника света,

Свойства материала задаются с помощью команды `glMaterial`. Характеристики, определяющие оптические свойства материала, и соответствующие им символьные константы являются следующими: рассеянный цвет (`GL_AMBIENT`), диффузный цвет (`GL_DIFFUSE`), зеркальный цвет (`GL_SPECULAR`), излучаемый цвет (`GL_EMISSION`), степень зеркального отражения (`GL_SHININESS`).

Значением последнего параметра может быть число из интервала [0,128], остальные параметры представляют собой массивы четырех вещественных чисел.

Зеркальный цвет задает цветовую гамму бликов материала, степень зеркального отражения определяет, насколько близка поверхность к идеальному зеркалу.

Поработайте с проектом из подкаталога Ex04 и выясните смысл всех характеристик материала "своими глазами".

Этот пример является продолжением предыдущего, в нем добавились пункты меню, соответствующие характеристикам материала. Важно заметить: чтобы цветовой фильтр не перебивал задаваемые свойства, в примере не включается режим GL_COLOR_MATERIAL.

Начните знакомство с примером с вариации значения степени зеркального отражения, которое определяет цветовую насыщенность блика от источника света. С помощью клавиши 'S' можно регулировать размер блика на сфере, нажимая ее одновременно с <Shift> — увеличить, а без <Shift> — уменьшить эти размеры.

Замечание

Чтобы блик появился на сфере, установите зеркальный цвет в значение, отличное от принятого по умолчанию.

Следующий пример, проект из подкаталога Ex05, знакомит нас с еще одним аспектом, связанным с источником света — с моделью освещения.

Модель освещения задается с помощью команды glLightModel.

В примере внутренняя и внешняя стороны цилиндра имеют разные свойства материала, внутренняя сторона окрашена синим, внешняя — красным. Свойства материала заданы в массивах ambFront и ambBack.

Для того чтобы включить освещенность для внутренней стороны многоугольников, вызывается команда glLightModel, у которой вторым аргументом задается символическая константа GL_LIGHT_MODEL_TWO_SIDE, а третьим аргументом — ноль или единица:

```
glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, 1 ) ; // для обеих сторон
glMaterialfv (GL_FRONT, GL_AMBIENT_AND_DIFFUSE, @ambFront); /7 задняя
                                                            // сторона
gIMaterialfv (GL_BACK, GL_AMBIENT_AND_DIFFUSE, @ambBack); // передняя
                                                          // сторона
```

Я советую вам не спеша разобраться с этим упражнением. Например, посмотрите результат, который получается при замене константы GL_AMBIENT_AND_DIFFUSE на GL_DIFFUSE и GL_AMBIENT. Вы увидите, что диффузный цвет материала наиболее сильно определяет цветовые характеристики поверхности.
Следующий пример, проект из подкаталога Ex06, совсем простой — на экране вращается объемная деталь, построенная на основе тестовой фигуры второй главы (рис. 4.2).

Рис. 4.2. Теперь тестовая деталь выглядит более реалистично

В этом примере интересно то, как при некоторых положениях детали ее грани отсвечивают слабым фиолетовым оттенком.

Массив свойств материала детали содержит небольшую составляющую красного, чем больше это число (до некоторых пределов), тем выразительнее будет выглядеть деталь:

```
MaterialColor: Array [0..3] of GLfloat = (0.1, 0.0, 1.0, 1.0);
```

Как правило, по ходу работы приложения требуется менять текущие свойства материала, как это делается в проекте из подкаталога Ex07. На экране вращается кубик, "проткнутый" тремя цилиндрами (рис. 4.3).

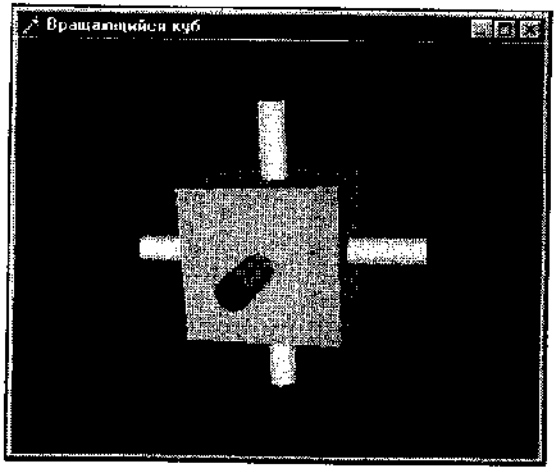


Рис. 4.3. Объекты сцены имеют различные свойства материала

В программе заданы два массива, определяющие различные цветовые гаммы. Перед воспроизведением элемента задаются свойства материала:

```
const
  MaterialCyan : Array[0..3] of GLfloat = (0.0, 1.0, 1.0, 1.0);
  MaterialYellow : Array[0..3] of GLfloat = (1.0, 1.0, 0.0, 1.0);
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, @MaterialCyan);
  glutSolidCube (2.0); // зеленоватый кубик
  glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, @MaterialYellow);
  // три желтых цилиндра
  glTranslatef (0.0, 0.0, -2.0);
  gluCylinder (qObj, 0.2, 0.2, 4.0, 10, 10);
  glRotatef (90, 1.0, 0.0, 0.0);
  glTranslatef (0.0, 2.0, -2.0);
  gluCylinder (qObj, 0.2, 0.2, 4.0, 10, 10);
  glTranslatef (-2.0, 0.0, 2.0);
  glRotatef (90, 0.0, 1.0, 0.0);
  gluCylinder (qObj, 0.2, 0.2, 4.0, 10, 10);
```

Теперь нам необходимо вернуться немного назад и обсудить положение источника света, задаваемое массивом четырех вещественных чисел. Если последнее число равно нулю, то, согласно документации, свет рассматривается как направленный источник, а диффузное и зеркальное освещение рассчитываются в зависимости от направления на источник, но не от его действительного положения, и ослабление заблокировано.

Посмотрим на практике, что это значит. Примером будет служить проект из подкаталога Ex08: на экране рисуется два четырехугольника, один покрыт равномерно серым, на поверхности второго видны блики (рис. 4.4).

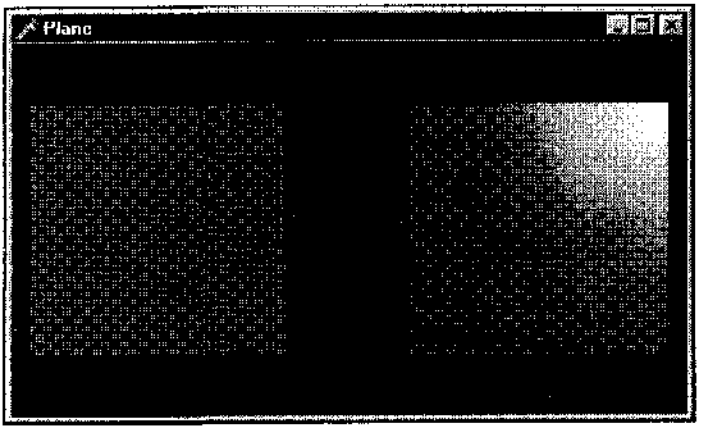


Рис. 4.4. Для прямоугольника, расположенного слева, источник света находится в бесконечности

Позиция источника света, освещающего первый четырехугольник, задана точно такой же, что и для второго квадрата, но значение четвертого компо-

нента равно нулю, источник света расположен где-то в бесконечности, и только направление на него влияет на оттенок серого, которым равномерно окрашивается плоскость.

Далее в нашей программе стоит проект из подкаталога Ex09. Этот пример, представляющий двенадцать сфер из разного материала, обычно сопровождает любой курс по OpenGL (рис. 4.5).

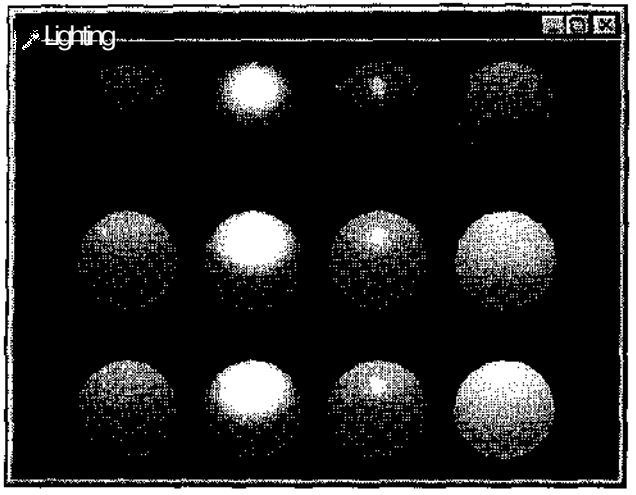


Рис. 4.5. Классический пример, иллюстрирующий свойства материала

Стоит сразу же обратить внимание, что при отключенном режиме GL_COLOR_MATERIAL цветовой фильтр никак не влияет на освещенность. Для регулирования ее гаммы используется команда задания модели освещения со вторым аргументом, равным GL_LIGHT_MODEL_AMBIENT, при этом последний аргумент, массив четырех вещественных чисел, задает цветовую палитру:

```
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, @lmodel_ambient);
```

Сфера в первом ряду и первой колонке нарисована с отключенными фоновой и зеркальной составляющими материала. Первый ряд, вторая колонка — диффузное и зеркальное освещение, следующая сфера более блестящая. Последняя сфера первого ряда — включены диффузия и эмиссия.

Фоновая и диффузная составляющие включены для первой сферы второй линии, нет зеркальной составляющей. Вторая и третья сферы — все включено, кроме эмиссии, различаются размером блика. У последней сферы второго ряда отсутствует зеркальная составляющая.

Сферы последней линии отличаются усилениостыо цветовой насыщенности составляющих свойств материала.

Еще один классический пример, проект из подкаталога Ex10: двадцать чайников из различного материала (рис. 4.6).

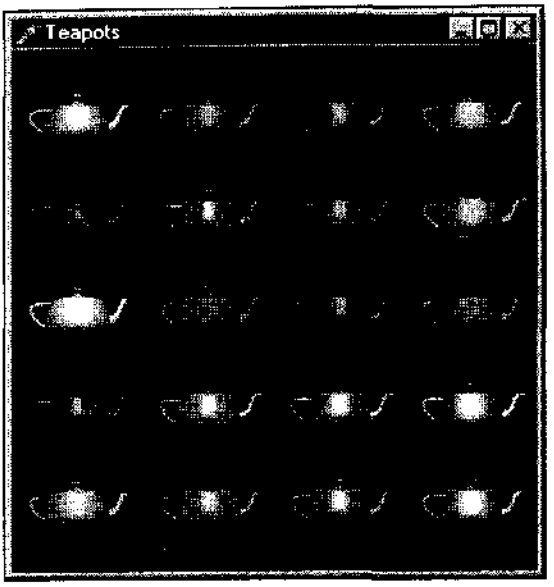


Рис. 4.6. Новичков такие картинки обычно впечатляют

Не будем рассматривать программу подробно, мы не встретим в ней ничего нового, однако пример этот не бесполезный — подобранный профессионалами набор материалов может вам пригодиться. В примерах пятой и шестой глав я использовал некоторые из этих материалов.

Поскольку чайники появляются на экране сравнительно медленно, в этом примере не используется двойная буферизация.

Дальше мы рассмотрим серию моих переложений на Delphi программ из набора примеров OpenGL SDK и закрепим наши знания по теме этого раздела.

Начнем с проекта из подкаталога Ex11, на котором основаны и несколько следующих примеров. На экране располагаются конус, тор и сфера (рис. 4.7).

Все параметры, за исключением позиции источника света, задаются по умолчанию.

Последнее число (w-компонент) в массиве, связанном с позицией источника света, равно нулю, источник света располагается на бесконечности.

Здесь все просто, можем перейти к следующему примеру, проекту из подкаталога Ex12. Объекты сцены окрашены в фиолетовый цвет, добавилась строка, задающая диффузную составляющую источника света:

```
glLightfv(GL_LIGHT0, GL_DIFFUSE, @light_diffuse) ;
```

В проекте из подкаталога Ex13 — другая картина, возникшая благодаря тому, что задается фоновое отражение, отличное от принятого по умолчанию:

```
glLightfv(GL_LIGHT0, GL_AMBIENT, @light_ambient);
```

Рис. 4.7. Эту композицию будем использовать в качестве тестовой

В последнем примере этой серии, проекте из подкаталога Ex14, модель тонирования задается значением, отличным от принятого по умолчанию:

```
glShadeModel (GL_FLAT);
```

На рис. 4.8 показан получающийся результат — фигуры композиции потеряли гладкость своих форм.

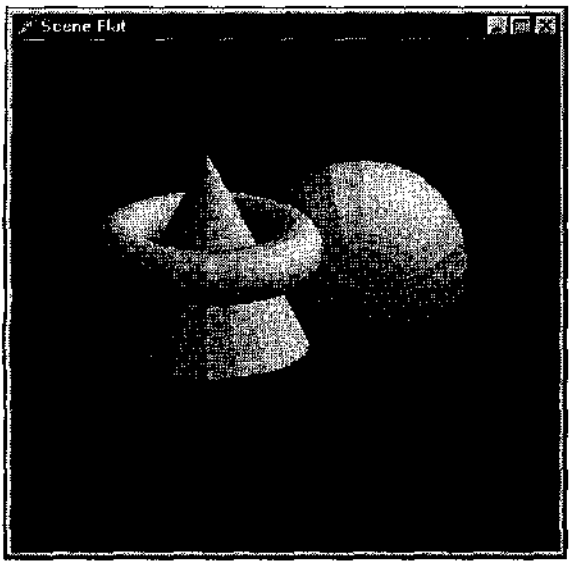


Рис. 4.8. Команда glShadeModel может существенно повлиять на получающиеся образы

Стоит сказать, что манипулирование свойствами источника света и оптическими свойствами материалов позволяет достичь весьма впечатляющих визульных эффектов, поэтому давайте изучим данную тему максимально подробно.

В следующем простом примере, проекте из подкаталога Ex15, рисуется сфера с красивым бликом на поверхности. У источника света задается позиция, остальные параметры — по умолчанию. Материал определяется диффузной и зеркальной составляющими, размер блика описывается следующей строкой:

```
glMaterialf(GL_FRONT, GL_SHININESS, 25.0);
```

В этом примере есть то, чего мы ранее не встречали: здесь включается режим коррекции цвета материала и вызывается новая для нас команда:

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
```

Команда `glColorMaterial` задает, какая из характеристик материала корректируется текущим цветом. Фактически это парная команда к режиму `GL_COLOR_MATERIAL`, однако в предыдущих примерах мы использовали значение, принятое по умолчанию.

Замечание

С точки зрения оптимизации эта команда предпочтительнее команды `glMaterial`.

В примере первоначальные значения элементов массива `diffuseMaterial` определяют диффузные свойства материала, по нажатию клавши 'R', 'G' и 'B' увеличивается вес красного, зеленого или синего в текущем цвете:

```
procedure changeRedDiffuse;
begin
    diffuseMaterial[0] := diffuseMaterial[0] + 0.1;
    if diffuseMaterial[0] > 1.0
       then diffuseMaterial[0] := 0.0;
    glColor4fv(@diffuseMaterial);
end;
```

Проект из подкаталога Ex16 представляет собой еще одну модель планетной системы: вокруг звезды вращается планета, вокруг которой вращается спутник (рис. 4.9).

Источник света располагается внутри звезды, чем объясняется эффектность освещения планеты и спутника, тени и блики на их поверхностях в точности соответствуют положению в пространстве относительно звезды.

Не пропустите важный момент: для сферы, моделирующей звезду, задается свойство материала, соответствующее излучающей составляющей материала:

```
const
  sColor: array [0..3] of GLfloat = (1, 0.75, 0, 1);
  black: array [0..3] of GLfloat = (0, 0, 0, 1);
  ...
  glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, @sColor);//излучение света
  glutSolidSphere(0.8, 32, 16);     // солнце
  glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, @black); // отключить
```

Благодаря этому наша звезда действительно светится.

На рис. 4.10 представлен результат работы следующего примера, проекта из подкаталога Ex17.

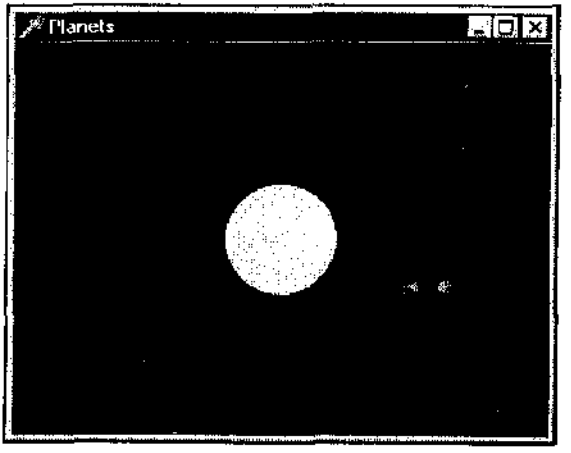


Рис. 4.9. Наши астрономические модели становятся все более совершенными

Рис. 4.10. Объекты сцены освещаются источниками различной фоновой интенсивности

Пример посвящен фоновой интенсивности света. Во-первых, задается полная интенсивность, как свойство источника света:

```
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, @global_ambient);
```

Перед воспроизведением каждого чайника задается свое, локальное значение фоновой составляющей материала. Это значение минимально для верхнего объекта и максимально для нижнего.

В примере встречаем новую для нас команду:

```
glFrontFace (GL_CW);
```

В данном случае она никак не влияет на работу приложения, но дает нам повод подробнее изучить то, как задаются передняя и задняя стороны многоугольников (в предыдущих главах мы уже об этом немного говорили).

По умолчанию обход вершин против часовой стрелки задает лицевую сторону многоугольника. Вызов команды с аргументом GL_CW меняет порядок обхода на противоположный.

В проекте из подкаталога Ex18 с течением времени вращается квадрат, передняя сторона которого окрашена красным, задняя — синим. Если вершины перечислять в обратном порядке, квадрат развернется к наблюдателю задней стороной. То же самое произойдет, если нажать третью цифровую клавишу, в обработчике нажатия которой вызывается команда glFrontFace с аргументом GL_CW:

```
If Key = 49 then glEnable  (GL_CULL_FACE);   // нажата  '1'
If Key = 50 then glDisable  (GL_CULL_FACE);  // нажата  '2'
If Key = 51 then glFrontFace  (GL_CCW);      // нажата  '3'
If Key = 52 then glFrontFace  (GL_CW);       // нажата  '4'
If Key = 53 then glCullFace  (GL_FRONT);     // нажата  '5'
If Key = 54 then glCullFace  (GL_BACK);      // нажата  '6'
```

При нажатии клавиши '1' включается режим отсечения, задние стороны многоугольников не рисуются. Клавиша '5' меняет правило отсечения, при включенном режиме отсечения не будут рисоваться передние стороны полигонов. Остальные клавиши позволяют вернуть режимы в значения, принятые по умолчанию.

Обратите внимание, что при переворотах квадрата нормаль необходимо разворачивать самому, иначе квадрат выглядит чересчур тускло.

Код для этого можно скорректировать так:

```
If Key = 51 then begin
            glFrontFace (GL_CCW);
            glNormal3f (0.0, 0.0, 1.0);
end;
```

```
If Key = 52 then begin
            glFrontFace (GL_CW);
            glNormal3f (0.0, 0.0, -1.0);
end;
```

Переходим к следующему примеру, проекту из подкаталога Ex19. Здесь мы помимо того, что закрепим тему этого раздела, вспомним, как производить отсечение в пространстве.

На экране нарисованы три чайника, у каждого из них небольшая часть отсечена так, что можно заглянуть внутрь объекта (рис. 4.11).

Рис. 4.11. Можно заглянуть внутрь чайников

Вырезка осуществляется способом, знакомым нам по предыдущей главе:

```
glClipPlane (GL_CLIP_PLANE0, @eqn);
glEnable (GL_CLIP_PLANE0);
```

Верхний объект нарисован без дополнительных манипуляций и его внутренности выглядят невзрачно.

Перед воспроизведением второго чайника включается режим расчета освещенности для обеих сторон поверхности. Материал для этих сторон задается одинаковым:

```
glLightModelf (GL_LIGHT_MODEL_TWO_SIDE, 1);
glMaterialfv (GL_FRONT_AND_BACK, GL_DIFFUSE, @mat_diffuse);
```

Последний чайник отличается от предыдущего тем, что его внутренняя и наружная поверхности покрыты разными материалами:

```
glMaterialfv (GL_FRONT, GL_DIFFUSE, @mat_diffuse);
glMaterialfv (GL_BACK, GL_DIFFUSE, @back_diffuse);
```

Замечание

Для того чтобы увидеть внутренности объекта, необходимо следить, чтобы режим GL_CULL_FACE не был включен, иначе внутренние стороны многоугольников рисоваться не будут.

В этом примере уже нельзя пренебречь вызовом glFrontFace, иначе второй и третий чайники будут вывернуты наизнанку.

Проект из подкаталога Ex20 является модификацией предыдущего примера, чайник здесь уже вращается и выглядит еще более живописным (рис. 4.15).

Рис. 4.12. Чайник снаружи изумрудный, внутри — золотой. Такое увидишь только на экране монитора

Материалы для внутренней и внешней поверхностей чайника я взял из примера с двадцатью чайниками, приведенного выше.

Теперь мы изучили все, что необходимо для знакомства с классической программой, переложение которой на Delphi я поместил в подкаталог Ex21. Программа рисует группу из трех вращающихся шестеренок (рис. 4.13).

С помощью клавиш управления курсором и клавиши 'Z' можно изменять положение точки зрения в пространстве.

Для закрепления материала рассмотрим еще несколько примеров. Вы спокойно можете пропустить эту часть, если чувствуете, что предыдущих примеров оказалось достаточно для прочного овладения приемами работы с материалами.

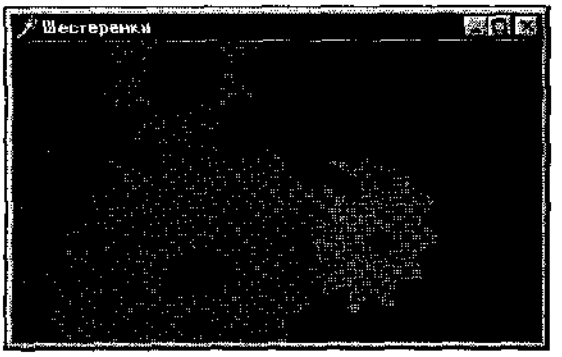


Рис. 4.13. Один из моментов работы проекта Gears

Рис. 4.14 иллюстрирует работу программы— проекта из подкаталога Ex22. в которой происходят колебания и вращения системы из трех объектов.

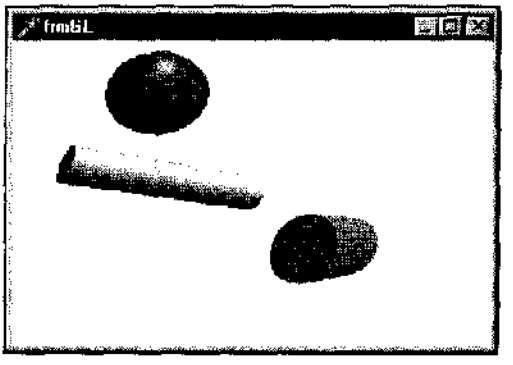


Рис. 4.14. Пример на колебания, объем фигур изменяется с помощью операции масштабирования

Первоначально объекты только колеблются, клавиша <Insert> позволяет управлять процессом вращения.

Для создания объектов системы используются функции библиотеки glu, quadric-объекты создаются перед описанием списков, после чего сразу же удаляются.

Обратите внимание, что для колебательного изменения объема фигур используется масштабирование, задаваемое перед вызовом списков, в таких случаях нельзя забывать о включении режима пересчета нормалей:

```
glEnable(GL_NORMALIZE);
```

Проект из подкаталога Ex23 является продолжением предыдущего примера, однако колебания системы здесь затухающие.

Еще один несложный пример на свойства материала — проект из подкаталога Ex24 переносит все дальше в глубины космоса (рис. 4.15).

В этом примере все объекты сцены являются quadric-объектами.

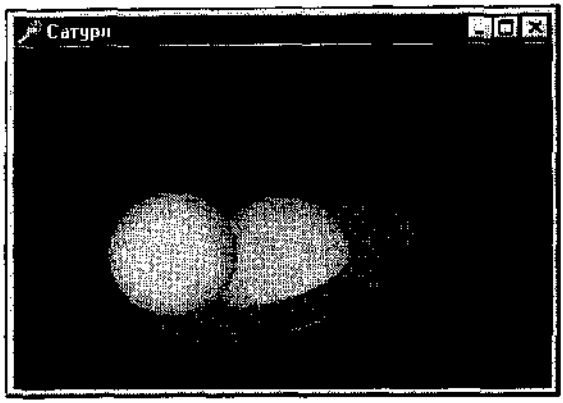


Рис. 4.15. Пора признаться: я в детстве мечтал стать космонавтом

Вывод на палитру в 256 цветов

В этом разделе я смогу вернуть некоторые свои долги, накопившиеся в предыдущих главах.

Ранее я обещал показать, почему наши самые первые примеры, где для упрощения не использовались операции с матрицей проекции, не годятся в качестве шаблона проектов.

Одну из таких программ я взял в качестве основы проекта из подкаталога Ex25, но вместо кубика на сцену поместил чайник.

Если при задании параметров вида не использовать ставшей привычной для нас последовательности операций, чайник выглядит на экране совсем невзрачным, будто вывернутым наизнанку.

Специально для этого случая приходится задавать модель освещенности с расчетом освещенности для обеих сторон многоугольников. Но и после этого тестовая фигура выглядит не очень эффектно.

Для тренировки я вставил строки с обычной последовательностью действий с матрицами проекций и матрицей моделирования, они первоначально закомментированы. Если вы уберете знаки комментария и удалите строки первого варианта, то тестовая фигура будет выглядеть на порядок эффектнее без каких-то дополнительных манипуляций;

```
procedure TfrmGL.FormResize(Sender: TObject);
begin
 glViewport(0, 0, ClientWidth, ClientHeight);

 // один вариант
 {----------}
 glLoadIdentity;
 glFrustum (-1, 1, -1, 1, 5, 10);
```

```
 glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, 1);
 {-------------}

 // второй вариант
 {-------------},
{ glMatrixMode(GL_PROJECTION);
  glLoadIdentity;
  glFrustum (-1, 1, -1, 1, 5, 10);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity;}
  {-----------}

 glTranslatef(0.0, 0.0, -8.0);   // перенос объекта - ось Z
 glRotatef(30.0, 1.0, 0.0, 0.0); // поворот объекта - ось X
 glRotatef(70.0, 0.0, 1.0, 0.0); // поворот объекта - ось Y

 InvalidateRect(Handle, nil, False);
end;
```

Надеюсь, это вас убедило в том, что подход, используемый в первых примерах, не годится для проектов с повышенными требованиями к качеству изображения.

Теперь по поводу палитры в 256 цветов. Все наши приложения не могут осуществлять корректный вывод на такую палитру, если текущие настройки экрана заданы с таким значением: экран приложения, использующего OpenGL, залит черным, а вместо объектов сцены выводятся чаще всего малопонятные скопления точек.

Предполагая, что такая палитра сегодня мало используется, я не стал корректировать палитру в каждом проекте, ограничусь единственным примером на эту тему.

Замечание

Если вы собираетесь распространять свои приложения, то я вам рекомендую позаботиться и о тех пользователях, которые по каким-либо причинам используют только 256 цветов. Качество воспроизведения, конечно, при этом страдает. но это все же лучше, чем ничего.

Проект из подкаталога Ex26 иллюстрирует, какие действия необходимо предпринять, если формат пиксела содержит установленный флаг, указывающий на необходимость корректировки палитры (PFD_NEED_PALETTE).

Код я снабдил комментариями, с помощью которых вы при наличии желания сможете в нем разобраться. Не будем подробно рассматривать производимые манипуляции, поскольку эта тема не относится непосредственно к библиотеке OpenGL. Замечу только, что обработку сообщений, связанных с изменением системной палитры, в принципе можно и не производить, делается это для корректности работы приложения.

Подробнее о поверхностях произвольной формы

В предыдущей главе уже были рассмотрены несколько примеров построения поверхности произвольной формы. В этом разделе мы продолжим этот разговор.

На рис. 4.16 представлен результат работы программы — проекта Ex22, моей модификации широко известной программы isosurf.c.

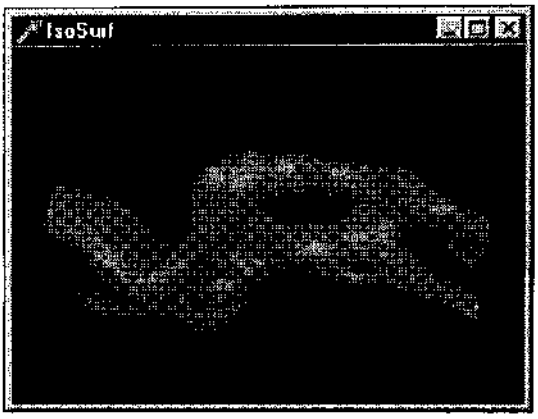


Рис. 4.16. Для построения поверхности используется более семи тысяч точек, поверхность строится как группа связанных треугольников

Программа рисует бесформенный полый объект по точкам, считываемым из текстового файла. Каждая строка файла содержит координаты вершины и нормали к ней.

Поверхность строится как единая группа связанных треугольников:

```
procedure   DrawSurface;
var
 i: GLuint;
begin
 glBegin( GL_TRIANGLE_STRIP);
  For i := 0 to numverts — 1 do begin
   glNormal3fv( @norms[i]);
   glVertex3fv( @verts[i]);
  end;
 glEnd;
end;
```

Клавишами управления курсором можно вращать фигуру, есть режим тестирования скорости воспроизведения, активизируемый нажатием клавиши T: фигура делает полный оборот по оси X.

Программа, несмотря на обширность кода, проста и не должна вызвать особых затруднений при разборе. Стоит только обратить внимание, что на сцене присутствует два источника света, различающиеся позицией.

Этот пример демонстрирует, что, в принципе, для воспроизведения поверхностей сложной формы не обязательно пользоваться сплайнами, а можно разбить поверхность на множество простых примитивов.

Следующий пример окажется очень полезным для тех, кто настроен на профессиональную работу с OpenGL.

В проекте из подкаталога Ex24 координаты вершин треугольников, образующих поверхность, считываются из файла формата `dxf` (рис. 4.17),

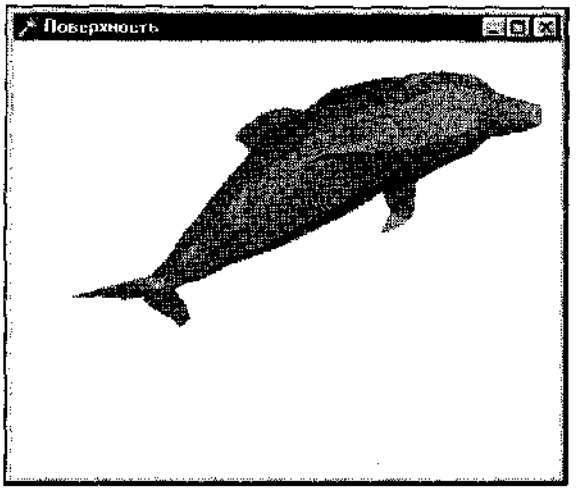


Рис. 4.17. Для построения поверхности используется файл формата `dxf`

Теперь вы можете подготавливать модели с помощью любого профессионального редактора — формат dxf является открытым и стандартным.

Замечание

Саму поверхность не обязательно строить по отдельным треугольникам, главное, чтобы используемый редактор мог разбивать поверхность на такие треугольники при экспорте в dxf-формат.

Я соглашусь с вами, что некоторые примеры, хоть и являются очень интересными,, годятся только для того, чтобы пару раз ими полюбоваться. Этот же пример является действительно полезным и достоин подробного разбора.

Программа является универсальной, в частности, в ней не ограничивается число точек поверхности.

В таких случаях можно использовать динамические массивы или, как в этом примере, списки (в терминах Delphi).

Списки введены в программе для хранения точек модели и нормалей к каждому треугольнику:

```
Model, Normals : TList;
```

Следующий тип введен для хранения координат отдельной точки:

```
type
  Vector = record
    x, y, z : GLfloat;
  end;
```

Одной из ключевых процедур примера является процедура чтения данных из файла формата dxf:

```
{$WARNINGS OFF} // отключить предупреждения компилятора о возможной
                // неинициализации переменных
procedure TfrmGL.LoadDXF (st : String);
var
  f : TextFile;
  wrkString : String;
  group, err : GLint;
  x1, x2, y1, y2, z1, z2, x3, y3, z3 : GLfloat;
// вспомогательная процедура добавления вектора в список Model
procedure AddToList (x, y, z : GLfloat);
var
  wrkVector : Vector;   // рабочая переменная, вектор
  pwrkVector ; ^Vector; // указатель на вектор
begin
  wrkVector.x := x;     // заполняем поля вектора
  wrkVector.y := y;
  wrkVector.z := z;
  New (pwrkVector);     // выделение памяти для нового элемента списка
  pwrkVector^ := wrkVector; // задаем указатель
  Model.Add (pwrkVector);   // собственно добавление вектора в список
end;
begin
  AssignFile(f,st); // открываем файл
  Reset(f);
  repeat /./ пропускаем файл до секции объектов "ENTITIES"
    ReadLn(f, wrkString);
  until (wrkString = 'ENTITIES') or eof(f);
  While not eof (f) do begin
      ReadLn (f, group);     // маркер
      ReadLn (f, wrkString); // идентификатор либо координата
      case group of
         0: begin              // начался следующий объект
            AddToList (x3, y3, z3); // добавляем в список треугольник
```

```
            AddToList (x2, y2, z2);
            AddToList (x1, y1, z1);
            end;
         10: val(wrkString, x1, err); // считываем вершины треугольника
         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}
```

После того как считаны координаты вершин треугольников, образующих поверхность, необходимо рассчитать векторы нормалей к каждому треугольнику:

```
{$HINTS OFF} // отключаем замечания компилятора с неиспользовании
             // переменной pwrkVector
procedure TfrmGL.CalcNormals;
var
  i : Integer;
  wrki, vx1, vy1, vz1, vx2, vy2, vz2 : GLfloat;
  nx, ny, nz : GLfloat;
  wrkVector : Vector;
  pwrkVector : "Vector;
  wrkVector1, wrkVector2, wrkVector3 : Vector;
  pwrkVector1, pwrkVector2, pwrkVector3 : ^Vector;
begin
  New (pwrkVector1); // выделение памяти под указатели
  New (pwrkVector2);
  Mew (pwrkVector3);

  For i := 0 to round (Model.Count / 3) - ") do begin
     pwrkVector1 := Model [i * 3]; // считываем по три вершины из списка
     wrkVector] := pwrkVector1^;   // модели
     pwrkVector2 := Model [i * 3 + 1];
     wrkVector2:= pwrkVector2^;
     pwrkVector3 := Model [i * 3 + 2];
     wrkVector3 := pwrkVector3^;

     // приращения координат вершин по осям
     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-перпендикуляр к центру треугольника
      nx := vy1 * vz2 - vz1 * vy2;
      ny := vz1 * vx2 - vx1 * vz2;
      nz := vx1 * vy2 - vy1 * vx2;

      // получаем унитарный 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);         // указатель на очередную нормаль
      pwrkVector^ := wrkVector;

      Normals.Add (pwrkVector); // добавляем нормаль в список Normals
    end;
  end;
  {$HINTS ON}
```

Собственно поверхность описывается в дисплейном списке:

```
Model := TList.Create;    // создаем список модели
Normals := TList.Create; // создаем список нормалей
LoadDxf ('Dolphin.dxf'); // считываем модель
CalcNormals;             // расчет нормалей
glNewList (SURFACE, GL_COMPILE); // поверхность хранится в списке
 For i := 0 to round (Model.Count / 3) - 1 do begin // по три вершины
  glBegin(GL_TRIANGLES);
    glNormal3fv(Normals.Items [i]); // задаем текущую нормаль
    glVertex3fv(Model.Items [i * 3]); // вершины треугольника
    glVertex3fv(Model.Items [i * 3 + 1]);
    glVertex3fv(Model.Items [i * 3 + 2]);
  glEnd;
 end;
glEndList;
Model.Free;          // списки больше не нужны, удаляем их
Normals.Free;
```

Замечание

Я не мог привести этот пример в предыдущей главе, поскольку в этом примере необходимо задать модель освещения с расчетом обеих сторон многоугольников: у нас нет гарантии, что все треугольники поверхности повернуты к наблюдателю лицевой стороной.

В программе все треугольники раскрашены одним цветом, можно добавить еще один список, аналогичный списку нормалей, но хранящий данные о цвете каждого отдельного треугольника.

Замечание

Обратите внимание, что в этом примере я меняю скорость поворота модели в зависимости от того, как быстро пользователь перемещает указатель при нажатой кнопке мыши.

Использование патчей

Одним из главных достоинств сплайнов является то, что поверхность задается небольшим количеством опорных точек — весьма экономный подход.

Замечание

Возможно, к достоинствам можно отнести и то, что нет необходимости самостоятельно рассчитывать нормали к поверхности, как это делалось в предыдущем примере.

Вспомним пример предыдущей главы с холмообразной поверхностью. Для хранения этой поверхности достаточно запомнить шестнадцать точек, если же ее разбивать по треугольникам, то шестнадцати вершин окажется явно не достаточно для сохранения гладкости.

Подкаталог Ex29 содержит модификацию примера предыдущей главы на построение NURBS-поверхности. Отличает этот проект то, что в нем задаются свойства материала, так что поверхность выглядит гораздо эффектнее.

Также здесь добавилось то, что среди опорных точек одна выделяется и рисуется красным цветом. Такую точку можно перемещать в пространстве, нажимая на клавиши 'X', 'У, 'Z', а также на эти клавиши совместно с <Shift>. Клавишами управления курсором можно задавать, какая из опорных точек является выделенной.

Создавать кусочки поверхностей с помощью сплайнов — дело несложное, но если попытаться подобрать опорные точки для того, чтобы нарисовать одним сплайном поверхности подобные чайнику, то это окажется занятием крайне трудоемким. Если же поверхность имеет негладкие участки, как, например, область соприкосновения носика чайника с его корпусом, то традиционными сплайнами это нарисовать окажется вообще невозможно.

Имеется стандартный способ рисования поверхностей, сочетающий в себе достоинства использования сплайнов и лишенный его недостатков: поверхность разбивается на отдельные гладкие участки, для каждого из которых строится отдельный сплайн. Такие кусочки называются *патчами* (patch, заплатка).

В частности, модель чайника строится именно таким способом, по отдельным патчам, вы можете заглянуть в модуль для того, чтобы убедиться в этом.

На рис. 4.18 отдельные патчи раскрашены разными цветами.

Рис. 4.18. Модель чайника строится из отдельных кусочков

Следующий пример, проект из подкаталога Ex25, строит поверхность на основе патчей, опорные точки для которых считываются из текстового файла (рис. 4.19).

Программа написана мною, но автором модели являюсь не я, а Геннадий Обухов, адрес его страницы я указал в приложении 1.

Модель состоит из 444 патча размером 4x4, по шестнадцать точек. Для хранения данных о патчах введен пользовательский тип:

```
type
  TVector = record
    x, y, z : GLfloat;
  end;
  TPatch = Array [0..15] of TVector;
```

Модель так же, как и в предыдущем примере, хранится в списке. Список заполняется данными, считываемыми из текстового файла, каждая строка

которого содержит координаты очередной вершины. Эта программа тоже является в некотором роде универсальной: требование к файлу заключается в том, что точки должны записываться порциями по шестнадцать, каждая группа представляет собой опорные точки очередного патча.

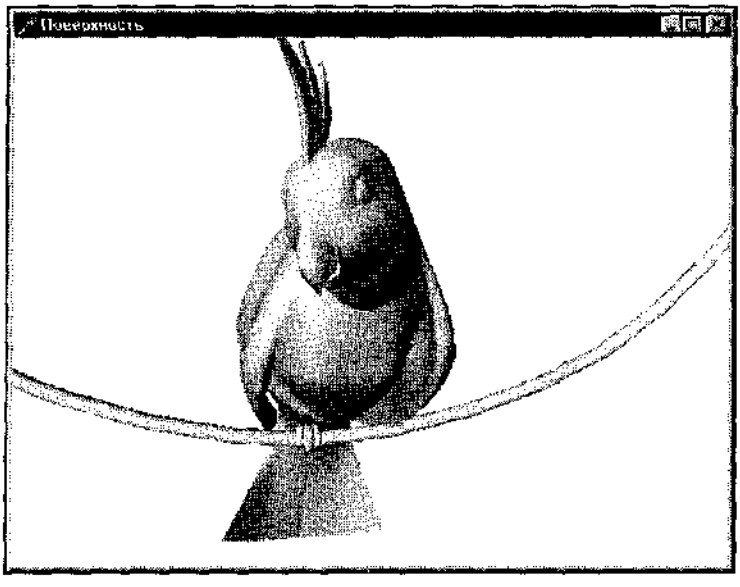


Рис. 4.19. Еще одна модель из отдельных кусочков

Я не могу поместить на дискету дополнительные примеры к этой программе, текстовые файлы слишком велики, поясню только, как создаются подобные файлы данных.

Я использовал свободно распространяемый модельер sPatch, позволяющий использовать встраиваемые модули (plug-in) для экспортирования моделей, созданных в нем из патчей, в произвольный формат. В частности, этот модельер позволяет записывать и в формате dxf.

Адрес, по которому можно получить sPatch, указан в приложении 1.

Ключевой процедурой разбираемого примера является процедура инициализации поверхности:

```
procedure TfrmGL.Init_Surface;
var
 f : TextFile;
 i : Integer;
 Model : TList;                  // список модели
 wrkPatch ; TPatch;              // вспомогательная переменная, патч
 pwrkPatch : ^TPatch;            // указатель на патч
```

```
begin
 Model := TList.Create;      // создание списка модели
 AssignFile (f, 'Parrot.txt'); // открытие файла
 ReSet <f);
 While not eof (f) do begin
   For i := 0 to 15 do // точки считываются по шестнадцать
      ReadLn (f,wrkPatch [i].x,wrkPatch [i].y, wrkPatch [i].z);
   New (pwrkPatch);     // выделение памяти под очередной элемент
   pwrkPatch^ := wrkPatch; // задаем указатель на элемент
   Model.Add (pwrkPatch);  // собственно добавление в список
 end;
 CloseFile (f);

 glNewList (SURFACE, GL_COMPILE);
   glPushMatrix; // матрица запоминается из-за масштабирования
   glScalef (2.5, 2.5, 2.5);
   For i := 0 to Model.Count — 1 do begin // цикл построения патчей
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4, 0, 1, 12, 4, Model.Items[i]);
    glEvalMesh2(GL_FILL, 0, 4, 0, 4);
   end;
   glPopMatrix;
 glEndList;
 Model.Free; // удаление списка
end;
```

Закончу рассмотрение примера двумя замечаниями:

- список модели заполняется порциями по шестнадцать точек, это обязательно для верной адресации в памяти отдельных патчей;
- файл опорных точек не годится для построения модели по отдельным многоугольникам, в этом случае она будет лишь отдаленно напоминать исходную поверхность.

Замечание

Несмотря на то, что при высоком уровне детализации разбиения поверхности на отдельные треугольники мы получим поверхность, не уступающую по качеству воспроизведения варианту со сплайнами, я настоятельно посоветую вам использовать все-таки поверхность Безье. Высокая детализация приведет к огромному количеству воспроизводимых примитивов, и скорость воспроизведения заметно упадет.

Приведу еще один прекрасный пример на использование патчей, проект из полкаталога Ex31 (рис. 4.20).

В примере используются патчи из 25 точек каждый, первые двенадцать "заплаток" предназначены для построения лепестков розы, четырнадцать следующих — для стебля цветка. При описании дисплейного списка последовательно задаем нужный цвет:

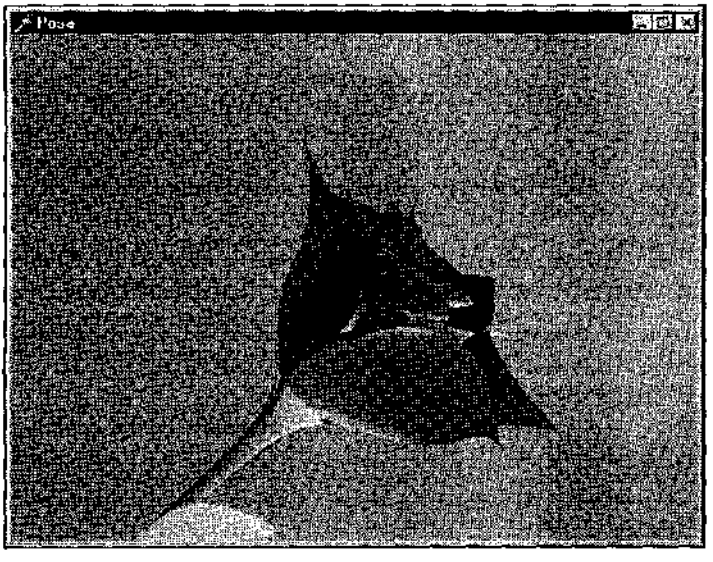


Рис. 4.20. Этот пример посвящается всем девушкам-программисткам

```
glNewList (ROZA, GL_COMPILE);
  glPushMatrix;
  glScalef (0.5, 0.5, 0.5);
  For i := 0 to 11 do begin // первые 12 патчей – лепестки
    glColor3f (1.0, 0.0, 0.0); // задаем цвет красным
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 5, 0, 1, 15, 5, Model.Items[i]);
    glEvalMesh2(GL_FILL, 0, 20, 0, 20);
  end;

  For i := 12 to Model.Count – 1 do begin // стебель цветка
    glColor3f (0.0, 1.0, 0.0); // цвет – зеленый
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 5, 0, 1, 15, 5, Model.Items[i]);
    glEvalMesh2(GL_FILL, 0, 20, 0, 20);
  end;

  glPopMatrix;
glEndList;
```

В последней главе книги будет еше один пример, где используются патчи. так что мы не прощаемся с этой темой насовсем.

Буфер трафарета

Многие специальные эффекты, самым простым из которых является вырез-ка объекта, основаны на использовании буфера трафарета (шаблона).

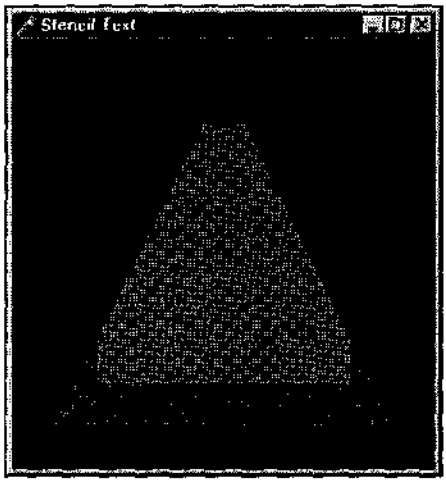
Замечание

Напомню, что одним из полей в формате пиксела является размер буфера трафарета, который надо задавать в соответствии с характеристиками вашей графической платы.

Выполнение теста трафарета разрешается командой `glEnable` с аргументом `GL_STENCIL_TEST`.

Команда `glStencilFunc` отвечает за сравнение, а команда `glStencilOp` позволяет определить действия, базирующиеся на результате проверки трафарета.

Другие команды, связанные с буфером трафарета, рассмотрим по ходу чения примеров, первым из которых станет проект из подкаталога Ex32, зультат работы которого представлен на рис. 4.21.



4.21. Простейший пример на операции с буфером трафарета

Инициализация работы приложения начинается с задания характеристик буфера трафарета:

```
// значение заполнения буфера трафарета при счистке
// число битов, задающее маску
        // включаем тест буфера трафарета
```

Первые две команды вызываются с аргументами, имеющими значение, принятое по умолчанию, поэтому могут быть удалены.

Замечание

Аргументы этих команд имеют смысл битовых масок, поэтому корректнее задавать их

Первая команда задает фоновое значение, которым будет заполнен буфер при выполнении команды с аргументом