СЕРГЕЙ ГАЙДУКОВ

bhv

OpenGL

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ ТРЕХМЕРНОЙ ГРАФИКИ НА C++

НОВЫЕ ВОЗМОЖНОСТИ OpenGL

СРЕДСТВА NVIDIA OpenGL SDK

ПРОГРАММИРОВАНИЕ ИГР

ЭКСПОРТ МОДЕЛЕЙ ИЗ 3ds max

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Сергей Гайдуков

OpenGL

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ ТРЕХМЕРНОЙ ГРАФИКИ НА C++

Санкт-Петербург
«БХВ-Петербург»
2004

УДК 681.3.068+800.92C++
ББК 32.973.26-018.1
Г12

Гайдуков С. А.

Г12 OpenGL. Профессиональное программирование трехмерной графики на C++. — СПб.: БХВ-Петербург, 2004. — 736 с.: ил.

ISBN 5-94157-363-4

Книга посвящена использованию новых возможностей графической библиотеки OpenGL версии выше 1.2 в приложениях, разрабатываемых на языке C++ в Microsoft Visual Studio .NET 2002. Описано применение средств NVIDIA OpenGL SDK для создания реалистичных трехмерных изображений. На примерах рассмотрены загрузка текстур из файлов форматов TGA и JPG, экспорт моделей из 3ds max, хранение данных в ZIP-архивах, отсечение невидимой геометрии, моделирование глянцевых объектов и др.

Прилагается компакт-диск с инструментальными средствами и демонстрационными версиями рассматриваемых примеров.

Для программистов

УДК 681.3.068+800.92C++
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Игорь Рыбинский*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Наталия Першакова*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.03.04.
Формат $70\times100^{1}/_{16}$. Печать офсетная. Усл. печ. л. 59,34.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02
от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в ФГУП ордена Трудового Красного Знамени "Техническая книга"
Министерства Российской Федерации по делам печати,
телерадиовещания и средств массовых коммуникаций.
190005, Санкт-Петербург, Измайловский пр., 29.

ISBN 5-94157-363-4

Содержание

Введение

Эта книга посвящена программированию компьютерной графики. Если точнее, она научит вас программировать с использованием расширений OpenGL в Microsoft Visual Studio.NET.

Базовая версия OpenGL 1.0 вышла в далеком 1992 году. С тех пор прошло более 10 лет. Это очень большой срок для компьютерной индустрии. Например, за это время операционные системы Windows прошли путь от 16-битной операционной оболочки Windows 3.11 до полноценной 32-битной операционной системы Windows XP. Тактовая частота процессоров Intel Pentium поднялась с 60 МГц до 3.2 ГГц, а на смену простеньким двумерным ускорителям ATI Mach32 корпорации ATI пришли "навороченные" ATI Radeon 9800 XT. По идее, за столь большой срок библиотека OpenGL должна была морально устареть, если бы не одно "но": создатели OpenGL заложили в нее очень важный принцип — расширяемость. Если производитель аппаратного обеспечения выпускал видеокарту, новые возможности которой не могли быть использованы в существующей версии OpenGL, то он мог выпустить свое расширение OpenGL. В результате, разработчики программного обеспечения могли использовать новые возможности этой видеокарты практически сразу после ее выхода, не дожидаясь выхода новой версии DirectX, как это случилось с видеокартой Radeon 9700 корпорации ATI. Другим классическим примером несбывшихся ожиданий могут служить печально известные пиксельные шейдеры GeForce2 GTS (GTS кстати расшифровывается, как Giga Texel Shader), когда компания Microsoft решила, что пиксельные шейдеры этого GPU таковыми не являются. В итоге, программа, использующая DirectX, не может задействовать все возможности GeForce2.

Любая следующая версия OpenGL отличается от предыдущей версии лишь списком обязательных расширений. Иными словами, OpenGL 1.4 можно описать, как OpenGL 1.0 + новые расширения версий 1.1, 1.2, 1.2.1, 1.3 и 1.4. Поэтому любая программа, написанная для ранних версий OpenGL, будет нормально работать с текущей версией OpenGL.

Однако использование расширений OpenGL связано с некоторыми проблемами. Самой большой трудностью является то, что компания Microsoft в настоящее время продвигает свой собственный API — DirectX. А поскольку OpenGL является прямым конкурентом DirectX, компания Microsoft не заинтересована в поддержке своими программными продуктами библиотеки OpenGL. В частности, в состав Microsoft Visual Studio .NET входят лишь за-

головочные файлы для OpenGL 1.1, причем сами файлы датируются 1996-м годом. С документацией дела обстоят аналогичным образом.

Из этого следует вывод, что для нормальной работы с расширениями OpenGL нам необходим дополнительный инструментарий. В качестве такого инструментария мы будем использовать NVIDIA SDK 5.21, который находится на CD-диске, сопровождающем книгу. В этот SDK входят NVIDIA Audio SDK, NVIDIA DirectX SDK и NVIDIA OpenGL SDK, причем, нас, разумеется, больше всего будет интересовать последний. NVIDIA OpenGL SDK содержит множество полезных вещей:

- набор заголовочных файлов для OpenGL версии 1.3;
- кроссплатформенную оконную библиотеку GLUT 3.7;
- библиотеку NVIDIA OpenGL Helper Library, которая фактически является объектно-ориентированной надстройкой над OpenGL и GLUT;
- математическую библиотеку NV_MATH;
- библиотеку утилит NV_UTIL, содержащую функции для работы с файлами форматов ZIP, ASE, JPG и TGA;
- множество других вспомогательных библиотек, облегчающих жизнь разработчику;
- множество примеров, демонстрирующих использование расширений OpenGL на практике;
- статьи и слайды докладов;
- многое другое.

Но у всего этого есть один недостаток: NVIDIA OpenGL SDK поставляется "как есть" — в исходных кодах и без какой-либо документации. Поэтому новичкам будет довольно сложно разобраться с этим продуктом.

Эту книгу я задумал как учебник, которого мне очень не хватало во время изучения NVIDIA OpenGL SDK и расширений OpenGL. Хотя книга является учебником и рассчитана на последовательное чтение, я решил разбить материал на тематические разделы. Книга не охватывает целиком ни NVIDIA OpenGL SDK, ни расширения OpenGL, т. к. эти темы очень объемные (одна только краткая спецификация расширений, поддерживаемых NVIDIA GeForce FX, занимает более 1000 страниц). Но, прочитав эту книгу, вы наверняка узнаете для себя очень много нового.

В этой книге основной упор делается на практические примеры. Из всех книг, которые я прочитал, наиболее полезными оказались те, которые содержали примеры и готовые решения. Книга содержит множество примеров, бóльшую часть которых я написал самостоятельно. При чтении этой книги постарайтесь просматривать эти примеры не очень бегло, иначе в какой-то момент времени вы рискуете потерять нить рассуждений.

На кого рассчитана эта книга

В первую очередь я адресую эту книгу студентам старших курсов и аспирантам первого года обучения, специализирующимся в области информатики и вычислительной техники. Книга будет полезна и профессионалам.

Книга рассчитана на читателей, имеющих опыт работы с Microsoft Visual C++ .NET (на уровне консольных приложений) и OpenGL 1.1. Очень желательно предварительно прочесть одну из книг, приведенных в списке использованной литературы ([1], [2], [3], [4], [5] или [6]). Кроме того, для понимания некоторых примеров понадобятся начальные знания Borland Delphi (на уровне первых 6-ти глав [7]).

Я принимаю нейтральную позицию в спорах на тему, что лучше: Visual C++ или Delphi. Каждый из этих языков имеет свои достоинства и недостатки, и программист просто должен уметь выбрать продукт, который лучше всего подходит для решения конкретной задачи. Visual C++ является идеальным инструментом для создания полноэкранных OpenGL-приложений, основным требованием к которым является быстродействие. Но в приложениях, где главную роль играет интерфейс, а вопросы производительности отходят на второй план, более рационально использовать Borland Delphi.

К книге прилагается CD-ROM, на котором вы найдете программное обеспечение, необходимое для работы с примерами из книги. Разумеется, вы не найдете на нем Microsoft Visual Studio .NET, Borland Delphi, 3D Studio MAX или какие-нибудь другие коммерческие продукты.

Структура книги

Книга состоит из двух частей. Первая часть посвящена вспомогательному инструментарию, без которого невозможно создать ни одну серьезную программу, использующую OpenGL, а вторая часть — собственно расширениям OpenGL.

Часть I. Использование NVIDIA OpenGL SDK

Главная задача первой части — познакомить читателя с основными компонентами NVIDIA OpenGL SDK и научить использовать их на практике.

Глава 1. Библиотека GLUT

Первая глава знакомит читателя с оконной библиотекой GLUT, позволяющей создавать кроссплатформенные OpenGL-приложения, не привязанные к определенной платформе. Даже если вы уже знакомы с этой библиотекой, я все равно советую вам бегло просмотреть эту главу, т. к. в ней приводятся

некоторые малоизвестные сведения о библиотеке GLUT, например, использование джойстика или создание полноэкранных приложений.

Глава 2. Библиотека GLH

Вторая глава посвящена одному из самых основных компонентов NVIDIA OpenGL SDK — библиотеке OpenGL Helper Library. Эта библиотека содержит множество классов, которые могут значительно облегчить жизнь программисту. Эти классы можно условно разделить на 3 большие группы:

- математические функции;
- объектно-ориентированная надстройка над GLUT, основанная на интеракторах;
- классы, инкапсулирующие функции OpenGL.

Кроме того, в этой главе рассматривается расширение библиотеки GLH — OpenGL Helper Library Extension (GLHE), добавляющее в библиотеку GLH ряд новых возможностей, в частности, интерактор консоли `glut_console` с поддержкой скриптов. Использование библиотеки GLHE позволяет значительно сократить размер демонстрационных программ: к примеру, программа, выводящая на экран цветной чайник, который пользователь может вращать и перемещать с использованием мыши и клавиатуры, менее 90 строк.

Глава 3. Библиотека NV_MATH

В третьей главе рассматривается математическая библиотека NV_MATH, являющаяся аналогом математической части библиотеки OpenGL Helper Library. Зачем она нужна? Если честно — не знаю. Скорее просто так исторически сложилось, что одни примеры из NVIDIA OpenGL SDK использовали библиотеку GLH_LINEAR, а другие — NV_MATH. В результате, для того, чтобы нормально ориентироваться в примерах NVIDIA OpenGL SDK, программисту очень желательно знать обе библиотеки. Поэтому если вы не собираетесь досконально разбирать примеры из состава NVIDIA OpenGL SDK, вы можете пропустить эту главу.

Глава 4. Библиотека NV_UTIL

В четвертой главе рассматривается библиотека NV_UTIL, которая содержит набор функций для работы с наиболее распространенными форматами файлов: TGA, JPG, ASE и ZIP. В принципе, этих форматов более чем достаточно для создания небольшой полноценной трехмерной игры. Кроме того, в этой главе рассматривается экспорт трехмерных сцен из 3D Studio MAX с использованием формата ASE при помощи библиотеки ASE Reader Library, являющейся объектной надстройкой над функциями NV_UTIL.

Часть II. Расширения OpenGL

Эта часть посвящена использованию новых расширений OpenGL версий 1.2 и выше, а также некоторых расширений, не вошедших в стандарт. Я решил не рассматривать расширения OpenGL версии 1.1, т. к. найти литературу на русском языке, посвященную OpenGL версий 1.0 и 1.1, не составит никакого труда.

Глава 5. Введение в расширения OpenGL

Пятая глава знакомит читателя с понятием расширения OpenGL и обучает использованию спецификации расширений. Кроме того, в этой главе рассматриваются семь относительно простых расширений OpenGL: EXT_separate_specular_color, ARB_window_pos, WGL_EXT_swap_control, SGIS_texture_lod, EXT_texture_lod_bias, EXT_texture_filter_anisotropic и SGIS_generate_mipmap. В заключение главы рассматривается инструментарий корпораций ATI и NVIDIA для быстрой инициализации расширений OpenGL: библиотеки ATI Extensions и NVIDIA OpenGL Helper Library.

Глава 6. Расширения EXT_texture_rectangle и NV_texture_rectangle

Шестая глава посвящена использованию текстур с разрешением, не кратным степени два.

Глава 7. Проверка видимости объектов с использованием расширений HP_occlusion_test и NV_occlusion_query

В седьмой главе рассматриваются расширения HP_occlusion_test и NV_occlusion_query, предназначенные для проверки видимости объектов на экране. Также рассматривается использование предварительной проверки видимости прямоугольных оболочек для отсечения невидимых объектов, что позволяет значительно повысить производительность приложения.

Глава 8. Использование внеэкранных буферов

Восьмая глава посвящена использованию внеэкранных буферов пикселов (pbuffer), представляющих собой специальную область видеопамяти, в которую программист может выводить изображение, как в обычное окно. При этом затрагиваются следующие расширения: WGL_ARB_pixel_format, WGL_ARB_pbuffer и ARB_render_texture. Кроме того, рассматривается практическое использование класса `PBuffer` из NVIDIA OpenGL SDK, значительно облегчающего работу с внеэкранными буферами pbuffer. В заключение главы создается мини-игра с видом от первого лица (FPS).

Глава 9. Сжатые текстуры

В девятой главе рассматривается использование сжатых текстур в OpenGL-программах, позволяющих сократить объем занимаемой видеопамяти, а также немного увеличить быстродействие программы. Кроме того, проводится исследование особенностей наиболее распространенного формата сжатия S3TC, а также его достоинства и недостатки. Примеры этой главы используют расширения ARB_texture_compression и EXT_texture_compression_s3tc. Кроме того, рассматриваются вопросы сохранения сжатых текстур на диске, а также использование формата DDS для хранения предварительно сжатых текстур с использованием формата S3TC.

Глава 10. Кубические текстурные карты

Десятая глава посвящена моделированию глянцевых объектов с использованием сферических и кубических текстурных карт. При этом подробно описываются все тонкости расширения ARB_texture_cube_map. Параллельно рассматриваются различные варианты хранения кубических текстур на диске, как с использованием классических графических форматов (TGA, JPG), так и с помощью формата DDS, позволяющего хранить в одном файле изображения всех граней кубической текстурной карты. Кроме того, затрагивается вопрос нетипичного использования кубических текстурных карт для моделирования закраски методом Фонга в реальном времени. В заключение рассматривается экспорт из 3D Studio MAX материалов, использующих текстурные карты отражения reflect/refract.

Требования к программному и аппаратному обеспечению

Для самостоятельного построения или модификации примеров программ, приведенных в книге, на вашем компьютере должно быть установлено следующее программное обеспечение.

- 32-разрядная операционная система семейства Windows, например Windows 2000 или XP.
- Microsoft Visual Studio .NET 2002.

Внимание

Часть примеров книги не компилируются с использованием Microsoft Visual Studio 6 или .NET 2003.

- Для сборки примеров, посвященных экспорту моделей из 3D Studio MAX, потребуется Discreet 3D Studio MAX 5.
- Для изменения разрешения текстур необходимо иметь Adobe Photoshop, ACDSee, IrfanView или аналогичную программу.

Остальное необходимое программное обеспечение находится на CD, сопровождающем книгу.

Для того чтобы на компьютере работали *все* примеры книги, вам понадобится видеокарта GeForce256 или выше. В табл. В1 приведены требования к видеокартам в зависимости от глав. Так как я в настоящее время располагаю только видеокартами корпораций ATI и NVIDIA, требования указаны только для видеокарт этих корпораций, хотя, скорее всего, бóльшая часть примеров будет работать и на видеокартах других фирм.

Таблица В1. *Требования к видеокартам по главам*

Глава	Минимально требуемая видеокарта
Глава 1	NVIDIA RivaTNT
Глава 2	NVIDIA RivaTNT
Глава 3	NVIDIA RivaTNT
Глава 4	NVIDIA RivaTNT2 32MB
Глава 5	NVIDIA GeForce256
Глава 6	NVIDIA GeForce256
Глава 7	NVIDIA GeForce256
Глава 8	NVIDIA GeForce256
Глава 9	NVIDIA GeForce256
Глава 10	NVIDIA GeForce256

К остальным компонентам компьютера не предъявляется каких-либо особенных требований. Это может быть, например, что-нибудь вроде Celeron 266MHz, 128MB RAM.

Все примеры этой книги проверялись на компьютере, конфигурация которого приведена в табл. В2. Часть "подозрительных" примеров тестировались на других компьютерах, на которых были установлены следующие видеокарты: NVIDIA GeForce4 Ti4600 128MB, NVIDIA GeForce2MX 32MB, NVIDIA RivaTNT M64 16MB, ATI Radeon 8500 и ATI 3D Rage128.

Таблица В2. *Конфигурация компьютера, на котором тестировались все примеры для книги*

Процессор	Intel Pentium-4 2.6C
Материнская плата	Asus P4B800 (i865PE)

Таблица B2 *(окончание)*

Оперативная память	1024MB PC3200
Процессор	Intel Pentium-4 2.6C
Видеокарты	ATI Radeon 9700 Pro 128MB
	NVIDIA GeForce FX 5800 Ultra 128MB
	NVIDIA GeForce2 MX 32MB
Операционная система	Microsoft Windows XP (Build 2600) Service Pack 1. English Version

Благодарности

За годы, которые потребовались на то, что бы разобраться с OpenGL, разработать учебный курс и написать эту книгу, я получил неоценимую помощь и поддержку от множества людей.

Во-первых, я хочу поблагодарить редакцию журнала "Программист", особенно Владимира Голубкова и Максима Туйкина, которые доверили неизвестному провинциальному программисту написание серии статей по OpenGL и прощали все задержки.

Отдельно хочется поблагодарить Михаила Краснова, автора книг "OpenGL. Графика в проектах Delphi" и "DirectX. Графика в проектах Delphi", который оказал мне неоценимую помощь в изучении OpenGL, подготовке книги и поиске издателя.

Я очень благодарен Геннадию Ригеру из ATI Technologies, который терпеливо отвечал на сотни моих вопросов по электронной почте, а также содействовал тому, чтобы примеры этой книги нормально работали на видеокартах ATI.

Алексей Лагуненко из ItLabs пригласил меня на конференцию NVIDIA для разработчиков. Именно это событие и вдохновило меня на написание книги.

Я признателен и многим другим людям, которые помогали мне при написании книги: Rev Lebaredian (NVIDIA), Rachel Lebaredian (Steamboat Software), Shawn Steiner (Discreet), Dan Lion (Turbo Squid) и Maxim Perminov (Intel).

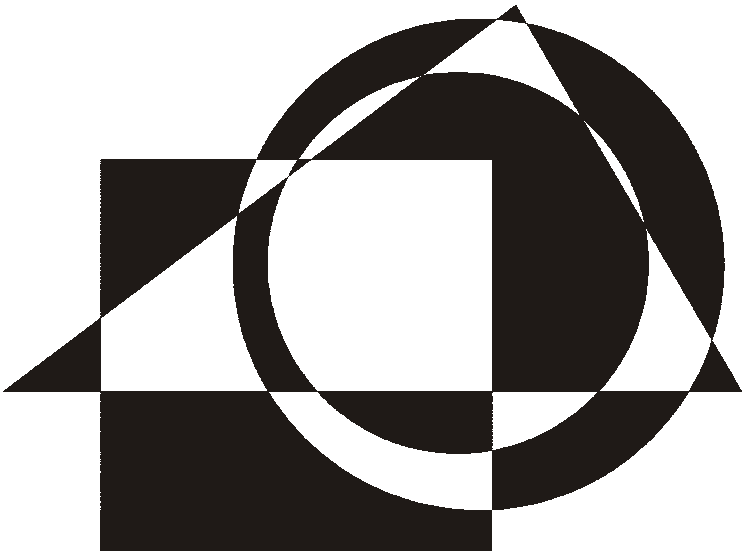
Мое понимание OpenGL формировалось под воздействием целого ряда публикаций, поэтому я очень хочу поблагодарить авторов статей на сайтах **www.ixbt.com**, **www.gamedev.ru**, **www.opengl.org**, **www.nvidia.com/developer**, **www.ati.com/developer**, а именно: Александра Медведева (iXBT), Андрея Воробьева (iXBT), Сергея Ваткина (GameDev), John Spitzer (NVIDIA), Sebastien Domine (NVIDIA) и Ричарда Хадди (ATI). Отдельная благодарность выражается сотрудникам NVIDIA Mark Kilgard и Cass Everitt, разработавшим библиотеки GLUT и OpenGL Helper Library, позволяющие на время забыть о существовании оконных функций, циклов обработки сообщений и прочих "ужасов" Win32.

Другим источником информации были форумы сайта iXBT (**forum.ixbt.com**), особенно форум разработчиков игр. Я бесконечно благодарен сообществу

этого форума за помощь в работе над книгой. Отдельная благодарность выражается Петру Попову, советы которого очень мне пригодились при написании книги.

В заключение хочу поблагодарить корпорации ATI и NVIDIA, которые предоставили в мое распоряжение видеокарты ATI Radeon 9700 Pro и NVIDIA GeForce FX 5800 Ultra, без которых написание этой книги было бы сильно затруднено.

Несмотря на поддержку и помощь других, ответственность за любые неточности и упущения полностью лежит на мне. Любые замечания и пожелания вы можете прислать по адресу **gsaf@sura.ru**.



ЧАСТЬ I

Использование NVIDIA OpenGL SDK

Глава 1

Библиотека GLUT

Как известно, библиотека OpenGL является кроссплатформенной: программа, написанная с использованием этой библиотеки, может работать под любой операционной системой, в которой имеется поддержка этой библиотеки. Но в действительности при реализации программ, написанных с ее помощью, возникает ряд трудностей.

Все дело в том, что в OpenGL отсутствуют средства для организации интерфейса с пользователем: инструменты для создания, работы с мышью и клавиатурой и т. д. Следовательно, при переносе программы на другую платформу ту часть программы, которая организует взаимодействие с пользователем, все равно придется переписывать.

Главная причина в том, что операции оконной подсистемы (создание окна, обработка сообщений, установка формата пикселов и многое другое) очень сильно зависят от операционной системы, и включение этих средств в OpenGL могло бы значительно сузить область применения этой библиотеки.

Поэтому существует потребность в надстройках над OpenGL, которые помогали бы в создании кроссплатформенных приложений. К ним относится библиотека GLUT (OpenGL Utility Toolkit). GLUT позволяет:

- создавать многооконные приложения;
- обрабатывать сообщения, используя процедуры обратного вызова (callback);
- работать с устройствами ввода информации, например, с мышью и клавиатурой;
- использовать таймеры;
- создавать всплывающие (pop-up) меню;
- работать со шрифтами и изображениями

и многое другое.

Программа, использующая GLUT версии 3.7, может работать в Windows 95, Windows NT, OS/2, UNIX и других ОС. Если программа использует только

возможности GLUT, то для переноса ее в любую операционную систему достаточно лишь перекомпилировать ее без изменения исходного текста. Показательно, что примеры, поставляемые с GLUT, работают во всех распространенных ОС без изменения исходного кода. Надо лишь использовать makefile от нужного компилятора.

Наряду с достоинствами у GLUT есть и недостатки. И как всегда, эти недостатки являются следствием достоинств. В данном случае это переносимость. Из-за переносимости приложения GLUT ограничены в функциональности и не имеют доступа ко всем возможностям операционной системы. В целом, библиотека является оптимальной для написания относительно небольших программ.

Но даже если вы не собираетесь использовать GLUT в своих программах, следует учитывать, что существует много программного обеспечения, которое использует эту библиотеку. Например, практически все примеры из NVIDIA OpenGL SDK написаны с использованием GLUT. Следовательно, чтобы разобраться в них, полезно будет знать основы GLUT.

1.1. Подключение GLUT к проекту

Создайте пустое приложение Win32:

1. В меню **File** выберите пункт **New | Project |Visual C++ Projects | Win32 Project**.
2. Выполните команду **Project | Properties | Linker | Command Line** (рис. 1.1).
3. В поле **Additional Options** впишите `glut32.lib` (разумеется, так же надо добавить и строку `opengl32.lib, glu32.lib /entry:"mainCRTStartup`, которая является обязательной для любой OpenGL-программы).

Все эти действия в окне **Property Pages** надо будет повторять каждый раз при создании нового проекта, иначе вы будете получать ошибки о неразрешенных ссылках при компоновке приложения.

Также надо учитывать, что для работы библиотеки необходим файл glut32.dll, который надо будет распространять вместе с программой, т. к. он не входит в стандартную поставку Windows. При установке программы его лучше всего размещать в каталоге \Windows\System 32.

Наконец, добавьте в проект новый файл (**Project | Add To Project | Files**) и назовите его, к примеру, main.cpp.

Теперь можно приступать к созданию нашей первой программы.

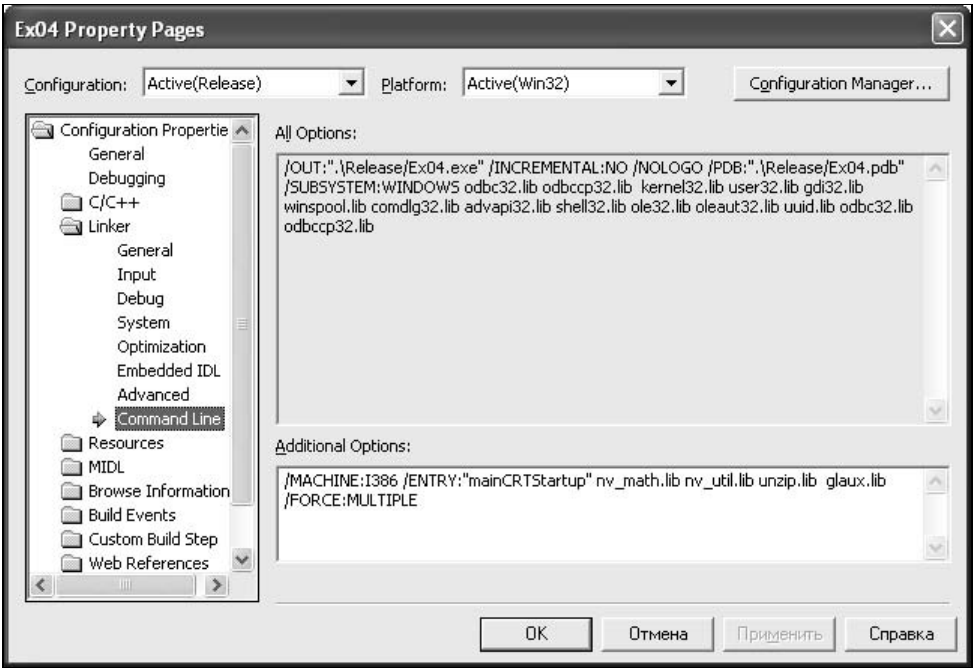


Рис. 1.1. Окно **Property Pages**

1.2. Пример простейшей программы, использующей GLUT

Далее приводится полный текст минимальной программы (листинг 1.1), использующей GLUT, которая рисует окно, заполняя его впоследствии синим цветом (Ex01) (рис. 1.2). Этот и все остальные примеры находятся на CD-диске, прилагаемом к книге. Бóльшая часть текстов примеров приводится частично.

Листинг 1.1

```
#include <gl\glut.h>

// Вывод изображения на экран
void Display()
{
  glClear(GL_COLOR_BUFFER_BIT);
// Смена переднего и заднего буферов экрана
```

```
  glutSwapBuffers();
}

int main(int argc, char* argv[])
{
  glutInit (&argc, argv);
// Задаем параметры окна
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
// Задаем размеры окна
  glutInitWindowSize(640, 480);
// Задаем позицию окна
  glutInitWindowPosition(100, 100);
// Создаем окно
  glutCreateWindow("My first OpenGL Application");

  glClearColor(0.5, 0.5, 0.75, 1);

// Задаем функцию обратного вызова изображения на экран
  glutDisplayFunc(Display);
  glutMainLoop();

  return 0;
}
```

Рис. 1.2. Простейшая GLUT-программа

Первая строка функции `main` вызывает команду `glutInit`, которая инициализирует библиотеку GLUT:

```
void glutInit (int *argcp, char **argv);
```

где:

- `argcp` — указатель на количество аргументов в командной строке;
- `argvv` — указатель на массив аргументов.

Обычно эти значения берутся из главной функции программы: `int main(int argc, char *argv[])`. Функция `glutInit` ищет в списке аргументов командной строки параметры, специфичные для GLUT, и инициализирует себя с учетом этих параметров. После чего убирает их из списка аргументов.

В настоящее время, ни один из параметров командой строки, распознаваемый функцией `glutInit`, не оказывает влияние на программу, работающую в операционных системах семейства Win32. Следовательно, если ваша программа будет работать только в операционных системах семейства Win32, вы можете опустить ее без ущерба для программы.

Следующая команда — `glutInitDisplayMode` — устанавливает формат пикселов окна программы.

```
void glutInitDisplayMode(unsigned int mode);
```

где:

- `mode` — битовая маска, каждый бит которой соответствует определенному атрибуту формата пикселов. Для более удобной установки формата пикселов в библиотеке GLUT имеется набор констант битовых масок (табл. 1.1), объединяя которые при помощи логической операции ИЛИ (OR), программист может создать требуемую ему битовую маску формата пикселов.

Таблица 1.1. *Основные предопределенные константы битовых масок формата пикселов библиотеки GLUT*

Константа	Описание
`GLUT_RGBA`	Необходим RGBA-видеорежим
`GLUT_RGB`	Псевдоним для `GLUT_RGB` (в библиотеке GLUT за поддержку альфа-канала отвечает битовая маска `GLUT_ALPHA`)
`GLUT_INDEX`	Необходим видеорежим цветовых индексов, т. е. целочисленные идентификаторы цветов должны использоваться в качестве индексов в палитре цветов
`GLUT_SINGLE`	Необходима поддержка одиночной буферизацией. Так как значение этой константы равно нулю, ее необязательно указывать

Таблица 1.1 *(окончание)*

Константа	Описание
GLUT_DOUBLE	Необходима поддержка двойной буферизации
GLUT_ACCUM	Необходим буфер аккумулятора
GLUT_ALPHA	Необходима поддержка альфа-канала
GLUT_DEPTH	Необходима поддержка буфера глубины
GLUT_STENCIL	Необходима поддержка буфера шаблона

В нашем случае флаг GLUT_RGBA указывает на то, что мы будем работать в режиме RGBA. В свою очередь флаг GL_DOUBLE свидетельствует о необходимости поддержки двойной буферизации.

Далее при помощи команд glutInitWindowSize и glutInitWindowPosition необходимо установить размеры окна и его позицию на экране:

```
void glutInitWindowSize(int width, int height);
void glutInitWindowPosition(int x, int y);
```

где:

- width — ширина окна в пикселах;
- height — высота окна в пикселах;
- x — положение левого верхнего угла окна (координата *x*);
- y — положение левого верхнего угла окна (координата *y*).

Мы устанавливаем размер окна (640 × 480) и положение его левого верхнего угла (100, 100). Собственно создание окна осуществляется функцией glutCreateWindow, которая создает окно с параметрами, заданными командами glutInitDisplayMode glutInitWindowSize и glutInitWindowPosition:

```
int glutCreateWindow(char *name);
```

где:

- name — заголовок создаваемого окна.

Наша программа создает окно с заголовком **My first OpenGL Application**.

Библиотека GLUT является событийно-ориентированной библиотекой, в результате чего любая GLUT-программа общается с внешним миром при помощи событий. Обработка событий в библиотеке GLUT реализована при помощи механизма функций обратного вызова (callback). Сначала программа передает библиотеке GLUT адрес функции обратного вызова, обрабатывающей данное событие, а затем, при наступлении этого события, биб-

лиотека вызывает указанную функцию обратного вызова, передавая ей, если необходимо, в качестве параметров информацию об особенностях события.

Для установки функции обратного вызова, отвечающей за перерисовку экрана, используется команда `glutDisplayFunc`:

```
void glutDisplayFunc(void (*func)(void));
```

где:

- `func` — адрес функции обратного вызова, отвечающей за обновление содержимого экрана. В Win32 эта функция фактически является обработчиком события `WM_PAINT`.

В нашей программе мы устанавливаем в качестве функции обратного вызова функцию `Display`. Функция `Display()` очень проста — она очищает экран и выводит полученное изображение, используя `glutSwapBuffers()`. Если при инициализации приложения мы используем флаг `GLUT_DOUBLE` вместо `GLUT_SINGLE`, то эту команду придется заменить командой `glFlush()`.

Предпоследняя строка функции `Main` при помощи команды `glutMainLoop()` запускает цикл обработки сообщений. В случае использования ОС семейства Windows команда `glutMainLoop` организует цикл обработки сообщений Win32.

Последняя строка функции `main (return 0)` — скорее дань традиции. В документации по библиотеке GLUT сказано, что функция `glutMainLoop()` никогда не возвращает управление программе. Поэтому если при завершении работы программы вам надо освободить системные ресурсы, то бессмысленно помещать этот код после вызова функции `glutMainLoop`.

Как видите, GLUT сильно облегчает написание простых программ. Кроме того, она делает их независимыми от платформы: в частности, эта программа после перекомпиляции сможет работать практически в любой операционной системе, имеющей поддержку OpenGL.

1.3. Работа с мышью и клавиатурой

В предыдущем разделе мы создали переносимое приложение, которое фактически ничего не делает. Сейчас мы попробуем добавить в него какие-нибудь дополнительные возможности. Для начала добавим в него обработку событий клавиатуры — завершение работы по нажатию клавиши <Esc>.

Для регистрации обработчика сообщений от клавиатуры в библиотеке GLUT используется команда `glutKeyboardFunc`:

```
void glutKeyboardFunc(void (*func)(unsigned char key,int x,   int y));
```

Как видно из определения этой функции, обработчик событий от клавиатуры принимает следующие параметры:

- `key` — идентификатор клавиши;
- `x`, `y` — координаты указателя мыши в момент нажатия клавиши.

Ниже приведен исходный код обработчика клавиатуры, который завершает работу программы при нажатии клавиши <Esc>

```
void Keyboard(unsigned char key, int x, int y)
{
  switch(key)
  {
  case 27:
  {
    exit(VK_ESCAPE);
    break;
  }
```

Работа с мышью организуется тоже очень просто (Ex02). Для регистрации обработчика события перемещения указателя мыши используются две команды: `glutMotionFunc` и `glutPassiveMotionFunc`:

```
void glutMotionFunc(void (*func)(int x, int y));
void glutPassiveMotionFunc(void (*func)(int x, int y));
```

Первая команда используется для обработки события перемещения мыши при нажатой кнопке мыши, вторая — при отжатой. Обработчик в качестве параметров принимает координаты (*x*, *y*) указателя мыши. Ниже приведен текст обработчика, который выводит в заголовке окна текущие координаты указателя мыши (рис. 1.3):

```
void MousePassiveMotion(int x, int y)
{
  char buf[80];
  sprintf(buf,"Mouse coords is: x=%d; y=%d", x, y);
  glutSetWindowTitle(buf);
}
glutPassiveMotionFunc(MousePassiveMotion);
```

Для установки заголовка окна используется функция `glutSetWindowTitle`:

```
void glutSetWindowTitle(char *name);
```

где:

- `name` — новый заголовок окна.

Рис. 1.3. Вывод точки на экран

Запустив программу на выполнение, обратите внимание на то, что при нажатой кнопке мыши обработка сообщения не происходит. Чтобы устранить этот недостаток, в функцию `main` необходимо добавить следующую строку:

```
glutMotionFunc(MousePassiveMotion);
```

Для того чтобы зарегистрировать обработчик событий нажатия кнопки мыши, применяется функция `glutMouseFunc`:

```
void glutMouseFunc(void (*func)(int button, int state, int x, int y));
```

где:

- `button` — идентификатор кнопки мыши (`GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON`, `GLUT_RIGHT_BUTTON` для левой, средней и правой соответственно);
- `state` — состояние кнопки мыши (`GLUT_DOWN` — нажата, `GLUT_UP` — отжата);
- `x`, `y` — координаты указателя мыши.

Приведенный далее фрагмент программы (часть кода опущена) при нажатии левой кнопки мыши рисует круглую точку размером 10 пикселов на месте курсора (листинг 1.2).

Листинг 1.2

```
#include <windows.h>
#include <gl\glut.h>
#include <stdio.h>

// Размеры окна
int WinWidth=640;
int WinHeight=480;
// Текущая позиция точки
int mx=0;
int my=0;

...

void Display()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(0, 1, 0);
// Рисуем точку
  glBegin(GL_POINTS);
    glVertex2f(mx, WinHeight-my);
  glEnd();

  glutSwapBuffers();
}

void Mouse(int button, int state, int x, int y)
{
// Если нажата левая кнопка мыши
  if ((button==GLUT_LEFT_BUTTON) | (state==GLUT_DOWN))
  {
// Устанавливаем координаты точки
```

```
    mx=x;
    my=y;
    glutPostRedisplay();
  }
}

int main(int argc, char* argv[])
{
...
  glClearColor(0.5, 0.5, 0.75, 1);

// Делаем точку круглой
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glEnable(GL_BLEND);
  glEnable(GL_POINT_SMOOTH);
  glPointSize(10);
// Параметры проекции
  gluOrtho2D(0, WinWidth-1, 0, WinHeight-1);

  glutDisplayFunc(Display);
  glutMouseFunc(Mouse);
...
```

Заметьте, что в функции `Mouse` для перерисовки окна используется команда `glutPostRedisplay()`:

```
void glutPostRedisplay(void);
```

Эта функция посылает приложению сообщение о необходимости обновления окна, что в нашем случае приводит к вызову функции обратного вызова `Display()`.

Также обратите внимание на преобразование координат, которое выполняется в команде `glVertex3f`. Оно связано с тем, что системы координат окна Windows и OpenGL не совпадают (в Windows по умолчанию начало координат находится в левом верхнем углу, а в OpenGL — в левом нижнем).

Если вы поработаете некоторое время с этим примером, то увидите, что при изменении размеров окна приложение перестает корректно работать. Это связано с тем, что при изменении размеров окна надо менять матрицу модели командой `gluOrtho2D` и область вывода командой `glViewport`. Выходом из данной ситуации может служить использование команды `glutReshapeFunc`,

которая определяет функцию обратного вызова, вызываемую при изменении размеров окна.

```
void glutReshapeFunc(void (*func)(int width, int height));
```

где:

- `width` — новая ширина окна;
- `height` — новая высота окна.

После внесения изменений в программу мы получим код, представленный в листинге 1.3 (Ex03):

Листинг 1.3

```
#include <windows.h>
#include <gl\glut.h>
#include <stdio.h>

int WinWidth=640;
int WinHeight=480;

int mx=0;
int my=0;

void Display()
{
  glClear(GL_COLOR_BUFFER_BIT);

  glColor3f(0, 1, 0);

  glBegin(GL_POINTS);
    glVertex2f(mx, WinHeight-my);
  glEnd();

  glutSwapBuffers();
}

void Keyboard(unsigned char key, int x, int y)
{
  switch(key)
  {
```

```
  case VK_ESCAPE:
  {
    exit(0);
    break;
  }
  }
}

void MousePassiveMotion(int x, int y)
{
  char buf[80];
 sprintf(buf,"Mouse coords is: x=%d; y=%d", x, y);
  glutSetWindowTitle(buf);
}

void Mouse(int button, int state, int x, int y)
{
  if ((button==GLUT_LEFT_BUTTON) | (state==GLUT_DOWN))
  {
    mx=x;
    my=y;
    glutPostRedisplay();
  }
}
// Обрабатываем изменение размеров окна
void Reshape(int width, int height)
{
// Запоминаем размеры окна
  WinWidth=width;
  WinHeight=height;

// Задаем размеры окна
  glViewport(0, 0, WinWidth, WinHeight);

// Задаем матрицу проекции
  glLoadIdentity();
  gluOrtho2D(0, WinWidth-1, 0, WinHeight-1);
}
```

```
int main(int argc, char* argv[])
{
  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
  glutInitWindowSize(WinWidth, WinHeight);
  glutInitWindowPosition(100, 100);
  glutCreateWindow("My first OpenGL Application");

  glClearColor(0.5, 0.5, 0.75, 1);

  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
  glEnable(GL_BLEND);
  glEnable(GL_POINT_SMOOTH);
  glPointSize(10);

  glutDisplayFunc(Display);
  glutKeyboardFunc(Keyboard);
  glutPassiveMotionFunc(MousePassiveMotion);
  glutMouseFunc(Mouse);
  glutReshapeFunc(Reshape);

  glutMainLoop();

  return 0;
}
```

Во время обработки событий мыши часто бывает нужно определить, нажата ли сейчас какая-нибудь из клавиш-модификаторов (<Ctrl>, <Alt> или <Shift>). Для получения этой информации в GLUT используется команда `glutGetModifiers`, которая возвращает битовую маску, каждый бит которой соответствует определенной клавише-модификатору.

```
int glutGetModifiers(void);
```

Если клавиша-модификатор нажата, то соответствующий бит устанавливается в 1. Для более удобной проверки в GLUT определены три константы (битовые маски): `GLUT_ACTIVE_SHIFT`, `GLUT_ACTIVE_CTRL` и `GLUT_ACTIVE_ALT`, которые соответствуют клавишам <Shift>, <Ctrl> и <Alt>. Примеры использования клавиш модификаторов будут рассмотрены в *главе 2*.

Библиотека GLUT предоставляет программисту и механизм воздействия на указатель мыши: принудительное изменение его позиции или формы. Пер-

вая операция выполняется командой `glutWarpPointer`, принимающей в качестве параметров новые координаты указателя мыши.

```
void glutWarpPointer(int x, int y);
```

где:

- `x`, `y` — новые координаты указателя мыши.

Для изменения формы указателя мыши используется команда `glutSetCursor`, принимающая в качестве параметра идентификатор курсора мыши:

```
void glutSetCursor(int cursor);
```

где:

- `cursor` — идентификатор курсора (табл. 1.2).

Таблица 1.2. *Идентификаторы курсоров*

Идентификатор	Описание
GLUT_CURSOR_RIGHT_ARROW	Стрелка, указывающая вверх и вправо
GLUT_CURSOR_LEFT_ARROW	Стрелка, указывающая вверх и влево
GLUT_CURSOR_INFO	Рука
GLUT_CURSOR_DESTROY	Череп и две кости
GLUT_CURSOR_HELP	Знак вопроса
GLUT_CURSOR_CYCLE	Стрелка в виде дуги окружности
GLUT_CURSOR_SPRAY	Перекрестие двух двунаправленных стрелок влево-вправо и вверх-вниз
GLUT_CURSOR_WAIT	Песочные часы
GLUT_CURSOR_TEXT	Указатель на точку вставки текста
GLUT CURSOR CROSSHAIR	Простое перекрестие
GLUT_CURSOR_UP_DOWN	Двунаправленный указатель вверх-вниз
GLUT_CURSOR_LEFT_RIGHT	Двунаправленный указатель влево-вправо
GLUT_CURSOR_TOP_SIDE	Стрелка, указывающая вверх
GLUT_CURSOR_BOTTOM_SIDE	Стрелка, указывающая вниз
GLUT_CURSOR_LEFT_SIDE	Стрелка, указывающая влево
GLUT_CURSOR_RIGHT_SIDE	Стрелка, указывающая вправо
GLUT_CURSOR_TOP_LEFT_CORNER	Стрелка, указывающая в верхний левый угол

Таблица 1.2 *(окончание)*

Идентификатор	Описание
GLUT_CURSOR_TOP_RIGHT_CORNER	Стрелка, указывающая в верхний правый угол
GLUT_CURSOR_BOTTOM_RIGHT_CORNER	Стрелка, указывающая в нижний правый угол
GLUT_CURSOR_BOTTOM_LEFT_CORNER	Стрелка, указывающая в нижний правый угол
GLUT_CURSOR_FULL_CROSSHAIR	Перекрестие полноэкранного курсора
GLUT_CURSOR_NONE	Невидимый курсор
GLUT_CURSOR_INHERIT	Курсор по умолчанию

Для демонстрации использования на практике команд `glutWarpPointer` и `glutSetCursor`, далее приведена программа, рисующая на экране прямоугольник, размеры которого можно изменять путем перемещения его углов и сторон посредством мыши (листинг 1.4). При этом курсор мыши автоматически изменяет свою форму в зависимости от того, какая грань или вершина находится под ним. Кроме того, программа при запуске автоматически помещает курсор в центр экрана.

Листинг 1.4

```
#include <windows.h>
#include <gl\glut.h>
#include <stdio.h>

// Ширина и высота экрана
int WinWidth=640;
int WinHeight=480;

// Позиция мыши
int mx=0;
int my=0;

// Координаты левого нижнего и правого верхнего углов прямоугольника
int p1x=50;
int p1y=50;
int p2x=150;
int p2y=150;
```

```
// Если перемещается координата p1x
bool b_p1x=false;
// Если перемещается координата p1y
bool b_p1y=false;
// Если перемещается координата p2x
bool b_p2x=false;
// Если перемещается координата p2y
bool b_p2y=false;

// Нажата левая кнопка мыши
bool bLeftDown=false;

// Установка курсора
void SetCursor()
{
// Не перемещается ни одна из точек
       b_p1x=false;
       b_p1y=false;
       b_p2x=false;
       b_p2y=false;

// Если курсор над левым нижним углом прямоугольника
       if ((abs(mx-p1x)<5) && ((abs(my-p1y)<5)))
       {
// Изменяем форму курсора
              glutSetCursor(GLUT_CURSOR_BOTTOM_LEFT_CORNER);
// Устанавливаем флаги модификации вершин
              b_p1x=true;
              b_p1y=true;
              return;
       }

// Если курсор над левым верхним углом прямоугольника
       if ((abs(mx-p1x)<5) && ((abs(my-p2y)<5)))
       {
              glutSetCursor(GLUT_CURSOR_TOP_LEFT_CORNER);
              b_p1x=true;
              b_p2y=true;
```

```
                return;
        }

// Если курсор над правым верхним углом прямоугольника
        if ((abs(mx-p2x)<5) && ((abs(my-p2y)<5)))
        {
                glutSetCursor(GLUT_CURSOR_TOP_RIGHT_CORNER);
                b_p2x=true;
                b_p2y=true;
                return;
        }

// Если курсор над правым нижним углом прямоугольника
        if ((abs(mx-p2x)<5) && ((abs(my-p1y)<5)))
        {
                glutSetCursor(GLUT_CURSOR_BOTTOM_RIGHT_CORNER);
                b_p2x=true;
                b_p1y=true;
                return;
        }

// Если курсор над левой стороной прямоугольника
        if ((abs(mx-p1x)<5) && (my>p1y) && (my<p2y))
        {
                glutSetCursor(GLUT_CURSOR_LEFT_RIGHT);
                b_p1x=true;
                return;
        }

// Если курсор над правой стороной прямоугольника
        if ((abs(mx-p2x)<5) && (my>p1y) && (my<p2y))
        {
                glutSetCursor(GLUT_CURSOR_LEFT_RIGHT);
                b_p2x=true;
                return;
        }

// Если курсор над нижней стороной прямоугольника
        if ((abs(my-p1y)<5) && (mx>p1x) && (mx<p2x))
```

```
        {
                glutSetCursor(GLUT_CURSOR_UP_DOWN);
                b_p1y=true;
                return;
        }

// Если курсор над верхней стороной прямоугольника
        if ((abs(my-p2y)<5) && (mx>p1x) && (mx<p2x))
        {
                glutSetCursor(GLUT_CURSOR_UP_DOWN);
                b_p2y=true;
                return;
        }

// Если курсор над прямоугольником
        if ((mx>p1x) && (mx<p2x) && (my>p1y) && (my<p2y))
        {
                glutSetCursor(GLUT_CURSOR_SPRAY);
                b_p1x=true;
                b_p1y=true;
                b_p2x=true;
                b_p2y=true;
                return;
        }

// Иначе — курсор по умолчанию
        glutSetCursor(GLUT_CURSOR_INHERIT);
}

// Рисует на экране прямоугольник
void Display()
{
        glClear(GL_COLOR_BUFFER_BIT);

        glRectf(p1x, p1y, p2x, p2y);

        glutSwapBuffers();
}
```

```
// Выходит из программы при нажатии клавиши VK_ESC
void Keyboard(unsigned char key, int x, int y)
{
        switch(key)
        {
                case VK_ESCAPE:
                        {
                                exit(0);
                                break;
                        }
        }
}

// Пассивное перемещение мыши
void MousePassiveMotion(int x, int y)
{
// Запоминает новые координаты мыши
        mx=x;
        my=WinHeight-y;

// Изменяет форму курсора
        SetCursor();
}

// Обработчик нажатия кнопок мыши
void Mouse(int button, int state, int x, int y)
{
// Если нажата левая кнопка мыши
        if ((button==GLUT_LEFT_BUTTON) && (state==GLUT_DOWN))
        {
// Сохраняем текущее состояние мыши
                mx=x;
                my=WinHeight-y;
                bLeftDown=true;

// Обновляем курсор (+ устанавливаем флажки изменяемых координат)
                SetCursor();
        }
```

```
// Если левая кнопка отжата
        if ((button==GLUT_LEFT_BUTTON) && (state==GLUT_UP))
// Сниманием флаг
                bLeftDown=false;
}

// Перемещение мыши
void MouseMotion(int x, int y)
{
// Если нажата левая кнопка
        if (bLeftDown)
        {
// Если установлен флаг модификации p1_x
                if (b_p1x)
// Модифицируем p1x
                        p1x+=x-mx;
// Аналогично модифицируем другие флаги
                if (b_p1y)
                        p1y+=WinHeight-y-my;
                if (b_p2x)
                        p2x+=x-mx;
                if (b_p2y)
                        p2y+=WinHeight-y-my;
                mx=x;
                my=WinHeight-y;
                glutPostRedisplay();
        }
}

// Обработка изменения размеров окна
void Reshape(int width, int height)
{
        WinWidth=width;
        WinHeight=height;

        glViewport(0, 0, WinWidth, WinHeight);

        glLoadIdentity();
        gluOrtho2D(0, WinWidth-1, 0, WinHeight-1);
}
```

```
int main(int argc, char* argv[])
{
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
        glutInitWindowSize(WinWidth, WinHeight);
        glutInitWindowPosition(100, 100);
        glutCreateWindow("GLUT Cursor Demo");

        glClearColor(0.5, 0.5, 0.75, 1);

        glutDisplayFunc(Display);
        glutKeyboardFunc(Keyboard);
        glutPassiveMotionFunc(MousePassiveMotion);
        glutMouseFunc(Mouse);
        glutMotionFunc(MouseMotion);
        glutReshapeFunc(Reshape);

// Перемещаем указатель мыши в центр экрана
        glutWarpPointer(WinWidth/2, WinHeight/2);

        glutMainLoop();

        return 0;
}
```

1.4. Работа с джойстиком

В текущей версии GLUT 3.7 используется бета-версия интерфейса для работы с джойстиком, который может быть изменен в следующих версиях. Кроме того, команды для работы с джойстиком не описаны в спецификации GLUT. Тем не менее, этот интерфейс вполне работоспособен и может использоваться в демонстрационных программах.

Для установки обработчика событий от джойстика используется функция `glutJoystickFunc`:

```
void glutJoystickFunc(void (*func)(unsigned int buttonMask, int x, int y,
int z), int pollInterval);
```

где:

- `buttomMask` — битовая маска состояния кнопок. Единица в каждом бите соответствует нажатию определенной кнопки джойстика;

- `x`, `y` и `z` — углы поворота рукояток джойстика относительно трех осей, которые находятся в диапазоне от −1000 до 1000;
- `pollInterval` — точность опроса состояний джойстика.

В качестве параметров эта функция принимает указатель на обработчик событий от джойстика и точность опроса состояния джойстика. Чем меньше эта величина, тем чаще будет опрашиваться джойстик. В большинстве случаев частота опроса должна быть равна приблизительно 100. Если точность опроса равна нулю, то ответственность за опрос состояния джойстика лежит на программисте, который должен периодически форсировать принудительный опрос состояния джойстика при помощи функции `glutForceJoystickFunc`:

```
void glutForceJoystickFunc(void);
```

После опроса состояния джойстика GLUT автоматически вызывает обработчик событий от джойстика.

Внимание

Из-за ошибки в GLUT 3.7 нормальный опрос состояний кнопок джойстика возможен только в режиме принудительного опроса.

Для демонстрации использования джойстика средствами библиотеки GLUT автор переделал пример Ex04 "под джойстик" (Ex06). Теперь перемещение точки на экране задается поворотом рукоятки джойстика относительно осей *x*, *y*, а цвет — поворотом относительно оси *z*. Исходный код примера находится в листинге 1.5.

Листинг 1.5

```
#include <windows.h>
#include <gl\glut.h>
#include <stdio.h>

int WinWidth=640;
int WinHeight=480;

// Положение точки
int mx=0;
int my=0;
// Цвет точки
float color=255;

// Рисует точку на экране
void Display()
```

```
{
        glClear(GL_COLOR_BUFFER_BIT);

        glBegin(GL_POINTS);
                glColor3f(color, color, color);
                glVertex2f(mx, my);
        glEnd();

        glutSwapBuffers();
}

// Выходит при нажатии клавиши ESC
void Keyboard(unsigned char key, int x, int y)
{
        switch(key)
        {
                case VK_ESCAPE:
                        {
                                exit(0);
                                break;
                        }
        }
}

// Обработчик изменений размеров окна
void Reshape(int width, int height)
{
        WinWidth=width;
        WinHeight=height;

        glViewport(0, 0, WinWidth, WinHeight);

        glLoadIdentity();
        gluOrtho2D(-1000.0, 1000.0, -1000.0, 1000.0);
}

// Обработчик событий от джойстика
void joystick(unsigned int buttonMask, int x, int y, int z)
```

```
{
        mx=x;
        my=-y;
        color=float(1000+z)/2000.0;

        char buf[1024];
        sprintf(buf, "joy 0x%x, x=%d y=%d z=%d", buttonMask, x, y, z);
        glutSetWindowTitle(buf);

        glutPostRedisplay();
}

int main(int argc, char* argv[])
{
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
        glutInitWindowSize(WinWidth, WinHeight);
        glutInitWindowPosition(100, 100);
        glutCreateWindow("My first OpenGL Application");

        glClearColor(0.5, 0.5, 0.75, 1);

        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        glEnable(GL_BLEND);
        glEnable(GL_POINT_SMOOTH);
        glPointSize(10);

        glutDisplayFunc(Display);
        glutKeyboardFunc(Keyboard);
        glutReshapeFunc(Reshape);
// Регистрируем обработчик событий от джойстика
        glutJoystickFunc(joystick, 100);

        glutMainLoop();

        return 0;
}
```

1.5. Пример трехмерного приложения

Теперь после того как мы разобрались с основами GLUT, попробуем создать полноценное OpenGL-приложение. Рассмотрим программу (Ex06), которая рисует на экране трехмерный чайник (рис. 1.4). Изображение можно будет вращать, нажимая левую кнопку мыши и перемещать, используя правую. Вследствие того, что нельзя однозначно определить соответствие между двухмерными координатами мыши и трехмерными координатами чайника, переключать плоскости, вдоль которых перемещается объект, будет возможно посредством клавиши <F1>. Название плоскости будем показывать в заголовке окна.

Рисование чайника вручную "по точкам" с использованием полигонов или сплайновых поверхностей — занятие не из приятных. К счастью, в этом нет необходимости. В библиотеке GLUT имеется набор функций для рисования наиболее распространенных объектов: сферы, куба, конуса, тора, додекаэдра, октаэдра, тетраэдра, икосаэдра и даже чайника. Все команды рисования этих объектов представлены в двух вариантах: команды рисования закрашенных объектов (префикс `solid`) и команды рисования каркасных объектов (префикс `wire`). Ниже приведен краткий перечень команд рисования геометрических объектов библиотеки GLUT.

Команды рисования сферы:

- `void glutSolidSphere(GLdouble radius, GLint slices, GLint stacks);`
- `void glutWireSphere(GLdouble radius, GLint slices, GLint stacks);`

где:

- `radius` — радиус сферы;
- `slices` — количество делений вокруг оси z (аналогично линиям географической долготы);
- `stacks` — количество делений вдоль оси z (аналогично линиям географической доготы).

Команды рисования куба:

- `void glutSolidCube(GLdouble size);`
- `void glutWireCube(GLdouble size);`

где:

- `size` — размер ребра куба.

Команды рисования конуса:

- `void glutSolidCone(GLdouble base, GLdouble height, GLint slices, GLint stacks);`

❒ `void glutWireCone(GLdouble base, GLdouble height, GLint slices, GLint stacks);`

где:

- `base` — радиус основания конуса;
- `height` — высота конуса;
- `slices` — количество делений вокруг оси z;
- `stacks` — количество делений вдоль оси z.

Команды рисования тора:

❒ `void glutSolidTorus(GLdouble innerRadius, GLdouble outerRadius,GLint nsides, GLint rings);`

❒ `void glutWireTorus(GLdouble innerRadius, GLdouble outerRadius,GLint nsides, GLint rings);`

где:

- `innerRadius` — внутренний радиус тора;
- `outerRadius` — внешний радиус тора;
- `sides` — количество сторон у каждого радиального сечения тора;
- `rings` — количество радиальных сечений тора.

Команды рисования додекаэдра радиусом $\sqrt{3}$:

❒ `void glutSolidDodecahedron(void);`

❒ `void glutWireDodecahedron(void).`

Команды рисования октаэдра радиусом 1:

❒ `void glutSolidOctahedron(void);`

❒ `void glutWireOctahedron(void).`

Команды рисования тетраэдра радиусом $\sqrt{3}$:

❒ `void glutSolidTetrahedron(void);`

❒ `void glutWireTetrahedron(void).`

Команды рисования икосаэдра радиусом 1:

❒ `void glutSolidIcosahedron(void);`

❒ `void glutWireIcosahedron(void).`

Команды рисования чайника.

❒ `void glutSolidTeapot(GLdouble size);`

❒ `void glutWireTeapot(GLdouble size),`

где:

- `size` — размер чайника

В описании библиотеки GLUT ничего не говориться о том, как параметр `size` связан с геометрическими размерами чайника. Однако в *главе 6* мы определим, что чайник размером 1.0 вписан в прямоугольный параллелепипед, крайние вершины которого имеют координаты (−1.5, −0.75, −1.0) и (1.717, 0.825, 1.0), а его стороны параллельны осям координат. Иными словами, высота чайника равна удвоенному значению параметра `size`.

Ниже приведен текст программы с подробными комментариями (листинг 1.6).

Листинг 1.6

```
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <gl\glut.h>
#include <string>
#include <windows.h>

using namespace std;

int WinWidth=640;   // Ширина окна
int WinHeight=480;  // Высота окна

GLfloat  rx=0;  // Угол поворота сцены вокруг оси X
GLfloat  ry=0;  // Y
GLfloat  tx=0;  // Сдвиг по оси X
Glfloat  ty=0;  // Y
Glfloat  tz=-9; // Z
Glint  tt=0;    // Активная плоскость: 0 - XY, 1 - XZ

const string tstr[2]={"Translate XY", "Translate XZ"};

int mx,my;  // Координаты мыши
bool ldown=false;  // Нажата левая кнопка мыши?
bool rdown=false;  // Нажата правая кнопка мыши?

GLuint list=0;

void Init()  // Инициализация OpenGL
{
```

```
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glEnable(GL_COLOR_MATERIAL);

  glColor3f(0.1,0.7,0.2);
  glClearColor(0.5, 0.5, 0.75, 1);

  list=glGenLists(1);

// Создание дисплейного списка объекта (чайника)
  glNewList(list, GL_COMPILE);
glutSolidTeapot(2);
  glEndList();
}

void Display()  // Обновление содержимого экрана
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glPushMatrix();
  glTranslatef(tx,ty,tz);  // Перемещение и поворот объекта
  glRotatef(rx,1,0,0);
  glRotatef(ry,0,1,0);

  glCallList(list);  // Вывод объекта на экран
  glPopMatrix();

  glutSwapBuffers();
}

// Обработка сообщений от клавиатуры
void Keyboard(unsigned char key,int x,int y)  {
  switch (key)
  {
  case VK_ESCAPE:  // Если нажата клавиша <ESC> - выход
    if (list)
      glDeleteLists(list,1);  // Удалить дисплейный список
    exit(0);
```

```
    break;
  }
}

void KeyboardSpecial(int key, int x, int y)
{
  switch (key)
  {
  case GLUT_KEY_F1:  // Если нажата клавиша <F1>
  {
    tt=(tt+1)%2;  // Смена плоскости перемещения
    glutSetWindowTitle(tstr[tt].c_str()); // Изменение заголовка
  }

  }
}

void Reshape(int Width,int Height)  // Обработка изменения размеров окна
{
  glViewport(0,0,Width,Height);
  WinWidth=Width;  // Запомнить новые размеры
  WinHeight=Height;

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45,glDouble(WinWidth)/WinHeight,1,100);

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  glutPostRedisplay();
}

void Mouse(int button, int state, int x, int y)  // Обработка щелчков мыши
{
  if (button==GLUT_LEFT_BUTTON)  // Левая кнопка
  {
    switch (state)
    {
```

```
    case GLUT_DOWN:  // Если нажата,
      ldown=true;  // установить флаг
      mx=x;  // Запомнить координаты
      my=y;
      break;
    case GLUT_UP:
      ldown=false;
      break;
    }
  }
  if (button==GLUT_RIGHT_BUTTON)  // Правая кнопка
  {
    switch (state)
    {
    case GLUT_DOWN:
      rdown=true;
      mx=x;
      my=y;
      break;
    case GLUT_UP:
      rdown=false;
      break;
    }
  }
}

void MouseMotion(int x, int y)  // Перемещение мыши
{
  if (ldown)  // Левая кнопка
  {
    rx+=0.5*(y-my);  // Изменение угла поворота
    ry+=0.5*(x-mx);
    mx=x;
    my=y;
    glutPostRedisplay();  // Перерисовать экран
  }
  if (rdown)  // Правая
  {
    tx+=0.01*(x-mx);  // Перемещение вдоль активной плоскости
```

```
    if (tt)
      tz+=0.01*(y-my);
    else
     ty+=0.01*(my-y);
    mx=x;
    my=y;
    glutPostRedisplay();
  }
}

int main(int argc, char* argv[])
{
  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
  glutInitWindowSize(WinWidth,WinHeight);
  glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                         (glutGet(GLUT_SCREEN_HEIGHT)-WinHeight)/2);
  glutCreateWindow("GLUT Teapot");

  Init();

  glutDisplayFunc(Display);
  glutKeyboardFunc(Keyboard);
  glutSpecialFunc(KeyboardSpecial);
  glutReshapeFunc(Reshape);
  glutMouseFunc(Mouse);
  glutMotionFunc(MouseMotion);

  glutMainLoop();

  return 0;
}
```

Этот пример содержит ряд интересных моментов. Во-первых, в нем демонстрируется создание окна в центре экрана. Необходимая информация о высоте и ширине экрана получается с помощью команды `glutGet`, которая используется для получения информации о различных параметрах оконной подсистемы:

```
int glutGet(GLenum state);
```

где:

- `state` — константа-идентификатор запрашиваемого параметра (табл. 1.3).

Рис. 1.4. Простейшее трехмерное приложение

Таблица 1.3. *Некоторые константы-идентификаторы запрашиваемых параметров команды* `glutGet`

Идентификатор	Описание
`GLUT_WINDOW_X`	*X*-координата левого верхнего угла окна
`GLUT_WINDOW_Y`	*Y*-координата левого верхнего угла окна
`GLUT_WINDOW_WIDTH`	Ширина окна
`GLUT_WINDOW_HEIGHT`	Высота окна
`GLUT_WINDOW_CURSOR`	Идентификатор текущего курсора мыши

Таблица 1.3 *(окончание)*

Идентификатор	Описание
GLUT_SCREEN_WIDTH	Ширина экрана
GLUT_SCREEN_HEIGHT	Высота экрана
GLUT_ELAPSED_TIME	Количество миллисекунд, прошедших с момента первого вызова функции glutInit

В нашем случае для определения размеров экрана мы должны вызвать функцию glutGet с параметрами GLUT_SCREEN_WIDTH и GLUT_SCREEN_HEIGHT. Разумеется, для этой цели можно использовать и функцию Win32 GetSystemMetrics, но в этом случае программа уже не будет переносимой на другие платформы.

Вторая особенность нашей программы заключается в том, что для регистрации обработчика клавиши <F1> используется команда glutSpecialFunc:

```
void glutSpecialFunc(void (*func)(int key, int x, int y));
```

где:

- key — идентификатор клавиши;
- x, y — координаты указателя мыши в момент нажатия клавиши.

Это связано с тем, что в GLUT для обработки обыкновенных и специальных клавиш (<F1>—<F12>, стрелки и т. д.) используются разные обработчики событий.

Советую хорошо разобраться с этим примером, т. к. большинство последующих программ будет основано на нем.

1.6. Создание анимации с использованием таймера библиотеки GLUT

Существуют два распространенных способа создания анимации: использование таймера и простоев приложения Idle. Эти способы подробно описаны в книге [1], поэтому мы здесь остановимся только на реализации этих способов в GLUT.

Таймер создается командой glutTimerFunc:

```
void glutTimerFunc(unsigned int msecs, void (*func)(int value), value);
```

где:

- msecs — интервал времени в миллисекундах, по прошествии которого будет вызвана функция обратного вызова;

- `func` — адрес функции обратного вызова;
- `value` — целое число, которое будет передано функции обратного вызова.

Обратите внимание, что таймер библиотеки GLUT не тикает с заданным периодом. После установки таймер срабатывает только один раз. Для того чтобы заставить таймер работать циклично, придется каждый раз заново его программировать внутри обработчика событий.

Ниже приведен фрагмент приложения (листинг 1.7) (Ex07), использующего таймер. Анимация запускается нажатием клавиши <T>. На время воспроизведения анимации прекращается обработка событий от мыши (посредством передачи в качестве адреса обработчика значения `NULL`).

Листинг 1.7

```
...
GLfloat  rx=0;  // Угол поворота сцены вокруг оси X
GLfloat  ry=0;  // Y
...
Glboolean bTimer=0;  // Состояние таймера


void Keyboard(unsigned char key,int x,int y)  // Обработка сообщений от
                                              // клавиатуры
{
  switch (key)
  {
...
  case 'T' | 't':
    bTimer=!bTimer;
// Если надо, таймер включен
    if (bTimer)
    {
// Устанавливаем следующий тик таймера через 25 миллисекунд
      glutTimerFunc(25, Timer, 1);
// Отключение обработки событий от мыши
      glutMouseFunc(NULL);
      glutMotionFunc(NULL);
    }
    break;
  }
}
```

```
void Timer(int value)  // Обработчик таймера
{
  if (bTimer)
  {
// Увеличиваем угол поворота
    ry+=2;
    if (rx>360)
      rx-=360;
    glutPostRedisplay();
// Устанавливаем следующий тик таймера через 25 миллисекунд
    glutTimerFunc(25, Timer, 1);
  }
  else
  {
// Включаем обработку событий от мыши и клавиатуры
    glutMouseFunc(Mouse);
    glutMotionFunc(MouseMotion);
  }
}
```

1.7. Создание анимации с использованием команды *glutIdleFunc*

Главный недостаток программы, создающей анимацию с использованием таймера — фиксированное число кадров. Если попытаться установить слишком большую частоту смены кадров, то компьютер может не справиться с высоким темпом обработки информации, в результате чего программа начнет работать некорректно (может переполниться очередь сообщений и т. д.).

Команда `glutIdleFunc` используется для регистрации обработчика события, происходящего в момент бездействия программы (когда очередь сообщений пуста):

```
void glutIdleFunc(void (*func)(void));
```

Если в этот обработчик вставить вызов команды `glutPostRedisplay`, то смена кадров происходит с максимально возможной частотой для данного компьютера. Однако при этом нельзя использовать фиксированные приращения координат объектов, как это делалось в примере с таймером, т. к. в этом случае анимация будет выполняться с разной скоростью на разных компьютерах.

Самое простое решение этой проблемы (под Windows) — использование функции `glutGet(GLUT_ELAPSED_TIME)`, возвращающей количество миллисекунд, прошедших с момента вызова функции `glutInit`. Зная время начала события, несложно вычислить координаты объекта в заданный момент времени.

Если же вы не создаете кроссплатформенную программу, то для замера интервалов времени можно воспользоваться и средствами Win32: например, функцией `GetTickCount()`.

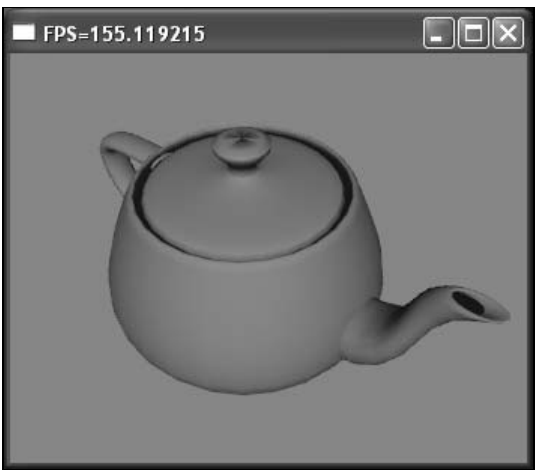


Рис. 1.5. Вывод FPS

Далее частично приводится программа (листинг 1.8) (Ex08), реализующая такой подход (рис. 1.5). Анимация включается клавишей <F2>. В программе также демонстрируется вычисление FPS (количество кадров в секунду — показатель скорости работы приложения).

Листинг 1.8

```
// Время, когда был нарисован прошлый кадр
GLuint OldTick;
// Счетчик кадров
GLuint FramesCount;
// Время начала запуска анимации
GLuint StartTick;
...
```

```
void Display()  // Обновление содержимого экрана
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glPushMatrix();
    glTranslatef(tx,ty,tz);  // Перемещение и поворот объекта
    glRotatef(rx,1,0,0);
    glRotatef(ry,0,1,0);
    glCallList(list);  // Вывод объекта на экран
  glPopMatrix();

  glutSwapBuffers();

  if (bTimer)
  {
    FramesCount++;
    char Title[80];
    if (GlutGet(GLUT_ELAPSED_TIME)!=StartTick)
    {
      sprintf(Title, "FPS=%f",
float(FramesCount)*1000(GlutGet(GLUT_ELAPSED_TIME)-StartTick));
      glutSetWindowTitle(Title);
    }
  }
}

...
void KeyboardSpecial(int key, int x, int y)
{
  switch (key)
  {
...
  case GLUT_KEY_F2:
    bTimer=!bTimer;
    if (bTimer)
    {
      OldTick=GlutGet(GLUT_ELAPSED_TIME);
```

```
      StartTick=OldTick;
      FramesCount=0;
// Включаем обработку события бездействия программы функций Idle
      glutIdleFunc(Idle);
      glutMouseFunc(NULL);
      glutMotionFunc(NULL);
    }
    else
    {
      glutMouseFunc(Mouse);
      glutMotionFunc(MouseMotion);
// Выключаем обработку события бездействия программы
      glutIdleFunc(NULL);
    }
    break;
  }
}
// Обработчик события Idle
void Idle()
{
// Увеличиваем угол поворота
       ry+=(GlutGet(GLUT_ELAPSED_TIME)-OldTick)*0.1;
       OldTick=GlutGet(GLUT_ELAPSED_TIME);
       if (rx>360)
              rx-=360;

       glutPostRedisplay();
}
```

Важно отметить, что на компьютерах с мощной видеокартой программа, использующая такой прием, ведет себя агрессивно по отношению к фоновым приложениям, отбирая у них ресурсы процессора. Если говорить более корректно, то ресурсы процессора у фоновых программ будет отбирать драйвер видеокарты, обладающий более высоким приоритетом по сравнению с другими потоками. Однажды я запустил такую программу во время фоновой записи CD-RW-диска, и через полминуты получил сообщение об опустошении буфера. Поэтому в такие программы имеет смысл вставлять ограничение количества кадров в секунду.

1.8. Работа с растровыми шрифтами и использование полноэкранного режима

Вывод растрового символа на экран в библиотеке GLUT осуществляется командой `glutBitmapCharacter`:

```
void glutBitmapCharacter(void *font, int character);
```

где:

- `font` — идентификатор шрифта. В GLUT имеется пять встроенных растровых шрифтов:
  - GLUT_BIMAP_8_BY_13;
  - GLUT_BIMAP_8_BY_15;
  - GLUT_BITMAP_TIMES_ROMAN_10;
  - GLUT_BITMAP_TIMES_ROMAN_24;
  - GLUT_BITMAP_HELVETICA_10.
- `character` — код символа.

Для вычисления ширины символа в пикселах используется команда `glutBitmapWidth`:

```
int glutBitmapWidth(GLUTbitmapFont font, int character);
```

где:

- `font` — идентификатор шрифта;
- `character` — код символа.

В библиотеке GLUT нет функции вычисления высоты символа, хотя, с другой стороны, в этом нет особой необходимости — в конце названия идентификатора каждого шрифта всегда указывается высота символов.

У всех шрифтов библиотеки GLUT есть существенный недостаток — они не поддерживают кириллицу. Если в вашем приложении необходима поддержка русского языка, то вам придется использовать команду OpenGL `wglUseFontBitmap`, которая доступна только в операционных системах семейства Win32.

Далее частично приведен пример полноэкранного приложения (расширенный вариант предыдущего примера). Программа показывает объект (чайник) и выводит его название и FPS, используя растровые шрифты (Ex09). Наиболее важные фрагменты исходного кода программы приведены в листинге 1.9.

Переключение в полноэкранный режим будет осуществляться командой `glutFullScreen`:

```
void glutFullScreen(void);
```

Хотя переключение в полноэкранный режим осуществляется очень просто, выход из него производится отнюдь не тривиально. Дело в том, что на некоторых компьютерах программа, находясь в полноэкранном режиме, не может корректно завершить работу, после чего привести Windows в чувство можно только путем смены видеорежима "вслепую", либо кнопкой <Reset>. Для того чтобы программа могла работать на максимально широком круге компьютеров, перед выходом из программы необходимо восстанавливать оконный режим. Он восстанавливается при изменении координат или размера окна. Это можно сделать, воспользовавшись командой `glutReshapeWindow`. Но эта команда изменяет размер окна не сразу после выполнения. Вместо этого она помещает в очередь сообщений окна сообщение о необходимости изменения размеров окна (в Win32 это команда `WM_SIZE`). Поэтому выход из полноэкранного режима гарантированно произойдет в обработчике `glutReshapeFunc`. Следовательно, выход из программы должен осуществляется именно в этом обработчике.

Листинг 1.9

```
...
GLboolean bExit=false; // Выход из программы
...
// Определяет ширину строки
int StringWidth(void* font, char* str)
{
  int Width=0;
  char *c=str;
  while (*c)
  {
    Width+=glutBitmapWidth(font, *c);
    c++;
  }
  return Width;
};

// Выводит строку на экран
void PrintString(void* font, char* str)
{
  char* c=str;
  while (*c)
  {
    glutBitmapCharacter(font, *c);
```

```
    c++;
  }
}

void Display() // Обновление содержимого экрана
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  glPushMatrix();
  glTranslatef(tx,ty,tz); // Перемещение и поворот объекта
  glRotatef(rx,1,0,0);
  glRotatef(ry,0,1,0);

  glCallList(list); // Вывод объекта на экран
  glPopMatrix();

// Установка матриц проекции и модели для вывода текста
  glMatrixMode(GL_PROJECTION);
  glPushMatrix();
  glLoadIdentity();
  gluOrtho2D(0, WinWidth, 0, WinHeight);

  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
  glLoadIdentity();

  glColor3f(1,1,1);

  // Установка позиции для вывода подписи
  glRasterPos2f((WinWidth-StringWidth(GLUT_BITMAP_TIMES_ROMAN_24,
"Teapot"))2, WinHeight7);
  // Вывод подписи
  PrintString(GLUT_BITMAP_TIMES_ROMAN_24, "Teapot");

  if (bTimer)
  {
    FramesCount++;
    char Title[80];
    if (GetTickCount()!=StartTick)
```

```
    {
      // Расчет FPS
      sprintf(Title, "%.1f", float(FramesCount)*1000(GetTickCount()-
StartTick));

      // Вывод FPS
      glRasterPos2f(WinWidth-StringWidth(GLUT_BITMAP_TIMES_ROMAN_24,
&Title[0])-10, WinHeight-30);
      PrintString(GLUT_BITMAP_TIMES_ROMAN_24, &Title[0]);
    }
  }

  glPopMatrix();
  glMatrixMode(GL_PROJECTION);
  glPopMatrix();
  glMatrixMode(GL_MODELVIEW);

  glutSwapBuffers();
}

void Keyboard(unsigned char key,int x,int y) // Обработка сообщений от
                                             // клавиатуры
{
  switch (key)
  {
    case VK_ESCAPE: // Если нажата клавиша <ESC> - выход
      bExit=true;
      glutReshapeWindow(5,5);
      break;
  }
}
...
void Reshape(int Width, int Height) // Обработка изменения размеров окна
{
  glViewport(0,0,Width,Height);
  WinWidth=Width; // Запомнить новые размеры
  WinHeight=Height;

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45,WinWidthWinHeight,1,100);
```

```
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();

  if (!bExit)
    glutPostRedisplay();
  else
  {
    if (list)
    glDeleteLists(list,1); // Удалить дисплейный список
    exit(0);
  }
}
...
```

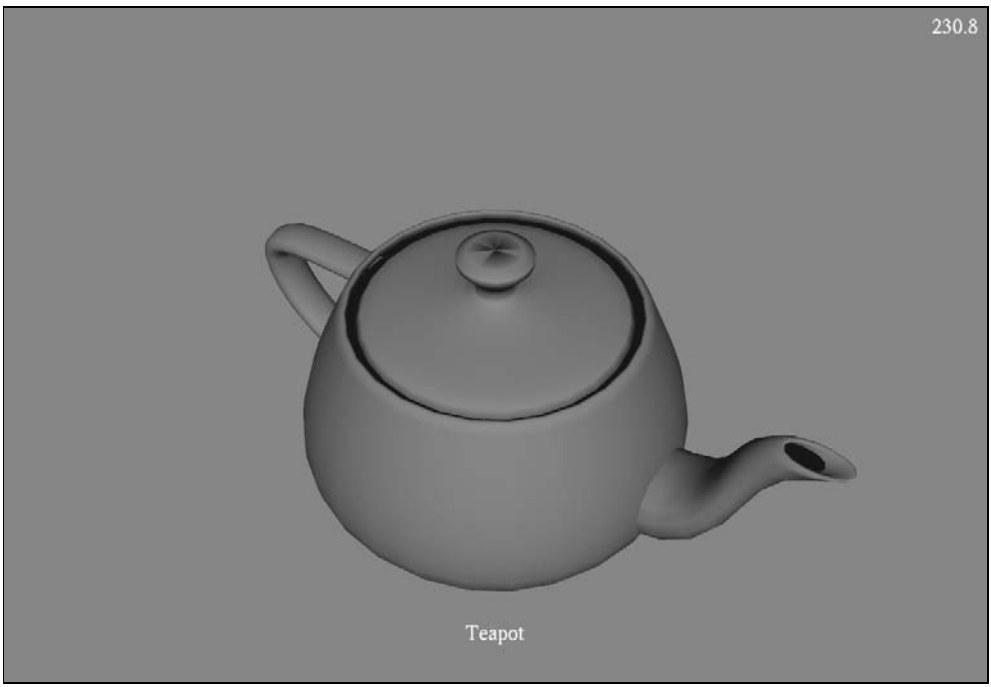


Рис. 1.6. Полноэкранное приложение с растровыми шрифтами

В связи с тем, что при выводе текста используются текущие матрицы проецирования и модели, сложно добиться нужного расположения текста на экране, не меняя эти матрицы в процессе использования перспективной проекции. Поэтому программа сохраняет матрицы проекций в стеке, после

чего устанавливает ортогональную проекцию и выводит текст, а затем восстанавливает старые матрицы.

1.9. Работа с объемными шрифтами

Работа с объемными шрифтами похожа на работу с растровыми шрифтами. Единственное отличие — использование команды `glutStrokeCharacter` вместо `glutBitmapCharacter`:

```
void glutStrokeCharacter(void *font, int character);
```

где:

- `font` — идентификатор, который в случае шрифта переменной ширины равен `GLUT_STROKE_ROMAN`, а для моноширинного — `GLUT_STROKE_MONO_ROMAN`;
- `character` — код символа.

Для определения ширины шрифта в пикселах в библиотеке GLUT имеется команда `glutStrokeWidth`:

```
int glutStrokeWidth(GLUTstrokeFont font, int character);
```

где:

- `font` — идентификатор шрифта;
- `character` — код символа.

Рис. 1.7. Объемный шрифт

К сожалению, эта команда практически бесполезна в реальных программах — размер шрифта может быть изменен при помощи команды `glScale`, а

сам текст может располагаться под углом. Далее приведен пример программы (листинг 1.10) (Ex10), рисующей на экране трехмерную надпись "Hello All", которую можно вращать вокруг оси, проходящей через середину надписи (рис. 1.7).

Так как надпись вращается вокруг своей середины, нам необходимо вычислить координаты середины надписи (координаты начала + половина длины надписи). Главная проблема в этом случае — найти длину надписи (функция `glutStrokeWidth` не подходит, она вычисляет ширину экрана в пикселах). Наиболее красивое решение — использование матрицы модели OpenGL.

Листинг 1.10

```
// Выводит строку на экран
void PrintString(void* font, char* str)
{
        char* c=str;
        while (*c)
        {
                glutStrokeCharacter(font, *c);
                c++;
        }
}

// Определяет ширину строки
vector3f StringWidth(void* font, char* str)
{
        float width;
        GLfloat ModelviewMatrix0[4][4];
        GLfloat ModelviewMatrix1[4][4];

        glPushMatrix();
        glPushAttrib(GL_ALL_ATTRIB_BITS);

// Сохраняем начальное значение матрицы модели
        glGetFloatv(GL_MODELVIEW_MATRIX, &ModelviewMatrix0[0][0]);
// Запрещаем модификацию экрана
        glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
        glDepthMask(GL_FALSE);
```

```
// Рисуем строку на экране
        PrintString(font, str);
// Получаем новое значение матрицы модели
        glGetFloatv(GL_MODELVIEW_MATRIX, &ModelviewMatrix1[0][0]);
// Определяем ширину строки
        width=(ModelviewMatrix1[3][0]-
ModelviewMatrix0[3][0])/ModelviewMatrix1[0][0];

        glPopAttrib();
        glPopMatrix();
        return width;
};
```

Для начала мы сохраняем текущую матрицу модели. Затем запрещаем вывод пикселов в буферы цвета и глубины. После этого рисуем строку и сохраняем итоговую матрицу модели. Далее все просто: нижняя строка матрицы модели содержит смещения текущий системы координат от начала глобальной системы координат, причем элементы (3, 0), (3, 1) и (3,2) содержат ее смещение вдоль осей *x*, *y* и *z*, а элементы (0, 0), (1, 1) и (2, 2) — коэффициенты масштабирования вдоль осей *x*, *y* и *z*. Команда `glutStrokeCharacter` автоматически смещает центр начала системы координат в позицию, в которую будет выведен следующий символ, поэтому разница между соответствующими элементами матриц, деленная на коэффициент масштабирования, покажет смещение центра координат вдоль соответствующих осей. А это и есть длина строки.

```
// Проводим преобразования
glTranslatef(tx,ty,tz);
glRotatef(rx,1,0,0);
glRotatef(ry,0,1,0);
glScalef(0.015, 0.015, 1);

// Получаем ширину строки
vector3f width=StringWidth(GLUT_STROKE_ROMAN, "Hello All");
// Проводим смещение вдоль оси (чтобы поворот выполнялся вокруг центра
// строки)
glTranslatef(-width/2, 0, 0);
// Выводим строку на экран
PrintString(GLUT_STROKE_ROMAN, "Hello All");
```

1.10. Работа с контекстными меню

Контекстное меню создается командой `glutCreateMenu`:

```
int glutCreateMenu(void (*func)(int value));
```

где:

- `func` — адрес функции-обработчика команд меню, которая вызывается всякий раз, когда пользователь вызывает команду меню;
- `value` — идентификатор выбранного элемента меню, который передается в обработчик команд меню.

Команда `glutCreateMenu` возвращает идентификатор созданного меню.

Элементы меню добавляются командой `glutAddMenuEntry`:

```
void glutAddMenuEntry(char *name, int value);
```

где:

- `name` — название элемента меню;
- `value` — идентификатор элемента меню (используется обработчиком команд меню).

С помощью команды `glutCreateMenu` можно создать несколько меню. По умолчанию активным считается последнее созданное меню, но оно может быть изменено командой `glutSetMenu`:

```
void glutSetMenu(int menu);
```

где:

- `menu` — идентификатор нового активного меню.

Меню может содержать несколько подменю, которые в свою очередь могут содержать свои подменю и т. д. Подменю добавляется к текущему меню командой `glutCreateSubMenu`:

```
void glutAddSubMenu(char *name, int menu);
```

где:

- `name` — название пункта подменю;
- `menu` — идентификатор меню, который встраивается в текущее меню в качестве подменю.

Пользователь вызывает контекстное меню при помощи кнопки мыши, которая задается командой `glutAttachMenu`:

```
void glutAttachMenu(int button);
```

где:

- `button` — идентификатор кнопки мыши, используемой для вызова меню (`GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` или `GLUT_RIGHT_BUTTON`).

Ниже частично приводится пример программы, подобной программе (Ex08), но с возможностью выбора вида объекта, выводимого на экран, через контекстное меню (листинг 1.11, рис. 1.8).

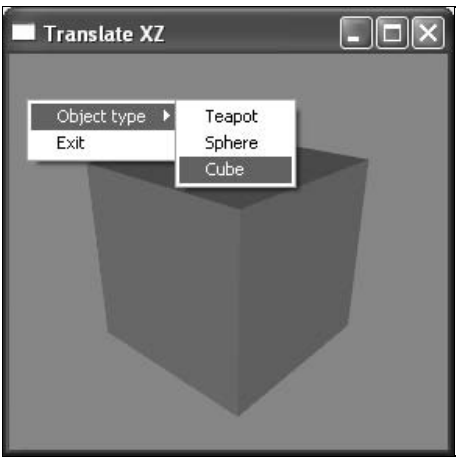


Рис. 1.8. Контекстное меню

Листинг 1.11

```
...
// Идентификаторы команд меню
enum
{
  OTM_TEAPOT, OTM_CUBE, OTM_SPHERE
};
enum
{
  M_EXIT
};
...
void ObjectTypeMenu(int value) // Обработка подменю выбора объекта
{
// Если дисплейный список уже существует — удаляем его
  if (list!=0)
    glDeleteLists(list, 1);
  else
```

```
// В противном случае — получаем идентификатор нового дисплейного списка
    list=glGenLists(1);
// Создаем новый дисплейный список
  glNewList(list,GL_COMPILE);
// Рисуем нужный объект
  switch (value)
  {
    case OTM_TEAPOT:
    {
      glutSolidTeapot(2);
      break;
    }
    case OTM_CUBE:
    {
      glutSolidCube(2);
      break;
    }
    case OTM_SPHERE:
    {
      glutSolidSphere(2, 32, 32);
      break;
    }
  }
  glEndList();

  glutPostRedisplay();
}
// Обработка команд главного меню
void Menu(int value)
{
  switch (value)
  {
    case M_EXIT:
    exit(0);
  }
}

int main(int argc, char* argv[])
{
...
```

```
// Создаем подменю Object type
  int nObjectTypeMenu=glutCreateMenu(ObjectTypeMenu);
  glutAddMenuEntry("Teapot", OTM_TEAPOT); // Чайник
  glutAddMenuEntry("Sphere", OTM_SPHERE); // Сфера
  glutAddMenuEntry("Cube", OTM_CUBE); // Куб

// Создаем главное меню
  glutCreateMenu(Menu);
  // Добавление подменю Object type
  glutAddSubMenu("Object type", nObjectTypeMenu);
  glutAddMenuEntry("Exit", M_EXIT); // Выход из программы

// Назначаем меню на правую кнопку
  glutAttachMenu(GLUT_RIGHT_BUTTON);

// Выполняем команду рисования чайника
  ObjectTypeMenu(OTM_TEAPOT);
  glutMainLoop();

  return 0;
}
```

1.11. Использование режима GameMode

При создании приложений реального времени вы рано или поздно столкнетесь с проблемой недостаточного быстродействия 3D-ускорителя. Одна из причин этого заключается в том, что приложения GLUT по умолчанию используют видеорежим рабочего стола, который часто устанавливается равным 1024 × 738:32 bpp и выше. Однако большинство относительно старых 3D-ускорителей резко теряют производительность вывода трехмерной графики при установке разрешения экрана более 1024 × 768.

Поэтому в библиотеке GLUT имеется специальный режим — GameMode, в котором программа может изменять видеорежим. Плата за скорость в режиме GameMode — невозможность использовать контекстные меню библиотеки GLUT.

Описание команд режима GameMode отсутствует в документации по GLUT, поэтому дополнительную информацию по ним можно найти только в исходном коде GLUT и примерах, поставляемых с этой библиотекой.

Перед переходом в режим GameMode приложение с помощью команды `glutGameModeString` задает требуемое разрешение экрана.

```
void glutGameModeString(const char *string);
```

где:

- `string` — строка, формата "Ширина экрана × Высота экрана : Глубина цвета @ Частота обновления экрана". Если частота обновления экрана не указана, она выбирается библиотекой GLUT исходя из параметров монитора.

Переход в режим GameMode осуществляется командой `glutEnterGameMode`.

```
int glutEnterGameMode(void);
```

После этого приложение обязано выполнить повторную инициализацию OpenGL (настройка параметров освещения, матриц модели и проекции и т. д.), т. к. при смене видеорежима изменяется идентификатор окна. Выход из GameMode осуществляется с помощью команды `glutLeaveGameMode`:

```
void glutLeaveGameMode(void);
```

Дополнительная информация

Необходимость повторной инициализации окна программы при переходе в режим GameMode связана с тем, что при переходе в режим GameMode библиотека GLUT на самом деле создает новое полноэкранное окно, а старое окно становится неактивным, продолжая существовать. При выходе из GameMode старое окно вновь становится активным, а созданное полноэкранное окно уничтожается. Поэтому для более рационального использования видеопамяти имеет смысл сразу при запуске переводить приложение в режим GameMode, без создания главного окна, т. к. в этом случае будет создано только одно полноэкранное окно.

В приведенном ниже фрагменте кода показывается, как можно сразу перейти в режим GameMode при запуске программы (Ex12):

```
int main(int argc, char* argv[])
{
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
// Задаем видеорежим
        glutGameModeString("1024x768:85");
// Переходим в режим GameMode
        glutEnterGameMode();

        Init();
...
}
```

Однако в реальных приложениях часто возникает необходимость позволить пользователю самому изменять видеорежим в процессе работы программы. Реализация такой возможности средствами GLUT демонстрируется в следующей программе (листинг 1.12) (Ex13).

Частота и разрешение экрана устанавливаются из контекстного меню (строка режима для команды `glutGameModeString` выводится в заголовок окна) (рис. 1.9). Вход и выход из GameMode осуществляется клавишей <пробел>. При этом надо помнить, что в режиме GameMode контекстное меню не работает, поэтому параметры экрана надо устанавливать до перехода в режим GameMode.

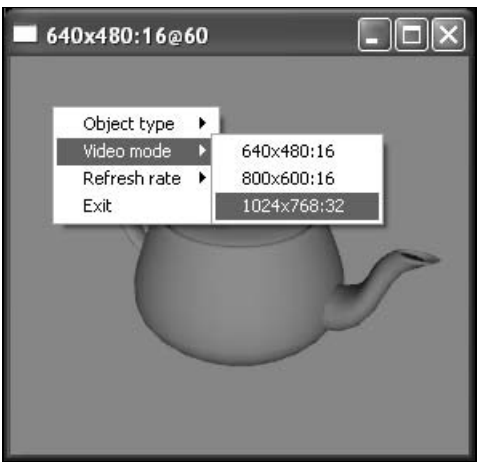


Рис. 1.9. Выбор параметров режима GameMode

Листинг 1.12

```
...
// Список видеорежимов
string VMMStr[]={"640x480:16", "800x600:16", "1024x768:32"};
// Текущий видеорежим
int CurrentVMM=0;

// Список частот вертикальной развертки
string RMStr[]={"60", "70", "75", "85"};
// Текущая частота
int CurrentRM=0;
...
```

```
void Keyboard(unsigned char key,int x,int y) // Обработка сообщений от
                                             // клавиатуры
{
  switch (key)
  {
    case VK_ESCAPE:    // Если нажата клавиша <ESC> - выход
      if (list)
         glDeleteLists(list,1); // Удалить дисплейный список
      exit(0);
      break;
// Если нажата клавиша <SPACE>
    case VK_SPACE:
      if (bGameMode)
// Выходим из режима GameMode
        glutLeaveGameMode();
      else
      {
// Задаем параметры режима GameMode
glutGameModeString((VMMStr[CurrentVMM]+"@"+RMStr[CurrentRM]).c_str());
// Переходим в режим GameMode
        glutEnterGameMode();
// Повторно инициализируем OpenGL
        Init();
      }
      bGameMode=!bGameMode;
      break;
  }
}
...
// Обработчик подменю Video Mode
void VideoModeMenu(int value)
{
  CurrentVMM=value;
// Обновляем заголовок окна
  glutSetWindowTitle((VMMStr[CurrentVMM]+"@"+RMStr[CurrentRM]).c_str());}
// Обработчик подменю Refresh Rate
void RefreshRateMenu(int value)
{
  CurrentRM=value;
```

```
// Обновляем заголовок окна
  glutSetWindowTitle((VMMStr[CurrentVMM]+"@"+RMStr[CurrentRM]).c_str());
}
 ...
int main(int argc, char* argv[])
{
  glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
  glutInitWindowSize(WinWidth,WinHeight);
  glutInitWindowPosition((GetSystemMetrics(SM_CXSCREEN)-WinWidth)2,
(GetSystemMetrics(SM_CYSCREEN)-WinHeight)2);
  glutCreateWindow((VMMStr[CurrentVMM]+"@"+RMStr[CurrentRM]).c_str());

  Init();

// Создаем подменю Object Type
  int nObjectType=glutCreateMenu(ObjectTypeMenu);
  glutAddMenuEntry("Teapot", OTM_TEAPOT);
  glutAddMenuEntry("Sphere", OTM_SPHERE);
  glutAddMenuEntry("Cube", OTM_CUBE);

// Создаем подменю Video Mode
  int nVideMode=glutCreateMenu(VideoModeMenu);
  int i;
  for (i=0; i<sizeof(VMMStr)sizeof(string); i++)
  {
    glutAddMenuEntry(VMMStr[i].c_str(), i);
  }

// Создаем подменю Refresh Rate
  int nRefreshRate=glutCreateMenu(RefreshRateMenu);
  for (i=0; i<sizeof(RMStr)sizeof(string); i++)
  {
    glutAddMenuEntry(RMStr[i].c_str(), i);
  }

// Создаем главное меню
  glutCreateMenu(Menu);
  glutAddSubMenu("Object type", nObjectType);
  glutAddSubMenu("Video mode", nVideMode);
```

```
  glutAddSubMenu("Refresh rate", nRefreshRate);
  glutAddMenuEntry("Exit", M_EXIT);

  glutAttachMenu(GLUT_RIGHT_BUTTON);

  glutMainLoop();

  return 0;
}
```

Поработав некоторое время с режимом GameMode, я пришел к выводу, что он еще достаточно "сырой". Так, например, в нем нет средств для получения списка поддерживаемых видеорежимов, а без этой функции нельзя создать полноценную программу. Как мне кажется, режим GameMode лучше всего использовать только в том случае, если вы хотите сделать свою программу полностью переносимой. В противном случае для изменения видеорежима лучше использовать средства ОС.

1.12. Корректное завершение работы программы при использовании GLUT

Во время анализа предыдущих примеров у вас мог возникнуть вопрос — почему я уничтожаю все объекты, созданные программой, только в обработчике клавиши <Esc>. Ведь если пользователь закроет программу, используя средства Windows (щелчок по крестику в углу окна или по команде меню **Close**), то в этом случае произойдет утечка памяти.

Существует три варианта решения этой проблемы:

1. Создание пользовательского класса, в деструкторе которого расположен код, уничтожающий все временные объекты, созданные приложением. После этого необходимо создать глобальный объект этого класса (C++ гарантирует, что при "раскрутке" стека для каждого объекта, размещенного в стеке, будет вызван деструктор).
2. Использование функции `atexit()`, которая позволяет указать функцию, которая будет выполнена при завершении работы.
3. Внесение изменений в исходный код функции `glutMainLoop()`, добавляющих возможность перехвата события уничтожения окна. Но это крайний способ, т. к. идя на этот шаг, вы рискуете тем, что ваша программа может оказаться несовместимой с будущими версиями GLUT. Поэтому мы не будем рассматривать этот способ в данной книге. Подробнее этот способ описан на сайте **www.opengl.org**.

Кажется, все очень просто. Но если вы попробуете реализовать какой-нибудь из этих вариантов в своей программе, вы столкнетесь с проблемой — они не работают в Win32. Это очень серьезная проблема. Фактически системы GLUT нарушают стандарт C++: при выходе из программы средствами операционной системы все объекты, находящиеся в стеке, не уничтожаются, а обработчик функции `atexit()` не вызывается, что может привести к довольно серьезным последствиям. Эта ошибка не позволяет создавать сложные оконные приложения, используя GLUT, и ограничивает область ее применения небольшими демонстрационными программами.

Чтобы убедиться в наличии этой ошибки, вы можете поработать с приведенным ниже примером (листинг 1.13) (Ex14).

Листинг 1.13

```
class CExit
{
       public:
              ~CExit() {     MessageBox(0, "~Cexit", "Message", MB_OK);}
};

CExit MyExit;

void MyExitProc(void)
{
       MessageBox(0, "MyExitProc", "Message", MB_OK);
}

// ……
void Keyboard(unsigned char key, int x, int y)
{
       switch(key)
       {
              case 27:
                     {
                            exit(VK_ESCAPE);
                            break;
                     }
       }
}
```

```
int main(int argc, char* argv[])
{
        glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
        glutInitWindowSize(WinWidth, WinHeight);
        glutInitWindowPosition(100, 100);
        glutCreateWindow("My first OpenGL Application");

        glClearColor(0.5, 0.5, 0.75, 1);

        glutDisplayFunc(Display);
        glutKeyboardFunc(Keyboard);

        atexit(MyExitProc);
        glutMainLoop();

        return 0;
}
```

Если вы завершите работу этой программы, нажав клавишу <Esc>, все произойдет в соответствии со стандартом C++: на экране поочередно появятся два диалоговых окна. Но если вы попробуете завершить работу, используя средства Windows, то увидите, что никаких сообщений на экране не появляется — т. е. программа завершает свою работу некорректно.

Из этого следует, что при работе в операционных системах семейства Win32 не имеет смысла использовать оба вышеуказанных способа; добавив простое уничтожение объектов в обработчик нажатия клавиши ESC, мы получим аналогичный результат.

1.13. Пример пользовательского интерфейса для GLUT-программ с использованием Borland Delphi 6

Хотя GLUT имеет некоторые средства для организации интерфейса программ, они очень ограничены. Чтобы выйти за рамки этих ограничений, можно воспользоваться стандартными средствами Visual C++. Но создавать интерфейс пользователя с помощью средств WinAPI очень тяжело: в настоящее время на рынке не существует среды программирования, которая бы превосходила все другие по всем параметрам. Поэтому было бы глупо

стрелять из пушки по воробьям — писать на Visual C++, который предназначен для решения задач системного программирования, сложный интерфейс пользователя. По крайней мере, это нерационально в случае небольшой команды разработчиков. С другой стороны, существует Borland Delphi, который сочетает в себе простоту Visual Basic и почти не уступает по эффективности кода (а кое-где и превосходит) Visual C++. Кроме того, для Borland Delphi в настоящее время существует огромное количество компонентов, значительно облегчающих создание пользовательского интерфейса.

Но для начала нам надо решить, как использовать код Borland Delphi в Visual C++. Наиболее оптимальный способ — использовании библиотек DLL, которые не привязаны к среде программирования, в которой они созданы.

Далее приведен пример, который демонстрирует использование формы выбора цвета (рис. 1.10), созданной в Delphi, в Visual C++ (Ex15).

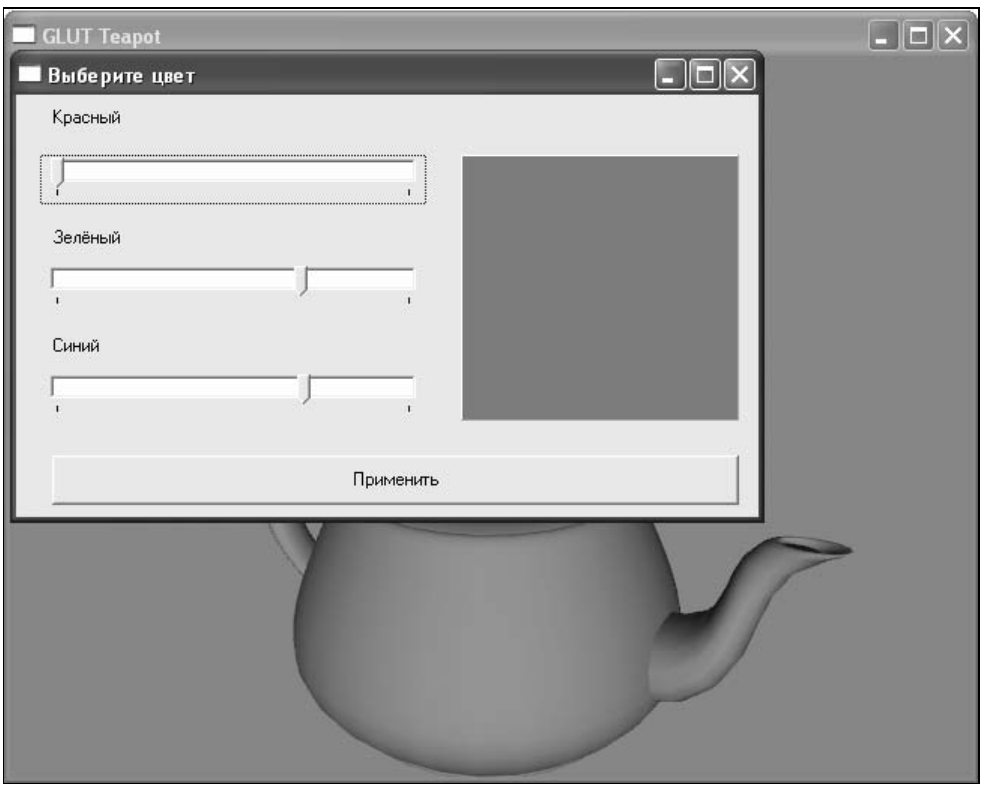


Рис. 1.10. Панель управления, написанная на Borland Delphi

Рассмотрим процесс создания такой программы. Для начала надо создать в Delphi проект библиотеки DLL и добавить в нее форму. Затем после создания тривиальных обработчиков для формы надо создать и экспортировать из DLL процедуру `GetColor`, которая принимает цвет, создает форму и устанавливает в ней этот цвет в качестве текущего, а затем, после выбора цвета пользователем, уничтожает форму и заносит новый цвет в те же переменные, в которых хранились исходные цвета (листинг 1.14).

Листинг 1.14

```
procedure GetColor(var r, g, b:byte);stdcall;
var
 SelectColorForm:TSelectColorForm;
begin
// Создаем форму
 SelectColorForm:=TSelectColorForm.Create(nil);
// Устанавливаем текущий цвет
 SelectColorForm.tbR.Position:=r;
 SelectColorForm.tbG.Position:=G;
 SelectColorForm.tbB.Position:=B;
// Показываем форму на экране
 SelectColorForm.ShowModal();
// Сохраняем полученный цвет
 r:=SelectColorForm.tbR.Position;
 g:=SelectColorForm.tbG.Position;
 b:=SelectColorForm.tbB.Position;
// Уничтожаем форму
 SelectColorForm.Free;
end;

exports
 GetColor;
...
```

Теперь можно переходить к написанию главной программы на Visual C++ (листинг 1.15). В первую очередь необходимо создать обычное GLUT-приложение, выводящее на экран изображение чайника и содержащее контекстное меню с командой **Select Color**. В обработчике команды меню **Select Color** сначала надо вызвать функцию Win32 API `LoadLibrary`, которая загру-

зит библиотеку DLL в память. Затем с помощью функции `GetProcAddress` надо получить адрес функции `GetColor`, после чего вызываем эту функцию и выгружаем ставшую ненужной библиотеку DLL из памяти. После, разумеется, необходимо перерисовать окно командой `glutPostRedisplay`.

Листинг 1.15

```
void SelectColor()
{
// Указатель на функцию GetColor
       void (WINAPI* GetColor) (BYTE&, BYTE&, BYTE&);
// Дескриптор DLL
       HINSTANCE hDLL;

// Загружаем DLL
       hDLL=LoadLibrary("Delphi.dll");
// Если не удалось загрузить DLL, выводим сообщение об ошибке
       if (hDLL==NULL)
       {
              glutSetWindowTitle("Can not find Delphi.dll");
              return;
       }

// Получаем адрес функции GetColor
       GetColor=(void(WINAPI*) (BYTE&, BYTE&,
BYTE&))GetProcAddress(hDLL, "GetColor");
// Если не получилось найти такую функцию, выводим сообщение об ошибке
       if (GetColorProc==NULL)
       {
              glutSetWindowTitle("Can not find GetColor Proc in
Delphi.dll");
              return;
       }
// Вызываем функцию GetColor
       GetColor(Color[0], Color[1], Color[2]);
// Выгружаем DLL из памяти
       FreeLibrary(hDLL);
}
```

```
void Menu(int value)  // Обработка команд главного меню
{
        switch (value)
        {
...
                case M_COLOR:  // Команда Select Color
                        SelectColor();
                        glutPostRedisplay();
                        break;
        }
}
```

Но у полученной программы есть один существенный недостаток. Дело в том, что диалоговое окно выбора цвета объекта является модальным, в результате чего оно полностью блокирует работу программы. В этой связи программа перестает реагировать на события Windows (включая событие `WM_PAINT`), что, в свою очередь, может привести к искажению содержимого окна программы.

Поэтому в GLUT-программах оптимальнее всего использовать немодальные окна. Но для этого нам надо научиться организовывать в GLUT-программе обработку внутренних событий библиотеки DLL. Здесь удобнее всего воспользоваться классическим подходом — использовать функции обратного вызова. Для этого мы при инициализации DLL передаем ей адрес процедуры-обработчика события, а DLL при наступлении этого события вызывает ее.

В приведенной ниже программе (листинг 1.16) (Ex16) показано, как использовать этот прием. На этот раз в ней используется немодальное окно. При изменении цвета в этом окне происходит автоматическое изменение цвета объекта.

Для этого в библиотеке DLL определяются четыре функции:

- `Init` — инициализация и вывод формы на экран;
- `Free` — уничтожение формы;
- `Show` — показ формы;
- `SetProc` — установка обработчика.

Листинг 1.16

```
...
procedure Init(var r, g, b:byte);stdcall;
begin
```

```
 if (SelectColorForm=nil) then
  SelectColorForm:=TSelectColorForm.Create(nil);
 SelectColorForm.tbR.Position:=r;
 SelectColorForm.tbG.Position:=G;
 SelectColorForm.tbB.Position:=B;
 SelectColorForm.Show();
end;

procedure SetProc(Proc:SetColorProc);stdcall;
begin
 SelectColorForm.SetColor:=Proc;
end;

procedure Show();
begin
 if SelectColorForm<>nil then
  SelectColorForm.Show();
end;

procedure Free;stdcall;
begin
 if SelectColorForm=nil then
 begin
  SelectColorForm.Free;
  SelectColorForm:=nil;
 end;
end;

exports
 Init,
 Free,
 SetProc,
 Show;

begin
 SelectColorForm:=nil;
end.
```

Обработчик события описан следующим образом:

```
type SetColorProc=procedure (r, g, b:byte);stdcall;
...
SetColor:SetColorProc;
```

Вызов этого обработчика происходит при изменении цвета:

```
procedure TSelectColorForm.tbRChange(Sender: TObject);
begin
 Panel1.Color:=RGB(tbR.Position, tbG.Position, tbB.Position);
 if @SetColor<>nil then
  SetColor(tbR.Position, tbG.Position, tbB.Position);
end;
```

Код проекта Visual C++ изменился не очень сильно (листинг 1.17). Теперь из DLL импортируются четыре функции, а выгрузка DLL из памяти происходит только по окончании работы с программой. После загрузки DLL ей через функцию `SetProc` передается адрес обработчика событий, а реакция на изменение цвета происходит уже в этом обработчике.

Листинг 1.17

```
...
// Тип обработчика указателя на обработчик событий из DLL
typedef void(WINAPI* SetColorProc) (BYTE, BYTE, BYTE);
// Указатели на функции
void (WINAPI* DelphiInit) (BYTE&, BYTE&, BYTE&);
void (WINAPI* DelphiFree) ();
void (WINAPI* DelphiProc) (SetColorProc FProc);
void (WINAPI* DelphiShow) ();

// Дескриптор DLL
HINSTANCE hDLL=NULL;
...
// Обработка сообщений от клавиатуры
void Keyboard(unsigned char key,int x,int y)
{
       switch (key)
       {
// Если нажата клавиша <ESC> - выход
              case VK_ESCAPE:
```

```
// Если был создан дисплейный список - удалить его
                        if (list)
                                glDeleteLists(list,1);
// Удалить форму
                        DelphiFree();
// Выгрузить DLL
                        FreeLibrary(hDLL);
                        exit(0);
                        break;
        }
}
...
// Обработчик изменения цвета
void WINAPI SetColor(BYTE r, BYTE g, BYTE b)
{
        Color[0]=r;
        Color[1]=g;
        Color[2]=b;
        glutPostRedisplay();
}
// Показывает окно выбора цвета
void SelectColor()
{
// Если DLL еще не загружена
        if (hDLL==NULL)
        {
// Загружаем библиотеку
                hDLL=LoadLibrary("Delphi.dll");
                if (hDLL==NULL)
                {
                        glutSetWindowTitle("Can not find Delphi.dll");
                        return;
                }
// Получаем адрес функции DelphiInit
                DelphiInit=(void(WINAPI*) (BYTE&, BYTE&, BYTE&))
GetProcAddress(hDLL, "Init");
                if (DelphiInit==NULL)
                {
                        glutSetWindowTitle("Can not find Init Proc
in Delphi.dll");
```

```
                return;
            }

// Получаем адрес функции DelphiFree
            DelphiFree=(void(WINAPI*)()) GetProcAddress(hDLL, "Free");
            if (DelphiFree==NULL)
            {
                glutSetWindowTitle("Can not find GetColor Proc in
Delphi.dll");
                return;
            }

// Получаем адрес функции DelphiProc
            DelphiProc=(void(WINAPI*)(SetColorProc FProc))
GetProcAddress(hDLL, "SetProc");
            if (DelphiProc==NULL)
            {
                glutSetWindowTitle("Can not find SetProc Proc in
Delphi.dll");
                return;
            }

            DelphiShow=(void(WINAPI*)())GetProcAddress(hDLL, "Show");
            if (DelphiShow==NULL)
            {
                glutSetWindowTitle("Can not find Show Proc
in Delphi.dll");
                return;
            }
// Создаем окно цвете Color
            DelphiInit(Color[0], Color[1], Color[2]);
// Задаем функцию обратного вызова
            DelphiProc((SetColorProc) SetColor);
       }
       else
       {
// Если DLL уже была загружена и форма была создана, то просто показываем
// ее на экране
        DelphiShow();
       }
```

```
}
...
void Menu(int value)  // Обработка команд главного меню
{
        switch (value)
        {
...
                case M_COLOR:
                        SelectColor();
                        glutPostRedisplay();
                        break;
        }
}
```

1.13.1. Использование статических библиотек DLL, созданных в Delphi 6, в Visual C++

Если библиотека DLL была создана в Visual C++, то проблем с ее подключением к программе в виде статической библиотеки DLL обычно не возникает: достаточно подключить автоматически созданный LIB-файл к проекту с описанием внутри программы списка эскпортируемых функций (используя идентификатор `_declspec(dllexport)`).

Но при использовании библиотеки DLL, созданной сторонними разработчиками, возникает одна нехорошая проблема — отсутствие LIB-файла. А без него в Visual C++, в отличие от того же Delphi, не получится обойтись без неявного связывания. Выход из этой ситуации один — необходимо создать файл-заглушку, который позволил бы программе нормально компилироваться и подключаться. При этом желательно создавать единый заголовочный файл, который можно будет использовать и для создания LIB-файла, и для подключения DLL к проекту. Один из вариантов решения проблемы приведен далее (Ex17). В этом примере показывается, как создать LIB-файл для DLL из примера Ex13.

Создайте новый пустой проект Win32 Dynamic-Link Library и назовите его, к примеру, LIBForDLL. Добавьте в него файл Dephi.h.

Прежде всего добавьте в него строки:

```
#ifndef _DELPHI
#define _DELPHI
...
#endif
```

Это необходимо сделать во избежание ошибок компиляции при повторном включении этого файла в проект.

Теперь вставим в файл функции-заглушки (листинг 1.18). При этом сделаем следующее: если в проекте определена строка _DEPLHI_DLL, то создаются функции-заглушки, иначе — определения функций, необходимые для подключения DLL к проекту. Содержимое функций-заглушек абсолютно не важно, т. к. "суррогатный" LIB-файл необходим только для успешной компоновки проекта. При загрузке программы "суррогатные" функции заменяются функциями из библиотеки DLL.

Листинг 1.18

```
typedef void (_stdcall* SetColorProc) (BYTE r, BYTE g, BYTE b);

void _stdcall Init(BYTE& r, BYTE& g, BYTE& b)
#ifdef _DELPHI_DLL
{
}
#endif
;

void _stdcall SetProc(SetColorProc Proc)
#ifdef _DELPHI_DLL
{
}
#endif
;

void _stdcall Show()
#ifdef _DELPHI_DLL
{
}
#endif
;

void _stdcall Free()
#ifdef _DELPHI_DLL
{
```

```
}
#endif
;
```

Теперь добавьте в проект файл Delphi.cpp и вставьте в него следующий код:

```
#define _DELPHI_DLL
#include "delphi.h"
```

Фактически мы включаем режим генерации функций-заглушек и подключаем файл delphi.h.

Теперь нам надо экспортировать созданные функции. Для этого добавьте в проект файл Delphi.def, содержащий следующий код:

```
LIBRARY delphi
EXPORTS
 Init;
 SetProc;
 Show;
 Free;
```

Вот и все. После компиляции проекта вы получите файл LibForDLL.lib, являющийся "суррогатным кодом" для Delphi.dll.

Теперь надо внести небольшие изменения в проект Ex13 (еще надо не забыть включить в список модулей проекта файл LibForDLL.lib):

```
...
#include "delphi.h"
...
void SelectColor()
{
       Show();
}
...
int main(int argc, char* argv[])
{
...
       Init(Color[0], Color[1], Color[2]);
       SetProc((SetColorProc) SetColor);
}
```

Как видно, код программы стал значительно проще.

Заключение

Библиотека GLUT очень сильно облегчает разработку кроссплатформенных OpenGL-приложений, в результате чего она идеально подходит для создания демонстрационных приложений. И хотя в этой главе мы рассмотрели лишь основные вопросы использования библиотеки GLUT в Visual C++, я думаю, что этих знаний вполне хватит для дальнейшего самостоятельного изучения этой библиотеки по спецификации GLUT. Ее можно найти на прилагаемом к книге компакт-диске.

Глава 2

Библиотека GLH

Библиотека GLUT, рассмотренная в предыдущей главе, является надстройкой, которая облегчает программирование с использованием OpenGL. Но, тем не менее, эта библиотека разрабатывалась для языка C, и не использует новых возможностей C++. В результате большинство программистов используют GLUT в качестве низкоуровневого API.

Поэтому NVIDIA создала собственную объектно-ориентированную надстройку над OpenGL и GLUT, названную OpenGL Helper Library (сокращенно GLH), которая находится в каталоге \NVSDK\OpenGL\include\glh. Эта библиотека содержит множество классов, которые могут значительно облегчить жизнь программисту. Эти классы можно условно разделить на три большие группы:

1. Математические функции.
2. Объектно-ориентированная надстройка над GLUT, основанная на интеракторах.
3. Классы, инкапсулирующие функции OpenGL.

Все классы этой библиотеки расположены в пространстве GLH-имен, поэтому перед их использованием вы должны сделать его активным с помощью команды `using namespace glh`.

Но, к сожалению, эта библиотека поставляется в исходных кодах и без документации. В этой главе я попытаюсь исправить этот недостаток. Мы начнем изучение библиотеки NVIDIA OpenGL Helper Library с группы классов, предназначенных для математических расчетов. Для удобства мы будем называть их библиотекой GLH_LINEAR (по имени заголовочного файла, в котором они расположены).

У библиотеки GLH_LINEAR есть одна особенность. Для того чтобы избавить пользователей библиотеки от утомительного подключения LIB-файлов библиотеки к проекту, создатель библиотеки (Cass Everitt) пошел по пути объявления классов и их реализации с помощью одного и того же файла. Это сильно упрощает написание простых демонстрационных программ. Но при использовании этой библиотеки в многомодульном проекте могут воз-

никнуть проблемы, связанные с тем, что в проекте окажется множество реализаций одной и той же функции. В результате компоновщик не сможет выбрать, какую из реализаций ему использовать, и завершит компоновку с множеством сообщений об ошибках.

Для борьбы с этим явлением необходимо указать в настройках компоновщика **Project | Properties | Linker | Command Line | Additional Options** ключ `/FORCE:MULTIPLE` (рис. 2.1). Этот ключ заставит компоновщик использовать в программе первую попавшуюся реализацию функции и игнорировать остальные.

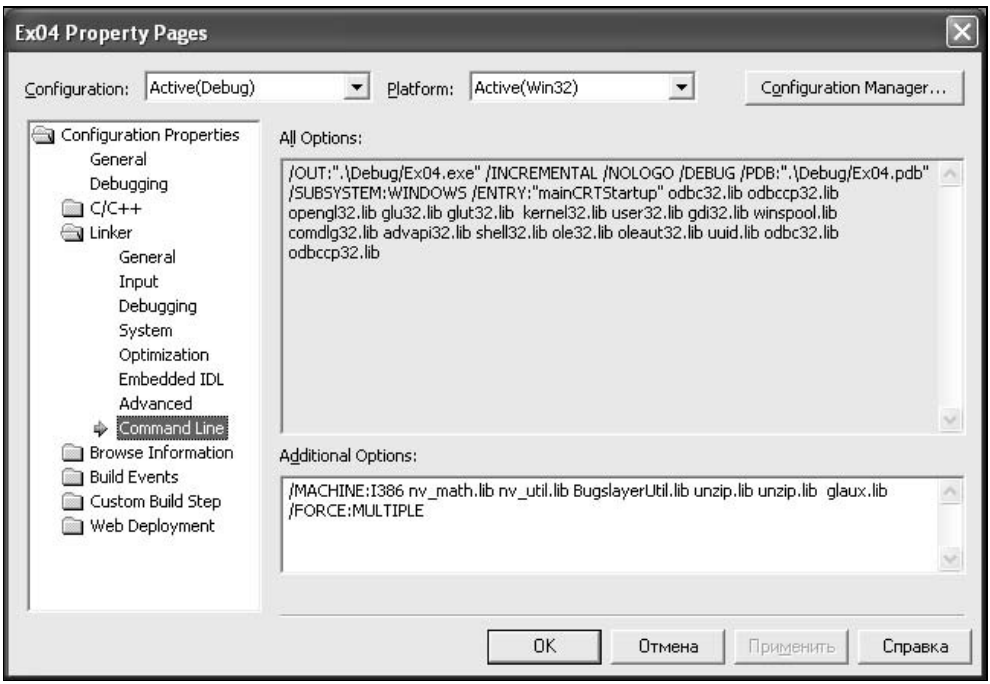


Рис. 2.1. Свойства компановщика

2.1. Математическая библиотека GLH_LINEAR

Во время разработки 3D-приложений программисту часто приходится выполнять одни и те же математические операции — операции с матрицами и векторами, аффинные преобразования и аналогичные операции. Поэтому для облегчения работы можно разработать свою математическую библиоте-

ку, фактически занимаясь изобретением велосипеда, либо воспользоваться готовыми библиотеками, написанными профессионалами. Если вы сторонник второго пути, то этот раздел для вас.

В составе NVIDIA OpenGL SDK имеется математическая библиотека GLH_LINEAR, содержащая множество классов и функций, которые могут серьезно облегчить жизнь программисту. Библиотека GLH_LINEAR не является лучшей ни по возможностям, ни по быстродействию. Но поскольку ее использует 99% примеров NVIDIA OpenGL SDK, знание этой библиотеки может помочь сэкономить вам драгоценное время. Кроме того, библиотека GLH_LINEAR обладает двумя важными достоинствами: она бесплатная и поставляется в исходных кодах.

Для того чтобы подключить эту библиотеку к проекту, вы должны добавить в начало программы следующие строки:

```
// Подключаем библиотеку GLH_LINEAR
#include <glh_linear.h>
// Активизируем пространство имен GLH
using namespace glh;
```

2.1.1. Классы для работы с векторами

Предком всех классов для работы с векторами является шаблонный класс `vec`, объявляющийся следующим образом:

```
template <int N, class T> class vec
{
...
}
```

Параметр `N` задает размерность векторов, а параметр `T` — тип его компонентов.

На основе этого шаблона в GLH определены три класса: `vec2`, `vec3` и `vec4`:

- `class vec2 : public vec<2,real>;`
- `class vec3 : public vec<3,real>;`
- `class vec4 : public vec<4,real>.`

Тип `real` определен следующим образом:

```
# define GLH_REAL float
typedef GLH_REAL real;
```

Следовательно, он является аналогом типа `float`.

Единственное различие между классами `vec2`, `vec3` и `vec4` — в числе параметров конструктора. Все эти три класса предназначены для "внутреннего исполь-

зования" и поэтому определены в пространстве имен GLH_REAL_NAMESPACE. Поэтому данные классы повторно переопределяются в пространстве GLH-имен:

```
typedef GLH_REAL_NAMESPACE::vec2 vec2f
typedef GLH_REAL_NAMESPACE::vec3 vec3f
typedef GLH_REAL_NAMESPACE::vec4 vec4f
```

Для начала мы рассмотрим конструкторы этих классов на примере класса vec4. Все сказанное далее верно и для классов vec2 и vec3, за исключением того, что они имеют меньшее количество параметров.

Класс vec4 имеет пять конструкторов, определенных следующим образом:

```
// Создает экземпляр класса на основе ссылки на массив чисел
vec4 (const real & t = real()) : vec<4,real>(t) {}
// Создает экземпляр класса на основе указателя на массив чисел
vec4 (const real * tp) : vec<4,real>(tp) {}
// Создает экземпляр класса на основе другого объекта
vec4 (const vec<4,real> & t) : vec<4,real>(t) {}
// Создает экземпляр класса на основе 3-мерного
// вектора и 4-го компонента
vec4 (const vec<3,real> & t, real fourth)
{ v[0] = t.v[0]; v[1] = t.v[1]; v[2] = t.v[2]; v[3] = fourth; }
// Создает 4-мерный вектор с заданными компонентами
vec4 (real x, real y, real z, real w)
{ v[0] = x; v[1] = y; v[2] = z; v[3] = w; }
```

Как видно, первые три конструктора используют конструкторы базового класса vec. Для того чтобы продемонстрировать использование этих конструкторов на практике, я создам пять 4-мерных векторов разными способами:

```
float values[4]={1, 2, 3, 4};
float* ptr=&values[0];
vec3f v3dim(1, 2, 3);

vec4f v1(&values[0]);
vec4f v2(ptr);
vec4f v3(v2);
vec4f v4(v3dim, 4);
vec4f v5(1, 2, 3, 4);
```

Этот фрагмент кода присваивает векторам v1, v2, v3, v4 и v5 одинаковое значение {1, 2, 3, 4}.

Для изменения значения вектора используется метод `set_value`:

```
vec4 & set_value ( const real & x, const real & y, const real & z, const real & w)
{ v[0] = x; v[1] = y; v[2] = z; v[3] = w; return *this; }
```

Для получения значений компонентов вектора используется метод `get_value`:

```
void get_value (real & x, real & y, real & z, real & w) const
{ x = v[0]; y = v[1]; z = v[2]; w = v[3]; }
```

Но в большинстве случаев удобнее всего использовать перегруженный оператор `[]`:

```
T & operator [] ( int i ) { return v[i]; }
```

Из определения этого метода видно, что значение вектора хранится в параметре `v`, который объявлен в классе `vec`:

```
T v[N]
```

В классе `vec4` это определение преобразуется в `real v[4]`. Такое определение позволяет использовать объекты семейства классов `vec` в векторных командах OpenGL. К примеру, мы можем установить текущий цвет OpenGL следующим образом:

```
vec3f v(0, 0.75, 0.5);
glColor3fv(&v[0]);
```

Шаблонный класс `vec` содержит ряд полезных методов, которые приведены в табл. 2.1.

Таблица 2.1. *Основные методы класса* `vec`

Определение метода	Назначение
`int size() const`	Возвращает размерность вектора
`T dot( const vec<N,T> & rhs ) const`	Вычисляет скалярное произведение векторов (текущий вектор `dot3 rhs`)
`T length() const`	Вычисляет модуль вектора
`T square_norm() const`	Вычисляет квадрат модуля вектора
`void negate()`	Поворачивает вектор в противоположное направление
`T normalize()`	Нормализует вектор

Кроме того, у каждого из классов, производных от класса `vec`, имеются свои дополнительные методы. Так, например, класс `vec3` умеет вычислять векторное произведение (метод `cross`).

Еще класс `vec4` перегружает практически все математические операторы C++, в результате чего мы можем работать с векторами как с обычными числами.

Для того чтобы продемонстрировать все сказанное выше на практике, напишем программу, решающую простую задачу (Ex01). Пусть у нас имеются два вектора a(1, 2, 3, 4), b(5, 6, 7, 8). Нам необходимо вычислить угол в градусах между векторами (a+b) и (a−b). Исходный код программы, решающей эту задачу приведен ниже. Для того чтобы не усложнять программу, векторы a и b заданы как константы. Исходный текст программ приведен в листинге 2.1.

Листинг 2.1

```
#include <iostream>
#include <string>
#include <glh_linear.h>

using namespace std;
using namespace glh;

void main()
{
        vec3f a(1, 2, 3);     // вектор a
        vec3f b(5, 6, 7);     // вектор b
        vec3f v1;
        vec3f v2;
        float c;

        // Находим v1=a+b и v2=a-b
        v1=a+b;
        v2=a-b;
        // Нормализуем векторы v1 и v2
        v1.normalize();
        v2.normalize();
        // Находим угол между векторами, который равен арккосинусу
        // скалярного произведения нормализованных векторов
        c=to_degrees(acos(v1.dot(v2)));

        cout<<c;

        getchar();
};
```

Эта программа использует классический прием вычисления косинуса угла между векторами — скалярное произведение нормализованных векторов. Для перевода радианов в градусы используется функция `to_degrees` библиотеки GLH. Кстати, в библиотеке GLH также имеется функция для обратного перевода (`to_radians`).

В этом разделе мы рассмотрели лишь основные операции над векторами. Если у вас при работе с библиотекой GLH_LINEAR возникнут вопросы, то все ответы вы сможете найти в файле \NVSDK\OpenGL\include\glh\glh_linear.h.

2.1.2. Класс *line*

Наряду с плоскостями программисту часто приходится иметь дело с бесконечными прямыми. Для работы с последними в библиотеке GLH_LINEAR имеется класс `line`, определенный в пространстве имен `GLH_REAL_NAMESPACE`. В результате, он имеет свой дубликат из пространства имен `glh` — `linef`:

```
typedef GLH_REAL_NAMESPACE::line linef;
```

Этот класс является самым простым из классов библиотеки GLH. Он имеет всего два конструктора, причем первый из них является конструктором по умолчанию:

```
// Создает прямую, совпадающую с осью z
        line ()
        { set_value (vec3 (0,0,0),vec3(0,0,1)); }
// Создает прямую, проходящую через точки p0 и p1
        line ( const vec3 & p0, const vec3 &p1)
        { set_value (p0,p1); }
```

Эти конструкторы настолько тривиальны, что, по-моему, нет необходимости демонстрировать их использование на практике.

Для того чтобы присвоить объекту класса `line` новое значение, используется метод `set_value`:

```
void set_value ( const vec3 &p0, const vec3 &p1)
        {
          position = p0;
          direction = p1-p0;
          direction.normalize();
        }
```

Этот метод создает прямую, проходящую через точки p0 и p1. Из исходного текста этого метода видно, что класс `line` хранит информацию о линии в двух полях:

```
    //protected:
// Точка на линии
        vec3 position;
// Нормализованный вектор направления линии
        vec3 direction;
```

Эта информация может оказаться полезной для быстрого создания линии, когда имеется информация о точке, через которую она проходит, и ее направлении.

Класс `line` имеет всего четыре метода, причем два из них возвращают значения полей `position` и `direction` (табл. 2.2).

Таблица 2.2. *Методы класс* `line`

Определение метода	Назначение
`bool get_closest_points (const line &line2 vec3 &pointOnThis, vec3 &pointOnThat)`	Находит точку на текущей прямой (`pointOnThis`), которая находится ближе всего к прямой `line2`. В `pointOnThat` заносятся координаты аналогичной точки прямой `line2`. Если прямые пересекаются, то функция находит точку пересечения прямых. Если прямые параллельны, то возвращает `false`, в противном случае — `true`
`vec3 get_closest_point (const vec3 &point)`	Возвращает ближайшую точку на прямой, которая находится ближе всего к точке `point`
`const vec3 & get_position() const`	Возвращает точку, через которую проходит прямая
`const vec3 & get_direction() const`	Возвращает нормализованный вектор наплавления прямой

Для демонстрации использования класса `line` на практике я написал небольшую программу, которая находит точку пересечения двух прямых (листинг 2.2) (Ex02):

Листинг 2.2

```
#include <iostream>
#include <string>
#include <glh_linear.h>
```

```
using namespace std;
using namespace glh;

void main()
{
// Первая прямая
       linef line1(vec3f(0, 1, 3), vec3f(0, 2, 5));
// Вторая прямая
       linef line2(vec3f(0, 10, 2), vec3f(0, 7, 3));

       vec3f p1;
       vec3f p2;

// Находим ближайшие точки на обоих прямых
       if (!line1.get_closest_points (line2, p1, p2))
       {
              cout<<"Lines don't cross. They are paralell";
              getchar();
       };
// Если они не совпадают, то прямые не пересекаются
       if ((p1-p2).square_norm()>GLH_EPSILON)
       {
              cout<<"Lines don't cross";
              getchar();
       };

       cout<<"Crossing point: ("<<p1[0]<<", "<<p1[1]<<", "<<p1[2]<<")";
       getchar();
};
```

Идея программы очень проста — при помощи метода `get_closest_points` находятся ближайшие точки на обоих прямых. Если они совпадают, то можно сделать вывод о пересечении прямых в полученной точке. Но мы не можем непосредственно проверять координаты векторов на равенство: из-за погрешности вычислений мы часто будем получать сообщение о непересечении прямых, даже когда на самом деле они пересекаются. Поэтому в программе используется классический прием — сначала находится квадрат модуля вектора, соединяющего две точки. Если он меньше константы `GLH_EPSILON` (10^{-6}), то можно сделать вывод о совпадении точек.

2.1.3. Работа с матрицами

В библиотеке GLH_LINEAR имеются только два класса для работы с матрицами 4×4 — `matrix4` и `matrix4f`. Различие между ними заключается в том, что класс `matrix4` объявлен в пространстве имен `GLH_REAL_NAMESPACE`, а класс `matrix4f` — в пространстве `GLH`:

```
typedef GLH_REAL_NAMESPACE::matrix4 matrix4f;
```

Класс `matrix4` имеет четыре конструктора:

```
// Создает единичную матрицу
    matrix4 () { make_identity(); }
// Делает все элементы матрицы равными r
    matrix4 ( real r ) { set_value (r); }
// Берет все элементы матрицы из одномерного массива на 16 элементов.
```

Внимание

Структура одномерного массива должна быть такой, как требуют команды OpenGL.

```
    matrix4 ( real * m ) { set_value (m); }
//Создает матрицу и присваивает ее элементам соответствующие значения
    matrix4 ( real a00, real a01, real a02, real a03,
              real a10, real a11, real a12, real a13,
                 real a20, real a21, real a22, real a23,
                 real a30, real a31, real a32, real a33 )
       {
              element(0,0) = a00;
              element(0,1) = a01;
              element(0,2) = a02;
              element(0,3) = a03;

              element(1,0) = a10;
              element(1,1) = a11;
              element(1,2) = a12;
              element(1,3) = a13;

              element(2,0) = a20;
              element(2,1) = a21;
              element(2,2) = a22;
              element(2,3) = a23;
```

```
            element(3,0) = a30;
            element(3,1) = a31;
            element(3,2) = a32;
            element(3,3) = a33;
        }
```

Ниже приведен небольшой пример, показывающий использование всех четырех конструкторов на практике:

```
float modelview[16];
// Получаем матрицу модели в одномерный массив
glGetFloatv(GL_MODELVIEW_MATRIX, &modelview[16]);

// Создает единичную матрицу
matrix4f m1;
// Создает матрицу с элементами, равными 5
matrix4f m2(5);
// Создает матрицу, равную матрице модели
matrix4f m3(&modelview[0]);
// Создает матрицу
/*
        1 2 3 4
        5 6 7 8
        9 0 1 2
        3 4 5 6
*/
matrix4f m4(1, 2, 3, 4,
            5, 6, 7, 8,
            9, 0, 1, 2,
            3, 4, 5, 6);
```

Вы, наверное, заметили, что класс `matrix4` не имеет конструктора кодирования для создания копии существующей матрицы. Дело в том, что этот класс, аналогично классу `vec`, перегружает множество операторов, включая равенство. Поэтому в конструкторе кодирования просто нет необходимости.

Для доступа к элементам матрицы используется перегруженный оператор `()`, вызывающий метод `elements`:

```
real & operator () (int row, int col) { return element(row,col); }
```

При помощи этого оператора мы можем работать с матрицей как с обычным массивом (нумерация строк и столбцов идет с нуля). Если же вам не-

обходимо получить одномерный массив из 16 элементов, содержащий элементы матрицы (для матричных команд OpenGL), то можно воспользоваться методом `get_value`:

```
void get_value ( real * mp ) const
{
        int c = 0;
        for(int j=0; j < 4; j++)
                for(int i=0; i < 4; i++)
                        mp[c++] = element(i,j);
}
```

Обратите внимание, что метод `get_value` не выделяет память для хранения элементов массива — об этом заранее должен позаботиться программист. Команда `set_value`, наоборот, присваивает матрице значение матрицы OpenGL, которая хранится в одномерном массиве:

```
void set_value ( real * mp)
```

Кстати, этот же одномерный массив хранится в поле `real m[16]`, поэтому приложение может получить к нему прямой доступ без использования операторов типа `get_value` и `set_value`:

```
//protected:
      real m[16];
```

Правда, закомментированный оператор `protected` недвусмысленно намекает на то, что скоро это поле может перестать быть доступным пользователю.

Класс `matrix4` содержит огромное количество методов, выполняющих различные операции над матрицами, включая некоторые аффинные преобразования (табл. 2.3).

Таблица 2.3. *Основные методы класса* `matrix4`

Определение метода	Назначение
`void set_scale (real s)`	Устанавливает коэффициенты масштабирования матрицы (три коэффициента главной диагонали) равными `s`
`void set_scale (const vec3 & s)`	Устанавливает коэффициенты масштабирования по осям *x*, *y*, *z* равными соответствующим компонентам вектора `s`
`void set_translate ( const vec3 & t)`	Устанавливает коэффициенты переноса (три верхних коэффициента третьего столбца (нумерация идет с нуля)) вдоль осей *x*, *y* и *z* равными соответствующим компонентам вектора `t`
`void set_row (int r, const vec4 & t)`	Копирует вектор `t` в строку с номером `r`

Таблица 2.3 *(окончание)*

Определение метода	Назначение
`void set_column (int c, const vec4 & t)`	Копирует вектор `t` в колонку с номером `c`
`vec4 get_row(int r) const`	Возвращает содержимое строки с номером `r`
`vec4 get_column(int c)`	Возвращает содержимое колонки с номером `c`
`matrix4 inverse() const`	Вычисляет обратную матрицу
`matrix4 transpose() const`	Вычисляет транспонированную матрицу
`matrix4 & mult_right (const matrix4 & b)`	Умножает текущую матрицу на матрицу `b`
`matrix4 & mult_left (const matrix4 & b)`	Умножает матрицу `b` на текущую матрицу
`void mult_matrix_vec (const vec3 &src, vec3 &dst) const`	Умножает матрицу на вектор `src` и помещает результат в вектор `dst`. Вектор `src` предварительно переводится в нормальную форму
`void mult_vec_matrix (const vec3 &src, vec3 &dst) const`	Умножает вектор на матрицу и помещает результат в `dst`
`void mult_vec_matrix (vec3 & src_and_dst) const`	Умножает вектор `src_and_dst` на матрицу и помещает результат в `src_and_dst`
`void mult_matrix_vec (const vec4 &src, vec4 &dst) const`	Умножает матрицу на вектор `src` и помещает результат в вектор `dst`
`void mult_vec_matrix (const vec4 &src, vec4 &dst) const`	Умножает вектор на матрицу и помещает результат в `dst`
`void mult_vec_matrix (vec4 & src_and_dst) const`	Умножает вектор `src_and_dst` на матрицу и помещает результат в `src_and_dst`
`void mult_dir_matrix (const vec3 &src, vec3 &dst) const`	Умножает вектор `src` на подматрицу 3 × 3 (которая получена путем удаления третьей (нумерация идет с нуля) строки и столбца матрицы 4 × 4) и помещает результат в `dst`. Эта подматрица содержит коэффициенты, отвечающие за поворот и масштабирование, но не отвечающие за перенос
`void mult_dir_matrix (vec3 & src_and_dst) const`	Умножает вектор `src_and_dst` на подматрицу 3 × 3 (которая получена путем удаления третьей (нумерация идет с нуля) строки и столбца матрицы 4 × 4) и помещает результат в `src_and_dst`

Для демонстрации возможностей класса `matrix4` я решил написать небольшую программу (листинг 2.3) (Ex03), решающую систему из четырех уравнений по формуле: $X = \begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}^{-1} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$.

Листинг 2.3

```
//#define test

#include <iostream>
#include <string>
#include <glh_linear.h>

using namespace std;
using namespace glh;

void input(matrix4f& a)      //ввод матрицы с клавиатуры
{
        for (unsigned int i=0; i<4; i++)
        {
                for (unsigned int j=0; j<4; j++)
                {
                        cout<<"a["<<i<<","<<j<<"]=";
                        cin>>a(i, j);
                }
        }
}

void input(vec4f& b)  //ввод вектора с клавиатуры
{
        for (unsigned int i=0; i<4; i++)
        {
                cout<<"b["<<i<<"]=";
                cin>>b[i];
        }
}
```

```
void output(vec4f& x) // вывод вектора на экран
{
	for (unsigned int i=0; i<4; i++)
	{
		cout<<"x["<<i<<"]="<<x [i]<<endl;
	}
}

void main()
{
	matrix4f a;		// матрица коэффициентов a00..a22
	vec4f b;		// вектор свободных членов

	vec4f x;		// вектор корней уравнения
/*
	если определена переменная test, то ввод тестового набора данных,
	иначе ввод данных с клавиатуры
*/

#ifndef test
	input(a);
	input(b);
#else
	a.element(0, 0)=2;
	a.element(0, 1)=1;
	a.element(0, 2)=-5;
	a.element(0, 3)=1;

	a.element(1, 0)=1;
	a.element(1, 1)=-3;
	a.element(1, 2)=0;
	a.element(1, 3)=-6;

	a.element(2, 0)=0;
	a.element(2, 1)=2;
	a.element(2, 2)=-1;
	a.element(2, 3)=2;
```

```
        a.element(3, 0)=1;
        a.element(3, 1)=4;
        a.element(3, 2)=-7;
        a.element(3, 3)=6;

        b.v[0]=8;
        b.v[1]=9;
        b.v[2]=-5;
        b.v[3]=0;
#endif

        // умножение обратной матрицы на вектор свободных членов
        a.inverse().mult_matrix_vec (b, x);
        output(x);

#ifndef test
        getchar();
#endif
        getchar();
};
```

Из этого примера видно, что библиотека GLH_LINEAR очень сильно облегчает написание программы — система уравнений решается одним оператором.

В примере демонстрируется полезный прием "вшивания" тестового примера в исходный код программы. Ведь в ходе тестирования довольно утомительно вводить при каждом запуске систему из четырех уравнений. В нашей же программе для переключения программы в "тестовый" режим достаточно раскомментировать директиву `#define test`. Кстати, в качестве тестового примера используется система уравнений из [10]

2.1.4. Кватернионы

Из всех современных русскоязычных книг по 3D-графике кватернионы упоминаются лишь в [2]. Поэтому вполне вероятно, что вы не знакомы с этой темой. Но огорчаться не стоит — при помощи классов библиотеки GLH_LINEAR можно использовать кватернионы, абсолютно не имея представления о них. Но, с другой стороны, программист всегда должен иметь хотя бы поверхностное представление об устройстве "черного ящика", с ко-

торым он работает — это позволяет ему писать более качественный код. Поэтому я начну главу с небольшого эссе о кватернионах, чтобы неподготовленный читатель смог войти в курс дела.

Кватернионы определяют каждое вращение в 4-мерном пространстве. Следовательно, для описания каждого кватерниона необходимо четыре значения: *x*, *y*, *z* и *w*. Координаты *x*, *y* и *z* обычно трактуются как компоненты трехмерного вектора, а *w* — как скаляр. Эти координаты вычисляются на основе трех координат оси вращения и угла поворота по следующими формулами:

```
sin_a = sin(angle / 2);
cos_a = cos(angle / 2);

qx = axisx × sin_a;
qy = axisy × sin_a;
qz = axisz × sin_a;
qw = cos_a,
```

где:

- `axis` — ось поворота;
- `angle` — угол поворота;
- `q` — кватернион.

Полученный кватернион должен быть нормализован. Кватернион называется нормализованным, если его модуль равен 1. Модуль кватерниона вычисляется по формуле

$$|q| = \sqrt{{q_x}^2 + {q_y}^2 + {q_z}^2 + {q_w}^2}\ .$$

Следовательно, для того чтобы нормализировать кватернион, надо поделить все компоненты кватерниона на его модуль:

$$q\_norm_x = \frac{q_x}{|q|};$$

$$q\_norm_y = \frac{q_y}{|q|};$$

$$q\_norm_z = \frac{q_z}{|q|};$$

$$q\_norm_w = \frac{q_w}{|q|}.$$

Наверное, у вас уже появился вопрос: "Зачем все это надо? Почему NVIDIA не могла реализовать аффинное преобразование поворота при помощи мат-

риц?" Дело в том, что кватернионы имеют по сравнению с матрицами ряд преимуществ.

- Кватернион занимает в четыре раза меньше места, чем матрица 4×4 и в два раза меньше чем матрица 3×3.
- Операции над кватернионами выполняются значительно быстрее, чем над матрицами. Например, перемножение кватернионов требует 16 умножений и 12 сложений, а перемножение матриц 3×3 — 27 умножений и 18 сложений.
- И, пожалуй, главное преимущество. Интерполяция кватернионов выполняется очень просто: требуется вычислить две трибометрические функции и выполнить несколько умножений. А вот интерполирование матриц требует выполнения численного интегрирования, а это очень ресурсоемкая задача. Аналогичная ситуация складывается и с нахождением обратного кватерниона.

Но кватернионы не могут непосредственно использоваться в OpenGL, на финальном этапе они должны быть преобразованы в матрицу 4×4.

Хочу еще раз подчеркнуть, что это было лишь небольшое введение в кватернионы, которое не заменит чтения специализированной литературы по математике. В качестве продолжения я хотел бы вам посоветовать прочитать [12]. Правда, этот материал на английском языке. Если же у вас проблемы с английским языком, то можно попробовать прочитать [13].

В библиотеке GLH_LINEAR за работу с кватернионами отвечают классы `quaternion`, `rotation`, `quaternionf` и `rotationf`. Первые два класса объявлены в пространстве имен `GLH_REAL_NAMESPACE`, а остальные — в `GLH`. В действительности, последние три класса являются ни чем иным, как классом `quaternion` с разными именами:

```
typedef quaternion rotation;
...
typedef GLH_REAL_NAMESPACE::quaternion quaternionf;
typedef GLH_REAL_NAMESPACE::quaternion rotationf;
```

Рассмотрение класса `quaternion` мы начнем, как всегда, с конструкторов, которых у него целых семь:

```
    class quaternion
    {
    public:
// Создает кватернион (0, 0, 0, 1)
    quaternion ()
```

```
    {
       q[0]=q[1]=q[2]=0.0;
       q[3]=GLH_ONE;
    }
// Создает кватернион на основе массива из 4-х элементов
    quaternion ( const real v[4] )
    {
        set_value ( v );
    }

// Создает кватернион на основе 4-х чисел
    quaternion ( real q0, real q1, real q2, real q3 )
    {
        set_value ( q0, q1, q2, q3 );
    }

// Создает кватернион на основе матрицы вращения!
    quaternion ( const matrix4 & m )
    {
        set_value ( m );
    }

// Создает кватернион на основе оси вращения и угла
// поворота вокруг этой оси
    quaternion ( const vec3 &axis, real radians )
    {
        set_value ( axis, radians );
    }

// Создает кватернион, который может повернуть
// вектор rotateFrom в направлении вектора rotateTo
    quaternion ( const vec3 &rotateFrom, const vec3 &rotateTo )
    {
        set_value ( rotateFrom, rotateTo );
    }
// Создает кватернион для поворота камеры
// из положения (from_look, from_up) в
// положение (to_look, to_up). Вектор look соответствует
```

```
// направлению взгляда, а up -
// показывает направление "вверх" в окне, т. е. эти параметры
// напоминают параметры команды gluLookAt.
    quaternion ( const vec3 & from_look, const vec3 & from_up,
                    const vec3 & to_look, const vec3& to_up)
    {
           set_value (from_look, from_up, to_look, to_up);
    }
...
  protected:
// Обратите внимание, что доступ к координатам кватерниона
// возможен как к массиву, так и по именам
        union
        {
            struct
            {
                real q[4];
            };
            struct
            {
                real x;
                real y;
                real z;
                real w;
            };
        };
}
```

Далее приведен небольшой пример использования всех этих конструкторов в действии:

```
float vec [4]={0, 0, 0, 1};
matrix4f mat;
glGetFloatv(GL_MODELVIEW_MATRIX, &mat.m[0]);

// Создает кватернион (0, 0, 0, 1)
quaternionf quat1;
// Создает кватернион на основе массива (0, 0, 0, 1)
quaternionf quat2(vec);
// Создает кватернион (0, 0, 0, 1)
```

```
quaternionf quat3(0, 0, 0, 1);
// Создает кватернион на основе матрицы модели (если матрица модели
// содержит операции преноса и масштабирования, то результат может быть
// непредсказуемым)
       quaternionf quat4(mat);
// Создает кватернион поворота на 90 градусов
// вокруг оси-вектора (1, 1, 1)
quaternionf quat5(vec3f(1, 1, 1), to_radians(90));
// Создает кватернион поворота вектора (0, 0, 1) в (1, 0, 1)
quaternionf quat6(vec3f(0, 0, 1), vec3f(1, 0, 1));
// Созлает кватернион поворота камеры (look(0, 0, 1), up(0, 1, 0)) в
// положение (look(1, 0, 1), up(0, 1, 0))
quaternionf quat7(vec3f(0, 0, 1), vec3f(0, 1, 0),
                  vec3f(1, 0, 1), vec3f(0, 1, 0));
```

Доступ к компонентам кватерниона осуществляется аналогично векторам при помощи перегруженного оператора `[]`.

Если вам надо поместить в кватернион новое значение, то вы можете воспользоваться одним из методов `set_value`, которые очень похожи на конструкторы этого класса (вы, наверное, уже обратили внимание, что все конструкторы, кроме конструктора по умолчанию, являются лишь надстройками над соответствующими методами `set_value`).

```
// Создает кватернион на основе 4-х чисел
quaternion & set_value ( real q0, real q1, real q2, real q3 )
// Создает кватернион на основе указателя на массив из 4-х элементов
quaternion & set_value ( const real * qp )
// Создает кватернион на основе матрицы вращения!
quaternion & set_value ( const matrix4 & m )
// Создает кватернион на основе оси вращения и угла поворота
// вокруг этой оси
quaternion & set_value ( const vec3 &axis, real theta )
// Создает кватернион, который может повернуть вектор rotateFrom
// в направлении вектора rotateTo
quaternion & set_value ( const vec3 & rotateFrom, const vec3 & rotateTo )
// Создает кватернион для поворота камеры из положения (from_look,
// from_up) в положение (to_look, to_up). Вектор look соответствует
// направлению взгляда, в свою очередь, up — показывает направление
// "вверх" в окне, т. е. эти параметры напоминают параметры
// команды gluLookAt.
```

```
quaternion & set_value ( const vec3 & from_look, const vec3 & from_up,
                         const vec3 & to_look, const vec3 & to_up)
```

Первые четыре метода `set_value` имеют свои антиподы — `get_value`, которые выполняют обратные задачи:

```
// Возвращает указатель на массив компонентов кватерниона
const real * get_value () const
// Используется для получения компонентов кватерниона
void get_value ( real &q0, real &q1, real &q2, real &q3 ) const
// Преобразует в кватернион связку вектор оси + угол вращения
void get_value ( vec3 &axis, real &radians ) const
// Создает на основе кватерниона матрицу вращения
void get_value ( matrix4 & m ) const
```

Последний метод является очень важным, т. к. все кватернионы перед использованием в OpenGL должны быть преобразованы в матрицы модели.

В классе `quaternion` имеется несколько полезных методов, которые приведены в табл. 2.4.

Таблица 2.4. *Основные методы класса* `quaternion`

Определение метода	Назначение
`void normalize()`	Выполняет нормализацию кватерниона

`bool equals(const quaternion & r, real tolerance) const`	Выполняет сравнение текущего кватерниона с кватернионом `r`. Параметр `tolerance`, по идее, должен задавать порог, по превышению которого кватернионы будут считаться разными. В действительности же код, сравнивающий два кватерниона, выглядит следующим образом:
	`t = ((q[0] - r.q[0]) Ч (q[0] - r.q[0]) +`
	`(q[1] - r.q[1]) Ч (q[1] - r.q[1]) +`
	`(q[2] - r.q[2]) Ч (q[2] - r.q[2]) +`
	`(q[3] - r.q[3]) Ч (q[3]-r.q[3]));`
	`if(t > GLH_EPSILON)`
	`return false;`
	`return 1;`
	Как видно из этого фрагмента, в качестве критерия идентичности двух кватернионов используется константа `GLH_EPSILON`

`quaternion & conjugate()`	Вычислят обратный кватернион — кватернион, который осуществляет поворот в противоположном направлении

Таблица 2.4 *(окончание)*

Определение метода	Назначение
`quaternion & invert()`	Вычислят обратный кватернион — кватернион, который осуществляет поворот в противоположном направлении
`quaternion inverse() const`	Возвращает обратный кватернион, не изменяя текущий
`void mult_vec (const vec3 &src, vec3 &dst) const`	Умножает вектор `src` на кватернион и помещает результат в `dst`. Иными словами поворачивает вектор `src`
`void mult_vec (vec3 & src_and_dst) const`	Умножает вектор `src_and_dst` на кватернион и помещает результат в `src_and_dst`
`void scale_angle (real scaleFactor)`	Масштабирует угол поворота вокруг оси с коэффициентом масштабирования `scaleFactor`
`static quaternion slerp(const quaternion & p, const quaternion & q, real alpha)`	Выполняет линейную интерполяцию между кватернионами `p` и `q`. Если `alpha = 0`, то результат равен кватерниону `p`. Если `alpha = 1`, то `q`. При `0 < alpha < 1` получаются соответствующие промежуточные значения

Кроме того, класс `quaternion` перегружает ряд операторов C++, например `*`, `* =`, `= =`, `! =` и т. д.

Для демонстрации использования класса `quaternionf` на практике я переписал пример Ex06 из *главы 1* без использования матричных функций OpenGL для работы с матрицей модели (Ex04). Большинство изменений затронули только функцию `Display` (листинг 2.4).

Листинг 2.4

```
void Display()                  // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glLoadIdentity();

       matrix4f Modelview;          // Матрица модели
       matrix4f rot_mat;            // Матрица вращения
       matrix4f translate;          // Матрица переноса
// Кватернион поворота вокруг оси x
       quaternionf quat_x(vec3f(1, 0, 0), to_radians(rx));
```

```
// Умножаем на кватернион поворота вокруг оси y
       quat_x * =quaternionf (vec3f(0, 1, 0), to_radians(ry));

// Преобразуем кватернион в матрицу вращения
       quat_x.get_value (rot_mat);

       translate.set_translate(vec3f(tx, ty, tz));       // Перенос
// Создаем матрицу модели
       Modelview = translate*rot_mat;

// Умножаем матрицу модели на текущую матрицу
       glMultMatrixf(&Modelview.m[0]);

       glCallList(list);                    // Вывод объекта на экран

       glutSwapBuffers();

       if (bTimer)
       {
              FramesCount++;
              char Title[80];
              if (GetTickCount() ! = StartTick)
              {
                     sprintf(Title, "FPS=%f",
                            float(FramesCount)*1000 /(GetTickCount() -
StartTick));
                     glutSetWindowTitle(Title);
              }
       }
}
```

Функция `Display` выполняет следующие аффинные преобразования:

1. Перенос вдоль осей *x*, *y*, *z* на величины `tx`, `ty` и `tz`.
2. Поворот вокруг оси *x* на угол `rx`.
3. Поворот вокруг оси *y* на угол `ry`.

Сначала функция вычисляет кватернион поворота вокруг оси *x*:

```
quaternionf quat_x(vec3f(1, 0, 0), to_radians(rx));
```

Затем она умножает полученный кватернион на кватернион поворота вокруг оси *y*. Иными словами, выполняется поворот вокруг оси *y*.

```
Quat_x * = quaternionf (vec3f(0, 1, 0), to_radians(ry));
```

На этом преобразования поворота заканчиваются. Для того чтобы кватернион смог участвовать в дальнейших аффинных преобразованиях, он должен быть преобразован в матрицу:

```
Quat_x.get_value (rot_mat);
```

Теперь нам остается только рассчитать матрицу переноса и умножить ее на матрицу поворота:

```
translate.set_translate(vec3f(tx, ty, tz));
Modelview = translate * rot_mat;
```

Вот мы и получили необходимую нам матрицу модели. Остается последний штрих — умножить текущую матрицу модели OpenGL, которая у нас равна единичной матрице, на полученную матрицу.

```
glMultMatrixf(&Modelview.m[0]);
```

Как видно, матричные преобразования OpenGL сильно упрощают вычисления. Но если по честному, то приведенный выше пример можно было написать и без библиотеки GLH (что мы и сделали в *главе 1*). Но, тем не менее, бывают случаи, когда без расчета матрицы модели вручную невозможно обойтись.

В качестве примера рассмотрим небольшую задачу, которую приходится решать при стыковке нескольких поверхностей (Ex05). Пусть у нас имеются два одинаковых прямоугольника A и B. Прямоугольник B смещается относительно прямоугольника A командами:

```
// t и r - произвольные числа
glTranslatef(1, 0, t);        // Смещаем плоскость на величину t
                              // вверх и 1 вправо
glTranslatef(1, 0, 0);        // Проводим поворот на r
                              // градусов вокруг левого края
glRotatef(r, 0, 1, 0);
glTranslatef(1, 0, 0);
```

Требуется нарисовать прямоугольник C, соединяющий прямоугольники A и B между собой (рис. 2.2).

Начинающий программист скорее всего скажет: "Что же тут сложного?" — и быстро напишет программу, подобную следующей:

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

```
glPushMatrix();
       // Смещаем плоскость на величину t вверх и 1 вправо
       glTranslatef(1, 0, t);
       // Проводим поворот на r градусов вокруг левого края
       glTranslatef(1, 0, 0);
       glRotatef(r, 0, 1, 0);
       glTranslatef(1, 0, 0);

       glBegin(GL_QUADS);
              glVertex2f(-1, -1);
              glVertex2f(-1,  1);
              glVertex2f( 1,  1);
              glVertex2f( 1, -1);
       glEnd();
glPopMatrix();

       //Так нельзя!!!
glBegin(GL_QUADS);
       glVertex2f( 1,  1);
       glVertex2f( 1, -1);

       glPushMatrix();
              glTranslatef(1, 0, -t);
              glTranslatef(1, 0, 0);
              glRotatef(r, 0, 1, 0);
              glTranslatef(1, 0, 0);

              glVertex2f(-1, -1);
              glVertex2f(-1, 1);
       glPopMatrix();
glEnd();
```

Вроде бы все просто, за исключением одной "маленькой" детали — этот фрагмент не будет работать. Дело в том, что в программе производится изменение текущей матрицы модели внутри блока `glBegin() glEnd()`, что делать *нельзя*. В результате команды `glTranslatef` и `glRotatef` внутри последнего блока `glBegin() glEnd()` будут просто проигнорированы.

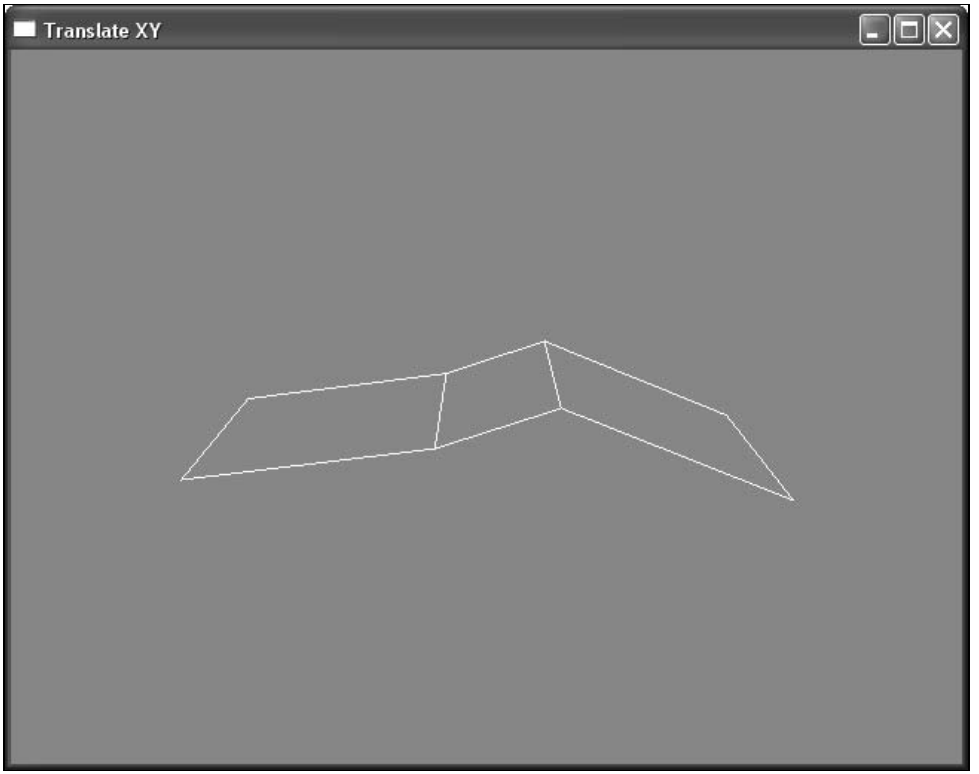


Рис. 2.2. Прямоугольники A, C, B (слева направо)

Единственный выход из данной ситуации — рассчитывать координаты вершин вручную. К счастью, у нас есть библиотека GLH, поэтому задача значительно упрощается. Ниже приведен фрагмент кода, который решает эту задачу (листинг 2.5).

Листинг 2.5

```
void Display()                  // Обновление содержимого экрана
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glPushMatrix();
                // Перемещение и поворот плоскости A
                glTranslatef(tx,ty,tz);
                glRotatef(rx,1,0,0);
                glRotatef(ry,0,1,0);
```

```
// Рисуем прямоугольник A
              glBegin(GL_QUADS);
                     glVertex2f(-1, -1);
                     glVertex2f(-1,  1);
                     glVertex2f( 1,  1);
                     glVertex2f( 1, -1);
              glEnd();

              glPushMatrix();
                     glTranslatef(1, 0, t);
// Смещаем плоскость на величину t вверх и 1 вправо

// Проводим поворот на r градусов вокруг левого края
                     glTranslatef(1, 0, 0);
                     glRotatef(r, 0, 1, 0);
                     glTranslatef(1, 0, 0);
// Рисуем прямоугольник B
                     glBegin(GL_QUADS);
                            glVertex2f(-1, -1);
                            glVertex2f(-1,  1);
                            glVertex2f( 1,  1);
                            glVertex2f( 1, -1);
                     glEnd();
              glPopMatrix();


       // Координаты правой стороны прямоугольника C
              vec4f v1(-1, -1, 0, 1);
              vec4f v2(-1, 1, 0, 1);

              matrix4f Translate1;
              matrix4f Translate2;
              quaternionf RotateQuat(vec3f(0, 1, 0), to_radians(r));
              matrix4f Rotate;

       // Рассчитываем матрицу первого переноса.
       // Обратите внимание — два оператора glTranslate объединены в один
              Translate1.set_translate(vec3f(2, 0, t));
              //Рассчитываем матрицу второго переноса (после поворота)
```

```
            Translate2.set_translate(vec3f(1, 0, 0));
            //Рассчитываем матрицу поворота
            RotateQuat.get_value (Rotate);

            matrix4f m1;

            m1=Translate1*Rotate*Translate2;

            m1.mult_matrix_vec (v1);
            m1.mult_matrix_vec (v2);
// Рисуем прямоугольник C
            glBegin(GL_QUADS);
                  glVertex2f( 1,  1);
                  glVertex2f( 1, -1);
                  glVertex4fv(&v1.v[0]);
                  glVertex4fv(&v2.v[0]);
            glEnd();
      glPopMatrix();

      glutSwapBuffers();



}
```

Как видно, ничего сложного. Плоскость C рисуется в системе координат плоскости A, причем координаты правого края пересчитываются из системы координат плоскости B (координаты левого края плоскости можно не рассчитывать, т. к. они совпадают с координатами правого края плоскости A).

2.1.5. Класс *plane*

При программировании трехмерных приложений программисту постоянно приходится работать с такой математической абстракцией, как бесконечная плоскость. Разумеется, создатели OpenGL Helper Library не могли безучастно смотреть на такое положение вещей. В результате в библиотеку GLH_LINEAR был включен класс `plane`, который абстрагирует понятие математической плоскости.

Аналогично предыдущим классам, класс `plane` имеет дубликат, объявленный в пространстве имен `glh` — `planef`.

```
typedef GLH_REAL_NAMESPACE::plane planef;
```

Класс `plane` имеет четыре конструктора:

```
class plane
  {
  public:
// Создает плоскость XY. Нормаль расположена вдоль положительного
// направления оси Z
         plane ()
      {
                planedistance = 0.0;
                planenormal.set_value ( 0.0, 0.0, 1.0 );
      }

// Создает плоскость, проходящую через 3 точки (p0, p1 и p2)
         plane ( const vec3 &p0, const vec3 &p1, const vec3 &p2 )
      {
                vec3 v0 = p1 - p0;
                vec3 v1 = p2 - p0;
                planenormal = v0.cross(v1);
                planenormal.normalize();
                planedistance = p0.dot(planenormal);
      }

// Создает плоскость на основе нормали и расстояния от плоскости до
// начала координат
         plane ( const vec3 &normal, real distance )
      {
                planedistance = distance;
                planenormal = normal;
                planenormal.normalize();
      }

// Создает плоскость на основе нормали и координат точки, принадлежащей
// плоскости
         plane ( const vec3 &normal, const vec3 &point )
      {
                planenormal = normal;
                planenormal.normalize();
                planedistance = point.dot(planenormal);
      }
```

```
  //protected:
// Нормаль к плоскости
        vec3 planenormal;
// Расстояние от плоскости до начала координат
        real planedistance;
  };
```

Обратите внимание, что плоскость внутри класса `plane` хранится в виде связки "нормаль + расстояние от плоскости до начала координат". Поначалу это может казаться необычным. Но если вы немного подумаете, то обнаружите, что связка нормаль к плоскости и расстояние до начала координат однозначно определяет заданную плоскость.

Для демонстрации использования на практике всех четырех конструкторов далее показано, как создать плоскость *XY* четырьмя способами. Нормаль направлена вдоль положительного направления оси *z*:

```
planef plane0;
planef plane1(vec3f(0, 0, 0), vec3f(7, 0, 0), vec3f(0, 5, 0));
planef plane2(vec3f(0, 0, 1), 0);
planef plane3(vec3f(0, 0, 1), vec3f(5, 15, 0));
```

Класс `plane` имеет множество полезных методов, которые приведены в табл. 2.5.

Таблица 2.5. *Основные методы класса* `plane`

Определение метода	Назначение
`void offset(real d)`	Увеличивает расстояние от плоскости до начала координат на величину `d` путем выполнения параллельного переноса
`bool intersect (const line &l, vec3 &intersection) const`	Находит точку пересечения (interception) текущей плоскости и линии (line). Если линия и плоскость параллельны (не имеют точки пересечения), возвращает `false`
`void transform (const matrix4 &matrix)`	Выполняет над плоскостью аффинное преобразование, заданное матрицей `matrix`
`bool is_in_half_space (const vec3 &point) const`	Определяет полупространство*, в котором лежит точка. Если точка принадлежит полупространству, в сторону которого направлена нормаль, то возвращает `false`. В противном случае возвращает `true`
`real distance (const vec3 & point) const`	Находит расстояние между точкой `point` и данной плоскостью

Таблица 2.5 *(окончание)*

Определение метода	Назначение
`const vec3 &get_normal() const`	Возвращает нормаль к плоскости
`real get_distance_from_origin() const`	Возвращает расстояние между плоскостью и началом координат

* Плоскость разбивает пространство на два полупространства.

Для того чтобы продемонстрировать использование класса `plane` на практике, я добавил в предыдущий пример (листинг 2.6) поддержку закраски (Ex06) (рис. 2.3).

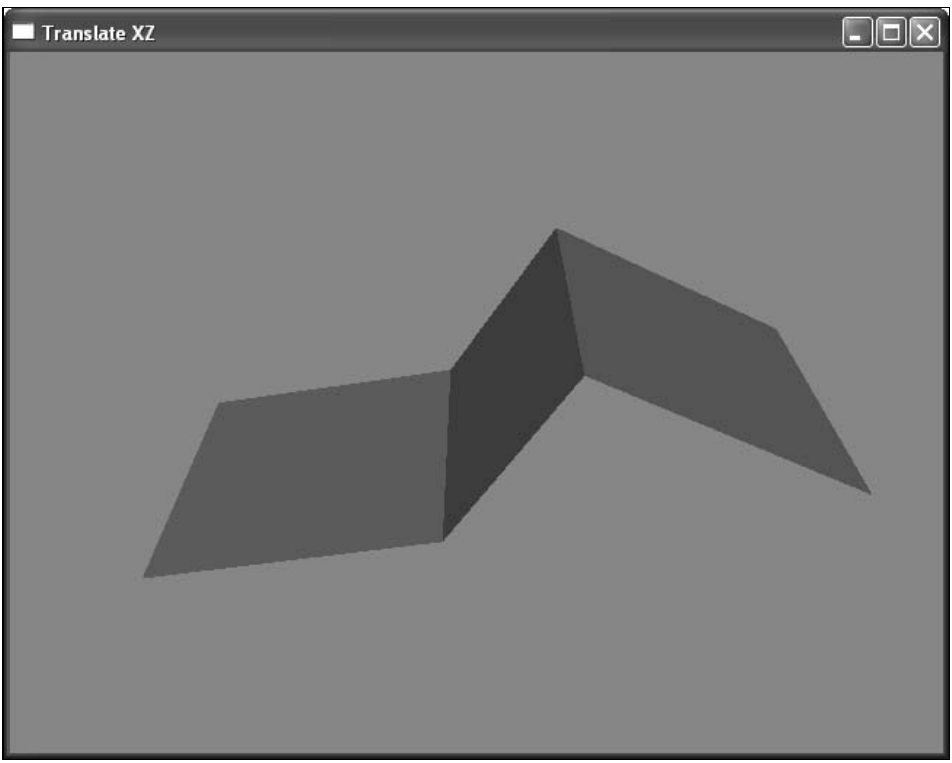


Рис. 2.3. Прямоугольники A, C, B (слева направо), закрашенные с учетом освещения

Для того чтобы OpenGL мог корректно закрасить полигоны, необходимо рассчитать к ним нормали. Эта задача очень просто решается средствами

класса `plane` — для этого в предыдущий пример придется добавить всего несколько строк. Ниже приведен текст новой функции `Display`.

Листинг 2.6

```
void Display()                  // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glPushMatrix();
              //Перемещение и поворот плоскости A
              glTranslatef(tx,ty,tz);
              glRotatef(rx,1,0,0);
              glRotatef(ry,0,1,0);

// Рассчитываем нормаль для плоскостей A и B
              planef plane1(vec3f(-1, -1, 0), vec3f(1, -1, 0), vec3f(1,
1, 0));
              vec3f normal1;
              normal1=plane1.get_normal();

              glBegin(GL_QUADS);
                     glNormal3fv(&normal1.v[0]);
                     glVertex2f(-1, -1);
                     glVertex2f( 1, -1);
                     glVertex2f( 1,  1);
                     glVertex2f(-1,  1);
              glEnd();

              glPushMatrix();
                     glTranslatef(1, 0, t);

                     glTranslatef(1, 0, 0);
                     glRotatef(r, 0, 1, 0);
                     glTranslatef(1, 0, 0);

                     glBegin(GL_QUADS);
                            glNormal3fv(&normal1.v[0]);
                            glVertex2f(-1, -1);
```

```
                        glVertex2f( 1, -1);
                        glVertex2f( 1,  1);
                        glVertex2f(-1,  1);
                    glEnd();
            glPopMatrix();

            vec4f v1(-1, -1, 0, 1);
            vec4f v2(-1, 1, 0, 1);

            matrix4f Translate1;
            matrix4f Translate2;
            quaternionf RotateQuat(vec3f(0, 1, 0), to_radians(r));
            matrix4f Rotate;

            Translate1.set_translate(vec3f(2, 0, t));
            Translate2.set_translate(vec3f(1, 0, 0));
            RotateQuat.get_value (Rotate);

            matrix4f m1;
            m1=Translate1*Rotate*Translate2;

            m1.mult_matrix_vec (v1);
            m1.mult_matrix_vec (v2);
// Рассчитываем нормаль для плоскости C
            planef plane2(vec3f(1, 1, 0), vec3f(1, -1, 0),
GLH_REAL_NAMESPACE::homogenize(v1));

            glBegin(GL_QUADS);
                    glNormal3fv(&plane2.get_normal().v[0]);
                    glVertex2f( 1,  1);
                    glVertex2f( 1, -1);
                    glVertex4fv(&v1.v[0]);
                    glVertex4fv(&v2.v[0]);
            glEnd();
    glPopMatrix();

    glutSwapBuffers();
...
}
```

Как видно, ничего сложно — мы просто передаем координаты первых трех вершин в качестве параметров конструктора класса `plane`, после чего получаем нормаль полученной плоскости при помощи метода `get_normal`.

На этом мы закончим обзор библиотеки `GLH_LINEAR`. Хочу еще раз обратить ваше внимание, что в этой главе мы рассмотрели лишь основы библиотеки GLH_LINEAR. Для продолжения изучения вам придется самим изучать исходный код этой библиотеки, который находится в файле \NVSDK\OpenGL\include\glh\glh_linear.h. Надеюсь, теперь это не будет трудной задачей для вас.

2.1.6. Библиотека GLH_CONVENIENCE

Библиотека GLH_LINEAR, рассмотренная выше, обладает многими достоинствами. Но, тем не менее, у нее есть один небольшой недостаток — она слабо связана с OpenGL. Например, для того, чтобы поместить какую-нибудь матрицу OpenGL в объект класса `matrix4`, приходится вручную создавать временный одномерный массив. Для того чтобы умножить текущую матрицу модели на кватернион, приходится конвертировать последний в матрицу. А для этого необходимо создавать временный объект. Список можно продолжить.

Эти недостатки не являются существенными, но, тем не менее, они довольно сильно досаждают программистам, делая исходный код программы менее понятным. Поэтому в состав OpenGL Helper Library входит библиотека GLH_CONVENIENCE, содержащая множество полезных функций, которые очень сильно облегчают жизнь программисту. Перечень этих функций приведен в табл. 2.6.

Таблица 2.6. *Функции библиотеки GLH_CONVENIENCE*

Определение функции	Назначение
`inline matrix4f get_matrix (GLenum matrix)`	Возвращает объект класса `matrix4f`, содержащий указанную матрицу OpenGL. Тип матрицы задается параметром matrix, который может принимать следующие значения: `GL_MODELVIEW_MATRIX`, `GL_PROJECTION_MATRIX`, `GL_TEXTURE_MATRIX`
`inline void glh_rotate (const quaternionf & r)`	Умножает текущую матрицу OpenGL на кватернион
`inline matrix4f object_lookat (const vec3f & from, const vec3f & to, const vec3f & Up)`	Аналог команды `gluLookAt`. Единственное отличие заключается в том, что она не умножает текущую матрицу OpenGL на полученную матрицу, которая возвращается как результат

***Таблица 2.6** (окончание)*

Определение функции	Назначение
`inline matrix4f frustum(float left, float right,float bottom, float top, float zNear, float zFar)`	Аналог команды `glFrustum`. Единственное отличие заключается в том, что она не умножает текущую матрицу OpenGL на полученную матрицу, которая возвращается как результат
`inline matrix4f frustum_inverse(float left, float right, float bottom, float top, float zNear, float zFar)`	Вычисляет обратную матрицу перспективного преобразования. Параметры аналогичны функции `frustum`. Если перемножить матрицы, возвращаемые командами `frustum` и `frustum_inverse`, то в результате получится единичная матрица
`inline matrix4f perspective(float fovy, float aspect, float zNear, float zFar)`	Аналог команды `gluPerspective`. Единственное отличие заключается в том, что она не умножает текущую матрицу OpenGL на полученную матрицу, которая возвращается как результат
`inline matrix4f perspective_inverse (float fovy, float aspect, float zNear, float zFar)`	Вычисляет обратную матрицу перспективного преобразования. Параметры аналогичны функции `perspective`. Если перемножить матрицы, возвращаемые командами `perspective` и `perspective _inverse`, то в результате получится единичная матрица
`inline void set_texgen_planes (GLenum plane_type, const matrix4f & m)`	Матричный аналог команды `glTexGenfv`. Параметры генерации координат `GL_S`, `GL_T`, `GL_R` и `GL_Q` хранятся в строках матрицы `m`. Нулевая строка — параметры автогенерации координат `GL_S`, первая — `GL_T` и т. д.
`inline vec3f range_compress (const vec3f & v)`	Переводит компоненты вектора `v` из диапазона [–1; 1] в диапазон [0..1]
`inline vec3f range_uncompress (const vec3f & v)`	Переводит компоненты вектора v из диапазона [0; 1] в диапазон [–1..1]

Подробности относительно особенностей работы этих команд вы сможете найти в файле \NVSDK\OpenGL\include\glh\glh_convenience.h.

Для того чтобы продемонстрировать использование этой библиотеки на практике, я переписал пример Ex04 без использования каких-либо матричных команд OpenGL, кроме `glMultMatrixf` (Ex07). Для этого в программу потребуется внести небольшое изменение: заменить команду `gluPerspective` функцией `perspective`, а функцию `glRotatef` — функцией `glh_rotate` (листинг 2.7).

Листинг 2.7

```
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <gl\glut.h>
#include <string>
#include <windows.h>
#include <glh_linear.h>
#include <glh_convenience.h>

using namespace std;
using namespace glh;
...
void Display()               // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glLoadIdentity();

       matrix4f rot_mat;            // Матрица вращения
       matrix4f translate;          // Матрица переноса
// Кватернион поворота вокруг оси x
       quaternionf quat_x(vec3f(1, 0, 0), to_radians(rx));

// умножаем на кватернион поворота вокруг оси y
       quat_x*=quaternionf (vec3f(0, 1, 0), to_radians(ry));
// Преобразуем кватернион в матрицу вращения
       quat_x.get_value (rot_mat);

       translate.set_translate(vec3f(tx, ty, tz));       //Перенос
// Создаем матрицу модели



// Умножаем матрицу модели на матрицу переноса
       glMultMatrixf(&translate.m[0]);
// Умножаем матрицу модели на кватернион
       glh_rotate (quat_x);
```

```
        glCallList(list);                    //Вывод объекта на экран

        glutSwapBuffers();
...
}

void Reshape(int Width,int Height)
{
        glViewport(0,0,Width,Height);
        WinWidth=Width;
        WinHeight=Height;

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        matrix4f projection;
// Рассчитываем матрицу проекции
        projection=perspective(45,GLdouble(WinWidth)/WinHeight,1,100);
// Умножаем текущую матрицу на матрицу проекции
        glMultMatrixf(&projection.m[0]);

        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        glutPostRedisplay();
}
...
```

В программе также демонстрируется использование функции `glh_rotate` для умножения текущей матрицы на кватернион.

На этом мы заканчиваем краткий обзор библиотеки GLH_CONVENIENCE и переходим к изучению наиболее важного компонента библиотеки OpenGL Helper Library — объектно-ориентированной надстройки над OpenGL.

2.2. Библиотека GLH_GLUT — объектная надстройка над GLUT

При разработке относительно простых демонстрационных программ программисту постоянно приходится программировать одни и те же операции:

реакцию на изменение размеров окна, вращение объекта мышью, выход из программы при нажатии клавиши <Esc> и т. д. Наиболее распространенный подход к этой проблеме — создание универсального шаблона и разработка приложений на основе этого шаблона[1]. Недостаток данного подхода — "разбухание" программного кода, т. к. в главный модуль приходится включать большое количество строк кода, выполняющего тривиальные задачи — инициализацию OpenGL, организацию простого интерфейса с пользователями т. д. Причем этот вспомогательный код, как правило, оказывается смешан с основным кодом программы, что затрудняет его анализ. Использование библиотеки GLUT лишь немного смягчает эту проблему. К счастью, библиотека OpenGL Helper Library содержит объектную надстройку над GLUT, которая позволяет автоматизировать многие часто встречающиеся операции. Исходный код этой надстройки находится в файле ...\NVSDK\OpenGL\include\glh\glh_glut.h.

Одна из причин появления надстройки над GLUT связана с тем, что в GLUT нельзя назначить одному событию несколько обработчиков. В принципе создатели GLH могли бы заставить пользователя запоминать адрес предыдущего обработчика сообщения и вызывать его в начале следующего, иными словами, производить суперклассирование (superclassing). Но это сильно усложнило бы разработку программ и увеличило бы риск возникновения ошибок. Вместо этого создатели GLH решили изменить механизм обработки сообщений GLUT.

В основе объектной надстройки над GLUT лежит набор интеракторов (interactors). *Интерактор* — это класс, производный от класса `glut_interactor`. Фактически интерактор является предопределенным обработчиком некоторой группы сообщений. Интеракторы, используемые программой, помещаются в коллекцию интеракторов:

```
std::list<glut_interactor *> interactors;
```

При наступлении события диспетчер сообщений GLH просматривает список интеракторов и вызывает метод-обработчик каждого интерактора для этого события. В качестве примера ниже приведен обработчик события `Display()` библиотеки GLH:

```
  void glut_display_function()
  {
       propagate = true;
// Перебираем все интеракторы из списка, пока не достигнем конца или
// один из интеракторов не сделает переменную propagate равной false
```

[1] Этот подход использовался при написании примеров предыдущей главы, в которой большинство примеров являются модификацией примера Ex04.

```
        for(std::list<glut_interactor *>::iterator
it=interactors.begin(); it != interactors.end() && propagate; it++)
// Вызываем метод display текущего интерактора
                (*it)->display();
  }
```

Для того чтобы библиотека GLH поместила свои стандартные обработчики в качестве обработчиков событий GLUT, программа должна вызвать функцию `glut_helpers_initialize()`:

```
inline void glut_helpers_initialize()
  {
        glutDisplayFunc(glut_display_function);
        glutIdleFunc(0);
        glutKeyboardFunc(glut_keyboard_function);
        glutMenuStatusFunc(glut_menu_status_function);
        glutMotionFunc(glut_motion_function);
        glutMouseFunc(glut_mouse_function);
        glutPassiveMotionFunc(glut_passive_motion_function);
        glutReshapeFunc(glut_reshape_function);
        glutSpecialFunc(glut_special_function);
        glutVisibilityFunc(glut_visibility_function);
  }
```

Обычно вызов этой функции размещается внутри функции `main()` после операторов создания окна.

Рассмотрим структуру класса `glut_interactor`:

```
class glut_interactor
{
public:
  glut_interactor () { enabled = true; }

  virtual void display() {} //Аналог обработчика glutDispayFunc.
  virtual void idle() {}    //Далее - по аналогии
  virtual void keyboard(unsigned char key, int x, int y) {}
  virtual void menu_status(int status, int x, int y) {}
  virtual void motion(int x, int y) {}
  virtual void mouse(int button, int state, int x, int y) {}
  virtual void passive_motion(int x, int y) {}
  virtual void reshape(int w, int h) {}
  virtual void special(int  key, int x, int y) {}
```

```
  virtual void timer(int value) {}
  virtual void visibility(int v) {}

  virtual void enable()  { enabled = true; }
  virtual void disable() { enabled = false; }

  bool enabled;
};
```

Основу класса составляют одиннадцать виртуальных функций-обработчиков событий GLUT. Свойство `enabled` указывает диспетчеру событий GLH, является ли интерактор в данный момент активным (`true`) или неактивным (`false`). В последнем случае интерактор игнорируется.

Читатель, наверное, уже заметил, что интерактор `glut_interactor` фактически представляет собой каркас программы, написанной на GLUT. Это обстоятельство позволяет создавать классы, производные от `glut_interactor`, которые представляют собой законченную мини-программу на GLUT.

В составе GLH имеется несколько типовых интеракторов, решающих распространенные задачи — создание матрицы проекции, ее коррекцию при изменении размеров окна (`glut_perspective_reshaper`), перемещение по сцене при помощи мыши (`glut_simple_mouse_interactor`) и т. д. Это приводит к тому, что программа строится как бы из готовых кирпичиков: добавили в коллекцию интерактор `glut_perspective_reshaper` — у нас появилась камера, добавили еще и `glut_perspective_reshaper` — теперь сцену можно вращать, приближать или удалять при помощи мыши.

Пользовательские обработчики событий тоже представляют собой интеракторы. Но создавать класс, производный от `glut_interactor`, довольно утомительно, поэтому в составе GLH имеется интерактор `glut_callbacks` (для экономии места приведено лишь объявление полей/методов, которые касаются событий `display` и `idle`):

```
class glut_callbacks : public glut_interactor
{
public:
     glut_callbacks() :
            display_function(0),
            idle_function(0),
            keyboard_function(0),
            menu_status_function(0),
            motion_function(0),
            mouse_function(0),
```

```
        passive_motion_function(0),
        reshape_function(0),
        special_function(0),
        timer_function(0),
        visibility_function()
{}

virtual void display()
{ if(display_function) display_function(); }

virtual void idle()
{ if(idle_function) idle_function(); }

virtual void keyboard(unsigned char k, int x, int y)
{ if(keyboard_function) keyboard_function(k, x, y); }

virtual void menu_status(int status, int x, int y)
{ if(menu_status_function) menu_status_function(status, x, y); }

virtual void motion(int x, int y)
{ if(motion_function) motion_function(x,y); }

virtual void mouse(int button, int state, int x, int y)
{ if(mouse_function) mouse_function(button, state, x, y); }

virtual void passive_motion(int x, int y)
{ if(passive_motion_function) passive_motion_function(x, y); }

virtual void reshape(int w, int h)
{ if(reshape_function) reshape_function(w, h); }

virtual void special(int key, int x, int y)
{ if(special_function) special_function(key, x, y); }

virtual void timer(int value)
{ if(timer_function) timer_function(value); }

virtual void visibility(int v)
{ if(visibility_function) visibility_function(v); }
```

```
// Указатели на пользовательские обработчики
        void (* display_function) ();
        void (* idle_function) ();
        void (* keyboard_function) (unsigned char, int, int);
        void (* menu_status_function) (int, int, int);
        void (* motion_function) (int, int);
        void (* mouse_function) (int, int, int, int);
        void (* passive_motion_function) (int, int);
        void (* reshape_function) (int, int);
        void (* special_function) (int, int, int);
        void (* timer_function) (int);
        void (* visibility_function) (int);

  };
```

Как видно, данный класс имеет набор полей с названием вида `xxxx_function`, которым можно присваивать адреса обработчиков событий GLUT. Этот класс позволяет быстро добавлять в готовые GLUT-программы поддержку интеракторов. Для этого необходимо выполнить следующие действия:

1. Подключить заголовочные файлы GLH и добавить в коллекцию interactors необходимые интеракторы.
2. Удалить из GLUT-программы весь код, который дублирует функции интеракторов.
3. Добавить в программу интерактор `glut_callbacks` и связать его с обработчиками сообщений, которые остались от старой GLUT-программы.

В этой главе мы постепенно перепишем пример Ex04 из *главы 1* с использованием интеракторов. Для начала мы поместим весь код обработчиков событий библиотеки GLUT в интерактор `glut_callbacks` (Ex08):

```
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <windows.h>

#include <glh/glh_glut.h>

using namespace std;
using namespace glh;
...
GLuint list0=0;
```

```
glut_callbacks cb;
...

int main(int argc, char* argv[])
{
       glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
       glutInitWindowSize(WinWidth,WinHeight);
       glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                                  (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
       glutCreateWindow("GLUT Teapot");

       Init();
// Инициализируем библиотеку glh
       glut_helpers_initialize();
// Устанавливаем обработчики сообщений интерактора cb
       cb.display_function=Display;
       cb.keyboard_function=Keyboard;
       cb.special_function=KeyboardSpecial;
       cb.reshape_function=Reshape;
       cb.mouse_function=Mouse;
       cb.motion_function=MouseMotion;

// Добавляем интерактор cb в список интеракторов
       glut_add_interactor(&cb);

       glutMainLoop();

       return 0;
}
```

Для этого мы создаем интерактор `cb` и присваиваем его обработчикам адреса бывших обработчиков GLUT, после чего помещаем его в список интеракторов при помощи функции `glut_add_interactor`:

```
inline void glut_add_interactor(glut_interactor *gi, bool append=true)
{
       glut_remove_interactor(gi);
       if(append)
              interactors.push_back(gi);
```

```
        else
                interactors.push_front(gi);
    }
```

Из определения этой функции видно, что она принимает два параметра: указатель на интерактор и флаг `append`, который показывает, куда следует добавить интерактор — в начало (`false`) или в конец (`true`) списка интеракторов.

Правда, в ходе модификации программы я наткнулся на небольшой подводный камень. В примере Ex04 *главы 1* мы храним идентификатор дисплейного списка в переменной с названием `list`. А объектная надстройка над GLUT использует `STL`. Следовательно, возникает конфликт имен между классом списка `list` и переменной `list`. Проблема решается путем изменения названия дисплейного списка с `list` на `list0`.

В настоящее время мы еще не получили никаких преимуществ от перехода на библиотеку GLH. В этом нет ничего удивительного — главная сила GLH заключена в наборе стандартных интеракторов, нацеленных на решение повседневных задач.

2.2.1. Интерактор glut_perspective_reshaper

Интерактор `glut_perspective_reshaper` предназначен для автоматической генерации перспективной проекции. Этот интерактор также автоматически корректирует параметры видового окна и матрицу проекции при изменении размеров окна.

Интерактор `glut_perspective_reshaper` имеет один конструктор:

```
glut_perspective_reshaper(float infovy = 60.f, float inzNear = .1f, float
inzFar = 10.f)
```

Параметры конструктора аналогичны параметрам команды `gluPerspective`, с единственной разницей — отсутствует параметр `aspect`, который стал не нужен — интерактор его рассчитывает автоматически исходя из текущих параметров окна.

Интерактор обрабатывает одно событие — `reshape`. Код обработчика приведен ниже:

```
  class glut_perspective_reshaper : public glut_interactor
  {
  public:
...
          void reshape(int w, int h)
```

```
        {
                width = w; height = h;

                if(enabled) apply ();
        }

        void apply ()
        {
                glViewport(0,0,width,height);
                glMatrixMode(GL_PROJECTION);
                glLoadIdentity();
                apply_perspective();
                glMatrixMode(GL_MODELVIEW);
        }
        void apply_perspective()
        {
             ...
         }
...
}
```

Как видно, метод `reshape`, являющийся обработчиком события изменения размеров окна, вызывает метод `apply ()` только при условии, что поле `enable` равно `true`. Это дает возможность легко "выключить" интерактор путем присвоения полю `enable` значения `false`. Обратите внимание, что метод `reshape` заносит в поля `width` и `height` текущие размеры окна — пользовательская программа может быстро получить информацию о размерах окна из этих полей.

Метод `apply` устанавливает параметры видового окна, делает текущей матрицу проекции и вызывает метод `apply_perspective`, умножающий текущую матрицу на матрицу проекции.

Чтобы все стало более понятно, мы перепишем пример Ex08 с использованием интерактора `glut_perspective_reshaper` (Ex09):

```
...
glut_perspective_reshaper reshaper(60, 0.1, 100);
glut_callbacks cb;
...
```

```
void Display()                // Обновление содержимого экрана
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glLoadIdentity();
        glTranslatef(tx,ty,tz);             //Перемещение и поворт объекта
        glRotatef(rx,1,0,0);
        glRotatef(ry,0,1,0);

        glCallList(list0);                  //Вывод объекта на экран

        glutSwapBuffers();
}
...
int main(int argc, char* argv[])
{
...
        glut_add_interactor(&reshaper);
        glut_add_interactor(&cb);
...
}
```

После этих преобразований программа-функция `Reshape` значительно упростилась. Теперь она лишь инициализирует матрицу модели и посылает сообщение перерисовки экрана. Все остальное делает интерактор `glut_perspective_reshaper`.

2.2.2. Интерактор *glut_simple_interactor*

В программах OpenGL, работающих с мышью, постоянно приходится выполнять некоторый набор действий: определять, нажата ли нужная кнопка мыши, не нажаты ли одновременно с ней клавиши наподобие <Ctrl>, <Alt> или <Shift>, на какое расстояние передвинулась мышь с момента предыдущего события и т. д.

Поэтому в библиотеку GLH был включен интерактор `glut_simple_interactor`, который облегчает написание подобных программ. Принцип работы этого интерактора можно описать следующим образом.

Сначала программа создает объект класса `glut_simple_interactor`. Затем она присваивает значения полям интерактора, чтобы указать, какие события он должен обрабатывать (к примеру, интерактор может быть настроен на пере-

мещение мыши с нажатой клавишей <Ctrl>). После того как программа поместит настроенный интерактор в список интеракторов, он начинает перехватывать события, на которые настроен. После наступления события интерактор запоминает состояние мыши в данный момент и определяет отличия текущего состояния мыши от предыдущего (на сколько пикселов сдвинулась стрелка мыши и т. д.), после чего вызывает виртуальный метод `update`, который ничего не делает.

А ничего не делает он потому, что класс `glut_simple_interactor` является абстрактным классом, на основе которого пользователь должен конструировать свои классы. Поэтому метод `update` фактически является пользовательским обработчиком события, на которое настроен интерактор.

Рассмотрение класса `glut_simple_interactor` начнем, как всегда, с конструктора:

```
  class glut_simple_interactor : public glut_interactor
  {
  public:
       glut_simple_interactor()
       {
// Активизироваться по левому щелчку мыши
         activate_on = GLUT_LEFT_BUTTON;
// Событие не активно (левая кнопка мыши еще не нажата)
         active = false;
// Реагировать на клавиши-модификаторы
         use_modifiers = true;
// Набор-модификатор, при котором должно срабатывать событие
         modifiers = 0;
// Ширина и высота окна равны 0
         width = height = 0;
// Все, что связано с координатами мыши, равно 0
         x0 = y0 = x = y = dx = dy = 0;
       }
...
// Поля
// Кнопка мыши, события от которой обрабатывает интерактор
       int activate_on;
// Использовать ли модификаторы (true =да)
       bool use_modifiers;
// Набор комбинация модификаторов, при которой будет
```

```
// активироваться интерактор
        int modifiers;
// Активен ли сейчас интерактор (нажата ли сейчас
// кнопка мыши + комбинация модификаторов)
        bool active;
// Координаты мыши до перемещения
        int x0, y0;
// Координаты мыши после перемещения
        int x, y;
// Изменения координат мыши
        int dx, dy;
// Размеры окна
        int width, height;
}
```

Как видно из определения, конструктор класса `glut_simple_interactor` по умолчанию создает интерактор, реагирующий на нажатие левой кнопки мыши. Но если при этом будет нажата хоть одна клавиша-модификатор, то событие будет проигнорировано.

Для того что бы вы лучше почувствовали процесс обработки событий от мыши интерактором `glut_simple_interactor`, ниже приведен код обработчиков событий щелчка и перемещения мыши с подробными комментариями:

```
// Обработчик щелчка мыши
        virtual void mouse(int button, int state, int X, int Y)
        {
// Если интерактор "включен" и нажатая кнопка мыши является кнопкой
//активации и она нажата в данный момент (модификаторы не используются
//или набор активных модификаторов равен полю modifiers)
          if(enabled && button == activate_on && state == GLUT_DOWN &&
                (! use_modifiers || (modifiers == glutGetModifiers())) )
          {
                 active = true;
                 x = x0 = X;
                 y = y0 = Y;
                 dx = dy = 0;
          }
// Если кнопка мыши не нажата
          else if (enabled && button == activate_on && state == GLUT_UP)
```

```
        {
            if(dx == 0 && dy == 0)
                update();
            active = false;
            dx = dy = 0;
        }
    }
// Обработка события перемещения мыши
    virtual void motion(int X, int Y)
    {
        if(enabled && active)
        {
            dx = X - x;   dy = y - Y;
            x = X;   y = Y;
// Вызвать пользовательский обработчик событий
            update();
        }
    }
```

Класс `glut_simple_interactor` имеет ряд абстрактных методов, которые определяются потомками:

```
// Умножает текущую матрицу на матрицу, рассчитанную интерактором
    virtual void apply_transform() = 0;
// Умножает текущую матрицу на обратную матрицу, рассчитанную
// интерактором
    virtual void apply_inverse_transform() = 0;
// Возвращает матрицу, рассчитанную интерактором
    virtual matrix4f get_transform() = 0;
// Возвращает обратную матрицу, рассчитанную интерактором
    virtual matrix4f get_inverse_transform() = 0;
// Пользовательский обработчик событий
    virtual void update() {}
```

Мы не будем рассматривать примеры использования интерактора `glut_simple_interactor`. Дело в том, что этот интерактор никогда не используется на практике. Но поскольку в библиотеке GLH он является предком для многих интеракторов, понимание механизма его работы вам очень пригодится при чтении следующих разделов.

2.2.3. Интерактор *glut_rotate*

Интерактор `glut_rotate` предназначен для облегчения написания программ, в которых поворот объектов (или сцены) осуществляется при помощи мыши. Кстати, в эту категорию попадает большинство примеров этой книги.

Когда пользователь двигает мышь вдоль оси при нажатой левой кнопке, интерактор считает, что он хочет осуществить поворот вокруг оси *x*. Если же мышь двигается вдоль оси *y*, поворот осуществляется вокруг оси *y*. Обратите внимание, что все наши программы в настоящее время используют такой же интерфейс.

Интерактор `glut_rotate` является наследником `glut_simple_interactor`. В результате он имеет очень простую структуру, т. к. всю "грязную" работу выполняет его предок:

```
class glut_rotate : public glut_simple_interactor
  {
  public:
// Конструктор
       glut_rotate()
       {
// Начальные углы поворота равны 0
         rotate_x = rotate_y = 0;
// Чувствительность мыши 1
         scale = 1;
       }
// Обработчик события перемещения мыши
       void update()
       {
// Изменяем угол поворота на расстояние, которое прошла мышь (с учетом
// чувствительности)
         rotate_x += dx * scale;
         rotate_y += dy * scale;
         glutPostRedisplay();
       }
// Умножаем текущую матрицу на матрицу поворота
// Иными словами, выполняем поворот вокруг осей x и y
       void apply_transform()
       {
         glRotatef(rotate_x, 0, 1, 0);
         glRotatef(rotate_y, -1, 0, 0);
       }
```

```
// Аналогичным образом, переопределяем остальные методы класса
// glut_simple_interactor
       void apply_inverse_transform()
       {
         glRotatef(-rotate_y, -1, 0, 0);
         glRotatef(-rotate_x, 0, 1, 0);
       }

       matrix4f get_transform()
       {
              rotationf rx(to_radians(rotate_x), 0, 1, 0);
              rotationf ry(to_radians(rotate_y), -1, 0, 0);
              matrix4f mx, my;
              rx.get_value (mx);
              ry.get_value (my);
              return mx * my;
       }

       matrix4f get_inverse_transform()
       {
              rotationf rx(to_radians(-rotate_x), 0, 1, 0);
              rotationf ry(to_radians(-rotate_y), -1, 0, 0);
              matrix4f mx, my;
              rx.get_value (mx);
              ry.get_value (my);
              return my * mx;
       }
// Углы поворота вокруг осей x и y и чувствительность
// мыши (по умолчанию 1)
       float rotate_x, rotate_y, scale;
  };
```

Для того чтобы продемонстрировать использование этого интерактора на практике, я переписал пример Ex09 с использованием интерактора `glut_rotate` (листинг 2.8) (Ex10).

Листинг 2.8

```
...
glut_perspective_reshaper reshaper(60, 0.1, 100);
```

```
glut_rotate mouse_rotate;
glut_callbacks cb;
...
void Display()              // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glLoadIdentity();
       glTranslatef(tx,ty,tz);            // Перемещение
       mouse_rotate.apply_transform(); // Поворот

       glCallList(list0);                 //Вывод объекта на экран

       glutSwapBuffers();
}

void Mouse(int button, int state, int x, int y)
              //Обработка щелчков мыши
{
       if (button==GLUT_RIGHT_BUTTON)      //Правая кнопка
       {
              switch (state)
              {
                     case GLUT_DOWN:
                            rdown=true;
                            mx=x;
                            my=y;
                            break;
                     case GLUT_UP:
                            rdown=false;
                            break;
              }
       }
}

void MouseMotion(int x, int y)      //Перемещение мыши
{
       if (rdown)     //Правая кнопка
       {
```

```
                tx+=0.01*(x-mx); // Перемещение вдоль активной плоскости
                if (tt)
                        tz+=0.01*(y-my);
                else
                        ty+=0.01*(my-y);
                mx=x;
                my=y;
                glutPostRedisplay();
        }
}

int main(int argc, char* argv[])
{
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
        glutInitWindowSize(WinWidth,WinHeight);
        glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                                (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
        glutCreateWindow("GLUT Teapot");

        Init();

        glut_helpers_initialize();
// Изменяем чувствительность мыши
        mouse_rotate.scale=0.3;

        cb.display_function=Display;
        cb.keyboard_function=Keyboard;
        cb.special_function=KeyboardSpecial;
        cb.mouse_function=Mouse;
        cb.motion_function=MouseMotion;

        glut_add_interactor(&reshaper);
        glut_add_interactor(&mouse_rotate);
        glut_add_interactor(&cb);

        glutMainLoop();

        return 0;
}
```

Обработчики событий мыши стали значительно проще — теперь они отвечают лишь за перемещение наблюдателя по сцене. А сама программа стала более "стройной" и понятной.

2.2.4. Интерактор *glut_trackball*

В интеракторе `glut_rotate` поворот сцены задается при помощи двух углов поворота вокруг осей *x* и *y* (так называемые Эйлеровы углы). Причем сначала поворот осуществляется вокруг оси x, а затем, вокруг оси *y*. Этот подход имеет один существенный недостаток — после поворота вокруг оси *x* изменяется ориентация других осей (*y* и *z*), в результате чего ось второго поворота зависит от угла поворота вокруг оси *x*. В результате у неподготовленного пользователя после нескольких минут работы с программой может "поехать крыша" — он просто запутается в осях координат. Подробнее об этой проблеме, известной под названием Gimbal lock, можно прочитать в [12].

Одним из решений этой проблемы является моделирование виртуального трекбола. Для этого необходимо определить соответствие между положением точки на поверхности шарика трекбола и позицией указателя мыши, которая передвигается на плоскость. Один из возможных вариантов такого соответствия изображен на рис. 2.4.

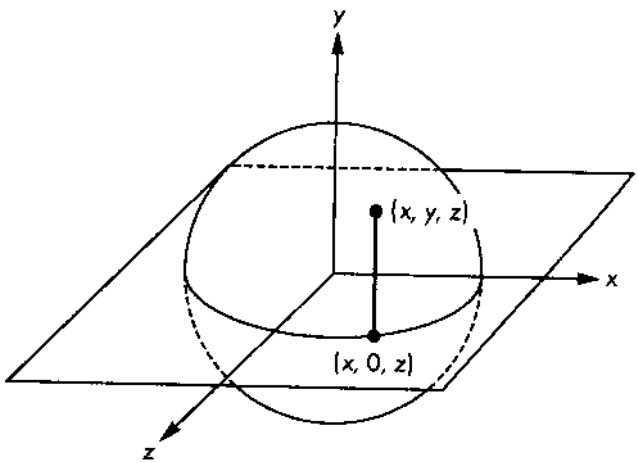


Рис. 2.4. Проекция на плоскость точки на поверхности шарика трекбола[2]

[2] Рисунок взят из [2].

Если мы предположим, что необходимая нам точка на поверхности шарика трекбола имеет положительную координату *y*, то мы сможем определить однозначное соответствие между проекцией точки на поверхность шарика трекбола и самой точкой на поверхности шарика. Иными словами, мы сможем по положению указателя мыши на экране восстановить трехмерную точку на поверхности трекбола. А зная две точки на поверхности трекбола, мы сможем определить направление и угол поворота трекбола (рис. 2.5).

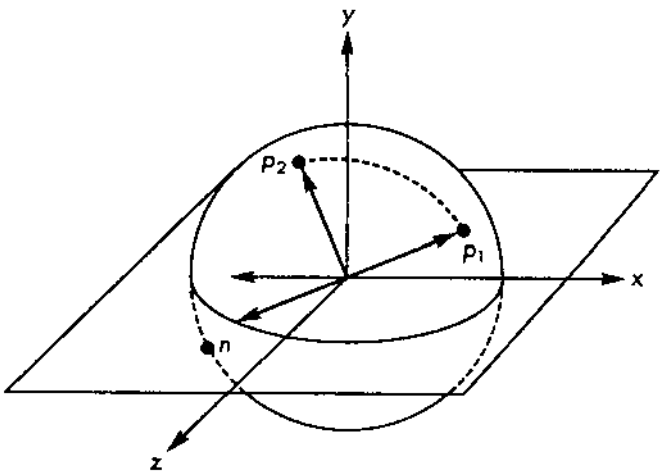


Рис. 2.5. Определение плоскости поворота трекбола[3]

Ось поворота можно найти как $\overline{n} = \overline{p}_1 \times \overline{p}_2$, а угол поворота определяется соотношением $|\sin\theta| = |\overline{n}|$, где $\overline{p}_1$ и $\overline{p}_2$ — векторы, проведенные из начала координат к точкам p_1 и p_2. Зная ось и угол поворота, мы сможем легко повернуть сцену. Более подробную информацию о моделировании виртуальных трекболов вы сможете найти в [2]. Также см. *разд. 4.10.2.*

Для эмуляции виртуального трекбола в библиотеке GLH используется интерактор `glut_trackball`. Мы не будем его подробно рассматривать т. к., во-первых, он довольно сложен для понимания, а во-вторых, полученная информация вам все равно вряд ли пригодится. Для использования интерактора трекбола достаточно знать только то, что он является наследником класса `glut_simple_interactor`. Для того чтобы продемонстрировать использование интерактора трекбола на практике, я переписал пример Ex10 с использованием этого интерактора (листинг 2.9).

[3] Рисунок взят из [2].

Листинг 2.9

```
glut_perspective_reshaper reshaper(60, 0.1, 100);
glut_trackball trackball;
glut_callbacks cb;

void Display()               // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glLoadIdentity();
       glTranslatef(tx,ty,tz);
       trackball.apply_transform();

       glCallList(list0);                  // Вывод объекта на экран

       glutSwapBuffers();
}

int main(int argc, char* argv[])
{
       glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
       glutInitWindowSize(WinWidth,WinHeight);
       glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                              (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
       glutCreateWindow("GLUT Teapot");

       Init();

       glut_helpers_initialize();

       cb.display_function=Display;
       cb.keyboard_function=Keyboard;
       cb.special_function=KeyboardSpecial;
       cb.mouse_function=Mouse;
       cb.motion_function=MouseMotion;
```

```
        glut_add_interactor(&reshaper);
        glut_add_interactor(&trackball);
        glut_add_interactor(&cb);

        glutMainLoop();

        return 0;
}
```

Как видно, все изменения в программе сводятся лишь к замене класса `glut_rotate` на `glh_trackball`. Остальные изменения носят чисто косметический характер.

2.2.5. Интеракторы *glut_pan* и *glut_dolly*

В большинстве программ пользователь должен иметь возможность не только изменять угол обзора сцены, но и перемещаться по сцене. Рассмотренные ранее интеракторы `glut_rotate` и `glut_trackball` решают лишь первую задачу из этого списка — осмотр сцены под разными углами.

Для решения второй задачи в библиотеке GLH используются два интерактора: `glut_pan` и `glut_dolly`. Первый из них реализует перемещение объекта вдоль осей *x* и *y*, а второй — вдоль оси *z*.

Эти интеракторы имеют очень похожую структуру, поэтому в данном разделе мы рассмотрим только исходный код интерактора `glut_pan`.

Конструктор класса `glut_pan` просто присваивает полям класса `glut_pan` значения по умолчанию:

```
class glut_pan : public glut_simple_interactor
  {
  public:
        glut_pan()
        {
// Чувствительность мыши=0.1
          scale = .01f;
// Не инвертировать мышь
          invert_increment = false;
// Не поворачивать систему координат (перенос будет производиться
// вдоль осей x и y)
          parent_rotation = 0;
        }
...
```

```
// Флаг "инвертировать мышь". Если равен true, то мышь начинает
// работать наоборот
// движение вверх трактуется как вниз, влево — как вправо
       bool invert_increment;
// Кватернион поворота системы координат, вдоль осей которой производится
// перемещение
       const rotationf * parent_rotation;
// Вектор переноса
       vec3f pan;
// Коэффициент масштабирования (чувствительность мыши)
       float scale;
  };
```

После того как интерактор `glut_pan` создан и помещен в список интеракторов, он начинает обрабатывать события от мыши при помощи переопределенного метода `update`:

```
       void update()
       {
// Вектор переноса. dx и dy рассчитываются
// предком (glut_simple_interactor)
         vec3f v(dx, dy, 0);
// Если определен интерактор поворота, то поворачиваем вектор переноса
         if(parent_rotation != 0) parent_rotation->mult_vec (v);
// Если флаг invert_increment равен true, то инвертируем мышь,
// иначе делаем перенос в нормальном направлении
         if(invert_increment)
                pan -= v * scale;
         else
                pan += v * scale;
         glutPostRedisplay();
       }
```

Доступ к информации о переносе организован обычным для класса `glut_simple_interactor` способом — при помощи четырех переопределенных методов:

```
// Выполняет перенос вдоль осей x и y
       void apply_transform()
       {
         //cerr << "Applying transform: " << (x - x0) << ", " << (y -
y0) << endl;
```

```
        glTranslatef(pan[0], pan[1], pan[2]);
      }

// Выполняет обратный перенос вдоль осей x и y
      void apply_inverse_transform()
      {
        //cerr << "Applying transform: " << (x - x0) << ", " << (y -
y0) << endl;
        glTranslatef(-pan[0], -pan[1], -pan[2]);
      }

// Возвращает матрицу переноса
      matrix4f get_transform()
      {
            matrix4f m;
            m.make_identity();
            m.set_translate(pan);
            return m;
      }

// Возвращает обратную матрицу переноса
      matrix4f get_inverse_transform()
      {
            matrix4f m;
            m.make_identity();
            m.set_translate(-pan);
            return m;
      }
```

Для демонстрации применения интеракторов `glut_pan` и `glut_dolly` я переписал пример Ex11 с использованием этих интеракторов (листинг 2.10) (Ex12).

Листинг 2.10

```
glut_perspective_reshaper reshaper(60, 0.1, 100);
glut_trackball trackball;
glut_pan pan;
```

```
glut_dolly dolly;
glut_callbacks cb;

...

void Display()               // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glLoadIdentity();
// Умножаем матрицу модели на матрицы всех интеракторов
       pan.apply_transform();
       dolly.apply_transform();
       trackball.apply_transform();

       glCallList(list0);                  // Вывод объекта на экран

       glutSwapBuffers();
}

int main(int argc, char* argv[])
{
       glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
       glutInitWindowSize(WinWidth,WinHeight);
       glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                              (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
       glutCreateWindow("GLUT Teapot");

       Init();

       glut_helpers_initialize();

// Устанавливаем начальный вектор перемещения равным (0, 0, -9)
       dolly.dolly[2]=-9;
// Интерактор dolly работает при нажатой левой кнопке мыши + <CTRL>
       dolly.modifiers=GLUT_ACTIVE_CTRL;
// Интерактор pan работает при нажатой левой кнопке мыши + <SHIFT>
       pan.modifiers=GLUT_ACTIVE_SHIFT;
```

```
        cb.display_function=Display;
        cb.keyboard_function=Keyboard;

        glut_add_interactor(&reshaper);
        glut_add_interactor(&pan);
        glut_add_interactor(&dolly);
        glut_add_interactor(&trackball);
        glut_add_interactor(&cb);

        glutMainLoop();

        return 0;
}
```

Перенос вдоль осей x и y производится при нажатой клавише-модификаторе <Ctrl>, а вдоль оси z — при нажатой клавише <Shift>. В результате модификации из программы исчезли обработчики мыши (`Mouse` и `MouseMotion`), а сама программа стала более компактной.

2.2.6. Интерактор *glut_simple_mouse_interactor*

В прошлом примере для перемещения по сцене мы использовали целых три интерактора. Но хотя этот подход дает нам определенную гибкость при написании программы, в большинстве случаев было бы удобнее использовать один универсальный интерактор, который умел бы и вращать сцену, и перемещаться по ней в трех направлениях.

В библиотеке GLH есть такой интерактор — это `glut_simple_mouse_interactor`, который является надстройкой над интеракторами `glut_trackball`, `glut_pan` и `glut_dolly`. Для того чтобы понять основную идею этого интерактора, достаточно только одного взгляда на метод `apply_transform()`:

```
  struct glut_simple_mouse_interactor : public glut_interactor
  {
  public:
...
        void apply_transform()
        {
// Умножаем текущую матрицу на матрицы интеракторов
          pan.apply_transform();
          dolly.apply_transform();
```

```
        trackball.apply_transform();
      }
// Остальные три метода генерации матриц устроены аналогичным образом
...
// Режим камеры (по умолчанию равен false). Когда равен true, камера
// начинает работать непонятным образом. Зачем нужен этот второй
// режим - я так и не понял...
      bool camera_mode;
// Интеракторы
      glut_trackball trackball;
      glut_pan  pan;
      glut_dolly dolly;
  };
```

Как видно, интерактор `glut_simple_mouse_interactor` объединяет интеракторы `glut_trackball`, `glut_pan` и `glut_dolly` в единый интерактор с общим конструктором, матрицей преобразования и т. д.

Для начала рассмотрим конструктор этого интерактора:

```
      glut_simple_mouse_interactor(int num_buttons_to_use=3)
      {
        configure_buttons(num_buttons_to_use);
        camera_mode = false;
      }
```

Как видно, этот конструктор передает свой единственный параметр методу `num_buttons_to_use` и устанавливает "классический" режим камеры. Метод `configure_buttons` устанавливает вид интерфейса с пользователем и может принимать значения 1, 2 и 3 (табл. 2.7).

Таблица 2.7. *Назначение кнопок мыши и клавиатуры интерактора* `glut_simple_mouse_interactor`

Значение, передаваемое методу `configure_buttons`	**Вращение объекта**	**Перемещение в плоскости экрана**	**Перемещение вдоль оси Z**
1	Левая кнопка мыши	<Shift> + левая кнопка мыши	<Ctrl> + левая кнопка мыши
2	Левая кнопка мыши	Средняя кнопка мыши	<Ctrl> + левая кнопка мыши
3	Левая кнопка мыши	Средняя кнопка мыши	Правая кнопка мыши

Для большей ясности далее приведен исходный код метода `configure_buttons`:

```
void configure_buttons(int num_buttons_to_use = 3)
{
  switch(num_buttons_to_use)
  {
  case 1:
       trackball.activate_on = GLUT_LEFT_BUTTON;

       trackball.modifiers = 0;
       pan.activate_on = GLUT_LEFT_BUTTON;
       pan.modifiers = GLUT_ACTIVE_SHIFT;
       dolly.activate_on = GLUT_LEFT_BUTTON;
       dolly.modifiers = GLUT_ACTIVE_CTRL;
       break;

  case 2:
       trackball.activate_on = GLUT_LEFT_BUTTON;
       trackball.modifiers = 0;

       pan.activate_on = GLUT_MIDDLE_BUTTON;
       pan.modifiers = 0;

       dolly.activate_on = GLUT_LEFT_BUTTON;
       dolly.modifiers = GLUT_ACTIVE_CTRL;
       break;

  case 3:
  default:
       trackball.activate_on = GLUT_LEFT_BUTTON;
       trackball.modifiers = 0;

       pan.activate_on = GLUT_MIDDLE_BUTTON;
       pan.modifiers = 0;

       dolly.activate_on = GLUT_RIGHT_BUTTON;
       dolly.modifiers = 0;

       break;
  }
}
```

Обратите внимание на то, что интерактор `glut_simple_mouse_interactor` не помещает дочерние интеракторы (`glut_trackball`, `glut_pan` и `glut_dolly`) в список интеракторов. На это есть две причины.

- Если бы дочерние интеракторы помещались в список интеракторов, то пользовательской программе при удалении интерактора `glut_simple_mouse_interactor` из списка интеракторов пришлось бы самостоятельно удалять из этого списка и дочерние интеракторы. А это усложнило бы программу и увеличило риск появления ошибок.
- Если в конструкторе помещать дочерние интеракторы в список интеракторов, то можно налететь на следующий подводный камень. Дело в том, что если интерактор является статическим объектом, то он может быть создан до списка интеракторов. Компилятор в целях оптимизации может создавать статические объекты не в порядке объявления. А помещение дочерних интеракторов в список интеракторов вручную усложняет проблему и увеличивает риск появления ошибок.

Поэтому класс `glut_trackball` использует оригинальный прием — в процессе обработки некоторого события он самостоятельно вызывает соответствующие методы дочерних интеракторов. В качестве подтверждения этого факта ниже приведен код обработчиков событий от мыши.

```
virtual void motion(int x, int y)
{

        trackball.motion(x,y);

        pan.motion(x,y);

        dolly.motion(x,y);
}
virtual void mouse(int button, int state, int x, int y)
{
  trackball.mouse(button, state, x, y);
  pan.mouse(button, state, x, y);
  dolly.mouse(button, state, x, y);
}
```

Для демонстрации использования интерактора `glut_simple_mouse_interactor` я переписал пример Ex12 с использованием этого интерактора (Ex13). Полный код примера приведен в листинге 2.11.

Листинг 2.11

```
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <windows.h>

#include <glh/glh_glut.h>

using namespace std;
using namespace glh;

int WinWidth=640;            // Ширина окна
int WinHeight=480;           // Высота окна

GLuint list0=0;

glut_perspective_reshaper reshaper(60, 0.1, 100);
glut_simple_mouse_interactor user;
glut_callbacks cb;

void Init()                          // Инициализация OpenGL
{
       glEnable(GL_DEPTH_TEST);
       glEnable(GL_LIGHTING);
       glEnable(GL_LIGHT0);
       glEnable(GL_COLOR_MATERIAL);

       glColor3f(0.1,0.7,0.2);
       glClearColor(0.5, 0.5, 0.75, 1);

       list0=glGenLists(1);

       glNewList(list0, GL_COMPILE);
       glutSolidTeapot(2);
       glEndList();
}
```

```
void Display()                // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glLoadIdentity();
// Умножение текущей матрицы на матрицу интерактора
       user.apply_transform();

       glCallList(list0);                   // Вывод объекта на экран

       glutSwapBuffers();
}

void Keyboard(unsigned char key,int x,int y)
              // Обработка сообщений от клавиатуры
{
       switch (key)
       {
              case VK_ESCAPE:       // Если нажата клавиша <ESC> — выход
                     if (list0)
                            glDeleteLists(list0,1); // Удалить
                                                    // дисплейный список
                     exit(0);
                     break;
       }
}

int main(int argc, char* argv[])
{
       glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
       glutInitWindowSize(WinWidth,WinHeight);
       glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                              (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
       glutCreateWindow("GLUT Teapot");

       Init();

       glut_helpers_initialize();
```

```
// Используем интерфейс для одной кнопки мыши
        user.configure_buttons(1);
// Переносим камеру на -9 единиц вдоль оси Z
        user.dolly.dolly[2]=-9;

        cb.display_function=Display;
        cb.keyboard_function=Keyboard;

        glut_add_interactor(&reshaper);
        glut_add_interactor(&user);
        glut_add_interactor(&cb);

        glutMainLoop();

        return 0;
}
```

Пример Ex13 можно считать шаблоном GLH-программы, на основе которой можно создавать новые программы.

Пример получился очень компактным. Но, тем не менее, программу можно еще больше упростить, заменив обработчик сообщений от клавиатуры стандартным обработчиком GLH_GLUT:

```
inline void glut_exit_on_escape(unsigned char k, int x = 0, int y = 0)
{ if(k==27) exit(0); }
```

Этот прием демонстрируется в следующем примере (Ex14):

```
...
// Удаление списка при выходе
void onExit()
{
        glDeleteLists(list0,1);
}

int main(int argc, char* argv[])
{
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
        glutInitWindowSize(WinWidth,WinHeight);
        glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
```

```
                                    (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
        glutCreateWindow("GLUT Teapot");

        Init();

        glut_helpers_initialize();

        user.configure_buttons(1);
        user.dolly.dolly[2]=-9;

        cb.display_function=Display;
// Обработчик событий от клавиатуры — стандартный обработчик glut
        cb.keyboard_function=glut_exit_on_escape;

        glut_add_interactor(&reshaper);
        glut_add_interactor(&user);
        glut_add_interactor(&cb);
// При выходе из программы будет вызываться функция Exit
        atexit(onExit);

        glutMainLoop();

        return 0;
}
```

При этом в программу пришлось внести небольшое изменение — удаление дисплейного списка было перенесено из обработчика клавиши <Esc> в функцию `onExit`, что, кстати, является более надежным решением.

2.2.7. Функции *glut_timer* и *glut_idle*

Как говорилось ранее, объектная надстройка над GLUT использует свои собственные обработчики событий GLUT. При возникновении события обработчик GLUT перебирает интеракторы из списка интеракторов и вызывает соответствующий метод-обработчик события каждого интерактора. С большинством событий такой механизм работает вполне нормально. Но есть два исключения: события таймера `glutTimerFunc` и бездействия программы `glutIdleFunc`.

При программировании таймера программа должна самостоятельно указать не только интервал, через который работает таймер, но и обработчик события от таймера. А этим обработчиком должен быть обработчик библиотеки GLH. Это противоречит самой идеологии объектной надстройки над GLUT — пользовательская программа не должна непосредственно работать со стандартными обработчиками библиотеки GLH.

Поэтому было найдено компромиссное решение — функция `glut_timer`:

```
inline void glut_timer (int msec, int value)
{
      glutTimerFunc(msec, glut_timer_function, value);
}
```

Как видно, функция принимает интервал, через который сработает таймер, и идентификатор таймера, после чего устанавливает в качестве обработчика таймера собственную функцию-обработчик. Обратите внимание: мы могли бы сделать то же самое и без функции `glut_timer`, но тогда есть вероятность, что наша программа перестанет работать с будущими версиями NVIDIA OpenGL SDK.

Для демонстрации использования функции `glut_timer` я добавил в пример Ex14 вращение чайника с использованием таймера (листинг 2.12).

Листинг 2.12

```
GLfloat rr=0;
...
void Display()               // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glLoadIdentity();
       user.apply_transform();
// Поворчаиваем чайник
       glRotatef(rr, 0, 1, 0);

       glCallList(list0);                  // Вывод объекта на экран

       glutSwapBuffers();
}
...
void Timer(int Value)
```

```
{
        rr+=2;
        if (rr>360)
                rr-=360;
// Устанавливаем следующий вызов таймера через 25 мс
        glut_timer (25, 1);
        glutPostRedisplay();
}
...
int main(int argc, char* argv[])
{
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
        glutInitWindowSize(WinWidth,WinHeight);
        glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                                (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
        glutCreateWindow("GLUT Teapot");

        Init();

        glut_helpers_initialize();

        user.configure_buttons(1);
        user.dolly.dolly[2]=-9;

        cb.display_function=Display;
        cb.keyboard_function=glut_exit_on_escape;
        cb.timer_function=Timer;

        glut_add_interactor(&reshaper);
        glut_add_interactor(&user);
        glut_add_interactor(&cb);

        atexit(onExit);
// Вызываем обработчик таймера
        Timer(1);

        glutMainLoop();

        return 0;
}
```

С обработчиком бездействия программы (idle) ситуация обстоит аналогичным образом — для включения и выключения обработки события idle используется функция glut_idle:

```
inline void glut_idle (bool do_idle)
{
      glutIdleFunc(do_idle ? glut_idle_function : 0);
}
```

Как видно, функция glut_idle принимает один параметр типа bool. Если он равен true, то обработка события idle включена, в противном случае — выключена.

Использование функции glut_idle демонстрируется в примере Ex16, который является модификацией примера Ex15 (листинг 2.13).

Листинг 2.13

```
GLfloat rr=0;
float OldTick;
...
void Idle()
{
       rr+=(GetTickCount()-OldTick)*0.1;
       OldTick=GetTickCount();
       if (rr>360)
              rr-=360;

       glutPostRedisplay();
}
...
int main(int argc, char* argv[])
{
       glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
       glutInitWindowSize(WinWidth,WinHeight);
       glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                              (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
       glutCreateWindow("GLUT Teapot");

       Init();

       glut_helpers_initialize();
```

```
        user.configure_buttons(1);
        user.dolly.dolly[2]=-9;

        cb.display_function=Display;
        cb.keyboard_function=glut_exit_on_escape;
        cb.idle_function=Idle;

        glut_add_interactor(&reshaper);
        glut_add_interactor(&user);
        glut_add_interactor(&cb);

        atexit(onExit);

        OldTick=GetTickCount();
        glut_idle (true);

        glutMainLoop();

        return 0;
}
```

2.2.8. Создание нового интерактора на примере интерактора консоли

Во время отладки OpenGL-приложений программисту часто необходимо получить информацию о текущем состоянии программы. Классический отладчик в этом случае оказывается бесполезным, т. к. его использование изменяет процесс выполнения программы — происходит смена видеорежима, главное окно программы становится неактивным, возможна потеря содержимого `PBuffer` и т. д. Кроме того, я сталкивался со множеством случаев, когда отладочная версия программы (DEBUG) вроде бы работает нормально, а RELEASE — некорректно. Писать "трассу" выполнения программы в файл — тоже не выход: результаты трассировки можно будет посмотреть только после выхода из программы.

Поэтому целесообразно в программу встроить свой собственный небольшой отладчик. В этой идее нет абсолютно ничего нового — все современные 3D-игры имеют в своем составе консоль (командный интерпретатор, который может работать параллельно с основной программой), при помощи которой можно подстроить игру под конкретную систему или просмотреть отладочную информацию.

Из всего вышесказанного следует, что хорошо бы создать универсальную консоль, которую можно будет легко подключать к произвольному проекту. При использовании стандартных средств GLUT такая задача практически невыполнима — GLUT не поддерживает обработку одного события несколькими обработчиками. А написание руководства по подключению консоли наподобие: "вставьте в начало обработчика события `XXX` вызов функции `YYY`, а в конец — вызов функции `ZZZ`", — не представляется решением проблемы.

Зато эта задача очень красиво решается с использованием библиотеки GLH. Для этого достаточно инкапсулировать весь код, отвечающий за поддержку консоли в отдельный интерактор, и подключить его к программе таким образом, чтобы обработчики этого интерактора вызывались после вывода сцены на экран.

Но для начала нам надо будет немного доработать наш шаблон GLH-программы из примера Ex03. Дело в том, что в этом шаблоне предполагается, что вывод информации на экран осуществляет пользовательский обработчик `display()`. В конце этого обработчика вызывается функция `glutSwapBuffers()`. Следовательно, если мы попытаемся нарисовать консоль, то она окажется в следующем кадре (который будет затерт командой `glClear()` обработчика `display()`). Одно из решений этой проблемы — перенести вызов команды `glutSwapBuffers()` из пользовательского обработчика в интерактор консоли. Но в этом случае в будущем мы рискуем столкнуться с проблемами при подключении нового интерактора, который обязательно должен вызываться сразу после интерактора консоли.

Поэтому я решил вынести команду `glutPostRedisplay` в отдельный интерактор `glut_swapbuffers`, который должен находиться в конце списка интеракторов (листинг 2.14).

Листинг 2.14

```
class glut_swapbuffers:public glut_interactor
{
public:
  void virtual display();
};

void glut_swapbuffers::display()
{
  glutSwapBuffers();
}
```

Для экономии времени создадим консоль минимальной функциональности — наша консоль будет только отображать несколько последних сообщений программы, а появляться и исчезать при помощи клавиши <~>. Но даже такая простая консоль часто оказывается полезной при отслеживании "трассы" выполнения программы.

Ниже приведено объявление интерактора консоли `glut_console`:

```
class glut_console:public glut_interactor
{
public:
  unsigned int max_lines; // Максимальное число строк в консоли
  list<string> lines;     // Список строк консоли
  bool visible;           // Видима ли консоль (true=да)
  glut_console (unsigned int amax_lines=15);
  void add(string s);     //Добавление новой строки
  void virtual display();
  virtual void keyboard(unsigned char k, int x, int y);
};
```

При создании консоли конструктору передается максимальное число строк, которое может быть отображено на экране. Затем приложение добавляет сообщения в консоль, используя метод `add`.

```
void glut_console::add(string s)
{
       lines.push_back(s);
// Если число строк в консоли больше max_lines, то удаляем лишние строки
       while (lines.size()>max_lines)
              lines.erase(lines.begin());
// Если консоль показывается на экране, то перерисовываем экран
       if (visible)
              glutPostRedisplay();
}
```

Если число сообщений превысит `max_lines`, старые сообщения будут удалены. Пользователь может в любой момент нажать клавишу ~ и ознакомиться со списком сообщений. Отображение окна консоли на экране осуществляется при помощи следующего кода (листинг 2.15).

Листинг 2.15

```
void  glut_console::display()
{
```

```
        if (visible)
        {
                glPushAttrib(GL_ALL_ATTRIB_BITS);
                unsigned int Width=glutGet(GLUT_WINDOW_WIDTH);
                unsigned int Height=glutGet(GLUT_WINDOW_HEIGHT);
                unsigned int ConsoleHeight=(15+2)*max_lines+2;

// Устанавливаем параметры видового окна + отсечение за пределами консоли
                glViewport(0, Height-ConsoleHeight, Width, ConsoleHeight);
                glEnable(GL_SCISSOR_TEST);
                glScissor(0, Height-ConsoleHeight, Width, ConsoleHeight);

// Устанавливаем ортографическую проекцию
                glMatrixMode(GL_PROJECTION);
                glPushMatrix();
                glLoadIdentity();
                gluOrtho2D(0, Width, 0, ConsoleHeight);
                glMatrixMode(GL_MODELVIEW);
                glLoadIdentity();

// Заполняем консоль черным цветом
                glClearColor(0, 0, 0, 1);
                glClear(GL_COLOR_BUFFER_BIT);

// Выводим текст в консоль
                glColor3f(0,1,1);
                int Pos=(lines.size()-1)*(15+2)+2;
                for (list<string>::iterator itor=lines.begin();
itor!=lines.end(); itor++, Pos-=17)
                {
                        glRasterPos2f(0, Pos);
                        const char *c=itor->c_str();
                        while (*c)
                        {
                                glutBitmapCharacter(GLUT_BITMAP_9_BY_15,
*c);
                                c++;
                        }
                }
```

```
// Восстанавливаем старые матрицы
                glMatrixMode(GL_PROJECTION);
                glPopMatrix();
                glMatrixMode(GL_MODELVIEW);

                glPopAttrib();
        }
}
```

Теперь остается добавить поддержку консоли в нашу программу (листинг 2.16) (Ex17).

Листинг 2.16

```
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <windows.h>

#include <glh/glh_glut.h>
#include "console.h"

using namespace std;
using namespace glh;

int WinWidth=640;            // Ширина окна
int WinHeight=480;           // Высота окна

const string tstr[2]={"Translate XY", "Translate XZ"};

GLuint list0=0;

glut_perspective_reshaper reshaper(60, 0.1, 100);
glut_simple_mouse_interactor user;
glut_callbacks cb;
// Интерактор консоли
glut_console console;
glut_swapbuffers swapbuffers;

...
```

```
void Display()                // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glLoadIdentity();
       user.apply_transform();

       glCallList(list0);                    // Вывод объекта на экран

// Команда glutSwapBuffers не нужна - переключение буферов выполняет
// интерактор glut_swapbuffers
}



...

int main(int argc, char* argv[])
{
       console.add("Initializing GLUT...");
       glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
       glutInitWindowSize(WinWidth,WinHeight);
       glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                              (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
       glutCreateWindow("GLUT Teapot");
       console.add("Initializing OpenGL...");

       Init();

       console.add("Initializing OpenGL Helper Library...");
       glut_helpers_initialize();

       user.configure_buttons(1);
       user.dolly.dolly[2]=-9;

       cb.display_function=Display;
       cb.keyboard_function=glut_exit_on_escape;

       glut_add_interactor(&reshaper);
       glut_add_interactor(&user);
```

```
        glut_add_interactor(&cb);
        glut_add_interactor(&console);
        glut_add_interactor(&swapbuffers);

        atexit(onExit);

        console.add("Initialize complete. Starting glutMainLoop");
        glutMainLoop();

        return 0;
}
```

Как видно, для того чтобы добавить (или убрать) консоль из нашей программы, достаточно изменить всего несколько строк программы. Внешний вид такой консоли показан на рис. 2.6.

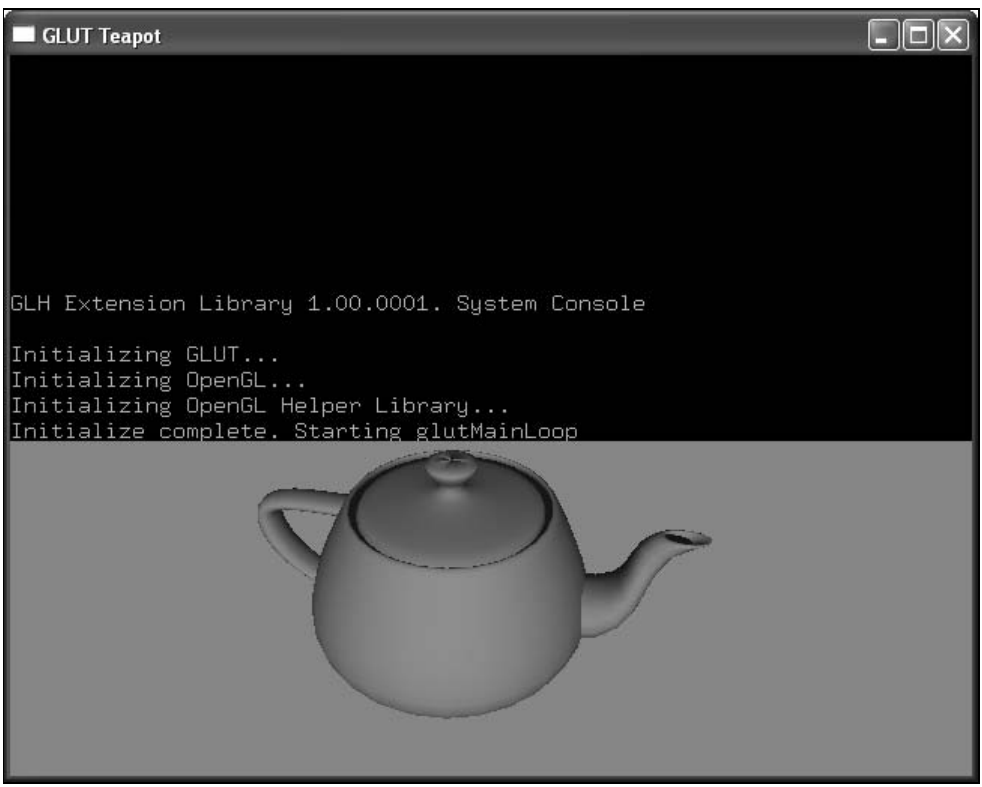


Рис. 2.6. Внешний вид консоли

Разумеется, эту консоль можно улучшить, добавив в нее командную строку, скроллинг сообщений и т. д. В следующем разделе мы рассмотрим одно из возможных расширений библиотеки GLH, включающее в себя довольно продвинутую консоль с поддержкой командной строки, скриптов, назначением клавишам команд и т. д.

2.3. Библиотека GLH_GLUT EXT — расширение GLH

В *разд. 2.2* было показано, что библиотека GLH_GLUT очень сильно упрощает разработку демонстрационных OpenGL-программ. Но, тем не менее, она не устраняет ряд недостатков GLUT, с которыми я столкнулся в процессе написания книги.

- Отсутствие удобного механизма вывода сообщений на экран (вроде функции `MessageBox` операционных систем Win32).
- Отсутствуют средства ввода текстовых и числовых данных от пользователя. Для того чтобы попросить пользователя указать цвет объекта (четыре целых числа), приходится писать огромное количество строк кода, что явно не способствует надежности и понятности программы.
- Отсутствует возможность сохранения конфигурации программы в файле. Программисту приходится самостоятельно реализовывать этот механизм.
- Отсутствует какая-либо возможность простого замера производительности программы (FPS).

И т. д. и т. п.

Существуют два варианта решения этой проблемы.

- Легкий. Написать об этом в службу поддержки NVIDIA и подождать, когда NVIDIA доработает библиотеку GLH_GLUT (если сочтет это необходимым).
- Тяжелый. Самостоятельно доработать библиотеку GLH_GLUT.

У нас не принято искать легких путей, поэтому я выбрал второй вариант, правда с одной поправкой — вместо того, чтобы вносить изменения в исходный код библиотеки GLH, я решил разработать свою надстройку над этой библиотекой — OpenGL Helper Library Extension, сокращенно — GLHE. Последнюю версию библиотеки можно найти на CD-диске в каталоге \NVIDIA SDK\Extensions\GLH (исходный код библиотеки расположен в файле glh_glut_ext.h).

В основе этой библиотеки лежат два интерактора: `glut_simple_user_interface` (объединение интеракторов `glut_perspective_reshaper` и `glut_simple_mouse_interactor`) и `glut_console` (интерактор консоли).

По идее, оба эти интерактора могли бы образовать один "мегаинтерактор", если бы не одно серьезное обстоятельство — обработчики интерактора glut_simple_user_interface должны выполняться до обработчиков пользователя, а glut_console — после пользовательских обработчиков. В результате конструктор интерактора glut_simple_user_interface принимает в качестве параметра указатель на интерактор glut_console. Мы не будем рассматривать внутреннюю структуру интеракторов, т. к. это в настоящее время просто бесполезно из-за того, что библиотека GLHE активно развивается. Кроме того, файл glh_glut_ext.h содержит множество комментариев, так что при желании вы сможете самостоятельно разобраться в устройстве этой библиотеки.

Рассмотрение возможностей библиотеки GLHE мы начнем с того, что перепишем пример Ex14 с использованием этой библиотеки (листинг 2.17) (Ex18).

Листинг 2.17

```
#define STRICT
#define WIN32_LEAN_AND_MEAN
// Подключаем библиотеку GLH
#include "glh_glut_ext.h"

using namespace std;
using namespace glh;

int WinWidth=640;            // Ширина окна
int WinHeight=480;           // Высота окна

GLuint list0=0;

glut_console console;
glut_simple_user_interface user(&console);
glut_callbacks cb;
glut_swapbuffers swapbuffers;

const string Title="GLHE Demo";

void Init()                          // Инициализация OpenGL
{
       glEnable(GL_DEPTH_TEST);
```

```
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_COLOR_MATERIAL);

        glColor3f(0.1,0.7,0.2);
        glClearColor(0.5, 0.5, 0.75, 1);

        list0=glGenLists(1);

        glNewList(list0, GL_COMPILE);       // Создание
                                    // дисплейного списка объекта (чайника)
                glutSolidTeapot(2);
        glEndList();
}

void Display()               // Обновление содержимого экрана
{
// Обратите внимание, нам не надо настраивать матрицу модели — интерактор
// glut_simple_user_interface сделает это за нас

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glCallList(list0);                  //Вывод объекта на экран
}

void onExit()
{
        glDeleteLists(list0,1);
}

int main(int argc, char* argv[])
{
        console.add("Initializing GLUT...");
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
        glutInitWindowSize(WinWidth,WinHeight);
        glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                                    (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
        glutCreateWindow("GLUT Teapot");
        console.add("Initializing OpenGL...");
```

```
        Init();

        console.add("Initializing OpenGL Helper Library...");
        glut_helpers_initialize();
// Устанавливаем заголовок окна (если заголовок будет затерт, он будет
// восстановлен из этого поля)
        console.title=Title;

// Настраиваем параметры подобъекта ("подинтерактора") user_mouse
// (glut_simple_mouse_interactor).
        user.user_mouse.configure_buttons(1);
        user.user_mouse.dolly.dolly[2]=-9;

        cb.display_function=Display;

// Интерактор glut_simple_user_interface должен быть расположен до
// пользовательских обработчиков
        glut_add_interactor(&user);
        glut_add_interactor(&cb);
// Интерактор glut_console должен располагаться после пользовательских
// обработчиков
        glut_add_interactor(&console);
        glut_add_interactor(&swapbuffers);

        atexit(onExit);

        console.add("Initialize complete. Starting glutMainLoop");
        glutMainLoop();

        return 0;
}
```

Для того чтобы подключить библиотеку к программе, достаточно добавить директиву `#include "glh_glut_ext.h"`. После чего необходимо внести в список используемых библиотек следующие файлы: nv_math.lib, nv_util.lib, BugslayerUtil.lib (команда меню **Project | Properties | Linker | Command Line**). Не забудьте, если вы этого еще не сделали, прописать пути к этим файлам *(см. Введение)*.

Суть интерактора glut_simple_user_interface можно описать фразой — это интеракторы glut_perspective_reshaper и glut_simple_mouse_interactor "в одном флаконе". Упомянутые выше интеракторы расположены соответственно в полях reshaper и user_mouse. После того как интерактор glut_simple_user_interface помещен в список сообщений, он начинает передавать все поступающие сообщения этим двум интеракторам. При обработке события display он также автоматически устанавливает в качестве текущей матрицы модели матрицу подобъекта user_mouse (интерактор glut_simple_mouse_interactor). Поэтому программист может сосредоточиться исключительно на программировании вывода сцены на экран, не заботясь об интерфейсе с пользователем.

2.3.1. Интерактор *glut_console*

Интерактор glut_console является продвинутой версией интерактора консоли из *разд. 2.2.8*, в который добавлено множество новых возможностей. Первым нововведением, бросающимся в глаза, является поддержка командной строки (рис. 2.7).

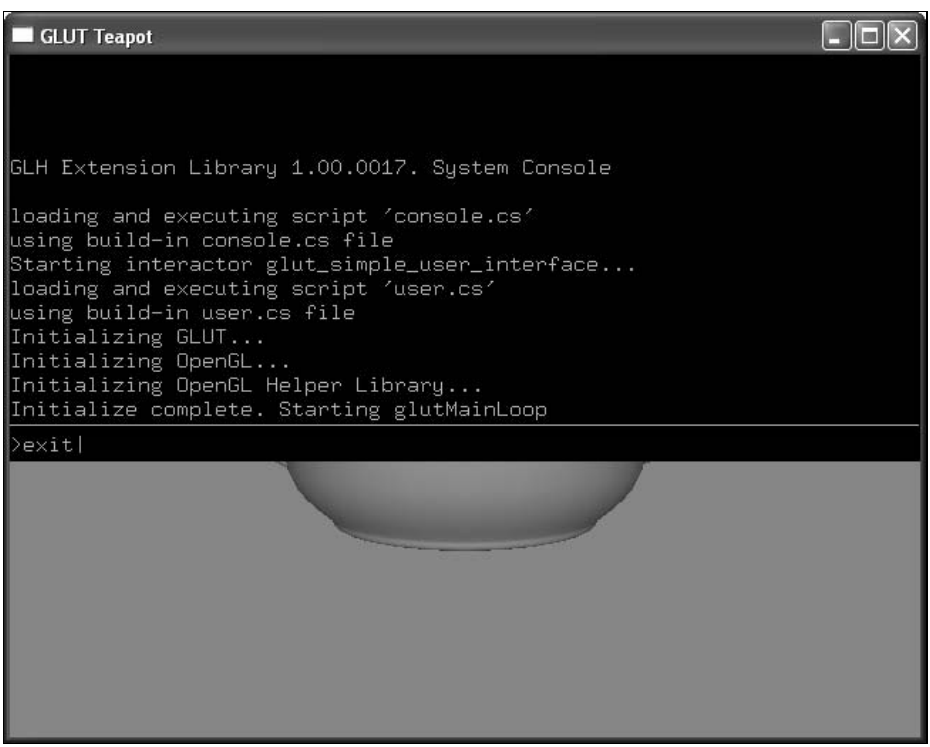


Рис. 2.7. Консоль библиотеки GLHE

Командная строка поддерживает историю команд (клавиши <↑> и <↓>), а так же быстрый ввод команды по ее первым буквам (клавиша <Tab>). Команда запускается на выполнение при помощи клавиши <Enter>. Прокрутка содержимого консоли осуществляется клавишами <Page Up> и <Page Down>.

Список всех команд с их кратким описанием может быть получен при помощи команды `help` или клавиши <F1> (рис. 2.8).

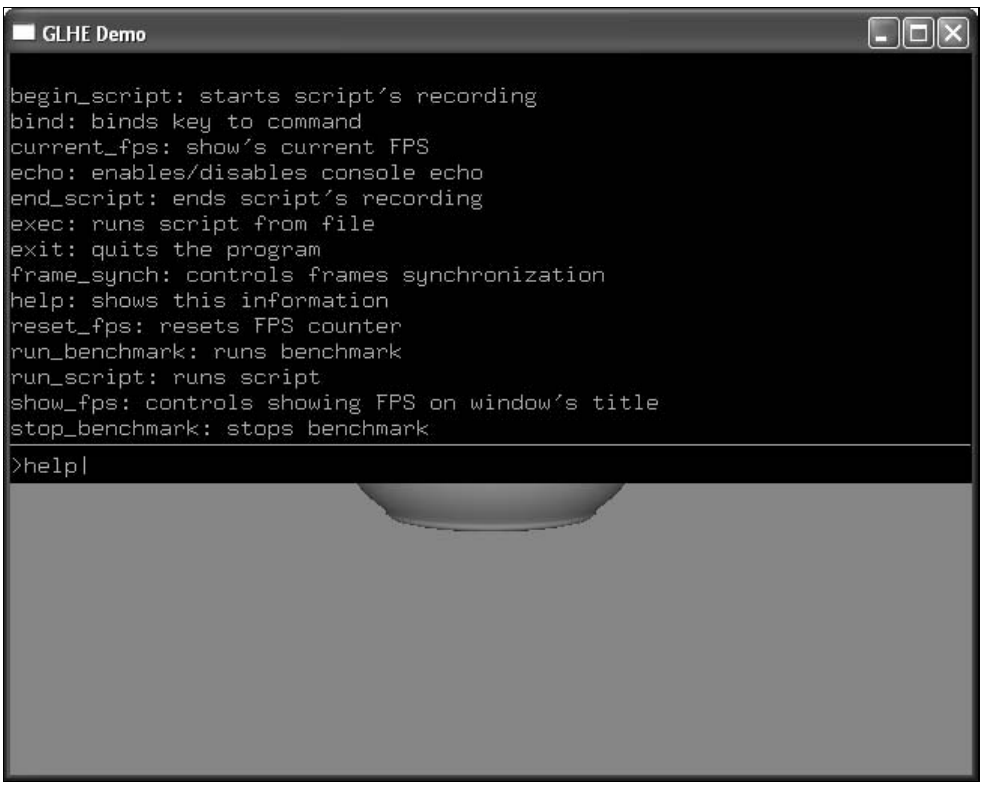


Рис. 2.8. Внешний вид консоли после выполнения команды `help`

Краткое описание стандартных консольных команд приведено в табл. 2.8.

Таблица 2.8. *Стандартные консольные команды библиотеки GLHE*

Команда	Описание
`help`	Показывает список поддерживаемых команд с их кратким описанием

Таблица 2.8 *(окончание)*

Команда	Описание
`echo`	Включает/выключает вывод "эхо" команд в консоль. Принимает один параметр, который может быть либо 0 (вывод эха запрещен) либо 1 (разрешен). Команда `echo` полезна в скриптах для того, чтобы не загромождать экран
`show_fps`	Включает/выключает вывод текущего среднего FPS в заголовке окна. Принимает один параметр, который может быть либо 0 (вывод FPS запрещен), либо 1 (разрешен)
`current_fps`	Выводит информацию о среднем FPS в командное окно
`reset_fps`	Сбрасывает информацию о FPS. Полезна перед началом измерения производительности
`frame_synch`	Включает/выключает кадровую синхронизацию. Принимает один параметр, который может быть либо 0 (синхронизация выключена), либо 1 (включена)
`run_benchmark`	Запускает циклический поворот сцены вокруг своей оси. Полезна для грубой оценки производительности приложения
`stop_benchmark`	Останавливает циклический поворот сцены вокруг своей оси
`exit`	Завершает работу программы
`begin_script`	Начинает запись скрипта. В качестве параметра принимает название скрипта
`end_script`	Завершает запись скрипта
`run_script`	Запускает скрипт на выполнение
`bind`	Привязывает определенную команду (второй параметр, команда должна быть внутри двойных кавычек) к данной клавише (первый параметр)
`exec`	Выполняет команды из данного файла (первый параметр)

Дополнительная информация о команде может быть получена при помощи ключа `/?` (как в MS-DOS).

Для демонстрации возможностей консоли мы решим с ее помощью две простые задачи.

Задача 1. Сделать так, чтобы программа завершала работу после нажатия клавиши <F10>.

Для ее выполнения достаточно ввести команду:

```
bind F10 "exit"
```

Правда, заставлять пользователя вводить эту команду при каждом запуске программы — это самое настоящее издевательство. Гораздо гуманнее сде-

лать так, чтобы программа автоматически при запуске выполняла какой-нибудь пакетный файл, например autoexec.cs. Для этого достаточно в функцию `main` перед командой `glutMainLoop` вставить следующую строку (Ex19):

```
console.processCmd("exec autoexec.cs");
```

Метод `processCmd` интерактора `glut_console` выполняет команду консоли, которая передается в качестве параметра. Теперь пользователь сможет самостоятельно задавать набор консольных команд, который будет выполняться при запуске программы, не внося изменений в исходный код программы.

Задача 2. Сделать так, чтобы при нажатии клавиши <F2> запускался встроенный бенчмарк с выключенной кадровой синхронизацией и включенным показом FPS в заголовке окна. При повторном нажатии клавиши <F2> бенчмарк должен остановиться, включить кадровую синхронизацию и выключить показ FPS в заголовке окна, после чего вывести информацию о FPS в консоль.

Решение этой задачи можно найти в файле user.cs, который автоматически выполняется конструктором интерактора `glut_simple_user_interface`. Кстати, интерактор `glut_console` также выполняет свой пакетный файл при запуске программы — console.cs. Если же интерактор не может найти пакетный файл, то он выполняет пакетный файл, "встроенный" в конструктор.

Ниже приведено содержимое встроенного файла user.cs (листинг 2.18).

Листинг 2.18

```
begin_script run_benchmark_script
bind F2 "run_script stop_benchmark_script"
show_fps 1
frame_synch 0
run_benchmark
end_script

begin_script stop_benchmark_script
bind F2 "run_script run_benchmark_script"
show_fps 0
frame_synch 1
stop_benchmark
end_script

bind F2 "run_script run_benchmark_script"
```

Для борьбы с ограничением команды `bind` (каждой клавише может быть назначена только одна команда) необходимый набор команд объединяется в скрипт. А вызов полученного скрипта привязывается к клавише <F2>. В нашем случае клавиша <F2> поочередно выполняет два скрипта. Для достижения этого эффекта в тело скрипта включается команда, назначающая клавише <F2> другой скрипт.

Возможно, вам сейчас кажется, что возможности этой консоли очень ограничены. Это утверждение было бы верным, если не учитывать то, что пользовательская программа может создавать свои консольные команды. Рассмотрим процесс создания консольной команды.

Сначала программа должна создать обработчик консольных команд, который должен выглядеть примерно так:

```
bool CmdProc(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
        if (Cmd=="пользовательская команда")
        {
                // Обрабатываем команду
                return true;
        }
        return false;
}
```

Библиотека GLHE передает пользовательскому обработчику четыре параметра:

- указатель (задается при регистрации разработчика, обычно равен 0);
- название команды, которая сейчас обрабатывается;
- массив параметров, переданных команде;
- указатель на интерактор `glut_console`, который вызвал обработчик.

В случае удачного выполнения команды обработчик должен вернуть `true`, в противном случае — `false`. Кстати, программа может иметь неограниченное число обработчиков консольных команд.

Для регистрации новой консольной команды используется метод `addCmd`:

```
void glut_console::addCmd(string command, string info, void* object, Con-
soleCmdProc proc)
```

Метод принимает четыре параметра:

- название команды;
- описание команды, которое показывается командой `help`;

- указатель (обычно равен 0);
- адрес пользовательского обработчика команд.

Для демонстрации создания пользовательских консольных команд мы добавим в пример Ex19 новую консольную команду `color`, которая позволяет изменять цвет чайника (листинг 2.19) (Ex20).

Листинг 2.19

```
float Color[3]={0.1, 0.7, 0.2};
...
void Display()                // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Устанавливаем цвет объекта
       glColor3fv(&Color[0]);
       glCallList(list0);                  // Вывод объекта на экран
}
// Обработчик консольных команд
bool CmdProc(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
       if (Cmd=="color")
       {
              if (Params.size()==3)
              {
                     Color[0]=atof(Params[0].c_str());
                     Color[1]=atof(Params[1].c_str());
                     Color[2]=atof(Params[2].c_str());
                     glutPostRedisplay();
              }
       }

       return true;
}

int main(int argc, char* argv[])
{
       console.add("Initializing GLUT...");
       glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
```

```
        glutInitWindowSize(WinWidth,WinHeight);
        glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                               (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
        glutCreateWindow(Title.c_str());
        console.add("Initializing OpenGL...");

        Init();

        console.add("Initializing OpenGL Helper Library...");
        glut_helpers_initialize();

        console.title=Title;
// Регистрируем новую консольную команду
        console.addCmd("color", "sets objects color", 0, CmdProc);

        user.user_mouse.configure_buttons(1);
        user.user_mouse.dolly.dolly[2]=-9;

        cb.display_function=Display;

        glut_add_interactor(&user);
        glut_add_interactor(&cb);
        glut_add_interactor(&console);
        glut_add_interactor(&swapbuffers);

        atexit(onExit);

        console.processCmd("exec autoexec.cs");

        console.add("Initialize complete. Starting glutMainLoop");
        glutMainLoop();

        return 0;
}
```

Теперь для того, чтобы изменить цвет чайника, например, на цвет морской волны, достаточно ввести команду

```
color 0 0.5 0.5
```

Но наша консольная команда пока не совсем полноценная — "настоящий" консольный обработчик команды `color` должен уметь выводить справку по команде, если команда запущена с ключом `/?`, а в случае ошибки возвращать `false` и сообщение об ошибке. Кроме того, очень желательно, чтобы команда `color`, запущенная без параметров, выводила в консоль информацию о текущем цвете. Такой консольный обработчик демонстрируется в листинге 2.20 (Ex21).

Листинг 2.20

```
bool CmdProc(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
       if (Cmd=="color")
       {
// Если параметр "/?", то показываем описание команды
              if ((Params.size()==1) && (Params[0]=="/?"))
              {
                     console->add("sets objects color");
                     console->add("");
                     console->add("color <red> <green> <blue>");
                     console->add("where");
                     console->add("  <red>, <green> and <blue> are
values between 0 and 1");
                     return true;
              }
// Если нет параметров, показывает текущий цвет
              if (Params.size()==0)
              {
                     char buf[200];
                     sprintf(buf, "current color is (%f, %f, %f)",
Color[0], Color[1], Color[2]);
                     console->add(buf);
                     return true;
              }
// Если три параметра, трактуем их как компоненты цвета
              if (Params.size()==3)
              {
                     Color[0]=atof(Params[0].c_str());
                     Color[1]=atof(Params[1].c_str());
```

```
                    Color[2]=atof(Params[2].c_str());
                    glutPostRedisplay();
                    return true;
                }
                else
                {
// В противном случае — неправильные параметры
                    console->add("Invalid parameters");
                    return false;
                }
        }

        return false;
}
```

На этом мы закончим это небольшое введение в библиотеку GLHE. Остальные возможности этой библиотеки мы будем рассматривать по мере необходимости в следующих главах.

2.4. Библиотека GLH_OBS — объектная надстройка над OpenGL

Как известно, библиотека OpenGL изначально разрабатывалась для языка C, который не поддерживает объектно-ориентированное программирование. Но, с другой стороны, в библиотеке имеется множество команд, работающих по следующей схеме.

1. Создание нового объекта OpenGL и получение его идентификатора.
2. Использование этого объекта в командах OpenGL. При этом идентификатор объекта явно предается в качестве одного из параметров команды.
3. "Ручное" удаление объекта OpenGL при выходе из программы.

Команды такого типа являются потенциальными кандидатами для помещения в объект, т. к. создание такой объектной надстройки сразу устраняет две потенциальные ошибки.

- Передача в команду OpenGL неправильного идентификатора объекта.
- Программист может забыть удалить объект при выходе из программы, что приведет к утечке памяти, или, что еще хуже, — видеопамяти.

Поэтому в библиотеку GLH была включена небольшая объектная надстройка над библиотекой OpenGL, которая находится в файле \OpenGL\include\glh\glh_obs.h. Для удобства мы будем называть ее библиотекой GLH_OBS.

В следующих разделах мы рассмотрим три класса этой библиотеки: `display_list`, `lazy_build_display_list` и `tex_object`. Первые два класса предназначены для работы с дисплейными списками, а третий — со списками текстур.

2.4.1. Класс *display_list*

Класс `display_list` инкапсулирует дисплейный список OpenGL. Основной принцип работы класса `display_list` можно описать следующими словами: при создании списка его OpenGL-идентификатор запоминается внутри объекта. Большинство команд OpenGL для работы со списками инкапсулируются в соответствующих методах класса. Эти методы принимают такие же параметры, как и команды OpenGL, за исключением идентификатора списка, который хранится в недрах объекта. Деструктор класса `display_list` содержит код, удаляющий дисплейный список. Поэтому, если объект дисплейного списка был создан как статический объект, то дисплейный список автоматически будет уничтожен при выходе из программы.

Класс `display_list` имеет очень простую структуру. Поэтому вместо того, чтобы рассматривать отдельные методы этого класса, я просто приведу его исходный код со своими комментариями.

```
class display_list
       {
       public:
// Конструктор. Создает пустой дисплейный список
              display_list()
                     : valid(false) {}
// Деструктор. Удаляет дисплейный список
              virtual ~display_list()
              { del(); }
// Выполняет дисплейный список (если он не пустой)
              void call_list()
              { if(valid) glCallList(dlist); }
// Создает новый дисплейный список (если он пустой — то сначала получает
// свободный идентификатор)
              void new_list(GLenum mode)
              { if(!valid) gen(); glNewList(dlist, mode); }
// Завершает создание дисплейного списка
              void end_list()
              { glEndList(); }
// Удаляет дисплейный список
              void del()
```

```
            { if(valid) glDeleteLists(dlist, 1); valid = false; }
// Проверяет, существует ли дисплейный список (связан ли данный
// объект с "настоящим" дисплейным списком
            bool is_valid() const { return valid; }

      private:
// Получает свободный идентификатор для нового дисплейного списка

            void gen() { dlist = glGenLists(1); valid=true; }
// Существует ли дисплейный список
            bool valid;
// Идентификатор дисплейного списка для команд OpenGL
            GLuint dlist;
      };
```

Для того чтобы продемонстрировать использование класса `display_list` на практике, я переписал пример (Ex19) с использованием этого класса:

```
display_list list0;
float Color[3]={0.1, 0.7, 0.2};
...

void Init()                          // Инициализация OpenGL
{
...
// Создание дисплейного списка
      list0.new_list(GL_COMPILE);
                        glColor3fv(&Color[0]);
            glutSolidTeapot(2);
      list0.end_list();
}

void Display()               // Обновление содержимого экрана
{
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

      list0.call_list();                  // Вывод объекта на экран
}
...
```

В результате этой переделки из программы исчез обработчик `onExit` — дисплейный список теперь удаляется автоматически при выходе из программы.

2.4.2. Класс *lazy_build_display_list*

Если класс `display_list` использует подход к дисплейными спискам, аналогичный OpenGL, то класс `lazy_build_display_list` использует значительно более "продвинутую" методику организации работы с дисплейными списками. Вместо того, чтобы "вручную" создавать дисплейный список, приложение должно передать классу адрес функции, которая "рисует" дисплейный список. Об остальном класс позаботится сам. Если дисплейный список надо обновить, то приложение сообщает об этом классу, который обновляет дисплейный список путем повторного вызова пользовательской функции генерации дисплейного списка.

Ниже приведен исходный код класса `lazy_build_display_list` с подробными комментариями:

```
        class lazy_build_display_list
        {
        public:
// Конструктор. Создает несуществующий список, который
// необходимо обновить при первой же возможности
// (need_rebuild=true). В качестве последнего параметра
// принимает адрес пользовательской функции "рисования"
// дисплейного списка
                lazy_build_display_list(void (* builder)() = 0)
                        : valid(false), needs_rebuild(true),
build_func(builder) {}
// Деструктор. Удаляет дисплейный список
                virtual ~lazy_build_display_list()
                { del(); }
// Устанавливает функцию "рисования" дисплейного списка
                void set_build_func( void (* builder)())
                { build_func = builder; }
// Выполняет дисплейный список
                void call_list()
                {
                        if(! valid) gen();
                        if(needs_rebuild) rebuild_list();
                        glCallList(dlist);
                }
// Удаляет дисплейный список
                void del()
```

```
                { if(valid) glDeleteLists(dlist, 1); valid = false;
needs_rebuild = true;}
// Проверяет, существует ли данный дисплейный список
                bool is_valid() const { return valid; }
// Пересоздает дисплейный список (реально дисплейный
// список пересоздается во время вызова метода call_list)
call_list()
                void rebuild() { needs_rebuild = true; }

        private:
// Получает свободный идентификатор для нового дисплейного списка

                void gen() { dlist = glGenLists(1); valid=true; }
// Пересоздает дисплейный список
                void rebuild_list()
                {
                        glNewList(dlist, GL_COMPILE);
                        if(build_func) (* build_func)();
                        glEndList();
                }
// Дисплейный список существует
                bool valid;
// Дисплейный список нуждается в обновлении
                bool needs_rebuild;
// Идентификатор дисплейного списка
                GLuint dlist;
// Адрес пользовательской функции рисования дисплейного списка
                void (* build_func)();
        };
```

Для демонстрации использования класса `lazy_build_display_list` на практике мы напишем программу, которая позволит пользователю при помощи консольной команды `object_type` выбрать тип объекта, который будет выводиться на экран (Ex23).

Команда `object_type` имеет следующий формат:

```
object_type <type>
```

где `type` может принимать следующие значения: `teapot`, `sphere` и `cube` (соответственно, для чайника, сферы и куба).

Например, для того, чтобы изменить текущий объект на куб, пользователю достаточно ввести в консоли следующую команду:

```
object_type cube
```

Текст программы приведен в листинге 2.21.

Листинг 2.21

```
enum {O_TEAPOT, O_CUBE, O_SPHERE};
int ObjectType=O_TEAPOT;

void BuildList();

lazy_build_display_list list0(BuildList);
float Color[3]={0.1, 0.7, 0.2};
...
void Display()               // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glColor3fv(&Color[0]);
       list0.call_list();                  // Вывод объекта на экран
}
// Обработчик консольных команд
bool CmdProc(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
...
// Если команда object_type
       if (Cmd=="object_type")
       {
              if (Params.size()==1)
              {
// Если параметр "/?", то выводим справку по команде
                     if (Params[0]=="/?")
                     {
                            console->add("sets objects type");
                            console->add("");
                            console->add("object_type <type>");
                            console->add("where");
```

```
                        console->add("  <type> one of 'cube',
'sphere' and 'teapot'");
                        console->add("");
                        console->add("Example:");
                        console->add("object_type cube");
                        console->add("sets object type to cube");
                        return true;
                }
// Если параметр "teapot", то объект будет чайник
                if (Params[0]=="teapot")
                {
                        ObjectType=O_TEAPOT;
// Дисплейный список надо обновить
                        list0.rebuild();
                        return true;
                }

                if (Params[0]=="sphere")
                {
                        ObjectType=O_SPHERE;
                        list0.rebuild();
                        return true;
                }

                if (Params[0]=="cube")
                {
                        ObjectType=O_CUBE;
                        list0.rebuild();
                        return true;
                }
        }

        console->add("Invalid parameters");
        return false;
    }

    return false;
}
```

```
// Функция рисования дисплейного списка
void BuildList()
{
// В зависимости от типа объекта рисуем этот объект
       switch(ObjectType)
       {
       case O_TEAPOT:
              glutSolidTeapot(2);
              break;
       case O_SPHERE:
              glutSolidSphere(2, 32, 32);
              break;
       case O_CUBE:
              glutSolidCube(2);
              break;
       }
}

int main(int argc, char* argv[])
{
       console.add("Initializing GLUT...");
       glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
       glutInitWindowSize(WinWidth,WinHeight);
       glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                              (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
       glutCreateWindow(Title.c_str());
       console.add("Initializing OpenGL...");

       Init();

       console.add("Initializing OpenGL Helper Library...");
       glut_helpers_initialize();

       console.title=Title;
       console.addCmd("color", "sets objects color", 0, CmdProc);
// Регистрируем обработчик команды "object_type"
       console.addCmd("object_type", "sets objects type", 0, CmdProc);
```

```
        user.user_mouse.configure_buttons(1);
        user.user_mouse.dolly.dolly[2]=-9;

        cb.display_function=Display;

        glut_add_interactor(&user);
        glut_add_interactor(&cb);
        glut_add_interactor(&console);
        glut_add_interactor(&swapbuffers);

        console.processCmd("exec autoexec.cs");

        console.add("Initialize complete. Starting glutMainLoop");
        glutMainLoop();

        return 0;
}
```

Но поскольку задавать тип объекта при помощи консольных команд довольно не удобно, пользователь может циклически переключать тип объекта при помощи клавиши <n>. Эта возможность реализуется при помощи пакетного файла autoexec.cs:

```
begin_script set_teapot_script
object_type teapot
bind n "run_script set_sphere_script"
end_script

begin_script set_sphere_script
object_type sphere
bind n "run_script set_cube_script"
end_script

begin_script set_cube_script
object_type cube
bind n "run_script set_teapot_script"
end_script

run_script set_teapot_script
```

Эта небольшая скриптовая программа очень похожа на программу из файла user.cs, запускающую бенчмарк по клавише <F2>, поэтому мы не будем ее рассматривать.

Возможно, у вас возник вопрос — какой класс лучше: `display_list` или `lazy_build_display_list`? Все зависит от ситуации. Если дисплейный список не изменяется, то лучше использовать класс `display_list`, в противном случае — `lazy_build_display_list`.

2.4.3. Класс *tex_object*

Класс `tex_object` предназначен для работы с текстурными объектами OpenGL (так называемыми текстурными списками). Работа с текстурными списками в OpenGL очень сильно похожа на работу с дисплейными списками. Класс `tex_object` очень похож на класс `display_list`.

Ниже приведен исходный текст класса `tex_object`:

```
        class tex_object
        {
        public:
// Конструктор класса tex_object. Создает несуществующий текстурный
// объект заданного типа
                tex_object(GLenum tgt)
                        : valid(false), target(tgt) {}
// Деструктор. Удаляет текстурный объект
                virtual ~tex_object()
                { del(); }
// Делает текстурный объект текущим
                void bind()
                { if(!valid) gen(); glBindTexture(target, texture); }

                // convenience methods
// Следующие 4 метода являются различными вариантами команды
// glTexParameter
                void parameter(GLenum pname, GLint i)
                { glTexParameteri(target, pname, i); }

                void parameter(GLenum pname, GLfloat f)
                { glTexParameterf(target, pname, f); }

                void parameter(GLenum pname, GLint * ip)
                { glTexParameteriv(target, pname, ip); }
```

```
            void parameter(GLenum pname, GLfloat * fp)
            { glTexParameterfv(target, pname, fp); }
// Активизирует наложение текстур того же типа, что и данный
// текстурный объект
            void enable() { glEnable(target); }
// Выключает наложение текстур того же типа, что и данный
// текстурный объект
            void disable() { glDisable(target); }

// Удаляет текстурный объект
            void del()
            { if(valid) glDeleteTextures(1, &texture); valid = false;
}
            bool is_valid() const { return valid; }
// Получает свободный идентификатор для нового текстурного объекта

            void gen() { glGenTextures(1, &texture); valid=true; }
// Связан ли данный объект с настоящим текстурным списком
            bool valid;
// Идентификатор текстурного объекта
            GLuint texture;
// Тип текстурного объекта (GL_TEXTURE_1D, GL_TEXTURE_2D и т.д.)
            GLenum target;
      };
```

Сам класс `tex_object` используется очень редко. Как правило, в реальных проектах используются его потомки: `tex_object_1D` и `tex_object_2D`, которые соответственно предназначены для работы с одномерными и двухмерными текстурами. Существует еще ряд потомков этого класса, например, `tex_object_3D` и `tex_object_cube_map`. Но их мы рассмотрим позже, в разделах, связанных с соответствующими типами текстур. Объекты `tex_object_1D` и `tex_object_2D` различаются между собой лишь конструктором:

```
      class tex_object_1D : public tex_object
      { public: tex_object_1D() : tex_object(GL_TEXTURE_1D) {} };

      class tex_object_2D : public tex_object
      { public: tex_object_2D() : tex_object(GL_TEXTURE_2D) {} };
```

Для демонстрации использования класса `tex_object` (точнее, `tex_object_2D`) я написал небольшую программу, которая выводит на экран деревянный чайник (рис. 2.9) (Ex23).

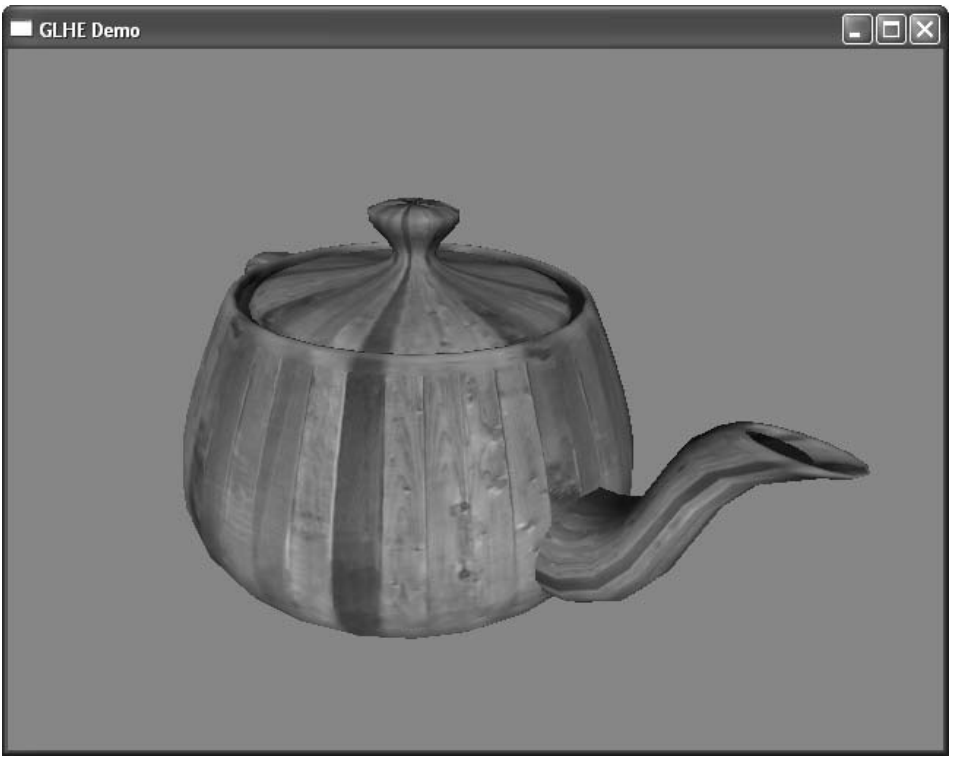


Рис. 2.9. Обыкновенный деревянный чайник. Текстура взята из NVIDIA OpenGL SDK

Исходный код программы приведен в листинге 2.22.

Листинг 2.22

```
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include "glh_glut_ext.h"
#include "gl/glaux.h"

using namespace std;
using namespace glh;
```

```
int WinWidth=640;            // Ширина окна
int WinHeight=480;           // Высота окна

// Текстурный объект
tex_object_2D wood;
// Указатель на загруженное изображение
AUX_RGBImageRec* pWoodImage;
display_list list0;
float Color[3]={1, 1, 1};

glut_console console;
glut_simple_user_interface user(&console);
glut_callbacks cb;
glut_swapbuffers swapbuffers;

const string Title="GLHE Demo";

void Init()                          // Инициализация OpenGL
{
       glEnable(GL_DEPTH_TEST);
       glEnable(GL_LIGHTING);
       glEnable(GL_LIGHT0);
       glEnable(GL_COLOR_MATERIAL);

       glClearColor(0.5, 0.5, 0.75, 1);

       list0.new_list(GL_COMPILE);  // Создание дисплейного
                                    // списка объекта (чайника)
              glColor3fv(&Color[0]);
              glutSolidTeapot(2);
       list0.end_list();
// Загружаем изображение из файла
       pWoodImage=auxDIBImageLoad("..\\wood.bmp");
// Устанавливаем параметры текстуры и загружаем ее в видеопамять
       wood.bind();
       wood.parameter(GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
       wood.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

```
        gluBuild2DMipmaps(wood.target, 3, pWoodImage->sizeX, pWoodImage-
>sizeY, GL_RGB, GL_UNSIGNED_BYTE,
                                        pWoodImage->data);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
}

void Display()                // Обновление содержимого экрана
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Активизируем текстуру, рисуем объект и выключаем наложение текстуры
        wood.enable();
        list0.call_list();                  // Вывод объекта на экран
        wood.disable();
}

void onExit()
{
// При выходе из программы выгружаем текстуру из памяти (сначала
// изображение текстуры, а затем структуру AUX_RGBImageRec)
        delete[] pWoodImage->data;
        delete pWoodImage;
}

int main(int argc, char* argv[])
{
        console.add("Initializing GLUT...");
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
        glutInitWindowSize(WinWidth,WinHeight);
        glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                                    (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
        glutCreateWindow(Title.c_str());
        console.add("Initializing OpenGL...");

        Init();

        console.add("Initializing OpenGL Helper Library...");
        glut_helpers_initialize();
```

```
    console.title=Title;

    user.user_mouse.configure_buttons(1);
    user.user_mouse.dolly.dolly[2]=-9;

    cb.display_function=Display;

    glut_add_interactor(&user);
    glut_add_interactor(&cb);
    glut_add_interactor(&console);
    glut_add_interactor(&swapbuffers);

    console.processCmd("exec autoexec.cs");

    atexit(onExit);

    console.add("Initialize complete. Starting glutMainLoop");
    glutMainLoop();

    return 0;
}
```

Для работы загрузки текстуры из файла bmp используется стандартная библиотека GL_AUX, описание которой можно найти, например, в [5].

Программа ничем не отличается от стандартной OpenGL-программы за исключением того, что вместо команд OpenGL для работы со списками используются методы соответствующих классов.

Заключение

В этой главе мы рассмотрели библиотеку NVIDIA OpenGL Helper Library (GLH). Эта библиотека содержит множество классов, которые могут значительно облегчить жизнь программисту. Эти классы можно условно разделить на 3 большие группы.

- Математические функции (модуль GLH_LINEAR).
- Объектно-ориентированная надстройка над GLUT, основанная на интеракторах (модули GLH_GLUT и GLH_CONVENIENCE).
- Классы, инкапсулирующие функции OpenGL (модуль GLH_OBS).

Мы рассмотрели каждую из этих групп в соответствующих разделах. Для демонстрации использования классов библиотеки на практике было написано множество примеров.

Кроме того, в этой главе было рассмотрено расширение библиотеки GLH — OpenGL Helper Library Extension (GLHE), добавляющее в библиотеку GLH ряд новых возможностей, в частности, интерактор консоли `glut_console` с поддержкой скриптов. Использование библиотеки GLHE позволяет значительно сократить размер демонстрационных программ: к примеру, программа, выводящая на экран цветной чайник, который пользователь может вращать и перемещать с использованием мыши и клавиатуры, займет менее 90 строк.

Глава 3

Библиотека NV_MATH

Во *главе 2* мы рассмотрели математическую библиотеку GLH_LINEAR, входящую в состав NVIDIA OpenGL Helper Library. Это довольно неплохая математическая библиотека, возможностей которой вполне хватает для рядового 3D-приложения. Тем не менее в состав NVIDIA OpenGL SDK входит еще одна математическая библиотека — NV_MATH. Зачем она нужна? Если честно — не знаю. Скорее всего, так исторически сложилось, что одни примеры из NVIDIA OpenGL SDK использовали библиотеку GLH_LINEAR, а другие NV_MATH. В результате для того, чтобы оставить только одну библиотеку, надо переписать множество примеров. А это скучно и не интересно. Правда, справедливости ради, надо отметить, что эти библиотеки отнюдь не равноценны — библиотека NV_MATH в целом обладает большими возможностями, чем GLH_LINEAR. С другой стороны, библиотеку GLH_LINEAR удобнее использовать из-за ее хорошей интеграции с другими компонентами OpenGL Helper Library.

Мне кажется, для того, чтобы нормально ориентироваться в примерах NVIDIA OpenGL SDK, программисту очень желательно знать обе библиотеки. Поэтому в данной главе мы рассмотрим основные нюансы использования библиотеки NV_MATH и ее отличия от библиотеки GLH_LINEAR.

Перед использованием библиотеки необходимо подключить к проекту файлы nv_math, nv_algebra.h и nv_math.lib. Помните, что в составе NVIDIA SDK имеется два варианта файла nv_math.lib: RELEASE и DEBUG, причем быстродействие последнего значительно ниже.

3.1. Работа с векторами

В библиотеке nv_math за работу с 2-, 3- и 4-мерными векторами отвечают классы `vec2`, `vec3` и `vec4` соответственно. Наряду с этими классами в библиотеке NV_MATH имеются классы транспонированных векторов `vec2t`, `vec3t` и `vec4t`.

В этой главе мы рассмотрим работу с классом `vec4`. Классы `vec2` и `vec3` являются просто модификацией этого класса с уменьшенным числом ко-

ординат, поэтому все сказанное далее будет, с небольшими оговорками, верно и для них. Классы транспонированных векторов мы рассмотрим в конце раздела.

Итак, класс `vec4` обладает восемью конструкторами:

```
// Тип nv_scalar является псевдонимом float
typedef float nv_scalar;
// Создает вектор с неопределенным значением
vec4() { }
// Создает вектор и присваивает его компонентам соответствующие значения
vec4(nv_scalar x, nv_scalar y, nv_scalar z, nv_scalar w) : x(x), y(y),
z(z), w(w) { }
// Создает вектор и присваивает его компонентам значения из массива,
// на который указывает параметр конструктора
vec4(const nv_scalar* xyzw) : x(xyzw[0]), y(xyzw[1]), z(xyzw[2]),
w(xyzw[3]) { }
// Создает четырехмерный вектор на основе трехмерного вектора.
// Последний компонент (w) полагается равным единице
vec4(const vec3& u) : x(u.x), y(u.y), z(u.z), w(1.0f) { }
// Создает четырехмерный вектор на основе трехмерного
// транспонированного вектора. Последний компонент (w)
// полагается равным единице
vec4(const vec3t& u);
// Создает вектор на основе существующего
vec4(const vec4& u) : x(u.x), y(u.y), z(u.z), w(u.w) { }
// Создает вектор на основе существующего транспонированного вектора
vec4(const vec4t& u);
```

Над классами векторов можно выполнять различные математические операции, как над обычными числами. Для этого библиотека NV_MATH перегружает большинство операторов C++ наподобие `+`, `-`, `×`, `/`, `+ =` и т. д. Обратите внимание, что оператор `×` просто перемножает соответствующие координаты векторов, а не вычисляет скалярное произведение векторов. Доступ к компонентам вектора осуществляется при помощи перегруженного оператора `[]`. Кроме того, в NV_MATH определены константы нулевого (0, 0, 0, 0) и единичного (1, 1, 1, 1) векторов `vec4_null` и `vec4_one`.

Ниже приведен пример вычисления выражения (Ex01) (листинг 3.1).

a = (1, 1, 1, 1); b = (1, 2, 3, 4); X = 3 × a+2 × b

Для упрощения программы векторы a и b задаются как константы.

Листинг 3.1

```
#include <iostream>
#include <string>
#include <nv_math/nv_algebra.h>

using namespace std;

void output(vec4& x)  //Выводит вектор на экран
{
        for (unsigned int i=0; i<4; i++)
        {
                cout<<"x["<<i<<"]="<<x[i]<<endl;
        }
}

void main()
{
        vec4 a(vec4_one);     //вектор a
        vec4 b(1,2,3,4);      //вектор b

        vec4 x=3*a+2*b;

        output(x);

        getchar();
};
```

Рассмотрим более подробно определение класса `vec4`:

```
struct vec4
{
   ... // определение методов пропущено
union {
        struct {
            nv_scalar x,y,z,w;          // геометрические координаты
        };
        struct {
            nv_scalar s,t,r,q;          // координаты текстуры
        };
```

```
        nv_scalar vec_array[4];     // доступ к массиву координат
    };
};
```

Из этого определения видно, что координаты вектора могут быть интерпретированы тремя способами:

- геометрические координаты;
- текстурные координаты;
- массив координат.

Это очень полезно при использовании команд OpenGL, допускающих векторную форму записи. Например, вывод вершины на экран при помощи команды `glVertex` может быть записан следующим образом:

```
vec3 vertex;
glVertex3f(vertex.x, vertex.y, vertex.z);
```

Если же рассматривать вектор `vertex` как массив, то мы можем воспользоваться векторным вариантом команды `glVertex3f`:

```
glVertex3fv(&vertex.vec_array[0]);
```

Ясно, что второй вариант — предпочтительнее. Но и его можно еще немного упростить, если учесть, что поля `x` и `vec_array[0]` ссылаются на одну и ту же область памяти (правда, такое упрощение на быстродействие уже не повлияет):

```
glVertex3fv(&vertex.x);
```

Над классами `vec2`, `vec3` и `vec4` можно выполнять множество операций. Операции, встречающиеся чаще всего, приведены в табл. 3.1.

Таблица 3.1. *Основные операции с векторами*

Название функции	Назначение
`vec4 & normalize(vec4 & u)`	Нормализует вектор. Нормализация 3-мерного вектора также может быть выполнена при помощи одноименного метода
`nv_scalar nv_sq_norm (const vec4 & n)`	Вычисляет квадрат модуля вектора
`nv_scalar nv_norm (const vec4 & n)`	Вычисляет модуль вектора
`vec3 & reflect(vec3 & r, const vec3 & n, const vec3 & l)`	Вычисляет отраженный вектор. `L` — исходный вектор, `n` — вектор нормали, `r` — отраженный вектор (рис. 3.1)
`vec3 & madd(vec3 & u, const vec3 & v, const nv_scalar & lambda);`	Быстрое вычисляет выражение `u = v × lambda + u`

Таблица 3.1 *(окончание)*

Название функции	Назначение
`vec4 & scale(vec4 & u, const nv_scalar s)`	Масштабирует вектор. Иными словами, умножает вектор на число
`nv_scalar & dot(nv_scalar & u, const vec4 & v, const vec4 & w)`	Вычисляет скалярное произведение векторов ($v = u \cdot v$)
`vec3 & cross(vec3 & u, const vec3 & v, const vec3 & w);`	Вычисляет векторное произведение векторов ($v = u \times w$)

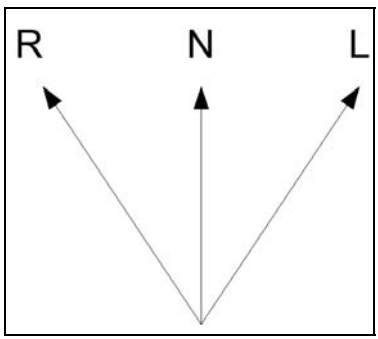


Рис. 3.1. Параметры функции `reflect`

При использовании этих функций надо учитывать одну особенность — *нельзя* указывать в качестве аргумента и результата этих функций одну и ту же переменную. То есть если функция `f(c, a, b)` принимает аргументы `a` и `b`, а возвращает `c`, то такой вызов функции, как `f(x, x, y)`, будет, скорее всего, работать неправильно.

Для демонстрации использования этой функции на практике напишем простую программу, решающую следующую задачу (Ex02).

На зеркальную плоскость, проходящую через точки a(0, 0, 0), b(1, 0, 1) и c(0, 1, 0), падает луч света, направление которого задается вектором L(1, 1, −1). Необходимо определить направление отраженного луча (вектор R).

Для того чтобы решить эту задачу, нам надо найти нормаль к плоскости, а затем воспользоваться функцией `reflect`. Нормаль вектора к плоскости можно найти через произведение двух неколлинеарных векторов, параллельных данной плоскости. Программа, решающая данную задачу, приведена в листинге 3.2.

Листинг 3.2

```
#include <iostream>
#include <string>
#include <nv_math/nv_algebra.h>

using namespace std;

void output(vec3& r)  // вывод вектора на экран
{
        for (unsigned int i=0; i<3; i++)
        {
                cout<<"r["<<i<<"]="<<r[i]<<endl;
        }
}

void main()
{
// Исходные данные
        vec3 a(0, 0, 0);
        vec3 b(1, 0, 0.5);
        vec3 c(0, 1, 0);
        vec3 l(-1, -1, -1);

        vec3 n;
// Находим вектор, перпендикулярный плоскости
        cross(n, b-a, c-a);
// Находим нормаль
        n.normalize();
        vec3 r;
// Находим вектор отражения от плоскости. Так как функция reflect
// вычисляет вектор отражения относительно нормали (а не от плоскости),
// в функцию reflect подставляется вектор "-l"
        reflect(r, n, -l);

        output(r);

        getchar();
};
```

В начале главы упоминалось, что, наряду с обычными векторами, в библиотеке NV_MATH имеются классы транспонированных векторов: `vec2t`, `vec3t` и `vec4t`. Классы транспонированных векторов очень похожи на классы обычных векторов. Ниже приведено определение класса `vec4t` (определение классов `vec3t` и `vec2t` имеет аналогичную структуру):

```
struct vec4t {
// Создает новый вектор с неопределенным значением
    vec4t() { }
// Создает новый вектор и присваивает его компонентам соответствующие
// значения
    vec4t(nv_scalar x, nv_scalar y, nv_scalar z, nv_scalar w) : x(x),
y(y), z(z), w(w) { }
// Создает вектор и присваивает его компонентам значения, которые берутся
// из массива, на который указывает параметр конструктора
    vec4t(const nv_scalar* xyzw) : x(xyzw[0]), y(xyzw[1]), z(xyzw[2]),
w(xyzw[3]) { }
// Создает вектор на основе другого транспонированного вектора
       vec4t(const vec4t& u) : x(u.x), y(u.y), z(u.z), w(u.w) { }
// Создает вектор на основе транспонированного трехмерного вектора.
// Четвертый компонент (w) полагается равным 1
       vec4t(const vec3t& u) : x(u.x), y(u.y), z(u.z), w(1.0f) { }
// Возвращает "нормальный" (не транспонированный вектор)
       vec4 T() const { return vec4(x, y, z, w); }
// Компоненты вектора
       nv_scalar x,y,z,w;
};
```

Над транспонированными векторами можно выполнять различные операции, но набор этих операций значительно меньше, чем у обычных векторов. Я не знаю, для чего NVIDIA включила классы транспонированных векторов в библиотеку NV_MATH — ведь любая программа может обойтись без использования этих классов. Например, умножение матрицы на транспонированный вектор справа равносильно умножению этой матрицы на обычный вектор слева.

Если у вас еще остались какие-либо вопросы относительно векторов библиотеки NV_MATH, то ответы на них вы сможете найти в файлах \NVSDK\OpenGL\\src\libs\nv_math\nv_algebra.cpp
\NVSDK\OpenGL\\include\nv_math\nv_algebra.h.

3.2. Работа с матрицами

В библиотеке NV_MATH имеются классы `mat3` и `mat4`, которые отвечают за работу с квадратными матрицами размера 3×3 и 4×4. Так как эти классы отличаются только количеством элементов, мы рассмотрим работу только с классом `mat4`.

Класс `mat4` имеет четыре конструктора:

```
Struct mat4
{
// Создает новую матрицу с неопределенным значением
    mat4();
// Создает матрицу на основе указателя на одномерный массив, в котором
// хранится матрица 4x4. Порядок элементов в массиве должен быть
// аналогичен порядку элементов в матрицах, которые используются в
// командах OpenGL
    mat4(const nv_scalar * array);
// Создает матрицу на основе другой матрицы
    mat4(const mat4 & M);
// Создает матрицу на основе 16-ти значений ее элементов (элементы
// передаются построчно)
    mat4( const nv_scalar& f0,  const nv_scalar& f1,  const nv_scalar&
f2,  const nv_scalar& f3,
                const nv_scalar& f4,  const nv_scalar& f5,  const
nv_scalar& f6,  const nv_scalar& f7,
                const nv_scalar& f8,  const nv_scalar& f9,  const
nv_scalar& f10, const nv_scalar& f11,
                const nv_scalar& f12, const nv_scalar& f13, const
nv_scalar& f14, const nv_scalar& f15 )
 : a00( f0 ), a10( f4 ), a20( f8 ), a30( f12),
            a01( f1 ), a11( f5 ), a21( f9 ), a31( f13),
            a02( f2 ), a12( f6 ), a22( f10), a32( f14),
            a03( f3 ), a13( f7 ), a23( f11), a33( f15) { }
// Объявление различных идентификаторов для более удобного доступа к
// элементам
    union {
// Доступ к элементам по индексам
        struct {
            nv_scalar a00, a10, a20, a30;   // standard names
//for components
            nv_scalar a01, a11, a21, a31;   // standard names
```

```
// for components
            nv_scalar a02, a12, a22, a32;   // standard names
// for components
            nv_scalar a03, a13, a23, a33;   // standard names
// for components
        };
// Доступ к элементам по индексам (индексы нумеруются от 1)
        struct {
            nv_scalar _11, _12, _13, _14;   // standard names
// for components
            nv_scalar _21, _22, _23, _24;   // standard names
// for components
            nv_scalar _31, _32, _33, _34;   // standard names
// for components
            nv_scalar _41, _42, _43, _44;   // standard names
// for components
        };
        union {
// Доступ к элементам по названиям, принятым в 3D-графике, удобно
// использовать, когда матрица предназначена для аффинных преобразований
            struct {
       nv_scalar b00, b10, b20, p; // standard names for components
       nv_scalar b01, b11, b21, q; // standard names for components
       nv_scalar b02, b12, b22, r; // standard names for components
       nv_scalar x, y, z, w;       // standard names for components
            };
        };
// Доступ к элементам матрицы, как к одномерному массиву. Удобно
// использовать в матричных командах OpenGL
        nv_scalar mat_array[16];     // array access
    };
};
```

Первый конструктор создает неинициализированную матрицу, второй — копию существующей матрицы. Третий конструктор предназначен для инициализации матрицы определенным значением. Это значение должно храниться в одномерном массиве, причем порядок элементов должен быть сле-

дующим {a00. a10, a20, a30, a01, a11, a21, ..., a33}[1]. В следующем примере создается матрица $v = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$:

```
float z[16]={1, 5, 9, 13,
             2 ,6, 10,14,
               3, 7, 11,15,
               4, 8, 12,16};
mat4 m1(&z[0]);
mat4 m2(m1);
mat4 m3(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16);
```

Если же нам надо создать единичную матрицу, то задача упрощается — в NV_MATH имеются единичные матрицы 3×3 и 4×4, которые называются соответственно `mat3_id` и `mat4_id`.

Класс `mat4` имеет набор перегруженных операций, аналогичный семейству классов `vec`. Единственное отличие заключается в работе оператора `[]`, который теперь возвращает строку матрицы класса `vec4`. Но класс вектора тоже имеет перегруженный оператор `[]`, следовательно, доступ к элементу матрицы можно осуществлять при помощи комбинации операторов `[строка][столбец]`.

Приложение может работать с элементами матриц, как с одномерным массивом при помощи поля `mat_array`. Если же вам надо работать с полем `mat_array`, как с двухмерным массивом, то воспользуйтесь следующим выражением: `mat_array[строка+столец*4]`. Для классов `mat2` и `mat3` это будут выражения `mat_array[строка+столец*2]` и `mat_array[строка+столец*3]` соответственно. Далее приведен пример функции ввода значений матрицы с консоли:

```
void input(mat3& a)   // ввод матрицы с клавиатуры
{
        for (unsigned int i=0; i<3; i++)
        {
                for (unsigned int j=0; j<3; j++)
                {
                        cout<<"a["<<i<<","<<j<<"]=";
                        cin>>a.mat_array[j*3+i];
```

[1] Первая цифра — номер строки, вторая — номер столбца.

```
            }
        }
}
```

Вы наверно уже заметили, что порядок хранения элементов в классах матриц отличается от общепринятого в C++. Если вы думаете, что NVIDIA специально решила испортить вам жизнь, то сильно ошибаетесь. Дело в том, что эта особенность позволяет использовать матрицы в командах OpenGL без дополнительного транспонирования. OpenGL не использует стандартный формат хранения матриц — матрицы хранятся в транспонированном виде. Например, вы можете получить доступ к матрице модели с использованием одной команды:

```
mat4 m;
glGetFloatv(GL_MODELVIEW_MATRIX, &m.mat_array[0]);
```

В табл. 3.2 приведены основные функции, работающие с матрицами (кроме аффинных преобразований).

Таблица 3.2. *Функции для работы с матрицами*

Название функции	Назначение
`vec4 & mult(vec4 & u, const mat4 & M, const vec4 & v)`	Умножает матрицу (M) на вектор (v)
`vec3 & mult_dir(vec3 & u, const mat4 & M, const vec3 & v);`	Умножает подматрицу 3×3 (M=$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ q_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}$*) на вектор v. Кстати эта подматрица 3×3 содержит только информацию о повороте и масштабировании без переноса, что часто бывает очень полезно.
`vec3 & mult_pos(vec3 & u, const mat4 & M, const vec3 & v)`	Умножает трехмерный вектор на матрицу 4×4. Перед умножением вектор переводится в четырехмерный (четвертый компонент полагается равным 1). После умножения все компоненты вектора делятся на четвертый компонент (w), который впоследствии отбрасывается
`mat4 & transpose (mat4 & B)`	Транспонирует матрицу
`mat4 & invert(mat4 & B, const mat4 & A)`	Вычисляет обратную матрицу ($B = A^{-1}$)
`nv_scalar det(const mat3 & A)`	Находит определитель

* Нумерация элементов матрицы начинается с 1.

В качестве примера ниже приведена программа решения системы из трех уравнений по формуле $X = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix}^{-1} \times \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$ (Ex03) (листинг 1.3).

Листинг 1.3

```
// Если определена переменная test, то программа переходит в тестовый
// режим – решает встроенное уравнение, взятое из [10]

//#define test

#include <iostream>
#include <string>
#include <nv_math/nv_algebra.h>

using namespace std;

void input(mat3& a)   // ввод матрицы с клавиатуры
{
        for (unsigned int i=0; i<3; i++)
        {
                for (unsigned int j=0; j<3; j++)
                {
                        cout<<"a["<<i<<","<<j<<"]=";
                        cin>>a.mat_array[j*3+i];
                }
        }
}

void input(vec3& b)   // ввод вектора с клавиатуры
{
        for (unsigned int i=0; i<3; i++)
        {
                cout<<"b["<<i<<"]=";
                cin>>b.vec_array[i];
        }
}
```

```
void output(vec3& x)  // вывод вектора на экран
{
	for (unsigned int i=0; i<3; i++)
	{
		cout<<"x["<<i<<"]="<<x.vec_array[i]<<endl;
	}
}

void main()
{
	mat3 a;		//матрица коэффициентов a00..a22
	vec3 b;		//вектор свободных членов

	mat3 inv_a;	//обратная матрица
	vec3 x;		//вектор корней уравнения

/*
	если определена переменная test, то ввод тестового набора данных,
	иначе ввод данных с клавиатуры
*/

#ifndef test
	input(a);
	input(b);
#else
	a.a00=4;
	a.a01=0.24;
	a.a02=-0.08;
	a.a10=0.09;
	a.a11=3;
	a.a12=-0.15;
	a.a20=0.04;
	a.a21=-0.08;
	a.a22=4;

	b.x=8;
	b.y=9;
	b.z=20;
#endif
```

```
        invert(inv_a, a);     // вычисление обратной матрицы
        mult(x, inv_a, b);    // умножение обратной матрицы на вектор
свободных членов

        output(x);

        getchar();
        getchar();
};
```

Для закрепления основ работы с матрицами рассмотрим решение системы из трех уравнений методом Крамера (Ex04) (листинг 3.4).

Листинг 3.4

```
//#define test

#include <iostream>
#include <string>
#include <nv_math/nv_algebra.h>

using namespace std;

...

void output(vec3& x)  // вывод вектора на экран
{
        for (unsigned int i=0; i<3; i++)
        {
                cout<<"x["<<i<<"]="<<x.vec_array[i]<<endl;
        }
}

void main()
{
        mat3 a;         // матрица коэффициентов a00..a22
        vec3 b;         // вектор свободных членов

        vec3 x;         // вектор корней уравнения
```

```
/*
       если определена переменная test, то ввод тестового набора данных,
       иначе ввод данных с клавиатуры
*/

#ifndef test
       input(a);      // ввод матрицы коэффициентов
       input(b);      // ввод вектора свободных членов
#else
...
#endif

// Временная матрица, у которой в цикле заменяются столбцы
       mat3 a1(a);
       for (int i=0; i<3; i++)
       {
              vec3 col;
              col.x=a1[0][i];              // сохраняем столбец матрицы
              col.y=a1[1][i];
              col.z=a1[2][i];

              a1.mat_array[0+i*3]=b.x;     // заменяем столбец матрицы
              a1.mat_array[1+i*3]=b.y;     // столбцом свободных членов
              a1.mat_array[2+i*3]=b.z;

              x.vec_array[i]=det(a1)/det(a);      // находим корень
                                                  // уравнения

              a1.mat_array[0+i*3]=col.x;   // восстанавливаем столбец
                                           // матрицы
              a1.mat_array[1+i*3]=col.y;
              a1.mat_array[2+i*3]=col.z;
       }

       output(x);

       getchar();
// Если ввод был завершен клавишей <Enter>, то для остановки программы
// надо 2 раза вызвать функцию getchar
```

```
#ifndef test
       getchar();
#endif
};
```

Теперь, после того, как мы разобрались с векторами и матрицами, можно переходить к самой интересной части NV_MATH.

3.3. Выполнение аффинных преобразований

Как вы знаете, аффинные преобразования сводятся к перемножению матриц размером 4×4. В предыдущих двух разделах мы уже рассмотрели основные операции над матрицами библиотеки NV_MATH. Поэтому при желании вы можете самостоятельно реализовать аффинные преобразования. Правда, матрицы переноса, поворота и т. д. придется рассчитывать вручную, что само по себе скучно и не интересно.

К счастью, библиотека NV_MATH позволяет сильно упростить этот процесс. Дело в том, что в составе этой библиотеки имеется набор функций для вычисления типовых матриц аффинных преобразований.

Класс `mat4` содержит два метода, предназначенных для работы с аффинными преобразованиями. Синтаксис этих методов очень похож на синтаксис аналогичных команд OpenGL (`glTranslatef` и `glRotatef`). Первый из них — `set_translation` — вычисляет матрицу переноса. В качестве параметра он принимает вектор смещения (класс `vec3`):

```
void set_translation(const vec3 & t);
```

Для вычисления матрицы вращения используется группа методов `set_rot`:

```
// Выполняет поворот, который задается кватернионом[2] q
void set_rot(const quat & q);
// Поворот задается матрицей 3×3
void set_rot(const mat3 & M);
// Поворот задается связкой: угол поворота + ось, вокруг которой
// происходит поворот. Угол поворота задается в радианах
void set_rot(const nv_scalar & theta, const vec3 & v);
// Осуществляет поворот, совмещающий вектор u с вектором v
void set_rot(const vec3 & u, const vec3 & v);
```

[2] Класс `quat`, отвечающий за работу с кватернионами, мы рассмотрим в *разд. 3.4*.

Обратите внимание на то, что эти методы не вычисляют новые матрицы модели. Они просто замещают соответствующие коэффициенты у существующих матриц. Поэтому для корректной работы желательно вызывать эти методы у единичных матриц.

Для демонстрации использования класса `quaternionf` на практике я переписал пример Ex06 из *главы 1* без использования матричных функций OpenGL для работы с матрицей модели (Ex05). Для этого пришлось внести небольшие изменения в функцию `Display` (листинг 3.5).

Листинг 3.5

```
// Обновление содержимого экрана
void Display()
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glPushMatrix();
// Матрица модели
              mat4 modelview(mat4_id);
// Матрица переноса
              mat4 translate(mat4_id);
// Матрица поворота вокруг оси x
              mat4 rotatex(mat4_id);
// Матрица поворота вокруг оси y
              mat4 rotatey(mat4_id);
// Вычисляем соответствующие матрицы
              translate.set_translation(vec3(tx, ty, tz));
              rotatex.set_rot(rx*nv_to_rad, vec3(1, 0, 0));
              rotatey.set_rot(ry*nv_to_rad, vec3(0, 1, 0));
// Получаем итоговую матрицу модели
              modelview=translate*rotatex*rotatey;
// Умножаем текущую матрицу на полученную матрицу модели
              glMultMatrixf(&modelview.a00);

// Вывод объекта на экран
              glCallList(list);
       glPopMatrix();
```

```
        glutSwapBuffers();
...
}
```

Обратите внимание, что для перевода градусов в радианы используется константа `nv_to_rad`. Кстати, в библиотеке NV_MATH, имеется константа для обратного перевода — `nv_to_deg`:

```
#define nv_to_rad            nv_pi / nv_scalar(180)
#define nv_to_deg            nv_scalar(180) / nv_pi
```

В библиотеке NV_MATH также имеются три функции, которые могут строить заданные матрицы-проекции (табл. 3.3).

Таблица 3.3. *Функции библиотеки NV_MATH для расчета матриц проекции**

Функция	Аналогичная команда OpenGL	Описание
`mat4 & look_at(mat4 & M, const vec3 & eye, const vec3 & center, const vec3 & up)`	`gluLookAt`	Создает видовую матрицу на основе положения наблюдателя и направления его взгляда
`mat4 & frustum(mat4 & M, const nv_scalar l, const nv_scalar r, const nv_scalar b, const nv_scalar t, const nv_scalar n, const nv_scalar f)`	`glFrustum`	Рассчитывает матрицу перспективной проекции
`mat4 & perspective(mat4 & M, const nv_scalar fovy, const nv_scalar aspect, const nv_scalar n, const nv_scalar f)`	`gluPerspective`	Рассчитывает матрицу перспективной проекции

* Подробную информацию об этих функциях смотрите в описании соответствующих команд OpenGL.

Поэтому пример Ex05 можно полностью переписать без использования матричных функций (Ex06) OpenGL (кроме финального умножения матрицы на вектор), заменив строку

```
gluPerspective(45,GLdouble(WinWidth)/WinHeight,1,100);
```

на

```
mat4 Projection;
perspective(Projection,  45, GLdouble(WinWidth)/WinHeight, 1, 100);
glMultMatrixf(&Projection.mat_array[0]);
```

В принципе, можно было бы пойти дальше и производить перемножение матрицы проекции, матрицы модели и координат вершины средствами NV_MATH, но это приведет к усложнению программы, т. к. в этом случае придется самостоятельно рассчитывать освещение.

В заключение этого раздела мы перепишем пример Ex06 *главы 1* с использованием библиотеки NV_MATH (Ex07). В примере демонстрируется использование функции `mult_pos`, которая позволяет отказаться от использования однородных координат там, где они не нужны. Но этой возможностью не стоит злоупотреблять — функция `mult_pos` медленнее, чем простое умножение четырехмерного вектора на матрицу.

Изменения коснулись только функции `Display()` (листинг 3.6).

Листинг 3.6

```
void Display()                  // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glPushMatrix();
// Перемещение и поворот прямоугольника A
              glTranslatef(tx,ty,tz);
              glRotatef(rx,1,0,0);
              glRotatef(ry,0,1,0);

              glBegin(GL_QUADS);
                     glVertex2f(-1, -1);
                     glVertex2f(-1,  1);
                     glVertex2f( 1,  1);
                     glVertex2f( 1, -1);
              glEnd();

              glPushMatrix();
// Смещаем плоскость на величину t вверх и 1 вправо
                     glTranslatef(1, 0, t);

                     // проводим поворот на r градусов вокруг левого
                     // края
                     glTranslatef(1, 0, 0);
                     glRotatef(r, 0, 1, 0);
                     glTranslatef(1, 0, 0);
```

```
                glBegin(GL_QUADS);
                        glVertex2f(-1, -1);
                        glVertex2f(-1,  1);
                        glVertex2f( 1,  1);
                        glVertex2f( 1, -1);
                glEnd();
        glPopMatrix();



// Координаты правой стороны прямоугольника C
        vec3 v1(-1, -1, 0);
        vec3 v2(-1, 1, 0);

        mat4 Translate1(mat4_id);
        mat4 Translate2(mat4_id);
        mat4 Rotate(mat4_id);

// Рассчитываем матрицу первого переноса. Обратите внимание: два
// оператора glTranslate объединены в один
        Translate1.set_translation(vec3(2, 0, t));
// Рассчитываем матрицу второго переноса (после поворота)
        Translate2.set_translation(vec3(1, 0, 0));
// Рассчитываем матрицу поворота
        Rotate.set_rot(r*nv_to_rad, vec3(0, 1, 0));

        mat4 m1=Translate1*Rotate*Translate2;

// Временный вектор — в функциях NV_MATH нельзя указывать одну и ту же
// переменную в качестве входного и выходного параметра
        vec3 tmp;

        mult_pos(tmp, m1, v1);
        v1=tmp;

        mult_pos(tmp, m1, v2);
        v2=tmp;

        glBegin(GL_QUADS);
                glVertex2f( 1,  1);
```

```
                    glVertex2f( 1, -1);
                    glVertex3fv(&v1.vec_array[0]);
                    glVertex3fv(&v2.vec_array[0]);
            glEnd();
      glPopMatrix();

      glutSwapBuffers();
...
}
```

3.4. Использование кватернионов

Для работы с кватернионами в библиотеке NV_MATH используется класс `quat`, аналогичный классам `quaternion/quaternionf` библиотеки GLH_LINEAR. Класс `quat` имеет четыре конструктора:

```
struct quat {
public:
// Создает кватернион и присваивает его компонентам соответствующие
// значения
      quat(nv_scalar x = 0, nv_scalar y = 0, nv_scalar z = 0, nv_scalar
w = 1);
// Создает кватернион на основе другого кватерниона
      quat(const quat& quat);
// Создает кватернион на основе оси (axis) и угла (angle) приворота
      quat(const vec3& axis, nv_scalar angle);
// Создает кватернион на основе матрицы поворота. Обратите внимание, что
// в качестве параметра принимается матрица 3×3.
// Это связано с тем, что в матрице 4×4 за поворот отвечают
// только девять коэффициентов: a00, a01, a02,
// a10, a11, a12, a20, a21 и a22
      quat(const mat3& rot);
...
// Поля, в которых хранятся коэффициенты кватерниона
      union {
// Доступ к компонентам кватерниона по их именам
            struct {
                  nv_scalar x, y, z, w;
            };
```

```
// Доступ к компонентам кватерниона, как к массиву
                nv_scalar comp[4];
        };
};
```

Для демонстрации использования конструкторов на практике я покажу создание кватерниона на основе матрицы модели:

```
mat4 m;
glGetFloatv(GL_MODELVIEW_MATRIX, &m.a00);
quat q(mat3(m));
```

В библиотеке NV_MATH имеется предопределенный объект единичного кватерниона (`quat_id`), который удобно использовать для быстрого задания единичного кватерниона:

```
quat q(quat_id);
```

Класс `quat` имеет множество полезных методов. Некоторые из них приведены в табл. 3.4.

Таблица 3.4. *Методы класса* `quat`

Метод	Описание
`quat Inverse()`	Вычисляет обратный кватернион. Текущий кватернион при этом не изменяется
`void Normalize()`	Нормализует кватернион
`void FromMatrix(const mat3& mat)`	Создает кватернион на основе матрицы 3×3 и присваивает его текущему кватерниону
`void ToMatrix(mat3& mat) const`	Создает матрицу 3×3 на основе текущего кватерниона
`static const quat Identity`	Возвращает единичный кватернион (0, 0, 0, 1)

Класс `quat` перегружает четыре оператора C++: `=`, `×=`, `×` и `[]`. Последний оператор используется для "безопасного" доступа к компонентам кватерниона — в будущем поля, в которых хранятся эти компоненты, могут получить статус `protected` или даже `private` (такие инциденты уже имели место в некоторых классах NVIDIA OpenGL SDK).

В библиотеке NV_MATH имеется ряд функций для работы с кватернионами. Наиболее распространенные функции приведены в табл. 3.5.

Таблица 3.5. *Функции для работы с кватернионами*

Функция	Назначение
`quat & conj(quat & p, const quat & q)`	Вычисляет обратный кватернион для кватерниона q и заносит результат в p
`quat & add_quats(quat& p, const quat& q1, const quat& q2)`	Складывает два кватерниона (p = q1 + q2)
`quat & axis_to_quat (quat & q, const vec3 & a, const nv_scalar phi)`	Вычисляет кватернион (q) на основе оси (a) и угла (phi) поворота
`mat3 & quat_2_mat (mat3 &M, const quat &q )`	Рассчитывает матрицу вращения (M) на основе кватерниона
`quat & mat_2_quat (quat &q,const mat3 &M)`	Вычисляет кватернион (q) на основе матрицы вращения (M)
`quat & trackball(quat& q, vec2& pt1, vec2& pt2, nv_scalar trackballsize)`	Очень полезная функция для эмуляции виртуального трекбола*. Функция рассчитывает кватернион поворота (q) на основе координат начального (p1) и конечного (p2) положения мыши. Компоненты этих координат должны лежать в диапазоне [–1..1]. Последний параметр (`trackballsize`) задает размер шарика трекбола. Чем больше размер шарика, тем более плавно реагирует трекбол на движение мыши. Я не рекомендую присваивать параметру `trackball` маленькие величины (<1), т. к. в противном случае виртуальный трекбол будет работать немного странновато из-за геометрических искажений проекции сферы на плоскость

*Информацию о виртуальных трекболах вы сможете найти в *разд. 2.2.4*. Более подробную информацию можно найти в [2].

Внимание

Из-за ошибки в библиотеке NV_MATH все методы и функции класса `quat` при переводе кватерниона в матрицу в действительности возвращают матрицу для обратного кватерниона. Поэтому перед тем, как переводить кватернион в матрицу, необходимо выполнить метод `Inverse()`.

Почти все эти функции, кроме `trackball`, очень сильно напоминают аналогичные функции библиотеки GLH_LINEAR. Функция `trackball` дает вам возможность самостоятельно эмулировать виртуальный трекбол наподобие интерактора `glut_trackball`. Для демонстрации использования функции `trackball` на практике я переписал пример Ex04 *главы 2* с использованием этой функции (Ex08).

Для этого в программу потребовалось добавить следующие операции:

1. Перевод координат указателя мыши к диапазону [−1..1].
2. Расчет кватерниона поворота с использованием функции `trackball`.
3. Умножение матрицы модели на полученный кватернион.

Ниже приведен код функций, в которые были внесены изменения (листинг 3.7).

Листинг 3.7

```
#define STRICT
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include <gl\glut.h>
#include <string>
#include <nv_math/nv_algebra.h>

using namespace std;

int WinWidth=640;            // Ширина окна
int WinHeight=480;           // Высота окна
// Кватернион поворота
quat q(0, 0, 0, 1);

GLfloat  tx=0;               // Сдвиг по оси X
GLfloat  ty=0;               // Y
GLfloat  tz=-9;                     // Z
GLint   tt=0;                // Активная плоскость: 0 - XY, 1 - XZ
GLboolean bTimer=0;          // Состояние анимации

const string tstr[2]={"Translate XY", "Translate XZ"};

int mx,my;                   // Координаты мыши
bool ldown=false;            // Нажата левая кнопка мыши?
bool rdown=false;            // Нажата правая кнопка мыши?

GLuint list=0;
GLuint OldTick;                     //Старое время
```

```
GLuint FramesCount;          // Счетчик кадров
GLuint StartTick;            // Начало отсчета

void Mouse(int button, int state, int x, int y);
void MouseMotion(int x, int y);

void Display()               // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glPushMatrix();
              glTranslatef(tx,ty,tz);      // Перемещение и поворот
                                           // объекта

// Переводим кватернион в матрицу и умножаем ее
// на текущую матрицу OpenGL
              mat4 m(mat4_id);
// Обратите внимание, что из-за ошибки в библиотеке NV_MATH
// мы предварительно вызываем метод Inverse, который возвращает
// обратный кватернион
              m.set_rot(q.Inverse());
              glMultMatrixf(&m.a00);

              glCallList(list);                  // Вывод объекта на
                                                 // экран
       glPopMatrix();

       glutSwapBuffers();
...
}
...
void MouseMotion(int x, int y)      // Перемещение мыши
{
       if (ldown)            // Левая кнопка
       {
// Приводим координаты прошлого положения мыши к диапазону [-1..1]
              vec2 norm_prev_mpos(-1.0+2.0*float(x)/WinWidth,
                                  1.0-2.0*float(y)/WinHeight);
// Приводим координаты текущего положения мыши к диапазону [-1..1]
```

```
                vec2 norm_mpos(-1.0+2.0*float(mx)/WinWidth,
                                1.0-2.0*float(my)/WinHeight);
                quat q1;
// Рассчитываем кватернион изменения поворота сцены
                trackball(q1, norm_prev_mpos, norm_mpos, 3);
// Умножаем кватернион поворота сцены на полученный кватернион
                q*=q1;

                mx=x;
                my=y;
                glutPostRedisplay(); // Перерисовать экран
        }
        if (rdown)     //Правая
        {
                tx+=0.01*(x-mx);      // Перемещение вдоль активной
                                      // плоскости
                if (tt)
                        tz+=0.01*(y-my);
                else
                        ty+=0.01*(my-y);
                mx=x;
                my=y;
                glutPostRedisplay();
        }
}
...
```

Для того чтобы вы смогли лучше почувствовать разницу между классом `quat` библиотеки NV_MATH и классом `quaternion` библиотеки GLH_LINEAR, я переписал пример Ex06 из *главы 2* с использованием библиотеки NV_MATH. Для этого пришлось внести некоторые косметические изменения, коснувшиеся функции `Display` (Ex09) (листинг 3.8).

Листинг 3.8

```
void Display()                // Обновление содержимого экрана
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glLoadIdentity();
```

```
        mat4 Modelview;                     // Матрица модели
        mat4 rot_mat(mat4_id);              // Матрица вращения
        mat4 translate(mat4_id);            // Матрица переноса
// Кватернион поворота вокруг оси x
        quat quat_x(vec3(1, 0, 0), rx*nv_to_rad);

// Умножаем на кватернион поворота вокруг оси y
        quat_x*=quat(vec3(0, 1, 0), ry*nv_to_rad);
// Преобразуем кватернион в матрицу вращения. Обратите внимание, что мы
// сначала вызываем метод Inverse
        rot_mat.set_rot(quat_x.Inverse());

        translate.set_translation(vec3(tx, ty, tz));      // Перенос
//Создаем матрицу модели
        Modelview=translate*rot_mat;

//Умножаем матрицу модели на текущую матрицу
        glMultMatrixf(&Modelview.a00);

        glCallList(list);                   // Вывод объекта на экран

        glutSwapBuffers();

        if (bTimer)
        {
                FramesCount++;
                char Title[80];
                if (GetTickCount()!=StartTick)
                {
                        sprintf(Title, "FPS=%f",
float(FramesCount)*1000/(GetTickCount()-StartTick));
                        glutSetWindowTitle(Title);
                }
        }
}
```

Программа практически не изменилась, если не считать того, что теперь мы вычисляем обратный кватернион перед тем, как преобразовать его в матрицу.

3.5. Другие полезные функции

В заключение главы мы поговорим о некоторых полезных функциях, которые хотя и используются довольно редко, но, тем не менее, могут заметно облегчить вам жизнь.

3.5.1. Линейная интерполяция

Для облегчения выполнения линейной интерполяции в библиотеке NV_NATH имеется функция `lerp`, которая может интерполировать как скалярные величины, так и векторы.

Оба варианта функции определены схожим образом:

```
// t величина в диапазоне [0..1], a и b - два числа, между которыми
// выполняется линейная интерполяция
nv_scalar lerp(nv_scalar t, nv_scalar a, nv_scalar b)
// t величина в диапазоне [0..1], u и v - два вектора, между которыми
// выполняется линейная интерполяция. Результат заносится в вектор w
vec3 & lerp(vec3 & w, const nv_scalar & t, const vec3 & u, const vec3 & v)
```

Фактически функция отображает значения из диапазона [0..1] на пользовательский диапазон. Рассмотрим простой пример: пусть у нас имеется текстура 512×512 и нам надо узнать, какому текселю текстуры соответствует координата текстуры (0.15, 0.25). Это можно сделать следующим образом:

```
int tx=ceil(lerp(0.15, 0, 511));
int ty=ceil(lerp(0.25, 0, 511));
```

Примечание

Этот пример не совсем корректный, т. к. при использовании фильтрации на каждый тексель текстуры оказывают влияние и соседние тексели.

3.5.2. Геометрические расчеты

В эту группу входят четыре функции:

```
// Вычисляет площадь треугольника. В качестве параметров принимает
// координаты трех вершин треугольника
nv_scalar nv_area(const vec3 & v1, const vec3 & v2, const vec3 &v3);


// Вычисляет периметр треугольника
nv_perimeter(const vec3 & v1, const vec3 & v2, const vec3 &v3);
```

```
// Вычисляет координаты центра вписанной окружности, которые заносятся в
// параметр circle. Функция возвращает радиус вписанной окружности
nv_scalar nv_find_in_circle(vec3& center, const vec3& v1, const vec3& v2,
const vec3& v3);


// Вычисляет координаты центра описанной окружности, которые заносятся в
// параметр circle. Функция возвращает радиус описанной окружности
nv_scalar nv_find_circ_circle( vec3& center, const vec3& v1, const vec3&
v2, const vec3& v3);
```

Для того чтобы продемонстрировать использование этих функций на практике, я написал небольшую программу, которая запрашивает у пользователя координаты углов вершин треугольника, а затем вычисляет характеристики этого треугольника (Ex10) (листинг 3.9).

Листинг 3.9

```
// Если определена переменная test, то программа переходит в тестовый
// режим (использует встроенный набор входных данных)

//#define test

#include <iostream>
#include <string>
#include <nv_math/nv_algebra.h>

using namespace std;

void output(vec3& x)  // Вывод вектора на экран
{
        cout<<"("<<x[0]<<", "<<x[1]<<", "<<x[2]<<")";
}

void input(vec3& x, string id="x") // Ввод вектора
{
        for (unsigned int i=0; i<3; i++)
        {
                cout<<id<<"["<<i<<"]=";
                cin>>x[i];
        }
}
```

```
void main()
{
	vec3 v1, v2, v3;
#ifndef test
	input(v1, "v1");
	input(v2, "v2");
	input(v3, "v3");
#else
	v1=vec3(-1, 0, 0);
	v2=vec3(1, 0, 0);
	v3=vec3(0, 1, 0);
#endif

	cout<<"perimeter="<<nv_perimeter(v1, v2, v3)<<endl;
	cout<<"area="<<nv_area(v1, v2, v3)<<endl;

	vec3 center;
	nv_scalar r=nv_find_in_circle(center, v1, v2, v3);
	cout<<"center of the inscribed circle is "; output(center);
cout<<endl;
	cout<<"radius of the inscribed circle is "<<r<<endl;

	r=nv_find_circ_circle(center, v1, v2, v3);
	cout<<"center of the circumscribed circle is "; output(center);
cout<<endl;
	cout<<"radius of the circumscribed circle is "<<r<<endl;

	getchar();
#ifndef test
	getchar();
#endif
};
```

3.5.3. Математические функции

В эту группу входят две функции приближенного вычисления косинуса угла `ffast_cos` и `fast_cos`. Первая из них вычисляет косинус с максимальной абсолютной погрешностью $1.18*10^{-3}$, а вторая — $2.3*10^{-9}$. Эти функции быстрее стандартной функции cos примерно в 2.14 и 1.47 раза соответственно (так написано в комментариях к файлу nv_algebra.cpp).

Кроме того, в хозяйстве не помешает и функция `nv_random`, возвращающая случайное число из диапазона [−1..1].

Заключение

В этой главе мы рассмотрели математическую библиотеку NV_MATH. И хотя эта библиотека является аналогом библиотеки GLH_LINEAR, она имеет перед ней ряд преимуществ. Например, библиотека NV_MATH умеет работать с матрицами 3×3, выполнять линейную интерполяцию чисел и векторов, вычислять некоторые параметры треугольников и т. д.

Возможно, вас сейчас интересует вопрос — какую библиотеку лучше использовать — GLH_LINEAR или NV_MATH. Однозначного ответа на этот вопрос не существует. Общие рекомендации следующие: если вашей программе не требуются какие-либо дополнительные возможности библиотеки NV_MATH, лучше использовать библиотеку GLH_LINEAR из-за ее хорошей интеграции с библиотекой GLH. В других случаях можно выбрать библиотеку NV_MATH.

Глава 4

Библиотека NV_UTIL

Большинство OpenGL-программ хранят трехмерные объекты и текстуры в файлах на диске. В то же время OpenGL не содержит каких-либо стандартных средств для работы с файлами. И хотя стандартная надстройка над OpenGL GLAUX умеет загружать изображения из файлов формата BMP, этот формат очень плохо подходит для хранения текстур. Ни для кого не секрет, что файлы формата BMP занимают гораздо больший объем по сравнению с другими распространенными графическими форматами.

Поэтому в состав NVIDIA OpenGL SDK была включена библиотека NV_UTIL, которая содержит набор функций для работы с наиболее распространенными форматами файлов: TGA, JPG, ASE и ZIP. Рассмотрим особенности этих форматов.

- Формат TGA является достаточно популярным форматом, подходящим для хранения текстур без потери качества. Для уменьшения размера файлов формат TGA поддерживает RLE-сжатие.
- Формат JPG, как наиболее распространенный, используется для хранения изображений. Файлы формата JPG имеют очень небольшой размер. Но, как известно, бесплатный сыр бывает только в мышеловке. Так, и у формата JPG есть недостаток — он использует алгоритм сжатия с потерями. В большинстве случаев этот недостаток не является существенным, т. к. искажения изображения практически незаметны. Но если в текстуре хранится не просто изображение, а определенная закодированная информация, то формат JPG уже не будет нашим выбором.
- Формат ASE служит для хранения трехмерных моделей. Этот формат поддерживается всеми известными 3D-редакторами: 3D Studio MAX, SoftImage, Maya и т. д. Достоинством формата ASE является то, что этот формат — текстовый. Это обстоятельство сильно облегчает жизнь разработчику. Ведь для того, чтобы увидеть, какие объекты хранятся в ASE-файле, достаточно посмотреть его в любом текстовом редакторе. Это же обстоятельство является и главным недостатком формата — ASE-файлы имеют очень большой объем и довольно медленно загружаются. Но по-

скольку в этой книге мы будем использовать сравнительно простые сцены, этим недостатком можно будет пренебречь.

- Формат ZIP. Любая 3D-программа содержит множество текстур, объектов, скрипов и прочих файлов. Большинство этих файлов никогда не изменяются в процессе работы программы. Кроме того, программа обычно читает эти файлы в режиме последовательного доступа. В то же время файловая система большинства операционных систем оптимизирована для работы с изменяющимися файлами с произвольным доступом. В результате хранение каждого файла на диске сопряжено с дополнительными накладными расходами (таблица размещения файлов, незанятое место в последнем кластере файла и т. д.). При большом количестве файлов эти дополнительные расходы становятся весьма ощутимыми. Например, известная игра Quake3 реально содержит около четырех тысяч файлов. Если бы она хранила их на диске в виде обычных файлов операционной системы, то при размере кластера 4 Кбайт, потери свободного места в остатках кластеров могли бы достичь 8 Мбайт. А это довольно ощутимый объем. Поэтому большинство программ, включая Quake3, хранят все файлы в одном большом файле. Так, Quake3 хранит большую часть файлов в файле pak0.pk3, который на самом деле является обычным ZIP-архивом. Использование ZIP-архивов дает нам еще одно преимущество перед простым хранением файлов на диске — многие файлы имеют свойство очень хорошо сжиматься. В результате — экономия места на диске. Кстати говоря, ASE-файлы сжимаются в несколько раз, поэтому имеет смысл хранить их только в архиве.

Теперь, после того как мы коротко рассмотрели особенности всех файловых форматов, поддерживаемых библиотекой, можно перейти к рассмотрению функций библиотеки NV_UTIL.

Перед тем как начать использовать библиотеку NV_UTIL, не забудьте добавить в список путей **Include** (**Tools** | **Options** | **Projects** | **VC++ Directories**) путь \NVSDK\OpenGL\include\nv_util. После этого необходимо подключить к проекту файлы nv_util\nv_util.h и nv_util.lib. Кроме того, программам, использующим функции для работы с zip-архивами, необходимо иметь доступ к библиотеке nv_unzip.dll, которую можно найти в каталоге \NVSDK\OpenGL\dll.

4.1. Использование файлов формата TGA

Функции, отвечающие за работу с файлами формата TGA, размещены в пространстве имен tga. Следовательно, для доступа к ним необходимо использовать либо директиву `using namespace tga`, либо указывать перед названием функций и типов этой библиотеки префикс `tga::`.

Загрузка файла в память осуществляется функцией `tga::read`, которой передается имя файла:

```
tgaImage * read(const char *filename)
```

Функция возвращает указатель на структуру `tga::tgaImage`:

```
    typedef struct
    {
// Ширина изображения
        GLsizei  width;
// Высота изображения
        GLsizei  height;
// Количество компонентов цвета в изображении
        GLint    components;
// Формат изображения
        GLenum   format;

// Эти поля используются для хранения информации о палитре
// Так как 256-цветные форматы практически полностью вытеснены
// HighColor и TrueColor-форматами, мы не будем их
// рассматривать в этой книге
        GLsizei  cmapEntries;
        GLenum   cmapFormat;
        GLubyte *cmap;

// Указатель на массив пикселов изображения
        GLubyte *pixels;
    }
    tgaImage;
```

Если при загрузке файла произошла ошибка, то функция возвратит 0. Еще раз хочу обратить ваше внимание на то, что функция `read` возвращает указатель на структуру `tgaImage`, поэтому не забывайте удалять его, когда информация станет ненужной. В противном случае вы столкнетесь с большой утечкой памяти (особенно при работе с текстурами большого размера). Хотя в примерах NVSDK это нигде не показано, перед удалением указателя на структуру `tgaImage` надо удалить из памяти еще и указатель на массив пикселов (поле `pixels`), иначе он будет находиться в памяти до окончания работы программы.

Для демонстрации чтения текстур из файлов формата TGA я написал небольшую программу, которая выводит на экран текстурированную прямоугольную плоскость (рис. 4.1).

Рис. 4.1. Прямоугольная плоскость с текстурой, загруженной из файла формата TGA

Исходный код программы приведен в листинге 4.1 (Ex01).

Листинг 4.1

```
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include "glh_glut_ext.h"
#include <nv_util/nv_util.h>
```

```
using namespace std;
using namespace glh;

int WinWidth=640;            // Ширина окна
int WinHeight=480;           // Высота окна

// Текстурный объект
tex_object_2D eburg;
// Указатель на структуру tgaImage
tga::tgaImage* pEburgImage=0;
display_list list0;

glut_console console;
glut_simple_user_interface user(&console);
glut_callbacks cb;
glut_swapbuffers swapbuffers;

const string Title="GLHE Demo";

void Init()                          // Инициализация OpenGL
{
       glEnable(GL_DEPTH_TEST);

       glClearColor(0.5, 0.5, 0.75, 1);

// Создание дисплейного списка объекта (плоскости)
       list0.new_list(GL_COMPILE);
                     glBegin(GL_QUADS);
                     glTexCoord2f(0, 0);
                     glVertex3f(-1, -1, 0);

                     glTexCoord2f(0, 1);
                     glVertex3f(-1,  1, 0);

                     glTexCoord2f(1, 1);
                     glVertex3f( 1,  1, 0);

                     glTexCoord2f(1, 0);
                     glVertex3f( 1, -1, 0);
```

```
            glEnd();
        list0.end_list();
// Загрузка текстуры из файла
        pEburgImage=tga::read("..\\eburg.tga");
// Если произошла ошибка
        if (pEburgImage==0)
        {
// Выводим сообщение об ошибке
            console.add("Can't open file eburg.tga");
// Завершаем работу программы
            console.exit(-1);
// Сразу запускаем цикл обработки сообщений GLUT
            glutMainLoop();
        }

// Настраиваем параметры текстурного объекта
        eburg.bind();
        eburg.parameter(GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        eburg.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Загружаем изображение текстуры в видеопамять
        gluBuild2DMipmaps(eburg.target, pEburgImage->components,
pEburgImage->width, pEburgImage->height,
                                    pEburgImage->format,
GL_UNSIGNED_BYTE, pEburgImage->pixels);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
}

void Display()              // Обновление содержимого экрана
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        eburg.enable();
        list0.call_list();                  //Вывод объекта на экран
        eburg.disable();
}

void onExit()
{
```

```
// Выгружаем изображение из памяти. Эту команду оптимальнее всего
// вставить сразу после gluBuild2DMipmaps (функция Init)
        if (pEburgImage)
        {
                if (pEburgImage->pixels)
                        delete[] pEburgImage->pixels;
                delete pEburgImage;
        }
}

int main(int argc, char* argv[])
{
        console.add("Initializing GLUT...");
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
        glutInitWindowSize(WinWidth,WinHeight);
        glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                                        (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
        glutCreateWindow(Title.c_str());

        console.add("Initializing OpenGL Helper Library...");
        glut_helpers_initialize();

        Init();

        console.title=Title;

        user.user_mouse.configure_buttons(1);
        user.user_mouse.dolly.dolly[2]=-2;

        cb.display_function=Display;

// Если не был вызван метод exit интерактора glut_console, то добавляем
// интеракторы в список. В противном случае, скорее всего,
// произошла какая-то ошибка и интерактор glut_console готовится к
// завершению работы
        if (!console.exiting)
```

```
        {
                glut_add_interactor(&user);
                glut_add_interactor(&cb);
                glut_add_interactor(&console);
                glut_add_interactor(&swapbuffers);
        }

        console.processCmd("exec autoexec.cs");

        atexit(onExit);

        console.add("Initialize complete. Starting glutMainLoop");
        glutMainLoop();

        return 0;
}
```

В этом примере демонстрируется использование метода `exit` интерактора `glut_console`, очищающего список интеракторов, оставляя в нем только интерактор `glut_console` с последующим выводом на экран консоли. В результате работа программы останавливается, и пользователь может, не торопясь, просмотреть сообщения в консоли.

Метод `exit` устанавливает поле `exiting` в значение `true`. Анализируя поле `exiting`, пользовательское приложение может узнать, был ли вызван метод `exit`. Это обстоятельство может использоваться, например, для определения необходимости добавлять пользовательские интеракторы в список интеракторов. Ведь если поле `exiting` равно `true`, то интерактор `glut_console` уже очистил список и готовиться к завершению работы. Следовательно, если в этот момент времени мы начнем модифицировать список интеракторов, то можем нарушить логику работы программы.

В заключении этого раздела хочу обратить ваше внимание на подводный камень, поджидающий начинающих программистов. При написании функций обработки изображения никогда не надо заранее предполагать, что функция `read` возвратит изображение в формате GL_RGB или GL_RGBA. Функция `read` может возвращать изображения и в "нестандартных" форматах. Например, GL_BGR_EXT или GL_BGRA_EXT. Поэтому, лучше всего, перед началом обработки изображения проанализировать поле `format` структуры `tgaImage`.

4.2. Использование файлов формата JPG

Для работы с файлами формата JPG используется функция `read`, объявленная в пространстве имен jpeg:

```
int read(const char * filename, int * width, int * height, unsigned char
** pixels, int * components);
```

В отличие от одноименной функции пространства имен TGA, функция `read` возвращает параметры загруженного изображения в виде множества параметров. Ниже приведено краткое описание параметров этой функции (первый параметр является входным, остальные — выходные):

- `filename` — указатель на имя файла в формате ASCIIZ;
- `width` — ширина загруженного изображения;
- `height` — высота загруженного изображения;
- `pixels` — указатель на массив пикселов загруженного изображения;
- `components` — число компонентов в изображении.

В случае успешной загрузки изображения функция `read` возвращает 0, в противном случае — ненулевой код ошибки.

У функции `read` есть одна нехорошая особенность, которую я склонен считать ошибкой — она возвращает перевернутое изображение. Точнее, изображение, отраженное относительно оси *y* (рис. 4.2).

Для корректного вывода изображения необходимо проводить его постобработку. Самым простым способом является модификация матрицы текстуры при помощи команды `glScalef(1, -1, 1)`. Использование этого способа демонстрируется в листинге 4.2 (Ex02).

Листинг 4.2

```
void Init()                          // Инициализация OpenGL
{
       glEnable(GL_DEPTH_TEST);

       glClearColor(0.5, 0.5, 0.75, 1);

// Создание дисплейного списка объекта (плоскости)
       list0.new_list(GL_COMPILE);
// Предварительно применяем к матрице текстуры команду glScale
              glMatrixMode(GL_TEXTURE);
```

```
            glLoadIdentity();
            glScalef(1, -1, 1);

            glBegin(GL_QUADS);
                  glTexCoord2f(0, 0);
                  glVertex3f(-1, -1, 0);

                  glTexCoord2f(0, 1);
                  glVertex3f(-1,  1, 0);

                  glTexCoord2f(1, 1);
                  glVertex3f( 1,  1, 0);

                  glTexCoord2f(1, 0);
                  glVertex3f( 1, -1, 0);
            glEnd();

            glPopMatrix();
            glMatrixMode(GL_MODELVIEW);
      list0.end_list();

      pEburgImage=new tga::tgaImage;
// Загружаем текстуру
      if (jpeg::read("..\\eburg.jpg", &pEburgImage->width,
&pEburgImage->height, &pEburgImage->pixels,
                              &pEburgImage->components))
      {
            console.add("Can't open file 'eburg.jpg'");
            console.exit(-1);
            glutMainLoop();
      }

      if (pEburgImage->components=3)
            pEburgImage->format=GL_RGB;
      else
      {
            console.add("Unsupported file format: 'eburg.jpg'");
            console.exit(-1);
            glutMainLoop();
      }
```

```
        eburg.bind();
        eburg.parameter(GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        eburg.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        gluBuild2DMipmaps(eburg.target, pEburgImage->components,
pEburgImage->width, pEburgImage->height,
                                        pEburgImage->format,
GL_UNSIGNED_BYTE, pEburgImage->pixels);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
}
```

Рис. 4.2. Текстура, загруженная из JPG-файла без предварительной обработки

Обратите внимание, что программа сохраняет информацию о загруженной текстуре в структуре `tgaImage`. Это сделано для облегчения модификации программы: чтобы заставить ее читать текстуры формата TGA, потребуется изменить всего несколько строк.

Но коррекцию текстуры путем модификации матрицы текстуры нельзя назвать оптимальным методом, т. к. у него есть два существенных недостатка.

- Многие видеокарты значительно снижают свою производительность, если матрица текстуры не является единичной.
- Этот прием усложняет программу, использующую текстуры разных форматов, т. к. теперь программа должна отслеживать формат текущей текстуры и корректировать соответствующим образом матрицу текстуры.

Поэтому самым оптимальным вариантом будет "ручной" переворот текстуры после загрузки. Для этого я написал функцию `MirrorTexture` (листинг 4.3), которая принимает указатель на структуру `tgaImage` и отражает изображение, хранящееся в поле `pixels`, относительно оси *y*.

Листинг 4.3

```
int MirrorTexture(tga::tgaImage* pTextureImage)
{
// Получаем указатель на начало массива пикселов изображения
       GLubyte* Image1=pTextureImage->pixels;
// Перебираем все пикселы первой половины изображения
       for (int j=0; j<pTextureImage->height/2; j++)
       {
// Вычисляем крайний левый пиксел строки второй половины изображения,
// симметричный относительно оси x с аналогичным пикселом первой половины
              GLubyte* Image2=pTextureImage->
pixels+(pTextureImage->height-j-1)*pTextureImage->width*
                     pTextureImage->components;
              for (int i=0; i<pTextureImage->
width*pTextureImage->components; i++)
              {
// Меняем пикселы двух строк
                     GLubyte tmp=*Image1;
                     *Image1=*Image2;
                     *Image2=tmp;
                     Image1++;
                     Image2++;
              }
       }

       return true;
}
```

Для демонстрации использования функции `MirrorTexture` я переписал предыдущий пример (Ex03). Для этого потребовалось лишь немного подправить код, отвечающий за загрузку текстуры из файла:

```
pEburgImage=new tga::tgaImage;


if (jpeg::read("..\\eburg.jpg", &pEburgImage->width,
&pEburgImage->height,
 &pEburgImage->pixels, &pEburgImage->components))
{
       console.add("Can't open file 'eburg.jpg'");
       console.exit(-1);
       glutMainLoop();
}

if (pEburgImage->components=3)
       pEburgImage->format=GL_RGB;
else
{
       console.add("Unsupported file format: 'eburg.jpg'");
       console.exit(-1);
       glutMainLoop();
}

MirrorTexture(pEburgImage);


eburg.bind();
eburg.parameter(GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
eburg.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
gluBuild2DMipmaps(eburg.target, pEburgImage->components,
pEburgImage->width,
 pEburgImage->height, pEburgImage->format, GL_UNSIGNED_BYTE,
 pEburgImage->pixels);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

Исходный код функции `MirrorTexture` вы можете найти в файле nv_util_ext.h, находящемся в каталоге проекта. В дальнейшем мы будем помещать в этот файл различные функции, расширяющие возможности библиотеки NV_UTIL. В результате чего файл nv_util_ext.h со временем превратится в своеобразную библиотеку расширений NV_UTIL. Для краткости мы будем называть эту библиотеку NV_UTIL_EXT.

Теперь мы умеем загружать текстуры из файлов трех форматов: BMP, TGA и JPG. Но, тем не менее, у нас есть проблема — для работы с каждым форматом мы используем функции, имеющие различный синтаксис, что очень неудобно. Например, для того, чтобы заставить предыдущий пример читать текстуру из файла формата TGA, придется вносить изменения в исходный код программы. А это может привести к ошибкам в программе и т. д.

Поэтому мы создадим универсальную функцию `read`, которая умеет читать текстуры форматов BMP, TGA и JPG. Исходный код этой функции приведен в листинге 4.4.

Листинг 4.4

```
// Заменяет все буквы строки на заглавные
string Upper(const string str)
{
	string result="";
	for (string::const_iterator itor=str.begin(); itor!=str.end();
itor++)
		result+=toupper(*itor);

	return result;
}

tga::tgaImage* read(string filename)
{
// Определяем расширение файла
	int pos=filename.find_last_of('.');
	if ((pos==-1) || (pos==filename.size()-1) || (pos==0))
		return false;

	string file_ext=filename.substr(pos+1, filename.size()-pos-1);
// Если расширение файла tga, то используем функцию tga::read
	if (Upper(file_ext)=="TGA")
		return tga::read(filename.c_str());
	else
// Если расширение файла jpg, то используем функции
// jpg::read + MirrorTexture
		if (Upper(file_ext)=="JPG")
		{
			tga::tgaImage* pImage=new tga::tgaImage;
```

```
                        if (jpeg::read(filename.c_str(), &pImage->width,
&pImage->height, &pImage->pixels,
                                &pImage->components))
                        {
                                delete pImage;
                                return 0;
                        };

                        if (pImage->components==3)
                                pImage->format=GL_RGB;
                        else
                                if (pImage->components==4)
                                        pImage->format=GL_RGBA;

                        MirrorTexture(pImage);

                        return pImage;
                }
                else
// Если расширение файла bmp, то используем функцию auxDIBImageLoad
                        if (Upper(file_ext)=="BMP")
                        {
                                AUX_RGBImageRec* pImage;
                                pImage=auxDIBImageLoad(filename.c_str());
                                if (!pImage)
                                        return 0;

                                tga::tgaImage* pImage1=new tga::tgaImage;
                                pImage1->width=pImage->sizeX;
                                pImage1->height=pImage->sizeY;
                                pImage1->components=3;
                                pImage1->format=GL_RGB;
                                pImage1->pixels=pImage->data;
                                delete pImage;
                                return pImage1;
                        }

        return false;
}
```

Как видно, синтаксис функции `read` аналогичен синтаксису функции `tga::read` — она принимает имя файла, а возвращает указатель на структуру `tgaImage`.

Но раз уж мы написали универсальную функцию чтения файлов, то имеет смысл создать универсальную функцию загрузки изображения из структуры `tgaImage` в видеопамять:

```
bool texImage2D(tga::tgaImage* image, tex_object* tex_obj, bool
bMipmap=true)
{
        assert(image);
        GLenum target=tex_obj->target;

        if (bMipmap)
                gluBuild2DMipmaps(target, image->components, image->width,
image->height, image->format,
                  GL_UNSIGNED_BYTE, image->pixels);
        else
                glTexImage2D(target, 0, image->components, image->width,
image->height, 0, image->format,
                  GL_UNSIGNED_BYTE, image->pixels);

        return true;
}
```

Как видно, функция принимает три параметра:

- `image` — указатель на структуру `tgaImage`;
- `tex_obj` — указатель на объект, являющийся потомком `tex_object`, который инкапсулирует текстурный объект с загружаемой текстурой;
- `bMipmap` — указывает, надо ли генерировать Mipmap-уровни. По умолчанию равен `true`: Mipmap-уровни создаются.

Для испытания этих двух функций я переписал предыдущий пример (Ex04). Новый вариант программы поддерживает консольную команду `load_texture`, при помощи которой пользователь может загрузить другую текстуру из файла. Исходный текст наиболее важных фрагментов примера приведен в листинге 4.5.

Листинг 4.5

```
...
void Init()                          // Инициализация OpenGL
```

```
{
        glEnable(GL_DEPTH_TEST);

        glClearColor(0.5, 0.5, 0.75, 1);

// Создание дисплейного списка объекта (чайника)
        list0.new_list(GL_COMPILE);
                glBegin(GL_QUADS);
                        glTexCoord2f(0, 0);
                        glVertex3f(-1, -1, 0);

                        glTexCoord2f(0, 1);
                        glVertex3f(-1,  1, 0);

                        glTexCoord2f(1, 1);
                        glVertex3f( 1,  1, 0);

                        glTexCoord2f(1, 0);
                        glVertex3f( 1, -1, 0);
                glEnd();
        list0.end_list();

        eburg.bind();
        eburg.parameter(GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        eburg.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
}

...

bool onConsole(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
// Если команда "load_texture"
        if (Cmd=="load_texture")
        {
                if (Params.size()==0)
                        return false;
                if (Params[0]=="/?")
```

```
                {
                        console->add("loads texture from file");
                        console->add("");
                        console->add("load_texture <filename>");
                        console->add("where <filename> is file with
texture");
                        return true;
                }
// Выгружаем старое изображение из памяти
                if (pEburgImage)
                {
                        if (pEburgImage->pixels)
                        {
                                delete[] pEburgImage->pixels;
                                pEburgImage->pixels=0;
                        }
                        delete pEburgImage;
                        pEburgImage=0;
                }
// Загружаем текстуру из файла
                pEburgImage=read(Params[0]);
// Если загрузка не удалась, выводим сообщение об ошибке
                if (!pEburgImage)
                {
                        console->add(string("Can't open file
'")+Params[0]+string("'"));
                        return false;
                };

                eburg.bind();
// Загружаем текстуру в видеопамять
                texImage2D(pEburgImage, &eburg);

                console->add(string("Texture \"")+Params[0]+string("\" has
been loaded"));
                return true;
        }
        return false;
}
...
```

```
int main(int argc, char* argv[])
{
	console.add("Initializing GLUT...");
	glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
	glutInitWindowSize(WinWidth,WinHeight);
	glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                                        (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
	glutCreateWindow(Title.c_str());

	console.add("Initializing OpenGL Helper Library...");
	glut_helpers_initialize();

	Init();

	console.title=Title;
// Регистрируем консольную команду load_texture
	console.addCmd("load_texture", "loads texture from file", 0,
onConsole);

	user.user_mouse.configure_buttons(1);
	user.user_mouse.dolly.dolly[2]=-2;

	cb.display_function=Display;

	glut_add_interactor(&user);
	glut_add_interactor(&cb);
	glut_add_interactor(&console);
	glut_add_interactor(&swapbuffers);

	console.processCmd("exec autoexec.cs");

	atexit(onExit);

	console.add("Initialize complete. Starting glutMainLoop");
	glutMainLoop();

	return 0;
}
```

4.3. Использование ZIP-архивов в качестве хранилища файлов

Вообще-то, этот материал было бы логичнее разместить где-нибудь в конце главы, после обзора формата ASE. Но поскольку одна из функций чтения файлов формата ASE умеет загружать эти файлы из ZIP-архива, я решил сначала рассмотреть формат ZIP, а уже потом — ASE.

Для извлечения файлов из ZIP-архива используется функция `open`, объявленная в пространстве имен unzip:

```
unsigned char * open(const char * filename, const char * inzipfile,
unsigned int * size)
```

- `filename` — имя ZIP-архива;
- `inzipfile` — имя файла, который надо извлечь из архива;
- `size` — указатель на переменную, в которую заносится размер файла, считанного из архива.

Функция возвращает указатель в область памяти, где находится распакованный файл. Если файл извлечь не удалось, то функция возвратит 0.

Перед тем как начать использовать эту функцию в своей программе, к проекту необходимо подключить библиотеку unzip.lib. Кроме того, программа должна иметь доступ к файлу unzip.dll, которой находится в каталоге \NVSDK\OpenGL\dll.

Для демонстрации применения функции `open` на практике я написал небольшую консольную программу (Ex05), которая извлекает из архива текстовый файл и выводит его на экран (листинг 4.6).

Листинг 4.6

```
#define TEST

#include <iostream>
#include <string>
#include <nv_util/nv_util.h>

using namespace std;

// Имя ZIP-файла, из которого будут извлекаться файлы
string zip_filename;
// Имя файла, который надо извлечь из ZIP-архива
string inzip_filename;
```

```
// Указатель на буфер, в который будет занесено содержимое распакованного
// файла
unsigned char* buffer;
// Размер буфера
unsigned int buffer_size;

void main()
{
#ifdef TEST
// Тестовый набор данных
       zip_filename="..\\data.zip";
       inzip_filename="config.sys";
#else
       cout<<"zip name:";cin>>zip_filename;
       cout<<"inzip name";cin>>inzip_filename;
#endif
// Извлекаем файл из архива
       buffer=unzip::open(zip_filename.c_str(), inzip_filename.c_str(),
&buffer_size);
       if (!buffer)
       {
// Если файл не удалось извлечь, выводим сообщение об ошибке
              cout<<"Can't open file:
"<<zip_filename<<"::"<<inzip_filename;
              getchar();
#ifndef TEST
              getchar();
#endif
              return;
       }

// Выводим содержимое буфера на экран
       cout.write((char*) buffer, buffer_size);

// Удаляем буфер из памяти
       delete[] buffer;

       getchar();
#ifndef TEST
```

```
        getchar();
#endif
};
```

По идее, программа сначала извлекает файл из архива и заносит его содержимое в буфер, а затем выводит этот буфер на экран.

Но в реальной жизни все обстоит намного сложнее. Дело в том, что большинство функций для работы с файлами различных форматов принимают в качестве параметра имя файла, а не указатель на буфер. Для выхода из сложившейся ситуации есть два пути.

1. Трудоемкий. Переписать функции таким образом, чтобы они принимали в качестве параметра указатель на буфер, в котором находится содержимое файла.
2. Легкий. Записать содержимое буфера во временный файл и передать его название функции.

Для демонстрации второго метода я добавил в пример Ex04 возможность чтения текстур из ZIP-архива (Ex06). Для этого пришлось написать новую функцию `read`, которая извлекает файл из ZIP-архива и передает его в функцию `read` примера Ex04. Но здесь есть один подводный камень — если писать извлекаемые файлы в текущий каталог, то программу нельзя будет запускать с CD-диска. Следовательно, возникает вопрос: куда извлекать временные файлы из ZIP-архива? Варианты вроде: "в каталог C:\TEMP" — не подходят, т. к. на некоторых системах диск C: может быть защищен от записи и т. д.

Самым оптимальным вариантом будет использование функции `GetTempPath`, для получения пути к каталогу временных файлов Windows и `GetTempFileName` для получения уникального имени файла в этом каталоге. Исходный текст функции `read` приведен в листинге 4.7.

Листинг 4.7

```
tga::tgaImage* read(string zip_filename, string inzip_filename)
{
// Получаем расширение файла
        int pos=inzip_filename.find_last_of('.');
        if ((pos==-1) || (pos==inzip_filename.size()-1) || (pos==0))
                return false;
```

```
        string file_ext=inzip_filename.substr(pos+1,
inzip_filename.size()-pos-1);

// Указатель на буфер, в котором хранится извлеченный из архива файл
        unsigned char* buffer;
// Размер буфера
        unsigned int buffer_size;
// Извлекаем файл из архива
        buffer=unzip::open(zip_filename.c_str(), inzip_filename.c_str(),
&buffer_size);
// Если ошибка, возвращаем 0
        if (!buffer)
                return 0;
// Буфер для хранения каталога временных файлов
        char szTmpPath[MAX_PATH+1];
// Буфер для хранения полного имени временного файла
        char szTmpFile[MAX_PATH+1];

// Получаем каталог временных файлов Windows
        GetTempPath(MAX_PATH, szTmpPath);
// Получаем полный путь к временному файлу с уникальным именем
        GetTempFileName(szTmpPath, "img", 0, szTmpFile);
// Удаляем расширение из имени файла (временный файл
// всегда имеет расширение tmp) и заменяем его расширением
// извлекаемого файла. Это связанно с тем,
// что для функции read(string filename) важно расширение файла
        string tmp_filename=szTmpFile;
        tmp_filename=tmp_filename.substr(0, tmp_filename.size()-
3)+file_ext;
// Создаем файл и копируем в него содержимое буфера
        ofstream ofile(tmp_filename.c_str(), ios::binary);
        ofile.write((char*) buffer, buffer_size);
        ofile.close();
// Удаляем буфер
        delete[] buffer;
// Вызываем и передаем функции read имя временного файла
        tga::tgaImage* image=read(tmp_filename);

// Удаляем временные файлы (функция GetTempFileName автоматически создает
// временный файл нулевого размера с расширением tmp)
```

```
        DeleteFile(tmp_filename.c_str());
        DeleteFile(szTmpFile);

        return image;
}
```

Теперь остается внести небольшие изменения в обработчик команд консоли:

```
bool onConsole(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
        if (Cmd=="load_texture")
        {
                if (Params.size()==0)
                        return false;
                if (Params[0]=="/?")
                {
                        console->add("loads texture from file");
                        console->add("");
                        console->add("load_texture <filename>");
                        console->add("where <filename> is file with
texture");
                        return true;
                }

                if (pEburgImage)
                {
                        if (pEburgImage->pixels)
                        {
                                delete[] pEburgImage->pixels;
                                pEburgImage->pixels=0;
                        }
                        delete pEburgImage;
                        pEburgImage=0;
                }

                if (Params.size()==1)
                        pEburgImage=read(Params[0]);
                else
```

```
                pEburgImage=read(Params[0], Params[1]);
        if (!pEburgImage)
        {
                if (Params.size()==1)
                        console->add(string("Can't open file
\"")+Params[0]+string("\""));
                else
                        console->add(string("Can't open file
\"")+Params[0]+string("::")+Params[1]+string("\""));
                return false;
        };

        eburg.bind();
        texImage2D(pEburgImage, &eburg);

        if (Params.size()==1)
                console->add(string("Texture
\"")+Params[0]+string("\" has been loaded"));
        else
                console->add(string("Texture
\"")+Params[0]+"::"+Params[1]+string("\" has been loaded"));
        return true;
    }
    return false;
}
```

Теперь консольная команда `load_texture` имеет два варианта:

- `load_texture <имя файла с текстурой>`
- `load_texture <имя zip-архива> <имя файла с текстурой внутри архива>`

В заключение раздела еще раз напомню вам о том, что прием с промежуточной распаковкой файлов из архива на жесткий диск подходит только для небольших приложений. В профессиональных программах, содержащих сотни мегабайт текстур и моделей, оптимальнее всего непосредственно читать данные напрямую из буфера, создаваемого функцией `open`.

4.4. Чтение моделей из файлов формата ASE

До сих пор мы использовали в примерах только стандартные объекты библиотек GLU и GLUT: кубики, шары, чайники и т. д. Но в реальных про-

граммах используются значительно более сложные полигональные объекты: самолеты, машины, деревья, люди...

Конечно, сложные модели можно рисовать "вручную" при помощи команд OpenGL. Но на практике такой подход используется разве что для простейших моделей. Тем более, что в больших проектах применяется разделение труда — программисты пишут программы, а художники делают модели с применением специальных пакетов 3D-моделирования наподобие 3D Studio MAX, SoftImage и Maya. И только после этого полученные модели переводятся во внутренний формат программы.

Следовательно, перед тем, как мы начнем использовать в своих программах сложные полигональные модели, мы должны определиться с тремя пунктами.

1. Пакет, который мы будем использовать для создания 3D-моделей.
2. Внутренний формат хранения модели.
3. Библиотека для чтения модели из внутреннего формата.

Для создания 3D-моделей мы будем использовать 3D Studio MAX 5. На это есть множество причин. Во-первых, этот пакет довольно прост в изучении, во-вторых, на российском рынке присутствует множество книг по 3D Studio MAX (например, [14], [15], [16], [17]), в-третьих, этот пакет один из самых распространенных в России, поэтому его довольно легко приобрести.

В качестве внутреннего формата для хранения моделей мы будем использовать текстовый формат ASE (ASCII Scene Export). У этого формата есть ряд преимуществ по сравнению с остальными форматами:

- этот формат поддерживают все распространенные, системы 3D-моделирования;
- ASE является текстовым форматом, что позволяет просматривать и изменять файлы этого формата в любом текстовом редакторе. Это обстоятельство очень сильно облегчает отладку приложений;
- формат ASE является открытым и бесплатным, что позволяет использовать его в приложениях без лицензионных отчислений и т. д.;
- формат ASE позволяет описывать сцены неограниченной степени сложности, поэтому наши возможности по экспорту сцен будут скорее ограничены возможностями современных ускорителей, чем возможностями этого формата.

Для работы с файлами формата ASE мы будем использовать уже знакомую библиотеку NV_UTIL. Правда, здесь мы можем столкнуться с некоторыми трудностями. Дело в том, что функции библиотеки NV_UTIL для работы с файлами формата ASE являются низкоуровневыми функциями, в результате этого чтение и вывод простейшей 3D-модели занимают более сотни строк. Выходом из ситуации является создание высокоуровневой надстройки над

библиотекой NV_UTIL. Но создание собственной надстройки с нуля — задача очень не простая.

К счастью, в каталоге \NVSDK\OpenGL\src\demos\height_fog уже находится пример объектно-ориентированной надстройки над библиотекой NV_UTIL (файлы ase.h и ase.cpp). Правда, возможности этой надстройки довольно ограничены. Например, она не умеет самостоятельно загружать текстуры для текстурированных моделей. Кроме того, в этой надстройке имеется ряд мелких ошибок, следствием которых является утечка памяти.

Поэтому я решил разработать собственную объектно-ориентированную надстройку над NV_UTIL под названием "ASE Reader Library" взяв за основу объектную надстройку NVIDIA.

Моя объектно-ориентированная надстройка, как и все функции библиотеки NV_UTILS, отвечающие за работу с файлами формата ASE, имеет довольно сложную структуру. Поэтому сначала мы рассмотрим пример простой программы, считывающей из ZIP-архива нетекстурированную модель стула, взятого из NVIDIA OpenGL SDK. Эту модель вы сможете найти в файле \NVSDK\Common\media\models\ase\room.zip\chair.ase (рис. 4.3). И только после этого мы перейдем к рассмотрению библиотек NV_UTIL и ASE Reader Library.

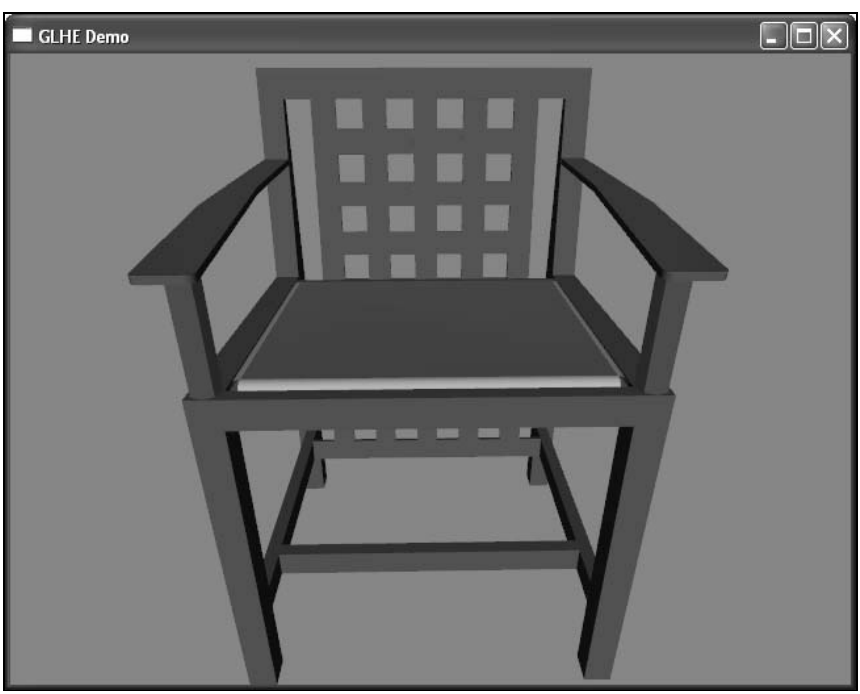


Рис. 4.3. Модель стула, взятая из NVIDIA OpenGL SDK

Для начала надо подключить к проекту саму объектную надстройку над NV_UTIL — ASE Reader Library. Для этого в каталог с программой необходимо скопировать файлы ase.h и ase.cpp (они уже имеются в каталоге с примером), после чего ase.h включается в главный модуль программы директивой `#include "ase.h"`, а ase.cpp просто подключается к проекту через меню `Project > Add existing item`.

В библиотеке ASE Reader за работу с моделями формата ASE отвечает класс `CAseModel`. Загрузка модели осуществляется при помощи метода `LoadModel`:

```
bool CAseModel::LoadModel(char *filename, char *modelname)
```

Как видно, этот метод принимает два параметра — название файла модели и имя ZIP-архива, в котором хранится модель. Если загрузка прошла успешно, то метод возвращает значение `true`, в противном случае — `false`. Загруженная модель сохраняется внутри объекта класса `CAseModel`.

Для вывода модели используется функция `DrawModel`:

```
void CAseModel::DrawModel(bool textured)
```

Если параметр `textured` равен `true`, то на модель накладываются текстуры, в противном случае наложение текстур выключено. Хотя метод `DrawModel` использует массивы вершин при выводе объектов, для вывода моделей оптимальнее всего использовать дисплейные списки. У метода `DrawModel` есть еще одна особенность — он задает цвет объекта при помощи команды `glColor`, которой передается диффузная составляющая материала. Поэтому для корректной закраски объекта необходимо включить режим цветных материалов командой `glEnable(GL_COLOR_MATERIAL)`.

Размеры модели, отрисованной командой, в общем случае могут быть любыми. Поэтому возможны ситуации, когда модель может быть едва различимой на экране или, напротив, быть такой огромной, что на экране поместится лишь маленькая ее часть. В то же время в реальных программах часто приходится масштабировать модель таким образом, чтобы ее координаты лежали в диапазоне [-1..1]. В этом случае очень может пригодиться команда `GetExtents`, возвращающая минимальное и максимальное значения каждой координаты вершин объекта.

```
void CAseModel::GetExtents(vec3f &minExtent, vec3f &maxExtent)
```

Команда возвращает два вектора: `minExtent`, в котором находятся минимальные значения соответствующих координат объекта, и `maxExtent`, в котором находятся максимальные значения координат объекта. Чтобы получить коэффициент масштабирования для приведения координат модели к диапазону [−1..1] можно, например, выполнить следующие операции:

```
model0.GetExtents(min, max);


float scale=max(fabs(min[0]), fabs(min[1]));
scale=max(scale, fabs(min[2]));
```

```
scale=max(scale, fabs(max[0]));
scale=max(scale, fabs(max[1]));
scale=max(scale, fabs(max[2]));
scale=1.0/scale;

glScalef(scale, scale, scale);
```

После такого преобразования ни одна из координат вершин объекта не выйдет из диапазона [−1..1].

Теперь мы знаем все для написания примера, загружающего модель из файла формата ASE с последующим выводом ее на экран. Исходный код примера с комментариями приведен в листинге 4.8.

Листинг 4.8

```
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include "glh_glut_ext.h"
#include <nv_util.h>
#include "ase.h"

using namespace std;
using namespace glh;

int WinWidth=640;            // Ширина окна
int WinHeight=480;           // Высота окна

void BuildList();
// Дисплейный список для хранения модели
lazy_build_display_list list0(BuildList);

glut_console console;
glut_simple_user_interface user(&console);
glut_callbacks cb;
glut_swapbuffers swapbuffers;

const string Title="GLHE Demo";
// Объект модели
CAseModel model0;
```

```
void Init()                              // Инициализация OpenGL
{
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_COLOR_MATERIAL);
        glEnable(GL_CULL_FACE);
        glCullFace(GL_BACK);

        glClearColor(0.5, 0.5, 0.75, 1);
}

void Display()                // Обновление содержимого экрана
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        list0.call_list();                  // Вывод объекта на экран
}

bool CmdProc(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
// Обработчик команды загрузки объекта load_model
        if (Cmd=="load_model")
        {
// Если команда была вызвана без параметров, то это ошибка
                if (Params.size()==0)
                {
                        console->add("Not enough parameters");
                        return false;
                }
// Если параметр = "/?", то выводим информацию о команде
                if (Params[0]=="/?")
                {
                        console->add("Loads ase-model from zip-arhiv");
                        console->add("");
                        console->add("load_model <zip-arhiv> <inzip
file>");
                        console->add("where");
```

```
                        console->add("  <zip-arhiv> is name of zip
archive");
                        console->add("  <inzip file> is model's filename
within zip arhiv");
                        return true;
                }
// Если число параметров = 2, то первый параметр — имя ZIP-архива,
// а второй — модель внутри архива
                if (Params.size()==2)
                {
                        char buf0[MAX_PATH+1];
                        char buf1[MAX_PATH+1];
                        strcpy(buf0, Params[0].c_str());
                        strcpy(buf1, Params[1].c_str());
                        if (!model0.LoadModel(buf0, buf1))
                        {
// Если загрузка не удалась - выводим сообщение об ошибке
                                console->add(string("Can't load model
\"")+Params[0]+"::"+Params[1]+"\"");
                                return false;
                        }
                        list0.rebuild();
                        return true;
                }
                else
                {
                        console->add("Invalid parameters");
                        return false;
                }
        }

        return false;
}

// Функция построения дисплейного списка, вызываемая классом
// lazy_build_display_list
void BuildList()
{
        glPushAttrib(GL_ALL_ATTRIB_BITS);
```

```
// Включаем режим нормализации нормали
        glEnable(GL_NORMALIZE);
        vec3f min, max;
// Получаем размеры объекта
        model0.GetExtents(min, max);
// Масштабируем объект таким образом, чтобы координаты всех его вершин
// находились в диапазоне [-3..3]
        float scale=max(fabs(min[0]), fabs(min[1]));
        scale=max(scale, fabs(min[2]));
        scale=max(scale, fabs(max[0]));
        scale=max(scale, fabs(max[1]));
        scale=max(scale, fabs(max[2]));
        scale=3.0/scale;
        glScalef(scale, scale, scale);
// Рисуем модель без текстур
        model0.DrawModel(false);
        glPopAttrib();
}


int main(int argc, char* argv[])
{
        console.add("Initializing GLUT...");
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
        glutInitWindowSize(WinWidth,WinHeight);
        glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                               (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
        glutCreateWindow(Title.c_str());
        console.add("Initializing OpenGL...");

        Init();

        console.add("Initializing OpenGL Helper Library...");
        glut_helpers_initialize();

        console.title=Title;
```

```
// Регистрируем консольную команду load_model
        console.addCmd("load_model", "loads ase-model from zip-arhiv", 0,
CmdProc);

        user.reshaper.zFar=1000;

        cb.display_function=Display;

        glut_add_interactor(&user);
        glut_add_interactor(&cb);
        glut_add_interactor(&console);
        glut_add_interactor(&swapbuffers);

        console.processCmd("exec autoexec.cs");

        console.add("Initialize complete. Starting glutMainLoop");
        glutMainLoop();

        return 0;
}
```

Как видно из исходного кода, загрузка модели осуществляется при помощи консольной команды `load_model`

```
load_model <zip-архив> <модель>
```

Обработчик команды считывает модель из файла и вызывает метод `rebuild` класса `lazy_build_display_list` для перестройки дисплейного списка. Собственно код генерации дисплейного списка расположен в функции `BuildList`. Функция `BuildList` включает режим нормализации нормалей, рассчитывает коэффициент масштабирования для приведения координат модели к диапазону [−3..3] и, наконец, рисует модель при помощи метода `DrawModel`. Обработчику события `Display` остается только вывести полученный дисплейный список на экран.

При загрузке пример автоматически выполняет консольные команды из файла autoexec.cs, который содержит единственную команду:

```
load_model ..\models.zip chair.ase
```

Кстати, в файле models.zip находится еще и модель стола (файл table.ase), также взятая из NVIDIA OpenGL SDK (рис. 4.4). Для ее загрузки в консоли достаточно ввести команду:

```
load_model ..\models.zip table.ase
```

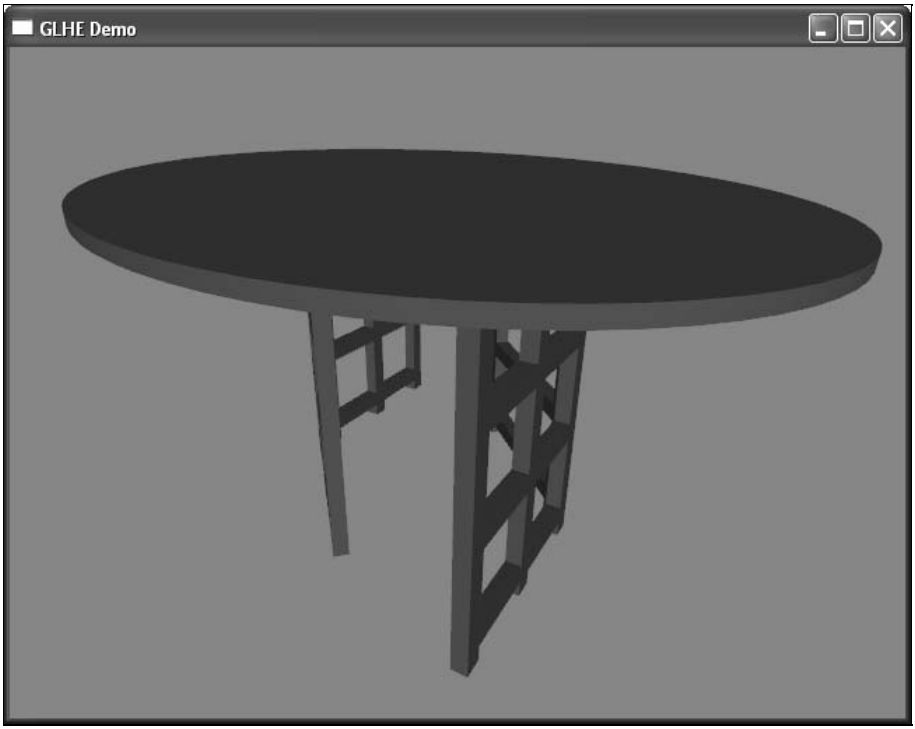


Рис. 4.4. Модель стола, взятая из NVIDIA OpenGL SDK

4.4.1. Экспорт моделей из 3D Studio MAX 5 в формат ASE

Сейчас вы уже умеете загружать модели из ASE-файлов. Но в реальной жизни вам редко будут встречаться готовые модели в формате ASE. Как правило, модель надо будет сначала нарисовать или подредактировать в 3D Studio MAX, и только потом экспортировать ее в формат ASE.

В этом разделе мы рассмотрим процесс экспорта в формат ASE модели космического корабля, которую можно найти на втором CD-диске 3D Studio MAX 5.

Запустите 3D Studio MAX 5 и откройте файл \Tutorials\Intro_to_Materials\polyfighterPaintstart.max[1]. На рис. 4.5 приведено окно 3D Studio MAX 5 с изображением этой модели.

[1] Эту модель также можно найти на CD-диске, поставляемом с книгой, в каталоге \Examples\Ch4-NV_UTIL\3DSMAX_MODELS\Polyfighter.

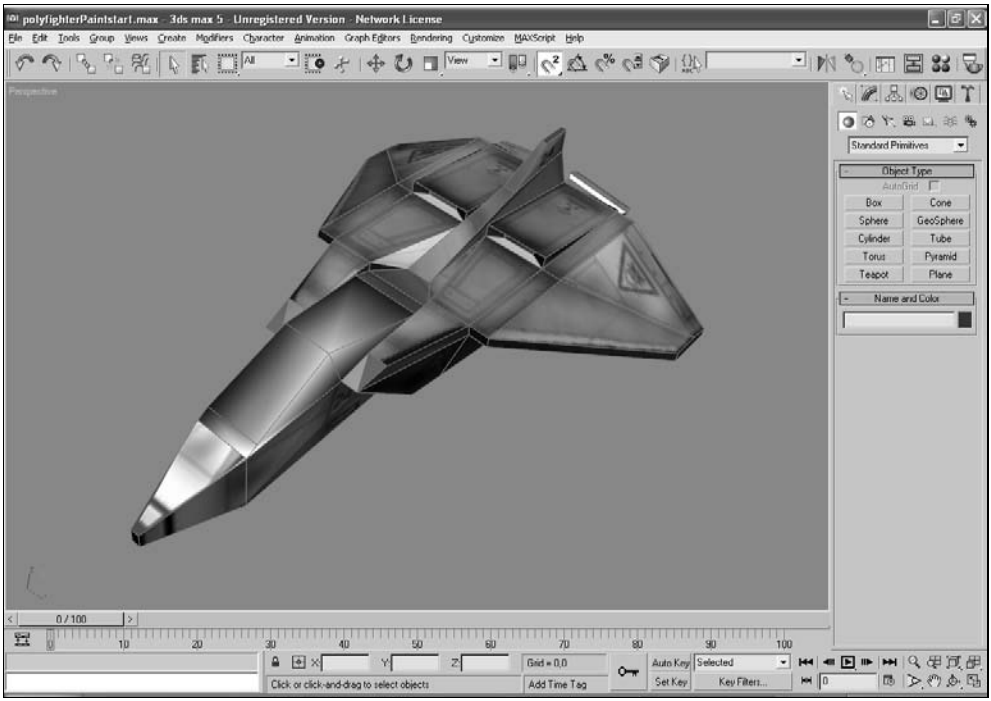


Рис. 4.5. Окно 3D Studio MAX 5 с загруженной моделью

Для начала откроем редактор материалов (клавиша <M>) и посмотрим, какой материал использует модель космолета (рис. 4.6). Следует учитывать, что текущая версия библиотеки ASE Reader использует только диффузную составляющую цвета материала (список **Anisotropic Basic Parameters**, поле **Diffuse**) и диффузную текстуру (список **Maps**, поле **Diffuse Color**), поэтому остальные параметры материала игнорируются.

Диффузный цвет объекта, как правило, должен быть белым или серым. Дело в том, что библиотека ASE Reader накладывает диффузную текстуру в режиме `GL_MODULATE`, т. е. путем умножения соответствующих компонентов цвета объекта на цвет текстуры. Следовательно, задав диффузный цвет объекта (0, 1, 0), мы придадим объекту зеленоватый оттенок.

Чтобы объект имел естественный цвет, мы должны изменить диффузный цвет материала, который у данной модели имеет красноватый оттенок (137, 50, 50), на белый (255, 255, 255). Для этого необходимо щелкнуть на цвете диффузного объекта и изменить его в открывшемся окне на (255, 255, 255).

Так как других материалов в этой сцене нет, мы можем приступать к собственно экспорту сцены их 3D Studio MAX. Для это выберите команду меню **File | Export** и укажите в открывшемся окне тип файла ASE. Теперь введите название файла (например, polyfighter.ASE) и нажмите кнопку **Сохранить**.

После этого откроется окно **ASCII Export**. Установите параметры экспорта так, как это показано на рис. 4.7, и нажмите кнопку **OK**. Остается только запаковать полученный файл формата ASE в ZIP-архив.

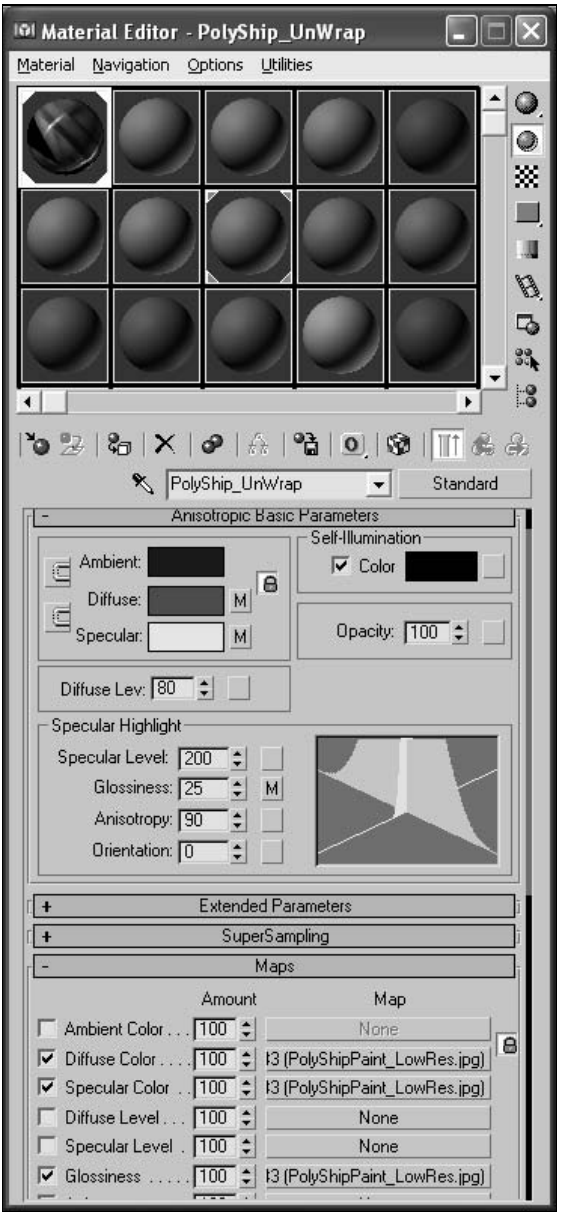


Рис. 4.6. Редактор материалов

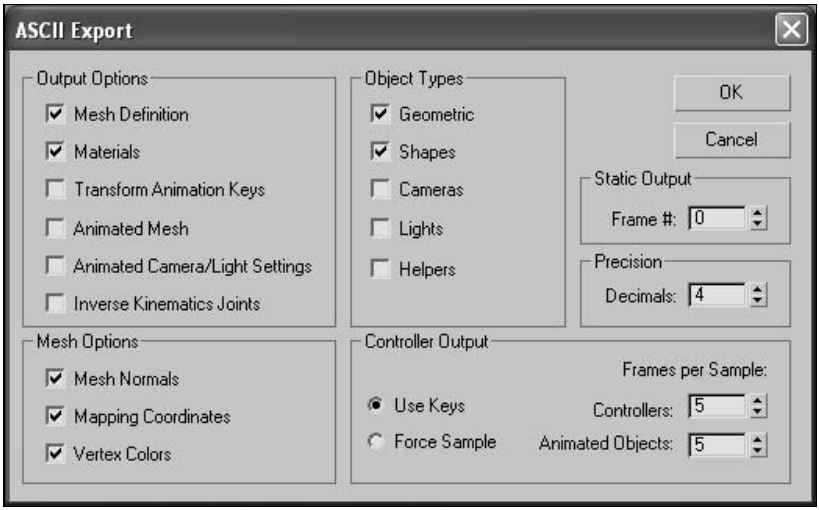


Рис. 4.7. Параметры экспорта сцены в формат ASE

Но это еще не все. Материал, наложенный на модель космолета, использует диффузную текстуру, хранящуюся в файле PolyShipPaint_LowRes.jpg, поэтому текстуру тоже надо запаковать в архив вместе с моделью самолета. Но, вот незадача, текстура самолета имеет разрешение 260×256, в то время как OpenGL поддерживает только текстуры, ширина и высота которых являются кратными степени 2. Следовательно, нам надо изменить разрешение текстуры при помощи какого-нибудь графического редактора: например, ADCSee или Photoshop.

В частности, если вы используете ADCSee, то для установки разрешения текстуры в 256×256 необходимо открыть текстуру и выполнить команду **Resize** меню **Tools**, затем установить в открывшемся окне такие же параметры, как на рис. 4.8, и нажать кнопку **OK**. Теперь остается запаковать полученную текстуру и ASE-файл в ZIP-архив, и модель готова.

Итак, мы получили готовую текстурированную модель формата ASE для использования библиотекой ASE Reader. Осталось только внести изменения в пример Ex07 для того, чтобы заставить его выводить текстурированные модели (Ex08). Для этого надо заменить в функции `BuildList` команду `model0.DrawModel(false)` на `model0.DrawModel(true)`.

Запустите полученную программу и введите в консоли следующую команду (предполагается, что модель самолета находится в файле models.zip):

```
load_model ..\models.zip polyfighter.ase
```

На экране должна появиться модель космического самолета (рис. 4.9).

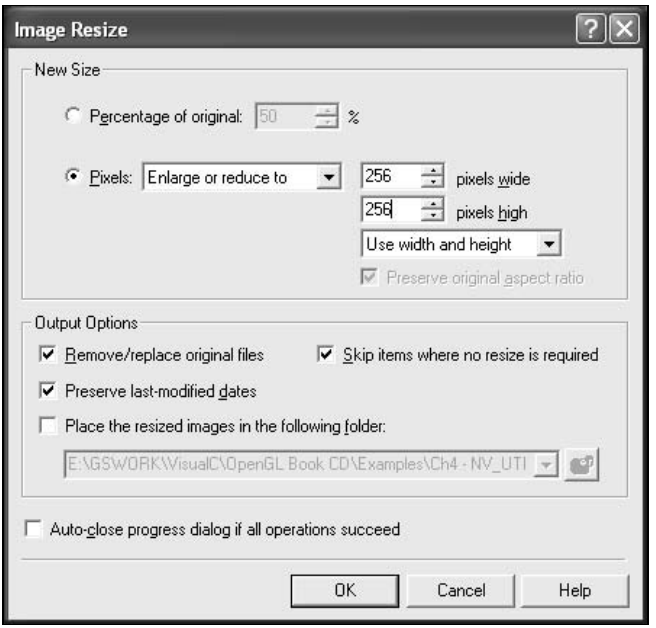


Рис. 4.8. Изменение разрешения текущего файла до 256×256

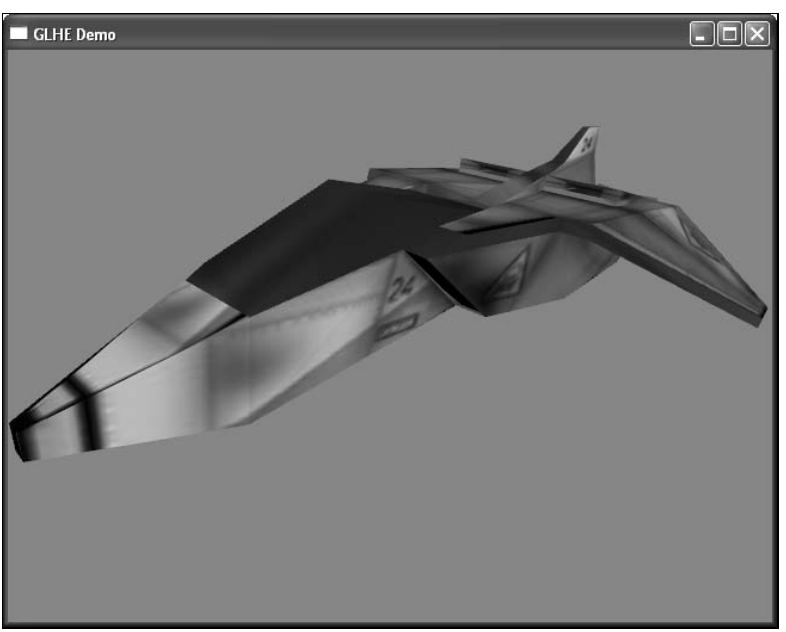


Рис. 4.9. Модель космического самолета, загруженная с использованием библиотеки ASE Reader

Если модель выводится без текстур, значит, вы наткнулись на одну "нехорошую" особенность 3D Studio MAX 5 — при экспорте моделей она сохраняет абсолютные пути к текстурам.

А библиотека NV_UTIL может иногда обрезать длинные строки путей к файлам, содержащие пробелы, что приводит к тому, что библиотека ASE Reader не может определить название файла текстуры. В этом случае необходимо открыть ASE-файл в любом текстовом редакторе и удалить пути к файлам, оставив только имена файлов без путей.

4.4.2. Создание демонстрационной программы "полет самолета"

В этом разделе мы рассмотрим процесс создания полноценной демонстрационной программы, выводящей на экран космический самолет, летающий на автопилоте над земной поверхностью (рис. 4.10).

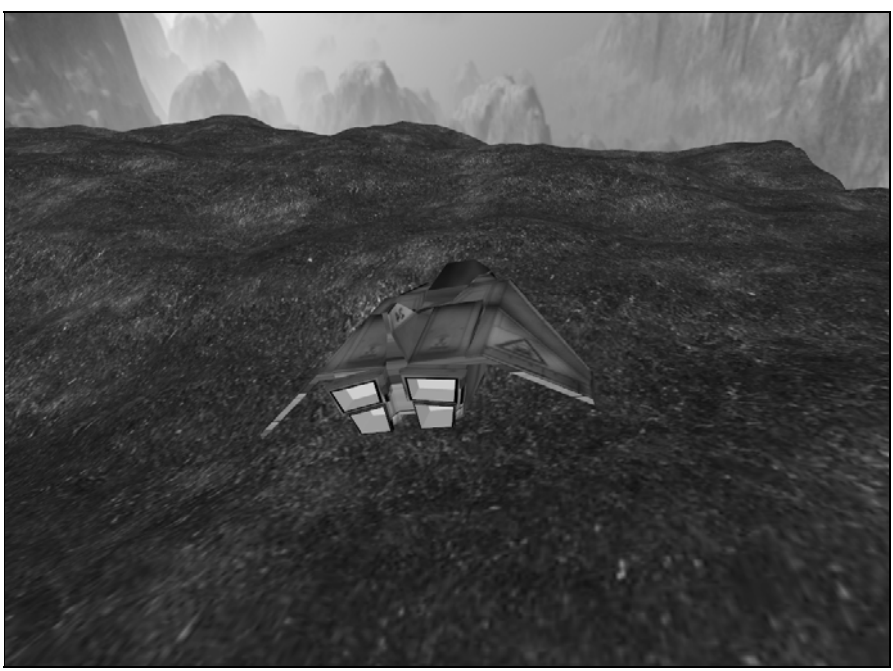


Рис. 4.10. Демонстрационная программа "полет самолета"

Чтобы написать эту программу, нам понадобятся модели самолета и земной поверхности. Модель самолета у нас уже есть (мы ее создали в предыдущем разделе). Значит, нам остается создать только модель земной поверхности.

Для начала запустите 3D Studio MAX и создайте прямоугольный параллелепипед длиной и шириной 100 единиц, а высотой — примерно 7.5 единиц. При этом желательно, чтобы центр полученного параллелепипеда совпадал с центром глобальной системы координат 3D Studio MAX. Количество сегментов по длине и ширине должно быть 75, а по высоте 1 (рис. 4.11).

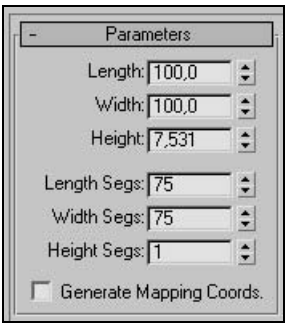


Рис. 4.11. Параметры прямоугольного параллелепипеда

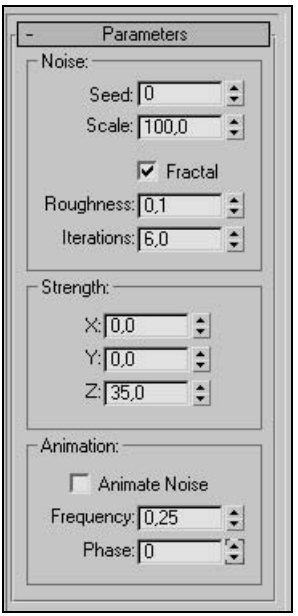


Рис. 4.12. Параметры модификатора **Noise**

Теперь необходимо применить к полученному прямоугольному параллелепипеду модификатор **Noise** (**Modify | Modifier List | Noise**). В параметрах

группы **Noise** установите флажок **Fractal**, а параметру **Roughness** присвойте значение 0.1. Параметру **Z** группы **Strength** надо присвоить значение 35 (рис. 4.12).

В результате мы получим довольно симпатичную холмистую местность (рис. 4.13).

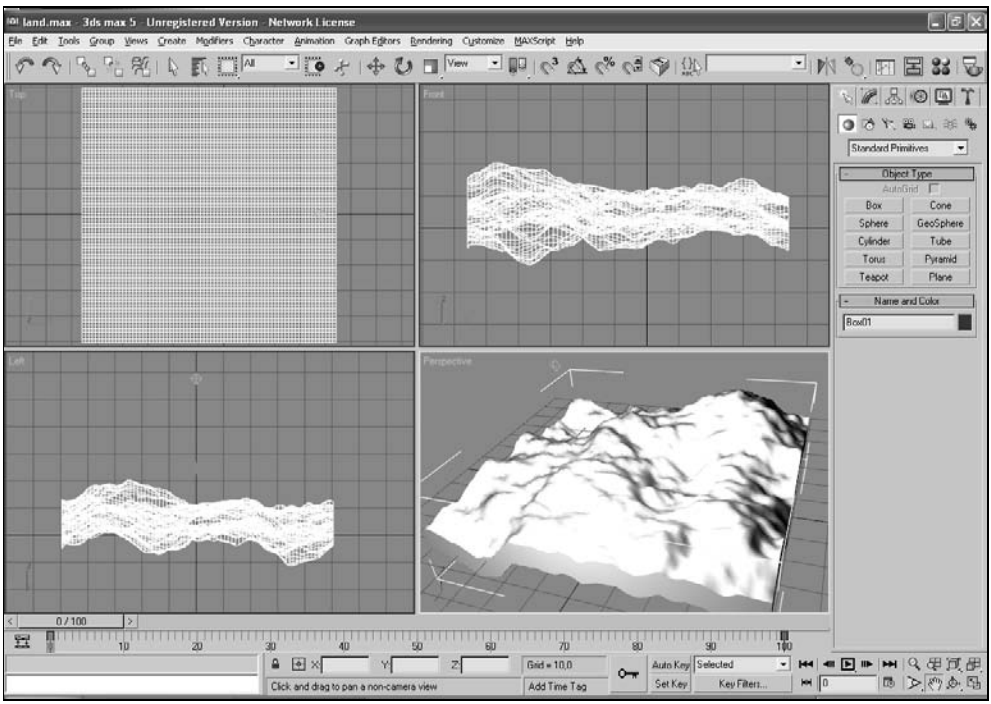


Рис. 4.13. Модель земной поверхности, созданная в 3D Studio MAX

Теперь надо создать материал для нашей местности. Для этого запустите редактор материалов Material Editor (**Renderinng | Material Editor**) и создайте новый стандартный материал **Standard**. В качестве диффузного цвета необходимо выбрать белый цвет, а в качестве диффузной текстуры — текстуру травы из файла \3dsmax5\maps\Ground\GRASS2.JPG, у которой надо предварительно изменить разрешение на 512×512. Теперь примените полученный материал к объекту и посмотрите на результат. Для удобства включите в свойствах текстуры земли режим **Show Map in Viewport** для того, чтобы увидеть результат без предварительной визуализации.

Текстура травы будет действительно наложена на поверхность, правда, немного не так, как надо. Для того чтобы исправить этот недостаток, надо

сгенерировать "правильные" текстурные координаты для поверхности земли. Для этого к модели поверхности земли надо применить модификатор **UVW Mapping**. В группе **Mapping** этого модификатора надо выбрать переключатель **Planar** и установить поля **U Title** и **V Title** в значение 3, чтобы текстура земли повторялась вдоль осей X и Y три раза (рис. 4.14).

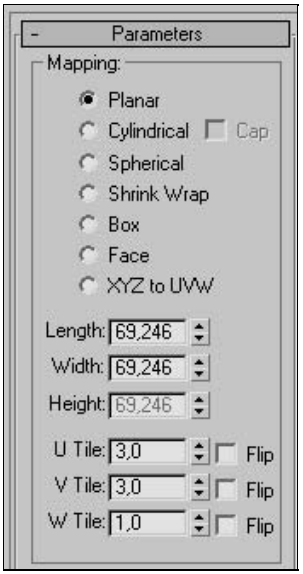


Рис. 4.14. Параметры модификатора **UVW Mapping**

Этот эффект позволит повысить кажущееся разрешение текстуры, в результате чего она не будет казаться слишком размытой. Также не забудьте проверить положение переключателя в группе **Alignment** — он должен быть установлен в положение **Z**.

Итоговый вид поверхности, созданной в 3D Studio MAX, показан на рис. 4.15.

Остается только экспортировать полученную поверхность в ASE-файл, который необходимо запаковать в ZIP-архив вместе с текстурой травы. Готовая сцена земной поверхности находится на CD, прилагаемом к книге, в каталоге \Examples\Ch4\NV_UTIL\3DSMAX_MODELS\Land.

Теперь мы можем приступать к написанию кода демонстрационной программы "полет самолета". Для начала необходимо написать код загрузки моделей из файлов формата ASE. Этот код с подробными комментариями приведен в листинге 4.9.

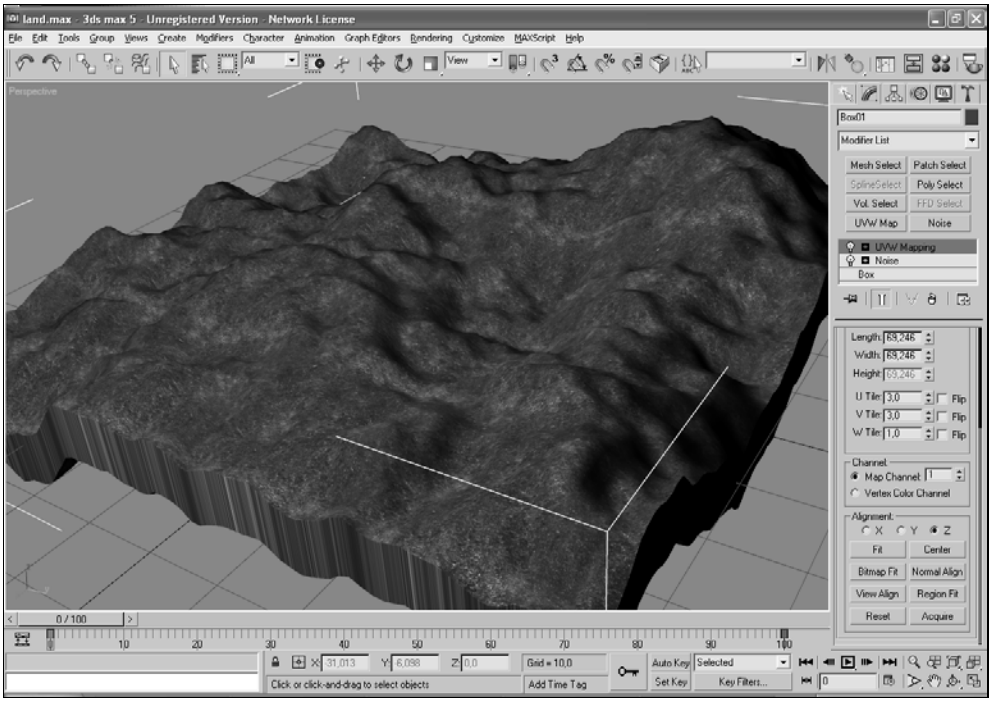


Рис. 4.15. Итоговый вид поверхности земли

Листинг 4.9

```
// Размер модели земли
const float Size=50.0;

// Дисплейный список земной поверхности
display_list land;
// Дисплейный список самолета
display_list airplane;

// Модель земной поверхности
CAseModel land_model;
// Модель самолета
CAseModel airplane_model;
```

```
// Масштабирует модель таким образом, чтобы координаты всех ее вершин
// находились в диапазоне [-1, -1, -1]…[1, 1, 1]
void scale_model(CAseModel& model)
{
        vec3f min, max;
        model.GetExtents(min, max);
        float scale=max(fabs(min[0]), fabs(min[1]));
        scale=max(scale, fabs(min[2]));
        scale=max(scale, fabs(max[0]));
        scale=max(scale, fabs(max[1]));
        scale=max(scale, fabs(max[2]));
        scale=1.0/scale;
        glScalef(scale, scale, scale);
}

// Инициализация OpenGL
void Init()
{
// Настраиваем OpenGL
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_COLOR_MATERIAL);
        glEnable(GL_CULL_FACE);
        glEnable(GL_NORMALIZE);
        glCullFace(GL_BACK);

        vec4 diffuse_color(0.7, 0.7, 0.7, 1);
        vec4 ambinet_color=vec4_one-diffuse_color+vec4(0, 0, 0, 1);

        glLightfv(GL_LIGHT0, GL_DIFFUSE, &diffuse_color[0]);
        glLightfv(GL_LIGHT0, GL_AMBIENT, &ambinet_color[0]);

        glClearColor(0.5, 0.5, 0.75, 1);

// Загрузка модели местности
// Загружаем модель из файла
        land_model.LoadModel("..\\models.zip", "land.ase");
```

```
// Создаем новый дисплейный список
        land.new_list(GL_COMPILE);

        glPushAttrib(GL_ALL_ATTRIB_BITS);
        glPushMatrix();
// Масштабируем модель, приводя ее координаты к диапазону
// [-1, -1, -1]...[1, 1, 1]
        scale_model(land_model);
// Масштабируем модель, приводя ее координаты к диапазону
// [-Size, -Size, -20]...[Size, Size, 20]
        glScalef(Size, Size, 20.0);
// Рисуем модель
        land_model.DrawModel(true);
        glPopMatrix();
        glPopAttrib();
// Завершаем дисплейный список
        land.end_list();
// Удаляем информацию о модели из памяти, кроме текстурных объектов
        land_model.Clear(false);

// Загрузка модели самолета
        airplane_model.LoadModel("..\\models.zip", "polyfighter.ase");
        airplane.new_list(GL_COMPILE);

        glPushAttrib(GL_ALL_ATTRIB_BITS);
        glPushMatrix();
        scale_model(airplane_model);
        glScalef(1.0, 1.0, 1.0);
        airplane_model.DrawModel(true);
        glPopMatrix();
        glPopAttrib();

        airplane.end_list();

        airplane_model.Clear(false);
...
}
```

Теперь нам надо определиться с тем, по какой траектории должен летать самолет. Перебрав несколько вариантов, я решил остановиться на функции вида:

```
X=0.6 × sin(2t) × Size;
Y=0.6 × cos(3t) × Size;
Z=9
```

где:

- `X`, `Y`, `Z` — координаты самолета в момент времени `t`.

График этой функции, построенный в MathCAD 2001, изображен на рис. 4.16.

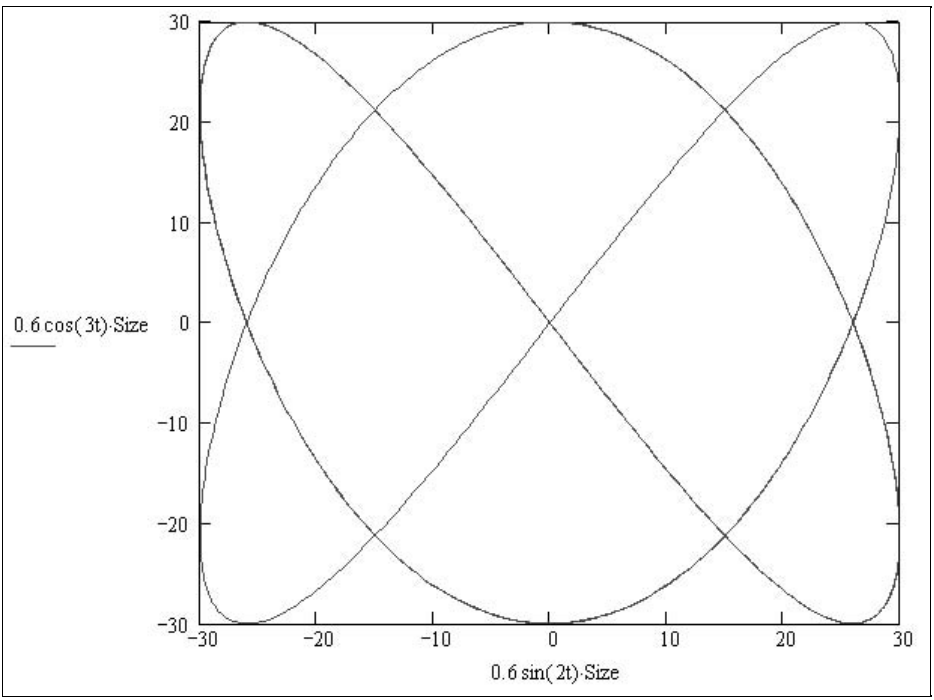


Рис. 4.16. Траектория полета самолета

Для того чтобы заставить наш самолет лететь носом вперед, нам надо автоматически его поворачивать таким образом, чтобы направление его носа совпадало с касательной к графику функции.

Вектор касательной к функции траектории равен:

$$\left(\frac{\partial X}{\partial t}, \frac{\partial Y}{\partial t}, \frac{\partial Z}{\partial t}\right) = (1.2 \cdot \sin(2 \cdot t) * Size, 1.8 \cdot \cos(3 \cdot t) * Size, 0)$$

Так как в OpenGL повороты задаются только при помощи связки "ось поворота + угол поворота", нам придется обратиться за помощью к кватернионам библиотеки GLH. В библиотеке GLH поворот может быть задан при помощи начального и конечного положений вектора. Поэтому предположив, что начальное направление самолета совпадает, к примеру, с вектором (0, −1, 0), мы сможем описать такой поворот, как кватернион, который совмещает вектор (0, −1, 0) с вектором $(1.2 \cdot \sin(2 \cdot t) \cdot Size, 1.8 \cdot \cos(3 \cdot t) \cdot Size, 0)$.

Но тут есть один подвох. Дело в том, что когда направление самолета будет антипараллельно вектору (0, −1, 0) (т. е. направление самолета будет совпадать с вектором (0, 1, 0)) , то класс `quaternionf` не сможет корректно рассчитать такой поворот. Это связано с тем, что класс `quaternionf` находит наикратчайший поворот, совмещающий два вектора. Если же векторы антипараллельны, то число таких поворотов будет бесконечно. В результате класс `quaternionf` выберет один поворот из бесконечного числа возможных поворотов, причем, как показывает практика, не тот, который нам нужен. Случай с антипараллельными векторами нужно рассматривать отдельно от остальных. К счастью, здесь все просто — искомый поворот может быть задан, как поворот на 180 градусов вокруг оси (0, 0, 1).

Еще один "подводный камень" связан с тем, что вектор $(1.2 \cdot \sin(2 \cdot t) \cdot Size, 1.8 \cdot \cos(3 \cdot t) \cdot Size, 0)$ может при некоторых значениях t вырождаться в нулевой вектор (0, 0, 0). В этом случае класс `quaternionf` не сможет найти правильный поворот самолета. Выходом из этой ситуации является распознавание нулевого вектора с последующим расчетом вектора направления полета самолета при чуть большем значении t.

С поворотом самолета мы вроде бы разобрались. Но самолеты имеют еще одну "вредную привычку" — крениться на бок во время поворота. При этом, чем резче поворот, тем сильнее крен. В нашей программе мы будем считать крен пропорциональным углу между вектором

$$(1.2 \cdot \sin(2 \cdot t) \cdot Size, 1.8 \cdot \cos(3 \cdot t) \cdot Size, 0)$$

и вектором

$$(1.2 \cdot \sin(2 \cdot (t + 0.001)) \cdot Size, 1.8 \cdot \cos(3 \cdot (t + 0.001)) \cdot Size, 0),$$

показывающим, на какой угол повернется самолет за 0.001 секунды.

Угол между векторами может быть найден при помощи скалярного или векторного произведения. Но поскольку нам важен знак угла (+ или −), то скалярное произведение, вычисляющее косинус угла, нам не подходит. Поэтому мы будем рассчитывать угол между двумя векторами *p1* и *p2* по формуле:

$$angle = \arcsin\left(\frac{p1.x \cdot p2.y - p1.y \cdot p2.x}{|p1| \cdot |p2|}\right).$$

Самолет в нашей демонстрационной программе летит не один — за ним должна лететь виртуальная камера, снимающая его. В нашей программе траектория камеры задается следующей формулой:

```
X=0.61 × sin(2 × (t-0.03)) × Size;
Y=0.61 × cos(3 × (t-0.03)) × Size;
Z=10
```

где:

- `X`, `Y`, `Z` — координаты камеры в момент времени `t`.

Зная координаты камеры, а так же то, что она нацелена на самолет, мы сможем легко смоделировать виртуальную камеру при помощи команды `gluLookAt`.

Теперь мы знаем всю необходимую информацию для написания кода, моделирующего полет самолета над землей. Этот код, а также код вывода сцены на экран приведены в листинге 4.10 (Ex09).

Листинг 4.10

```
// Момент времени старта программы
int start_tick;
// Текущее положение самолета
vec3 airplane_pos;
// Текущее положение камеры
vec3 camera_pos;
// Кватернион поворота самолета
quaternionf airplane_quat;
// Угол крена самолета
float airplane_lurch;

void Idle()
{
// Рассчитываем время, прошедшее с момента старта программы
       int time=GetTickCount()-start_tick;
// Рассчитываем время полета самолета
       float airplane_time=float(time)/30000.0;
// Время полета камеры
       float camera_time=airplane_time-0.03;

// Рассчитываем текущие координаты самолета
       airplane_pos.x=0.6*sin(2*airplane_time)*Size;
```

```
        airplane_pos.y=0.6*cos(3*airplane_time)*Size;
        airplane_pos.z=9;

// Расчет кватерниона поворота самолета (используется для того,
// чтобы самолет всегда летел носом вперед

// Рассчитываем частные производные от функции, задающей полет
// самолета. При этом dxdt и dydt не должны быть одновременно равны 0
        float dxdt;
        float dydt;
        do
        {
                dxdt=1.2*cos(2*airplane_time)*Size;
                dydt=-1.8*sin(3*airplane_time)*Size;
                airplane_time+=0.001;
        }
        while (dxdt*dxdt+dydt*dydt<GLH_EPSILON);

// Вектор, задающий начальное направление носа самолета
        vec3f p1(0, -1, 0);
// Вектор, задающий конечное направление носа самолета
        vec3f p2(dxdt, dydt, 0);
// Нормализируем вектор p2
        p2.normalize();
// Если скалярное произведение равно -1
// (т.е. вектора p1 и p2 антипараллельны), то явно
// рассчитываем кватернион поворота самолета.
// Иначе кватернион поворота рассчитывается классом quaternionf
        if (equivalent(p1.dot(p2), -GLH_ONE))
                airplane_quat.set_value(vec3f(0, 0, 1), GLH_PI);
        else
                airplane_quat.set_value(p1, p2);

// Рассчитываем направление носа самолета через интервал времени 0.001
        float dxdt1;
        float dydt1;
        do
        {
                dxdt1=1.2*cos(2*(airplane_time+0.001))*Size;
```

```
                dydt1=-1.8*sin(3*(airplane_time+0.001))*Size;
                airplane_time+=0.001;
        }
        while (dxdt1*dxdt1+dydt1*dydt1<GLH_EPSILON);
// Находим синус угла между векторами (dxdt, dydt) и (dxdt1, dydt1).
        float sina=(dxdt*dydt1-
dxdt1*dydt)/sqrt((dxdt*dxdt+dydt*dydt)*(dxdt1*dxdt1+dydt1*dydt1));
// Находим угол крена самолета
        airplane_lurch=asinf(sina)*3000.0;

// Находим текущие координаты камеры
        camera_pos.x=0.61*sin(2*camera_time)*Size;
        camera_pos.y=0.61*cos(3*camera_time)*Size;
        camera_pos.z=10;

        glutPostRedisplay();
}

void Display()                // Обновление содержимого экрана
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glLoadIdentity();
// Позиционируем камеру
        gluLookAt(camera_pos.x, camera_pos.y, camera_pos.z,
                airplane_pos.x, airplane_pos.y, airplane_pos.z,
                0, 0, 1);
// Рисуем земную поверхность
        land.call_list();
// Переносим самолет в точку airplane_pos
        glTranslatef(airplane_pos.x, airplane_pos.y, airplane_pos.z);
// Поворачиваем самолет с использованием кватерниона airplane_quat
        glh_rotate(airplane_quat);
// Креним самолет на угол airplane_lurch
        glRotatef(airplane_lurch, 0, 1, 0);
// Рисуем самолет
        airplane.call_list();
}
```

Один из скриншотов работы программы изображен на рис. 4.17.

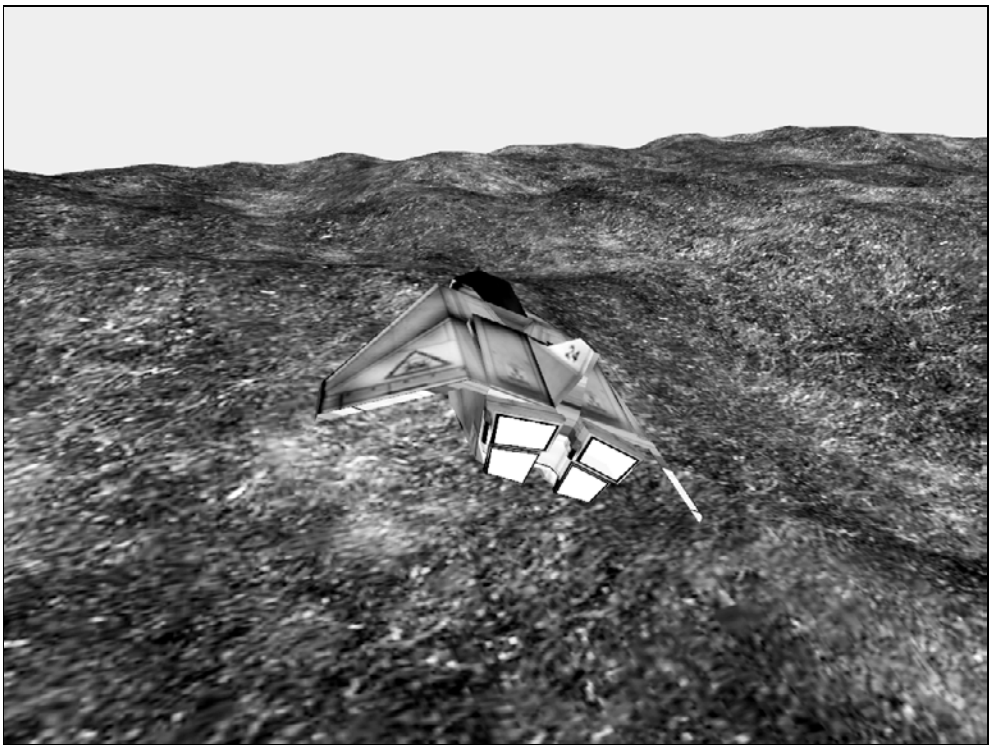


Рис. 4.17. Самолет, летящий над земной поверхностью

Все вроде бы нормально. Но вот отсутствие неба и прочей окружающей среды сильно портит впечатление от программы. Поэтому нам не помешало бы добавить какой-нибудь ландшафт, на фоне которого будет летать наш самолет (см. рис. 4.10).

Для моделирования окружающей среды мы воспользуемся классическим приемом, заключающимся в том, что окружающая среда рисуется при помощи куба, на стороны которого нанесено изображение окружающей среды вокруг объекта. Текстуры для граней куба я взял из NVIDIA OpenGL SDK. Это файлы CloudyHills_negx.tga, CloudyHills_negy.tga, CloudyHills_negz.tga, CloudyHills_posx.tga, CloudyHills_posy.tga, CloudyHills_posz.tga из каталога \NVSDK\Common\media\textures\cubemaps (рис. 4.18).

Ниже приведен фрагмент кода, отвечающий за загрузку текстур из файлов, а также за рисование граней куба с наложенными текстурами (листинг 4.11) (Ex10).

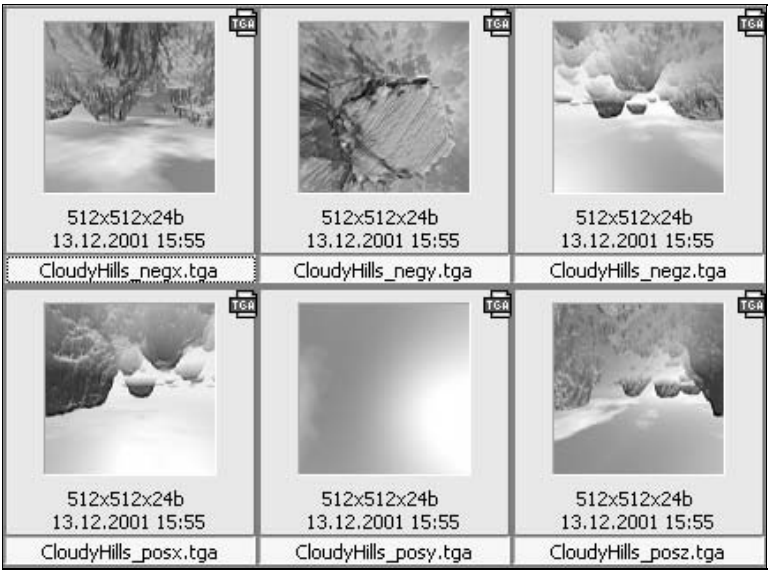


Рис. 4.18. Текстуры окружающей среды

Листинг 4.11

```
// Размер куба с окружающей средой
const float offset=100;

// Текстурные объекты граней куба
tex_object_2D wall_negx;
tex_object_2D wall_posx;
tex_object_2D wall_negy;
tex_object_2D wall_posy;
tex_object_2D wall_negz;
tex_object_2D wall_posz;

// Загружаем текстуру из файла в объект класса tex_object_2D
void read_to_tex_object_2D(string filename, tex_object_2D* tex_obj)
{
// Загружаем текстуру из ZIP-архива models.zip
       tga::tgaImage* image=read("..\\models.zip", filename);
// Если загрузка прошла не успешно, завершаем работу программы
       if (!image)
```

```
        {
                console.add(string("Can't load texture from
\"..\\models::")+filename+string("\""));
                console.exit();
                return;
        }
// Создаем текстурный объект и настраиваем параметры текстуры
        tex_obj->bind();
        tex_obj->parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        tex_obj->parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        tex_obj->parameter(GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        tex_obj->parameter(GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
// Загружаем текстуру в видеопамять
        texImage2D(image, tex_obj);
// Удаляем текстуру из памяти
        delete[] image->pixels;
        delete image;
}

void Init()                          // Инициализация OpenGL
{
...
// Загрузка текстур для рисования окружающего пространства
        read_to_tex_object_2D("CloudyHills_negx.tga", &wall_negx);
        read_to_tex_object_2D("CloudyHills_posx.tga", &wall_posx);
        read_to_tex_object_2D("CloudyHills_negy.tga", &wall_negy);
        read_to_tex_object_2D("CloudyHills_posy.tga", &wall_posy);
        read_to_tex_object_2D("CloudyHills_negz.tga", &wall_negz);
        read_to_tex_object_2D("CloudyHills_posz.tga", &wall_posz);

// Если произошла ошибка, то выходим из функции
        if (console.exiting)
                return;
...
}

// Рисуем куб, на грани которого наложены текстуры с
// изображением окружающей среды
```

```
void draw_walls()
{
// Настраиваем параметры OpenGL
       glPushAttrib(GL_ALL_ATTRIB_BITS);
       glEnable(GL_CULL_FACE);
       glCullFace(GL_BACK);
       glDisable(GL_LIGHTING);
       glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

// Рисуем грани куба
       wall_negz.bind();
       wall_negz.enable();
       glBegin(GL_QUADS);
              glTexCoord2f(1, 1);
              glVertex3f(-offset, -offset, -offset);

              glTexCoord2f(0, 1);
              glVertex3f(offset, -offset, -offset);

              glTexCoord2f(0, 0);
              glVertex3f(offset, offset, -offset);

              glTexCoord2f(1, 0);
              glVertex3f(-offset, offset, -offset);
       glEnd();

       wall_negz.disable();

       wall_posz.bind();
       wall_posz.enable();
       glBegin(GL_QUADS);
              glTexCoord2f(0, 0);
              glVertex3f(-offset, offset, offset);

              glTexCoord2f(1, 0);
              glVertex3f(offset, offset, offset);

              glTexCoord2f(1, 1);
              glVertex3f(offset, -offset, offset);
```

```
        glTexCoord2f(0, 1);
        glVertex3f(-offset, -offset, offset);
glEnd();

wall_posz.disable();

wall_negy.bind();
wall_negy.enable();
glBegin(GL_QUADS);
        glTexCoord2f(0, 1);
        glVertex3f(-offset, -offset, -offset);

        glTexCoord2f(0, 0);
        glVertex3f(-offset, -offset, offset);

        glTexCoord2f(1, 0);;
        glVertex3f(offset, -offset, offset);

        glTexCoord2f(1, 1);
        glVertex3f(offset, -offset, -offset);
glEnd();
wall_negy.disable();

wall_posy.bind();
wall_posy.enable();
glBegin(GL_QUADS);
        glTexCoord2f(1, 0);
        glVertex3f(offset, offset, -offset);

        glTexCoord2f(1, 1);;
        glVertex3f(offset, offset, offset);

        glTexCoord2f(0, 1);
        glVertex3f(-offset, offset, offset);

        glTexCoord2f(0, 0);
        glVertex3f(-offset, offset, -offset);
glEnd();
wall_posy.disable();
```

```
        wall_negx.bind();
        wall_negx.enable();
        glBegin(GL_QUADS);
                glTexCoord2f(0, 0);
                glVertex3f(-offset, offset, -offset);

                glTexCoord2f(1, 0);
                glVertex3f(-offset, offset, offset);

                glTexCoord2f(1, 1);
                glVertex3f(-offset, -offset, offset);

                glTexCoord2f(0, 1);
                glVertex3f(-offset, -offset, -offset);
        glEnd();
        wall_negx.disable();

        wall_posx.bind();
        wall_posx.enable();
        glBegin(GL_QUADS);
                glTexCoord2f(1, 1);
                glVertex3f(offset, -offset, -offset);

                glTexCoord2f(0, 1);
                glVertex3f(offset, -offset, offset);

                glTexCoord2f(0, 0);
                glVertex3f(offset, offset, offset);

                glTexCoord2f(1, 0);
                glVertex3f(offset, offset, -offset);
        glEnd();
        wall_posx.disable();

        glPopAttrib();
}

void Display()                // Обновление содержимого экрана
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
        glLoadIdentity();

        gluLookAt(camera_pos.x, camera_pos.y, camera_pos.z,
               airplane_pos.x, airplane_pos.y, airplane_pos.z,
               0, 0, 1);

        land.call_list();                    // Вывод объекта на экран

        glPushMatrix();
        glTranslatef(airplane_pos.x, airplane_pos.y, airplane_pos.z);
        glh_rotate(airplane_quat);
        glRotatef(airplane_lurch, 0, 1, 0);
        airplane.call_list();
        glPopMatrix();

// Позиционируем куб
        glTranslatef(0, 0, 50);
        glRotatef(90, 1, 0, 0);
// Рисуем куб с окружающей средой
        draw_walls();
}
```

Так как каждая текстура имеет объемный код загрузки, а нам надо загрузить в видеопамять целых шесть текстур, в программе объявлена специальная функция `read_to_tex_object_2D`, загружающая текстуру из файла в видеопамять. Кроме собственно загрузки, программа настраивает параметры фильтрации текстуры и устанавливает параметры оболочки для текстурных координат `s` и `t` (`GL_TEXTURE_WRAP_S` и `GL_TEXTURE_WRAP_T`) в `GL_CLAMP_TO_EDGE` для борьбы со швами между текстурами. В целом код довольно тривиален, и поэтому не нуждается в комментариях.

4.4.3. Краткое описание структур и функций библиотеки NV_UTIL, отвечающих за работу с файлами формата ASE

В этом разделе мы рассмотрим основные структуры и функции библиотеки NV_UTIL, отвечающие за работу с файлами формата ASE. Исходный текст этих структур и функций вы можете найти в файлах \NVSDK\OpenGL\include\nv_util\nv_ase.h и \NVSDK\OpenGL\src\libs\nv_util\nv_ase.cpp.

Я долго думал о том, стоит ли включать этот раздел в книгу. Дело в том, что исходный код библиотеки NV_UTIL, отвечающий за работу с файлами формата ASE, имеет очень большой объем и очень сложную структуру. В то же время большинство читателей этой книги, скорее всего, будут создавать свои программы с использованием библиотеки ASE Reader, которая является надстройкой над библиотекой NV_UTIL. Но, с другой стороны, для того, чтобы эффективно пользоваться какой-либо надстройкой, очень полезно иметь хотя бы поверхностные знания о нижнем уровне под надстройкой.

Поэтому в этом разделе я попробую рассказать об основных принципах работы функций библиотеки NV_UTIL, отвечающих за работу с файлами формата ASE, расположенных в пространстве имен ase.

Библиотека NV_UTIL позволяет загружать модель из файла формата ASE. Вывод сцены на экран осуществляет пользовательская программа, которая анализирует данные, загруженные при помощи библиотеки NV_UTIL. При этом надо учитывать одно обстоятельство — формат ASE предназначен для хранения сцен для систем 3D-моделирования, наподобие 3D Studio MAX. Поэтому современные ускорители не в состоянии выводить на экран сцены и файлы формата ASE один-в-один. В результате программа, как правило, игнорирует большинство "продвинутых" составляющих сцены вроде композитных материалов и т. д. А художники, создающие модели, заранее информируются о том, какие именно возможности не поддерживает программа вывода ASE-моделей на экран, чтобы не использовать их в своих моделях.

Для загрузки сцен из файлов формата ASE используются две функции:

```
extern model * load(const char * filename, float scale);
extern model * load(const char * buf, unsigned int size, float scale);
```

Первая функция загружает сцену из файла `filename` с последующим масштабированием с коэффициентом `scale`. Вторая функция загружает сцену из буфера `buf` размера `size`, масштабируя ее с коэффициентом `scale`. Буфер может быть сформирован, например, функцией чтения файлов из ZIP-архивов библиотеки NV_UTIL.

Обе функции возвращают указатель на экземпляр класса `model`, в котором хранится информация о загруженной сцене. Определение класса `model` приведено ниже:

```
    struct model
    {
        model() : name(0), time(0) {}
        ~model();

// Название сцены
        char *      name;
        geom_array  root_geom;
```

```
// Массив указателей на геометрические объекты
        geom_array  geom;
// Массив указателей на материалы
        mat_array   mat;
        tex_map     tex;
        int         time;
    };
```

В этой структуре для нас интересны два поля — `geom` и `mat`, которые представляют собой массивы указателей на геометрические объекты и материалы:

```
    typedef std::vector<geomobj*>   geom_array;
    typedef std::vector<matobj*>    mat_array;
```

При выводе сцены на экран программа выводит геометрические объекты сцены. Для этого она просматривает все геометрические объекты путем перебора всех элементов массива `geom_array` с последующим анализом экземпляров классов `geomobj`. Ниже приведено объявление структуры `geom_obj`:

```
    struct geomobj
    {
        geomobj();
        ~geomobj();

        typedef enum {
            polygonal,
            patched
        } geomtype;

        geomtype        type;   // tells if the geomobj is polygonal or
patched

        char *          name;   // object name
        geomobj *       parent; // ref to the parent geomobj
        geom_array      children;

        float           bone_offset_tm[16]; // bone offset transform
        float           tm[16]; // world transform
        float           rtm[16]; // 4x4 transform from the parent

        // polygonal
        float *         v;      // vertices
```

```
        float *         n;      // normals
        float *         t;      // texcoords
        float *         c;      // vertex colors
        unsigned int    numn;   // number of vertex normals
        unsigned int    numt;   // number of texcoords
        unsigned int    numv;   // number of vertices
        unsigned int    numc;   // number of vertices color
        unsigned int *  fvn;    // indices for the vertex normals indexed
by face index

        unsigned int *  fsubmat; // submaterial per face
        unsigned int *  f;      // faces
        unsigned int *  tf;     // texture faces
        float *         fn;     // face normals
        unsigned int *  fmapv;  // face map vertices
        unsigned int    numf;   // number of faces
        unsigned int    numtf;  // number of texture faces
        unsigned int *  smg;    // smoothing group

        // patches
        int             numverts;
        float *         verts;
        int             numvecs;
        float *         vecs;
        int             numedges;
        int *           edges;
        int             numpatches;
        patch *         patches;



        int             numtvchannels; // normally 2, where channel 0 is
vertex colors
                                       // and channel 1 is texcoords
        int *           numtvverts;  // number of tvverts per channel
        float **        tvverts;     // array of tvverts per channel
        int **          tvpatches;   // array of indices to tvverts per
channel
                                       // (defining a tvpatch (dim = 4))
```

```
        // animation
        animdata *      anim;   // ref to the animation data


        // material
        int             matidx; // index in the material array


        // vertex weighting data - may have to be reordered
        unsigned int    numbv;      // number of blended vertices
        float *         bv;         // blended vertices {x,y,z,w} where w
is the weight
        float **        bmatref;    // per blended vertex geom obj matrix
tm reference
        unsigned int *  vbv;        // vertex indices
        geomobj **      bgeomref;   // per blended vertex geom obj
reference
    };
```

В этой структуре находится описание геометрического объекта. Как видно из описания, объект может быть полигональным или криволинейным. Имя объекта храниться в поле `name`.

Если объект является полигональным, то в области памяти, на которую указывает указатель `f`, хранится массив граней объектов. Каждая грань задается тройкой индексов вершин. Каждая такая тройка — это индексы в массиве вершин, на который указывает указатель `v`. Индексы нормалей в вершинах хранятся в массиве `*fvn`, а сами нормали — в массиве `*n`. Индексы текстурных координат хранятся в массиве `*tf`, а сами текстурные координаты хранятся в массиве `*t`.

Индекс материала объекта находится в поле `matidx`. Зная этот индекс материала, мы сможем легко получить из массива `mat` структуры `model` указатель на сам материал объекта.

Материал объекта хранится в структуре `matobj`:

```
    typedef std::vector<mapobj*>    map_array;


    struct matobj
    {
        matobj() : name(0), classname(0), shader(0), twosided(false),
falloff(false), xp_type(0) {};
        ~matobj();


        char *          name;
        char *          classname;
```

```
    float           ambient[3];
    float           diffuse[4];
    float           specular[4];
    float           shine;
    float           shinestrength;
    float           transparency;
    float           wiresize;
    char *          shader;
    float           xp_falloff;
    float           selfillum;
    bool            twosided;
    bool            falloff;
    char *          xp_type;

    mat_array       submat;

    map_array       map_ambient;
    map_array       map_generic;
    map_array       map_diffuse;
    map_array       map_specular;
    map_array       map_bump;
    map_array       map_selfillum;
    map_array       map_reflect;
    map_array       map_shine;
    map_array       map_shinestrength;
    map_array       map_opacity;
    map_array       map_refract;
    map_array       map_filtercolor;
};
```

Из всего множества полей для нас, в первую очередь, интересны лишь несколько полей: название материала (`name`), диффузный цвет материала (`diffuse`), рассеянный цвет материала (`ambient`), цвет зеркальных бликов (`specular`), прозрачность (`transparency`), указатели на массив диффузных текстурных карт (`map_diffuse`) и зеркальных текстурных карт (`map_refract`) материала. Информация о текстурной карте материала хранится в структуре `mapobj`:

```
struct mapobj
{
```

```
        mapobj() : name(0), classname(0), bitmap(0), map_type(0),
bitmap_filter(0) {};
        ~mapobj();

        char *          name;
        char *          classname;
        unsigned int    subno;
        float           amount;
        char *          bitmap;
        char *          map_type;
        float           u_offset;
        float           v_offset;
        float           u_tiling;
        float           v_tiling;
        float           angle;
        float           blur;
        float           blur_offset;
        float           noise_amt;
        float           noise_size;
        unsigned int    noise_level;
        float           noise_phase;
        char*           bitmap_filter;

        map_array       map_generic;
};
```

Из всех полей этой структуры обычно используется только поле с именем файла, в котором храниться текстура — `bitmap`.

Возможно, у вас сейчас рябит в глазах от такого количества структур и полей. Ничего страшного. После прочтения следующего раздела все встанет на свои места.

4.4.4. Краткое описание внутреннего устройства библиотеки ASE Reader

Этот раздел преследует две цели. Во-первых, в нем мы рассмотрим пример библиотеки, анализирующей информацию, возвращаемую функциями загрузки ASE-файлов библиотеки NV_UTIL с последующим выводом сцены на экран. Во-вторых, вы сможете самостоятельно доработать библиотеку ASE Reader под свои задачи, добавив в нее недостающие возможности.

Возможно, код библиотеки ASE Reader местами покажется вам нелогичным и некрасивым. Дело в том, что библиотека ASE Reader написана не "с нуля", а на базе библиотеки из NVIDIA OpenGL SDK, которую можно найти в каталоге \NVSDK\OpenGL\src\demos\height_fog. Для того чтобы NVIDIA могла отслеживать изменения в своей библиотеке, я воспользовался следующим приемом. В начале файла ase.h находится директива "#define __ASE_EXT 1, а весь новый код расположен между директивами #ifdef __ASE_EXT и #endif. Таким образом, для получения из библиотеки ASE Reader оригинальной библиотеки NVIDIA достаточно убрать всего одну строку (#define __ASE_EXT 1) в файле ase.h. Для экономии места, в приводимых фрагментах исходного кода будет показан только итоговый код без директив вроде #ifdef и #endif.

Рассмотрение библиотеки ASE Reader мы начнем с объявления класса CAseModel, отвечающего за работу с файлами формата ASE:

```
class CAseModel
{
    public:
        CAseModel();
        ~CAseModel();
// Загружает модель из ZIP-архива
        bool LoadModel(char *filename, char *modelname);
// Выводит модель на экран
        void DrawModel(bool textured);
// Определяет размеры модели
        void GetExtents(vec3f &minExtent, vec3f &maxExtent);
// Выгружает модель из оперативной памяти
        void Clear(bool clearTextures=true);
// Возвращает информацию о модели
        list<string> get_info();

// Указатель на массив объектов модели
        modelData *data;
// Количество объектов в модели
        int numChunks;
};
```

Как видно, вся информация об объектах храниться в массиве, на который показывает указатель *data. Каждый элемент массива является экземпляром класса modelData, определение которого приведено ниже:

```
typedef struct _modelData
{
```

```
// Указатель на массив граней, в котором хранятся индексы вершин,
// нормалей и текстурных координат других массивов
    int *indices;
// Указатель на массив координат вершин
    float *vertices;
// Указатель на массив нормалей
    float *normals;
// Указатель на массив текстурных координат
    float *texcoords;
// Количество граней в объекте
    int numfaces;
// Диффузный цвет объекта
    float diffuse[4];
// Матрица текущей трансформации (на которую умножается матрица модели)
    float tm[16];
// Текстурный объект с диффузной текстурой
    tex_object_2D* diffuse_texture;
// Название объекта
    string* name;
} modelData;
```

Как вы помните, в библиотеке ASE Reader загрузка модели формата ASE осуществляется при помощи метода `LoadModel`. Первой командой, которую выполняет метод `LoadModel`, является вызов метода `Clear` удаления информации о предыдущей модели (листинг 4.12).

Листинг 4.12

```
void CAseModel::Clear(bool clearTextures)
{
// Перебираем все объекты модели
    for (int i = 0; i < numChunks; i++)
    {
// Удаляем массив индексов текущего объекта
        if (data[i].indices)
            {
                delete [] data[i].indices;
                data[i].indices=0;
            }
```

```
// Удаляем массив нормалей
        if (data[i].normals)
              {
                    delete [] data[i].normals;
                    data[i].normals=0;
              }
// Удаляем массив текстурных координат
        if (data[i].texcoords)
              {
                    delete [] data[i].texcoords;
                    data[i].texcoords=0;
              }
// Удаляем массив вершин
        if (data[i].vertices)
              {
                    delete [] data[i].vertices;
                    data[i].vertices=0;
                    data[i].numfaces=0;
              }
// Удаляем массив текстур
        if (data[i].name && clearTextures)
              {
                    delete data[i].name;
                    data[i].name=0;
              }
              if (data[i].diffuse_texture && clearTextures)
              {
// Выгружаем текстуру из оперативной памяти с использованием
// диспетчера текстур

       TextureManager.unload_texture(data[i].diffuse_texture);
                    data[i].diffuse_texture=0;
              }
    }
// Удаляем сам массив объектов
       if (data && clearTextures)
       {
              delete [] data;
```

```
            data=0;
            numChunks=0;
        }
}
```

Обратите внимание, что программа-метод `Clear` не удаляет текстурные объекты модели, если в качестве параметра передано `false`. Эта особенность полезна, когда пользователь поместил модель в дисплейный список и хочет выгрузить ее из оперативной памяти. Но поскольку текстурные объекты не помещаются в дисплейный список, нельзя выгружать их из памяти, т. к. в противном случае объект будет отображаться без текстур. После того как отпадает необходимость в дисплейном списке, следует еще раз вызвать метод `Clear`, но у же с параметром `true`, для того, чтобы окончательно выгрузить модель из оперативной памяти. Но, по большому счету, в этом нет необходимости, т. к. этот метод автоматически вызывается методом `LoadModel` и деструктором `~CAseModel`.

Но вернемся к рассмотрению метода `LoadModel`. После удаления информации о предыдущей модели он загружает новую модель средствами библиотеки NV_UTIL.

```
// Различные служебные переменные, которые используются методом LoadModel
// Указатель на модель, загруженную средствами NV_UTIL
    ase::model *m;
// Указатель на текущий материал
    matobj * mat = NULL;
// Указатель на текущую карту
    mapobj * map= NULL;
// Буфер, в который будет загружен файл из ZIP-архива
    unsigned char * buf;
// Размер буфера
    unsigned int bufsize;
// Текущий геометрический объект
    geomobj * geom;
// Номер обрабатываемого геометрического объекта
    int n = 0;
// В этих переменных хранятся i*3, i*6 и i*9
    int i_times_3;
    int i_times_6;
    int i_times_9;
```

```
// Загружаем файл из ZIP-архива
    buf = unzip::open(filename, modelname, &bufsize);
       if (!buf)
              return false;

// Загружаем модель из буфера
    m = load((char *)buf, bufsize, 1.0f);
// Удаляем буфер (он нам больше не нужен)
    delete [] buf;
```

После этого метод `LoadModel` анализирует количество геометрических объектов в загруженной модели и выделяет для них память:

```
// Определяем количество объектов в модели
    numChunks = m->geom.size();
// Выделяем для них память
    data = (modelData*)malloc(numChunks * sizeof(modelData));
```

Далее метод перебирает все объекты модели и заносит их в свой экземпляр объекта `CAseModel`. Далее приведен начальный фрагмент цикла перебора объектов. В этом фрагменте кода программа пропускает пустые геометрические объекты, а также определяет имя объекта и количество граней в нем.

```
// Создаем интерактор и устанавливаем его позицию в начало
// вектора объектов
    geom_it it = m->geom.begin();
// Пока интерактор не дошел до конца вектора объектов...
    while (it != m->geom.end())
    {
        geom = *it;

// Если в геометрическом объекте 0 граней или его имя Bip,
// то пропускаем его
        if ((geom->numf == 0) || (strstr(geom->name,"Bip")))
        {
            ++it;
            continue;
        }

// Определяем количество полигонов в объекте
        data[n].numfaces = geom->numf;
```

```
// Определяем имя объекта
        data[n].name=new string;
        *(data[n].name)=geom->name;
```

Следующим этапом является определение параметров материала текущего объекта, в частности, диффузного цвета и диффузной текстуры:

```
// Установка диффузного цвета по умолчанию в белый
              data[n].diffuse[0]=1.0;
              data[n].diffuse[1]=1.0;
              data[n].diffuse[2]=1.0;
              data[n].diffuse[3]=1.0;
// Объект по умолчанию не имеет диффузной текстуры
              data[n].diffuse_texture=0;

// Если индекс материала находится в допустимых пределах
              if (geom->matidx >= 0)
        {
            mat = m->mat[geom->matidx];
// И если материал несоставной
            if (mat->submat.size() == 0)
            {
// Получаем диффузный цвет объекта
                memcpy(data[n].diffuse, mat->diffuse, 4*sizeof(float));
// Если в векторе диффузных карт есть хотя бы одна карта
              if (mat->map_diffuse.size()!=0)
              {
// Берем в качестве файла с текстурой текстуру первой текстурной карты
                    char* bitmap=mat->map_diffuse[0]->bitmap;
// Загружаем текстуру из файла при помощи диспетчера текстур

                    data[n].diffuse_texture=TextureManager.load_texture
              (filename, bitmap);
              }
            }
        }
```

Вы, наверное, заметили, что для создания текстурных объектов используется диспетчер текстур. Дело в том, что этот диспетчер текстур автоматически распознает ситуации, когда программа пытается загрузить несколько копий одной и той же текстуры в видеопамять и автоматически заменяет эти копии одной ссылкой на текстурный объект. Текстурный объект про-

должает существовать до тех пор, пока не будет выгружена последняя текстура, ссылающаяся на него. Подробнее диспетчер текстур будет рассмотрен в конце этого подраздела.

После загрузки материала объекта функция `LoadModel` получает матрицу трансформации объекта (на эту матрицу впоследствии будет умножаться текущая матрица модели):

```
// Если в объекте есть матрица трансформации
if (!geom->numbv)
        {
// Получаем матрицу трансформации
            memcpy(data[n].tm, geom->tm, 16*sizeof(float));
        }
        else
       {
// Иначе полагаем, что матрица трансформации равна единичной
              matrix4f m1;
              m1.get_value(&data[n].tm[0]);
       }
```

Следующий этап — получение массива нормалей геометрического объекта. При этом данные из массивов копируются в одномерный массив для того, чтобы потом объект мог быть нарисован одной командой `glDrawElements`.

```
// Если количество нормалей больше 0
        if (geom->numn)
        {
// Выделяем память для хранения нормалей
            data[n].normals = (float*)malloc(geom->numf *
sizeof(GLfloat)*3*3);
// Перебираем все нормали и копируем их в массив нормалей
            for (unsigned int i = 0; i < geom->numf; ++i)
            {
                i_times_3 = i*3;
                i_times_9 = i*9;
                memcpy(&data[n].normals[i_times_9],   &geom->n
[geom->fvn[i_times_3] * 3],     sizeof(GLfloat)*3);
                memcpy(&data[n].normals[i_times_9+3], &geom->n
[geom->fvn[i_times_3 + 1] * 3], sizeof(GLfloat)*3);
                memcpy(&data[n].normals[i_times_9+6], &geom->n
[geom->fvn[i_times_3 + 2] * 3], sizeof(GLfloat)*3);
            }
```

```
        }
// Если нормалей нет, то устанавливаем указатель на массив нормалей в 0
        else data[n].normals = 0;
```

Затем метод `LoadModel` аналогичным образом копирует в свои внутренние массивы текстурные координаты и вершины объекта:

```
// Если количество текстурных координат больше 0
        if (geom->numt)
        {
// Выделяем память для хранения текстурных координат
              data[n].texcoords = (float*)malloc(geom->numf *
sizeof(GLfloat)*3*2);
// Копируем текстурные координаты в массив
            for (unsigned int i = 0; i < geom->numf; i++)
            {
                i_times_3 = i*3;
                i_times_6 = i*6;
                memcpy(&data[n].texcoords[i_times_6],
&geom->t[geom->tf[i_times_3] * 3], sizeof(GLfloat)*2);
                memcpy(&data[n].texcoords[i_times_6+2],
&geom->t[geom->tf[i_times_3 + 1] * 3], sizeof(GLfloat)*2);
                memcpy(&data[n].texcoords[i_times_6+4],
&geom->t[geom->tf[i_times_3 + 2] * 3], sizeof(GLfloat)*2);
data[n].texcoords[i_times_6+1]=-data[n].texcoords[i_times_6+1];
data[n].texcoords[i_times_6+3]=-data[n].texcoords[i_times_6+3];
data[n].texcoords[i_times_6+5]=-data[n].texcoords[i_times_6+5];
            }
        }
        else data[n].texcoords = 0;


// Выделяем память для массива вершин
        data[n].vertices = (GLfloat*)malloc(geom->numf *
sizeof(GLfloat)*3*3);
// Выделяем память для массива индексов
        data[n].indices = (int*)malloc(geom->numf * sizeof(int)*3*3);
// Перебираем грани для массива граней
        for (unsigned int i = 0; i < geom->numf; ++i)
        {
            i_times_3 = i*3;
            i_times_9 = i*9;
```

```
// Копируем индексы в массив индексов
            for (int j = 0; j < 9; j++)
                data[n].indices[i_times_3+j] = i_times_3+j;
// Копируем вершины в массив вершин
            memcpy(&data[n].vertices[i_times_9],
&geom->v[geom->f[i_times_3] * 3],     sizeof(GLfloat)*3);
            memcpy(&data[n].vertices[i_times_9+3],
&geom->v[geom->f[i_times_3 + 1] * 3], sizeof(GLfloat)*3);
            memcpy(&data[n].vertices[i_times_9+6],
&geom->v[geom->f[i_times_3 + 2] * 3], sizeof(GLfloat)*3);
        }
```

На этом цикл обработки одного объекта заканчивается. Остается только увеличить счетчик количества объектов и перевести интерактор текущего объекта на следующий объект.

```
        n++;
        ++it;
// Переходим к следующей итерации цикла
    }
```

Последнее, что выполняет функция `LoadModel` — сохраняет количество непустых геометрических объектов и удаляет уже ненужную ASE-модель, загруженную библиотекой NV_UTIL.

```
// Сохраняем число объектов
    numChunks = n;
// Удаляем модель
    delete m;
    return true;
}
```

Что делает приложение, когда загружена модель? Конечно же, выводит ее на экран или в дисплейный список при помощи метода `DrawModel`. Но перед рисованием модели пользовательская программа обычно получает размеры модели при помощи метода `GetExtents`, который перебирает все объекты сцены и определяет минимальное и максимальное значения компонентов координат вершин объектов (листинг 4.13).

Листинг 4.13

```
void CAseModel::GetExtents(vec3f &minExtent, vec3f &maxExtent)
{
// Устанавливаем размерности в минимальное и максимальное значения,
```

```
// которое может принимать тип float
    vec3f smallest(FLT_MAX, FLT_MAX, FLT_MAX);
    vec3f largest(-FLT_MAX, -FLT_MAX, -FLT_MAX);

    int i_times_3;
    vec3f vertex;

// Перебираем объекты
    for (int n = 0; n < numChunks; n++)
    {
// Перебираем вершины объектов
        for (int i = 0; i < data[n].numfaces; i++)
        {
            i_times_3 = i*3;
// Получаем координаты текущей вершины и сравниваем ее
// компоненты с smallest и largest
            vertex = vec3f(data[n].vertices[i_times_3],
data[n].vertices[i_times_3+1], data[n].vertices[i_times_3+2]);
            if (vertex[0] < smallest[0]) smallest[0] = vertex[0];
            if (vertex[1] < smallest[1]) smallest[1] = vertex[1];
            if (vertex[2] < smallest[2]) smallest[2] = vertex[2];

            if (vertex[0] > largest[0]) largest[0] = vertex[0];
            if (vertex[1] > largest[1]) largest[1] = vertex[1];
            if (vertex[2] > largest[2]) largest[2] = vertex[2];
        }
    }

    minExtent = smallest;
    maxExtent = largest;
}
```

После этого приложение масштабирует модель с использованием матрицы модели, потом рисует модель с использованием метода `DrawModel`. Этот метод имеет очень простую структуру. Он просто перебирает в цикле все геометрические объекты модели и потом выводит их при помощи команды `glDrawElements` (листинг 4.14).

Листинг 4.14

```
void CAseModel::DrawModel(bool textured)
{
    glPushAttrib(GL_ALL_ATTRIB_BITS);
    glPushMatrix();
// Если модель не пустая
    if (data)
// Перебираем все объекты модели
    for (int n = 0; n < numChunks; n++)
    {
       if (data[n].vertices)
       {
          glPushMatrix();
// Умножаем текущую матрицу модели на матрицу трансформации
          glMultMatrixf(data[n].tm);
// Устанавливаем текущий цвет модели
          glColor4fv(data[n].diffuse);
// Если есть массив нормалей
          if (data[n].normals)
          {
// Устанавливаем массив нормалей
            glEnableClientState(GL_NORMAL_ARRAY);
            glNormalPointer(GL_FLOAT, 0, data[n].normals);
          }
          else glDisableClientState(GL_NORMAL_ARRAY);
// Если есть массив текстурных координат и разрешено
// текстурирование модели
          if ((data[n].texcoords) && (textured))
          {
// Устанавливаем массив текстурных координат
            glEnableClientState(GL_TEXTURE_COORD_ARRAY);
            glTexCoordPointer(2, GL_FLOAT, 0, data[n].texcoords);
          }
          else glDisableClientState(GL_TEXTURE_COORD_ARRAY);
// Устанавливаем массив вершин
          glEnableClientState(GL_VERTEX_ARRAY);
          glVertexPointer(3, GL_FLOAT, 0, data[n].vertices);
```

```
// Если разрешено текстурирование и имеется диффузная текстура
          if (textured && data[n].diffuse_texture)
          {
// Привязываем диффузную текстуру к объекту
               data[n].diffuse_texture->bind();
               data[n].diffuse_texture->enable();
               glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_MODULATE);
          }
// Рисуем объект
          glDrawElements(GL_TRIANGLES, data[n].numfaces*3,
GL_UNSIGNED_INT, data[n].indices);

          if (textured && data[n].diffuse_texture)
                    data[n].diffuse_texture->disable();

          glPopMatrix();
        }
    }
    glPopAttrib();
    glPopMatrix();
}
```

После того как мы рассмотрели класс `CAseModel`, мы можем переходить к рассмотрению диспетчера текстур. В библиотеке ASE Reader диспетчером текстур является объект `TextureManager` класса `CTextureManager`. Класс `CTextureManager` объявлен следующим образом:

```
class CTextureManager
{
public:
// Список текстур
       list<CTexture> textures;
       ~CTextureManager();
// Загружает текстуру в видеопамять и возвращает текстурный объект
       tex_object_2D* load_texture(string zip_filename, string
inzip_filename);
// Выгружает текстурный объект из памяти
       void unload_texture(tex_object_2D* pTexture);
// Возвращает текстовую информацию о текущем состоянии диспетчера текстур
       list<string> get_info();
};
```

Как видно, информация о текстурах хранится в списке `textures`. Каждый элемент списка (класс `CTexture`) хранит информацию об имени файла, из которого была взята текстура из текстурного объекта, и количество ссылок на текстурный объект:

```
class CTexture
{
public:
// Имя файла текстуры
       string filename;
// Количество ссылок на текстурный объект, использующий текстуру из этого
// файла
       int count;
// Указатель на текстурный объект
       tex_object_2D* pTexture;
};
```

Загрузка текстуры в видеопамять осуществляется с использованием метода `load_texture`, который возвращает указатель на текстурный объект с загруженной текстурой (листинг 4.15).

Листинг 4.15

```
tex_object_2D* CTextureManager::load_texture(string zip_filename, string
inzip_filename)
{
// Выделяем имя файла (без пути) и помещаем в short_inzip_filename
       string short_inzip_filename=inzip_filename;
       int pos=inzip_filename.find_last_of("\\");
       if (pos!=-1)
       {
              if (pos!=short_inzip_filename.size()-1)
short_inzip_filename=inzip_filename.substr(pos+1, inzip_filename.size()-
pos-1);
              else
                     return 0;
       }

// Получаем "суперполное" имя файла (имя zip-архива::имя файла)
       string fullname=zip_filename+"::"+short_inzip_filename;
// Ищем в списке текстур текстуру с таким же именем
       for (list<CTexture>::iterator itor=textures.begin();
itor!=textures.end(); itor++)
```

```
        {
// Если текстура с таким именем есть
                if (itor->filename==fullname)
                {
// Увеличиваем счетчик ссылок
                        itor->count++;
// Возвращаем указатель на текстурный объект
                        return itor->pTexture;
                }
        }
// Создаем новую текстуру
        CTexture texture;
// Устанавливаем имя файла
        texture.filename=fullname;
// Устанавливаем счетчик количества ссылок в 1
        texture.count=1;
// Создаем новый текстурный объект
        texture.pTexture=new tex_object_2D;
// Делаем его активным и настраиваем параметры фильтрации
        texture.pTexture->bind();
        texture.pTexture->parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        texture.pTexture->parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Пытаемся загрузить файл из ZIP-архива
        tga::tgaImage* image=read(zip_filename, short_inzip_filename);
        if (!image)
        {
// Если такого файла в ZIP-архиве нет, для "очистки совести"
// пытаемся загрузить его по полному пути
                image=read(inzip_filename);
                if (!image)
                {
// Если файл загрузить не получилось — удаляем текстурный
// объект и возвращаем false
                        delete texture.pTexture;
                        return false;
                }
        }
// Загружаем текстуру (изображение) в видеопамять
        texImage2D(image, texture.pTexture);
```

```
// Помещаем текстуру в стек
       textures.push_back(texture);
// Возвращаем текстурный объект
       return texture.pTexture;
}
```

Для выгрузки текстуры из видеопамяти используется метод `unload_texture` (листинг 4.16).

Листинг 4.16

```
void CTextureManager::unload_texture(tex_object_2D* pTexture)
{
// Перебираем все текстуры
       for (list<CTexture>::iterator itor=textures.begin();
itor!=textures.end(); itor++)
       {
// Если текстура с таким текстурным объектом найдена
              if (itor->pTexture->texture==pTexture->texture)
              {
// Уменьшаем счетчик ссылок
                     itor->count--;
// Если счетчик ссылок равен 0
                     if (itor->count==0)
                     {
// Удаляем текстурный объект из видеопамяти
                            delete itor->pTexture;
// Удаляем текстуру из списка
                            textures.erase(itor);
                            return;
                     }
              }

       }
}
```

Если же вы забудете выгрузить все текстуры из видеопамяти, то деструктор диспетчера текстур автоматически удалит все текстурные объекты:

```
CTextureManager::~CTextureManager()
{
```

```
// Перебираем все текстуры
        for (list<CTexture>::iterator itor=textures.begin();
itor!=textures.end(); itor++)
        {
// Удаляем текстурный объект
                delete itor->pTexture;
                itor->pTexture=0;
        }
}
```

Иногда пользователю необходимо получить информацию о текущем состоянии диспетчера текстур. Сколько текстур загружено в видеопамять, сколько имеется ссылок на каждый текстурный объект и т. д. Эта информация может быть очень полезна, к примеру, для поиска разнообразных утечек видеопамяти. Для получения этой информации в классе `CTextureManager` имеется метод `get_info`, который возвращает список строк с информацией о текущем состоянии диспетчера текстур (листинг 4.17).

Листинг 4.17

```
list<string> CTextureManager::get_info()
{
// Список строк
        list<string> info;
        info.push_back("Texture Manager Information:");
// Определяем количество текстур, которые обслуживает диспетчер текстур
        char buf[200];
        sprintf(buf, "Total textures: %d", textures.size());
        info.push_back(buf);
// Если количество текстур не равно 0
        if (textures.size()!=0)
        {
                info.push_back("Textures list:");
// Счетчик общего количества ссылок на текстурные объекты
// диспетчера текстур
                int references=0;
// Перебираем все текстуры
                for (list<CTexture>::iterator itor=textures.begin();
itor!=textures.end(); itor++)
```

```
            {
                  info.push_back("------------------------------");
// Выводим название файла, из которого загружена текстура
                  info.push_back("Name: "+itor->filename);
// Выводим количество ссылок на текущий текстурный объект
                  sprintf(buf, "References count: %d", itor->count);
                  info.push_back(buf);
                  references+=itor->count;
            }
            info.push_back("-----------SUMMARY-------------");
// Выводим общее число ссылок на все текстурные объекты
            sprintf(buf, "Total references count: %d", references);
            info.push_back(buf);
// Определяем эффективность диспетчера текстур как (количество
// ссылок на текстурные объекты)/(количество текстур)
            sprintf(buf, "Texture manager effectuality: %f",
float(references)/textures.size());
            info.push_back(buf);
      }

// Возвращаем информацию о диспетчере текстур
      return info;
}
```

Класс `CAseModel` также умеет выдавать информацию о загруженной модели при помощи метода `get_info` (листинг 4.18).

Листинг 4.18

```
list<string> CAseModel::get_info(bool bShowTextureManagerInfo)
{
      list<string> info;

      info.push_back("ASE Model Information:");
// Выводим общее количество объектов в модели
      char buf[200];
      sprintf(buf, "Total objects: %d", numChunks);
      info.push_back(buf);
// Обнуляем счетчик общего количества граней в модели
      int faces=0;
```

```
// Перебираем все объекты модели
    for (int n = 0; n < numChunks; n++)
    {
            info.push_back("-------------------------------");
// Имя текущего объекта
            info.push_back(string("Object name: ") + *data[n].name);
// Если в объекте есть массив вершин
            if (data[n].vertices)
            {
// Выводим количество граней в текущем объекте
                    sprintf(buf, "Faces count: %d", data[n].numfaces);
                    info.push_back(buf);
// Увеличиваем счетчик количества граней в модели
                    faces+=data[n].numfaces;
            }
// Выводим диффузный цвет объекта
            sprintf(buf, "Diffuse Color: (%f, %f, %f, %f)",
float(data[n].diffuse[0]),
                    float(data[n].diffuse[1]),
float(data[n].diffuse[2]), float(data[n].diffuse[3]));
            info.push_back(buf);
// Имеется ли текстурная текстура у объекта
            if (data[n].diffuse_texture)
                    info.push_back("Diffuse texture: Yes");
            else
                    info.push_back("Diffuse texture: No");
// Имеет ли объект "текстурные координаты"
            if (data[n].texcoords)
                    info.push_back("Texture coordinates: Yes");
            else
                    info.push_back("Texture coordinates: No");
    }

    info.push_back("-----------SUMMARY-------------");
// Общее количество граней в модели
    sprintf(buf, "Total of faces count: %d", faces);
    info.push_back(buf);
    info.push_back("");
```

```
// Если необходимо, добавляем информацию о диспетчере текстур
        if (bShowTextureManagerInfo)
        {
                list<string> manager_info(TextureManager.get_info());
                info.splice(info.end(), manager_info);
        }
// Выводим информацию об объекте
        return info;
}
```

На этом рассмотрение исходного кода библиотеки ASE Reader можно считать законченным.

4.4.5. Создание сложной сцены в 3D Studio MAX и доработка библиотеки ASE Reader для отображения текстур отражения

Пока мы использовали в примерах относительно простые модели, содержащие один объект с одной текстурой. Но реальные программы, как правило, используют модели, состоящие из десятков объектов и текстур.

Поэтому для того, чтобы протестировать библиотеку ASE Reader в "боевых условиях", мы создадим в 3D Studio MAX сложную модель земной поверхности, содержащую холмы, дороги, реки, мосты и т. д. Но в связи с тем, что большинство читателей этой книги не являются художниками, мы немного упростим задачу. Вместо того чтобы создавать сложную модель "с нуля", мы возьмем одну из готовых сцен 3D Studio MAX и "раскрасим" ее при помощи редактора материалов.

В качестве сцены "для раскраски" я решил использовать сцену долины из файла valley.max, которая находится на втором диске 3D Studio MAX 5 в каталоге \ModelLibrary\Misc\. Внешний вид этой сцены приведен на рис. 4.19.

Сцена содержит более 110 объектов с общим количеством полигонов около 70 000. Изначально она не содержит текстур. Но это не должно стать для нас большой проблемой. Мы просто раскрасим сцену по своему вкусу.

Первым делом надо разгруппировать объекты сцены. Для этого выделите сцену (объект `Valley`) и выполните команду меню **Group | Ungroup**.

Раскраску мы начнем с озера (объект G_WATER01). Для его быстрого выделения удобнее всего воспользоваться диалоговым окном **Select Objects** (клавиша <H>), изображенным на рис. 4.20.

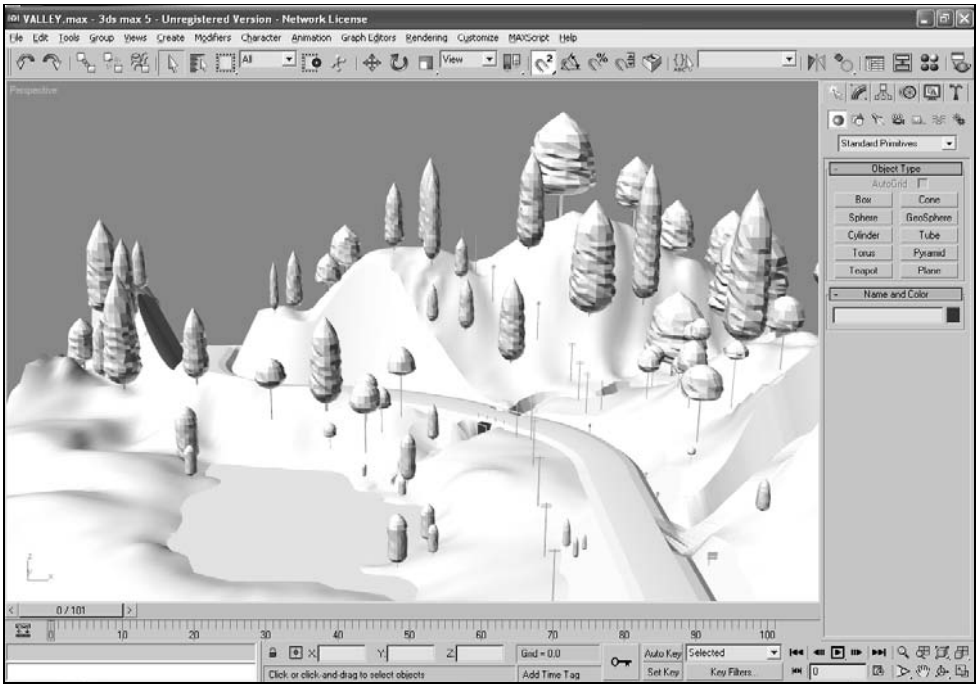


Рис. 4.19. Сцена долины из файла valley.max

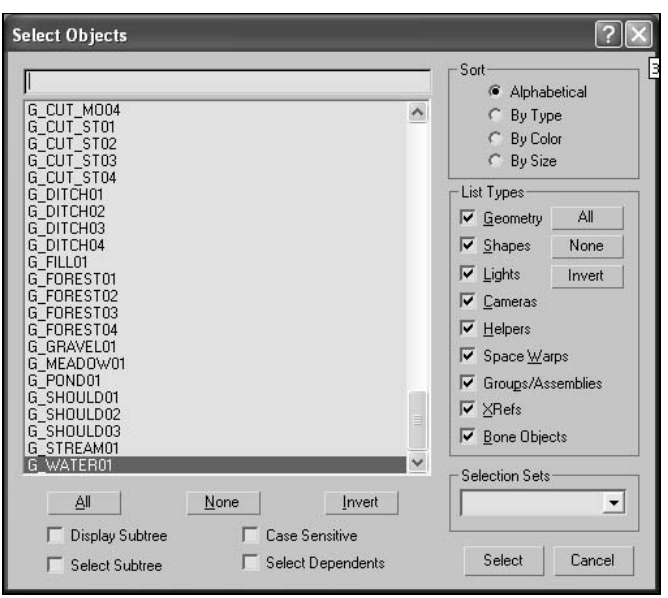


Рис. 4.20. Диалоговое окно **Select Object**

Создадим новый стандартный материал. В качестве диффузного цвета выберем белый цвет (255, 255, 255), а в качестве диффузной текстуры возьмем какую-нибудь текстуру воды, например, OPOCEAN2.JPG, из каталога \3dsmax5\maps\Water (разумеется, ее разрешение надо изменить на кратное степени 2). Теперь применяем полученный материал к объекту озера. Остается только сгенерировать текстурные координаты при помощи модификатора **UVW Map**, и объект озера готов.

Аналогичным образом "раскрасим" остальные элементы объекты сцены. В связи с большим количеством объектов на сцене и однотипностью операций мы не будем рассматривать процесс раскраски остальных объектов сцены. Очевидно, что он мало чем отличается от раскраски местности для примера с самолетом, который был рассмотрен в *разд. 4.4.2.*

Если же вам по каким-либо причинам некогда возиться с раскраской сцены, вы можете найти готовую "раскрашенную" сцену на CD с книгой в каталоге \Examples\Ch4-NV_UTIL\3DSMAX_MODELS\Valley (рис. 4.21).

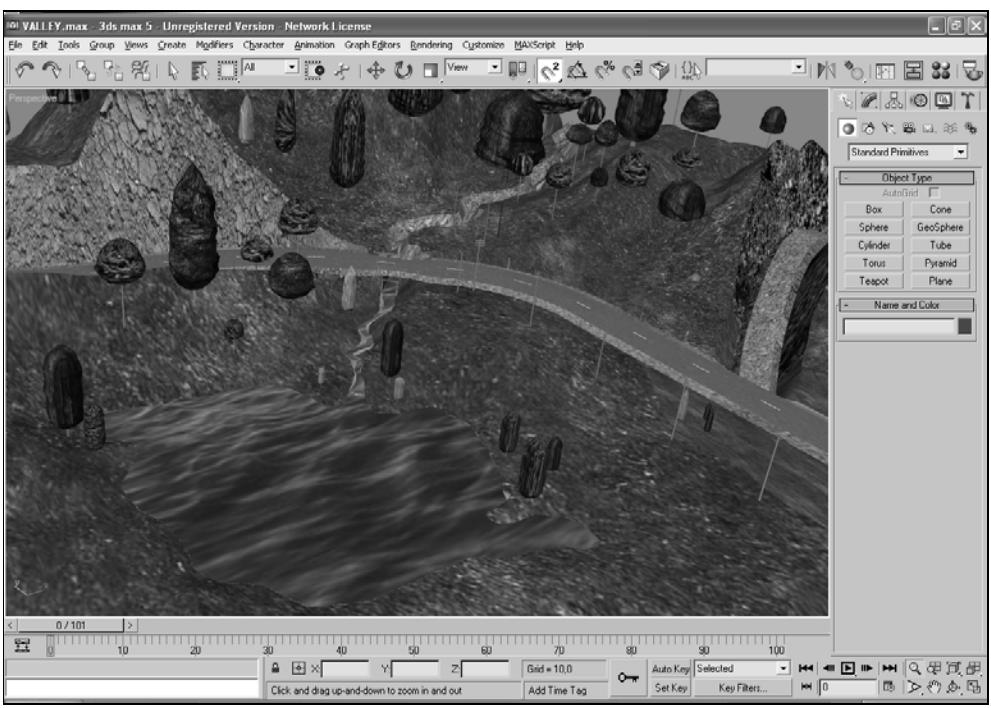


Рис. 4.21. Текстурированная модель долины

Пример, выводящий эту сцену на экран (Ex11), очень похож на пример Ex08. Главное различие между ними заключается в том, что в пример Ex11

был добавлен вывод статистики сцены в консоль (Ex11). Для этого после фрагмента кода, загружающего сцену из файла, была вставлена команда `console.add(model0.get_info())`. В результате для того, чтобы посмотреть информацию о загруженной модели достаточно нажать клавишу <~>. Из этой информации видно, что модель, изображенная на рис. 4.21, использует 115 текстур, которые преобразованы диспетчером текстур в 16 текстурных объектов. Общее количество полигонов в модели равно 69 153.

Долина в примере Ex11 имеет довольно симпатичный вид. Правда картину несколько портит неестественная вода, которая отличается от остальных объектов долины разве что текстурой, в то время как реальная вода частично отражает окружающую среду. Для имитации отражения в воде окружающей среды мы добавим в материал воды карту отражения. Для этого откройте в 3D Studio MAX 5 сцену долины, откройте редактор материалов и выберите материал воды, который наложен на озеро (в сцене, находящейся на CD-диске, поставляемом с книгой, он имеет название "Material #2"). Разверните свиток **Maps** и щелкните по кнопке напротив надписи **Reflection**. В открывшемся диалоговом окне выберите текстуру неба CLOUD2.JPG из каталога \3dsmax5\maps\Skies. Степень влияния карты отражения (поле **Amount**) установите в значение 70 (рис. 4.22).

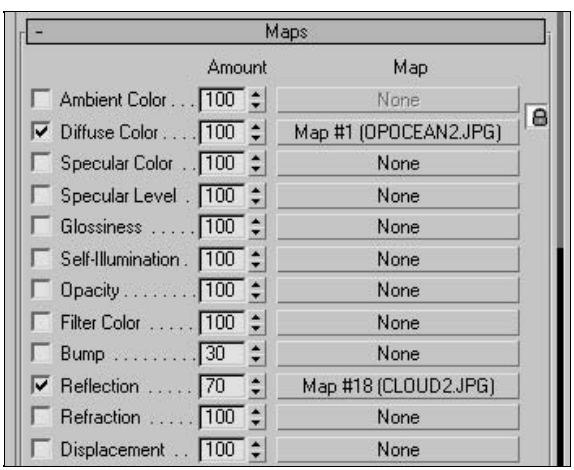


Рис. 4.22. Настройки карт материала воды для озера

Все было бы нормально, если бы не одно "но". Дело в том, что библиотека ASE Reader обрабатывает только диффузную текстурную карту, а остальные карты текстур, включая карту отражения, просто игнорируются. Выход из этой ситуации — доработать библиотеку ASE Reader. Что мы сейчас и сделаем (Ex12).

Для начала нам надо модифицировать структуру `modelData`, добавив в нее два новых поля, в которых будет храниться информация о карте отражения:

```
typedef struct _modelData
{
    int *indices;
    float *vertices;
    float *normals;
    float *texcoords;
    int numfaces;
    float diffuse[4];
    float tm[16];
// Цвет карты отражения. Принимает значение вида (1, 1, 1, Amount)
    float reflect_color[4];
    tex_object_2D* diffuse_texture;
// Текстурный объект, в котором хранится текстура отражения
    tex_object_2D* reflect_texture;
    string* name;
} modelData;
```

Теперь надо модифицировать код загрузки модели, расположенный в методе `LoadModel` (листинг 4.19).

Листинг 4.19

```
bool CAseModel::LoadModel(char *filename, char *modelname)
{
...
       if (geom->matidx >= 0)
        {
            mat = m->mat[geom->matidx];
            if (mat->submat.size() == 0)
            {
...
// Если имеется карта текстуры отражения
                            if (mat->map_reflect.size()!=0)
                            {
// Получаем имя файла текстуры отражения
                                    char* bitmap=mat->map_reflect[0]->
bitmap;
```

```
// Загружаем текстуру в видеопамять
data[n].reflect_texture=TextureManager.load_texture(filename, bitmap);
// Устанавливаем цвет карты отражения в (1.0, 1.0, 1.0, Amount)

                                    data[n].reflect_color[0]=1.0;
                                    data[n].reflect_color[1]=1.0;
                                    data[n].reflect_color[2]=1.0;
                                    data[n].reflect_color[3]=mat->
map_reflect[0]->amount;
                              }
...
```

Наконец, надо добавить в код вывода модели на экран второй проход для рисования карты отражения (листинг 4.20).

Листинг 4.20

```
void CAseModel::DrawModel(matrix4f world, bool textured)
{
...
// Первый проход (рисуется объект с диффузной текстурой)
...
// Второй проход
              if (textured && data[n].reflect_texture)
              {
// Включаем режим смешивания
                    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
                    glEnable(GL_BLEND);
// Выключаем модификацию Z-буфера для ускорения работы
                    glDepthMask(false);
// Изменяем функцию сравнения на GL_LEQUAL (по умолчанию она
// равна GL_LESS)
                    glDepthFunc(GL_LEQUAL);
// Включаем автоматическую генерацию сферических текстурных координат
                    glEnable(GL_TEXTURE_GEN_S);
                    glEnable(GL_TEXTURE_GEN_T);
                    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,
GL_SPHERE_MAP);
                    glTexGeni(GL_T, GL_TEXTURE_GEN_MODE,
GL_SPHERE_MAP);
```

```
// Устанавливаем режим наложения текстуры в GL_MODULATE
                    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_MODULATE);
// Активируем карту отражения
                    data[n].reflect_texture->bind();
                    data[n].reflect_texture->enable();
// Устанавливаем цвет отражения
                    glColor4fv(&data[n].reflect_color[0]);
// Рисуем объект
glDrawElements(GL_TRIANGLES, data[n].numfaces*3, GL_UNSIGNED_INT,
data[n].indices);
// Возвращаем OpenGL в исходное состояние
                    data[n].reflect_texture->disable();
                    glDisable(GL_TEXTURE_GEN_S);
                    glDisable(GL_TEXTURE_GEN_T);
                    glDisable(GL_BLEND);
                    glDepthMask(true);
                    glDepthFunc(GL_LESS);
                }
...
```

Как видно, объект с текстурой отражения рисуется поверх того же объекта, но с диффузной текстурой. При этом прозрачность отражения определяется альфа-каналом цвета объекта отражения. Возможно, вам довольно тяжело представить, глядя на исходный код программы, по какой формуле рассчитывается итоговый цвет изображения. Поэтому, на всякий случай, эта формула приведена ниже[2]:

$RGB = RGB_{cd} \times RGB_{td} \times (1-A_{cr}) + RGB_{tr} \times A_{cr}$;

$A=1$,

где:

- RGB и A — RGB и альфа-составляющие итогового изображения;
- RGB_{cd} — цвет объекта на первом проходе (когда на него накладывается диффузная текстура);
- RGB_{td} — диффузная текстура;

[2] Формула немного преобразована — для упрощения из нее убрано умножение на цвет объекта на втором проходе, который всегда равен (1, 1, 1). Альфа-компонент на самом деле рассчитывается по более сложной формуле, результат которой всегда равен 1.

- RGB_{tr} — текстура отражения;
- A_{cr} — альфа-компонент цвета объекта на втором проходе.

Хочу обратить ваше внимание на то, что на втором проходе используется функция глубины `GL_LEQUAL`. В принципе, эту функцию можно было бы заменить на `GL_EQUAL`. Но этого лучше не делать. Дело в том, что при использовании функций глубины `GL_EQUAL` и `GL_NOTEQUAL` у всех современных видеокарт выключается иерархический Z-буфер. В результате этого скорость работы программы может упасть до 30%. Правда, на видеокартах без иерархического Z-буфера скорость работы программы останется прежней. Но поскольку доля таких видеокарт на рынке с каждым годом уменьшается, нужно стараться избегать использования функций Z-буфера `GL_EQUAL` и `GL_NOTEQUAL` в своих программах.

На рис. 4.23 приведен скриншот из примера Ex12. На нем хорошо видно, что в озере появилось отражение неба.

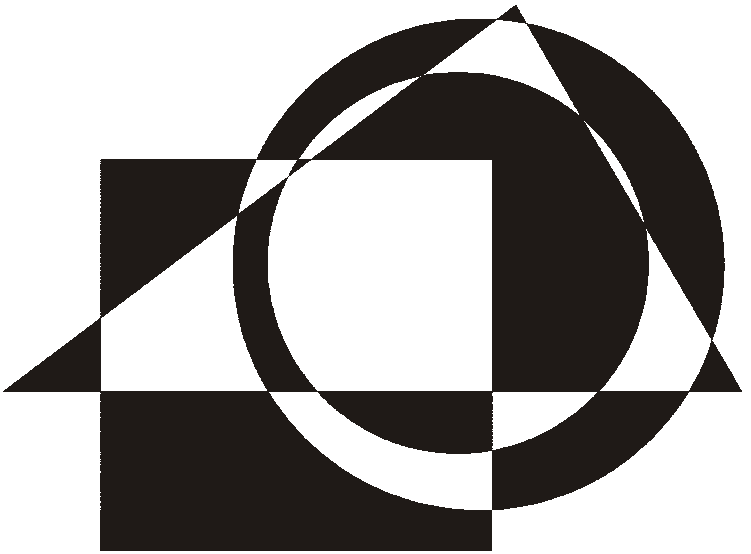
Рис. 4.23. Озеро, в котором отражается небо

В следующих главах книги мы продолжим дорабатывать нашу библиотеку ASE Reader, добавляя в нее новые возможности.

Заключение

В этой главе мы рассмотрели библиотеку NV_UTIL, которая позволяет работать с файлами форматов TGA, JPG, ASE и ZIP. Комплексное использование этих форматов удовлетворит потребности большинства программистов: формат TGA идеально подойдет для хранения высококачественных текстур, JPG — для текстур среднего качества, ASE — для 3D-моделей, а ZIP — для организации хранилищ файлов.

Мы создали универсальную функцию `read`, позволяющую программисту загружать текстуры из файлов форматов BMP, TGA и JPG, а также из архивов ZIP. Кроме того, мы разработали библиотеку ASE Reader, являющуюся настройкой над библиотекой NV_UTIL и позволяющую выводить на экран модель формата ASE практически любой степени сложности.



ЧАСТЬ II

РАСШИРЕНИЯ OPENGL

Глава 5

Введение в расширения OpenGL

Во введении уже говорилось о том, что OpenGL "в чистом виде" обладает довольно ограниченными возможностями по сравнению, например, с DirectX. Так, например, вы не найдете в нем средств для работы с внеэкранными поверхностями, шейдерами и многого другого. Но зато с самого начала OpenGL обладал открытой архитектурой, которая заключается в поддержке расширений.

Если производитель обнаруживает, что OpenGL не может использовать все аппаратные возможности его "железа", он выпускает расширение для OpenGL с поддержкой новых возможностей. Причем это расширение, как правило, появляется почти на год раньше, чем новая версия DirectX, поддерживающая новые возможности. Так, поддержка пиксельных шейдеров для GeForce2 в OpenGL появилась намного раньше 8-й версии DirectX. А когда DirectX 8 все-таки вышел, то выяснилось, что про пиксельные шейдеры GeForce2 он ничего не знает.

Расширение OpenGL обычно определяется спецификацией, представляющей собой обычный текстовый файл. Как правило, спецификации расширений OpenGL пишутся для разработчиков реализаций (implementations) OpenGL, а не для начинающих программистов. Другими словами, спецификация OpenGL предоставляет достаточно информации для реализации расширения разработчиками видеокарт и драйверов к ним. Спецификация расширений OpenGL не является учебником по расширениям OpenGL, поэтому вы, скорее всего, не найдете в ней ответ на вопрос, где и как использовать новые возможности, предоставляемые данным расширением.

Спецификации всех расширений OpenGL вы сможете найти по адресу **http://oss.sgi.com/projects/ogl-sample/registry/**. Но этот способ поиска расширений не совсем удобен: в настоящий момент времени существует несколько сотен расширений OpenGL, при этом ни одна существующая видеокарта не поддерживает все существующие расширения. Поэтому ведущие производители графических процессоров (GPU) для видеокарт выпускают

собственные сборники спецификаций расширений OpenGL для своего оборудования. На CD-диске, поставляемом с книгой, вы сможете найти последние версии сборников спецификаций OpenGL корпораций ATI и NVIDIA, которые находятся в файлах atiopengl.pdf и nvOpenGLspecs.pdf соответственно (каталог \DOC\PDF).

На первых страницах этих документов приведены таблицы со сводной информацией о поддержке расширений различными GPU этих корпораций. Те же самые таблицы расширений, переведенные на русский язык, находятся в *Приложении 2*.

Просматривая названия расширений из этих таблиц, вы, скорее всего, заметите некоторую закономерность в их названиях. Во-первых, в названиях расширений вместо пробелов используются символы подчеркивания. Во-вторых, название каждого расширения состоит из двух частей: префикса и собственно названия расширения. Префикс расширения идентифицирует разработчика расширения. Ниже приведен список наиболее часто встречающихся префиксов:

- ARB — расширение официально одобрено наблюдательным советом по архитектуре OpenGL (ARB);
- ATI — расширение корпорации ATI;
- EXT — расширение поддерживается многими поставщиками OpenGL;
- HP — расширение корпорации Hewlett-Packard;
- IBM — расширение корпорации IBM;
- KTX — расширение компании Kinetix;
- INTEL — расширение корпорации Intel;
- NV — расширение корпорации NVIDIA;
- MESA — расширение MESA, кроссплатформенного клона OpenGL, разработанного Brian Paul;
- SGI — расширение корпорации Silicon Graphics;
- SGIS — расширение корпорации Silicon Graphics, одобренное другими поставщиками OpenGL;
- SGIX — экспериментальное расширение Silicon Graphics;
- WIN — расширение Microsoft;
- WGL — расширение, которое реализовано только в Windows. Расширение, скорее всего, не может быть реализовано в других операционных системах, т. к. использует особенности операционных систем Windows.

В Microsoft Windows все расширения OpenGL можно разделить на две большие группы: GL-расширения и WGL-расширения. GL-расширения являются кроссплатформенными расширениями, которые могут быть реализо-

ваны в любой операционной системе. WGL-расширения, напротив, привязаны к архитектуре операционной системы Windows. Например, вследствие явного использования дескрипторов окон и т. д. эти расширения, как правило, не могут быть перенесены на другую операционную систему.

Внимание

У WGL-расширений есть важное отличие от "обычных" GL-расширений, которое является следствием их тесной интеграции с Windows: для получения информации об ошибках WGL-расширений приложение должно использовать функцию `GetLastError` (функция Win32) вместо команды `glGetError`.

В других операционных системах также существуют свои непереносимые расширения: например, в Linux это GLX-расширения, а в Mac OS — AGL-расширения. Но т. к. эта книга посвящена использованию OpenGL в операционных системах семейства Win32, мы не будем рассматривать расширения, предназначенные для других операционных систем.

5.1. Как читать спецификацию расширения OpenGL (на примере расширения EXT_separate_specular_color)

Ранее говорилось о том, что спецификации расширений OpenGL предназначены в первую очередь для профессионалов. По этой причине они довольно сложны для понимания начинающих OpenGL-программистов.

В этом разделе мы рассмотрим структуру типичной спецификации расширения OpenGL на примере расширения EXT_separate_specular_color.

Перед тем как рассматривать любое расширение, надо понять, для чего оно нужно. Рассмотрим простой пример, выводящий на экран деревянный чайник с лаковым покрытием (а почему бы и нет — в компьютерной графике все возможно). Для этого мы выставляем в качестве цвета бликов (`GL_SPECULAR`) белый цвет, а параметру "глянцевитость" (`GL_SHINENESS`) присваиваем значение 128. Текстура дерева накладывается в режиме `GL_MODULATE`. Полный код примера приведен в листинге 5.1.

Листинг 5.1

```
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include "glh_glut_ext.h"
#include "nv_util_ext.h"
```

```
using namespace std;
using namespace glh;

int WinWidth=640;            // Ширина окна
int WinHeight=480;           // Высота окна

// Текстура дерева
tex_object_2D wood;
// Изображение с текстурой дерева
tga::tgaImage* pWoodImage;
// Дисплейный список чайника
display_list list0;

glut_console console;
glut_simple_user_interface user(&console);
glut_callbacks cb;
glut_swapbuffers swapbuffers;

const string Title="GLHE Demo";

void Init()    // Инициализация OpenGL
{
       glEnable(GL_DEPTH_TEST);
       glEnable(GL_LIGHTING);
       glEnable(GL_LIGHT0);

       glClearColor(0.5, 0.5, 0.75, 1);

       pWoodImage=read("..\\wood.tga");

       if (!pWoodImage)
       {
              console.add("Can't load file");
              console.exit(-1);
              return;
       }
// Создаем текстурный объект дерева
       wood.bind();
```

```
        wood.parameter(GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        wood.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        texImage2D(pWoodImage, &wood);
// Выгружаем изображение из памяти
        Clear(pWoodImage);

// Создание дисплейного списка чайника
        list0.new_list(GL_COMPILE);
        wood.enable();
                float DiffuseColor[4]={1.0, 0.0, 0.0, 1.0};
                float SpecularColor[4]={1.0, 1.0, 1.0, 1.0};
// Устанавливаем диффузный и рассеянный цвета материала
                glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
&DiffuseColor[0]);
// Устанавливаем зеркальный цвет материала
                glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,
&SpecularColor[0]);
// Устанавливаем рассеянный цвет материала
                glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 128);
                glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_MODULATE);
                glutSolidTeapot(2);
                wood.disable();
        list0.end_list();

}

void Display()                // Обновление содержимого экрана
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

//Вывод объекта на экран
        list0.call_list();

int main(int argc, char* argv[])
{
        console.add("Initializing GLUT...");
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
        glutInitWindowSize(WinWidth,WinHeight);
```

```
        glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
        (glutGet(GLUT_SCREEN_HEIGHT)-WinHeight)/2);
        glutCreateWindow(Title.c_str());

        console.add("Initializing OpenGL Helper Library...");
        glut_helpers_initialize();

        console.add("Initializing OpenGL...");
        Init();
// Если произошла ошибка при инициализации, выходим из
// программы
        if (console.exiting)
                glutMainLoop();

        console.title=Title;

        user.user_mouse.configure_buttons(1);
        user.user_mouse.dolly.dolly[2]=-9;

        cb.display_function=Display;

        glut_add_interactor(&user);
        glut_add_interactor(&cb);
        glut_add_interactor(&console);
        glut_add_interactor(&swapbuffers);

        console.processCmd("exec autoexec.cs");

        console.add("Initialize complete. Starting glutMainLoop");
        glutMainLoop();

        return 0;
}
```

Все было бы нормально, если бы не одно "но" — на чайнике нет никаких бликов (рис. 5.1).

Рис. 5.1. Чайник с лаковым покрытием, который не блестит

Это связано с тем, что и блики, и диффузный цвет материала имеют одинаковый цвет — белый. А OpenGL рассчитывает цвет объекта по формуле `(diffuse + specular) × tex`, где:

- `diffuse` — диффузный цвет объекта;
- `specular` — зеркальный цвет объекта;
- `tex` — текстура.

В результате белые блики рисуются на белом диффузном материале или, иными словами, блики просто сливаются с белым цветом материала.

Выход из данной ситуации один — рассчитывать цвет объекта по формуле `diffuse·tex + specular`. Но т. к. стандартный OpenGL не поддерживает расчет цвета объекта по этой формуле, нам придется рисовать чайник в два прохода: сначала нарисуем чайник с диффузной текстурой и выключенными бликами, а затем поверх полученного изображения наложим чайник, но с бликами и без текстуры (листинг 5.2).

Листинг 5.2

```
// Создание дисплейного списка объекта (чайника)
list0.new_list(GL_COMPILE);
// Первый проход
       wood.enable();
       float DiffuseColor[4]={1.0, 1.0, 1.0, 1.0};
       float SpecularColor[4]={1.0, 1.0, 1.0, 1.0};
       float Black[4]={0.0, 0.0, 0.0, 1.0};
       glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
&DiffuseColor[0]);
// Устанавливаем цвет бликов в черный
       glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, &Black[0]);


       glutSolidTeapot(2);


       wood.disable();


// Второй проход
// Выключаем запись в буфер глубины
       glDepthMask(false);
// Устанавливаем функцию Z-буфера в GL_LEQUAL
       glDepthFunc(GL_LEQUAL);
// Устанавливаем диффузный и рассеянный цвета материала в черный
       glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
&Black[0]);
// Настраиваем смешивание материалов
       glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, &SpecularColor[0]);
       glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 128);
// Включаем смешивание цветов
       glBlendFunc(GL_ONE, GL_ONE);
       glEnable(GL_BLEND);
// Рисуем чайник
       glutSolidTeapot(2);
// Восстанавливаем старые параметры
       glDisable(GL_BLEND);
       glDepthFunc(GL_LESS);
       glDepthMask(true);
list0.end_list();
```

Теперь блики выводятся корректно (рис. 5.2).

Рис. 5.2. Чайник с "правильным" лаковым покрытием

Все было бы нормально, если бы не то, что за эти красивые блики нам приходится расплачиваться значительным падением производительности: двухпроходные алгоритмы никогда не отличались высокой скоростью. В табл. 5.1 приведены результаты сравнения производительности примеров Ex01 и Ex02 на видеокартах ATI Radeon 9700 Pro и NVIDIA GeForce FX 5800 Ultra.

Таблица 5.1. *Сравнение производительности примеров Ex01 и Ex02*

Видеокарта	Ex01 (FPS)	Ex02 (FPS)
ATI Radeon 9700 Pro	326	168
NVIDIA GeForce FX 5800 Ultra	326	210

Как видно, падение производительности достигает почти 50%. Вот если бы OpenGL умел рассчитывать освещение по формуле `diffuse·tex + specular`.

Впрочем, все не так уж плохо. Все современные GPU с давних времен научились рассчитывать освещение по этой формуле. А для того чтобы программисты могли задействовать эту функцию, все производители GPU поддерживают расширение EXT_separate_specular_color, которое отвечает за поддержку этой функции.

Значит, нам надо переписать пример Ex01 с использованием этого расширения. Но перед тем как использовать это расширение, неплохо бы понять, как оно работает. А для этого надо прочитать спецификацию этого расширения.

Первый вопрос, который у нас возникает, — какие видеокарты поддерживают это расширение? Для этого изучаем таблицы, которые находятся в начале файлов atiopengl.pdf и nvOpenGLspecs.pdf (если у вас поблизости нет компьютера, то вы можете заглянуть в *Приложение 1*). После минутного разглядывания таблиц мы получаем следующую информацию. Расширение EXT_separate_specular_color поддерживается всеми GPU корпорации NVIDIA, а именно: RivaTNT, NV1x (GeForce 256, GeForce 2, GeForce 4 MX), NV2x (GeForce 3, GeForce 4) и NV3x (GeForce FX). Не поддерживается оно только семейством Riva 128, которое не выпускается уже около пяти лет. С GPU корпорации ATI ситуация обстоит аналогичным образом — расширение EXT_separate_specular_color поддерживается всеми современными видеокартами этой корпорации, такими как: Radeon 7000, 7200, 7500, 8500, 9000, 9500, 9700. Таким образом, использование расширения EXT_separate_specular_color никак не повлияет на спектр видеокарт, на которые ориентирована наша программа.

Теперь переходим непосредственно к спецификации расширения. Начнем с названия.

5.1.1. Раздел Name

Название расширения EXT_separate_specular_color начинается с префикса EXT, который означает, что данное расширение поддерживается множеством поставщиков OpenGL. Более того, в таблице расширений в начале файла nvOpenGLspecs.pdf говорится о том, что это расширение входит в стандарт OpenGL 1.2. Следовательно, оно поддерживается любой OpenGL 1.2-совместимой видеокартой независимо от производителя.

5.1.2. Раздел Name Strings

Переходим к следующему разделу спецификации расширения: Name Strings. Строка имени расширения указывает на строку, идентифицирующую данное расширение в OpenGL. Как правило, строка имени GL-расширения имеет вид GL_название_расширения, а WGL-расширения — GL_название_расширения.

В частности, строкой расширения EXT_separate_specular_color является строка GL_EXT_separate_specular_color.

5.1.3. Раздел Version

В разделе Version указывается дата последней модификации расширения и номер ревизии (версии) этого расширения. Наше расширение EXT_separate_specular_color имеет последнее обновление от 5 октября 1997 года. Обновление произошло в 00:16:23 по всемирному времени.

5.1.4. Раздел Number

Номер расширения. Каждое расширение OpenGL имеет свой уникальный номер, в частности, номер расширения EXT_separate_specular_color — 144.

5.1.5. Раздел Dependencies

Очень часто спецификация расширения OpenGL построена на базе спецификаций других расширений. Список таких расширений указывается в данном разделе. В этом случае для поддержки GPU данного расширения необходима поддержка и всех расширений из раздела Dependencies.

Наше расширение EXT_separate_specular_color — это "кот, который гуляет сам по себе" и не зависит от других расширений.

5.1.6. Раздел Overview

В этом разделе находится краткий обзор расширения. Прочитав этот раздел, вы узнаете, что расширение EXT_separate_specular_color добавляет в стандартную команду `glLightModel` новый параметр `GL_LIGHT_MODEL_COLOR_CONTROL_EXT`, который может принимать значения `GL_SINGLE_COLOR_EXT` и `GL_SEPARATE_SPECULAR_COLOR_EXT`. При этом значение `GL_SINGLE_COLOR_EXT` задает расчет освещения объекта по стандартной формуле OpenGL, а `GL_SEPARATE_SPECULAR_COLOR_EXT` — по формуле `diffuse·tex + specular`.

Примечание

В спецификации расширений OpenGL все команды и константы OpenGL приводятся без соответствующих префиксов `gl` и `GL_`. Таким образом, команда `LightModel` и константы `LIGHT_MODEL_COLOR_CONTROL_EXT`, `SINGLE_COLOR_EXT`, `SEPARATE_SPECULAR_COLOR_EXT` в действительности означают `glLightModel`, `GL_LIGHT_MODEL_COLOR_CONTROL_EXT`, `GL_SINGLE_COLOR_EXT` и `GL_SEPARATE_SPECULAR_COLOR_EXT`.

Команды и константы WGL-расширений, напротив, *обычно* приводятся с префиксами `wgl` и `WGL_`.

В дальнейшем во избежание путаницы в книге всегда будут использоваться полные имена команд и констант для любого типа расширения.

5.1.7. Раздел Issues

В этом разделе рассматриваются различные вопросы, которые могут возникнуть у разработчиков драйверов и GPU при реализации расширения. Как правило, вопросы связаны с тем, как должен вести себя GPU в нестандартной ситуации.

Вот пример одного вопроса по расширению EXT_separate_specular_color: "что должен делать GPU, когда включен режим расчета освещения `GL_SEPARATE_SPECULAR_COLOR_EXT` и выключено текстурирование?" Ответ: использовать режим `GL_SINGLE_COLOR_EXT`.

5.1.8. Раздел New Procedures and Functions

В этом разделе находится описание новых процедур и функций, которые добавляет расширение. Расширение EXT_separate_specular_color не содержит никаких новых функций.

5.1.9. Раздел New Token

В этом разделе находится список констант, которые добавляет расширение. У расширения EXT_separate_specular_color этот список выглядит следующим образом:

- `GL_LIGHT_MODEL_COLOR_CONTROL_EXT` 0x81F8
- `GL_SINGLE_COLOR_EXT` 0x81F9
- `GL_SEPARATE_SPECULAR_COLOR_EXT` 0x81FA

5.1.10. Группа разделов вида Additions to Chapter XX of the X.X Specification (XXX)

Добавление нового расширения OpenGL обычно требует внесения изменений в спецификацию OpenGL (и, возможно, в спецификации других библиотек). Казалось бы, мы можем написать большую единую спецификацию OpenGL, где будут учтены спецификации всех расширений. Но на практике это невозможно. Дело в том, что расширения разрабатываются различными корпорациями, и иногда два разных расширения не могут даже просто "ужиться" вместе.

Поэтому в спецификациях OpenGL используется другой подход. Вместо того чтобы модифицировать спецификацию расширения OpenGL, в спецификации расширения OpenGL описывается, *как надо* модифицировать спецификацию OpenGL, чтобы она учитывала все особенности этого расширения.

Чтобы понять смысл этих разделов, очень полезно иметь под рукой спецификацию всех версий OpenGL. Спецификации всех версий OpenGL можно найти на сайте **www.opengl.org**. Кроме того, на CD-диске, приложенном к книге, в каталоге \DOC\PDF находятся спецификации OpenGL 1.2.1, OpenGL 1.3 и OpenGL 1.4 в файлах OpenGL_spec_1.2.1.pdf, glspec13.pdf и glspec14.pdf.

В этой группе разделов обычно содержится наибольшее количество информации о расширении. Но в то же время это самые трудночитаемые разделы, т. к. они требуют совершенного знания спецификации OpenGL.

5.1.11. Раздел Errors

В этом разделе описывается реакция расширения на различные ошибки. Наше расширение EXT_separate_specular_color в принципе не может вызвать никаких ошибок, поэтому в спецификации расширения в этом разделе написано "none".

5.1.12. Раздел New State

Здесь приводятся новые переменные состояния OpenGL, которые добавляются расширением. Также в этом разделе указывается способ, с помощью которого можно получить значения этих переменных.

Расширение EXT_separate_specular_color добавляет в OpenGL одну новую переменную состояния `GL_LIGHT_MODEL_COLOR_CONTROL_EXT`, значение которой может быть получено при помощи команды `GetIntegerv`. Эта переменная может принимать два значения: `GL_SINGLE_COLOR_EXT`, когда используется стандартная модель освещения OpenGL, и `GL_SEPARATE_SPECULAR_COLOR_EXT`, когда используется раздельный расчет диффузной и зеркальной составляющих цвета.

5.2. Использование расширений OpenGL (на примере расширения EXT_separate_specular_color)

Теперь о расширении EXT_separate_specular_color мы знаем достаточно, чтобы переписать пример Ex01 с его использованием (Ex03).

Но мы не можем просто так взять и начать использовать это расширение, т. к. оно использует несколько новых констант *(см. разд. 5.1.9)*, которых нет в стандартном файле Microsoft gl.h. Поэтому у нас есть два варианта.

1. Вставить объявление этих констант прямо в текст программы.
2. Использовать комплект заголовочных файлов для драйверов корпораций NVIDIA и ATI.

Второй вариант в большинстве случаев является более предпочтительным. Заголовочные файлы для драйверов NVIDIA входят в комплект NVIDIA OpenGL SDK и находятся в каталоге \NVSDK\OpenGL\include\glh\GL.

- glext.h — кроссплатформенные расширения OpenGL;
- wglext.h — расширения OpenGL для операционных систем семейства Windows;
- glxext.h — расширения OpenGL для Linux;
- glut.h — библиотека GLUT, которая была подробно рассмотрена в первой главе.

Заголовочные файлы для драйверов корпорации ATI находятся на CD-диске, приложенном к книге, в каталоге \ATI SDK\Include:

- glATI.h — кроссплатформенные расширения OpenGL;
- wglATI.h — расширения OpenGL для операционных систем семейства Windows.

Если вы используете библиотеку GLH_GLUT_EXT, то вам не о чем беспокоиться, т. к. файл glh_glut_ext.h автоматически подключает к программе заголовочные файлы драйверов корпораций ATI и NVIDIA:

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glext.h>
#include <gl/wglext.h>
#include <gl/glATI.h>
#include <gl/wglATI.h>
```

Далее мы должны проверить, поддерживает ли расширение EXT_separate_specular_color компьютер, на котором работает наше приложение.

Для этого мы должны получить список поддерживаемых расширений при помощи стандартной команды `glGetString` с параметром `GL_EXTENSIONS`. Эта команда возвращает ASCIIZ-строку, в которой находятся имена (точнее name string) всех поддерживаемых кроссплатформенных расширений, разделенных символом-разделителем (обычно это пробел). Для того чтобы проверить поддержку расширения EXT_separate_specular_color, достаточно найти в строке, возвращенной командой `glGetString(GL_EXTENSIONS)`, подстроку

GL_EXT_separate_specular_color. Положительный результат означает то, что расширение поддерживается данным компьютером.

Далее приведен фрагмент кода функции Init из примера Ex03, который отвечает за проверку поддержки расширения EXT_separate_specular_color:

```
// Получаем строку со списком кроссплатформенных расширений
const GLubyte* extensions=glGetString(GL_EXTENSIONS);
// Ищем в этой строке расширение EXT_separate_specular_color
if (!strstr((char*) extensions, "GL_EXT_separate_specular_color"))
{
// Если строка не найдена, выводим сообщение об ошибке
       console.add("Unsupported extension:
EXT_separate_specular_color");
       console.exit(-1);
       return;
}
```

После проверки поддержки расширения EXT_separate_specular_color, мы можем начинать использовать его в своей программе. Для того чтобы включить расчет освещения по "супер-формуле", нам надо добавить в функцию Init следующую команду (раздел 5.1.6):

```
glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL_EXT,
GL_SEPARATE_SPECULAR_COLOR_EXT);
```

И это все! Теперь блики на чайнике выглядят, как в примере Ex02. Но поскольку рисование чайника осуществляется за один проход, скорость работы примера Ex03 не отличается от скорости Ex01 (см. табл. 5.2). Самое интересное, что на NVIDIA GeForce FX 5800 Ultra пример Ex03 работает даже немного быстрее примера Ex01.

Таблица 5.2. *Сравнение производительности примеров Ex01 и Ex02.*

Видеокарта	Ex01 (FPS)	Ex03 (FPS)
ATI Radeon 9700 Pro	326	315
NVIDIA GeForce FX 5800 Ultra	326	352

5.3. Инициализация расширений OpenGL, добавляющих в OpenGL новые команды (на примере расширения ARB_window_pos)

В *главе 1*, мы рассматривали вывод растровых шрифтов на экран (пример Ex07 *главы 1*). Главная проблема, с которой мы столкнулись при этом, за-

ключалась в том, что OpenGL умножал координаты позиции вывода на матрицы модели и проекции. Из-за этого нам пришлось корректировать матрицы модели и проекции, а старые матрицы сохранять в стеке матриц OpenGL. Недостаток данного метода заключается в том, что в большинстве реализаций OpenGL стек текстурных матриц может хранить не более двух матриц. Следовательно, сохраняя текстурную матрицу в стеке, мы увеличиваем риск переполнения стека матриц.

Библиотека MESA (клон OpenGL) с давних времен имеет расширение MESA_window_pos, позволяющее задавать координаты вывода растровых изображений в оконных координатах (команды `glWindowPos4fMESA(x,y,z,w)` и `glWindowPos2fMESA(x,y)`).

Но в OpenGL аналогичное расширение появилось только в 2002. Это расширение ARB_window_pos, которое сразу вошло в стандарт OpenGL 1.4. Расширение ARB_window_pos позволяет задать позицию вывода растрового изображения в оконных координатах. Для этой цели расширение ARB_window_pos предоставляет программисту целых шестнадцать новых команд, которые позволяют задать координаты вывода растрового изображения всеми возможными способами:

1. `void WindowPos2dARB(double x, double y);`
2. `void WindowPos2fARB(float x, float y);`
3. `void WindowPos2iARB(int x, int y);`
4. `void WindowPos2sARB(short x, short y);`
5. `void WindowPos2dvARB(const double *p);`
6. `void WindowPos2fvARB(const float *p);`
7. `void WindowPos2ivARB(const int *p);`
8. `void WindowPos2svARB(const short *p);`
9. `void WindowPos3dARB(double x, double y, double z);`
10. `void WindowPos3fARB(float x, float y, float z);`
11. `void WindowPos3iARB(int x, int y, int z);`
12. `void WindowPos3sARB(short x, short y, short z);`
13. `void WindowPos3dvARB(const double *p);`
14. `void WindowPos3fvARB(const float *p);`
15. `void WindowPos3ivARB(const int *p);`
16. `void WindowPos3svARB(const short *p).`

Из всего этого списка на практике чаще всего используются двухмерные функции `WindowPos2fARB` и `WindowPos2fvARB`. Трехмерные функции, позво-

ляющие задать значение глубины в данной точке, применяются значительно реже.

Координаты, задаваемые функцией `glRasterPos`, проходят тест отсечения. В результате, иногда возможна ситуация, когда битовая матрица не будет выведена на экран, несмотря на то, что ее часть видна на экране. А вот координаты, задаваемые функцией `glWindowPos`, не проходят тест отсечения. В результате функция `glWindowPos` свободна от таких побочных эффектов.

Для того чтобы окончательно разобраться с расширением ARB_window_pos, мы перепишем пример Ex07 *главы 1* с использованием этого расширения (Ex04).

Для начала мы должны вставить в функцию `Init` проверку поддержки расширения ARB_window_pos:

```
const GLubyte* extensions=glGetString(GL_EXTENSIONS);
if (!strstr((char*) extensions, "GL_ARB_window_pos"))
{
       MessageBox(0, "Unsupported extension: ARB_window_pos", "Error",
MB_ICONERROR | MB_OK);
       exit(-1);
}
```

Теперь нам надо внести изменения в функцию `display`, выводящую изображение на экран (листинг 5.3).

Листинг 5.3

```
void Display()                // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glPushMatrix();
// Перемещение и поворот чайника
               glTranslatef(tx,ty,tz);             /
               glRotatef(rx,1,0,0);
               glRotatef(ry,0,1,0);

//Вывод чайника на экран
               glCallList(list);
       glPopMatrix();
// Выключаем тест глубины (чтобы текст не перекрывался другими объектами)
       glDisable(GL_DEPTH_TEST);
```

```
        glColor3f(1,1,1);

        //Установка позиции для вывода подписи
        glWindowPos2fARB((WinWidth-
StringWidth(GLUT_BITMAP_TIMES_ROMAN_24, "Teapot"))/2, WinHeight/7);
        //Вывод подписи
        PrintString(GLUT_BITMAP_TIMES_ROMAN_24, "Teapot");

        if (bTimer)
        {
                FramesCount++;
                char Title[80];
                if (GetTickCount()!=StartTick)
                {
                        //Расчет FPS
                        sprintf(Title, "%.1f",
float(FramesCount)*1000/(GetTickCount()-StartTick));

                        // Вывод FPS
                        glWindowPos2fARB(WinWidth-
StringWidth(GLUT_BITMAP_TIMES_ROMAN_24, &Title[0])-10, WinHeight-30);
                        PrintString(GLUT_BITMAP_TIMES_ROMAN_24, &Title[0]);
                }
        }

        glEnable(GL_DEPTH_TEST);

        glutSwapBuffers();
}
```

Как видно, все изменения свелись к тому, что из функции `Display` были убраны функции, отвечающие за настройку матриц модели и проекции. В результате программа стала более логичной и понятной.

Но у полученной программы есть один большой недостаток — она не работает. Если вы попробуете откомпилировать ее, то компилятор выдаст сообщение о том, что команда `glWindowPos2fARB` не найдена. Для того чтобы понять причину такого поведения программы, мы должны взглянуть на исходный код файла glATI.h, в котором находится определение этого расширения:

```
/*
** GL_ARB_window_pos
```

```
**
**  Support:
**   Rage 128         based : Supported
**   Radeon 7xxx      based : Supported
**   Radeon 8xxx/9000 based : Supported
**   Radeon 9500/9700 based : Supported
*/
#ifndef GL_ARB_window_pos
#define GL_ARB_window_pos   1

typedef void (APIENTRY * PFNGLWINDOWPOS2DARBPROC) (double x, double y);
typedef void (APIENTRY * PFNGLWINDOWPOS2FARBPROC) (float x, float y);
typedef void (APIENTRY * PFNGLWINDOWPOS2IARBPROC) (int x, int y);
typedef void (APIENTRY * PFNGLWINDOWPOS2SARBPROC) (short x, short y);
typedef void (APIENTRY * PFNGLWINDOWPOS2IVARBPROC) (const int *p);
typedef void (APIENTRY * PFNGLWINDOWPOS2SVARBPROC) (const short *p);
typedef void (APIENTRY * PFNGLWINDOWPOS2FVARBPROC) (const float *p);
typedef void (APIENTRY * PFNGLWINDOWPOS2DVARBPROC) (const double *p);
typedef void (APIENTRY * PFNGLWINDOWPOS3IARBPROC) (int x, int y, int z);
typedef void (APIENTRY * PFNGLWINDOWPOS3SARBPROC) (short x, short y,
short z);
typedef void (APIENTRY * PFNGLWINDOWPOS3FARBPROC) (float x, float y,
float z);
typedef void (APIENTRY * PFNGLWINDOWPOS3DARBPROC) (double x, double y,
double z);
typedef void (APIENTRY * PFNGLWINDOWPOS3IVARBPROC) (const int *p);
typedef void (APIENTRY * PFNGLWINDOWPOS3SVARBPROC) (const short *p);
typedef void (APIENTRY * PFNGLWINDOWPOS3FVARBPROC) (const float *p);
typedef void (APIENTRY * PFNGLWINDOWPOS3DVARBPROC) (const double *p);

#endif /* GL_ARB_window_pos */
```

Итак, что мы видим? В файле glATI.h находятся только прототипы новых команд расширения ARB_window_pos. Сами команды не объявляются. Почему же корпорация ATI "поленилась" сделать "нормальный" заголовочный файл для своих драйверов, который автоматически производил бы статическую компоновку с динамической DLL-библиотекой драйвера (как это делает файл от Microsoft — gl.h)?

На это есть очень веская причина. Дело в том, что если в процессе статической компоновки Windows обнаруживает, что в загружаемой динамической

библиотеке отсутствуют некоторые функции, то прерывает процесс загрузки приложения. В результате приложение, использующее файл glATI.h, смогло бы запуститься только на компьютерах с видеокартами корпорации ATI, т. к. только GPU корпорации поддерживают *все* расширения корпорации ATI.

Поэтому в операционных системах Windows используется другой механизм инициализации расширений. Для получения адреса точки входа команды OpenGL приложение должно использовать функцию `wglGetProcAddress`. Таким образом, пользовательская программа должна самостоятельно объявить указатели на все шестнадцать функций OpenGL, а затем, в драйвере, присвоить им адреса точек входа в эти функции при помощи команды `wglGetProcAddress`.

Технические подробности

В действительности все немного сложнее. При запуске прикладная программа пытается загрузить программную реализацию OpenGL корпорации Microsoft, расположенную в файле opengl32.dll, которая в свою очередь пытается найти и загрузить Installable Client Driver (ICD). ICD выбирается в зависимости от загруженного драйвера дисплея: например, если установлен драйвер корпорации ATI, то ICD будет загружен из файла atioglxx.dll, а если NVIDIA — то nvoglint.dll. Если ICD загружен нормально, то opengl32.dll передает управление на соответствующие точки входа в функции ICD. Если же ICD не может быть удачно загружен, то используется программная реализация OpenGL корпорации Microsoft (не отличающаяся, кстати, хорошей производительностью).

Следствием из всего вышесказанного является невозможность статической компоновки приложения с ICD, т. к. имя DLL-библиотеки ICD заранее не известно.

В нашем случае мы можем немного упростить задачу, если учтем, что наша программа использует только одну функцию ARB_window_pos — `glWindowPos2fARB`. Поэтому мы можем обойтись получением адреса входа всего одной функции (листинг 5.4).

Листинг 5.4

```
// Объявляем указатель на новую команду OpenGL
PFNGLWINDOWPOS2FARBPROC glWindowPos2fARB;
void Init()    // Инициализация OpenGL
{
...
// Получаем указатель на команду OpenGL
       glWindowPos2fARB=(PFNGLWINDOWPOS2FARBPROC)
wglGetProcAddress("glWindowPos2fARB");
// Если произошла ошибка (wglGetProcAddress вернул нулевой указатель),
// то выводим сообщение об ошибке
```

```
        if (!glWindowPos2fARB)
        {
                MessageBox(0, "Unsupported OpenGL command:
glWindowPos2fARB", "Error", MB_ICONERROR | MB_OK);
                exit(-1);
        }
...
```

Теперь программа компилируется и работает нормально.

Вполне возможно, что такой способ инициализации расширений OpenGL, использующих новые функции, покажется вам, мягко говоря, немного неудобным. К сожалению, это единственный способ инициализации расширений стандартными средствами OpenGL.

5.4. Использование WGL-расширений (на примере расширения WGL_EXT_swap_control)

Базовая версия OpenGL не имеет каких-либо встроенных средств управления вертикальной синхронизацией. В то же время эта возможность бывает очень полезна в реальных программах. Пользователи достаточно мощных компьютеров обычно работают с включенной кадровой синхронизацией, в то время как пользователи слабых систем, скорее всего, захотят отключить кадровую синхронизацию. Кроме того, когда программа измеряет производительность компьютера, то кадровую синхронизацию надо обязательно выключать. В противном случае на мощных компьютерах максимальный FPS будет приблизительно равен частоте регенерации изображения на мониторе.

Для того чтобы предоставить программисту возможность управления кадровой синхронизацией, было разработано расширение WGL_EXT_swap_control.

В состав расширения входят две функции: `wglSwapIntervalEXT` и `wglGetSwapInterval`:

```
BOOL wglSwapIntervalEXT(int interval)
int wglGetSwapIntervalEXT(void)
```

Функция `wglSwapIntervalEXT` устанавливает максимальную частоту смены кадров. В качестве параметра данная функция принимает число, показывающее, какое число кадров будет выведено перед тем, как произойдет переключение буферов командой `glutSwapBuffers` (или аналогичной командой). Например, команда `wglSwapIntervalEXT(4)` означает, что буфер будет переключаться на каждом четвертом кадре, *показываемом монитором*, т. е.

при вертикальной развертке монитора 85 Гц максимальный FPS не превысит 85 / 4 = 21 кадра в секунду. На практике для включения вертикальной синхронизации используется значение 1. Выключение вертикальной синхронизации происходит путем передачи функции значения 0. Для получения текущих параметров синхронизации используется функция `wglGetSwapInterval`.

Расширение WGL_EXT_swap_control является WGL-расширением, т. е. относится к непереносимым расширениям. Из-за этого расширение WGL_EXT_swap_control *может* отсутствовать в строке расширений, возвращаемой командой `glGetString(GL_EXTENSIONS)`.

Для получения списка WGL-расширений в операционных системах семейства Windows используется расширение WGL_EXT_extensions_string. Это расширение — своеобразное исключение из правил. Расширение добавляет в OpenGL одну новую команду — `wglGetExtensionsStringEXT`, которая возвращает ASCIIZ-строку со списком расширений, разделенных пробелами.

```
const char *wglGetExtensionsStringEXT(void);
```

Так как расширение WGL_EXT_extensions_string также является WGL-расширением, оно может отсутствовать в списке расширений `glGetString(GL_EXTENSIONS)`. В результате единственно надежный способ проверить поддержку этого расширения —попытаться получить адрес точки входа в функцию `wglGetExtensionsStringEXT`. Если адрес точки входа окажется ненулевым, то расширение WGL_EXT_extensions_string поддерживается данным компьютером.

Поэтому для того, чтобы проверить поддержку расширения WGL_EXT_swap_control, приложение должно выполнить следующие операции.

1. Инициализировать расширение WGL_EXT_extensions_string. Для этого программа должна получить адрес функции `wglGetExtensionsStringEXT` при помощи команды `wglGetProcAddress`. Если адрес функции не удалось получить, то это означает, что расширение WGL_EXT_extensions_string не поддерживается.
2. Получить строку WGL-расширений при помощи команды `wglGetProcAddress` и проверить поддержку расширения WGL_EXT_extensions_string.

Для того чтобы продемонстрировать использование расширения WGL_EXT_swap_control на практике, я добавил в пример Ex04 управление кадровой синхронизаций при помощи клавиши F3 (Ex05). Текущее состояние кадровой синхронизации отображается в правом верхнем углу экрана.

Наиболее важные фрагменты кода получившейся программы приведены в листинге 5.5.

Листинг 5.5

```
PFNGLWINDOWPOS2FARBPROC glWindowPos2fARB;
PFNWWGLGETEXTENSIONSSTRINGEXTPROC wglGetExtensionsStringEXT;
PFNWGLSWAPINTERVALEXTPROC wglSwapIntervalEXT;

void Init()    // Инициализация OpenGL
{
// Проверяем поддержку расширения ARB_window_pos
       const GLubyte* extensions=glGetString(GL_EXTENSIONS);
       if (!strstr((char*) extensions, "GL_ARB_window_pos"))
       {
              MessageBox(0, "Unsupported extension: ARB_window_pos",
"Error", MB_ICONERROR | MB_OK);
              exit(-1);
       }
// Получаем адрес команды glWindowPos2fARB
       glWindowPos2fARB=(PFNGLWINDOWPOS2FARBPROC) wglGetProcAddress
( "glWindowPos2fARB" );
       if (!glWindowPos2fARB)
       {
              MessageBox(0, "Unsupported OpenGL command:
glWindowPos2fARB", "Error", MB_ICONERROR | MB_OK);
              exit(-1);
       }
// Проверяем поддержку расширения WGL_EXT_extensions_string
       if (!strstr((char*) extensions, "WGL_EXT_extensions_string"))
       {
              MessageBox(0, "Unsupported extension:
WGL_EXT_extensions_string", "Error", MB_ICONERROR | MB_OK);
              exit(-1);
       }
// Получаем адрес команды wglGetExtensionsStringEXT
       wglGetExtensionsStringEXT=(PFNWGLGETEXTENSIONSSTRINGEXTPROC)
wglGetProcAddress("wglGetExtensionsStringEXT");
       if (!glWindowPos2fARB)
       {
              MessageBox(0, "Unsupported OpenGL command:
wglGetExtensionsStringEXT", "Error", MB_ICONERROR | MB_OK);
              exit(-1);
       }
```

```
// Проверяем поддержку расширения WGL_EXT_swap_control
       const char* wgl_extensions=wglGetExtensionsStringEXT();
       if (!strstr((char*) extensions, "WGL_EXT_swap_control"))
       {
              MessageBox(0, "Unsupported extension:
WGL_EXT_swap_control", "Error", MB_ICONERROR | MB_OK);
              exit(-1);
       }
// Получаем адрес команды PFNWGLSWAPINTERVALEXTPROC
       wglSwapIntervalEXT=(PFNWGLSWAPINTERVALEXTPROC) wglGetProcAd-
dress("wglSwapIntervalEXT");
       if (!wglSwapIntervalEXT)
       {
              MessageBox(0, "Unsupported OpenGL command: wglSwapInter-
valEXT", "Error", MB_ICONERROR | MB_OK);
              exit(-1);
       }

       glEnable(GL_DEPTH_TEST);
       glEnable(GL_LIGHTING);
       glEnable(GL_LIGHT0);
       glEnable(GL_COLOR_MATERIAL);

       glClearColor(0.5, 0.5, 0.75, 1);

       list=glGenLists(1);

       glNewList(list, GL_COMPILE);       // Создание дисплейного
                                          // списка объекта (чайника)
              glColor3f(0.1,0.7,0.2);
              glutSolidTeapot(2);
       glEndList();
// Включаем кадровую синхронизацию
       wglSwapIntervalEXT(1);
       bFrameSynch=1;
}
...
void Display()               // Обновление содержимого экрана
{
...
```

```
// Выводим текущее состояние кадровой синхронизации
        glWindowPos2fARB(30, WinHeight-30);
        if (bFrameSynch)
                PrintString(GLUT_BITMAP_TIMES_ROMAN_24, "Frame synch on");
        else
                PrintString(GLUT_BITMAP_TIMES_ROMAN_24, "Frame synch
off");
...
}
void KeyboardSpecial(int key, int x, int y)
{
        switch (key)
        {
...
// Включаем/выключаем кадровую синхронизацию
                case GLUT_KEY_F3:
                        bFrameSynch=!bFrameSynch;
                        wglSwapIntervalEXT(bFrameSynch);
                        glutPostRedisplay();
                        break;
        }
}
```

Данная программа позволяет управлять текущим состоянием кадровой синхронизации при помощи клавиши <F3>. При этом в верхнем левом углу экрана отображается текущее состояние кадровой синхронизации.

Хотя пример Ex05 и работает более-менее нормально, у него есть один небольшой недостаток. Дело в том, что программа делает предположение, что функция `wglSwapIntervalEXT` всегда изменяет режим кадровой синхронизации. К сожалению, это не всегда так — большинство драйверов современных видеокарт позволяют принудительно установить режим кадровой синхронизации в одно фиксированное значение (например, Always On или Always Off). В этом случае драйвер будет всегда игнорировать вызовы функции `wglSwapIntervalEXT`. В результате состояние кадровой синхронизации может отображаться неправильно.

Следовательно, программа должна проверять текущее состояние кадровой синхронизации при помощи команды `wglGetSwapIntervalEXT`. По идее эта функция должна возвращать число кадров, которые показываются перед тем, как произойдет переключение буферов командой `glutSwapBuffers` (или аналогичной). Если же в драйверах видеокарты установлен режим кадровой

синхронизации (Always Off), то видеокарты ATI и NVIDIA ведут себя по-разному.

Драйверы ATI просто скрывают от прикладной программы факт, что они взяли на себя управление кадровой синхронизацией. Поэтому функция `wglGetSwapIntervalEXT` всегда возвращает параметр предыдущей команды `wglSwapIntervalEXT`.

Драйверы NVIDIA ведут себя по-другому: в этом случае функция `wglGetSwapIntervalEXT` всегда возвращает −1.

В листинге 5.6 приведены фрагменты кода исправленной программы, которая более корректно отображает состояние кадровой синхронизации на видеокартах NVIDIA (Ex06).

Листинг 5.6

```
// Текущее состояние кадровой синхронизации
int bFrameSynch;

PFNGLWINDOWPOS2FARBPROC glWindowPos2fARB;
PFNWGLGETEXTENSIONSSTRINGEXTPROC wglGetExtensionsStringEXT;
PFNWGLSWAPINTERVALEXTPROC wglSwapIntervalEXT;
PFNWGLGETSWAPINTERVALEXTPROC wglGetSwapIntervalEXT;

void Init()                          // Инициализация OpenGL
{
...
// Получаем адрес команды wglGetSwapIntervalEXT
       wglGetSwapIntervalEXT=(PFNWGLGETSWAPINTERVALEXTPROC)
wglGetProcAddress("wglGetSwapIntervalEXT");
       if (!wglGetSwapIntervalEXT)
       {
              MessageBox(0, "Unsupported OpenGL command:
wglGetSwapIntervalEXT", "Error", MB_ICONERROR | MB_OK);
              exit(-1);
       }
...
// Получаем текущее состояние кадровой синхронизации
       bFrameSynch=wglGetSwapIntervalEXT();
}
...
```

```
void KeyboardSpecial(int key, int x, int y)
{
        switch (key)
        {
...
                case GLUT_KEY_F3:
                        bFrameSynch=!bFrameSynch;
                        wglSwapIntervalEXT(bFrameSynch);
// Получаем текущее состояние кадровой синхронизации
                        bFrameSynch=wglGetSwapIntervalEXT();
                        glutPostRedisplay();
                        break;
        }
}
void Display()                // Обновление содержимого экрана
{
...
// Вывод текущего состояния кадровой синхронизации
        switch (bFrameSynch)
        {
        case 0:
                PrintString(GLUT_BITMAP_TIMES_ROMAN_24, "Frame synch off");
                break;
        case 1:
                PrintString(GLUT_BITMAP_TIMES_ROMAN_24, "Frame synch on");
                break;
        case -1:
                PrintString(GLUT_BITMAP_TIMES_ROMAN_24, "Frame synch al-
ways off");
                break;
        default:
                PrintString(GLUT_BITMAP_TIMES_ROMAN_24, "Unusual frame
synch");
                break;
        }
...
}
```

5.5. Инициализация расширений с использованием библиотеки NVIDIA OpenGL Helper Library

На примерах предыдущего раздела становится ясно, что инициализировать GL- и WGL-расширения непосредственно через OpenGL API довольно неудобно, особенно если надо подключить больше десятка различных расширений, каждое из которых содержит больше десятка функций. Поэтому большинство фирм — производителей современных GPU имеют собственные высокоуровневые средства для инициализации расширений OpenGL.

В состав библиотеки OpenGL Helper Library (GLH) входит набор функций, предназначенных для инициализации OpenGL. Определения этих функций расположены в файле \NVSDK\OpenGL\include\glh\glh_extensions.h. Из всего множества функций этого файла нам интересны только две функции:

```
int glh_init_extensions(const char *origReqExts)
const char* glh_get_unsupported_extensions()
```

Функция `glh_init_extensions` проверяет поддержку текущей реализацией OpenGL списка расширений (параметр `origReqExts`) и инициализирует адреса точек входа в функции этих расширений. Список расширений задается ASCIIZ-строкой, в которой названия (Name Strings) расширений разделены пробелом. Если инициализация прошла удачно, функция `glh_init_extensions` возвращает `GL_TRUE`, в противном случае — `GL_FALSE`. В последнем случае приложение может получить список расширений, которые не удалось инициализировать, при помощи функции `glh_get_unsupported_extensions`.

У файла glh_extensions.h имеется одна интересная особенность, которая часто ставит в тупик многих программистов, впервые сталкивающихся с NVIDIA OpenGL SDK. Дело в том, что в C++ все библиотеки состоят из двух файлов: заголовочного файла с объявлениями функций (.h) и собственного файла с исходным кодом библиотеки (.cpp). Но, с другой стороны, такая структура немного затрудняет подключение библиотеки к программе, т. к. проекту необходимо "вручную" подключать CPP-файл библиотеки.

Файл glh_extensions.h использует другой подход. Он содержит внутри себя и заголовочный код, и реализацию библиотеки. А для борьбы с появлением множества экземпляров одной и той же функции в многомодульном проекте он использует интересный прием. Файл glh_extensions.h имеет примерно следующую структуру:

```
#ifdef GLH_EXT_SINGLE_FILE
// Реализация библиотеки
#else
```

```
// Определение функций библиотеки (без реализации)
#endif
```

Таким образом, если в пользовательском файле определен макрос GLH_EXT_SINGLE_FILE, то директива #include glh_extensions.h включает реализацию библиотеки, а в противном случае — только определение функций библиотеки без реализации. Следовательно, если проект содержит несколько модулей, использующих заголовочный файл glh_extensions.h, то один из файлов должен содержать макрос GLH_EXT_SINGLE_FILE.

Для демонстрации использования средств библиотеки GLH для инициализации расширений OpenGL на практике, я переписал пример Ex06 с использованием этой библиотеки (Ex07):

```
...
#include <gl/glut.h>
#include <gl/glext.h>
#include <gl/wglext.h>
#include <gl/glATI.h>
#include <gl/wglATI.h>
// Подключение файла glh_extensions.h
#define GLH_EXT_SINGLE_FILE
#include <glh/glh_extensions.h>
...
void Init()                          // Инициализация OpenGL
{
       if (!glh_init_extensions("GL_ARB_window_pos
WGL_EXT_swap_control"))
       {
// Если не получилось "проинициализировать" все расширения, то выводим
// сообщение об ошибке (и перечень неподдерживаемых расширений)
MessageBox(0, (string("Necessary extensions are not
supported:")+glh_get_unsupported_extensions()).c_str(), "Error", MB_OK);
              exit(-1);
       }
...
}
```

Однако, если вы попробуете откомпилировать эту программу, то вас ожидает неприятный сюрприз — компилятор выдаст сообщение о том, что не знает функций wglSwapIntervalEXT, glWindowPos2fARB и т. д. Дело в том, что версия библиотеки GLH, входящая в состав NVIDIA OpenGL SDK, поддерживает довольно небольшое количество расширений. В частности, библиотека

GLH не умеет инициализировать расширения ARB_window_pos и WGL_EXT_swap_control. К счастью, ситуация не является безвыходной. Дело в том, что в библиотеке GLH предусмотрен механизм, позволяющий программисту легко добавить в библиотеку GLH поддержку новых расширений.

Для этого пользователь должен открыть в любом текстовом редакторе файл \NVSDK\OpenGL\include\glh\extgen\extfile.txt и вставить в него текст следующего вида:

```
Строка_названия_расширения  {
Команда_1
Команда_2
...
Команда_n
}
```

В частности, для того, чтобы добавить в библиотеку GLH поддержку расширений ARB_window_pos и WGL_EXT_swap_control, программист должен вставить в файл extfile.txt следующий текст:

```
WGL_EXT_swap_control {
wglSwapIntervalEXT
wglGetSwapIntervalEXT
}
GL_ARB_window_pos {
glWindowPos2dARB
glWindowPos2fARB
glWindowPos2iARB
glWindowPos2sARB
glWindowPos2dvARB
glWindowPos2fvARB
glWindowPos2ivARB
glWindowPos2svARB
glWindowPos3dARB
glWindowPos3fARB
glWindowPos3iARB
glWindowPos3sARB
glWindowPos3dvARB
glWindowPos3fvARB
glWindowPos3ivARB
glWindowPos3svARB
}
```

Теперь необходимо открыть проект, находящийся в файле \NVSDK\OpenGL\include\glh\extgen\extgen.dsw и запустить его на выполнение. Полученная программа создаст на основе файла extfile.txt новый заголовочный файл \NVSDK\OpenGL\include\glh\glh_genext.h, содержащий необходимый код для нормальной инициализации расширений OpenGL, включая ARB_window_pos и WGL_EXT_swap_control (файл glh_extensions.h подключает заголовочный файл glh_genext.h директивой #include).

Попробуйте еще раз откомпилировать программу. На этот раз компиляция пройдет успешно, и программа запустится на выполнение.

Как видно, библиотека GLH значительно упрощает инициализацию расширений OpenGL. Единственный недостаток этой библиотеки — поддержка довольно ограниченного числа расширений с лихвой компенсируется простым добавлением в эту библиотеку поддержки новых расширений OpenGL, без необходимости "ручного" исправления исходного кода.

Если же вам неохота возиться с добавлением поддержки новых расширений в библиотеку GLH, вы можете воспользоваться новыми версиями файлов extfile.txt и glh_genext.h с поддержкой всех расширений, описанных в этой книге (вы можете найти их на CD-диске, поставляемом с книгой, в каталоге \NVIDIA SDK\Update). Кроме того, содержимое доработанного файла extfile.txt находится в *Приложении 3*.

5.6. Инициализация расширений при помощи библиотеки ATI Extensions

Корпорация ATI также имеет свой инструментарий для быстрой инициализации расширений OpenGL — библиотеку ATI Extensions, предоставляющую программисту набор функций для быстрой инициализации OpenGL.

Вы можете найти библиотеку ATI Extensions на сайте **www.ati.com** или на CD-диске, поставляемом с книгой (файлы \ATI SDK\Include\gl\ATIExtensions.h и \ATI SDK\Lib\ATIExtensions.lib). Исходный текст файла ATIExtensions.lib находится в каталоге \ATI SDK\Src.

Для того чтобы подключить библиотеку к проекту, необходимо добавить в исходный текст программы директиву `#include gl/ATIExtensions.h`, а также пополнить список используемых библиотек файлом ATIExtensions.lib.

В отличие от библиотеки NVIDIA GLH, библиотека ATI Extensions предоставляет программисту довольно внушительный набор функций для инициализации расширений:

```
// Инициализация всех расширений, входящих в стандарт OpenGL 1.2
// (не путать с OpenGL 1.2.1!)
int SetupGL1_2();
```

```
// Инициализация всех расширений, входящих в стандарт OpenGL 1.3
int SetupGL1_3();
// Инициализация всех расширений, входящих в стандарт OpenGL 1.4
int SetupGL1_4();
// Инициализация всех WGL-расширений, поддерживаемых видеокартами ATI
int SetupWGLExtentions();
// Инициализация всех ARB-расширений, поддерживаемых видеокартами ATI
int SetupARBExtensions();
// Инициализация всех EXT-расширений, поддерживаемых видеокартами ATI
int SetupEXTExtensions();
// Инициализация всех ATI-расширений
int SetupATIExtensions();
// Инициализация списка расширений, заданных ASCIIZ-строкой
// (аналог функции glh_init_extensions библиотеки NVIDIA GLH)
int SetupExtensionsFromString(const char *);
```

Если любой из вышеуказанных функций удается инициализировать все расширения, то она возвращает `true`, в противном случае — `false`. К сожалению, в библиотеке ATI Extensions не предусмотрена возможность получения информации о том, какие именно расширения не удалось инициализировать.

Из всех вышеуказанных функций в реальных программах используется только функция `SetupExtensionsFromString`, позволяющая проинициализировать заданный список расширений. Остальными функциями надо пользоваться очень осторожно, т. к. они имеют довольно много подводных камней. К примеру, у вас может возникнуть "соблазн" производить в любой программе инициализацию расширений при помощи одной из функций вида SetupGL1_X. Лучше этого не делать. Дело в том, что функции вида SetupGL1_X проверяют поддержку *всех* расширений OpenGL 1.x. В результате, функция SetupGL1_2 будет возвращать `false` даже на GeForce 2. А все из-за того, что GeForce 2 не поддерживает расширение EXT_texture3D, входящее в стандарт OpenGL 1.2. В то же время GeForce 2 поддерживает *почти* все расширения OpenGL 1.3.

Для демонстрации использования библиотеки ATI Extensions я переписал пример Ex07 с использованием этой библиотеки (Ex08).

```
...
#include <gl/glut.h>
#include <gl/glext.h>
#include <gl/wglext.h>
#include <gl/glATI.h>
```

```
#include <gl/wglATI.h>
#include <gl/ATIExtensions.h>
...
void Init()    // Инициализация OpenGL
{
//
       if (SetupExtensionsFromString("GL_ARB_window_pos
WGL_EXT_swap_control")==FALSE)
       {
              MessageBox(0, "Necessary extensions are not supported",
"Error", MB_OK);
              exit(-1);
       }
...
}
```

Как видно, программа, написанная с использованием библиотеки ATI Extensions, отличается от аналогичной GLH-программы лишь названиями функций. Однако функции инициализации библиотеки ATI Extensions имеют ряд преимуществ перед аналогичными функциями библиотеки NVIDIA GLH:

- поддержка значительно большего количества расширений (включая все расширения OpenGL 1.4);
- наличие дополнительных функций, позволяющих проинициализировать, к примеру, все расширения OpenGL 1.4 при помощи одной команды.

Тем не менее, библиотека ATI Extensions имеет и недостатки:

- отсутствие возможности получения информации о неподдерживаемых расширениях;
- отсутствие простого механизма добавления поддержки новых расширений без внесения изменений в исходный код библиотеки;
- конфликт имен с библиотекой NVIDIA GLH, что делает невозможным нормальное сосуществование NVIDIA GLH и ATI Extensions.

Следствие из предыдущих двух пунктов. Библиотеку ATI Extensions практически невозможно использовать для написания программ, использующих все возможности GPU NVIDIA, в то время как библиотека NVIDIA GLH может быть легко приспособлена для работы на видеокартах ATI.

Сравнив все достоинства и недостатки библиотек ATI Extensions и NVIDIA GLH, я решил использовать в примерах этой книги для инициализации OpenGL библиотеку NVIDIA GLH.

5.7. Простые расширения OpenGL

В этом разделе мы рассмотрим относительно простые расширения OpenGL, которым было бы нерационально посвящать персональную главу, а именно:

- расширения SGIS_texture_lod и EXT_texture_lod_bias, предоставляющее программисту расширенные возможности по управлению Mipmap-уровнями текстур;
- расширение EXT_texture_filter_anisotropic, предоставляющее возможность управления анизотропной фильтрацией из OpenGL;
- расширение SGIS_generate_mipmap, позволяющее генерировать Mipmap-уровни в реальном времени при помощи GPU.

5.7.1. Расширение SGIS_texture_lod

Расширение SGIS_texture_lod предоставляет программисту возможность управлять диапазоном Mipmap-уровней, используемых OpenGL при Mipmap-фильтрации текстур (режимы `GL_NEAREST_MIPMAP_NEAREST`, `GL_LINEAR_MIPMAP_NEAREST` и `GL_LINEAR_MIPMAP_LINEAR`).

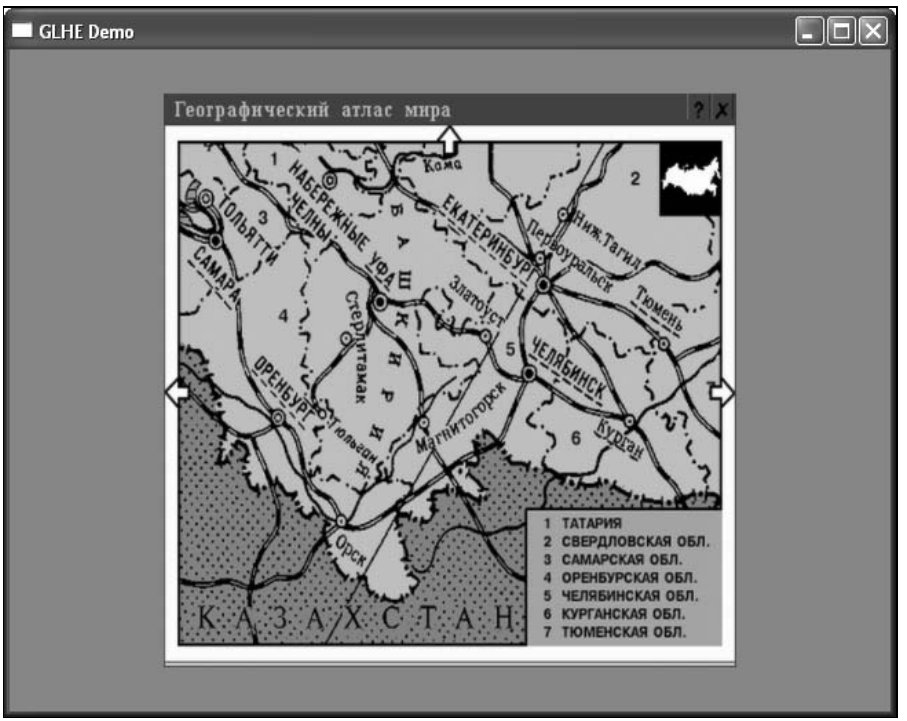


Рис. 5.3. Географическая карта, взятая из "Энциклопедии Кирилла и Мифодия 2003"

Для того чтобы понять причину появления этого расширения, мы напишем простую программу, выводящую на экран плоскость с наложенной географической картой (Ex09) (рис. 5.3). Так как этот пример отличается от примера Ex04 *главы 4* только текстурой, я не буду приводить исходного кода примера.

Запускаем программу. Все вроде бы нормально, однако если вы попытаетесь рассмотреть плоскость с картой под углом к плоскости экрана, то столкнетесь с очень неприятным эффектом — изображение карты станет настолько размытым, что с ней будет просто невозможно работать (рис. 5.4).

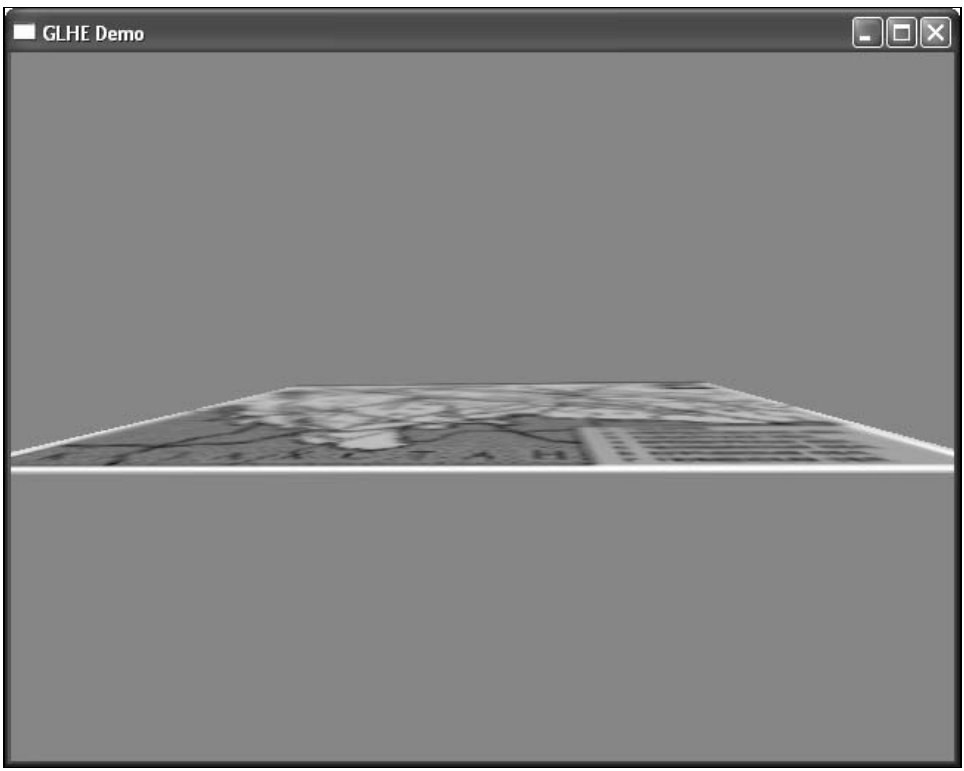


Рис. 5.4. Географическая карта, повернутая под углом к плоскости экрана

С чем это связано? Как известно, номер Mipmap-уровня, используемого OpenGL, зависит от коэффициента масштаба между изображением текстуры и размером текстурируемого многоугольника в пикселах. Причем при вычислении номера Mipmap-уровня OpenGL использует следующий коэффициент:

$$\lambda = \log_2 p,$$

где:

- ρ — коэффициент масштаба;
- λ — коэффициент, определяющий Mipmap-уровень.

Если $\lambda \leq 0$, то используется нулевой Mipmap-уровень, если $0 \leq \lambda \leq 1$, то первый и т. д.

> ***Технические подробности***
>
> Точки переключения Mipmap-уровней зависят от используемой реализации OpenGL и необязательно находятся в точках 0.0, 1.0, 2.0 и т. п. Например, реализация OpenGL может переключаться между нулевым и первым Mipmap-уровнем в точке 0.5.

Коэффициент масштаба между изображением текстуры и размером текстурируемого прямоугольника — это максимальный коэффициент масштаба для всех измерений. Например, если изображение текстуры 256×256 накладывается на прямоугольник 128×8 текселей, то $p = 32$ (x масштабируется на 2, y — на 32, используется максимальное значение, т. е. 32), поэтому λ — 5. Иными словами, в данном случае OpenGL наложит на прямоугольник 128×8 текселей текстуру размером 8×8 текселей. В результате изображение текстуры вдоль оси x будет очень размытым, а вдоль оси y — четким.

Обратите внимание, что на прямоугольники 8×8, 16×8, 32×8, 64×8 и т. д. будет наложен все тот же Mipmap-уровень с разрешением 8×8. Как видно, OpenGL выбирает Mipmap-уровень исходя из коэффициента масштабирования по одной размерности, т. е. используется *изотропная* Mipmap-фильтрация.

Как же нам избавиться от замыливания текстуры в примере Ex09? Один из вариантов решения проблемы — при помощи расширения SGIS_texture_lod запретить OpenGL использовать Mipmap-уровни с низким разрешением.

Для этой цели расширение SGIS_texture_lod добавляет в команду `glTexParameter` четыре новых параметра:

- `GL_TEXTURE_MIN_LOD_SGIS` — задает минимальный номер Mipmap-уровня текстуры, используемого при Mipmap-фильтрации;
- `GL_TEXTURE_MAX_LOD_SGIS` — задаем максимальный номер Mipmap-уровня текстуры, используемого при Mipmap-фильрации;
- `GL_TEXTURE_BASE_LEVEL_SGIS` — задает минимальный номер Mipmap-уровня текстурной карты;
- `GL_TEXTURE_MAX_LEVEL_SGIS` — задает максимальный номер Mipmap-уровня текстурной карты.

Например, для того, чтобы ограничить номера Mipmap-уровней, используемых при Mipmap-фильрации диапазоном [2..5], необходимо добавить следующие две команды:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_LOD_SGIS, 2);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LOD_SGIS, 5);
```

Однако зачем нам хранить в видеопамяти остальные неиспользуемые Mipmap-уровни? Гораздо разумнее их также ограничить диапазоном [2,5]:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL_SGIS, 2);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL_SGIS, 5);
```

Теперь мы можем при помощи команд `glTexImage2D` загружать в видеопамять только необходимые Mipmap-уровни вместо всего набора Mipmap-уровней.

Для демонстрации использования расширения SGIS_texture_lod на практике я переписал пример Ex09 с использованием этого расширения (Ex10). Для этого мне пришлось добавить в функцию `Init` всего две строки:

```
// Ограничиваем Mipmap-уровни диапазоном [0..2]
eburg.parameter(GL_TEXTURE_MIN_LOD_SGIS, 0);
eburg.parameter(GL_TEXTURE_MAX_LOD_SGIS, 2);
```

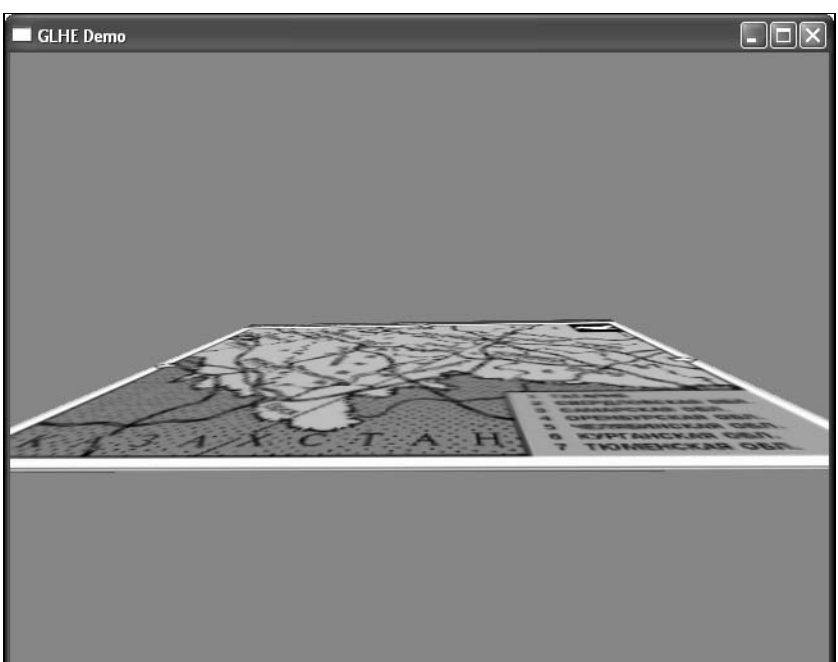


Рис. 5.5. Географическая карта, повернутая под углом к плоскости экрана. Номера используемых Mipmap-уровней ограничены диапазоном [0..2]

Использовать же значения параметров GL_TEXTURE_BASE_LEVEL_SGIS и GL_TEXTURE_MAX_LEVEL_SGIS в нашем случае бессмысленно, т. к. мы загружаем текстуры в видеопамять при помощи команды gluBuild2DMipmaps.

Как видно на рис. 5.5, использование расширения SGIS_texture_lod позволило улучшить четкость изображения. Однако общее качество картинки немного ухудшилось, т. к. мы фактически отказались от Mipmap-фильрации на дальней части плоскости. Иными словами, расширение SGIS_texture_lod не решило проблему, а лишь немного ее сгладило.

Немного другой подход к решению этой проблемы предлагает расширение EXT_texture_lod_bias.

5.7.2. Расширение EXT_texture_lod_bias

В *разд. 5.7.1* рассказывалось про то, что OpenGL определяет номер Mipmap-уровня, используемого при Mipmap-фильрации при помощи коэффициента λ, который равен коэффициенту масштаба между изображением текстуры и размером текстурируемого многоугольника в пикселах. Расширение EXT_texture_lod_bias предоставляет программисту возможность коррекции коэффициента λ путем его смещения на определенную величину.

Для этой цели расширение EXT_texture_lod_bias добавляет в команду glTexEnv новую цель (target) GL_TEXTURE_FILTER_CONTROL_EXT, имеющую параметр GL_TEXTURE_LOD_BIAS, задающий смещение коэффициента λ. Таким образом, для смещения коэффициента λ на + 0.5 необходимо выполнить следующую команду:

```
glTexEnvi(GL_TEXTURE_FILTER_CONTROL_EXT, GL_TEXTURE_LOD_BIAS_EXT, bias);
```

Не трудно догадаться, что положительное смещение коэффициента λ повышает четкость изображения, а отрицательное его размывает. Для демонстрации использования этого расширения, я изменил функцию консольной команды load_texture примера Ex09. Теперь команда load_texture загружает размытую текстуру, а затем постепенно увеличивает ее четкость, причем управление степенью размытости текстуры осуществляется при помощи расширения EXT_texture_lod_bias. Наиболее важные фрагменты получившейся программы (Ex11) приведены в листинге 5.7.

Листинг 5.7

```
float bias;
int last_tick;

// Обработчик события Idle
void Idle()
```

```
{
// Если смещение больше -1
       if (bias>-1.0)
       {
// Изменяем смещение
              int current_tick=glutGet(GLUT_ELAPSED_TIME);
              bias=bias-(current_tick-last_tick)*0.001;
              last_tick=current_tick;
// Смещение не должно быть больше -1
              if (bias<-1.0)
                     bias=-1.0;
// Выводим в заголовке окна текущее смещение
              char buf[128];
              sprintf(buf, "Bias=%f", bias);
              glutSetWindowTitle(buf);
// Устанавливаем смещение
              glTexEnvf(GL_TEXTURE_FILTER_CONTROL_EXT,
GL_TEXTURE_LOD_BIAS_EXT, bias);

// Перерисовываем изображение
              glutPostRedisplay();
       }
}

// Обработчик консольных команд
bool onConsole(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
// Если текущая консольная команда load_texture
       if (Cmd=="load_texture")
       {
...
// Загружаем текстуру
              pEburgImage=read(Params[0]);
              if (!pEburgImage)
              {
                     console->add(string("Can't open file
'")+Params[0]+string("'"));
                     return false;
              };
```

```
// Определяем максимальную размерность текстуры
              int size=max(pEburgImage->width, pEburgImage->height);
// Определяем значение смещения, при котором текстура точно окажется
// размытой
              bias=0;
              while (size!=0)
              {
                     bias++;
                     size/=2;
              }
// Устанавливаем смещение
              glTexEnvi(GL_TEXTURE_FILTER_CONTROL_EXT,
GL_TEXTURE_LOD_BIAS_EXT, bias);

// Загружаем текстуру в видеопамять
              eburg.bind();
              texImage2D(pEburgImage, &eburg, true);

              console->add(string("Texture \"")+Params[0]+string("\" has
been loaded"));
              last_tick=glutGet(GLUT_ELAPSED_TIME);
              return true;
       }
       return false;
}
```

Обратите внимание, что программа повышает четкость текстуры до тех пор, пока значение смещения не станет равно +1. Иными словами, текстура становится немного более четкой — это особенно заметно при рассмотрении плоскости с текстурой под углом к плоскости экрана. Однако четкость удаленных участков плоскости до сих пор оставляет желать лучшего. Если же вы попытаетесь поднять их четкость путем дальнейшего увеличения смещения параметра λ, то это лишь приведет к появлению артефактов.

Единственный способ избавиться от этого недостатка — отказаться от использования изотропной фильтрации в пользу анизотропной, которая будет рассмотрена в следующем разделе.

5.7.3. Расширение EXT_texture_filter_anisotropic

Расширение EXT_texture_filter_anisotropic предоставляет программисту возможность управления анизотропной фильтрацией непосредственно из своей

программы. Перед тем как перейти к рассмотрению этого расширения, мы попробуем понять, что представляет собой процесс фильтрации текстур, и зачем нужна анизотропная фильтрация.

Что такое текстура? Текстура — это изображение, натягиваемое на полигональный объект для придания реалистичности итоговому изображению. В общем виде двухмерная текстура задается текстурной функцией `f(s, t)`, ставящей в соответствие каждой паре текстурных координат `s` и `t` определенный цвет. В OpenGL текстурная функция задается таблично при помощи прямоугольного массива данных: данных о цвете, альфа-компоненте и т. д.

Текстурная фильтрация — это процесс выборки значения из текстурной функции для текущего пиксела. Это отнюдь не тривиальная задача. Дело в том, что частота текстурной функции в системе координат текстуры (разрешение текстуры) очень редко совпадает с частотой выборки пикселов из плоскости изображения (рис. 5.6).

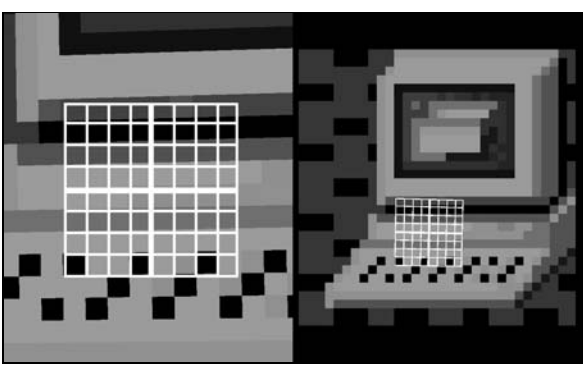


Рис. 5.6. Размер текселя текстуры совпадает с размером пиксела экрана

Чаще случается, что размер текселя текстуры или меньше размера пиксела экрана (рис. 5.7), или больше (рис. 5.8). В первом случае OpenGL использует фильтр, заданный командой `glTexParameter` с параметром `GL_TEXTURE_MIN_FILTER`, а во втором — `GL_TEXTURE_MAG_FILTER`.

Три ситуации, описанные выше, возникают только в том случае, если полигон, на который накладывается текстура, параллелен плоскости экрана. Однако на практике большинство полигонов повернуто к плоскости экрана, в результате чего ориентация пикселов экрана не совпадает с ориентацией текселей (рис. 5.9). В этом случае OpenGL сталкивается с неразрешимым противоречием, описанным в *разд. 5.7.1*, — коэффициент масштаба между изображением текстуры и размером текстурируемого многоугольника в пикселах имеет разное значение для осей `s` и `t`. При использовании изотропной фильрации OpenGL должен отдать предпочтение одному из коэффициентов

масштаба вдоль одной из осей, пожертвовав другим. В результате изображение теряет четкость вдоль одной из осей, становясь размытым.

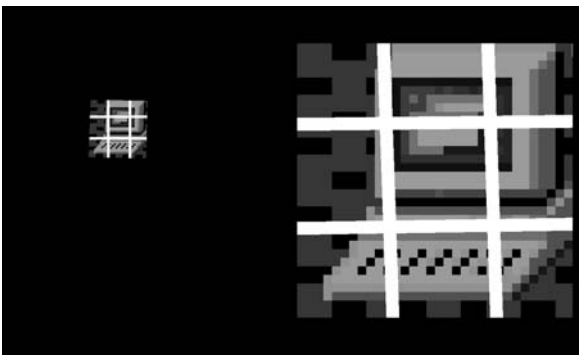


Рис. 5.7. Размер текселя текстуры меньше размера пиксела экрана

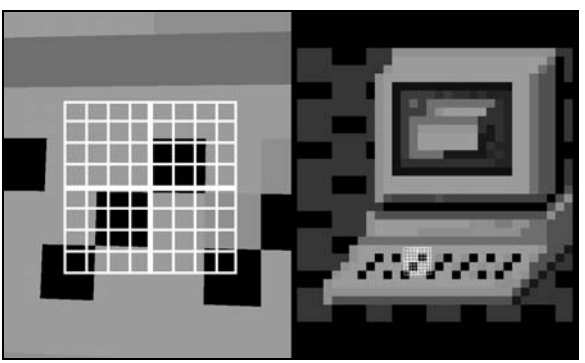


Рис. 5.8. Размер текселя текстуры больше размера пиксела экрана

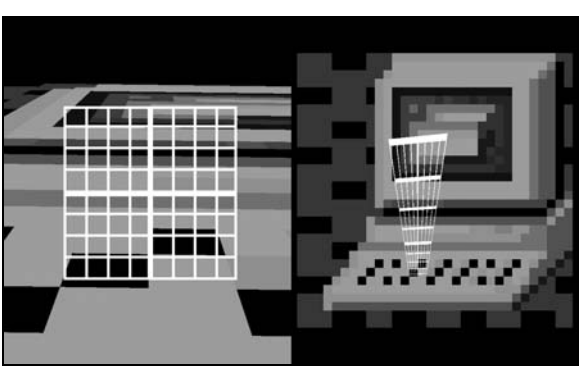


Рис. 5.9. Полигон, на который накладывается текстура, лежит под углом к экрану

При использовании анизотропной фильтрации GPU старается учитывать угол между полигоном и плоскостью экрана. Например, GPU может делать выборку из текстуры вдоль осей `s` и `t` с разной частотой. В настоящее время практически каждый производитель GPU использует свой собственный метод анизотропной фильтрации. Поэтому мы не будем лезть в дебри, а сразу перейдем к рассмотрению расширения EXT_texture_filter_anisotropic, тем более, что это расширение абсолютно не привязано к какому-нибудь определенному методу анизотропной фильтрации. Более подробную информацию об особенностях различных реализаций анизотропной фильтрации можно найти, например, в [26], [27] и [28].

Расширение EXT_texture_filter_anisotropic добавляет в команду `glTexParameter` новый параметр `GL_TEXTURE_MAX_ANISOTROPY_EXT`, задающий степень анизотропии. Какой физический смысл имеет степень анизотропии? Никакого. Просто чем больше степень анизотропии, тем качественнее фильтрация. Но за качество в большинстве случаев приходится расплачиваться падением производительности.

Анизотропная фильтрация включается только при степени анизотропии больше 1. Если же степень анизотропии равна 1, то используется классическая изотропная фильтрация. Максимальная степень анизотропии, поддерживаемая текущей реализацией OpenGL, хранится в переменной состояния `GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT`. Таким образом, для того, чтобы добавить в пример Ex09 поддержку анизотропной фильтрации наилучшего качества, в него достаточно добавить всего несколько строк (Ex12):

```
GLfloat largest_supported_anisotropy;
// Получаем максимальную степень анизотропии, поддерживаемую текущей
// реализацией OpenGL
glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT,
&largest_supported_anisotropy);
// Устанавливаем степень анизотропии
eburg.parameter(GL_TEXTURE_MAX_ANISOTROPY_EXT,
&largest_supported_anisotropy);
// Выводим в консоль информацию о текущей реализации OpenGL
char buf[128];
sprintf(buf, "Largest supported anisotropy = %f", larg-
est_supported_anisotropy);
console.add(buf);
```

Небольшое изменение программы привело к резкому улучшению качества изображения — от замыливания текстуры при повороте плоскости не осталось и следа.

Примечание

Драйверы большинства современных видеокарт, в частности, ATI и NVIDIA, позволяют вручную форсировать принудительное использование анизотропной фильтрации вместо изотропной. Однако приложению не следует полагаться на то, что пользователь вручную включит анизотропную фильтрацию из драйверов — эту операцию способны выполнить лишь, продвинутые, пользователи, знакомые с азами 3D-графики.

5.7.4. Использование расширения SGIS_generate_mipmap

Как известно, в базовой версии OpenGL приложение обязано самостоятельно создавать все Mipmap-уровни текстур и загружать их в видеопамять при помощи команды `glTexImage2D`. Команда `gluBuild2DMipmaps` в действительности является простой надстройкой над командой `glTexImage2D`, программно создающей Mipmap-уровни текстур с последующей их загрузкой в видеопамять командой `glTexImage2D`.

В результате создание Mipmap-уровней текстур превращается в очень затратную задачу:

- приложение тратит драгоценные ресурсы CPU на генерацию Mipmap-уровней;
- объем текстур, загружаемых в видеопамять через медленную шину AGP, возрастает на 33%;
- для того чтобы создать Mipmap-уровни динамически генерируемой текстуры, ее приходится копировать из видеопамяти в оперативную память, затем создавать Mipmap-уровни и, наконец, пересылать полученную информацию обратно в видеопамять. При этом все пересылки выполняются через медленную шину AGP! Следствием этого является невозможность практического использования Mipmap-фильтрации с динамическими текстурами — проще уж совсем отказаться от Mipmap-фильтрации, чем платить за нее такую цену.

К счастью, в начале 1997 года появилось расширение SGIS_generate_mipmap, позволяющее OpenGL самостоятельно создавать набор Mipmap-уровней на основе заданной текстуры. Правда в те далекие времена на рынке царствовали Voodoo и Riva128, поэтому ни о какой аппаратной поддержке этого расширения бытовыми ускорителями не могло быть и речи.

Ситуация изменилась после выхода GeForce 256, который "научился" работать с расширением SGIS_generate_mipmap. Однако "учился" он это делать

довольно долго — поддержка этих расширений появилась в драйверах Detonator 10.xx, которые вышли после поступления в продажу GeForce 2.

Для управления автоматической генерацией Mipmap-уровней расширение SGIS_generate_mipmap добавляет в команду `glTexParameter` новый параметр `GL_GENERATE_MIPMAP_SGIS`. Установка параметра `GL_GENERATE_MIPMAP_SGIS` в значение `GL_TRUE` включает автоматическую генерацию Mipmap-уровней, а `GL_FALSE` — выключает.

Для демонстрации использования расширения SGIS_generate_mipmap я переписал предыдущий пример Ex12, добавив в него автоматическую генерацию Mipmap-уровней (Ex13). При этом все изменения фактически свелись к замене строки:

```
texImage2D(pEburgImage, &eburg);
```

на

```
eburg.parameter(GL_GENERATE_MIPMAP_SGIS, TRUE);
texImage2D(pEburgImage, &eburg, false)
```

Обратите внимание, что мы загружаем в видеопамять только исходную текстуру (нулевой уровень), а все остальное видеокарта делает за нас! Следует помнить, что генерация Mipmap-уровней всегда происходит только в момент изменения текстуры. Следовательно, если мы поменяем две последние команды местами, то Mipmap-уровни не будут созданы, и мы не увидим текстуру на плоскости.

В данном примере использование расширения SGIS_generate_mipmap лишь слегка увеличивает скорость загрузки текстуры в видеопамять. Использование этого расширения для динамических текстур способно увеличить производительность программы в несколько раз. Такие примеры будут рассмотрены в *главе 8*, посвященной работе с буферами пикселов (`pbuffer`).

Заключение

Расширения OpenGL позволяют программистам задействовать в своих программах все возможности новейших ускорителей. В этой главе мы изучили основные типы расширений OpenGL, а также научились пользоваться спецификацией расширений. Кроме того, мы рассмотрели семь относительно простых расширений OpenGL:

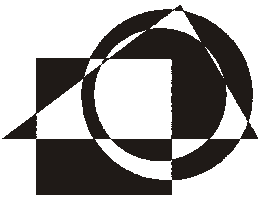
- EXT_separate_specular_color;
- ARB_window_pos;
- WGL_EXT_swap_control;

- SGIS_texture_lod;
- EXT_texture_lod_bias;
- EXT_texture_filter_anisotropic;
- SGIS_generate_mipmap.

Так как проверка и инициализация расширений OpenGL при помощи OpenGL API — довольно трудоемкий процесс, мы изучили инструментарий корпораций ATI и NVIDIA для быстрой инициализации расширений OpenGL: библиотеки ATI Extensions и NVIDIA OpenGL Helper Library.

Если после прочтения этой главы у вас остались некоторые вопросы по расширениям OpenGL, то я бы посоветовал вам обратиться к [18].

Глава 6



Расширения EXT_texture_rectangle и NV_texture_rectangle

При разработке OpenGL приложений многие начинающие программисты обычно используют готовые текстуры сторонних производителей. Однако в большинстве случаев они не имеют разрешения, кратного степени 2. Мы помним, что OpenGL не поддерживает работу с такими текстурами. Конечно, профессиональные разработчики могут попросить художников рисовать текстуры с нужным разрешением, но программистам-любителям это, как правило, недоступно. В итоге им приходится искусственно повышать разрешение текстуры до степени, кратной 2, но такой подход имеет два недостатка:

- избыточные затраты памяти. Например, текстуру 640 × 480 : 32 придется увеличить до 1024 × 512 : 32, т. е. объем необходимой видеопамяти увеличится на 868 Кбайт (41%);
- увеличение размера текстуры происходит не в целое число раз, что негативно сказывается на качестве изображения.

Кроме того, для рисования многих элементов интерфейса программ также удобно использовать текстуры, размеры которых не кратны степени 2. Использование вместо них обычных текстур приводит к необоснованному перерасходу памяти.

К счастью, все современные видеокарты умеют работать с текстурами размера, не кратного степени 2 (в дальнейшем для краткости мы будем называть такие текстуры NPOTD-текстурами (non-power-of-two dimensioned), а "обычные" текстуры — POTD-текстурами (power-of-two dimensioned)).

Использование NPOTD-текстур на большинстве GPU осуществляется при помощи расширения EXT_texture_rectangle. К сожалению, в это "большинство" не входят GPU корпорации NVIDIA, поддерживающие вместо расширения EXT_texture_rectangle собственное расширение NV_texture_rectangle. Другие производители GPU (и корпорация ATI в частности) не поддерживают расширение NV_texture_rectangle. В результате, как всегда, страдают разработчики, которые вынуждены поддерживать в своих про-

граммах оба расширения. К счастью, все не так уж плохо. Дело в том, что эти расширения очень похожи друг на друга и различаются разве что названиями констант.

Расширения EXT_texture_rectangle и NV_texture_rectangle очень просты в использовании. Включение наложения NPOTD-текстур осуществляется командой `glEnable` с параметрами `GL_TEXTURE_RECTANGLE_EXT` и `GL_TEXTURE_RECTANGLE_NV`. Но при работе с NPOTD-текстурами следует учитывать их отличие от POTD-текстур. NPOTD-текстуры используют ненормализованные текстурные координаты, значения которых находятся в диапазоне [0 ширина_текстуры, 0 высота_текстуры]. Кроме того, перед использованием NPOTD-текстур необходимо установить выравнивание строк пикселов на границу 1 байта (по умолчанию строки пикселов выравниваются на границу слова (4 байта)). Если этого не сделать, то текстура, скорее всего, будет искажена (рис. 6.1).

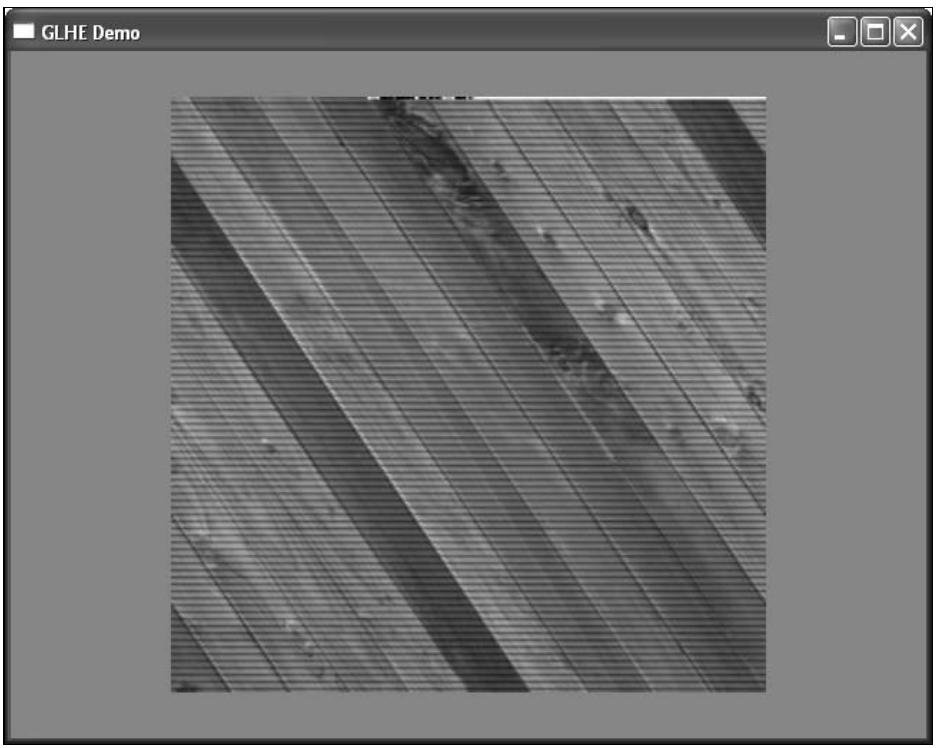


Рис. 6.1. Текстура размера 254×256, загруженная в режиме выравнивания строк пикселов на границу слова

NPOTD-текстуры имеют ряд существенных ограничений по сравнению с обычными двухмерными текстурами:

- NPOTD-текстуры не поддерживают Mipmap-фильтрацию;
- NPOTD-текстуры не поддерживают режим наложения текстуры GL_REPEAT;
- NPOTD-текстуры не поддерживают границу вокруг текстуры шириной в 1 тексель (border);
- максимальный размер NPOTD-текстуры, как правило, несколько меньше максимального размера POTD-текстуры. Например, максимальный размер POTD- и NPOTD-текстур на GPU семейства NV1x (GeForce256 — GeForce4 MX) соответственно равен 2048×2048 и 2046×2046;
- NPOTD-текстуры, как правило, не могут использовать все внутренние форматы хранения текстур, доступные POTD-текстурам.

Однако эти ограничения не являются существенными, т. к. в большинстве случаев NPOTD-текстуры используются для рисования пользовательского интерфейса или аналогичных задач.

Рис. 6.2. Плоскость с NPOTD-текстурой дерева размером 512×768

Для того чтобы продемонстрировать использование расширения EXT_texture_rectangle на практике, я написал небольшой пример (Ex01), выводящий на экран плоскость с наложенной текстурой дерева 512×768 (рис. 6.2).

Исходный код программы приведен в листинге 6.1.

Листинг 6.1

```
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include "glh_glut_ext.h"
#include "nv_util_ext.h"

using namespace std;
using namespace glh;

int WinWidth=640;            // Ширина окна
int WinHeight=480;           // Высота окна

// Указатель на текстурный объект
GLuint wood;
//Изображение текстуры дерева
tga::tgaImage* pWoodImage=0;
// Дисплейный список с плоскостью
display_list list0;

glut_console console;
glut_simple_user_interface user(&console);
glut_callbacks cb;
glut_swapbuffers swapbuffers;

const string Title="GLHE Demo";

void Init()                          // Инициализация OpenGL
{
// Проверяем поддержку расширения EXT_texture_rectangle
       if (!glh_init_extensions("GL_EXT_texture_rectangle"))
       {
```

```
                console.add(string("Unsupported extension:
")+glh_get_unsupported_extensions());
                console.exit(-1);
                return;
        }
        glEnable(GL_DEPTH_TEST);
// Устанавливаем выравнивание строк пикселей на границу 1 байта
        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

        glClearColor(0.5, 0.5, 0.75, 1);
// Создаем текстурный объект и настраиваем его параметры
        glGenTextures(1, &wood);
        glBindTexture(GL_TEXTURE_RECTANGLE_EXT, wood);
        glTexParameterf(GL_TEXTURE_RECTANGLE_EXT, GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
        glTexParameterf(GL_TEXTURE_RECTANGLE_EXT, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
        glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
}

void Display()                // Обновление содержимого экрана
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        list0.call_list();           //Вывод плоскости на экран
}

void onExit()
{
// Удаляем текстурный объект
        glDeleteTextures(1, &wood);
// Удаляем изображение текстуры из памяти
        if (pWoodImage)
        {
                if (pWoodImage->pixels)
                        delete[] pWoodImage->pixels;
                delete pWoodImage;
        }
}
```

```
// Обработчик консольных команд
bool onConsole(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
// Если консольная команда "load_texture"
       if (Cmd=="load_texture")
       {
// Если у команды нет параметров, то — ошибка
              if (Params.size()==0)
                     return false;
// Если параметр = "/?", выводим справку по команде
              if (Params[0]=="/?")
              {
                     console->add("loads texture from file");
                     console->add("");
                     console->add("load_texture <filename>");
                     console->add("where <filename> is file with tex-
ture");
                     return true;
              }

// Выгружаем старую текстуру из памяти
              if (pWoodImage)
              {
                     if (pWoodImage->pixels)
                     {
                            delete[] pWoodImage->pixels;
                            pWoodImage->pixels=0;
                     }
                     delete pWoodImage;
                     pWoodImage=0;
              }
// Загружаем новую текстуру файла
              pWoodImage=read(Params[0]);
// Если не получилось загрузить текстуру из файла, возвращаем false
              if (!pWoodImage)
              {
                     console->add(string("Can't open file
'")+Params[0]+string("'"));
```

```
                        return false;
                };
// Загружаем текстуру в видеопамять
                glBindTexture(GL_TEXTURE_RECTANGLE_EXT, wood);
                glTexImage2D(GL_TEXTURE_RECTANGLE_EXT, 0,
pWoodImage->components, pWoodImage->width,
                        pWoodImage->height, 0, pWoodImage->format,
GL_UNSIGNED_BYTE, pWoodImage->pixels);
// Создаем дисплейный список плоскости
                list0.new_list(GL_COMPILE);
                        glBindTexture(GL_TEXTURE_RECTANGLE_EXT, wood);
// Включаем наложение NPOTD-текстур
                        glEnable(GL_TEXTURE_RECTANGLE_EXT);
// Рисуем плоскость
                        glBegin(GL_QUADS);
                                glTexCoord2f(0, 0);
                                glVertex3f(-1, -1, 0);


                                glTexCoord2f(0, pWoodImage->height);
                                glVertex3f(-1,  1, 0);

                                glTexCoord2f(pWoodImage->width,
pWoodImage->height);
                                glVertex3f( 1,  1, 0);

                                glTexCoord2f(pWoodImage->width, 0);
                                glVertex3f( 1, -1, 0);
                        glEnd();
                        glDisable(GL_TEXTURE_RECTANGLE_EXT);
                list0.end_list();

                console->add(string("Texture \"")+Params[0]+string("\" has
been loaded"));
                return true;
        }
        return false;
}

int main(int argc, char* argv[])
{
```

```
// Инициализируем GLUT
        console.add("Initializing GLUT...");
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
        glutInitWindowSize(WinWidth,WinHeight);
        glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                                              (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
        glutCreateWindow(Title.c_str());
// Инициализируем GLHE
        console.add("Initializing OpenGL Helper Library...");
        glut_helpers_initialize();

        console.title=Title;
// Регистрируем консольную команду "load_texture"
        console.addCmd("load_texture", "loads texture from file", 0,
onConsole);

        user.user_mouse.configure_buttons(1);
        user.user_mouse.dolly.dolly[2]=-2;

        cb.display_function=Display;

        glut_add_interactor(&user);
        glut_add_interactor(&cb);
        glut_add_interactor(&console);
        glut_add_interactor(&swapbuffers);

        console.add("Initializing OpenGL...");
        Init();

        atexit(onExit);
// Выполняем скрипт из файла autoexec.cs. По умолчанию он загружает
// текстуру из файла "..\CEDFENCE.tga"
        console.processCmd("exec autoexec.cs");

        console.add("Initialize complete. Starting glutMainLoop");
// Запускаем цикл обработки сообщений
        glutMainLoop();

        return 0;
}
```

Для того чтобы "заставить" пример Ex01 использовать расширение NV_texture_rectangle вместо EXT_texture_rectangle, в него необходимо внести два изменения (Ex02):

- заменить проверку наличия расширения EXT_texture_rectangle на проверку наличия расширения NV_texture_rectangle. Иными словами, надо заменить первую строку функции `Init` на `if (!glh_init_extensions ("GL_NV_texture_rectangle"));`
- заменить везде в тексте программы константу `GL_TEXTURE_RECTANGLE_EXT` на `GL_TEXTURE_RECTANGLE_NV`.

Второе изменение является больше "косметическим", чем реально необходимым. Дело в том, что константы `GL_TEXTURE_RECTANGLE_EXT` и `GL_TEXTURE_RECTANGLE_NV` имеют одно и то же значение — 0 × 84F5. Кстати, это утверждение верно и для других констант расширений EXT_texture_rectangle и NV_texture_rectangle. Следствием из этого факта является отсутствие необходимости писать два варианта кода для поддержки расширений EXT_texture_rectangle и NV_texture_rectangle: достаточно только убедиться в поддержке видеокартой одного из этих расширений, а затем при работе с NPOTD-текстурами использовать константы одного из двух расширений по выбору.

Так как при работе с NPOTD-текстурами используется уникальная целевая константа (GL_TEXTURE_RECTANGLE_EXT или GL_TEXTURE_RECTANGLE_NV), для работы с этими текстурами нельзя использовать классы `tex_object_1d` и `tex_object_2d` библиотеки NVIDIA GLH. К счастью, заботливые создатели библиотеки GLH предусмотрели отдельный класс для NPOTD-текстур: `tex_object_rectangle`, который отличается от других классов семейства `tex_object_xxx` лишь конструктором:

```
class tex_object_rectangle : public tex_object
{
public: tex_object_rectangle() : tex_object(GL_TEXTURE_RECTANGLE_NV) {}
};
```

Для демонстрации использования класса `tex_object_rectangle` я переписал примеры Ex01 и Ex02 с использованием этого класса (листинг 6.2) (Ex03).

Листинг 6.2

```
...
tex_object_rectangle wood;
tga::tgaImage* pWoodImage=0;
display_list list0;
...
```

```
void Init()                              // Инициализация OpenGL
{
       if (!glh_init_extensions("GL_NV_texture_rectangle"))
              if (!glh_init_extensions("GL_EXT_texture_rectangle"))
              {
                     console.add("Unsupported extension:
EXT_texture_rectangle/NV_texture_rectangle");
                     console.exit(-1);
                     return;
              }
       glEnable(GL_DEPTH_TEST);
       glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

       glClearColor(0.5, 0.5, 0.75, 1);

       wood.bind();
       wood.parameter(GL_TEXTURE_MIN_FILTER, GL_LINEAR);
       wood.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
       glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
}
...
bool onConsole(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
       if (Cmd=="load_texture")
       {
...
              pWoodImage=read(Params[0]);
              if (!pWoodImage)
              {
                     console->add(string("Can't open file
'")+Params[0]+string("'"));
                     return false;
              };

              wood.bind();
              texImage2D(pWoodImage, &wood, false);

              list0.new_list(GL_COMPILE);
                     wood.bind();
```

```
                        wood.enable();
                        glBegin(GL_QUADS);
                                glTexCoord2f(0, 0);
                                glVertex3f(-1, -1, 0);


                                glTexCoord2f(0, pWoodImage->height);
                                glVertex3f(-1,  1, 0);

                                glTexCoord2f(pWoodImage->width,
pWoodImage->height);
                                glVertex3f( 1,  1, 0);

                                glTexCoord2f(pWoodImage->width, 0);
                                glVertex3f( 1, -1, 0);
                        glEnd();
                        wood.disable();
                list0.end_list();

                console->add(string("Texture \"")+Params[0]+string("\" has
been loaded"));
                return true;
        }
        return false;
}
```

Как видно, пример упростился за счет того, что отпала необходимость "ручного" управления текстурными объектами.

До сих пор мы задавали ненормализованные текстурные координаты объектов "вручную" в тексте в программы. Но "ручное" задание координат работает только с теми объектами, для которых мы можем явным образом задавать ненормализованные координаты текстур. Если же используются готовые объекты библиотек GLU/GLUT или включается режим автоматической генерации текстурных координат (сферических и т. д.), то текстурные координаты будут автоматически нормализованы (т. е. приведены к диапазону [0..1]. Следовательно, нам надо как-нибудь привести их к диапазону [0 .. ширина_текстуры, 0 .. высота_текстуры]. Это можно сделать путем масштабирования матрицы текстуры. Продемонстрируем этот метод на примере Ex04. Программа из примера Ex04 выводит на экран чайник из библиотеки GLUT с наложенной на него NPOTD-текстурой (рис. 6.3).

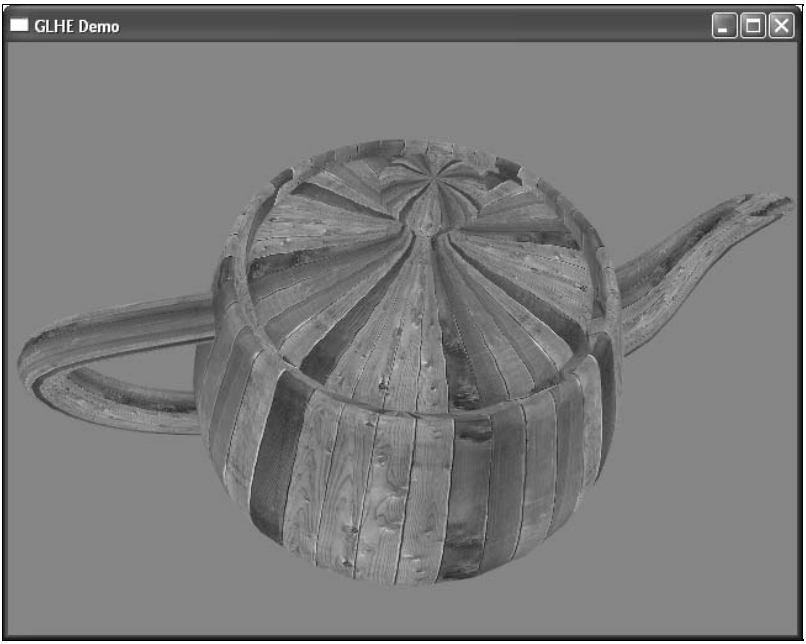


Рис. 6.3. Чайник из библиотеки GLUT с наложенной NPOTD-текстурой

Ниже приведен фрагмент кода из этого примера, отвечающий за создание дисплейного списка чайника:

```
list0.new_list(GL_COMPILE);
        wood.bind();
        wood.enable();
// Масштабируем матрицу текстуры
        glMatrixMode(GL_TEXTURE);
        glPushMatrix();
        glScalef(pWoodImage->width, pWoodImage->height, 1);
        glMatrixMode(GL_MODELVIEW);
// Рисуем чайник
        glutSolidTeapot(2.0);

// Восстанавливаем матрицу текстуры
        glMatrixMode(GL_TEXTURE);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
        wood.disable();
list0.end_list();
```

Этот прием позволяет накладывать NPOTD-текстуры на любой объект, кроме объектов, использующих режим наложения текстуры с повторением (GL_REPEAT), т. к. NPOTD-текстуры не поддерживают этот режим.

6.1. Добавление в библиотеку ASE Reader поддержки NPOTD-текстур

Как вы помните, в *главе 4* мы разработали библиотеку ASE Reader, позволяющую загружать модели из файлов формата ASE. Так как текстуры в 3D Studio MAX могут иметь любое разрешение, эта библиотека должна уметь использовать NPOTD-текстуры, как говорится, "на всякий случай". И самое интересное, что библиотека ASE Reader почти умеет это делать. Дело в том, что для загрузки текстур в видеопамять она использует команду `gluBuild2DMipmaps`, которая автоматически масштабирует NPOTD-текстуру до ближайшей по размерам POTD-текстуры. Правда такое решение нельзя признать удовлетворительным, т. к. при масштабировании текстура сильно теряет в качестве. Решением данной проблемы является использование расширений EXT_texture_rectangle и NV_texture_rectangle для вывода NPOTD-текстур.

Для этого в библиотеку ASE Reader придется внести некоторые изменения (Ex05). Все дело в том, что диспетчер текстур — `CTextureManager` — хранит только указатель на текстурный объект. В то же время для наложения NPOTD-текстуры библиотеке нужна информация о размерах текстуры. Следовательно, диспетчер текстур должен возвращать не просто указатель на текстурный объект, а целую структуру, содержащую внутри себя наряду с указателем на текстурный объект и другую информацию об объекте. А это, в свою очередь, потребует внесения множества мелких изменений в разные фрагменты кода библиотеки ASE Reader. В листинге 6.3 приведены новые определения классов библиотеки ASE Reader.

Листинг 6.3

```
// Константы для выбора режима работы диспетчера текстур с NPOTD-текстуры
enum
{
// Никогда не использовать NPOTD-текстуры
       TR_NEVER,
// Использовать NPOTD-текстуры только там, где это реально необходимо
       TR_AUTO,
// Всегда использовать NPOTD-текстуры вместо POTD-текстур
       TR_ALWAYS
};
```

```
// Структура с информацией о текстурном объекте
class CTexture
{
public:
// Файл, из которого была загружена текстура
       string filename;
// Счетчик ссылок на текстуру
       unsigned int count;
// Ширина текстуры
       unsigned int width;
// Высота текстуры
       unsigned int height;
// Указатель на текстурный объект
       tex_object* pTextureObj;
};
// Диспетчер текстур
class CTextureManager
{
public:
// Режим работы с NPOTD-текстурами
       unsigned int TextureRectangleMode;
// Список текстур
       list<CTexture> textures;
       CTextureManager();
       ~CTextureManager();
// Загружает текстуру из файла и возвращает структуру с описанием
// текстурного объекта
       CTexture* load_texture(string zip_filename, string
inzip_filename);
// Выгружает текстуру из памяти
       void unload_texture(CTexture* pTexture);
// Возвращает информацию о текстуре
       list<string> get_info();
};

#endif
// Структура с информацией о модели
typedef struct _modelData
```

```
{
    int *indices;
    float *vertices;
    float *normals;
    float *texcoords;
    int numfaces;
    float diffuse[4];
    float tm[16];
#ifdef __ASE_EXT
       float reflect_color[4];
// Указатель на структуру с информацией о диффузной текстуре
       CTexture* diffuse_texture;
// Указатель на структуру с информацией о текстуре отражения
       CTexture* reflect_texture;
       string* name;
#endif
} modelData;
// Класс для работы с моделями формата ASE
class CAseModel
{
    public:
        CAseModel();
        ~CAseModel();
// Загружает модель из файла
        bool LoadModel(char *filename, char *modelname);
// Рисует модель на экране
        void DrawModel(bool textured);
// Возвращает размеры модели
       void GetExtents(vec3f &minExtent, vec3f &maxExtent);
// Выгружает модель из памяти
       void Clear(bool clearTextures=true);
// Выводит информацию о модели
       list<string> get_info();
// Указатель на структуру
        modelData *data;
// Количество объектов в ASE-модели
        int numChunks;
};
```

Наибольшие изменения претерпел метод `CTextureManager::load_texture`, отвечающий за загрузку текстур в видеопамять (листинг 6.4). Режим работы с NPOTD-текстурами задается полем `TextureRectangleMode`. Если оно установлено в `TR_NEVER`, метод `load_texture` никогда не использует NPOTD-текстуры. Если поле `TextureRectangleMode` равно `TR_AUTO`, то метод `load_texture` использует NPOTD-текстуры только для текстур, размеры которых не кратны степени 2. Если же оно равно `TR_ALWAYS`, то метод `load_texture` считает любую текстуру NPOTD-текстурой.

Листинг 6.4

```
// Загрузка текстуры из файла
CTexture* CTextureManager::load_texture(string zip_filename, string
inzip_filename)
{
// Получаем короткое и полное имя файла
        string short_inzip_filename=inzip_filename;
        int pos=inzip_filename.find_last_of("\\");
        if (pos!=-1)
        {
                if (pos!=short_inzip_filename.size()-1)
                        short_inzip_filename=inzip_filename.substr(pos+1,
inzip_filename.size()-pos-1);
                else
                        return 0;
        }
        string fullname=zip_filename+"::"+short_inzip_filename;
// Перебиваем все загруженные текстуры
        for (list<CTexture>::iterator itor=textures.begin();
itor!=textures.end(); itor++)
        {
// Если такая текстура уже загружена
                if (itor->filename==fullname)
                {
// Увеличиваем счетчик ссылок и возвращаем указатель на нее
                        itor->count++;
                        return &(*itor);
                }
        }
```

```
// Создаем новый экземпляр структуры с описанием текстурного объекта
       CTexture* pTexture=new CTexture;
// Устанавливаем счетчик ссылок в 1
       pTexture->count=1;
// Загружаем изображение из файла
       tga::tgaImage* image=read(zip_filename, short_inzip_filename);
// Если загрузка не удалась, убираем
       if (!image)
       {
// Пытаемся загрузить ее по "настоящему" пути
              image=read(inzip_filename);
              if (!image)
              {
                     delete pTexture;
                     return false;
              }
              else
                     pTexture->filename=inzip_filename;
       }
       else
              pTexture->filename=fullname;
// Сохраняем размеры текстуры
       pTexture->width=image->width;
       pTexture->height=image->height;
// Определяем, какой тип текстуры выбрать для изображения (GL_TEXTURE_2D
// или GL_TEXTURE_RECTANGLE_NV)
       if (((isPower2(image->width) && isPower2(image->height)) ||
(TextureRectangleMode==TR_NEVER)) && (TextureRectangleMode!=TR_ALWAYS))
              pTexture->pTextureObj=new tex_object_2D;
       else
              pTexture->pTextureObj=new tex_object_rectangle;

       pTexture->pTextureObj->bind();
// Выбираем режим фильтрации в зависимости от типа текстуры
       if (pTexture->pTextureObj->target==GL_TEXTURE_RECTANGLE_NV)
              pTexture->pTextureObj->parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR);
       else
```

```
                pTexture->pTextureObj->parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);


        pTexture->pTextureObj->parameter(GL_TEXTURE_MAG_FILTER,
GL_LINEAR);


// Загружаем текстуру в видеопамять. Загрузка NPOTD-текстур
// и POTD-текстур выполняется по-разному
        if (pTexture->pTextureObj->target==GL_TEXTURE_RECTANGLE_NV)
        {
                glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
                texImage2D(image, pTexture->pTextureObj, false);
        }
        else
                texImage2D(image, pTexture->pTextureObj);
// Помещаем информацию о текстурном объекте в список текстур
        textures.push_back(*pTexture);
// Возвращаем указатель на информацию о текстурном объекте
        return pTexture;
}
// Кроме того, довольно серьезные изменения были внесены метод
// CAseModel::DrawModel:
void CAseModel::DrawModel(bool textured)
{
...
// Если включено наложение диффузной текстуры
        if (textured && data[n].diffuse_texture)
        {
                data[n].diffuse_texture->pTextureObj->bind();
                data[n].diffuse_texture->pTextureObj->enable();
                glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_MODULATE);
// Если диффузная текстура является NPOTD-текстурой
                if (data[n].diffuse_texture->pTextureObj->
target==GL_TEXTURE_RECTANGLE_NV)
                {
// Масштабируем матрицу текстуры
                        glMatrixMode(GL_TEXTURE);
                        glPushMatrix();
```

```
                        glScalef(data[n].diffuse_texture->width,
data[n].diffuse_texture->height, 1.0);
                }
        }


// Рисуем объект
         glDrawElements(GL_TRIANGLES, data[n].numfaces*3, GL_UNSIGNED_INT,
data[n].indices);
// Делаем "все, как было раньше"
        if (textured && data[n].diffuse_texture)
        {
                data[n].diffuse_texture->pTextureObj->disable();
                if (data[n].diffuse_texture->pTextureObj->
target==GL_TEXTURE_RECTANGLE_NV)
                {
                        glPopMatrix();
                        glMatrixMode(GL_MODELVIEW);
                }
        }
// Если имеется зеркальная текстура, то делаем второй проход
        if (textured && data[n].reflect_texture)
        {
// Настраиваем параметры OpenGL
                glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
                glEnable(GL_BLEND);
                glDepthMask(false);
                glDepthFunc(GL_LEQUAL);
                glEnable(GL_TEXTURE_GEN_S);
                glEnable(GL_TEXTURE_GEN_T);
                glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
                glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
                glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
L_MODULATE);
                data[n].reflect_texture->pTextureObj->bind();
                data[n].reflect_texture->pTextureObj->enable();
                glColor4fv(&data[n].reflect_color[0]);
// Если текстура является NPOTD-текстурой
                if (data[n].reflect_texture->pTextureObj->
target==GL_TEXTURE_RECTANGLE_NV)
                {
// Корректируем матрицу текстуры
```

```
                    glMatrixMode(GL_TEXTURE);
                    glPushMatrix();
                    glScalef(data[n].reflect_texture->width,
data[n].reflect_texture->height, 1.0);
                }
// Рисуем объект (второй проход)
                glDrawElements(GL_TRIANGLES, data[n].numfaces*3,
GL_UNSIGNED_INT, data[n].indices);

// Восстанавливаем прежние настройки OpenGL
                if (data[n].reflect_texture->pTextureObj->
target==GL_TEXTURE_RECTANGLE_NV)
                {
// Восстанавливаем матрицу текстуры
                    glPopMatrix();
                    glMatrixMode(GL_MODELVIEW);
                }
                data[n].reflect_texture->pTextureObj->disable();
                glDisable(GL_TEXTURE_GEN_S);
                glDisable(GL_TEXTURE_GEN_T);
                glDisable(GL_BLEND);
                glDepthMask(true);
                glDepthFunc(GL_LESS);
        }
...
}
```

Остальные изменения в библиотеке ASE Reader довольно тривиальны, поэтому мы не будем их рассматривать.

Для того чтобы оценить производительность современных ускорителей при работе с NPOTD-текстурами, я переписал пример Ex12 из *главы 4* с использованием новой версии библиотеки ASE Reader (Ex05). Было создано три варианта сцены долины *(см. разд. 4.4.5)* с использованием текстур размером 512×512, 768×512 и 1024×512. Тестирование проводилось в двух режимах: автоматическом `TR_AUTO` и режиме форсированного использования NPOTD-текстур `TR_ALWAYS`.

Кроме того, в пример Ex05 была добавлена новая команда `texture_rectangle_mode`, позволяющая выбирать режим работы с NPOTD-текстурами без перекомпиляции программы:

```
// Диспетчер текстур из файла ase.cpp
extern CTextureManager TextureManager;
...
```

```
//  Обработчик консольных команд
bool CmdProc(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
...
// Если консольная команда "texture_rectangle_mode"
       if (Cmd=="texture_rectangle_mode")
       {
// Если нет параметров, то это ошибка
              if (Params.size()==0)
              {
                     console->add("Not enough parameters");
                     return false;
              }
// Если параметр "/?", выводим информацию о команде
              if (Params[0]=="/?")
              {
                     console->add("Sets mode of using NPOTD-textures");
                     console->add("");
                     console->add("texture_rectangle_mode <mode>");
                     console->add("where");
                     console->add("  <mode> is \"NEVER\" if NPOTD never
used,");
                     console->add("            \"AUTO\" if NPOTD used if
necessary, and");
                     console->add("            \"ALWAYS\" if NPOTD
always used");
                     return true;
              }

// Если параметр-строка NEVER, устанавливаем режим TR_NEVER
              if (Params[0]=="NEVER")
                     TextureManager.TextureRectangleMode=TR_NEVER;
              else
// Если параметр-строка AUTO, устанавливаем режим TR_AUTO
                     if (Params[0]=="AUTO")
                            TextureManager.TextureRectangleMode=TR_AUTO;
                     else
// Если параметр-строка ALWAYS, устанавливаем режим TR_ALWAYS
                            if (Params[0]=="ALWAYS")
                          TextureManager.TextureRectangleMode=TR_ALWAYS;
```

```
                                else
                                {
// Иначе - ошибка
                                        console->add("Invalid parameter");
                                        return false;
                                }
                return true;
        }

        return false;
}...
int main(int argc, char* argv[])
{
...
// Регистрируем новую консольную команду
        console.addCmd("texture_rectangle_mode", "sets mode of using
NPOTD-textures", 0, CmdProc);
...
}
```

Результаты измерений приведены в табл. 6.1.

Таблица 6.1. *Производительность современных GPU при работе с NPOTD-текстурами*

Ускоритель \ Текстура	512 × 512 POTD MIPMAP[1]	512 × 512 POTD[2]	512 × 512 NPOTD[3]	768 × 512 NPOTD	1024 ×512 POTD MIPMAP	1024 × 512 POTD	1024 × 512 NPOTD
ATI Radeon 9700 Pro	106	100	100	93	102	90	90
NVIDIA GeForce FX 5800 Ultra	91	55	34	28	87	43	23

[1] POTD MIPMAP означает, что использовалась целевая текстура GL_TEXTURE_2D с режимом фильтрации GL_LINEAR_MIPMAP_LINEAR.

[2] POTD означает, что использовалась целевая текстура GL_TEXTURE_2D с режимом фильтрации GL_LINEAR.

[3] NPOTD означает, что использовалась целевая текстура GL_TEXTURE_RECTANGLE_NV с режимом фильтрации GL_LINEAR.

Анализ результатов начнем с GeForce FX. Как видно из табл. 6.1, производительность ускорителя при работе с NPOTD-текстурами значительно ниже

его производительности при работе с POTD-текстурами на аналогичном разрешении. При этом разница в производительности может достигать двух раз. Видно, что с 512 × 512 NPOTD-текстурой ускоритель работает на 25% медленнее, чем с 1024 × 512 POTD-текстурой. Но в действительности ситуация еще хуже. Дело в том, что NPOTD-текстуры не поддерживают Mipmap-фильтрацию, в то время как в девяносто девяти процентах случаев POTD-текстуры используют Mipmap-фильтрацию. В итоге, реальная разница в производительности между NPOTD-текстурами и POTD-текстурами может достигать четырех раз!

С ускорителем Radeon 9700 Pro все не так просто. При внимательном рассмотрении таблицы можно заметить две странности:

- при использовании "псевдо-NPOTD" текстур 512 × 512 и 1024 × 512 производительность NPOTD и POTD абсолютно одинакова;
- при использовании "настоящих" NPOTD-текстур 768 × 512 производительность Radeon 9700 Pro ниже, чем при использовании "псевдо-NPOTD" текстур большего размера (1024 × 512).

Из этого напрашивается следующий вывод. Если NPOTD-текстура в действительности является POTD-текстурой, то драйвер ATI автоматически распознает такую ситуацию, после чего начинает работать с этой текстурой, как с POTD-текстурой. В результате производительность Radeon 9700 Pro при использовании "псевдо-NPOTD" текстур 512 × 512 и 1024 × 512 не отличается от обычных POTD-текстур. А вот использование "настоящих" NPOTD-текстур приводит к незначительному снижению производительности, которое, впрочем, не является столь "драматическим", как в случае с GeForce FX.

Как видно, использование NPOTD-текстур в приложении приводит к снижению производительности. Но это еще не самое худшее: отсутствие поддержки NPOTD-текстурами Mipmap-фильтрации привело к появлению множества артефактов. В целом качество изображения тоже ухудшилось.

Из всего вышесказанного следует простой вывод: поддержка NPOTD-текстур библиотекой ASE Reader в целом бессмысленна. Поэтому в следующих главах будет использоваться прошлая версия библиотеки ASE Reader без поддержки NPOTD-текстур — это позволит нам уменьшить размер примеров и сделать их более понятными.

Заключение

NPOTD-текстуры часто бывают полезны для рисования пользовательского интерфейса и наложения текстурных карт, не преобразующихся в POTD-текстуры без потери качества. Тем не менее следует избегать неоправданного использования NPOTD-текстур, заменяя их, по возможности, POTD-текстурами.

Глава 7

Проверка видимости объектов с использованием расширений HP_occlusion_test и NV_occlusion_query

Как известно, в реальных программах наблюдателю в каждый момент времени видна лишь небольшая часть объектов сцены, т. к. большинство объектов либо находятся за пределами экрана, либо закрыты объектами, находящимися на переднем плане. Если приложение будет рисовать все объекты сцены, включая невидимые, то использование ресурсов 3D-ускорителя не будет оптимальным.

В качестве примера такого приложения мы рассмотрим программу, выводящую на экран шесть чайников, вращающихся вокруг зеркала (рис. 7.1). Зеркало прикреплено к деревянной плоскости, поэтому окружающая среда отражается только одной из его сторон (рис. 7.2).

Для того чтобы нарисовать такое зеркало, мы должны выполнить следующие действия:

1. Нарисовать деревянную плоскость со смещением в буфере глубины + 2. Если не выполнить смещение, то при рисовании собственно зеркала (третий этап) возникнут артефакты, связанные с недостаточной точностью буфера глубины (рис. 7.3).
2. Нарисовать сцену с "летающими чайниками".
3. Выключить расчет освещения, включить тест шаблона и нарисовать плоскость зеркала. Цвет плоскости зеркала должен совпадать с цветом фона. В буфер шаблона пикселов, которыми нарисована плоскость, должна быть занесена единица.
4. Включить расчет освещения и переключить буфер шаблона в режим `glStencilFunc(GL_EQUAL, 1, 0xffff)`. Теперь отражение (девятый этап) будет рисоваться только в зеркале.

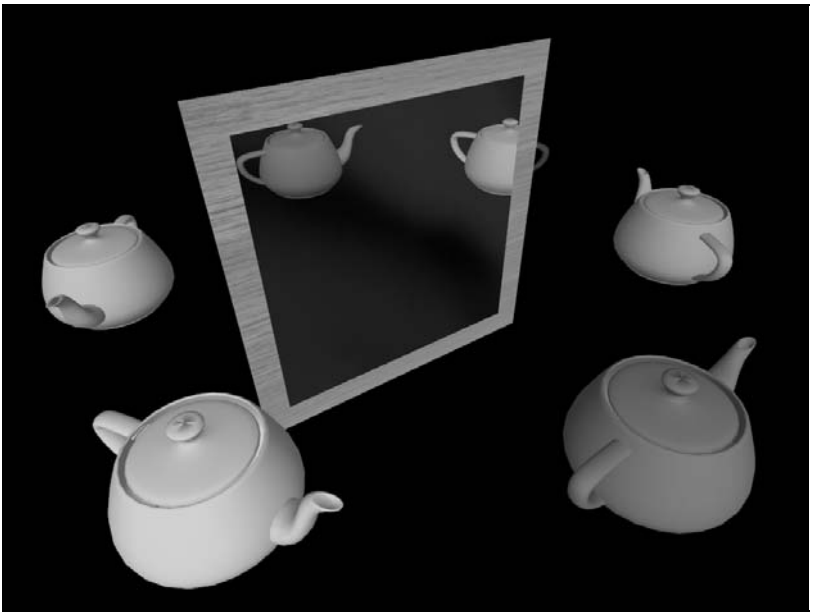


Рис. 7.1. Чайники, вращающиеся вокруг зеркала. Вид спереди

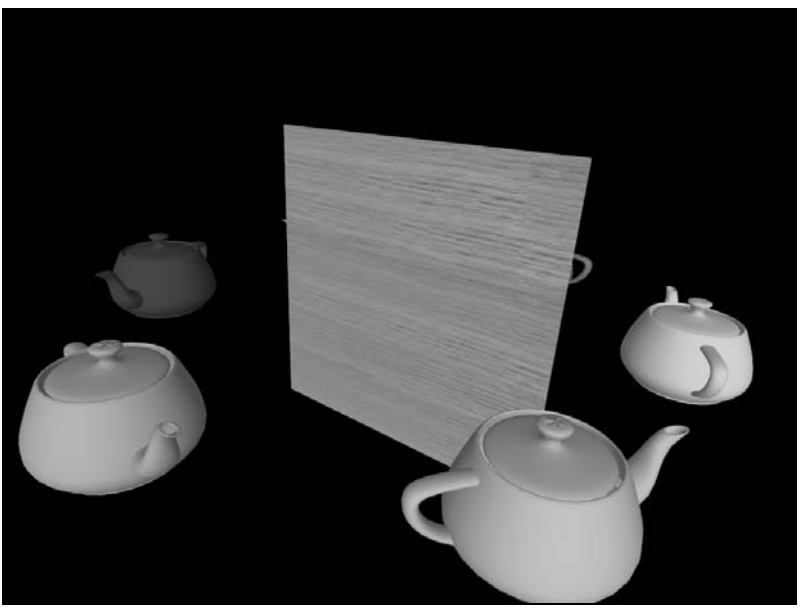


Рис. 7.2. Чайники, вращающиеся вокруг зеркала. Вид сзади

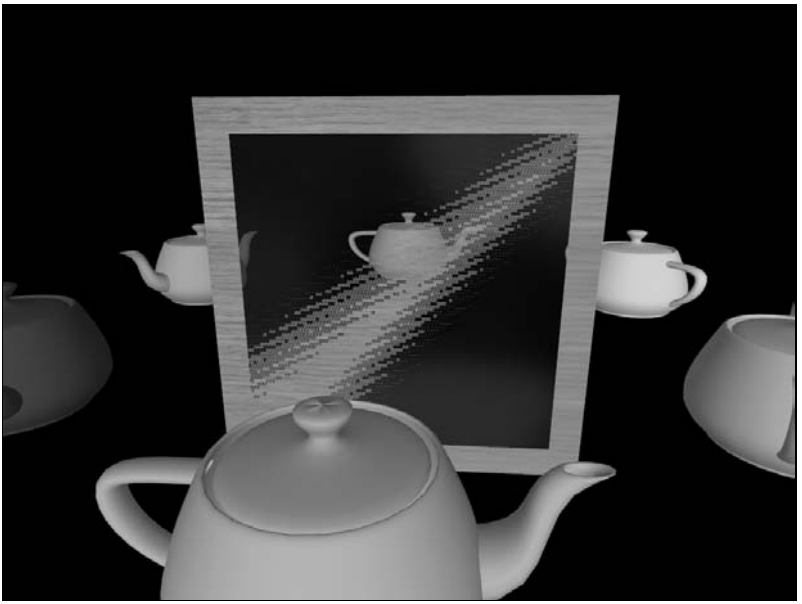


Рис. 7.3. Артефакты,
связанные с недостаточной точностью буфера глубины

5. Умножить текущую матрицу модели на матрицу отражения относительно плоскости зеркала.
6. Задать плоскость отсечения $0 \times x + 0 \times y + 1 \times z + 0 = 0$. В противном случае, отражение (девятый этап) будет видно не только за зеркалом, но и перед ним.
7. Очистить буфер глубины. Если этого не сделать, то отражение, которое будет создано на девятом этапе, не пройдет тест глубины.
8. Изменить режим лицевых граней на противоположный (команда `glFrontFace(GL_CCW)`). Это связано с тем, что умножение матрицы модели на матрицу отражения (шестой этап) делает нелицевые грани лицевыми и наоборот.
9. Нарисовать отражение (нарисовать сцену с летающими чайниками).
10. Для имитации стекла, нарисовать поверх зеркала полупрозрачную плоскость с текстурой. Для имитации неровности зеркала можно включить режим генерации сферических текстурных координат.

В листинге 7.1 приведен исходный текст программы, рисующей сцену с чайниками по приведенному выше алгоритму (Ex01).

Листинг 7.1

```
#define __FULLSCREEN

#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <gl/glut.h>
#include <gl/glext.h>

#include "glh_glut_ext.h"
#include "nv_util_ext.h"

// Размеры окна (в оконном режиме)
const int WinWidth=640;
const int WinHeight=480;

// Текстурный объект текстуры дерева
tex_object_2D WoodTexture;
// Текстурный объект поверхности зеркала
tex_object_2D ChromicTexture;
// Изображение текстуры дерева
tga::tgaImage *pWoodTextureImage=0;
// Изображение текстуры стекла
tga::tgaImage *pChromicTextureImage=0;
// Дисплейный список плоскости
display_list plane;
// Дисплейный список чайника
display_list teapot;
// Интеракторы
glut_console console;
glut_simple_user_interface user(&console);
glut_callbacks cb;
glut_swapbuffers swapbuffers;
// Время запуска программы
GLuint StartTick;
// Заголовок окна (в оконном режиме)
string Title("GLHE Demo");
```

```
// Структура с информацией об одном чайнике
struct Teapot
{
// Цвет чайника
       vec3 color;
// Позиция чайника
       vec3 pos;
// Угол поворота чайника (вокруг оси y)
       float rotate;
};
// Количество чайников
const TeapotCount=6;

// Массив с информацией о чайниках
Teapot Teapots[TeapotCount];
// Матрица отражения
GLfloat Matrix[4][4];
// Функция расчета матрицы отражения. Взята из [19].
// В качества параметров принимает массив (m) 4 × 4(в который будет
// занесена рассчитанная матрица модели), точку, лежащую на плоскости (p)
// и перпендикуляр к плоскости (v)
void MirrorMatrix(GLfloat m[4][4], GLfloat p[3], GLfloat v[3])
{
       GLfloat dot=p[0]*v[0]+p[1]*v[1]+p[2]*v[2];

       m[0][0]=1-2*v[0]*v[0];
       m[1][0]=-2*v[0]*v[1];
       m[2][0]=-2*v[0]*v[2];
       m[3][0]=2*dot*v[0];

       m[0][1]=-2*v[1]*v[0];
       m[1][1]=1-2*v[1]*v[1];
       m[2][1]=-2*v[1]*v[2];
       m[3][1]=2*dot*v[1];

       m[0][2]=-2*v[2]*v[0];
       m[1][2]=-2*v[2]*v[1];
       m[2][2]=1-2*v[2]*v[2];
       m[3][2]=2*dot*v[2];
```

```
        m[0][3]=0;
        m[1][3]=0;
        m[2][3]=0;
        m[3][3]=1;
}
// Начальная инициализация программы
bool Init()
{
// Настраиваем начальные параметры OpenGL
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_COLOR_MATERIAL);
        glEnable(GL_NORMALIZE);
        glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, 1);

        vec4 diffuse_color(0.7, 0.7, 0.7, 1);
        vec4 ambinet_color=vec4_one-diffuse_color+vec4(0, 0, 0, 1);

        glLightfv(GL_LIGHT0, GL_DIFFUSE, &diffuse_color[0]);
        glLightfv(GL_LIGHT0, GL_AMBIENT, &ambinet_color[0]);

        glClearColor(0,0,0,0);
// Загружаем текстуру из дерева
        pWoodTextureImage=read("..\\oak1.tga");
        if (!pWoodTextureImage)
        {
                console.add("Cannot load file ..\\oak1.tga");
                console.exit(-1);
                return false;
        }
// Загружаем текстуру зеркала
        pChromicTextureImage=read("..\\refmap.jpg");
        if (!pChromicTextureImage)
        {
                console.add("Cannot load file ..\\refmap.jpg");
                console.exit(-1);
                return false;
        }
```

```
// Создаем текстурный объект дерева и настраиваем его параметры
       WoodTexture.bind();
       WoodTexture.parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
       WoodTexture.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
       texImage2D(pWoodTextureImage, &WoodTexture);


// Создаем текстурный объект стекла и настраиваем его параметры
       ChromicTexture.bind();
       ChromicTexture.parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
       ChromicTexture.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
       texImage2D(pChromicTextureImage, &ChromicTexture);


// Создаем дисплейный список плоскости
       plane.new_list(GL_COMPILE);
              glBegin(GL_QUADS);
                     glNormal3f(0, 0 ,1);
                     glTexCoord2f(0, 1);
                     glVertex2f(-1, 1);

                     glTexCoord2f(0, 0);
                     glVertex2f(-1, -1);

                     glTexCoord2f(1, 0);
                     glVertex2f(1, -1);

                     glTexCoord2f(1, 1);
                     glVertex2f(1, 1);
              glEnd();
       plane.end_list();

// Заполняем массив с информацией о чайниках
       for (unsigned int i=0; i<TeapotCount; i++)
       {
              Teapots[i].pos.y=0;
              do
              {
                     Teapots[i].color.x=nv_random()/2.0+0.5;
```

```
                    Teapots[i].color.y=nv_random()/2.0+0.5;
                    Teapots[i].color.z=nv_random()/2.0+0.5;
                }
// Если цвет чайника получился недостаточно ярким, то генерируем
// цвет еще раз
                while
(Teapots[i].color.x+Teapots[i].color.y+Teapots[i].color.z<1.0);
        }

// Создаем текстурный объект чайника
        teapot.new_list(GL_COMPILE);
                glutSolidTeapot(1.0);
        teapot.end_list();
// Рассчитываем матрицу отражения
        GLfloat Point[3]={0, 0, 0};
        GLfloat Up[3]={0, 0, 1};
        MirrorMatrix(Matrix, Point, Up);
// Запоминаем момент старта программы
        StartTick=GetTickCount();

        return true;
}

// Рисуем сцену с чайниками. В качестве параметра принимаем режим лицевых
// граней (GL_CW или GL_CCW)
void DrawScene(GLenum FaceMode)
{
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        glDisable(GL_CULL_FACE);
        glFrontFace(FaceMode);
// Перебираем все чайники
        for (unsigned int i=0; i<TeapotCount; i++)
        {
                glPushMatrix();
// Задаем положение и цвет чайника
                glTranslatef(Teapots[i].pos.x, Teapots[i].pos.y, Tea-
pots[i].pos.z);
                glColor3fv(&Teapots[i].color.x);
                glRotatef(Teapots[i].rotate, 0, 1.0, 0);
```

```
// Рисуем чайник
                teapot.call_list();
                glPopMatrix();
        }
        glPopAttrib();
}

// Вывод сцены на экран
void Display()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
GL_STENCIL_BUFFER_BIT);
// Рисуем деревянную плоскость
        glEnable(GL_POLYGON_OFFSET_FILL);
        glPolygonOffset(1, 2);
        glPushMatrix();
        glScalef(3, 3, 1);
        WoodTexture.bind();
        WoodTexture.enable();
        glColor3f(1.0, 1.0, 1.0);
        plane.call_list();
        WoodTexture.disable();
        glPopMatrix();
        glDisable(GL_POLYGON_OFFSET_FILL);
// Рисуем сцену с чайниками
        DrawScene(GL_CW);
// Готовимся к рисованию зеркала
        glEnable(GL_STENCIL_TEST);
        glStencilFunc(GL_ALWAYS, 1, 0xffff);
        glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);

        glPushMatrix();
        glScalef(2.5, 2.5, 1);
        glEnable(GL_CULL_FACE);
        glDisable(GL_LIGHTING);
        glColor3f(0, 0, 0);
// Рисуем зеркало
        plane.call_list();
        glEnable(GL_LIGHTING);
```

```
        glDisable(GL_CULL_FACE);
        glPopMatrix();
// Рисуем отражение только в зеркале
        glStencilFunc(GL_EQUAL, 1, 0xffff);
// Очищаем буфер глубины (чтобы отражение рисовалось и под зеркалом)
        glClear(GL_DEPTH_BUFFER_BIT);
        glPushMatrix();
// Умножаем матрицу модели на матрицу отражения
        glMultMatrixf(&Matrix[0][0]);
// Устанавливаем плоскость отсечения (чтобы отражение не рисовалось перед
// зеркалом)
        GLdouble plane0[4]={0, 0, 1, 0};
        glClipPlane(GL_CLIP_PLANE0, &plane0[0]);
        glEnable(GL_CLIP_PLANE0);
// Рисуем отражение (сцену с чайниками)
        DrawScene(GL_CCW);

        glDisable(GL_CLIP_PLANE0);

        glPopMatrix();
// Создаем эффект неровного стекла при помощи рисования полупрозрачной
// сферической текстуры
        glEnable(GL_BLEND);
        glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        ChromicTexture.bind();
        ChromicTexture.enable();
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
        glPushMatrix();
        glScalef(2.5, 2.5, 1);
        glEnable(GL_CULL_FACE);
        glEnable(GL_LIGHTING);
        glColor4f(1, 1, 1, 0.2);
        glEnable(GL_TEXTURE_GEN_S);
        glEnable(GL_TEXTURE_GEN_T);
        glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
        glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
        plane.call_list();
        glDisable(GL_TEXTURE_GEN_S);
```

```
        glDisable(GL_TEXTURE_GEN_T);
        glEnable(GL_LIGHTING);
        glDisable(GL_CULL_FACE);
        glPopMatrix();

        ChromicTexture.disable();
        glDisable(GL_BLEND);

        glDisable(GL_STENCIL_TEST);
}
// Расчет положения чайников в данный момент времени
void Idle()
{
        float t=float(GetTickCount()-StartTick)/3000.0;
        for (unsigned int i=0; i<TeapotCount; i++)
        {
// Расчет положения чайника
                Teapots[i].pos.x=5.0*sin(t);
                Teapots[i].pos.z=5.0*cos(t);
// Расчет угла поворота чайника вокруг оси y (в градусах)
                Teapots[i].rotate=(t*180.0/nv_pi);
                t+=2.0*nv_pi/float(TeapotCount);
        }
        glutPostRedisplay();
}
// Удаляем "мусор" при выходе из программы
void onExit()
{
        clear(pWoodTextureImage);
        clear(pChromicTextureImage);
}
int main()
{
// Инициализация GLUT
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGBA |
GLUT_STENCIL);
        glutInitWindowSize(WinWidth, WinHeight);
        glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
```

```
glutGet(GLUT_SCREEN_HEIGHT)-WinHeight)/2);
#ifdef __FULLSCREEN
       glutGameModeString("1024x768@85");
       glutEnterGameMode();
#else
       glutCreateWindow(Title.c_str());
#endif
       atexit(onExit);
// Инициализация интеракторов библиотеки GLH
       glut_helpers_initialize();

       user.user_mouse.dolly.dolly[2]=-8;
       user.bAutoOffIdle=false;

       cb.display_function=Display;
       cb.idle_function=Idle;

       glut_add_interactor(&user);
       glut_add_interactor(&cb);
       glut_add_interactor(&console);
       glut_add_interactor(&swapbuffers);

       glut_idle(true);
// Если инициализация прошла успешно, выполняем скрипт из файла
// autoexec.cs
       if (Init())
              console.processCmd("exec autoexec.cs");

       glutMainLoop();

}
```

Я измерил производительность программы в двух режимах — вид спереди (FRONT), когда на экране видно около 60—70% всех чайников сцены (рис. 6.4), и вид сзади (BACK), когда ни один из чайников не виден на экране (рис. 6.5).

Результаты измерений приведены в табл. 7.1 (см. стр. 417).

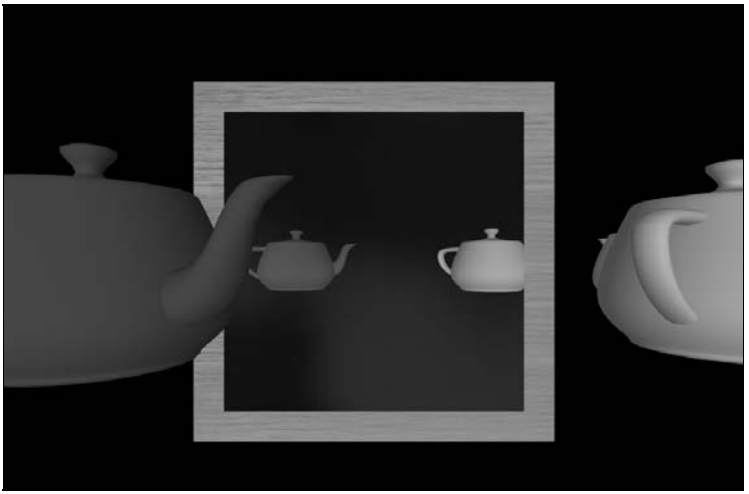


Рис. 7.4. Вид сцены спереди (FRONT)

Рис. 7.5. Вид сцены сзади (BACK)

Технические подробности

Для измерения производительности программы из примера Ex06 на компьютерах с различной конфигурацией необходим механизм сохранения положений наблюдателя и быстрого переключения между ними. Для этого я добавил в библиотеку GLHE четыре новые консольные команды: `user_pan`, `user_dolly`, `user_trackball`, `save_user_lookat`. Команды `user_pan`,

`user_dolly`, `user_trackball` позволяют настроить параметры подобъектов `pan`, `dolly` и `trackball` интерактора `glut_simple_user_mouse`, т. е. при помощи этих трех команд можно задать любое положение наблюдателя. Команда `save_user_lookat` позволяет записать скрипт для восстановления текущего положения наблюдателя в заданный файл. Например, для положения наблюдателя, показанного на рис. 6.8, команда `save_user_lookat lookat.cs` создаст файл lookat.cs со следующим содержанием:

```
user_pan 0.140000 -0.010000 0.000000
user_dolly 0.000000 0.000000 -3.570004
user_trackball 0.000541 -0.999859 -0.015257 0.006944
```

Теперь для того, чтобы восстановить положение наблюдателя, достаточно ввести в консоли команду `exec lookat.cs`.

Как видно из результатов измерений, пример Ex06 работает очень медленно. Это связано с тем, что команда рисования чайника `glutSolidTeapot` имеет очень низкую производительность. А наша программа рисует все 12 чайников даже в том случае, когда на экране не виден ни один из них. В результате программа в режимах `FRONT` и `BACK` показывает практически одинаковую производительность. В то же время если бы программа не рисовала невидимые чайники, то в режиме `BACK` ее производительность была бы в несколько раз выше (см. табл. 7.1, стр. 417) (колонка "`BACK` без чайников").

Единственный выход из данной ситуации — проверять видимость каждого чайника перед выводом его на экран. Но проверка видимости на экране чайника, состоящего из большого количества полигонов, очень непростая задача, решение которой может занять даже большее время, чем само рисование чайника. Однако эту задачу можно сильно упростить, если заменить проверку видимости чайника проверкой видимости *прямоугольной оболочки* чайника. Прямоугольная оболочка 3-мерного объекта — это прямоугольный параллелепипед минимального размера, в который данный объект может быть вписан.

7.1. Построение прямоугольной оболочки объекта

При использовании моделей формата ASE для нахождения прямоугольной оболочки модели можно использовать метод `CAseModel::GetExtents(vec3f &minExtent, vec3f &maxExtent)` библиотеки ASE Reader. Вообще, определение прямоугольной оболочки — это очень простая задача, которая сводится к последовательному просмотру всех вершин объекта и выявлению максимального и минимального значений координат x, y, z.

К сожалению, наш чайник не является моделью формата ASE. А библиотека GLUT не предоставляет нам средств для нахождения прямоугольных оболо-

чек своих объектов. Поэтому придется изобретать свой метод для построения прямоугольной оболочки чайника. Существуют два способа решения этой задачи:

- Простой, но неточный. Определить размеры прямоугольной оболочки "на глаз".
- Сложный, но точный: визуализировать чайник в режиме обратной связи `GL_FEEDBACK` и затем, на основании полученной информации, определить размеры его прямоугольной оболочки.

Мы будем использовать метод, позволяющий точно определить прямоугольную оболочку объекта. Это позволит программе более эффективно отсекать невидимые объекты, что, в свою очередь, уменьшит количество ложных построений прямоугольной оболочки (когда прямоугольная оболочка видна на экране, а объект — нет).

Суть этого метода состоит в следующем. Мы должны в режиме обратной связи с использованием ортогональной проекции нарисовать на экране объект, повернутый к наблюдателю боком. То есть плоскость X0Y в системе координат объекта должна быть параллельна плоскости экрана (рис. 7.6).

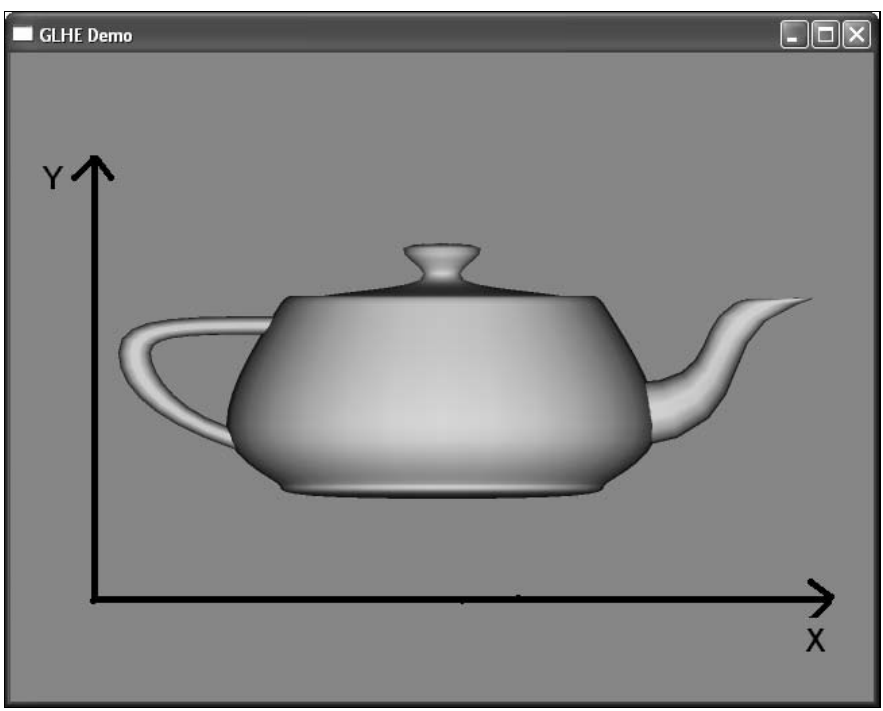


Рис. 7.6. Плоскость X0Y в системе координат объекта параллельна плоскости экрана. Ортогональная проекция

Анализируя буфер обратной связи, мы сможем определить минимальное и максимальное значения координат проекций вершин на экране. Так как проекция является ортогональной, а плоскость экрана совпадет с плоскостью X0Y в системе координат объекта, то мы сможем легко определить максимальное и минимальное значение координат X и Y. Минимальное и максимальное значения координаты Z определяются аналогичным образом. Для этого надо повторно нарисовать объект в режиме обратной связи, повернув его таким образом, чтобы плоскость Y0Z (или X0Z) в системе координат объекта была параллельна экрану (рис. 7.7).

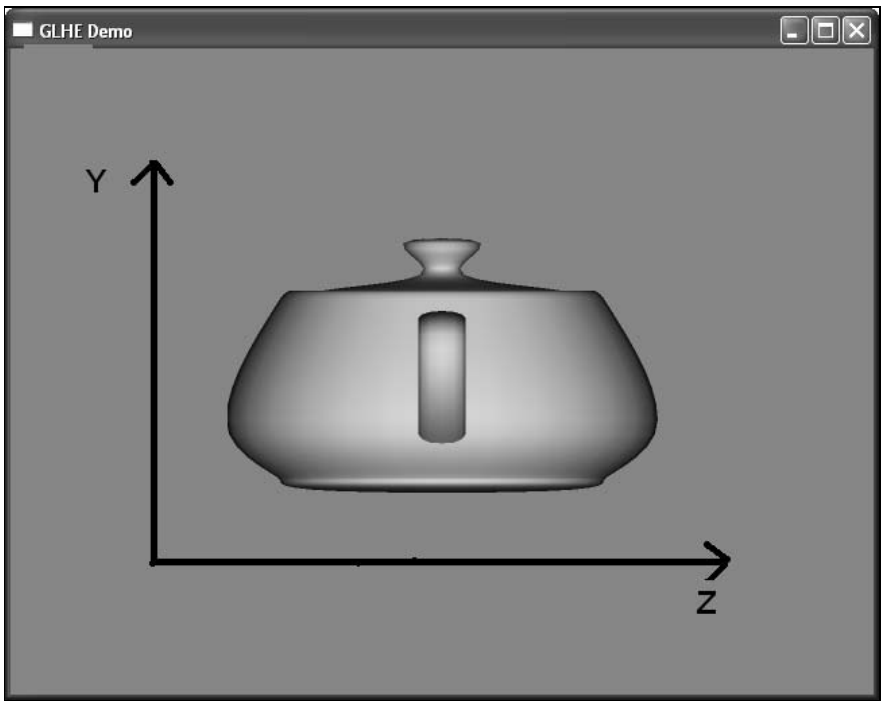


Рис. 7.7. Плоскость Y0Z
в системе координат объекта (чайника)
параллельна плоскости экрана. Ортогональная проекция

Однако следует помнить, что у этого метода есть несколько "подводных камней", связанных с особенностями режима обратной связи OpenGL.

- Объект должен умещаться на экране целиком. В противном случае OpenGL автоматически "отсечет" вершины, которые не уместились на экране, в результате чего программа не сможет правильно определить размеры объекта.

- Буфер обратной связи, в который OpenGL будет заносить информацию об объекте, должен иметь достаточный размер. Это обусловлено тем, что в нем будет содержаться информация обо всех примитивах объекта. А поскольку количество примитивов в объекте заранее неизвестно, то необходимо предусмотреть динамическую настройку буфера обратной связи под конкретный объект.

Рисование прямоугольной оболочки должно занимать как можно меньше времени, поэтому лучше всего поместить ее в дисплейный список. Но поскольку перед созданием дисплейного списка необходимо провести визуализацию в режиме обратной связи `GL_FEEDBACK`, создание дисплейного списка должно производиться при выводе изображения на экран (т. е. внутри функции `Display` или аналогичной функции). Для этого идеально подходит класс `lazy_build_display_list` библиотеки GLH, который рассматривался в *разд. 2.4.2*.

Но и здесь есть подвох. Дело в том, что перед рисованием прямоугольной оболочки мы должны визуализировать объект в режиме обратной связи. Следовательно, этот код придется вынести за пределы функции обратной связи класса `lazy_build_display_list` (режим `GL_FEEDBACK` не работает внутри дисплейного списка). Это приведет к усложнению программы.

Однако если приложить небольшие усилия, код определения размеров чайника можно поместить в функцию обратной связи класса `lazy_build_display_list`. Для этого код функции обратной связи класса `lazy_build_display_list` должен выполнять следующие действия:

1. Запомнить идентификатор дисплейного списка, который создается в текущий момент времени.
2. Завершить создание этого списка командой `glEndList`, а затем удалять его.
3. Нарисовать две проекции объекта в режиме обратной связи и определить его прямоугольную оболочку.
4. Создать дисплейный список с идентификатором, сохраненным на шаге 1.
5. Нарисовать прямоугольную оболочку объекта.

В листинге 7.2 приведены наиболее существенные фрагменты программы, рисующей на экране прямоугольную оболочку чайника (Ex02).

Листинг 7.2

```
...
// Дисплейный список чайника
display_list list0;
```

```
// Дисплейный список прямоугольной оболочки объекта
lazy_build_display_list bound_box_list0;
...
// Выбираем одну пару координат x, y (или z, y) из буфера обратной связи,
// переводим их из системы экранных координат в систему координат объекта
// и сравниваем их значения с координатами векторов min и max.
// i — текущая позиция в буфере обратной связи
// buf — буфер обратной связи
// min, max — минимальные и максимальные значения координат объекта
// viewport — текущее видовое окно
// k — если плоскость экрана параллельна плоскости X0Y в системе
// координат объекта, то k=0. Если же плоскость экрана параллельна
// плоскости Y0Z в системе координат объекта, то k=1
void minmax(int& i, GLfloat* buf, vec3& min, vec3& max, GLint view-
port[4], int k)
{
       float tmp;
// Если плоскость X0Y
       if (k==0)
       {
// Переводим координату X из оконных координат в систему координат
// объекта
              tmp=(buf[i]-viewport[0])/float(viewport[2]-
viewport[0])*20.0-10.0;
// Сравниваем ее с min.x и max.x
              if (tmp<min.x)
                     min.x=tmp;
              if (tmp>max.x)
                     max.x=tmp;
// Переходим к следующему элементу буфера обратной связи (координата y)
              i++;


// Переводим координату Y из оконных координат в систему координат
// объекта
              tmp=(buf[i]-viewport[1])/float(viewport[3]-
viewport[1])*20.0-10.0;
              if (tmp<min.y)
                     min.y=tmp;
              if (tmp>max.y)
                     max.y=tmp;
```

```
                i++;
        }
        else
// Если плоскость Y0Z
        {
// Переводим координату Z из оконных координат в систему координат
// объекта
                tmp=(buf[i]-viewport[0])/float(viewport[2]-
viewport[0])*20.0-10.0;
                if (tmp<min.z)
                        min.z=tmp;
                if (tmp>max.z)
                        max.z=tmp;
                i+=2;
        }
}
// Создаем дисплейный список прямоугольной оболочки объекта
void BuildList()
{
        GLint list;
// Получаем идентификатор дисплейного списка, который создается в данный
// момент времени
        glGetIntegerv(GL_LIST_INDEX, &list);
// Завершаем создание дисплейного списка и удаляем его
        glEndList();
        glDeleteLists(list, 1);
// Устанавливаем начальные значения min и max
        vec3 min, max;
        min.x=FLT_MAX;
        min.y=FLT_MAX;
        min.z=FLT_MAX;
        max.x=-FLT_MAX;
        max.y=-FLT_MAX;
        max.z=-FLT_MAX;

// Настраиваем параметры ортогональной проекции
        glPushMatrix();
        glLoadIdentity();
        glMatrixMode(GL_PROJECTION);
```

```
        glPushMatrix();
        glLoadIdentity();
        glOrtho(-10.0, 10.0, -10.0, 10.0, -10.0, 10.0);
        glMatrixMode(GL_MODELVIEW);
        glPushAttrib(GL_ALL_ATTRIB_BITS);
// Отключаем расчет освещения
        glDisable(GL_LIGHTING);
// Получаем параметры видового окна
        GLint viewport[4];
        glGetIntegerv(GL_VIEWPORT, &viewport[0]);

// Начальное значение буфера обратной связи
        unsigned int buf_size=1024;
// Адрес буфера глубины
        GLfloat* buf=0;
// Реальный размер буфера глубины
        GLint n;
// Счетчик полигонов в объекте
        int polygon_count=0;
// Сначала рисуем проекцию плоскости X0Y на экран (k=0), а затем Y0Z
// (k=1)
        for (int k=0; k<2; k++)
        {
                do
                {
// Если буфер обратной связи уже был создан — удаляем его
                        if (buf!=0)
                        {
                                delete[] buf;
// Увеличиваем размер буфера обратной связи в два раза
                                buf_size*=2;
                        }
// Выделяем память для буфера обратной связи
                        buf=new GLfloat[buf_size];
// Создаем буфер обратной связи
                        glFeedbackBuffer(buf_size, GL_2D, buf);
// Переключаемся в режим обратной связи
                        glRenderMode(GL_FEEDBACK);
```

```
// Если необходимо, поворачиваем объект
                    if (k==1)
                           glRotatef(90, 0, 1, 0);
// Рисуем объект
                    list0.call_list();
// Переключаемся в обычный режим
                    n=glRenderMode(GL_RENDER);
             }
// Если размер буфера оказался недостаточным, то увеличиваем его в два
// раза и повторяем попытку
             while (n<0);

// Устанавливаем текущую позицию в начало буфера обратной связи
             int i=0;
             int vertex_count, j;
// Пока не достигли конца буфера обратной связи
             while (i!=n)
             {
// Получаем идентификатор текущего примитива
                    int taken=buf[i];
                    i++;
                    switch (taken)
                    {
// Если примитив — точка
                    case GL_POINT_TOKEN:
// Сравниваем ее координаты (не оконные!) с min и max
                           minmax(i, buf, min, max, viewport, k);
                           break;
// Если примитив — полигон
                    case GL_POLYGON_TOKEN:
// Получаем количество вершин полигона
                           vertex_count=buf[i];
// Если идет первый проход цикла, увеличиваем счетчик полигонов
                           if (k==0)
                                  polygon_count+=vertex_count-2;
                           i++;
// Сравниваем координаты вершин полигона (не оконные!) с min и max
                           for (j=0; j<vertex_count; j++)
```

```
                                        minmax(i, buf, min, max, viewport, k);
                                break;
// Если какой-нибудь другой примитив (например, линия или битовая карта)
                        default:
// Сообщаем об необходимости доработки программы.
// Дело в том, что чайник использует только примитивы-полигоны, поэтому
// поддержка остальных примитивов приведет к усложнению и без того
// большой программы
                                console.add("Unknow token. Program required
modify");
                                glNewList(list, GL_COMPILE);
                                delete[] buf;
                                console.exit(-1);
                                return;
                                break;
                        }
                }
        }
// Удаляем буфер обратной связи
        delete[] buf;
// Восстанавливаем параметры программы
        glPopAttrib();
        glMatrixMode(GL_PROJECTION);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
        glPopMatrix();

        char str[1024];
// Выводим информацию о количестве полигонов
        sprintf(str, "polygons count=%d", polygon_count);
        console.add(str);
// Если буфер обратной связи оказался пустой, то нет необходимости
// рисовать прямоугольную оболочку объекта
        if (n=0)
        {
                glNewList(list, GL_COMPILE);
                return;
        }
```

```
// Выводим информацию о минимальном и максимальном значениях координат
// объекта
       sprintf(str, "min=(%f, %f, %f)", min.x, min.y, min.z);
       console.add(str);
       sprintf(str, "max=(%f, %f, %f)", max.x, max.y, max.z);
       console.add(str);

// Создаем дисплейный список с прямоугольной оболочкой объекта
       glNewList(list, GL_COMPILE);
              glPushAttrib(GL_ALL_ATTRIB_BITS);
              glDisable(GL_LIGHTING);
              glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
              glBegin(GL_QUADS);
                     glVertex3f(min.x, min.y, max.z);
                     glVertex3f(max.x, min.y, max.z);
                     glVertex3f(max.x, max.y, max.z);
                     glVertex3f(min.x, max.y, max.z);

                     glVertex3f(max.x, min.y, min.z);
                     glVertex3f(min.x, min.y, min.z);
                     glVertex3f(min.x, max.y, min.z);
                     glVertex3f(max.x, max.y, min.z);

                     glVertex3f(min.x, min.y, min.z);
                     glVertex3f(max.x, min.y, min.z);
                     glVertex3f(max.x, min.y, max.z);
                     glVertex3f(min.x, min.y, max.z);

                     glVertex3f(min.x, max.y, min.z);
                     glVertex3f(min.x, max.y, max.z);
                     glVertex3f(max.x, max.y, max.z);
                     glVertex3f(max.x, max.y, min.z);

                     glVertex3f(max.x, min.y, min.z);
                     glVertex3f(max.x, max.y, min.z);
                     glVertex3f(max.x, max.y, max.z);
                     glVertex3f(max.x, min.y, max.z);
```

```
                    glVertex3f(min.x, min.y, min.z);
                    glVertex3f(min.x, min.y, max.z);
                    glVertex3f(min.x, max.y, max.z);
                    glVertex3f(min.x, max.y, min.z);
            glEnd();
            glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
            glPopAttrib();
}

void Init()                          // Инициализация OpenGL
{
       glEnable(GL_DEPTH_TEST);
       glEnable(GL_LIGHTING);
       glEnable(GL_LIGHT0);

       glClearColor(0.5, 0.5, 0.75, 1);

// Создаем дисплейный список объекта (чайника)
list0.new_list(GL_COMPILE);
       glutSolidTeapot(2);
       list0.end_list();
// Устанавливаем функцию обратной связи дисплейного списка прямоугольной
// оболочки объекта
       bound_box_list0.set_build_func(BuildList);
}

void Display()               // Обновление содержимого экрана
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

//Выводим объект на экран
       list0.call_list();
//Рисуем прямоугольную оболочку
       bound_box_list0.call_list();
}
```

На рис. 7.8 показана прямоугольная оболочка, построенная этой программой для чайника, созданного на экране командой `glutSolidTeapot(2)`.

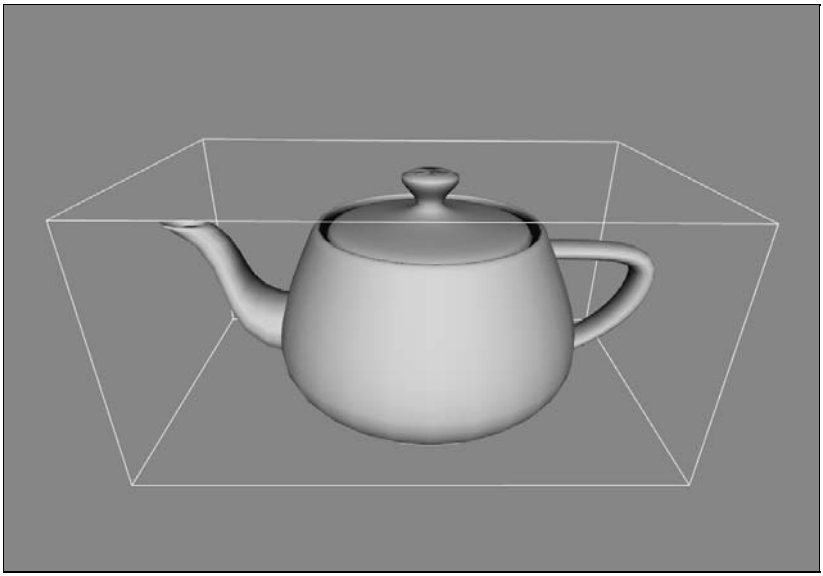


Рис. 7.8. Прямоугольная оболочка для чайника

Если вы запустите программу на выполнение, то в консоли появится информация о том, что прямоугольная оболочка чайника представляет собой прямоугольный параллелепипед, все ребра которого параллельны осям координат (системы координат чайника), а координаты двух крайних вершин этого прямоугольного параллелепипеда равны (−3.000002, −1.500003, −1.999998) и (3.433819, 1.649996, 1.999998). Таким образом, чайник размером в 2 единицы из библиотеки GLUT имеет размеры (6.433821, 3.149999, 3.999996) вдоль осей координат X, Y и Z. Кроме того, нас информируют о том, что чайник размером в 2 единицы состоит из 3136 (!) полигонов. Поэтому нет ничего удивительного в том, что вывод на экран чайника из библиотеки GLUT — очень ресурсоемкая операция.

7.2. Использование расширения HP_occlusion_test для проверки видимости прямоугольной оболочки объекта на экране

Расширение HP_occlusion_test предоставляет приложению механизм определения видимости изображения на экране. Для этого используется тест перекрытия (occlusion test). Тест перекрытия считается пройденным в том случае, если на экран был выведен хотя бы один пиксел изображения, нарисованного после активации этого теста. При этом на тест перекрытия не оказывают влияния маски записи в буферы цвета, глубины и шаблона.

Это дает возможность приложению проверять видимость объекта на экране, не изменяя его содержимого. В результате расширение HP_occlusion_test может быть использовано для проверки видимости прямоугольной оболочки на экране. Для этого приложение должно выполнить следующие действия:

1. Запретить модификацию буферов цвета, глубины и шаблона при помощи команд `glDepthMask(GL_FALSE)`, `glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE)` и `glStencilMask(GL_FALSE)`.
2. Включить тест перекрытия командой `glEnable(GL_OCCLUSION_TEST_HP)`.
3. Нарисовать прямоугольную оболочку объекта.
4. Запретить текст перекрытия командой `glDisable(GL_ OCCLUSION_TEST_HP)`.
5. Разрешить (если необходимо) запись в буферы цвета, глубины и шаблона.
6. Получить результат теста перекрытия при помощи команды `glGetBooleanv(GL_OCCLUSION_TEST_RESULT_HP,&result)`.

Если параметр `result = GL_TRUE`, то это означает, что тест перекрытия прошел успешно, т. е. прямоугольная оболочка (или хотя бы ее часть) была нарисована на экране. Следовательно, объект необходимо вывести она экран.

Для того чтобы продемонстрировать использование расширения HP_occlusion_test, я переписал пример Ex02 с использованием этого расширения (Ex03). В листинге 7.3 приведен исходный текст функции `DrawScene` из примера Ex03.

Листинг 7.3

```
...
// Дисплейный список плоскости зеркала
display_list plane;
// Дисплейный список чайника
display_list teapot;
// Дисплейный список прямоугольной оболочки чайника
lazy_build_display_list teapot_bound;
...
// Рисуем сцену с чайниками на экране
void DrawScene(GLenum FaceMode)
{
       glPushAttrib(GL_ALL_ATTRIB_BITS);
       glDisable(GL_CULL_FACE);
// Перебираем все чайники
       for (unsigned int i=0; i<TeapotCount; i++)
```

```
        {
// Устанавливаем режим лицевых граней
                glFrontFace(FaceMode);
                glPushMatrix();
// Позиционируем чайник и устанавливаем его цвет
                glTranslatef(Teapots[i].pos.x, Teapots[i].pos.y, Tea-
pots[i].pos.z);
                glColor3fv(&Teapots[i].color.x);
                glRotatef(Teapots[i].rotate, 0, 1.0, 0);
// Включаем тест перекрытия
                glEnable(GL_OCCLUSION_TEST_HP);
// Запрещаем модификацию буферов цвета, глубины и шаблона
                glDepthMask(GL_FALSE);
                glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
                glStencilMask(GL_FALSE);
// Рисуем прямоугольную оболочку. Прямоугольная оболочка чайника
// автоматически рассчитывается при первом вызове метода call_list.
                teapot_bound.call_list();
// Разрешаем модификацию буферов цвета, глубины и шаблона
                glDepthMask(GL_TRUE);
                glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
                glStencilMask(GL_TRUE);
// Запрещаем тест перекрытия
                glDisable(GL_OCCLUSION_TEST_HP);
// Получаем отчет о результате теста перекрытия
                GLboolean result;
                glGetBooleanv(GL_OCCLUSION_TEST_RESULT_HP, &result);
// Если тест перекрытия прошел успешно, то рисуем чайник
                if (result==GL_TRUE)
                        teapot.call_list();
                glPopMatrix();
        }
        glPopAttrib();
}
```

Остальные изменения в программе довольно тривиальны, поэтому мы не будем их рассматривать.

Для оценки прироста производительности, который дает использование расширения HP_occlusion_test, я сравнил производительность примеров Ex02 и

Ex03 на видеокартах NVIDIA GeForce FX 5800 Ultra и ATI Radeon 9700 Pro. Результаты измерений приведены в табл. 7.1 (см. стр. 417).

Как видно из табл. 7.1, использование расширения HP_occlusion_test обеспечивает довольно существенный прирост производительности. Если в кадр попадает около 50—60% процентов всех объектов сцены (режим `FRONT`), то расширение HP_occlusion_test обеспечивает прирост производительности в районе 25—50%. Если же тест перекрытия не проходит ни один объект (режим BACK), то использование расширения HP_occlusion_test увеличивает производительность приложения почти на порядок. При этом на ускорителе NVIDIA GeForce FX 5800 Ultra производительность примера Ex08 в режиме `BACK` составляет примерно 84% от предельно возможной, когда проверка видимости одного чайника занимает бесконечно мало времени.

Тем не менее расширение HP_occlusion_test имеет некоторые существенные недостатки.

- Расширение HP_occlusion_test не поддерживает параллельное выполнение нескольких тестов перекрытия. Перед тем как провести новый тест перекрытия, приложение должно получить результаты предыдущего теста. В то же время запрос результата теста перекрытия — довольно трудоемкая операция, приводящая к остановке графического конвейера GPU. Поэтому операции выполнения теста перекрытия и получения результатов теста перекрытия должны быть разнесены во времени. Причем в идеале результаты теста перекрытия должны запрашиваться только в следующем кадре (или даже через кадр).
- Расширение HP_occlusion_test не содержит средств для проверки готовности отчета о результатах теста перекрытия. В результате программист должен действовать "наугад". Если он запросит данные теста перекрытия слишком рано, то вызовет остановку графического конвейера. Если слишком поздно — то GPU не будет загружен на 100%.
- Тест перекрытия возвращает результат булевского типа (`GL_FALSE` / `GL_TRUE`), в то же время приложению часто необходимо знать, какая часть изображения прошла тест перекрытия.

Эти недостатки затрудняют использование расширения HP_occlusion_test в реальных приложениях. Поэтому корпорация NVIDIA разработала улучшенную версию расширения HP_occlusion_test — расширение NV_occlusion_query, которое будет рассмотрено в следующем разделе.

7.3. Расширения NV_occlusion_query

Расширение NV_occlusion_query лишено недостатков, перечисленных в конце предыдущего раздела. Хотя расширения NV_occlusion_query и HP_occlusion_test похожи друг на друга, они имеют ряд существенных различий.

Главное различие заключается в том, что расширение NV_occlusion_query поддерживает параллельное выполнение нескольких тестов перекрытия.

Примечание

Хотя в названии расширения NV_occlusion_query присутствует префикс NV, оно поддерживается большинством поставщиков OpenGL, в частности, и корпорацией ATI.

Перед началом использования тестов перекрытия расширения NV_occlusion_query, приложение должно получить дескрипторы объектов запроса результатов теста перекрытия. Далее мы будем называть их просто дескрипторы теста перекрытия. Эта операция выполняется при помощи новой команды `glGenOcclusionQueriesNV`:

```
void glGenOcclusionQueriesNV(GLsizei n, GLuint *ids);
```

Как видно, команда принимает в качестве параметров количество запрашиваемых дескрипторов и указатель на массив дескрипторов теста перекрытия.

Запуск теста перекрытия осуществляется командой `glBeginOcclusionQueryNV`, принимающей в качестве параметра дескриптор теста перекрытия:

```
void glBeginOcclusionQueryNV(GLuint id);
```

Окончание теста перекрытия осуществляется командой `glEndOcclusionQueryNV`:

```
void glEndOcclusionQueryNV(void);
```

Результаты теста перекрытия запрашиваются командой `glGetOcclusionQueryuivNV`:

```
void glGetOcclusionQueryuivNV(GLuint id, GLenum pname, GLuint *params)
```

где:

- `id` — дескриптор теста перекрытия;
- `pname` — название запрашиваемого параметра;
- `params` — указатель на область памяти, в которую команда занесет результат.

Команда `glGetOcclusionQueryuivNV` может возвращать два параметра теста перекрытия (`pname`):

- Доступность результатов теста перекрытия (`GL_PIXEL_COUNT_AVAILABLE_NV`). Если результаты теста перекрытия уже готовы, то команда `glGetOcclusionQueryuivNV` возвратит `GL_TRUE`, в противном случае возвратит `GL_FALSE`.

- Результат теста перекрытия (`GL_PIXEL_COUNT_NV`). Возвращает количество пикселов изображения, прошедших тест перекрытия. Если результаты теста перекрытия еще не готовы, то приостанавливает работу приложения и ждет, пока не будут получены результаты теста перекрытия.

После того как объекты запросов теста перекрытия станут бесполезными, приложение должно удалить их командой `glDeleteOcclusionQueriesNV`:

```
void glDeleteOcclusionQueriesNV(GLsizei n, const GLuint *ids);
```

Параметры этой команды аналогичны параметрам команды `glGenOcclusionQueriesNV`.

Для наглядности ниже приведена последовательность действий, которую необходимо выполнить для проверки видимости прямоугольных оболочек объектов на экране.

1. Запретить модификацию буферов цвета, глубины и шаблона при помощи команд `glDepthMask(GL_FALSE)`, `glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE)` и `glStencilMask(GL_FALSE)`.
2. Получить дескрипторы объектов запроса результатов теста перекрытия.
3. Включить *i*-й тест перекрытия.
4. Нарисовать прямоугольную оболочку *i*-го объекта.
5. Выключить тест перекрытия.
6. Повторить шаги 3—5 для всех объектов.
7. Разрешить (если необходимо) запись в буферы цвета, глубины и шаблона.
8. Проверить доступность результата *i*-теста перекрытия.
9. Если результат теста доступен, то получить результат теста перекрытия. В противном случае желательно подождать с получением результата теста.
10. Если количество пикселов больше нуля, то это означает, что тест перекрытия прошел успешно, т. е. прямоугольная оболочка (или хотя бы ее часть) была нарисована на экране. Следовательно, объект необходимо вывести она экран.
11. Если объекты запроса результатов теста перекрытия больше не нужны, то уничтожить их.

В описании шага 9 говорится, что приложению, скорее всего, придется подождать с получением результата теста перекрытия. Но сколько ему придется ждать? Оказывается, бывают ситуации, когда результаты теста перекрытия могут быть готовы только во время подготовки следующего кадра (или даже через кадр). В этом случае у программиста есть два пути:

1. Получить результаты теста перекрытия в текущем кадре, пожертвовав скоростью работы приложения.

2. Не получать результаты теста перекрытия до тех пор, пока они не будут готовы. А в расчетах использовать результаты прошлого теста перекрытия. В этом случае придется пожертвовать точностью расчета видимости прямоугольной оболочки объекта.

Как видно, каждый вариант имеет свои достоинства и недостатки. Но в программах, критичных к быстродействию компьютера (например, в играх), обычно используется второй вариант.

Для демонстрации использования расширения NV_occlusion_query я переписал пример Ex08 с использованием этого расширения. В листинге 7.4 приведены наиболее важные фрагменты кода полученной программы.

Листинг 7.4

```
...
// Дисплейный список плоскости зеркала
display_list plane;
// Дисплейный список чайника
display_list teapot;
// Дисплейный список прямоугольной оболочки чайника
lazy_build_display_list teapot_bound;

// Структура с информацией о чайнике
struct Teapot
{
// Цвет чайника
       vec3 color;
// Положение чайника
       vec3 pos;
// Угол поворота чайника
       float rotate;
};

// Количество чайников
const TeapotCount=6;
// Массив с информацией о чайниках
Teapot Teapots[TeapotCount];

// Массив с дескрипторами объектов запроса о результатах тестов
// перекрытия
GLuint occlusionQueries[TeapotCount*2];
```

```
// Результаты тестов перекрытия
GLuint occlusionResults[TeapotCount*2];
// Текущий тест перекрытия
int currentOcclusion;
// Выполнены ли тесты перекрытия
int IsOcclusionReady;

// Счетчик кадров, к началу рисования которых результаты теста перекрытия
// были готовы
int ready_count=0;
// Счетчик кадров, к началу рисования которых результаты теста перекрытия
// не были готовы
int not_ready_count=0;
...
// Инициализируем программу и OpenGL
bool Init()
{
...
// Получаем дескрипторы объектов запроса результатов теста перекрытия.
// Обратите внимание, что число дескрипторов в 2 раза больше числа
// объектов. Это связано с тем, что программа проверяет как видимость
// прямоугольных оболочек самих чайников, так и их отражений
       glGenOcclusionQueriesNV(TeapotCount*2, occlusionQueries);
// Инициализируем начальные значения результатов теста перекрытия
// (считаем, что все объекты видны на экране)
       for (int i=0; i<TeapotCount*2; i++)
              occlusionResults[i]=GL_TRUE;
// Тесты перекрытия еще ни разу не выполнялись
       IsOcclusionReady=2;
...
       return true;
}
// Рисуем сцену с чайниками
// Функция использует глобальную переменную currentOcclusion, которая
// равна 0, если рисуется сцена, или равна TeapotCount, если рисуется
// отражение
void DrawScene(GLenum FaceMode)
{
// Настраиваем параметры OpenGL
```

```
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        glDisable(GL_CULL_FACE);
        glFrontFace(FaceMode);
// Если готовы результаты тестов перекрытия, то запускаем на выполнение
// следующую "партию" тестов перекрытия
        if (IsOcclusionReady==1)
        {
// Запрещаем модификацию буферов глубины, цвета и шаблона
                glDepthMask(GL_FALSE);
                glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
                glStencilMask(GL_FALSE);
// Перебираем все прямоугольные оболочки чайников
                for (unsigned int i=0; i<TeapotCount; i++)
                {
                        glPushMatrix();
// Позиционируем прямоугольную оболочку
                        glTranslatef(Teapots[i].pos.x, Teapots[i].pos.y,
Teapots[i].pos.z);
                        glRotatef(Teapots[i].rotate, 0, 1.0, 0);
// Запускаем тест перекрытия
        glBeginOcclusionQueryNV(occlusionQueries[currentOcclusion]);
// Рисуем прямоугольную оболочку
                        teapot_bound.call_list();
// Заканчиваем тест перекрытия
                        glEndOcclusionQueryNV();
                        glPopMatrix();
// Увеличиваем индекс текущего объекта запроса...
                        currentOcclusion++;
                }
// Восстанавливаем все как было
                glDepthMask(GL_TRUE);
                glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
                glStencilMask(GL_TRUE);
// Устанавливаем индекс текущего объекта запроса результатов теста
// перекрытия (нам ведь еще и саму сцену надо рисовать...)
                currentOcclusion-=TeapotCount;
        }
// Рисуем сцену (перебираем все объекты)
        for (unsigned int i=0; i<TeapotCount; i++)
```

```
        {
// Если прямоугольная оболочка объекта видна на экране, то рисуем объект
                if (occlusionResults[currentOcclusion]>0)
                {
                        glPushMatrix();
                        glTranslatef(Teapots[i].pos.x, Teapots[i].pos.y,
                        Teapots[i].pos.z);
                        glRotatef(Teapots[i].rotate, 0, 1.0, 0);
                        glColor3fv(&Teapots[i].color.x);
                        teapot.call_list();
                        glPopMatrix();
                }
// Увеличиваем индекс текущего дескриптора теста отсечения
                currentOcclusion++;
        }

        glPopAttrib();
}
...
// Рисуем изображение на экране
void Display()
{
// Очищаем экран
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
GL_STENCIL_BUFFER_BIT);
// Если тест перекрытия был хотя бы один раз выполнен
        if (IsOcclusionReady!=2)
        {
// Получаем состояние последнего объекта запроса результатов теста
// перекрытия
                GLuint result;
                glGetOcclusionQueryuivNV(occlusionQueries[TeapotCount*2-
1], GL_PIXEL_COUNT_AVAILABLE_NV, &result);
// Если тест перекрытия выполнен
                if (result==GL_TRUE)
                {
// Получаем информацию обо всех тестах перекрытия
                        for (int i=0; i<TeapotCount*2; i++)
```

```
        glGetOcclusionQueryuivNV(occlusionQueries[i], GL_PIXEL_COUNT_NV,
&occlusionResults[i]);
// Тест перекрытия выполнен
                        IsOcclusionReady=1;
// Увеличиваем счетчик кадров
                        ready_count++;
                }
                else
                {
// Увеличиваем счетчик кадров
                        not_ready_count++;
// Тест перекрытия не выполнен
                        IsOcclusionReady=0;
                }
        }
        else
// Тест перекрытия выполнен нормально (необходимо для того, чтобы
// был запущен первый тест перекрытия)
                IsOcclusionReady=1;
// Текущий дескриптор теста перекрытия
        currentOcclusion=0;
// Далее рисуется сцена (код пропущен для экономии места)
...
}
```

В этом примере используется довольно интересный прием: когда программе необходимо узнать готовность результатов всех тестов отсечения, то она запрашивает состояние только того теста перекрытия, который был выполнен последним. Этот прием основан на том, что тесты перекрытия всегда выполняются в порядке их вызова программой. Поэтому, если готовы результаты *i*-го теста, то результаты тестов [0..i—1] готовы наверняка.

Анализируя колонку `Ready` примера Ex04, можно заметить, что в режиме `FRONT` результаты теста перекрытия всегда оказываются доступными уже в следующем кадре, а в режиме `BACK` — только через кадр. Из этого можно сделать следующий вывод: если в программе бóльшая часть объектов видна на экране, то результаты теста перекрытия будут доступны в следующем кадре, в противном случае — только через кадр.

Таблица 7.1. *Сравнение производительности примеров Ex01, Ex03и Ex04 в различных режимах*

Видеокарта	Ex01			Ex03		Ex04			
	FRONT FPS	BACK FPS	BACK без чайников* FPS	FRONT FPS	BACK FPS	FRONT		BACK	
						FPS	Ready** (%)	FPS	Ready (%)
ATI Radeon 9700 Pro	27	27	468	44	216	50	100	443	50
NVIDIA GeForce FX 5800 Ultra	35	35	188	44	145	59	100	182	50

* В этом режиме дисплейный список чайника заменен пустым списком. Таким образом, режим `BACK` "без чайников" показывает предельную производительность примера Ex01 в режиме `BACK`, которой он сможет достичь только в том случае, если рисование невидимого чайника займет бесконечно мало времени.

** Показывает отношение количества кадров, к началу вывода которых результат предыдущего теста отсечения оказался готовым, к количеству кадров, к началу вывода которых результат предыдущего теста отсечения не был готов.

Сравнивая производительность примера Ex01 в режиме BACK "без чайников" и примера Ex03 в режиме BACK, можно заметить, что применение расширения NV_occlusion_query позволило нам вплотную приблизиться к теоретическому пределу, который можно достичь только в случае, когда проверка видимости прямоугольной оболочки объекта занимает бесконечно мало времени. Поэтому дальнейшую оптимизацию программы с целью уменьшить время проверки видимости прямоугольной оболочки на экране вряд ли можно назвать оправданной.

7.4. Пример программной проверки попадания прямоугольной оболочки в пирамиду видимости

Применение расширения NV_occlusion_query помогло нам значительно увеличить производительность программы. Однако следует помнить, что расширение NV_occlusion_query не следует использовать для проверки видимости объектов, состоящих из небольшого количества полигонов, например, высотных домов, представляющих собой "коробку" с "натянутой" на нее текстурой. Дело в том, что при рисовании таких объектов приложение упирается не в производительность блока T&L, а в пропускную способность памяти видеокарты. В результате рисование прямоугольной оболочки, содержащей обычно большее количество пикселов, чем оригинальный объект, может оказаться более трудоемким процессом, чем рисование самого объекта.

Но что делать, если нам все-таки необходимо организовать проверку видимости малополигональных объектов. Выходом из сложившейся ситуации может служить программная проверка попадания прямоугольной оболочки объекта в пирамиду видимости камеры. Эту проверку можно реализовать множеством алгоритмов, каждый из которых обладает своими достоинствами и недостатками. Один из таких алгоритмов сводится к проверке положения прямоугольной оболочки объекта относительно шести отсекающих плоскостей, образующих пирамиду видимости куба. Если прямоугольная оболочка целиком отсекается хотя бы одной из плоскостей пирамиды видимости, то объект не попадает в эту пирамиду. Если же прямоугольная оболочка не отсекается целиком ни одной из плоскостей пирамиды видимости, то делается вывод, что прямоугольная оболочка видна на экране. Главный плюс этого алгоритма — очень высокая производительность. Тем не менее и у него имеются существенные недостатки.

- В некоторых редких случаях он может давать "ложные срабатывания", сообщая о "видимости" невидимого объекта.

- Этот алгоритм корректно работает только для тех прямоугольных оболочек объекта, геометрические размеры которых не превышают размер пирамиды видимости.

Вместо того чтобы "изобретать велосипед", мы рассмотрим готовую реализацию этого алгоритма, созданную Петром Поповым. В этой реализации понятие прямоугольной оболочки объекта инкапсулируется классом `psAABB`:

```
class psAABB
{
public:
// Центр прямоугольной оболочки
       float cen[3];
// Половина размера прямоугольной оболочки вдоль осей x, y и z
       float dx,dy,dz;

// Конструктор и деструктор (не выполняют никаких операций)
       psAABB();
       virtual ~psAABB();

};
```

Для инкапсуляции понятия "камера" используется класс `psCamera`:

```
class psCamera
{
public:
// Проверяем видимость камерой данной прямоугольной
// оболочки. Если прямоугольная оболочка видна в камере, то метод
// возвращает false, в противном случае — true
       bool psTest(psAABB &bb);
// Настраиваем параметры камеры исходя из текущих значений матриц
// проекции (GL_PROJECTION_MATRIX) и модели (GL_MODELVIEW)
       void psGetPositionGL();
// Конструктор и деструктор (не выполняют никаких действий)
       psCamera();
       virtual ~psCamera();
// Другие поля и методы, не представляющие для нас никакого интереса
...
};
```

Ниже приведен небольшой пример проверки видимости на экране куба, выводимого при помощи команды `glutSolidCube(1.0)`:

```
// Создаем объект прямоугольной оболочки
psAABB box;
// Задаем центр прямоугольной оболочки
box.cen[0]=0;
box.cen[1]=0;
box.cen[2]=0;
// Задаем размеры прямоугольной оболочки
box.dx=0.5;
box.dy=0.5;
box.dz=0.5;
// Создаем камеру
psCamera cam;
// Инициализируем объект камеры на основе текущих значений матриц модели
// и проекции
cam.psGetPositionGL();
// Если в камере видна прямоугольная оболочка, рисуем объект
if (!cam.psTest(box))
       glutSolidCube(1);
```

А теперь попробуйте ответить на вопрос: в чем заключается главная разница между проверкой видимости объекта с использованием расширения NV_occlusion_query и классом `psCamera`?

Класс `psCamera` проверяет только попадание прямоугольной оболочки в пирамиду видимости камеры, в то время как расширение NV_occlusion_query по-настоящему проверяет видимость объекта на экране с учетом значений *z*-буфера. Иными словами, класс `psCamera` используется для быстрой грубой проверки видимости прямоугольной оболочки, а расширение NV_occlusion_query — для медленной, но точной. Поэтому приложение должно использовать двухуровневую проверку видимости: на первом этапе с использованием класса `psCamera` приложение быстро отсекает все заведомо невидимые объекты (не попадающие в пирамиду видимости), а на втором этапе оставшиеся высокополигональные объекты проверяются на видимость с использованием расширения NV_occlusion_query.

Для демонстрации практического использования классов `psAABB` и `psCamera` я переписал пример Ex04, заменив проверку видимости чайников с использованием расширения NV_occlusion_query на проверку видимости при помощи классов `psAABB` и `psCamera` (Ex05).

Для начала необходимо скопировать в каталог с проектом все файлы из каталога psCamera, после чего включить в состав проекта файлы psAABB.cpp и psCamera.cpp, а в начало файла main.cpp добавить директиву `#include "psCamera.h"`. Далее необходимо внести небольшие изменения в исходный текст программы, которые фактически сводятся к следующим двум пунктам:

1. Функция `BuildBoundBox` вместо формирования дисплейного списка вывода теперь должна создавать объект класса `psAABB`, содержащий информацию о параметрах прямоугольной оболочки.
2. Функция `DrawScene` должна использовать для определения видимости прямоугольной оболочки класс `psCamera` вместо средств расширения NV_occlusion_query.

Как видно, изменения в программе более чем тривиальны, поэтому я сразу приведу модифицированные фрагменты исходного кода программы (листинг 7.5).

Листинг 7.5

```
// Прямоугольная оболочка объекта
psAABB bound;
// Рассчитана ли прямоугольная оболочка объекта
bool bBuildBound=false;

// Рассчитываем и возвращаем прямоугольную оболочку чайника
psAABB BuildBoundBox()
{
// Создаем прямоугольную оболочку нулевого размера
       psAABB box;
       ZeroMemory(&box, sizeof(box));
// Рисуем чайник в режиме GL_FEEDBACK и определяем два угла (min и max)
// его прямоугольной оболочки
...

// Определяем центр прямоугольной оболочки чайника
       box.cen[0]=(max.x+min.x)/2.0;
       box.cen[1]=(max.y+min.y)/2.0;
       box.cen[2]=(max.z+min.z)/2.0;
// Определяем половину размера прямоугольной оболочки вдоль
// осей x, y и z
       box.dx=fabs(max.x-box.cen[0]);
```

```
        box.dy=fabs(max.y-box.cen[1]);
        box.dz=fabs(max.z-box.cen[2]);

        glPopAttrib();

// Возвращаем полученную прямоугольную оболочку
        return box;
}

// Обновляем содержимое экрана
void Display()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
GL_STENCIL_BUFFER_BIT);

// Если прямоугольная оболочка не рассчитана
        if (!bBuildBound)
        {
// Рассчитываем ее
                bound=BuildBoundBox();
                bBuildBound=true;
        }
...
}

// Рисуем набор чайников
void DrawScene(GLenum FaceMode)
{
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        glDisable(GL_CULL_FACE);
        glFrontFace(FaceMode);

// Объект камеры
        psCamera camera;

// Рисуем чайники
        for (unsigned int i=0; i<TeapotCount; i++)
        {
                glPushMatrix();
```

```
// Устанавливаем позицию и поворот чайника
              glTranslatef(Teapots[i].pos.x, Teapots[i].pos.y, Tea-
pots[i].pos.z);
              glRotatef(Teapots[i].rotate, 0, 1.0, 0);

// Инициализируем камеру на основе матриц OpenGL
              camera.psGetPositionGL();
// Проверяем видимость чайника
              if (!camera.psTest(bound))
              {
                     glColor3fv(&Teapots[i].color.x);
                     teapot.call_list();
              }

              glPopMatrix();
       }

       glPopAttrib();
}
```

В качестве самостоятельного упражнения я бы посоветовал вам попробовать объединить примеры Ex04 и Ex05, реализовав двухуровневую проверку видимости объектов на экране с использованием класса `psCamera` и расширения NV_occlusion. Правда, прирост производительности такой программы по сравнению с примером Ex04 будет минимальным — из табл. 7.1 следует, что пример Ex04 и так достиг практически предельной производительности.

Заключение

В этой главе мы рассмотрели два расширения HP_occlusion_test и NV_occlusion_query. Расширение HP_occlusion_test позволяет программисту определять видимость объектов на экране. Его, как правило, используют для проверки видимости на экране прямоугольных оболочек. Расширение NV_occlusion_query является улучшенной версией расширения HP_occlusion_test и отличается от него расширенными возможностями и быстродействием. Кроме того, в конце главы был рассмотрен пример программной проверки попадания прямоугольной оболочки в пирамиду видимости.

Глава 8

Использование внеэкранных буферов

При написании программ, использующих многопроходные алгоритмы, часто возникает потребность осуществить промежуточный вывод изображения на виртуальный экран, невидимый пользователю. Такой виртуальный экран может использоваться для создания динамических текстур, зеркальных поверхностей, кубических текстур и т. д.

К сожалению, базовая версия OpenGL не содержит стандартных средств для работы с внеэкранными поверхностями, не считая второго кадрового буфера в режиме двойной буферизации. Но и при использовании кадрового буфера мы сталкиваемся с существенным ограничением — размер образа, создаваемого во втором кадровом буфере, не может превышать разрешение экрана. Например, если пользователь работает в режиме 800×600, не получится использовать динамические текстуры с разрешением 1024×1024.

В начале 1999 года появилось WGL-расширение WGL_ARB_pbuffer, предназначенное для работы с буфером пикселов (pixel buffer или сокращенно `pbuffer`). Буфер пикселов — это область видеопамяти, в которой размещается виртуальный экран. С точки зрения программы, виртуальный экран не отличается от настоящего — он имеет свой контекст OpenGL, поддерживает практически все команды OpenGL и т. д.

Поэтому буфер пикселов должен создаваться аналогично любому другому окну путем выбора нужного формата пикселов командой `ChoosePixelFormat` с последующим созданием внеэкранного буфера с этим же форматом пикселов.

Но в действительности все не так хорошо. Дело в том, что команда `ChoosePixelFormat` использует структуру `PIXELFORMATDESCRIPTOR` с фиксированным набором полей, среди которых нет ни одного поля для указания того, что этот формат пикселов будет использоваться буфером пикселов. В результате функция `ChoosePixelFormat` оказывается непригодной для задания формата пикселов для буфера пикселов.

Поэтому для задания формата пикселов буфера pbuffer используется расширение WGL_ARB_pixel_format, которое мы сейчас и рассмотрим.

8.1. Расширение WGL_ARB_pixel_format

Расширение WGL_ARB_pixel_format предоставляет в распоряжение программиста набор из трех команд:

- wglGetPixelFormatAttribivARB
- wglGetPixelFormatAttribfvARB
- wglChoosePixelFormatARB

Эти команды, являющиеся аналогами функций Win32 API — DescribePixelFormat, DescribeLayerPlane и ChoosePixelFormat, — позволяют приложению запрашивать у операционной системы список поддерживаемых форматов пикселов с последующей установкой требуемого формата пикселов. Главное преимущество команд расширения WGL_ARB_pixel_format перед аналогичными функциями Win32 заключается в том, что они работают с форматом пикселов как с черным ящиком: значение атрибута формата пиксела задается двумя параметрами — названием атрибута и значением атрибута. В результате любая реализация OpenGL может добавлять новые атрибуты формата пикселов, не внося изменения в аргументы команд wglGetPixelFormatAttribivARB, wglGetPixelFormatAttribfvARB и wglChoosePixelFormatARB. Рассмотрим эти команды подробнее.

Команда wglGetPixelFormatAttribivARB используется для запроса атрибутов определенной плоскости с заданным форматом пикселов:

```
BOOL wglGetPixelFormatAttribivARB(HDC hdc, int iPixelFormat, int
iLayerPlane, UINT nAttributes, const int *piAttributes, int *piValues);
```

где:

- hdc — дескриптор контекста OpenGL;
- iPixelFormat — индекс запрашиваемого формата пикселов;
- iLayerPlane — номер запрашиваемой плоскости. Главная плоскость имеет номер 0. Плоскости, находящиеся под главной плоскостью, имеют номера меньше 0 (−1, −2 и т. д.). Плоскости, находящиеся над главной плоскостью, имеют номера больше 0 (1, 2 и т. д.);
- nAttributes — количество запрашиваемых атрибутов;
- piAttributes — указатель на массив с идентификаторами запрашиваемых атрибутов. Идентификаторы атрибутов можно найти в спецификации расширения WGL_ARB_pixel_format;
- piValues — указатель на буфер, в который будут занесены значения запрашиваемых атрибутов.

Для демонстрации использования команды `wglGetPixelFormatAttribivARB` я модифицировал библиотеку GLHE, добавив в нее консольную команду `pixel_format_info`, выводящую на экран информацию о текущем формате пикселов (Ex01) (листинг 8.1).

Листинг 8.1

```
// Обработчик консольных команд интерактора glut_console
bool glut_console::stdCmdProc(void* Object, string Cmd, vector<string>
Params, glut_console* console)
{
...
// Если текущая команда — pixel_format_info
       if (Cmd=="pixel_format_info")
       {
// Инициализируем расширение WGL_ARB_pixel_format
              if (glh_init_extensions("WGL_ARB_pixel_format"))
              {
// Размер массивов атрибутов
                     const MAX_ATTRIBS=32;
// Массив идентификаторов атрибутов
                     int iAttributes[MAX_ATTRIBS];
// Массив значений атрибутов
                     int iValues[MAX_ATTRIBS];

// Получаем контекст устройства, связанного с текущим контекстом
// OpenGL
                     HDC dc=wglGetCurrentDC();
// Получаем индекс текущего формата пикселов
                     int iPixelFormat=GetPixelFormat(dc);
// Индекс текущего атрибута и, одновременно, суммарное количество
// атрибутов
                     unsigned int nAttributes=0;
// Составляем список запрашиваемых атрибутов
// Поддержка GDI
                     iAttributes[nAttributes++]=WGL_SUPPORT_GDI_ARB;
// Количество бит, отведенных под красный цветовой канал
                     iAttributes[nAttributes++]=WGL_RED_BITS_ARB;
// Количество бит, отведенных под зеленый цветовой канал
                     iAttributes[nAttributes++]=WGL_GREEN_BITS_ARB;
```

```
// Количество бит, отведенных под синий цветовой канал
                    iAttributes[nAttributes++]=WGL_BLUE_BITS_ARB;
// Количество бит, отведенных под альфа-канал
                    iAttributes[nAttributes++]=WGL_ALPHA_BITS_ARB;
// Глубина буфера глубины
                    iAttributes[nAttributes++]=WGL_DEPTH_BITS_ARB;
// Глубина буфера шаблона
                    iAttributes[nAttributes++]=WGL_STENCIL_BITS_ARB;
// Запрашиваем информацию о данном формате пикселов
                    wglGetPixelFormatAttribivARB(dc, iPixelFormat, 0,
nAttributes, &iAttributes[0], &iValues[0]);
// Расшифровываем информацию из массива iValues
                    console->add("Pixel format information:");
                    if (iValues[0])
                           console->add("GDI rendering is
unsupported");
                    else
                           console->add("GDI rendering is supported");
                    char buf[1024];
                    sprintf(&buf[0], "The number of red bitplanes: %d",
iValues[1]);
                    console->add(buf);
                    sprintf(&buf[0], "The number of green bitplanes:
%d", iValues[2]);
                    console->add(buf);
                    sprintf(&buf[0], "The number of blue bitplanes:
%d", iValues[3]);
                    console->add(buf);
                    sprintf(&buf[0], "The number of alpha bitplanes:
%d", iValues[4]);
                    console->add(buf);
                    sprintf(&buf[0], "Depth of depth buffer: %d",
iValues[5]);
                    console->add(buf);
                    sprintf(&buf[0], "Depth of stencil buffer: %d",
iValues[6]);
                    console->add(buf);
                    return true;
             }
             else
             {
```

```
                    console->add("WGL_ARB_pixel_format extension does
not supported");
                    return false;
                }
        }

        return false;
}
```

Команда `wglGetPixelFormatAttribfvARB` является аналогом команды `wglGetPixelFormatAttribivARB`, предназначенной для получения информации об атрибутах вещественного типа:

```
BOOL wglGetPixelFormatAttribfvARB(HDC hdc, int iPixelFormat, int
iLayerPlane, UINT nAttributes, const int *piAttributes, FLOAT *pfValues);
```

где:

- `hdc` — дескриптор контекста OpenGL;
- `iPixelFormat` — индекс запрашиваемого формата пикселов;
- `iLayerPlane` — номер запрашиваемой плоскости. Главная плоскость имеет номер 0. Плоскости, находящиеся под главной плоскостью, имеют номера меньше 0 (−1, −2 и т. д.). Плоскости, находящиеся над главной плоскостью, имеют номера больше 0 (1, 2 и т. д.);
- `nAttributes` — количество запрашиваемых атрибутов;
- `piAttributes` — указатель на массив с идентификаторами запрашиваемых атрибутов. Идентификаторы атрибутов можно найти в спецификации расширения WGL_ARB_pixel_format;
- `pfValues` — указатель на буфер, в который будут занесены значения запрашиваемых атрибутов.

Как мы можем видеть, единственное отличие команды `wglGetPixelFormatAttribfvARB` от команды `wglGetPixelFormatAttribivARB` сводится к замене последнего параметра с целочисленного указателя `piValues` на вещественный `pfValues`.

Команда `wglChoosePixelFormatARB`, являющаяся расширенным аналогом функции Win32 API `ChoosePixelFormat`, предназначена для поиска формата пикселов, удовлетворяющих заданному критерию:

```
BOOL wglChoosePixelFormatARB(HDC hdc, const int *piAttribIList, const
FLOAT *pfAttribFList, UINT nMaxFormats, int *piFormats, UINT
*nNumFormats);
```

где:

- `hdc` — дескриптор контекста OpenGL;

- piAttribIList — указатель на массив со списком требуемых целочисленных атрибутов. Атрибуты задаются парами "идентификатор атрибута" и "значение атрибута". Список атрибутов оканчивается нулевым значением;
- nMaxFormat — максимальное количество форматов, которое может вернуть команда wglChoosePixelFormatARB;
- piFormats — указатель на массив, в который будут занесены индексы форматов пикселов, удовлетворяющих заданному критерию;
- nNumFormats — суммарное количество найденных форматов, удовлетворяющих заданному критерию. Это количество может быть больше, чем nMaxFormat.

Если в ходе работы функции не произошло никаких ошибок, она возвращает GL_TRUE, в противном случае — GL_FALSE. Если не было найдено ни одного формата, удовлетворяющего заданному критерию, то nNumFormats будет равен 0, а функция возвратит GL_TRUE.

Обратите внимание, что команда wglChoosePixelFormatARB не заменяет функцию WIN32 API ChoosePixelFormat. Дело в том, что функция wglChoosePixelFormatARB является функцией расширения WGL_ARB_pixel_format. Следовательно, приложение перед началом ее использования должно получить точку входа этой функции. А для этого приложение должно иметь доступ к контексту OpenGL. В результате команда wglChoosePixelFormatARB оказывается малопригодной для установки формата пикселов главного окна программы.

Теперь, после того как мы познакомились с расширением WGL_ARB_pixel_format, можно вернуться к рассмотрению буфера пикселов.

8.2. Расширение WGL_ARB_pbuffer

Расширение WGL_ARB_pbuffer предоставляет программисту возможность использовать буферы пикселов (pixels buffers), или сокращенно pbuffer. С точки зрения OpenGL, буферы пикселов — это просто дополнительные окна-невидимки, имеющие соответствующий формат пикселов. Правда, на pbuffer накладываются некоторые ограничения:

- в pbuffer нельзя выводить информацию с использованием GDI;
- Pbuffer может не поддерживать некоторые форматы пикселов, поддерживаемые главным окном;
- Pbuffer не может быть создан, если не хватает ресурсов для его создания (например, видеопамяти);
- Pbuffer может быть потерян при смене видеорежима.

Pbuffer создается командой wglCreatePbufferARB, возвращающей дескриптор созданного буфера pbuffer:

```
HPBUFFERARB wglCreatePbufferARB(HDC hDC, int iPixelFormat, int iWidth,
int iHeight, const int *piAttribList);
```

где:

- hDC — дескриптор контекст устройства, на котором создается буфер pbuffer;
- iPixelFormat — формат пикселов буфера pbuffer. Для получения формата пикселов лучше всего воспользоваться командой wglChoosePixelFormatARB, указав в списке атрибутов параметр WGL_DRAW_TO_PBUFFER_ARB равным TRUE;
- iWidth — ширина буфера pbuffer;
- iHeight — высота буфера pbuffer;
- piAttribList — список атрибутов буфера pbuffer. Список включает в себя пары "идентификатор атрибута" и "значение атрибута". Последний элемент списка — 0.

Если буфер pbuffer не удалось создать (например, из-за недостаточного объема видеопамяти), команда wglCreatePbufferARB возвратит 0.

Следует помнить, что команда wglCreatePbufferARB не гарантирует создание буфера pbuffer заданного размера, т. к. некоторые реализации OpenGL могут накладывать ограничения на размер буфера pbuffer. Например, некая видеокарта может поддерживать только pbuffer шириной, кратной 16, и высотой, кратной 2. Если же программа попытается создать на такой видеокарте буфер pbuffer размером 13×15, то драйвер OpenGL, скорее всего, увеличит его размер до 16×16. Поэтому программа должна после создания буфера pbuffer проверить его размер.

Для получения текущего состояния буфера pbuffer используется команда wglQueryPbufferARB:

```
BOOL wglQueryPbufferARB(HPBUFFERARB hPbuffer,
int iAttribute,
int *piValue);
```

где:

- hPbuffer — дескриптор буфера pbuffer;
- iAttribute — идентификатор запрашиваемого атрибута буфера pbuffer;
- piValue — указатель на область памяти, в которую будет занесено значение запрашиваемого атрибута.

Идентификатор атрибута буфера pbuffer (iAttribute) может принимать одно из следующих значений:

- WGL_PBUFFER_WIDTH_ARB — текущая ширина буфера pbuffer;
- WGL_PBUFFER_HEIGHT_ARB — текущая высота буфера pbuffer;
- WGL_PBUFFER_LOST_ARB — потерян ли pbuffer. Если pbuffer потерян, то параметр будет равен GL_TRUE, в противном случае — GL_FALSE. Этот параметр будет рассмотрен позже.

Часто бывают ситуации, когда приложению необходимо создать pbuffer максимального размера. Но поскольку pbuffer создается в видеопамяти, его максимально допустимый размер зависит от многих факторов, в частности, от размера свободной видеопамяти. Поэтому перед созданием буфера pbuffer программа должна запросить максимальный его размер. Это можно сделать при помощи команды из расширения WGL_ARB_pixel_format — wglGetPixelFormatAttribivARB с одним из следующих атрибутов:

- WGL_MAX_PBUFFER_WIDTH_ARB — максимальная ширина pbuffer;
- WGL_MAX_PBUFFER_HEIGHT_ARB — максимальная высота pbuffer;
- WGL_MAX_PBUFFER_PIXELS_ARB — максимальное количество пикселов в буфере pbuffer.

Помните, что значение параметра WGL_MAX_PBUFFER_PIXELS_ARB меньше, чем произведение параметров WGL_MAX_PBUFFER_WIDTH_ARB и WGL_MAX_PBUFFER_HEIGHT_ARB. В результате программа сама должна решить, какой pbuffer ей подходит в данной ситуации: pbuffer максимальной длины, pbuffer максимальной ширины, pbuffer максимальной площади, квадратный pbuffer максимальной площади или какой-нибудь другой.

Для того чтобы pbuffer можно было использовать как обыкновенное окно, для него необходимо создать контекст устройства командой wglGetPbufferDCARB, принимающей в качестве параметра дескриптор буфера pbuffer:

```
HDC wglGetPbufferDCARB(HPBUFFERARB hPbuffer);
```

Следующий этап — создание контекста OpenGL для pbuffer. Здесь у программиста есть выбор из двух вариантов.

- Создать для pbuffer собственный контекст OpenGL при помощи команды wglCreateContext. В этом случае pbuffer будет полностью независимым окном со своим набором матриц, текстур, атрибутов и т. д. Если же буферу pbuffer необходимо будет получить доступ к текстурным объектам или дисплейным спискам другого окна, то необходимо воспользоваться командой wglShareLists. Недостатком данного способа является необходимость частого переключения контекстов OpenGL. Такие переключения — очень медленная операция.

- Использовать для работы с буфером pbuffer один из существующих контекстов OpenGL, например, контекст главного окна программы. В этом случае окно, контекст которого использует программа, и pbuffer будут использовать общие ресурсы OpenGL: текстурные объекты, дисплейные списки, матрицы текстур и т. д. Например, изменение матрицы проекции окна приведет к изменению матрицы проекции в буфере pbuffer. Недостатком этого варианта является необходимость совпадения форматов пикселов буфера pbuffer и пикселов окна, которое он использует. Согласитесь, это сильно ограничивает область применения данного метода.

Далее программа может работать с буфером pbuffer, как с обыкновенным окном. Но программист всегда должен помнить о том, что pbuffer использует один из самых ценных ресурсов видеокарты — видеопамять. Поэтому следует всегда удалять ненужные буферы pbuffer, которые больше не используются программой. Удаление буфера pbuffer происходит в 2 этапа:

1. Освобождение контекста буфера pbuffer, полученного командой wglGetPbufferDCARB, при помощи команды wglReleasePbufferDCARB
2. Удаление собственно буфера pbuffer командой wglDestroyPbufferARB.

Определение этих команд приведено ниже:

```
int wglReleasePbufferDCARB(HPBUFFERARB hPbuffer,
HDC hDC);
BOOL wglDestroyPbufferARB(HPBUFFERARB hPbuffer);
```

где:

- hPbuffer — дескриптор буфера pbuffer;
- hDC — контекст буфера pbuffer.

В начале раздела уже говорилось о том, что у буфера pbuffer есть важное свойство: он может быть потерян при смене видеорежима. При этом следует помнить, что смена видеорежима может произойти спонтанно в любой момент времени: из-за случайного нажатия клавиш <Alt>+<Tab> или <Ctrl>+ +<Alt>+<Del>, появления всплывающего окна какой-нибудь резидентной программы, временного перехода в текстовый режим в момент запуска консольного приложения, ухода компьютера в "спящий режим" и т. д. Поэтому приложение обязательно должно отслеживать ситуацию потери буфера pbuffer и, при необходимости, восстанавливать его.

Для проверки потери буфера pbuffer используется команда wglQueryPbufferARB с параметром WGL_PBUFFER_LOST_ARB, которая была рассмотрена выше.

Если pbuffer был потерян, то OpenGL будет игнорировать все попытки модификации буфера pbuffer. В этом случае для того чтобы "привести OpenGL в чувство", программа должна уничтожить буфер пикселов, а затем

создать его заново. Разумеется, в ходе этих действий будут полностью потеряны содержимое буфера `pbuffer` и информация о его состоянии. Поэтому программе, вероятно, придется выполнить повторную инициализацию буфера `pbuffer`.

Теперь настало время закрепить все полученные знания на практике. Для этого мы напишем простой пример (Ex02), который рисует в буфере `pbuffer` вращающийся чайник с последующим копированием содержимого буфера `pbuffer` на экран (рис. 8.1).

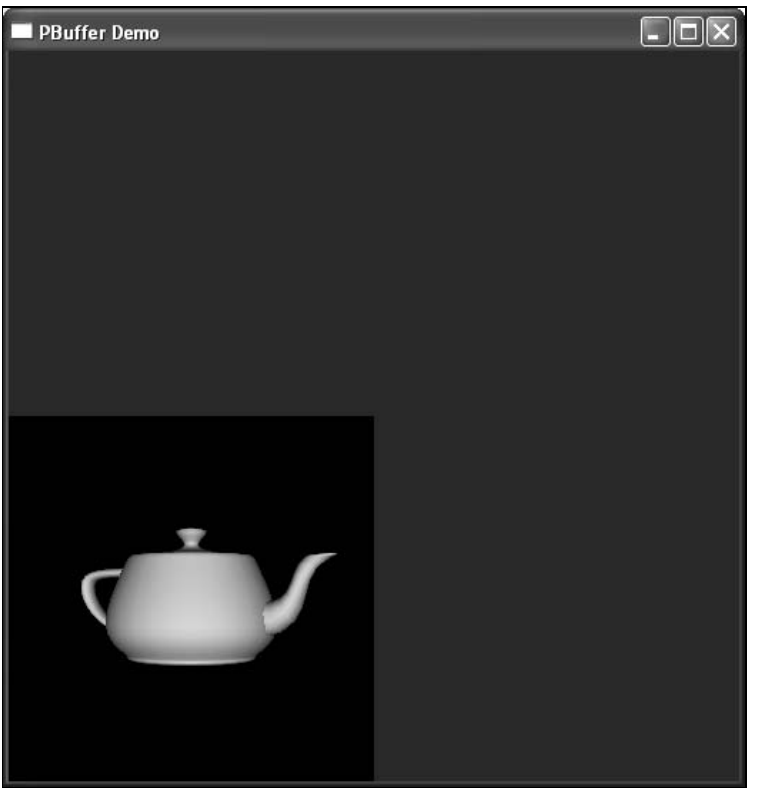


Рис. 8.1. Вращающийся чайник, рисующийся в буфере `pbuffer`

Исходный текст примера Ex02 приведен в листинге 8.2.

Листинг 8.2

```
#include <windows.h>
#include <iostream>
```

```
#include <gl/glut.h>
#include <gl/glext.h>
#include <gl/wglext.h>

#define GLH_EXT_SINGLE_FILE
#include <glh_nveb.h>
#include <glh_extensions.h>
#include "glh_obs.h"
#include <fstream>

using namespace std;

// Максимальное количество атрибутов в формате пикселов
const MAX_ATTRIBS=32;
// Максимальное число анализируемых форматов пикселов
const MAX_PFORMATS=256;
// Ширина и высота буфера pbuffer
const pbuffer_width=256;
const pbuffer_height=256;

// Дескриптор буфера pbuffer
HPBUFFERARB hbuffer1;
// Контекст буфера pbuffer
HDC hpbufdc1;
// Контекст OpenGL pbuffer
HGLRC pbufglctx1;
// Массив целочисленных атрибутов
int iattributes[2*MAX_ATTRIBS];
// Массив атрибутов с плавающей точкой
float fattributes[2*MAX_ATTRIBS];
// Массив доступных форматов пикселов
int pformat[MAX_PFORMATS];
// Количество доступных форматов пикселов
unsigned int nformats;

// Буфер для временного хранения изображения, копируемого из буфера
// pbuffer
GLubyte* imgdata=0;
```

```
// Идентификатор главного окна программы
int WinID;
// Запоминаем момент запуска программы
int Time=GetTickCount();

// Инициализируем буфер pbuffer
void InitPBuffer()
{
// Делаем буфер pbuffer текущим окном
       wglMakeCurrent(hpbufdc1, pbufglctx1);

// Запрашиваем размер буфера pbuffer
       wglQueryPbufferARB(hbuffer1, WGL_PBUFFER_WIDTH_ARB,
&pbuffer_width);
    wglQueryPbufferARB(hbuffer1, WGL_PBUFFER_HEIGHT_ARB,
&pbuffer_height);

// Если буфер для хранения изображения из буфера pbuffer уже был
// создан — удаляем его
       if (imgdata)
              delete[] imgdata;

// Создаем новый буфер для хранения изображения
       imgdata=new GLubyte[pbuffer_width*pbuffer_height*4];

// Инициализируем pbuffer
       glEnable(GL_DEPTH_TEST);
       glEnable(GL_LIGHTING);
       glEnable(GL_LIGHT0);
       glClearColor(0, 0, 0, 0);

       glMatrixMode(GL_PROJECTION);
       glLoadIdentity();
       gluPerspective(30, 1, 1, 10);
       glMatrixMode(GL_MODELVIEW);
       glTranslatef(0,0,-4);

// Активизируем главное окно программы
       glutSetWindow(WinID);
}
```

```
void Init()
{
// Проверяем поддержку необходимых расширений
       if (!glh_init_extensions("WGL_ARB_pbuffer WGL_ARB_pixel_format"))
       {
              MessageBox(NULL, glh_get_unsupported_extensions(),
"Unsupported  extensions", MB_OK | MB_ICONSTOP);
              exit(-1);
       }

// Получаем идентификатор текущего окна GLUT
       WinID=glutGetWindow();
// Получаем контекст текущего окна
       HDC hdc=wglGetCurrentDC();

// Номер текущего атрибута
       int niattribs=0;
       int nfattribs=0;
// Обнуляем список атрибутов
       for (int i=0; i<2*MAX_ATTRIBS; i++)
       {
              iattributes[i]=0;
              fattributes[i]=0;
       }
// Заносим необходимые атрибуты
// Формат пикселов должен быть совместимым с pbuffer
       iattributes[2*niattribs]=WGL_DRAW_TO_PBUFFER_ARB;
       iattributes[2*niattribs+1]=true;
       niattribs++;
// Нужна поддержка 24-битного буфера глубины
       iattributes[2*niattribs]=WGL_DEPTH_BITS_ARB;
       iattributes[2*niattribs+1]=24;
       niattribs++;

// Необходима поддержка 32-битного цвета (8, 8, 8, 8)
       iattributes[2*niattribs]=WGL_RED_BITS_ARB;
       iattributes[2*niattribs+1]=8;
       niattribs++;
```

```
        iattributes[2*niattribs]=WGL_GREEN_BITS_ARB;
        iattributes[2*niattribs+1]=8;
        niattribs++;

        iattributes[2*niattribs]=WGL_BLUE_BITS_ARB;
        iattributes[2*niattribs+1]=8;
        niattribs++;

        iattributes[2*niattribs]=WGL_ALPHA_BITS_ARB;
        iattributes[2*niattribs+1]=8;
        niattribs++;


// Получаем список доступных форматов пикселов
        if (!wglChoosePixelFormatARB(hdc, iattributes, fattributes,
MAX_PFORMATS, pformat, &nformats))
        {
                MessageBox(NULL, "PBuffer creating error: request pixel
format is unsupported",
                                    "Unsupported pixel format", MB_OK |
MB_ICONSTOP);
                exit(-1);
        }


// Если не было найдено ни одного подходящего формата
if ( nformats <= 0 )
{
          MessageBox(NULL, "pbuffer creation error:  Couldn't find a
suitable pixel format.",
                                "Unsupported pixel format", MB_OK |
MB_ICONSTOP);
         exit(-1);
}
        iattributes[0]=0;


// Создаем буфер pbuffer
        hbuffer1=wglCreatePbufferARB(hdc, pformat[0], pbuffer_width,
pbuffer_height, iattributes);
// Получаем контекст устройства буфера pbuffer
        hpbufdc1=wglGetPbufferDCARB(hbuffer1);
// Получаем контекст OpenGL буфера pbuffer
        pbufglctx1=wglCreateContext(hpbufdc1);
```

```
// Инициализируем буфер pbuffer
InitPBuffer()
// Инициализируем главное окно программы
       glClearColor(0.1, 0.1, 0.7, 0);
       glMatrixMode(GL_PROJECTION);
       glLoadIdentity();
       glOrtho(0, 1, 0, 1, -1, 1);
       glMatrixMode(GL_MODELVIEW);
}

void Display()
{
       GLint PBufferLost;
// Проверяем потерю буфера pbuffer
       wglQueryPbufferARB(hbuffer1, WGL_PBUFFER_LOST_ARB, &PBufferLost);
// Если буфер pbuffer потерян
       if (PBufferLost)
       {
// Уничтожаем буфер pbuffer
              wglDeleteContext(pbufglctx1);
              wglReleasePbufferDCARB(hbuffer1, hpbufdc1);
              wglDestroyPbufferARB(hbuffer1);

// Создаем новый буфер pbuffer
              HDC hdc=wglGetCurrentDC();
              hbuffer1=wglCreatePbufferARB(hdc, pformat[0],
pbuffer_width, pbuffer_height, iattributes);
              hpbufdc1=wglGetPbufferDCARB(hbuffer1);
              pbufglctx1=wglCreateContext(hpbufdc1);

// Инициализируем буфер pbuffer
              InitPBuffer();
       }

// Активизируем буфер pbuffer
       wglMakeCurrent(hpbufdc1, pbufglctx1)

       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
       glPushMatrix();
```

```
// Рисуем вращающийся чайник
              glColor3f(0.5, 0.5, 0.5);
              glRotatef(GLfloat((GetTickCount()-Time))/10, 0, 1, 0);
              glutSolidTeapot(0.5);
// Копируем полученное изображение в буфер
              glReadPixels(0, 0, pbuffer_width, pbuffer_height, GL_RGBA,
GL_UNSIGNED_BYTE, imgdata);
       glPopMatrix();
// Активизируем главное окно программы
       glutSetWindow(WinID);

       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Выводим содержимое буфера pbuffer в левом нижнем углу экрана.
       glRasterPos2i(0,0);
       glDrawPixels(pbuffer_width, pbuffer_height, GL_RGBA,
GL_UNSIGNED_BYTE, imgdata);

       glutSwapBuffers();
}

void Keyboard(unsigned char k, int x, int y)
{
       switch (k)
       {
// Если нажата клавиша <ESC>
              case VK_ESCAPE:
// Удаляем временный буфер
                     if (imgdata)
                            delete[] imgdata;
 // Удаляем контекст OpenGL
                     wglDeleteContext(pbufglctx1);
// Освобождаем контекст устройства буфера pbuffer
                     wglReleasePbufferDCARB(hbuffer1, hpbufdc1);
// Удаляем буфер pbuffer
                     wglDestroyPbufferARB(hbuffer1);
// Выходим из программы
                     exit(0);
       }
}
```

```
void Idle()
{
// Если система бездействует — перерисовываем экран
       glutPostRedisplay();
}

int main()
{
       glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGBA);
       glutInitWindowSize(512, 512);
       glutInitWindowPosition((GetSystemMetrics(SM_CXSCREEN)-512)/2,
(GetSystemMetrics(SM_CYSCREEN)-512)/2);
       glutCreateWindow("PBuffer Demo");

       Init();

       glutDisplayFunc(Display);
       glutIdleFunc(Idle);
       glutKeyboardFunc(Keyboard);

       glutMainLoop();
}
```

Как видно из этого примера, работать с буфером `pbuffer` при помощи функций WGL довольно неудобно — одна только инициализация буфера `pbuffer` занимает более 70 строк. К счастью, в составе NVIDIA OpenGL SDK имеется класс `PBuffer`, значительно облегчающий использование буфера `pbuffer`. Этот класс будет рассмотрен в следующем разделе.

8.2.1. Класс *PBuffer*

Как вы, наверное, уже заметили, работа с буфером `pbuffer` состоит из следующих этапов:

1. Создание буфера `pbuffer` с сохранением его дескриптора и контекстов.
2. Использование буфера `pbuffer`. При этом все команды, использующие `pbuffer`, принимают в качестве параметра дескриптор и/или контексты буфера `pbuffer`.
3. Обязательное удаление буфера `pbuffer` перед завершением работы программы.

Таким образом, буфер `pbuffer` является идеальным кандидатом для инкапсуляции в классе. Не удивительно, что в состав NVIDIA OpenGL SDK уже входит класс `PBuffer`, инкапсулирующий всю работу с буфером `pbuffer`.

Исходный текст класса `pbuffer` расположен в файлах \NVSDK\OpenGL\include\shared\pbuffer.h и \NVSDK\OpenGL\src\shared\pbuffer.cpp. Перед использованием класса `PBuffer` необходимо скопировать оба этих файла в каталог с проектом. После этого первый файл подключается к проекту директивой `#include`, а второй — командой меню **Project | Add Existing Item**.

Ниже приведено определение класса `PBuffer` из файла pbuffer.h.

```
class PBuffer
{
private:
// Набор битовых флагов, задающих тип буфера PBuffer
    unsigned int mode;
// Разделяется ли контекст буфера pbuffer с другим окном
    bool         sharedctx;
// Разделяются ли дисплейные списки буфера pbuffer с другими контекстами
// OpenGL
    bool         sharedlists;
// Если проект компилируется под Win32
#ifdef _WIN32
// Контекст устройства буфера pbuffer
    HDC          myDC;
// Контекст OpenGL буфера pbuffer
    HGLRC        myGLctx;
// Дескриптор буфера pbuffer
    HPBUFFERARB  buffer;
// Иначе полагаем, что проект компилируется под Linux
#else
    Display *m_pDisplay;
    GLXPbuffer m_glxPbuffer;
    GLXContext m_glxContext;
#endif
// Ширина и высота буфера pbuffer
    int          width;
    int          height;
public:
// Конструктор буфера pbuffer
    PBuffer( int width, int height, unsigned int mode );
```

```
// Деструктор
    ~PBuffer();
// Проверяем потерю буфера pbuffer. Если pbuffer оказался потерян, то
// восстанавливаем его
    void HandleModeSwitch();
// Делаем pbuffer текущим окном
    void MakeCurrent();
// Создаем и инициализируем новый буфер pbuffer
    void Initialize(bool sharecontexts = true, bool sharelists = false);
// Возвращаем ширину буфера pbuffer
       int GetWidth()
       { return width; }
// Возвращаем высоту буфера pbuffer
       int GetHeight()
       { return height; }
};
```

Уже из определения класса `PBuffer` становится ясно, что он предназначен для использования буфера `pbuffer` как в операционных системах семейства Win32, так и в операционных системах семейства Linux. В последнем случае для работы с буфером `pbuffer` используются средства библиотеки GLX (аналог библиотеки WGL в Linux). Так как эта книга посвящена программированию в операционных системах Windows, мы не будем рассматривать использование этого класса в операционных системах Linux. Да в этом и нет особой необходимости: все детали реализации буфера `pbuffer`, различающиеся в операционных системах Windows и Linux, спрятаны в секции `private` класса `PBuffer`.

Рассмотрим процесс использования класса `PBuffer`. Как известно, создание любого экземпляра класса начинается с вызова его конструктора. Конструктор класса `PBuffer` объявлен следующим образом:

```
PBuffer( int width, int height, unsigned int mode );
```

где:

- `width` — ширина буфера `pbuffer`;
- `height` — высота буфера `pbuffer`;
- `mode` — атрибуты буфера `pbuffer`, которые хранятся в виде набора битовых флагов, идентификаторы которых приведены в табл. 8.1.

Таблица 8.1. *Идентификаторы атрибутов класса* `PBuffer`

Идентификатор	Описание
`GLUT_INDEX`	Задает индексный режим цвета буфера `pbuffer`
`GLUT_ALPHA`	Класс `PBuffer` должен поддерживать альфа-канал буфером `pbuffer`
`GLUT_DOUBLE`	Класс `PBuffer` должен поддерживать двойную буферизацию
`GLUT_DEPTH`	Класс `PBuffer` должен поддерживать буфер глубины
`GLUT_STENCIL`	Класс `PBuffer` должен поддерживать буфер шаблона
`GLUT_ACCUM`	Класс `PBuffer` должен поддерживать буфер аккумулятора

Ниже приведен исходный код конструктора класса `PBuffer`:

```
PBuffer::PBuffer( int w, int h, unsigned int mode ) : width(w),
height(h), mode(mode), myDC(NULL), myGLctx(NULL), buffer(NULL)
{
    sharedctx = false;
    sharedlists = false;
}
```

Как видно, конструктор только инициализирует поля класса `PBuffer`, но не создает собственно сам буфер `pbuffer`. Это обусловлено тем, что для создания буфера `pbuffer` классу `PBuffer` необходим доступ к контексту OpenGL. Если же экземпляр класса `PBuffer` будет статическим глобальным объектом, то он будет автоматически создан в момент запуска программы, до создания контекста OpenGL.

Для разрешения этого противоречия создание буфера `pbuffer` было выделено в отдельный метод `Initialize`:

```
void PBuffer::Initialize(bool sharecontexts, bool sharelists)
```

где:

- `sharecontexts` — флаг, указывающий на необходимость создания буфера `pbuffer`, разделяющего контекст OpenGL с текущим окном;
- `sharelists` — флаг, указывающий на необходимость создания буфера `pbuffer` с разделяющимися дисплейными списками.

Исходный текст метода `Initialize` с подробными комментариями приведен в листинге 8.3.

Листинг 8.3

```
void PBuffer::Initialize(bool sharecontexts, bool sharelists)
{
// Получаем контексты устройства и OpenGL
    HDC hdc = wglGetCurrentDC();
    HGLRC hglrc = wglGetCurrentContext();
    int format;
// Сохраняем значения параметров в полях
    sharedctx = sharecontexts;
    sharedlists = sharelists;

// Если создаем pbuffer с разделяемым контекстом OpenGL
    if ( sharedctx )
    {
// Запрашиваем формат пикселов текущего окна
           format = GetPixelFormat( hdc );
    }
    else
    {
// Запрашиваем формат пикселов на основе атрибутов, переданных
// конструктору

// Атрибуты формата пикселов
        int   iattributes[2*MAX_ATTRIBS];
        float fattributes[2*MAX_ATTRIBS];
        int nfattribs = 0;
        int niattribs = 0;

// Обнуляем атрибуты
        for ( int a = 0; a < 2*MAX_ATTRIBS; a++ )
            {
            iattributes[a] = 0;
            fattributes[a] = 0;
            }

// Устанавливаем атрибуты формата пикселов

// Формат пикселей должен быть совместим с буфером pbuffer
        iattributes[2*niattribs  ] = WGL_DRAW_TO_PBUFFER_ARB;
```

```
        iattributes[2*niattribs+1] = true;
        niattribs++;

// Если присутствует флаг GLUT_INDEX
        if ( mode & GLUT_INDEX )
            {
// Задаем индексный режим
            iattributes[2*niattribs  ] = WGL_PIXEL_TYPE_ARB;
            iattributes[2*niattribs+1] = WGL_TYPE_COLORINDEX_ARB;
            niattribs++;
            }
        else
            {
// Задаем RGBA-режим
            iattributes[2*niattribs  ] = WGL_PIXEL_TYPE_ARB;
            iattributes[2*niattribs+1] = WGL_TYPE_RGBA_ARB;
            niattribs++;
            }

        if ( mode & GLUT_ALPHA )
            {
// pbuffer должен поддерживать альфа-канал
            iattributes[2*niattribs  ] = WGL_ALPHA_BITS_ARB;
            iattributes[2*niattribs+1] = 1;
            niattribs++;
            }

        if ( mode & GLUT_DOUBLE )
            {
// pbuffer должен поддерживать двойную буферизацию
            iattributes[2*niattribs  ] = WGL_DOUBLE_BUFFER_ARB;
            iattributes[2*niattribs+1] = true;
            niattribs++;
            }

// Внимание: ошибка. Условие в операторе if всегда будет равно false,
// т. к. GLUT_SINGLE равен 0
        if (mode & GLUT_SINGLE)
            {
```

```
            iattributes[2*niattribs  ] = WGL_DOUBLE_BUFFER_ARB;
            iattributes[2*niattribs+1] = false;
            niattribs++;
            }

        if ( mode & GLUT_DEPTH )
            {
// Необходима двойная буферизация
            iattributes[2*niattribs  ] = WGL_DEPTH_BITS_ARB;
            iattributes[2*niattribs+1] = 1;
            niattribs++;
            }

        if ( mode & GLUT_STENCIL )
            {
// Необходим буфер шаблона
            iattributes[2*niattribs  ] = WGL_STENCIL_BITS_ARB;
            iattributes[2*niattribs+1] = 1;
            niattribs++;
            }

        if ( mode & GLUT_ACCUM )
            {
// Необходим буфер аккумулятора
            iattributes[2*niattribs  ] = WGL_ACCUM_BITS_ARB;
            iattributes[2*niattribs+1] = 1;
            niattribs++;
            }

// Формат пикселов должен быть совместим с OpenGL
        iattributes[2*niattribs  ] = WGL_SUPPORT_OPENGL_ARB;
        iattributes[2*niattribs+1] = true;
        niattribs++;

// Список совместимых форматов
        int pformat[MAX_PFORMATS];
        unsigned int nformats;
// Получаем список совместимых форматов
```

```
        if ( !wglChoosePixelFormatARB( hdc, iattributes, fattributes,
MAX_PFORMATS, pformat, &nformats ) )
            {
            fprintf( stderr, "pbuffer creation error:
wglChoosePixelFormatARB() failed.\n" );
            exit( -1 );
            }
// Если не найдено ни одного подходящего формата
           if ( nformats <= 0 )
            {
            fprintf( stderr, "pbuffer creation error:  Couldn't find a
suitable pixel format.\n" );
            exit( -1 );
            }
        format = pformat[0];
    }

    // Создаем pbuffer
    int properties[1] = { 0 };
    buffer = wglCreatePbufferARB( hdc, format, width, height, properties );
// Если pbuffer не получилось создать - выводим сообщение об ошибке
    if ( !buffer )
        {
        DWORD err = GetLastError();
        fprintf( stderr, "pbuffer creation error:  wglCreatePbufferARB()
failed\n" );
        if ( err == ERROR_INVALID_PIXEL_FORMAT )
            {
            fprintf( stderr, "error:  ERROR_INVALID_PIXEL_FORMAT\n" );
            }
        else if ( err == ERROR_NO_SYSTEM_RESOURCES )
            {
            fprintf( stderr, "error:  ERROR_NO_SYSTEM_RESOURCES\n" );
            }
        else if ( err == ERROR_INVALID_DATA )
            {
            fprintf( stderr, "error:  ERROR_INVALID_DATA\n" );
            }
        exit( -1 );
        }
```

```
// Получаем контекст устройства pbuffer-а
    myDC = wglGetPbufferDCARB( buffer );
// Если не удалось получить контекст
    if ( !myDC )
        {
        fprintf( stderr, "pbuffer creation error:  wglGetPbufferDCARB()
failed\n" );
        exit( -1 );
        }

// Если pbuffer разделяет контекст OpenGL с текущим окном
    if ( sharecontexts )
    {
// Используем контекст OpenGL текущего окна
           myGLctx = hglrc;
    }
    else
    {
// Создаем новый контекст OpenGL для буфера pbuffer
        myGLctx = wglCreateContext( myDC );
// Если не получилось создать новый контекст — выводим сообщение об
// ошибке
        if ( !myGLctx )
            {
            fprintf( stderr, "pbuffer creation error:  wglCreateContext()
failed\n" );
            exit( -1 );
            }

// Если pbuffer разделяет дисплейные списки с текущим окном
           if( sharelists )
                  {
// Выполняем команду wglShareLists
                  if( !wglShareLists(hglrc, myGLctx) )
                         {
                         fprintf( stderr, "pbuffer: wglShareLists()
failed\n" );
                         exit( -1 );
                         }
```

```
                }
    }

// Получаем размеры созданного буфера pbuffer
    wglQueryPbufferARB( buffer, WGL_PBUFFER_WIDTH_ARB, &width );
    wglQueryPbufferARB( buffer, WGL_PBUFFER_HEIGHT_ARB, &height );

    fprintf( stderr, "Created a %d x %d pbuffer\n", width, height );
}
```

Для активизации буфера `pbuffer` используется метод `MakeCurrent`, являющийся просто оболочкой над командой `wglMakeCurrent`:

```
void PBuffer::MakeCurrent()
{
    if ( !wglMakeCurrent( myDC, myGLctx ) )
        {
        fprintf( stderr, "PBuffer::MakeCurrent() failed.\n" );
        exit( -1 );
        }
}
```

Для восстановления потерянного буфера `pbuffer` в классе `PBuffer` имеется метод `HandleModeSwitch`, который в случае потери буфера `pbuffer` автоматически пересоздает его.

```
void PBuffer::HandleModeSwitch()
{
    int lost = 0;

    wglQueryPbufferARB( buffer, WGL_PBUFFER_LOST_ARB, &lost );

    if ( lost )
        {
        this->~PBuffer();
        Initialize( sharedctx, sharedlists );
        }
}
```

У метода `HandleModeSwitch` есть один существенный недостаток — он не возвращает никакого результата. При помощи метода `HandleModeSwitch` не-

возможно установить, был ли потерян pbuffer, а поле дескриптора buffer, необходимое для команды wglQueryPbufferARB, расположено в секции private, и поэтому является закрытым.

Этот недостаток фактически сводит к нулю все достоинства класса PBuffer, делая невозможным его использование в реальных программах. Ведь после восстановления буфера pbuffer его, как правило, надо проинициализировать, а для этого надо иметь возможность определить факт потери буфера pbuffer.

Существует два способа борьбы с этим недостатком.

1. Модификация исходного кода класса PBuffer. Довольно эффективный метод, имеющий, однако, серьезный недостаток — для модификации исходного кода класса PBuffer необходимо получить разрешение NVIDIA. Кроме того, возможны проблемы совместимости с будущими версиями NVIDIA OpenGL SDK.
2. Получить доступ к полю дескриптора buffer в секции private при помощи "махинаций" с указателями. Этот метод тоже имеет недостатки, но они менее значительные, чем в случае с первым методом.

Взвесив все "за" и "против", я решил остановиться на втором варианте — доступе к полю buffer в секции private с использованием указателей. Но как это реализовать?

Первое, что приходит в голову, — определить опытным путем смещение поля buffer относительно начала класса и считывать его значение по полученному адресу. У этого способа есть один существенный недостаток — он предполагает, что компилятор C++ хранит поля класса в определенном порядке. Но этот порядок никак не стандартизирован — в стандарте C++ говориться о том, что программист не должен делать никаких предположений о том, каким именно образом компилятор размещает поля объекта в памяти. Поэтому программа, написанная с использованием этого метода, будет, скорее всего, работать только с компиляторами Microsoft, да и то не со всеми. Даже компилятор Microsoft Visual C++ .NET может по-разному размещать поля в зависимости от опции выравнивания (Struct Member Alignment), режимов компиляции Debug и Release и т. д. В результате этот метод можно использовать, разве что для небольших программ, но никак не для примеров книги.

К счастью, существует и второй вариант решения этой проблемы. Для этого необходимо создать копию класса PBuffer, например, класс PublicPBuffer, которая будет отличаться от оригинала лишь одной деталью — все поля этого класса должны находиться в секции public:

```
class PublicPBuffer
{
public:
    unsigned int mode;
```

```
    bool         sharedctx;
    bool         sharedlists;

#ifdef _WIN32
    HDC          myDC;
    HGLRC        myGLctx;
    HPBUFFERARB  buffer;
#else
    Display *m_pDisplay;
    GLXPbuffer m_glxPbuffer;
    GLXContext m_glxContext;
#endif

    int          width;
    int          height;

       PublicPBuffer( int width, int height, unsigned int mode ) {};
       ~PublicPBuffer() {};
       void HandleModeSwitch() {};
       void MakeCurrent() {};
       void Initialize(bool sharecontexts = true, bool sharelists =
false) {};

       int GetWidth() {}

       int GetHeight() {}
};
```

Так как класс `PBuffer` является фактически точной копией класса `PublicPBuffer`, компилятор с вероятностью 99,9% (к сожалению, 100%-ную гарантию здесь никто не даст) разместит поля этих классов по одинаковым адресам. В результате мы сможем получить доступ к полю `buffer` при помощи операции преобразования типа указателя:

```
((PublicPBuffer*) (&pbuffer1))->buffer
```

Этим методом можно воспользоваться для получения доступа к любым полям класса `PBuffer`, в частности, к полям `width` и `height`, в которых хранится информация о ширине и высоте буфера `pbuffer`. К счастью, в этом нет необходимости — заботливые разработчики предусмотрели отдельные методы для доступа к этим полям:

```
int GetWidth() { return width; }
int GetHeight() { return height; }
```

Для демонстрации использования класса `PBuffer` я переписал пример Ex02 с использованием этого класса. Исходный код измененного примера приведен в листинге 8.4.

Листинг 8.4

```
...
// Объект буфера pbuffer
PBuffer pbuffer1(256, 256, GLUT_SINGLE | GLUT_DEPTH);
// Буфер для временного хранения изображения из pbuffer
GLubyte* imgdata=0;
// Идентификатор главного окна
int WinID;
// Время запуска программы
int Time=GetTickCount();

// Фиктивный класс для доступа к private-полям класса PBuffer
class PublicPBuffer
{
public:
    unsigned int mode;
    bool         sharedctx;
    bool         sharedlists;
#ifdef _WIN32
    HDC          myDC;
    HGLRC        myGLctx;
    HPBUFFERARB  buffer;
#else
    Display *m_pDisplay;
    GLXPbuffer m_glxPbuffer;
    GLXContext m_glxContext;
#endif

    int          width;
    int          height;

        PublicPBuffer( int width, int height, unsigned int mode ) {};
        ~PublicPBuffer() {};
        void HandleModeSwitch() {};
```

```
        void MakeCurrent() {};
        void Initialize(bool sharecontexts = true, bool sharelists =
false) {};

        int GetWidth() {}

        int GetHeight() {}
};

// Инициализируем контекст OpenGL буфера pbuffer
void InitPBuffer()
{
        if (imgdata)
                delete[] imgdata;
        imgdata=new GLubyte[pbuffer1.GetWidth()*pbuffer1.GetHeight()*4];
// Делаем буфер pbuffer текущим
        pbuffer1.MakeCurrent();

        glEnable(GL_DEPTH_TEST);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glClearColor(0, 0, 0, 0);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(30, 1, 1, 10);
        glMatrixMode(GL_MODELVIEW);
        glTranslatef(0,0,-4);

        glutSetWindow(WinID);
}

void Init()
{
        if (!glh_init_extensions("WGL_ARB_pbuffer WGL_ARB_pixel_format"))
        {
                MessageBox(NULL, glh_get_unsupported_extensions(),
"Unsupported  extensions", MB_OK | MB_ICONSTOP);
                exit(-1);
        }
```

```
// Создаем pbuffer
        pbuffer1.Initialize(false);
// Инициализируем контекст буфера pbuffer
        InitPBuffer();

// Инициализируем контекст главного окна
        glClearColor(0.1, 0.1, 0.7, 0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        glOrtho(0, 1, 0, 1, -1, 1);
        glMatrixMode(GL_MODELVIEW);
}

void Display()
{
// Проверяем потерю буфера pbuffer
        GLint PBufferLost;
        wglQueryPbufferARB(((PublicPBuffer*) (&pbuffer1))->buffer,
WGL_PBUFFER_LOST_ARB, &PBufferLost);
// Если pbuffer потерян
        if (PBufferLost)
        {
// Восстанавливаем pbuffer после потери
                pbuffer1.HandleModeSwitch();
// Переинициализируем контекст OpenGL буфера pbuffer
                InitPBuffer();
        }
// Делаем буфер pbuffer активным
        pbuffer1.MakeCurrent();
// Рисуем изображение в буфере pbuffer
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glPushMatrix();
                glColor3f(0.5, 0.5, 0.5);
                glRotatef(GLfloat((GetTickCount()-Time))/10, 0, 1, 0);
                glutSolidTeapot(0.5);
// Копируем изображение из буфера pbuffer в другой буфер
                glReadPixels(0, 0, pbuffer1.GetWidth(),
pbuffer1.GetHeight(), GL_RGBA, GL_UNSIGNED_BYTE, imgdata);
        glPopMatrix();
```

```
// Активизируем главное окно программы
        glutSetWindow(WinID);

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Выводим изображение из буфера pbuffer на экран
        glRasterPos2i(0,0);
        glDrawPixels(pbuffer1.GetWidth(), pbuffer1.GetHeight(), GL_RGBA,
GL_UNSIGNED_BYTE, imgdata);

        glutSwapBuffers();
}

void Keyboard(unsigned char k, int x, int y)
{
        switch (k)
        {
                case VK_ESCAPE:
                        if (imgdata)
                                delete[] imgdata;
                        exit(0);
        }
}

void Idle()
{
        glutPostRedisplay();
}

int main()
{
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_DEPTH | GLUT_RGBA);
        glutInitWindowSize(512, 512);
        glutInitWindowPosition((GetSystemMetrics(SM_CXSCREEN)-512)/2,
(GetSystemMetrics(SM_CYSCREEN)-512)/2);
        glutCreateWindow("PBuffer Demo");

// Сохраняем идентификатор главного окна OpenGL
        WinID=glutGetWindow();
```

```
        Init();

        glutDisplayFunc(Display);
        glutIdleFunc(Idle);
        glutKeyboardFunc(Keyboard);

        glutMainLoop();

}
```

8.2.2. Моделирование виртуального экрана с использованием *pbuffer*

Как известно, во множестве трехмерных игр персонажи видят экраны мониторов, телевизоров, проекционных экранов и аналогичных устройств, способных отображать на виртуальном экране динамическое изображение. При этом содержимое виртуального экрана может даже зависеть от действий пользователя. Например, виртуальный монитор может быть соединен с виртуальной камерой, которой может управлять пользователь. В результате, перемещая виртуальную камеру, пользователь будет менять изображение на виртуальном экране.

Существует множество методов моделирования таких сцен. Но самым простым и интуитивно понятным является метод, основанный на использовании внеэкранных буферов. В этом случае рисование виртуального экрана выполняется в два этапа:

1. Рисование изображения на виртуальном мониторе в буфере пикселов.
2. Копирование полученного изображения в текстуру, с последующим наложением полученной текстуры на экран виртуального монитора, проектора и т. д.

Этот метод позволяет с легкостью моделировать как плоские, так и выпуклые экраны.

В этом разделе мы разработаем демонстрационную программу, выводящую на экран плоскость, на которой рисуется вращающийся деревянный чайник (Ex04) (рис. 8.2). При этом пользователь будет иметь возможность изменять как положение плоскости на экране, так и чайника на плоскости.

Итак, нам фактически предстоит разработать две программы:

- программу, рисующую на экране вращающийся деревянный чайник;
- программу, рисующую на экране плоскость с наложенной на нее тектурой.

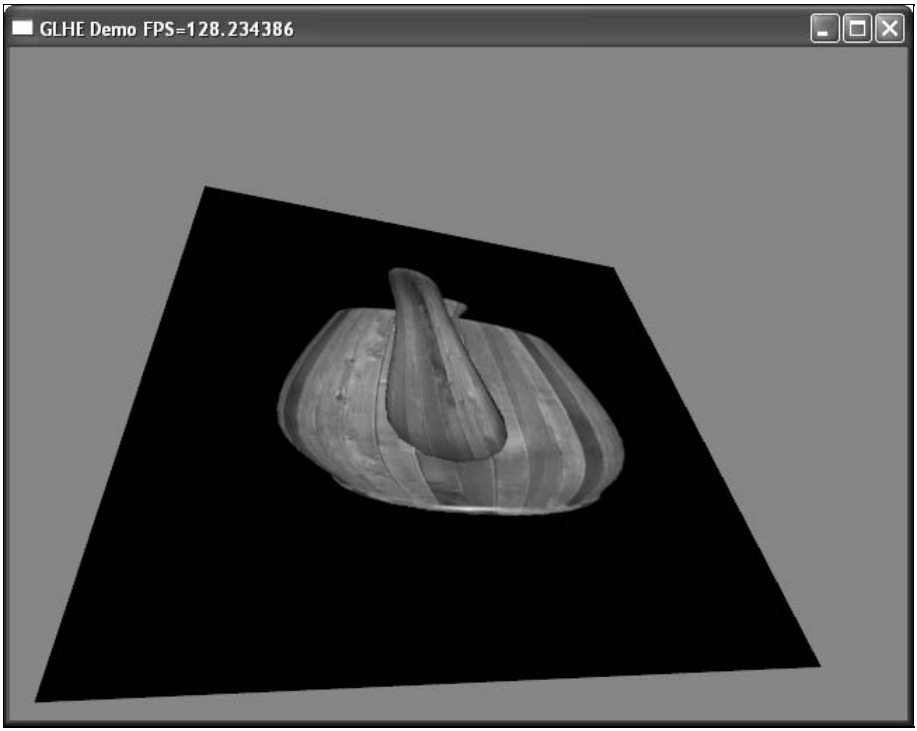


Рис. 8.2. Плоскость с вращающимся деревянным чайником

При этом содержимое экрана первой программы копируется в текстуру, которая подается на вход второй программы. И первая, и вторая программы должны предоставлять пользователю возможность изменять свое положение в пространстве. Но для избежания конфликтов они должны изменять положение наблюдателя различными способами. Ведь если обе программы при нажатии левой кнопки мыши будут одновременно изменять положение наблюдателя, то у пользователя пропадет возможность раздельного поворота каждого объекта. И поворот плоскости приведет к повороту чайника или наоборот. Один из вариантов бесконфликтного управления чайником и плоскостью приведен в табл. 8.2.

Таблица 8.2. *Управление объектами примера Ex04*

Объект	Вращение объекта	Перемещение в плоскости экрана	Перемещение вдоль оси Z в системе координат экрана
Плоскость	Левая кнопка мыши	<Shift> + левая кнопка мыши	<Ctrl> + левая кнопка мыши

Таблица 8.2 *(окончание)*

Объект	Вращение объекта	Перемещение в плоскости экрана	Перемещение вдоль оси Z в системе координат экрана
Чайник	Правая кнопка мыши	<Shift> + правая кнопка мыши	<Ctrl> + правая кнопка мыши

Написать каждую из приведенных программ по отдельности не составит труда. Первая программа должна будет состоять из совокупности интеракторов:

```
// Интерактор управления положением пользователя
glut_simple_user_interface pbuffer_user;
// Интерактор обрисовки содержимого экрана с последующим
// копированием в текстуру
glut_callbacks pbuffer_cb;
...
glut_add_interactor(&pbuffer_user);
glut_add_interactor(&pbuffer_cb);
```

Эта программа должна менять положение наблюдателя путем реакции на левую кнопку мыши. Но поскольку интерактор `glut_simple_user_interface` по умолчанию "умеет" использовать только левую кнопку мыши, нам придется перенастроить его вручную на использование правой кнопки мыши:

```
pbuffer_user.user_mouse.configure_buttons(1);
pbuffer_user.user_mouse.trackball.activate_on=GLUT_RIGHT_BUTTON;
pbuffer_user.user_mouse.dolly.activate_on=GLUT_RIGHT_BUTTON;
pbuffer_user.user_mouse.pan.activate_on=GLUT_RIGHT_BUTTON;
```

Интерактор `glut_simple_user_interface` имеет еще один подводный камень. Он содержит внутри себя интерактор `glut_perspective_reshaper`, который автоматически изменяет матрицу проекции и порт просмотра (viewport) буфера `pbuffer` при изменении размеров главного окна. Поэтому во избежание искажения изображения буфера `pbuffer` при изменении размеров главного окна необходимо отключить подинтерактор `reshaper`:

```
pbuffer_user.reshaper.disable();
```

Вторая программа отличается от первой фактически только наличием консоли:

```
// Интерактор консоли
glut_console console;
```

```
// Интерактор управления положением пользователя
glut_simple_user_interface window_user(&console);
// Интерактор рисования содержимого экрана
glut_callbacks window_cb;
// Интерактор переключения буферов экрана
glut_swapbuffers swapbuffers;
...
glut_add_interactor(&window_user);
glut_add_interactor(&window_cb);
glut_add_interactor(&console);
glut_add_interactor(&swapbuffers);
```

Первая программа работает при активном буфере `pbuffer`, а вторая — при активном главном окне. Следовательно, перед началом работы интеракторов первой программы текущим окном должен становиться `pbuffer`, а перед началом работы интеракторов второй программы — главное окно программы. Следовательно, возникает вопрос: в каких местах программы разместить переключения текущего окна? Интеракторы `pbuffer_cb` и `window_cb` не подходят для этой цели, т. к. они находятся не в самом начале цепочки интеракторов.

Самым красивым решением проблемы является создание нового интерактора, который при обработке события `display` активизирует заданное окно. Поместив такой интерактор в список интеракторов перед группами интеракторов первой и второй программы, мы легко решим проблему активизации нужного окна.

Итак, нам надо разработать интерактор, который мог бы "запоминать" контекст OpenGL текущего окна, а затем, при обработке события `display`, делать текущим этот контекст.

Интерактор, удовлетворяющий этим условиям, будет довольно тривиальным, поэтому я сразу приведу исходный текст интерактора с комментариями (листинг 8.5).

Листинг 8.5

```
class glut_dc_switcher:public glut_interactor
{
public:
// Контекст устройства
       HDC hdc;
// Контекст OpenGL
       HGLRC hglrc;
```

```
        glut_dc_switcher();
        void bind();
        void display();
};

// Конструктор.
glut_dc_switcher::glut_dc_switcher()
{
        hdc=0;
        hglrc=0;
};

// Запоминаем текущий контекст
void glut_dc_switcher::bind()
{
        hdc=wglGetCurrentDC();
        hglrc=wglGetCurrentContext();
}

// Переключаемся в запомненный контекст
void glut_dc_switcher::display()
{
        if (enabled)
                wglMakeCurrent(hdc, hglrc);
}
```

Так как интерактор `glut_dc_switcher` будет использоваться многими программами этой книги, его лучше всего поместить в библиотеку GLH_GLUT_EXT.

Теперь остается только преобразовать все приведенные выше рассуждения в программный код и получить готовую программу (листинг 8.6).

Листинг 8.6

```
#define __FULLSCREEN 1

#define STRICT
#define WIN32_LEAN_AND_MEAN
```

```
#include "glh_glut_ext.h"
#include "nv_util_ext.h"

#include "pbuffer.h"

// Ширина и высота окна
int WinWidth=640;
int WinHeight=480;

// Текстурный объект дерева
tex_object_2D wood;
// Изображение текстуры дерева
tga::tgaImage* pWoodImage;
// Текстурный объект виртуального экрана
tex_object_2D screen;
// Дисплейный список плоскости
display_list list0;
// Дисплейный список чайника
display_list list1;

// Интерактор консоли
glut_console console;
// Интерактор пользовательского интерфейса главного окна
glut_simple_user_interface window_user(&console);
// Интерактор пользовательского интерфейса буфера pbuffer
glut_simple_user_interface pbuffer_user;
// Пользовательский интерактор главного окна
glut_callbacks window_cb;
// Пользовательский интерактор буфера pbuffer
glut_callbacks pbuffer_cb;
// Меняем местами кадровые буферы
glut_swapbuffers swapbuffers;
// Делаем активным главное окно программы
glut_dc_switcher to_window;
// Делаем активным pbuffer
glut_dc_switcher to_pbuffer;

const string Title="GLHE Demo";
// Объект буфера pbuffer
PBuffer pbuffer0(512, 512, GLUT_RGBA | GLUT_DEPTH);
```

```
// Угол поворота чайника
float a=0;
// Момент времени, в который была запущена программа
float start_tick=GetTickCount();

// Класс-пустышка для доступа к private-полям класса PBuffer
class PublicPBuffer
{
...
};

// Инициализируем буфер pbuffer
bool InitPBuffer()
{
// Делаем буфер pbuffer текущим окном
       pbuffer0.MakeCurrent();
// Запоминаем контекст буфера pbuffer
       to_pbuffer.bind();

// Выводим информацию о формате пикселов буфера pbuffer
       console.add("");
       console.add("Pixel format of pbuffer");
       console.processCmd("pixel_format_info");
       console.add("");

// Настраиваем матрицу проекции и viewport
       pbuffer_user.reshaper.width=pbuffer0.GetWidth();
       pbuffer_user.reshaper.height=pbuffer0.GetHeight();
       pbuffer_user.reshaper.apply();

// Настройка параметров OpenGL
       glEnable(GL_DEPTH_TEST);
       glEnable(GL_LIGHTING);
       glEnable(GL_LIGHT0);
       glLightModeli(GL_LIGHT_MODEL_COLOR_CONTROL_EXT,
GL_SEPARATE_SPECULAR_COLOR_EXT);

       return true;
}
```

```
// Инициализируем OpenGL
bool Init()
{
// Проверяем поддержку необходимых расширений
       if (!glh_init_extensions("WGL_ARB_pbuffer WGL_ARB_pixel_format
GL_EXT_separate_specular_color GL_SGIS_generate_mipmap"))
       {
              console.add(string("Unsupported extension:
")+glh_get_unsupported_extensions());
              console.exit(-1);
              return false;
       }

// Запоминаем контекст главного окна программы
       to_window.bind();

// Создаем pbuffer с разделяемыми списками
       pbuffer0.Initialize(false, true);

// Настраиваем параметры OpenGL
       glEnable(GL_DEPTH_TEST);
       glClearColor(0.5, 0.5, 0.75, 1);

// Загружаем текстуру дерева
       pWoodImage=read("..\\wood.tga");
       if (!pWoodImage)
       {
              console.add("Can't load file \"..\\wood.tga\"");
              console.exit(-1);
              return false;
       }

// Настраиваем параметры текстуры дерева
       wood.bind();
       wood.parameter(GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
       wood.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
       texImage2D(pWoodImage, &wood);

// Настраиваем параметры текстуры главного окна программы
       screen.bind();
```

```
        screen.parameter(GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        screen.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Включаем автоматическую генерацию mipmap-уровней
        screen.parameter(GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
// Загружаем в видеопамять пустую текстуру размером с pbuffer
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, pbuffer0.GetWidth(),
pbuffer0.GetHeight(), 0, GL_RGB, GL_UNSIGNED_BYTE, 0);

// Выгружаем текстуру дерева из видеопамяти
        clear(pWoodImage);

// Создаем дисплейный список плоскости
        list0.new_list(GL_COMPILE);
                screen.bind();
                screen.enable();
                glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_REPLACE);
                glBegin(GL_QUADS);
                        glTexCoord2f(0, 0);
                        glVertex3f(-1, -1, 0);


                        glTexCoord2f(0, 1);
                        glVertex3f(-1,  1, 0);

                        glTexCoord2f(1, 1);
                        glVertex3f( 1,  1, 0);

                        glTexCoord2f(1, 0);
                        glVertex3f( 1, -1, 0);
                glEnd();
                screen.disable();
        list0.end_list();

// Создаем дисплейный список чайника
        list1.new_list(GL_COMPILE);
                glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_MODULATE);
                wood.bind();
                wood.enable();
```

```
                float DiffuseColor[4]={1.0, 1.0, 1.0, 1.0};
                float SpecularColor[4]={1.0, 1.0, 1.0, 1.0};
                glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
&DiffuseColor[0]);
                glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR,
&SpecularColor[0]);
                glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 128);
                glutSolidTeapot(2);
                wood.disable();
        list1.end_list();

// Инициализируем pbuffer
        InitPBuffer();

        return true;
}

// Рисуем на экране плоскость с наложенной текстурой
void WindowDisplay()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        list0.call_list();
}

// Рисуем в буфере pbuffer чайник
void PBufferDisplay()
{
// Проверяем потерю буфера pbuffer
        GLint PBufferLost;
        wglQueryPbufferARB(((PublicPBuffer*) (&pbuffer0))->buffer,
WGL_PBUFFER_LOST_ARB, &PBufferLost);
        if (PBufferLost)
        {
// Пересоздаем pbuffer в случае его потери
                console.add("Pbuffer was lost. Recreating...");
                pbuffer0.HandleModeSwitch();

                InitPBuffer();
        }
```

```
// Рисуем чайник на черном фоне
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glRotatef(a, 0, 1, 0);

        list1.call_list();

// Копируем полученное изображение чайника в текстуру
        screen.bind();
        glCopyTexSubImage2D(screen.target, 0, 0, 0, 0, 0,
pbuffer0.GetWidth(), pbuffer0.GetHeight());
}

// Поворачиваем чайник в момент бездействия программы
void PBufferIdle()
{
        a=float(GetTickCount()-start_tick)/50.0;
        if (a>360.0)
                a-=360.0;

        glutPostRedisplay();
}

int main(int argc, char* argv[])
{
        console.add("Initializing GLUT...");
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
        glutInitWindowSize(WinWidth,WinHeight);
        glutInitWindowPosition((glutGet(GLUT_SCREEN_WIDTH)-WinWidth)/2,
                                        (glutGet(GLUT_SCREEN_HEIGHT)-
WinHeight)/2);
#ifdef __FULLSCREEN
        glutGameModeString("1024x768@85");
        glutEnterGameMode();
#else
        glutCreateWindow(Title.c_str());
#endif

        console.add("Initializing OpenGL Helper Library...");
        glut_helpers_initialize();
```

```
        console.add("Initializing OpenGL...");
        Init();
        if (console.exiting)
                glutMainLoop();

        console.title=Title;

// Настраиваем интерактор пользовательского интерфейса главного окна
        window_user.user_mouse.configure_buttons(1);
        window_user.user_mouse.dolly.dolly[2]=-3;
        window_user.bAutoOffIdle=false;

// Настраиваем интерактор пользовательского интерфейса
        pbuffer_user.user_mouse.configure_buttons(1);
        pbuffer_user.user_mouse.dolly.dolly[2]=-6;
        pbuffer_user.user_mouse.trackball.activate_on=GLUT_RIGHT_BUTTON;
        pbuffer_user.user_mouse.dolly.activate_on=GLUT_RIGHT_BUTTON;
        pbuffer_user.user_mouse.pan.activate_on=GLUT_RIGHT_BUTTON;
        pbuffer_user.reshaper.disable();

// Обработчики событий
        window_cb.display_function=WindowDisplay;
        pbuffer_cb.display_function=PBufferDisplay;
        pbuffer_cb.idle_function=PBufferIdle;

// Помещаем интеракторы в список интеракторов
        glut_add_interactor(&to_pbuffer);
        glut_add_interactor(&pbuffer_user);
        glut_add_interactor(&pbuffer_cb);
        glut_add_interactor(&to_window);
        glut_add_interactor(&window_user);
        glut_add_interactor(&window_cb);
        glut_add_interactor(&console);
        glut_add_interactor(&swapbuffers);

        glut_idle(true);

        console.processCmd("exec autoexec.cs");
```

```
        console.add("Initialize complete. Starting glutMainLoop");
        glutMainLoop();

        return 0;
}
```

Обратите внимание, что программа для создания Mipmap-уровней текстуры виртуального экрана использует расширение SGIS_generate_mipmap. Если бы мы использовали для этой цели команду `gluBuild2DMipmap`, то нам бы пришлось выполнять копирование содержимого буфера `pbuffer` из видеопамяти в оперативную память и обратно, что привело бы к значительной потере производительности.

В следующем примере мы создадим настоящий виртуальный компьютер (точнее, монитор от компьютера), на котором будет выполняться пример Ex12 из *главы 4* (рис. 8.3).

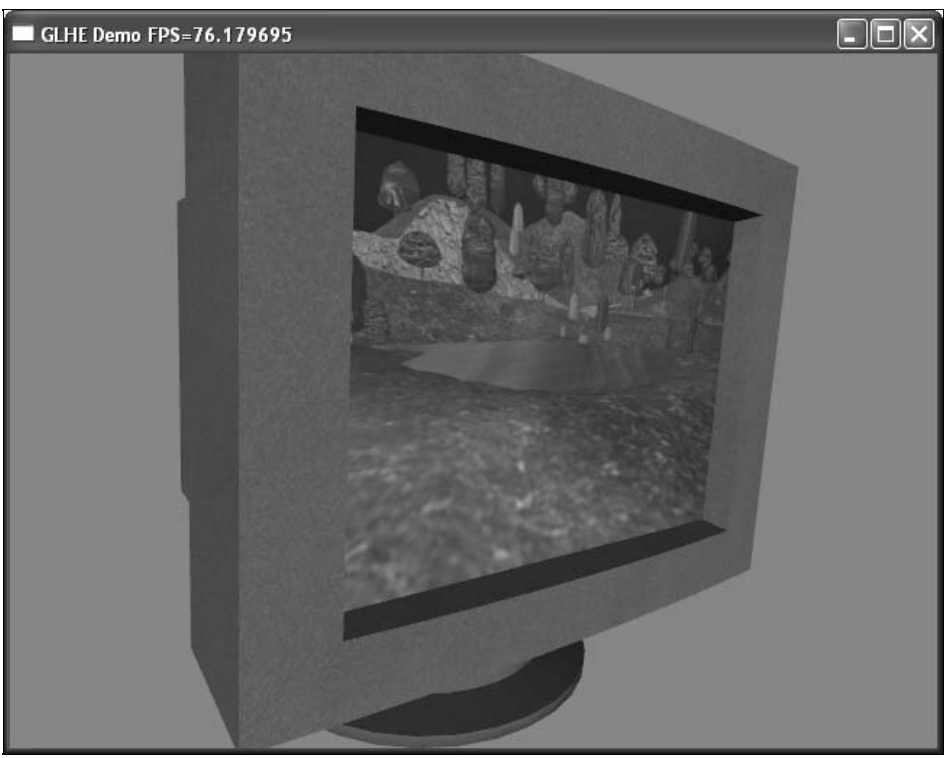


Рис. 8.3. Монитор виртуального компьютера, на котором выполняется пример Ex12 из *главы 4*

В качестве основы для нашего монитора мы возьмем стандартную модель монитора из файла \ModelLibrary\Media\MONITOR.max, выполненного в 3D Studio MAX. Внешний вид этого монитора изображен на рис. 8.4.

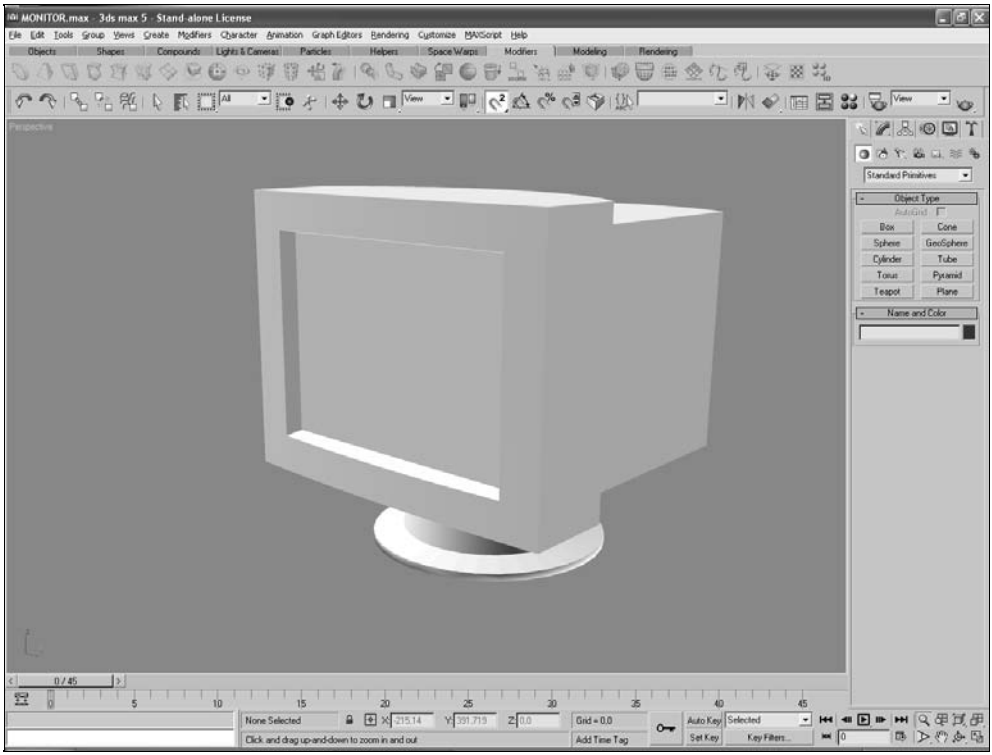


Рис. 8.4. Стандартная модель монитора, выполненная в 3D Studio MAX

Эта модель монитора в действительности состоит из трех объектов: Base (подставка от монитора), Monitor (корпус монитора) и Tv Tube (экран монитора). Для придания монитору более реалистичного вида можно наложить на подставку и корпус монитора текстуру пластика из файла \TextureLibrary\Plastics\GRPLAST2.JPG, а на экран монитора — текстуру отражения из файла \maps\Reflection\REFMAP.GIF. Обратите внимание, что на экран монитора не накладывается диффузная текстура. Это связано с тем, что на экране монитора будет отображаться содержимое виртуального экрана. Полученная модель монитора находится на CD-диске, прилагаемом к книге, в каталоге \Examples\Ch8-PBuffer\3DSMAX_MODELS\Monitor.

Теперь мы можем приступать к написанию программы, которая, аналогично примеру Ex04, будет состоять из двух больших частей: программы, рабо-

тающей на виртуальном компьютере, и программы, рисующей на экране модель монитора.

Ниже приведен исходный текст основных функций программы, рисующей на виртуальном мониторе долину, хранящуюся в файле valley.ase (листинг 8.7).

Листинг 8.7

```
display_list list1;
...
CAseModel model1;
...
glut_simple_user_interface pbuffer_user;
glut_callbacks pbuffer_cb;
glut_dc_switcher to_pbuffer;
...
// Инициализируем pbuffer
bool InitPBuffer()
{
        pbuffer0.MakeCurrent();
        to_pbuffer.bind();

        console.add("");
        console.add("Pixel format of pbuffer");
        console.processCmd("pixel_format_info");
        console.add("");

        pbuffer_user.reshaper.width=pbuffer0.GetWidth();
        pbuffer_user.reshaper.height=pbuffer0.GetHeight();
        pbuffer_user.reshaper.apply();

        glEnable(GL_DEPTH_TEST);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_COLOR_MATERIAL);
        glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);

        vec4 diffuse_color(0.7, 0.7, 0.7, 1);
        vec4 ambinet_color=vec4_one-diffuse_color+vec4(0, 0, 0, 1);
```

```
        glLightfv(GL_LIGHT0, GL_DIFFUSE, &diffuse_color[0]);
        glLightfv(GL_LIGHT0, GL_AMBIENT, &ambinet_color[0]);
//      glClearColor(0.5, 0.5, 0.75, 1);


        return true;
}
...
// Инициализируем OpenGL
bool Init()
{
// Проверяем поддержку необходимых расширений
        if (!glh_init_extensions("WGL_ARB_pbuffer WGL_ARB_pixel_format
GL_SGIS_generate_mipmap"))
        {
                console.add(string("Unsupported extension:
")+glh_get_unsupported_extensions());
                console.exit(-1);
                return false;
        }


        to_window.bind();
// Создаем pbuffer
        pbuffer0.Initialize(false, true);


...
// Загружаем модель долины из файла
        if (!model1.LoadModel("..\\models.zip", "valley.ase"))
        {
                console.add("Can't load model
..\\models.zip::valley.ase");
                console.exit(-1);
                return false;
        }
...
// Создаем дисплейный список с моделью долины
        list1.new_list(GL_COMPILE);
                glPushAttrib(GL_ALL_ATTRIB_BITS);
                glEnable(GL_NORMALIZE);
```

```
                glPushMatrix();
                model1.GetExtents(min, max);
                scale=max(fabs(min[0]), fabs(min[1]));
                scale=max(scale, fabs(min[2]));
                scale=max(scale, fabs(max[0]));
                scale=max(scale, fabs(max[1]));
                scale=max(scale, fabs(max[2]));
                scale=3.0/scale;
                glScalef(scale, scale, scale);
                model1.DrawModel(true);
                glPopMatrix();
                glPopAttrib();
        list1.end_list();
// Инициализируем pbuffer
        InitPBuffer();

        to_window.display();
...
// Выгружаем лишнюю информацию о модели долины из памяти
        model1.Clear(false);

        return true;
}

// Рисуем в pbuffer долину и копируем полученное изображение в текстуру
void PBufferDisplay()
{
//
        GLint PBufferLost;
        wglQueryPbufferARB(((PublicPBuffer*) (&pbuffer0))->buffer,
WGL_PBUFFER_LOST_ARB, &PBufferLost);
        if (PBufferLost)
        {
                console.add("Pbuffer was lost. Recreating...");
                pbuffer0.HandleModeSwitch();

                InitPBuffer();
        }
```

```
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        list1.call_list();

// Копирование изображения в текстуру
...
}

int main(int argc, char* argv[])
{
...
        Init();
        if (console.exiting)
                glutMainLoop();
...
        pbuffer_user.user_mouse.configure_buttons(1);
        pbuffer_user.user_mouse.dolly.dolly[2]=-9;
        pbuffer_user.user_mouse.trackball.activate_on=GLUT_RIGHT_BUTTON;
        pbuffer_user.user_mouse.dolly.activate_on=GLUT_RIGHT_BUTTON;
        pbuffer_user.user_mouse.pan.activate_on=GLUT_RIGHT_BUTTON;
        pbuffer_user.reshaper.disable();
...
        pbuffer_cb.display_function=PBufferDisplay;

        glut_add_interactor(&to_pbuffer);
        glut_add_interactor(&pbuffer_user);
        glut_add_interactor(&pbuffer_cb);
        glut_add_interactor(&to_window);
...
}
```

Как видно, этот фрагмент программы практически не отличается от примера Ex12 *главы 4*.

Полученное изображение необходимо скопировать в текстуру, которая будет наложена на экран монитора. Но поскольку на экран модели монитора не наложена диффузная текстура, нам придется вручную создать текстурный объект изображения на экране монитора и "встроить" его в модель монитора. Для этого необходимо выполнить следующие операции:

1. Загрузить модель монитора.
2. Создать новую текстуру (экземпляр класса `CTexture`).

3. Создать новый текстурный объект и настроить его параметры. Поместить в поле `pTextureObj` указатель на созданный экземпляр класса `CTexture`.
4. Поместить в список диспетчера текстур (`TextureManager`) экземпляр класса `CTexture`.
5. Найти в списке объектов модели монитора объект с названием `"Tv Tube"` и поместить в поле `diffuse_texture` этого объекта указатель на экземпляр класса `CTexture`.

Приведенный в листинге 8.8 фрагмент кода создает новый текстурный объект и встраивает его в модель монитора как диффузную текстуру экрана.

Листинг 8.8

```
// Указатель на текстурный объект экрана монитора
tex_object_2D* screen=0;
// Указатель на текстуру экрана монитора
CTexture* screen_texture=0;
// Дисплейный список монитора
display_list list0;

// Модель монитора
CAseModel model1;

// Диспетчер текстур (расположен в файле ase.cpp)
extern CTextureManager TextureManager;
...

bool Init()
{
...
// Загружаем модель монитора
       if (!model0.LoadModel("..\\monitor.zip", "monitor.ase"))
       {
              console.add("Can't load model
..\\monitor.zip::monitor.ase");
              console.exit(-1);
              return false;
       }

// Создаем новую текстуру для диспетчера текстур
       screen_texture=new CTexture;
```

```
// Счетчик ссылок
       screen_texture->count=1;
// Размеры текстуры как у pbuffer-а
       screen_texture->width=pbuffer0.GetWidth();
       screen_texture->height=pbuffer0.GetHeight();
// Имя несуществующего файла
       screen_texture->filename="::pbuffer0";

// Создаем новый текстурный объект
       screen=new tex_object_2D;
       screen->bind();
       screen->parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
       screen->parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
       screen->parameter(GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
// Создаем пустую текстуру размером с pbuffer
       glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, pbuffer0.GetWidth(),
pbuffer0.GetHeight(), 0, GL_RGB, GL_UNSIGNED_BYTE, 0);

// Привязываем текстурный объект к текстуре экрана
       screen_texture->pTextureObj=screen;

// Помещаем в список диспетчера текстур информацию о текстуре
       TextureManager.textures.push_back(*screen_texture);

// Перебираем все объекты сцены
    for (int n = 0; n < model0.numChunks; n++)
    {
// Если объект имеет имя "Tv Tube"
              if (*model0.data[n].name=="\"Tv Tube\"")
              {
// Помещаем в поле diffuse_texture указатель на диффузную текстуру
                     model0.data[n].diffuse_texture=screen_texture;
                     break;
              }
       }
```

После этого работа с полученным объектом монитора практически не отличается от работы с плоскостью в примере Ex04 (листинг 8.9).

Листинг 8.9

```
glut_console console;
glut_simple_user_interface window_user(&console);
glut_simple_user_interface pbuffer_user;
glut_callbacks window_cb;
glut_callbacks pbuffer_cb;
glut_swapbuffers swapbuffers;
glut_dc_switcher to_window;
glut_dc_switcher to_pbuffer;

bool Init()
{
...
// Создаем дисплейный список для монитора
       list0.new_list(GL_COMPILE);
              glPushAttrib(GL_ALL_ATTRIB_BITS);
              glEnable(GL_NORMALIZE);
              glPushMatrix();
              vec3f min, max;
              model0.GetExtents(min, max);
              float scale=max(fabs(min[0]), fabs(min[1]));
              scale=max(scale, fabs(min[2]));
              scale=max(scale, fabs(max[0]));
              scale=max(scale, fabs(max[1]));
              scale=max(scale, fabs(max[2]));
              scale=3.0/scale;
              glScalef(scale, scale, scale);
              model0.DrawModel(true);
              glPopMatrix();
              glPopAttrib();
       list0.end_list();
...
}
void WindowDisplay()
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Выводим монитор на экран
```

```
        list0.call_list();
}

void PBufferDisplay()
{
// Рисуем изображение в pbuffer
...
// Копируем изображение в текстуру
        screen->bind();
        glCopyTexSubImage2D(screen->target, 0, 0, 0, 0, 0,
pbuffer0.GetWidth(), pbuffer0.GetHeight());
}

int main(int argc, char* argv[])
{
...
        glut_add_interactor(&to_pbuffer);
        glut_add_interactor(&pbuffer_user);
        glut_add_interactor(&pbuffer_cb);
        glut_add_interactor(&to_window);
        glut_add_interactor(&window_user);
        glut_add_interactor(&window_cb);
        glut_add_interactor(&console);
        glut_add_interactor(&swapbuffers);
...
}
```

Если вы запустите полученную программу и будете рассматривать экран монитора под разными углами, то увидите, что экран монитора выглядит немного неестественно. Программа считает, что экран монитора видим на экране только благодаря освещению источниками света. В реальности все обстоит совершенно по-другому. Изображение на экране монитора видно только благодаря тому, что экран монитора излучает свет. Сторонние источники света только делают изображение бледнее, добавляя на экран различные блики, и т. д.

Следовательно, мы должны сделать экран монитора самосветящимся объектом, наложив на него самосветящийся материал (self-illuminated). Для этого необходимо изменить у материала экрана монитора в редакторе материалов свойство `self-illuminated`, присвоив ему значение в диапазоне [80..100%].

Но это еще не все. Дело в том, что библиотека ASE Reader не умеет работать с самосветящимися материалами. Поэтому нам необходимо "научить" библиотеку ASE Reader использовать такие материалы.

Значение параметра `self-illumination` хранится в поле `selfillum`. Но здесь перед нами встает следующая проблема — OpenGL не поддерживает модель освещения, использующую параметр `self-illumination`. В OpenGL параметр самосвечения материала задается цветом самосвечения материала (`GL_EMISSION`).

Следовательно, нам придется эмулировать параметр `self-illumination` через параметр `GL_EMISSION`. Ниже приведен фрагмент кода нового варианта библиотеки ASE Reader, вычисляющий новый диффузный цвет и цвет свечения объекта на основе диффузного цвета и коэффициента самосвечения:

```
typedef struct _modelData
{
...
    float diffuse[4];
...
    float emission[4];
} modelData;

bool CAseModel::LoadModel(char *filename, char *modelname)
{
...
// Вычисляем цвет самосвечения объекта
data[n].emission[0]=data[n].diffuse[0]*mat->selfillum;
data[n].emission[1]=data[n].diffuse[1]*mat->selfillum;
data[n].emission[2]=data[n].diffuse[2]*mat->selfillum;
                            data[n].emission[3]=1.0;
// Понижаем яркость диффузного цвета обратно пропорционально яркости
// самосвечения объекта
data[n].diffuse[0]=data[n].diffuse[0]*(1.0-mat->selfillum);
data[n].diffuse[1]=data[n].diffuse[1]*(1.0-mat->selfillum);
data[n].diffuse[2]=data[n].diffuse[2]*(1.0-mat->selfillum);
...
}
```

Остальные изменения в библиотеке ASE Reader довольно тривиальны, поэтому мы не будем их рассматривать. Исходный текст нового варианта библиотеки ASE Reader приведен в каталоге примера Ex06.

8.3. Использование расширения ARB_render_texture

До сих пор мы осуществляли доступ к изображению в буфере pbuffer при помощи простого копирования его содержимого в текстуру. Из-за того что копирование производится из одной области видеопамяти в другую, оно занимает очень мало времени. Но при использовании текстур большого размера это время может стать вполне ощутимым. Кроме того, расходуется драгоценная видеопамять, в которой наряду с буфером pbuffer приходится хранить и его копию. Если бы мы могли непосредственно использовать pbuffer в качестве текстуры.

К счастью, это вполне возможно. Относительно недавно, в июне 2001 года, комитет ARB разработал расширение ARB_render_texture, позволяющее непосредственно использовать содержимое буфера pbuffer в качестве текстуры.

Расширение ARB_render_texture не только предоставляет программисту набор новых функций, но и расширяет функциональность расширений WGL_ARB_pixel_format и WGL_ARB_pbuffer, добавляя в их команды поддержку новых атрибутов.

Чтобы pbuffer мог использоваться расширением ARB_render_texture в качестве RGB- или RGBA-текстуры, необходимо при задании формата пикселов командой wglChoosePixelFormatARB установить параметр WGL_BIND_TO_TEXTURE_RGB_ARB (для RGB-текстуры) или WGL_BIND_TO_TEXTURE_RGBA_ARB (для RGBA-текстуры) в значение GL_TRUE.

В функцию wglCreatePbufferARB, создающую pbuffer, также добавляется несколько новых параметров (табл. 8.3). Необходимость в этих параметрах связана с тем, что pbuffer является менее гибким объектом, чем текстура. Созданный однажды pbuffer не может изменить ни своего размера, ни формата пикселов, ни объема занимаемой видеопамяти. Поэтому перед созданием буфера pbuffer OpenGL обязана знать всю информацию о текстуре, в качестве которой он будет использоваться. Ведь если, к примеру, реализация OpenGL не выделит при создании буфера pbuffer дополнительной видеопамяти для хранения Mipmap-уровней, то pbuffer нельзя будет использовать в качестве текстуры с Mipmap-фильтраций.

Таблица 8.3. *Параметры функции* `wglCreatePbufferARB`

Параметр	Назначение	Возможные значения параметра	Описание
WGL_TEXTURE_FORMAT_ARB	Указывает формат пикселов текстуры	WGL_TEXTURE_RGB_ARB	pbuffer будет использоваться в качестве RGB-текстуры

Таблица 8.3 *(окончание)*

Параметр	Назначение	Возможные значения параметра	Описание
		WGL_TEXTURE_RGBA_ARB	pbuffer будет использоваться в качестве RGBA-текстуры
WGL_TEXTURE_TARGET_ARB	Указывает размерность текстуры	WGL_TEXTURE_1D_ARB	pbuffer будет использоваться в качестве одномерной текстуры
		WGL_TEXTURE_2D_ARB	pbuffer будет использоваться в качестве двумерной текстуры
WGL_MIPMAP_TEXTURE_ARB	Резервирует дополнительную видеопамять для хранения Mipmap-уровней	GL_FALSE	Текстура не будет использовать Mipmap-фильтрацию
		GL_TRUE	Текстура будет использовать Mipmap-фильтрацию

После этого мы можем использовать изображение из буфера pbuffer практически как обычное изображение текстуры с единственным отличием: для указания того, что текущий текстурный объект будет хранить изображение текстуры в буфере pbuffer, используется команда wglBindTexImageARB (аналог команды glTexImage):

```
BOOL wglBindTexImageARB (HPBUFFERARB hPbuffer, int iBuffer)
```

где:

- hPbuffer — дескриптор буфера pbuffer;
- iBuffer — идентификатор кадрового буфера pbuffer, который будет использоваться в качестве текстуры (WGL_FRONT_LEFT_ARB, WGL_FRONT_RIGHT_ARB, WGL_BACK_LEFT_ARB или WGL_BACK_RIGHT_ARB).

Когда буфер pbuffer используется в качестве текстуры, его содержимое нельзя изменять (это обстоятельство сильно упрощает жизнь разработчикам видеокарт). Для того чтобы "освободить" буфер pbuffer и сделать его "модифицируемым", используется функция wglReleaseTexImageARB:

```
BOOL wglReleaseTexImageARB (HPBUFFERARB hPbuffer, int iBuffer)
```

Параметры этой команды аналогичны параметрам функции wglBindTexImageARB.

Для того чтобы продемонстрировать практическое использование расширения ARB_render_texture, я переписал пример Ex02 (Ex07). В листинге 8.10 приведены наиболее интересные фрагменты кода этого примера.

Листинг 8.10

```
void Init()
{
// Проверяем поддержку необходимых расширений
       if (!glh_init_extensions("WGL_ARB_pbuffer WGL_ARB_pixel_format
WGL_ARB_render_texture GL_SGIS_generate_mipmap"))
       {
              MessageBox(NULL, glh_get_unsupported_extensions(),
"Unsupported  extensions", MB_OK | MB_ICONSTOP);
              exit(-1);
       }

// Получаем дескрипторы текущего окна
       WinID=glutGetWindow();
       HDC hdc=wglGetCurrentDC();

// Очищаем список атрибутов
       int niattribs=0;
       int nfattribs=0;
       for (int i=0; i<2*MAX_ATTRIBS; i++)
       {
              iattributes[i]=0;
              fattributes[i]=0;
       }

// Формат пикселов должен поддерживаться буфером pbuffer
       iattributes[2*niattribs]=WGL_DRAW_TO_PBUFFER_ARB;
       iattributes[2*niattribs+1]=true;
       niattribs++;

// Необходим 24-битный буфер глубины
       iattributes[2*niattribs]=WGL_DEPTH_BITS_ARB;
       iattributes[2*niattribs+1]=24;
       niattribs++;
```

```
// Необходим 24-битный цвет (8, 8, 8)
       iattributes[2*niattribs]=WGL_RED_BITS_ARB;
       iattributes[2*niattribs+1]=8;
       niattribs++;

       iattributes[2*niattribs]=WGL_GREEN_BITS_ARB;
       iattributes[2*niattribs+1]=8;
       niattribs++;

       iattributes[2*niattribs]=WGL_BLUE_BITS_ARB;
       iattributes[2*niattribs+1]=8;
       niattribs++;

// Формат пикселов должен быть совместим с расширением ARB_render_texture
       iattributes[2*niattribs]=WGL_BIND_TO_TEXTURE_RGB_ARB;
       iattributes[2*niattribs+1]=GL_TRUE;
       niattribs++;

// Создаем pbuffer
       if (!wglChoosePixelFormatARB(hdc, iattributes, fattributes,
MAX_PFORMATS, pformat, &nformats))
       {
              MessageBox(NULL, "PBuffer creating error: request pixel
format is unsupported",
                                 "Unsupported pixel format", MB_OK |
MB_ICONSTOP);
              exit(-1);
       }

// Если формат пикселов не поддерживается
    if ( nformats <= 0 )
    {
        MessageBox(NULL, "pbuffer creation error:  Couldn't find a
suitable pixel format.",
                     "Unsupported pixel format", MB_OK | MB_ICONSTOP);
        exit(-1);
    }

// Очищаем список атрибутов
       niattribs=0;
       nfattribs=0;
```

```
        for (int i=0; i<2*MAX_ATTRIBS; i++)
        {
                iattributes[i]=0;
                fattributes[i]=0;
        }

// Необходимо выделить видеопамять для хранения Mipmap-уровней
        iattributes[2*niattribs]=WGL_MIPMAP_TEXTURE_ARB;
        iattributes[2*niattribs+1]=GL_TRUE;
        niattribs++;

// pbuffer будет использоваться в качестве RGB-текстуры
        iattributes[2*niattribs]=WGL_TEXTURE_FORMAT_ARB;
        iattributes[2*niattribs+1]=WGL_TEXTURE_RGB_ARB;
        niattribs++;

// pbuffer будет использоваться в качестве двумерной текстуры
        iattributes[2*niattribs]=WGL_TEXTURE_TARGET_ARB;
        iattributes[2*niattribs+1]=WGL_TEXTURE_2D_ARB;
        niattribs++;

// Создаем pbuffer
        hbuffer1=wglCreatePbufferARB(hdc, pformat[0], pbuffer_width,
pbuffer_height, iattributes);
// Получаем контекст буфера pbuffer
        hpbufdc1=wglGetPbufferDCARB(hbuffer1);
// Получаем контекст OpenGL
        pbufglctx1=wglCreateContext(hpbufdc1);

// pbuffer должен разделять дисплейные списки с главным окном
        wglShareLists(wglGetCurrentContext(), pbufglctx1);

// Создаем текстурный объект
        tex.bind();
// Настраиваем параметры фильтрации
        tex.parameter(GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
        tex.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Включаем автоматическую генерацию Mipmap-уровней
        tex.parameter(GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
```

```
// Изображение текстуры хранится в pbuffer
       wglBindTexImageARB(hbuffer1, WGL_FRONT_LEFT_ARB);

// Инициализируем pbuffer
       InitPBuffer();
// Инициализируем главное окно программы
       glClearColor(0.1, 0.1, 0.7, 0);
       glMatrixMode(GL_PROJECTION);
       glLoadIdentity();
       glOrtho(0, 1, 0, 1, -1, 1);
       glMatrixMode(GL_MODELVIEW);
}

void Display()
{
// Проверяем потерю буфера pbuffer
       GLint PBufferLost;
       wglQueryPbufferARB(hbuffer1, WGL_PBUFFER_LOST_ARB, &PBufferLost);
       if (PBufferLost)
       {
// Восстанавливаем pbuffer в случае его потери
              wglDeleteContext(pbufglctx1);
              wglReleasePbufferDCARB(hbuffer1, hpbufdc1);
              wglDestroyPbufferARB(hbuffer1);

              HDC hdc=wglGetCurrentDC();
              hbuffer1=wglCreatePbufferARB(hdc, pformat[0],
pbuffer_width, pbuffer_height, iattributes);
              hpbufdc1=wglGetPbufferDCARB(hbuffer1);
              pbufglctx1=wglCreateContext(hpbufdc1);

              InitPBuffer();
       }

// Активизируем pbuffe
       wglMakeCurrent(hpbufdc1, pbufglctx1);

// Освобождаем pbuffer
       tex.bind();
       wglReleaseTexImageARB(hbuffer1, WGL_FRONT_LEFT_ARB);
```

```
// Рисуем изображение в буфере pbuffer
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glPushMatrix();
                glColor3f(0.5, 0.5, 0.5);
                glRotatef(GLfloat((GetTickCount()-Time))/10, 0, 1, 0);
                glutSolidTeapot(0.5);
        glPopMatrix();

// Привязываем pbuffer к текстуре
        wglBindTexImageARB(hbuffer1, WGL_FRONT_LEFT_ARB);

// Активизируем главное окно
        glutSetWindow(WinID);

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Включаем наложение текстуры и рисуем квадрат
        tex.bind();
        tex.enable();
        glBegin(GL_QUADS);
                glTexCoord2f(0, 0);
                glVertex2f(0, 0);

                glTexCoord2f(0, 1);
                glVertex2f(0, 0.5);

                glTexCoord2f(1, 1);
                glVertex2f(0.5, 0.5);

                glTexCoord2f(1, 0);
                glVertex2f(0.5, 0);
        glEnd();
        tex.disable();

        glutSwapBuffers();
}
```

К сожалению, класс `PBuffer` из NVIDIA OpenGL SDK, рассмотренный в *разд. 8.2.1*, не годится для работы с буферами `pbuffer` расширения ARB_render_texture, т. к. он ничего не знает о новых атрибутах команд `wglChoosePixelFormatARB` и `wglCreatePbufferARB`. А работа с буфером `pbuffer` только при помощи функций WGL — задача не из приятных.

Поэтому мы модифицируем класс `PBuffer` из NVIDIA OpenGL SDK, добавив в него поддержку расширения ARB_render_texture. Из-за того, что этот класс не содержит виртуальных функций, единственный способ модифицировать класс `PBuffer` — модифицировать исходный код. Правда, учитывая, что перед использованием этого класса необходимо скопировать в каталог программы файлы с исходными текстами этого класса (pbuffer.h и pbuffer.cpp), модификация исходного текста класса `PBuffer` не должна привести к проблемам совместимости.

Для начала мы должны объявить новые битовые флажки, которые будут передаваться конструктору класса `PBuffer`:

```
// Текстура формата RGB
const PBUFFER_TEXTURE_RGB=64;
// Текстура формата RGBA
const PBUFFER_TEXTURE_RGBA=128;
// Одномерная текстура
const PBUFFER_TEXTURE_1D=256;
// Двумерная текстура
const PBUFFER_TEXTURE_2D=512;
// Резервировать память для хранения Mipmap-уровней
const PBUFFER_TEXTURE_MIPMAP=1024;
```

Теперь надо добавить в метод `Initialize` код, анализирующий новые флажки и создающий список соответствующих атрибутов для функций `wglChoosePixelFormatARB` и `wglCreatePbufferARB` (листинг 8.11).

Листинг 8.11

```
void PBuffer::Initialize(bool sharecontexts, bool sharelists)
{
...
// Заполняем список атрибутов новыми атрибутами расширения
// ARB_render_texture
// Если текстура будет иметь формат RGB
              if (mode & PBUFFER_TEXTURE_RGB)
              {
```

```
        iattributes[2*niattribs]=WGL_BIND_TO_TEXTURE_RGB_ARB;
                        iattributes[2*niattribs+1]=GL_TRUE;
                        niattribs++;
                }
// Если текстура будет иметь формат RGBA
                if (mode & PBUFFER_TEXTURE_RGBA)
                {

        iattributes[2*niattribs]=WGL_BIND_TO_TEXTURE_RGBA_ARB;
                        iattributes[2*niattribs+1]=GL_TRUE;
                        niattribs++;
                }
         iattributes[2*niattribs  ] = WGL_SUPPORT_OPENGL_ARB;
         iattributes[2*niattribs+1] = true;
         niattribs++;


// Выбираем формат пикселов
         int pformat[MAX_PFORMATS];
         unsigned int nformats;
         if ( !wglChoosePixelFormatARB( hdc, iattributes, fattributes,
MAX_PFORMATS, pformat, &nformats ) )
             {
             fprintf( stderr, "pbuffer creation error:
wglChoosePixelFormatARB() failed.\n" );
             exit( -1 );
             }
            if ( nformats <= 0 )
             {
             fprintf( stderr, "pbuffer creation error:  Couldn't find a
suitable pixel format.\n" );
             exit( -1 );
             }
         format = pformat[0];
    }


// Очищаем список атрибутов
        int properties[MAX_ATTRIBS*2];
        int niattribs=0;
    for ( int i = 0; i < 2*MAX_ATTRIBS; i++ )
```

```
    {
            properties[i] = 0;
    }

// Заполняем список атрибутов

// Если текстура будет иметь RGB-формат
       if (mode & PBUFFER_TEXTURE_RGB)
       {
              properties[2*niattribs]=WGL_TEXTURE_FORMAT_ARB;
              properties[2*niattribs+1]=WGL_TEXTURE_RGB_ARB;
              niattribs++;
       }

// Если текстура будет иметь RGBA-формат
       if (mode & PBUFFER_TEXTURE_RGBA)
       {
              properties[2*niattribs]=WGL_TEXTURE_FORMAT_ARB;
              properties[2*niattribs+1]=WGL_TEXTURE_RGBA_ARB;
              niattribs++;
       }

// Если текстура будет одномерной
       if (mode & PBUFFER_TEXTURE_1D)
       {
              properties[2*niattribs]=WGL_TEXTURE_TARGET_ARB;
              properties[2*niattribs+1]=WGL_TEXTURE_1D_ARB;
              niattribs++;
       }

// Если текстура будет двумерной
       if (mode & PBUFFER_TEXTURE_2D)
       {
              properties[2*niattribs]=WGL_TEXTURE_TARGET_ARB;
              properties[2*niattribs+1]=WGL_TEXTURE_2D_ARB;
              niattribs++;
       }
```

```
// Если текстура будет использовать Mipmap-фильтрацию
        if (mode & PBUFFER_TEXTURE_MIPMAP)
        {
                properties[2*niattribs]=WGL_MIPMAP_TEXTURE_ARB;
                properties[2*niattribs+1]=GL_TRUE;
                niattribs++;
        }
// создаем буфер pbuffer
        buffer = wglCreatePbufferARB( hdc, format, width, height,
properties );
...
}
```

Эти изменения позволяют приложению создать буфер `pbuffer`, совместимый с расширением ARB_render_texture. Однако для комфортной работы с буфером `pbuffer` в качестве текстуры необходимо добавить в класс `PBuffer` методы, инкапсулирующие команды расширения ARB_render_texture:

```
class PBuffer
{
...
// Привязываем pbuffer к текстуре
        void BindTexImage(int iBuffer=WGL_FRONT_LEFT_ARB);
// Освобождаем pbuffer
        void ReleaseTexImage(int iBuffer=WGL_FRONT_LEFT_ARB);
}
...
void PBuffer::BindTexImage(int iBuffer)
{
        wglBindTexImageARB(buffer, iBuffer);
}

void PBuffer::ReleaseTexImage(int iBuffer)
{
        wglReleaseTexImageARB(buffer, iBuffer);
}
```

В принципе, на этом можно было бы и остановиться. Но раз уж мы начали модификацию класса `PBuffer`, то мы можем заодно исправить и некоторые

недостатки класса `PBuffer`. Например, мы можем заставить метод `HandleModeSwitch` возвращать значение `true` в случае потери буфера `pbuffer`:

```
bool PBuffer::HandleModeSwitch()
{
    int lost = 0;

    wglQueryPbufferARB( buffer, WGL_PBUFFER_LOST_ARB, &lost );

    if ( lost )
        {
        this->~PBuffer();
        Initialize( sharedctx, sharedlists );
        return true;
        }
       return false;
}
```

Для демонстрации практического использования модифицированного класса `PBuffer` я переписал пример Ex08 с использованием этого класса (листинг 8.12).

Листинг 8.12

```
// Задаем параметры буфера pbuffer
PBuffer pbuffer1(256, 256, GLUT_SINGLE | GLUT_DEPTH | PBUFFER_TEXTURE_RGB
| PBUFFER_TEXTURE_2D | PBUFFER_TEXTURE_MIPMAP);

void Init()
{
...
// Создаем pbuffer
       pbuffer1.Initialize(false, true);

       tex.bind();
       tex.parameter(GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
       tex.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
       tex.parameter(GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
// Привязываем pbuffer к текстурному объекту
       pbuffer1.BindTexImage();
...
```

```
}
...
void Display()
{
// Если pbuffer был потерян, пересоздаем его
       if (pbuffer1.HandleModeSwitch())
              InitPBuffer();

       pbuffer1.MakeCurrent();

// Освобождаем pbuffer
       pbuffer1.ReleaseTexImage();

// Рисуем изображение в буфере pbuffe
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       glPushMatrix();
              glColor3f(0.5, 0.5, 0.5);
              glRotatef(GLfloat((GetTickCount()-Time))/10, 0, 1, 0);
              glutSolidTeapot(0.5);
       glPopMatrix();

       glutSetWindow(WinID);

       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

       tex.bind();
// Привязываем буфер pbuffer к текстурному объекту
       pbuffer1.BindTexImage();
       tex.enable();
       glBegin(GL_QUADS);
              glTexCoord2f(0, 0);
              glVertex2f(0, 0);

              glTexCoord2f(0, 1);
              glVertex2f(0, 0.5);

              glTexCoord2f(1, 1);
              glVertex2f(0.5, 0.5);
```

```
            glTexCoord2f(1, 0);
            glVertex2f(0.5, 0);
        glEnd();
        tex.disable();

        glutSwapBuffers();
}
```

Как видно, использование модифицированного класса `PBuffer` сделало программу значительно более удобной.

8.4. Пример создания виртуального мира

К настоящему моменту мы узнали достаточно много всякой информации об OpenGL и NVIDIA OpenGL SDK. Но для демонстрации практического использования полученных знаний в основном использовались небольшие компактные примеры, которые хотя и отражают суть изучаемой технологии, слабо связаны с реальными проектами, при создании которых программисту приходится одновременно использовать большой спектр технологий, решая множество прикладных задач.

Поэтому для закрепления полученных знаний мы напишем полноценную программу, моделирующую небольшой фрагмент виртуального мира, в котором пользователь может свободно перемещаться, подчиняясь основным физическим законам реального мира. То есть наша программа будет фактически обычной игрой от первого лица (First Person Shooter), отличаясь от последней лишь отсутствием врагов и конечной цели.

В качестве сцены для нашего виртуального мира мы воспользуемся моделью долины, которую мы создали в *главе 4* для примеров Ex11 и Ex12 (см. рис. 4.23). Сам процесс загрузки сцены долины из файла с последующим ее рисованием на экране не доставит нам больших хлопот: библиотека ASE Reader выполнит эти операции за нас. Но мы должны не просто вывести долину на экран, а ходить по ней. Иными словами, наблюдатель должен свободно перемещаться по поверхности долины, не проваливаясь под ее поверхность. Следовательно, нам необходимо научиться определять столкновение наблюдателя с объектами цены, чтобы не допустить его попадание вовнутрь сплошных (solid) объектов. Кроме того, наблюдатель не должен свободно парить в воздухе — в реальной жизни сила тяжести заставляет его падать вниз то тех пор, пока он не столкнется с землей или каким-нибудь другим объектом. При этом голова наблюдателя окажется над поверхностью земли примерно на высоте его роста.

Но задача определения столкновения двух полигональных объектов очень сложная и ресурсоемкая. К счастью, существует другой, более простой способ.

Если вы внимательно изучите сцену долины, то обнаружите, что она на большем своем протяжении является однозначной поверхностью, которая может быть описана уравнением $z = f(x, y)$ (оси x и y лежат в плоскости поверхности земли, а ось z направлена вверх). Следовательно, положение головы наблюдателя в точке с координатами x, y задается уравнением $z = f(x, y) + h$, где h — его рост.

Функция $z = f(x, y)$ задает высоту местности в точке с координатами x, y. В случае многозначной поверхности (а наша долина в некоторых местах является двузначной или даже трехзначной поверхностью) она задает *наибольшую* высоту местности в точке с координатами x, y. Но и аналитическое определение высоты произвольной полигональной модели в конкретной точке — это сложная задача. Правда, нам не обязательно находить точное аналитическое ее решение — нас вполне устроит приближенное определение высоты поверхности в данной точке средствами OpenGL.

Для этого необходимо выполнить следующие действия:

1. Установить размер видового окна равным 1×1. То есть на экране будет отображаться всего один пиксел.
2. Установить ортогональную проекцию командой вида `glOrtho( -user, user, -user, user, 0, max_z-min_z )`, где `user` — приближенные размеры наблюдателя, а (`max_z`—`min_z`) — максимальная высота поверхности земли (разница между самой высокой и самой низкой точками поверхности).
3. Приподнять камеру над землей на высоту, равную максимальной высоте поверхности земли, и заставить ее смотреть вниз.
4. Совместить камеру вдоль осей x и y с текущей позицией наблюдателя.
5. Нарисовать сцену.

В результате выполнения этой последовательности действий на экран будет выведен один пиксел, z-компонент которого (`zbuf`), хранящийся в `z-buffer`, будет линейно связан с расстоянием от камеры до наивысшей точки фрагмента поверхности, находящейся под камерой. Если `zbuf = 0`, то высота фрагмента поверхности под камерой совпадает с наивысшей высотой поверхности (`max_z`), а при `zbuf = 1` — с наименьшей высотой поверхности (`min_z`). Следовательно, высота участка поверхности (`fz`) связана с `zbuf` следующим выражением:

$$fz = \mathtt{max\_z} - (\mathtt{max\_z} - \mathtt{min\_z}) \cdot zbuf .$$

Еще раз обратите внимание на то что, мы рисуем фрагмент поверхности, совпадающий по размеру с наблюдателем. Но т. к. размер видового окна равен единице, в нем будет только один пиксель, принадлежащий точке на

поверхности с наибольшей высотой. В результате, наблюдатель никогда не провалится под землю даже на очень неровной поверхности.

Остается лишь очистить экран, установить положение головы наблюдателя на высоту `z=fz+h` и повторно нарисовать модель долины. Так как два прохода рисования сцены (промежуточное рисование модели для определения высоты и итоговое рисование модели долины) используют разное разрешение экрана, промежуточное рисование сцены оптимальнее всего осуществлять в `p-buffer`.

Как поместить наблюдателя на поверхность земли, мы вроде бы разобрались. Но он должен не просто стоять на земле, а *перемещаться* по ней. Для перемещения по поверхности земли мы будем использовать такое же управление, как в большинстве Quake-подобных игр: клавиша <Up> перемещает наблюдателя вперед, <Down> — назад, <Left> — налево, <Right> — направо, <Shift> — переключение с ходьбы на бег, и наоборот, <Tab> — включение режима "автоматический бег" (`auto run`). Поворот головы наблюдателя выполняется с использованием мыши.

Положение наблюдателя будет задаваться тремя параметрами: позиция наблюдателя (`pos.x, pos.y, pos.z`), поворот наблюдателя относительно оси *x* (`rotx`), т. е. поворот типа вверх-вниз, и поворот наблюдателя относительно оси *z* (`rotz`). Следовательно, при нажатии клавиши <Up> наблюдатель будет перемещаться из позиции `pos` в новую позицию `pos1` со следующими координатами (`speed` — текущая скорость объекта):

$$pos1.x = pos.x + speed * \sin(rotz)$$

$$pos1.y = pos.y + speed * \cos(rotz)$$

А при нажатии клавиши <Left>:

$$pos1.x = pos.x - speed * \sin(rotz)$$

$$pos1.y = pos.y + speed * \cos(rotz)$$

Формулы перемещения для клавиш <Down> и <Right> получаются из формул для клавиш <Left> и <Right> путем простой замены знака "+" на "-", и наоборот.

С изменением положения углов поворота при перемещении мыши ситуация намного интереснее. Дело в том, что до сих пор мы считали изменение угла поворота пропорциональным расстоянию, которое проходит указатель мыши вдоль одной из осей экранной системы координат. Этот прием нормально работает до тех пор, пока указатель мыши не остановится, упершись в одну из границ экрана. Реакция программы на мышь также приостановится. Если поворот осуществляется только при нажатой кнопке мыши и указатель мыши виден на экране, то пользователь легко распознает проблему, после чего, отпустив кнопку мыши, передвинет указатель поближе к центру экрана и продолжит вращение. К сожалению, в нашем случае все обстоит по-другому: в программах виртуальной реальности указатель мыши обычно

всегда скрыт, а вращение наблюдателя осуществляется независимо от состояния кнопок мыши.

Следовательно, нам необходимо научиться решать проблему блокировки мыши краями экрана. К счастью, решение лежит на поверхности — необходимо отслеживать столкновение указателя мыши с краями экрана, а при обнаружении столкновения переносить указатель мыши в центр экрана:

```
void MousePassiveMotion(int x, int y)
{
...
    int width=glutGet(GLUT_WINDOW_WIDTH);
    int height=glutGet(GLUT_WINDOW_HEIGHT);

    if ((x==0) || (y==0) || (x==width-1) || (y==height-1))
    {
            x=width/2;
            y=height/2;
            glutWarpPointer(x, y);
    }
...
}
```

После такой модификации программа будет работать лучше. Но, тем не менее, на краях экрана возможны некоторые эксцессы. Например, если пользователь рывком передвигает в сторону края экрана указатель мыши, находящийся в окрестностях края экрана, то указатель упрется в край экрана, не пройдя весь путь. Это будет воспринято пользователем, как кратковременный скачек чувствительности мыши. Для борьбы с этим явлением необходимо ввести на краях экрана некоторые "мертвые зоны", попадание в которые будет вызывать моментальное перемещение указателя в центр экрана. В большинстве случаев ширины этих зон, составляющей 20% от размера экрана, будет вполне достаточно:

```
    if ((x<width/5) || (y<height/5) || (x>width*4/5) ||
(y>height*4/5))
    {
            x=width/2;
            y=height/2;
            glutWarpPointer(x, y);
    }
```

В принципе, этого вполне достаточно для написания первого прототипа программы моделирования виртуального мира. В листинге 8.13 приведен

исходный текст наиболее важных фрагментов программы с подробными комментариями.

Листинг 8.13

```
// Дисплейный список модели долины
display_list valley;
// Модель долины
CAseModel valley_model;

// Высота наблюдателя
const float h=0.15;
// Требуемый размер долины, до которого она масштабируется
const float model_size=15.0;
// Чувствительность мыши
const float sens=0.05;
// Скорость бега
const float run_speed=0.015;
// Скорость ходьбы
const float walk_speed=0.005;
// Размер наблюдателя
const float user_size=0.015;
// Позиция наблюдателя
vec3f pos(0, 0, -2);
// Углы поворота наблюдателя
float rotx=90;
float rotz=0;
// Предыдущая позиция мыши
float last_mouse_pos_x;
float last_mouse_pos_y;
// Прямоугольная оболочка долины
vec3f min_extent, max_extent;
// Коэффициент масштабирования долины
float model_scale;
// pbuffer для промежуточного рисования сцены
PBuffer pbuffer0(1, 1, GLUT_RGB | GLUT_SINGLE | GLUT_DEPTH);
// Идентификатор главного окна программы
int WinID;
// Режим AutoRun
```

```
bool bAutoRun=false;
bool Init()                          // Инициализация OpenGL
{
// Инициализируем необходимые расширения
        if (!glh_init_extensions("WGL_ARB_pbuffer WGL_ARB_pixel_format"))
        {
                console.add(string("Unsupported extension:
")+glh_get_unsupported_extensions());
                console.exit(-1);
                return false;
        }
// Получаем идентификатор текущего окна
        WinID=glutGetWindow();
// Настраиваем параметры OpenGL
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
// Скрываем курсор
        glutSetCursor(GLUT_CURSOR_NONE);

        vec4 diffuse_color(0.7, 0.7, 0.7, 1);
        vec4 ambinet_color=vec4_one-diffuse_color+vec4(0, 0, 0, 1);

        glLightfv(GL_LIGHT0, GL_DIFFUSE, &diffuse_color[0]);
        glLightfv(GL_LIGHT0, GL_AMBIENT, &ambinet_color[0]);

        glClearColor(0.5, 0.5, 0.75, 1);

// Загружаем модель долины из файла
        if (!valley_model.LoadModel("..\\models.zip", "valley.ase"))
        {
                console.add("Can't load model
..\\models.zip::valley.ase");
                return false;
        }

// Создаем дисплейный список
        valley.new_list(GL_COMPILE);
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        glEnable(GL_NORMALIZE);
```

```
// Получаем координаты прямоугольной оболочки объекта
       valley_model.GetExtents(min_extent, max_extent);
// Определяем коэффициент масштабирования
       model_scale=max(fabs(min_extent[0]), fabs(min_extent[1]));
       model_scale=max(model_scale, fabs(min_extent[2]));
       model_scale=max(model_scale, fabs(max_extent[0]));
       model_scale=max(model_scale, fabs(max_extent[1]));
       model_scale=max(model_scale, fabs(max_extent[2]));
       model_scale=model_size/model_scale;
// Масштабируем модель
       glScalef(model_scale, model_scale, model_scale);
// Рисуем модель
       valley_model.DrawModel(true);
       glPopAttrib();
       valley.end_list();

// Выводим информацию о состоянии модели
       console.add("");
       console.add(valley_model.get_info());

// Выгружаем ненужную информацию о модели
       valley_model.Clear(false);
// Создаем pbuffer с разделяемыми списками
       pbuffer0.Initialize(false, true);

// Делаем его текущим
       pbuffer0.MakeCurrent();
       glEnable(GL_DEPTH_TEST);

// Настраиваем матрицу проекции
       glMatrixMode(GL_PROJECTION);
       glLoadIdentity();
// Создаем ортогональную проекцию, размер которой пропорционален
// скорости бега наблюдателя. Такой размер проекции в будущем упростит
// определение столкновений с объектами
       glOrtho(-user_size/2.0*model_scale, user_size/2.0*model_scale, -
user_size/2.0*model_scale, user_size/2.0*model_scale, 0, (max_extent[2]-
min_extent[2])*model_scale);
       glMatrixMode(GL_MODELVIEW);
```

```
        glViewport(0, 0, pbuffer0.GetWidth(), pbuffer0.GetHeight());
        glutSetWindow(WinID);
}

void Display()                // Обновление содержимого экрана
{
// Активизируем pbuffer
        pbuffer0.MakeCurrent();
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glLoadIdentity();
// Переносим камеру в позицию наблюдателя и приподнимаемся над
// поверхностью (так как в OpenGL ось z направлена вниз, координаты
// идут со знаком "-")
        glTranslatef(-pos[0], -pos[1], -max_extent[2]*model_scale);
// Рисуем долину
        valley.call_list();

// Глубина
GLfloat depth;
// Получаем значение глубины
        glReadPixels(0, 0, 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &depth);

// Вычисляем высоту в данной точке на основе глубины
        pos[2]=(max_extent[2]-(max_extent[2]-
min_extent[2])*depth)*model_scale+h;

// Переключаемся в главное окно
        glutSetWindow(WinID);

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        glLoadIdentity();
// Поворот вверх-вниз
        glRotatef(rotx, -1, 0, 0);
// Поворот влево-вправо
        glRotatef(rotz, 0, 0, 1);
// Перемещаем камеру в позицию наблюдателя
        glTranslatef(-pos[0], -pos[1], -pos[2]);
```

```
// Рисуем сцену
        valley.call_list();
}

// Корректируем углы поворота наблюдателя при движении мыши
void MousePassiveMotion(int x, int y)
{
// Вычисляем расстояние, пройденное мышью вдоль осей x и y
        int dx=x-last_mouse_pos_x;
        int dy=y-last_mouse_pos_y;
// Изменяем углы поворота
        rotz+=float(dx)*sens;
        rotx-=float(dy)*sens;

// Если ограничиваем угол поворота "вверх-вниз"
        if (rotx<0.0)
                rotx=0.0;

        if (rotx>180.0)
                rotx=180.0;

        int width=glutGet(GLUT_WINDOW_WIDTH);
        int height=glutGet(GLUT_WINDOW_HEIGHT);
// Если указатель мыши попал в мертвую зону, переносим его в центр экрана
        if ((x<width/5) || (y<height/5) || (x>width*4/5) ||
(y>height*4/5))
        {
                x=width/2;
                y=height/2;
                glutWarpPointer(x, y);
        }

// Сохраняем позицию мыши
        last_mouse_pos_x=x;
        last_mouse_pos_y=y;

        glutPostRedisplay();
}
```

```
void Keyboard(unsigned char key, int x, int y)
{
	if (console.visible)
		return;
	switch(key)
	{
// Если нажата клавиша <Tab>
		case VK_TAB:
			{
// Изменяем режим AutoRun
				bAutoRun=!bAutoRun;
				break;
			}
	}
}

void KeyboardSpecial(int key, int x, int y)
{
// Если на экране консоль — ничего не делаем
	if (console.visible)
		return;

// Определяем текущую скорость исходя из состояния клавиш-модификаторов
	float speed;
	int modifiers=glutGetModifiers();
	if (modifiers && GLUT_ACTIVE_SHIFT)
		if (bAutoRun)
			speed=walk_speed;
		else
			speed=run_speed;
	else
		if (bAutoRun)
			speed=run_speed;
		else
			speed=walk_speed;

	switch (key)
	{
```

```
// Если нажата клавиша <Up> — идем вперед
       case GLUT_KEY_UP:
              {
                     pos[0]+=speed*sinf(to_radians(rotz));
                     pos[1]+=speed*cosf(to_radians(rotz));
                     break;
              }
// Если нажата клавиша <DOWN> — идем назад
       case GLUT_KEY_DOWN:
              {
                     pos[0]-=speed*sinf(to_radians(rotz));
                     pos[1]-=speed*cosf(to_radians(rotz));
                     break;
              }
       case GLUT_KEY_LEFT:
              {
                     pos[0]-=speed*cosf(to_radians(rotz));
                     pos[1]+=speed*sinf(to_radians(rotz));
                     break;
              }
       case GLUT_KEY_RIGHT:
              {
                     pos[0]+=speed*cosf(to_radians(rotz));
                     pos[1]-=speed*sinf(to_radians(rotz));
                     break;
              }
       }

       glutPostRedisplay();
}
```

Теперь нам стоит подумать о моделировании реалистичного перемещения наблюдателя по поверхности земли. Наблюдатель должен подчиняться основным физическим законам. Например, скорость наблюдателя должна зависеть от наклона поверхности (в гору всегда идти труднее, чем с горы). Кроме того, наш наблюдатель будет уметь подпрыгивать вверх при нажатии клавиши <Ctrl>.

Для этого мы должны построить физическую модель движения наблюдателя по пересеченной местности, а затем реализовать ее в своей програм-

ме. Для начала нам необходимо определиться с силами, действующими на наблюдателя.

Первая сила — это движущая сила наблюдателя, заставляющая его идти вперед по поверхности земли ($\overrightarrow{F1}$). Вектор этой силы направлен по касательной к поверхности земли в направлении, зависящем от нажатых клавиш на клавиатуре. А его модуль зависит от физической подготовки наблюдателя (т. е. мощности наблюдателя) и его текущей скорости. При этом в режиме бега наблюдатель развивает бóльшую мощность, чем во время ходьбы. Из курса школьной физики известно, что сила (F), скорость (U) и мощность (N) связаны между собой следующим соотношением:

$$N = F1 \cdot U \cdot \cos\alpha,$$

где:

α — угол между векторами $\overrightarrow{F1}$ и $\vec{U}$.

Из этого соотношения можно легко определить зависимость модуля вектора силы от скорости и мощности:

$$F1 = \frac{N}{U \cdot \cos(\alpha)}, \tag{8.1}$$

То есть чем больше скорость, тем слабее сила. Эта формула довольно неплохо работает на средних и высоких скоростях. А вот при маленьких скоростях начинаются проблемы: знаменатель дроби стремится к нулю, в результате чего модуль силы устремляется в бесконечность. Что никак не соответствует действительности. Человек способен развивать вполне определенную силу тяги из-за ограниченной прочности своего организма — попытка развить сверхбольшую силу тяги приведет к травмам: деформации скелета, разрыву мышц и т. д. Поэтому мы ограничим максимальную силу тяги, развиваемую наблюдателем, путем добавления дополнительного слагаемого ($U_{\min}$) в знаменатель дроби формулы (8.1).

$$F1 = \frac{N}{U \cdot \cos(\alpha) + U_{\min}}, \tag{8.2}$$

где:

$U_{\min}$ — константа, ограничивающая максимальное значение силы тяги.

Кроме того, у этой формулы есть еще одно существенное ограничение — она "работает" только когда угол между векторами $\overrightarrow{F1}$ и $\vec{U}$ лежит в диапазоне [−90°..90°] (т. е. когда $\cos(\alpha) \geq 0$). В нашем же, случае этот угол может принимать любое значение из диапазона [−180°..180°]. К счастью, формулу

(8.2) можно легко приспособить и для таких углов при помощи небольшой модификации:

$$F1 = \frac{N}{U \cdot \cos a + U_{\min}}, \tag{8.3}$$

где:

cos*a* = cos(α), если cos(α) > 0. В противном случае cos*a* = 0.

Вторая сила — это сила тяжести ($\overrightarrow{F2}$), вектор которой направлен к центру земли. Так как наша долина имеет небольшой размер, мы можем считать, что земля в окрестностях долины является плоской, а сила тяжести направлена вниз вдоль оси *z*. А ее модуль всегда равен:

$$F2 = m \cdot g.$$

В нашем случае вектор силы тяжести можно разложить на два перпендикулярных вектора, один из которых ($\overrightarrow{F2Normal}$) перпендикулярен к поверхности (т. е. противоположно направлен по отношению к нормали поверхности), а второй ($\overrightarrow{F2Plane}$) направлен вдоль касательной к поверхности (т. е. этот вектор перпендикулярен к нормали). Вектор $\overrightarrow{F2Plane}$ оказывает влияние на скорость движения наблюдателя по наклонным поверхностям, а $\overrightarrow{F2Normal}$ — влияет на величину силы трения.

Вектор силы трения ($\overrightarrow{F3}$) направлен вдоль касательной к поверхности, в сторону, противоположную вектору скорости. Модуль этой силы зависит от силы реакции опоры, которая в нашем случае равна модулю вектора $\overrightarrow{F2Normal}$:

$$F3 = \mu \cdot F2Normal.$$

И, наконец, последняя сила — это сила удержания на траектории ($\overrightarrow{F4}$). Эта сила старается удержать наблюдателя на траектории, вдоль которой он движется. Например, если наблюдатель, двигавшийся с большой скоростью вдоль оси *x*, повернется на 90° и начнет двигаться вдоль оси *y*, сила удержания на траектории будет стремиться уменьшить его скорость движения вдоль оси *x* до 0. Другой пример: если наблюдатель остановится на склоне, то сила удержания на траектории будет препятствовать его движению вниз по склону холма. Для упрощения расчетов мы будем считать, что направление вектора $\overrightarrow{F4}$ совпадает с направлением разницы нормализованных векторов движущей силы наблюдателя и скорости: $norm(\overrightarrow{F1}) - norm(\overrightarrow{U})$, где norm — операция нормализации. А модуль вектора $\overrightarrow{F4}$ всегда равен константе, определяемой опытным путем.

Наш наблюдатель умеет не только ходить по поверхности земли, но и прыгать (клавиша <Ctrl>). Прыжок осуществляется путем прибавления к вектору скорости наблюдателя вектора, направленного вверх вдоль оси *z*. При этом сам наблюдатель переводится в режим полета, в котором на него не действуют никакие силы, кроме силы тяжести, заставляющей наблюдателя снова приземлиться на землю. Приземление на землю автоматически выводит наблюдателя из режима полета. Клавиша <Ctrl> — не единственный способ перевода наблюдателя в режим полета. Например, наблюдатель может оказаться в воздухе, свалившись с высокого обрыва, и т. д.

Если наблюдатель наткнется на невысокое отвесное препятствие (например, ступеньку лестницы), то программа должна автоматически приподнять его на него. В противном случае наблюдатель будет вынужден перемещаться прыжками даже по ступенькам лестницы. Если же препятствие окажется очень высоким (например, стена дома), то программа должна уменьшить скорость наблюдателя до нуля, чтобы не дать ему пройти сквозь дом (или взлететь на крышу).

С вопросом моделирования физически корректного перемещения наблюдателя по поверхности земли мы, похоже, разобрались. Но для написания полноценной программы мы должны решить еще две небольшие задачи.

Часть физических формул, приведенных выше, используют нормаль к поверхности. Следовательно, нам надо научиться определять нормаль в произвольной точке поверхности. Вообще определение нормали в произвольной точке полигональной поверхности — довольно непростая задача. Но т. к. нас вполне устроит и приближенное решение, задачу можно значительно упростить. Если мы увеличим размер буфера `pbuffer`, используемого в примере Ex09, с 1×1 до 2×2, то каждому пикселу буфера `pbuffer` будет соответствовать максимальная высота в окрестностях данного пиксела. На основе этой информации можно приближенно определить значения частных производных уравнения поверхности в данной точке по следующим формулам:

$$dx = \frac{\Delta hx}{sx};$$

$$dy = \frac{\Delta hy}{sy},$$

где:

- Δhx и Δhy — средний перепад высот вдоль осей x и y;
- sx и sy — размер буфера `pbuffer`.

А определение вектора нормали к поверхности на основе частных производных не составит труда — он равен `norm(dx, dy, 1)`.

Вторая проблема связана с обработкой событий от клавиатуры. Как известно, библиотека GLUT позволяет обрабатывать только событие нажатия одной клавиши клавиатуры. Следовательно, если пользователь одновременно нажмет несколько клавиш (например, подпрыгнет, двигаясь вперед), то библиотека GLUT не сможет корректно среагировать на такое событие. Кроме того, библиотека GLUT не содержит средств для непосредственного определения момента нажатия клавиш. Это не позволяет точно определить момент, когда клавиша была отпущена пользователем. Поэтому нам необходимо научиться работать с клавиатурой напрямую, в обход GLUT.

Первое, что приходит в голову, — обрабатывать события от клавиатуры при помощи средств Win32. К сожалению, это невозможно — в программу, использующую библиотеку GLUT, невозможно встроить цикл обработки сообщений — для этого пришлось бы внести изменения в исходный код библиотеки GLUT. К счастью, в Win32 имеются функции, позволяющие непосредственно определять состояние клавиш клавиатуры в текущий момент времени, без использования механизма обработки событий. Одна из таких функций — `GetKeyboardState`, копирующая состояние всех 256 виртуальных клавиш в указанный буфер:

```
BOOL GetKeyboardState(BYTE pbKeyState)
```

В качестве индексов в массиве используются коды виртуальных клавиш. Если клавиша нажата, то старший бит байта указанной клавиши будет равен 1, в противном случае — 0. Если функция `GetKeyboardState` будет вызываться с достаточной частотой, то программа сможет отслеживать состояние клавиш клавиатуры.

Теперь мы знаем информацию, необходимую для добавления в пример Ex09 моделирования физически корректного движения наблюдателя (Ex10). Наиболее важные фрагменты исходного кода полученной программы с подробными комментариями приведены в листинге 8.14.

Листинг 8 14

```
// Константы физической модели мира
// Рост наблюдателя
const float h=0.15;
// Размер долины
const float model_size=15.0;
// Чувствительность мыши
const float sens=0.05;
// Размер наблюдателя
const float user_size=0.015;
```

```
// Мощность наблюдателя во время ходьбы
const float walk_power=40;
// Мощность наблюдателя во время бега
const float run_power=80;
// Масса наблюдателя
const float mass=80;
// Ускорение свободного падения
const float g=3.0;
// Коэффициент падения мощности при "стрейфе" (боковом беге)
const float strafe_coeff=0.5;
// Модуль силы удержания на траектории
const float coerce=120.0;
// Параметр, используемый при расчете силы тяги наблюдателя, не дающий ей
// уйти в бесконечность при маленьких скоростях
const float min_u=0.4;
// Коэффициент трения между наблюдателем и поверхностью земли
const float nju=0.18;
// Максимальная скорость наблюдателя (человек может бежать с
// ограниченной скоростью)
const float max_speed=1.0;
// Прирост скорости во время прыжка вверх
const float jump_speed=1.0;
// Максимальная высота, на которую автоматически может быть приподнят
// наблюдатель
const float upborne=0.1;
// Текущая скорость наблюдателя
vec3f speed(0, 0, 0);
// Текущая позиция наблюдателя
vec3f pos(0, 0, 0);
// Текущая нормаль к плоскости
vec3f normal(0, 0, 1);
// Текущая высота поверхности
float z;
// Углы поворота наблюдателя
float rotx=90;
float rotz=0;
// Режим автоматического бега
bool bAutoRun=true;
```

```
// Режим полета
bool bFly=false;
// Прошлое состояние клавиши <Ctrl>
bool bCtrl=false;
...
// Координаты крайних вершин модели местности
vec3f min_extent, max_extent;
// Коэффициент масштабирования модели
float model_scale;
// Pbuffer, использующийся для определения высоты местности под
// наблюдателем и ее нормали
PBuffer pbuffer0(2, 2, GLUT_RGB | GLUT_SINGLE | GLUT_DEPTH);
...
// Текущее время
int current_tick=GetTickCount();
...
// Инициализация
bool Init()
{
...
// Задаем начальное положение наблюдателя на наивысшей высоте
// поверхности
       z=max_extent[2]*model_scale+h;
       pos[2]=z;

// Запоминаем время запуска программы
       current_tick=GetTickCount();
}
...
// Обновление содержимого экрана
void Display()
{
// Определяем, сколько прошло времени с момента рисования предыдущего
// кадра
       int delta_tick=GetTickCount()-current_tick;
// Запоминаем текущее время
       current_tick=GetTickCount();
// Переводим время в секунды (для удобства вычислений)
       float delta_sec=float(delta_tick)/1000.0;
```

```
// Сохраняем позицию наблюдателя
	vec3f last_pos=pos;
// Сохраняем высоту местности под наблюдателем
	float last_z=z;
// Сохраняем нормаль под наблюдателем
	vec3f last_normal=normal;
// Текущая мощность наблюдателя
	float power;
// Получаем состояние клавиш клавиатуры
	BYTE KeyboadState[256];
	GetKeyboardState(&KeyboadState[0]);
// Если нажата клавиша <Shift>, инвертируем мощность
	if (KeyboadState[VK_SHIFT] & 128)
	{
		if (bAutoRun)
			power=walk_power;
		else
			power=run_power;
	}
	else
	{
		if (bAutoRun)
			power=run_power;
		else
			power=walk_power;
	}
// Направление вектора силы тяги наблюдателя
	vec3f F1Basis(0, 0, 0);

// Если не активна консоль
	if (!console.visible)
{
// Определяем направление вектора силы тяги в плоскости x, y
// Если нажата клавиша <UP>, вектор силы направлен вперед
		if (KeyboadState[VK_UP] & 128)
		{
			F1Basis[0]+=sinf(to_radians(rotz));
			F1Basis[1]+=cosf(to_radians(rotz));
		}
```

```
// Если нажата клавиша <Down>, вектор силы направлен назад
              if (KeyboadState[VK_DOWN] & 128)
              {
                     F1Basis[0]+=-sinf(to_radians(rotz));
                     F1Basis[1]+=-cosf(to_radians(rotz));
              }
// Если нажата клавиша <Left>, вектор силы направлен влево и ослаблен
              if (KeyboadState[VK_LEFT] & 128)
              {
                     F1Basis[0]+=-cosf(to_radians(rotz))*strafe_coeff;
                     F1Basis[1]+=sinf(to_radians(rotz))*strafe_coeff;
              }
// Если нажата клавиша <Right>, вектор силы направлен вправо и ослаблен
              if (KeyboadState[VK_RIGHT] & 128)
              {
                     F1Basis[0]+=cosf(to_radians(rotz))*strafe_coeff;
                     F1Basis[1]+=-sinf(to_radians(rotz))*strafe_coeff;
              };

// Если нажата клавиша <Ctrl>
              if (KeyboadState[VK_CONTROL] & 128)
              {
// Если клавиша <Ctrl> до этого не была нажата (для следующего
// подпрыгивания необходимо нажать <Ctrl> еще раз)
                     if (!bCtrl)
                     {
// Если наблюдатель не находится в режиме полета
                            if (!bFly)
                            {
// Переводим наблюдателя в режим полета
                                   bFly=true;
// Увеличиваем его скорость
                                   speed[2]+=jump_speed;
                            }
                     }
                     bCtrl=true;
              }
              else
```

```
                    bCtrl=false;
}
// Если наблюдатель не находится в режиме полета (идет по земле)
       if (!bFly)
       {
// Если вектор силы тяги не равен 0
              if (!equivalent(F1Basis.square_norm(), 0))
              {
// Определяем направление вектора силы тяги с учетом того, что он
// направлен перпендикулярно нормали
                    F1Basis[2]=-
(normal[0]*F1Basis[0]+normal[1]*F1Basis[1])/normal[2];
                    F1Basis.normalize();
              }

// Определяем направление вектора скорости
              vec3f speed_basis;
// Если модуль скорости очень мал, то вектор направления скорости нулевой
              if (!equivalent(speed.square_norm(), 0))
              {
                    speed_basis=speed;
                    speed_basis.normalize();
              }
              else
              {
                    speed_basis=vec3f(0, 0, 0);
              }
// Определяем косинус угла между векторами силы тяги и скорости
              float cosa=speed_basis.dot(F1Basis);
              if (cosa<0)
                    cosa=0;
// Определяем модуль вектора силы тяги
              float F1Abs=power/(speed.length()*cosa+min_u);
// Находим вектор силы тяги
              vec3f F1=F1Basis*F1Abs;
// Находим вектор силы тяжести
              vec3f F2=vec3f(0, 0, -1)*mass*g;
// Раскладываем вектор силы тяжести на два взаимно перпендикулярных
// вектора, один из которых обратно параллелен вектору нормали
```

```
// (F2Normal), а второй — перпендикулярен (F2Plane).
// F2PlaneBasis — направление вектора F2Plane
// Вектор F2Normal влияет на силу трения, а F2Plane — заставляет
// наблюдателя скатываться с наклонных поверхностей
              vec3f F2Plane;
              vec3f F2PlaneBasis=normal;
              F2PlaneBasis[2]=-
(normal[0]*normal[0]+normal[1]*normal[1])/normal[2];
              if (!equivalent(F2PlaneBasis[2], 0.0))
              {
                     F2PlaneBasis.normalize();
                     float
k=F2PlaneBasis.dot(F2)/F2PlaneBasis.dot(F2PlaneBasis);
                     F2Plane=F2PlaneBasis*k;
              }
              else
              {
                     F2PlaneBasis=vec3f(0, 0, 0);
                     F2Plane=vec3f(0, 0, 0);
              }
// Находим сумму векторов силы тяги и проекции силы тяжести на
// поверхность
              vec3f F=F1+F2Plane;
// Определяем направление результирующего вектора
              vec3f FBasis;
              if (!equivalent(F.square_norm(), 0))
              {
                     FBasis=F;
                     FBasis.normalize();
              }
              else
                     FBasis=vec3f(0, 0, 0);
// Определяем ускорение
              vec3f acc=F.length()/mass*FBasis;
// Находим конечную скорость
              speed+=acc*delta_sec;
// Если вектор скорости не равен 0, определяем силу трения
              if (!equivalent(speed.length(), 0))
              {
```

```
// Находим проекцию вектора силы тяжести на вектор нормали
                    vec3f F2Normal=F2-F2Plane;
// Находим модуль вектора силы трения
                    vec3f F3Abs=F2Normal.length()*nju;
// Находим направление вектора силы трения (он направлен в
// противоположную сторону по сравнению с вектором скорости)
                    vec3f F3Basis=-speed_basis;
// Определяем изменение вектора скорости за интервал времени между двумя
// кадрами
                    vec3f delta_speed=F3Basis*F3Abs/mass*delta_sec;
// Если сила трения не изменяет направление вектора скорости
                    if (delta_speed.square_norm()<speed.square_norm())
// Изменяем вектор скорости
                           speed+=delta_speed;
                    else
// Иначе обнуляем вектор скорости (сила трения не может двигать объект)
                           speed=vec3f(0, 0, 0);
             }
// Если скорость наблюдателя не равна нулю, рассчитываем силу удержания
// траектории
             if (!equivalent(speed.length(), 0))
             {
// Определяем направление вектора скорости
                    speed_basis=speed;
                    speed_basis.normalize();
// Определяем направление вектора удержания траектории
                    vec3f F4Basis=F1Basis-speed_basis;
// Определяем максимальный модуль вектора изменения скорости, на который
// сила удержания скорости может изменить скорость наблюдателя (сила
// удержания траектории лишь удерживает объект на траектории, а не
// тянет объект за собой)
                    float
max_delta_speed_abs=(F4Basis*speed.length()).length();
// Определяем нормализованный вектор направления силы удержания
// траектории
                    if (!equivalent(F4Basis.square_norm(), 0))
                           F4Basis.normalize();
                    else
                           F4Basis=vec3f(0, 0, 0);
```

```
// Определяем вектор силы удержания траектории
                    vec3f F4=F4Basis*coerce;
// Определяем ускорение
                    acc=F4.length()/mass*F4Basis*delta_sec;
// Рассчитываем изменение скорости за интервал времени между кадрами
                    vec3f delta_speed=acc*delta_sec;
// Если скорость не изменяет направления
                    if (delta_speed.length()<max_delta_speed_abs)
// Запоминаем новую скорость
                           speed=speed+delta_speed;
                    else
                    {
// Иначе вектор скорости совпадает с вектором силы тяги
                           speed_basis=F1Basis;
                           speed=speed_basis*speed.length();
                    }
             }
             else
                    speed=vec3f(0, 0, 0);
// Если вектор скорости превышает максимально допустимую скорость
             if (speed.length()>max_speed)
             {
// Уменьшаем скорость до максимально допустимой
                    speed.normalize();
                    speed*=max_speed;
             }
// Определяем новое положение наблюдателя
             pos[0]+=speed[0]*delta_sec;
             pos[1]+=speed[1]*delta_sec;
       }
       else
       {
// Если пользователь находится в режиме полета
// Определеяем изменение скорости под действием силы тяжести
             speed[2]-=g*delta_sec;
// Определяем новую позицию наблюдателя
             pos+=speed*delta_sec;
       }
```

```
// Рисуем в буфере pbuffer модель долины. Ортогональная проекция, вид
// сверху
       pbuffer0.MakeCurrent();
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
       glLoadIdentity();
       glTranslatef(-pos[0], -pos[1], -max_extent[2]*model_scale);
       valley.call_list();
// Считываем значения глубины из буфера pbuffer
       GLfloat depth[2][2];
       glReadPixels(0, 0, 2, 2, GL_DEPTH_COMPONENT, GL_FLOAT,
&depth[0][0]);
// Определяем максимальное значение глубины в буфере pbuffer
       GLfloat max_depth=max(depth[0][0], depth[0][1]);
       max_depth=max(max_depth, depth[1][0]);
       max_depth=max(max_depth, depth[1][1]);
// Находим нормаль к поверхности под наблюдателем
       normal[0]=-(depth[0][0]-depth[0][1]+depth[1][0]-
depth[1][1])*(max_extent[2]-min_extent[2])/user_size/2.0;
       normal[1]=-(depth[0][0]-depth[1][0]+depth[0][1]-
depth[1][1])*(max_extent[2]-min_extent[2])/user_size/2.0;
       normal[2]=1;
       normal.normalize();
// Определяем высоту поверхности под наблюдателем
       z=(max_extent[2]-(max_extent[2]-
min_extent[2])*max_depth)*model_scale+h;
// Если наблюдатель идет по земле
       if (!bFly)
       {
// Если высота ниже критического уровня, наблюдатель переходит в
// режим полета
              if (z<pos[2]-upborne)
                     bFly=true;
              else
// Если новая высота лежит ниже критической
                     if (z<pos[2]+upborne)
                     {
// Ставим наблюдателя на поверхность
                            pos[2]=z;
// Если скорость наблюдателя не нулевая
                            if (!equivalent(speed.square_norm(),0))
```

```
                              {
// Поворачиваем вектор скорости (вектор скорости должен быть
// перпендикулярен к нормали)
                                    vec3f new_speed=speed;
                                    new_speed[2]=-
(normal[0]*new_speed[0]+normal[1]*new_speed[1])/normal[2];
                                    if
(!equivalent(new_speed.square_norm(), 0))
                                    {
                                          new_speed.normalize();

       speed=new_speed*speed.length();
                                    }
                                    else
                                          new_speed=vec3f(0, 0, 0);
                              }
                        }
                        else
                        {
// Иначе — наблюдатель столкнулся с препятствием
// Обнуляем скорость наблюдателя
                              speed[0]=0;
                              speed[1]=0;
speed[2]=0;
// Возвращаем его в старое положение
                              pos=last_pos;
                              z=last_z;
                              normal=last_normal;
                        }
       }
       else
       {
// Если наблюдатель находится в режиме полета
// Если наблюдатель приземлился или столкнулся с поверхностью
              if (pos[2]<z)
              {
// Если наблюдатель приземлился на поверхность
                        if (pos[2]>z-upborne)
                        {
```

```
// Ставим наблюдателя на поверхность
                              pos[2]=z;
                              bFly=false;
// Обнуляем z-компонент скорости
                              speed[2]=0;
                        }
                        else
                        {
// Если наблюдатель столкнулся с поверхностью
// Возвращаем его в прошлую позицию (по осям x и y)
                              pos[0]=last_pos[0];
                              pos[1]=last_pos[1];
                              z=last_z;
                              normal=last_normal;
// Если наблюдатель до сих пор в воздухе
                              if (last_z<pos[2]-upborne)
                              {
// Обнуляем скорость по осям x и y
                                    speed[0]=0;
                                    speed[1]=0;
                              }
                              else
                              {
// Иначе — ставим наблюдателя
                                    pos[2]=last_z;
speed[2]=0;
                                    bFly=false;
                              }
                        }
                  }
            }
// Переключаемся обратно в главное окно программы, устанавливаем камеру
// в позицию наблюдателя и рисуем сцену
            glutSetWindow(WinID);
            glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
            glLoadIdentity();
// Поворот вверх-вниз
            glRotatef(rotx, -1, 0, 0);
```

```
// Поворот влево-вправо
        glRotatef(rotz, 0, 0, 1);
        glTranslatef(-pos[0], -pos[1], -pos[2]);
        valley.call_list();
}
```

Полученная программа работает довольно стабильно. Но если вы "побродите" по долине некоторое время, то обнаружите существенный недостаток: при попытке зайти под крону дерева наблюдатель упирается в невидимую стену. В этом нет ничего удивительного. Наша программа предназначена для работы с однозначными поверхностями, поэтому она подразумевает, что высота поверхности в окрестностях кроны дерева равна высоте самой кроны. Следствием этого является то, что программа пытается приподнять наблюдателя на вершину кроны дерева. А поскольку крона, как правило, находится на существенной высоте над землей, программа просто останавливает наблюдателя перед кроной.

К сожалению, полное устранение этого недостатка требует пересмотра всей архитектуры программы, что в свою очередь приведет к лавинообразному увеличению ее исходного кода. Правда, существует альтернативный способ решения проблемы, который позволяет значительно смягчить этот недостаток (но не устранить его полностью).

Идея заключается в том, чтобы заставить программу использовать для определения высоты и вывода на экран две различные модели местности. Модель местности, используемая для определения высоты под наблюдателем, будет содержать "укороченные" кроны, чтобы наблюдатель смог ходить под ними, утолщенные столбы, чтобы не мог пройти сквозь них "с разбегу", высокие стены по краям модели, чтобы наблюдатель не мог упасть в пропасть с края сцены. В то же время все эти нововведения ни коим образом не повлияют на внешний вид программы, т. к. модифицированная модель будет использоваться только для служебных целей, а на экран будет выводится оригинальная модель.

Главный недостаток этого решения — нерациональное использование видеопамяти. Придется хранить в видеопамяти две практически одинаковые модели. Правда, при внимательном рассмотрении этот недостаток не кажется таким уж существенным: вторая (служебная) модель не использует текстуры, поэтому в памяти придется хранить только информацию о вершинах и полигонах модели, которая занимает относительно мало места.

Модифицированная модель находится на CD-диске, прилагаемом к книге, в файле \Examples\Ch7-PBuffer\3DSMAX_MODELS\Valley\VALLEY_HEGHT_MAP.max. Внешний вид измененной модели приведен на рис. 8.5.

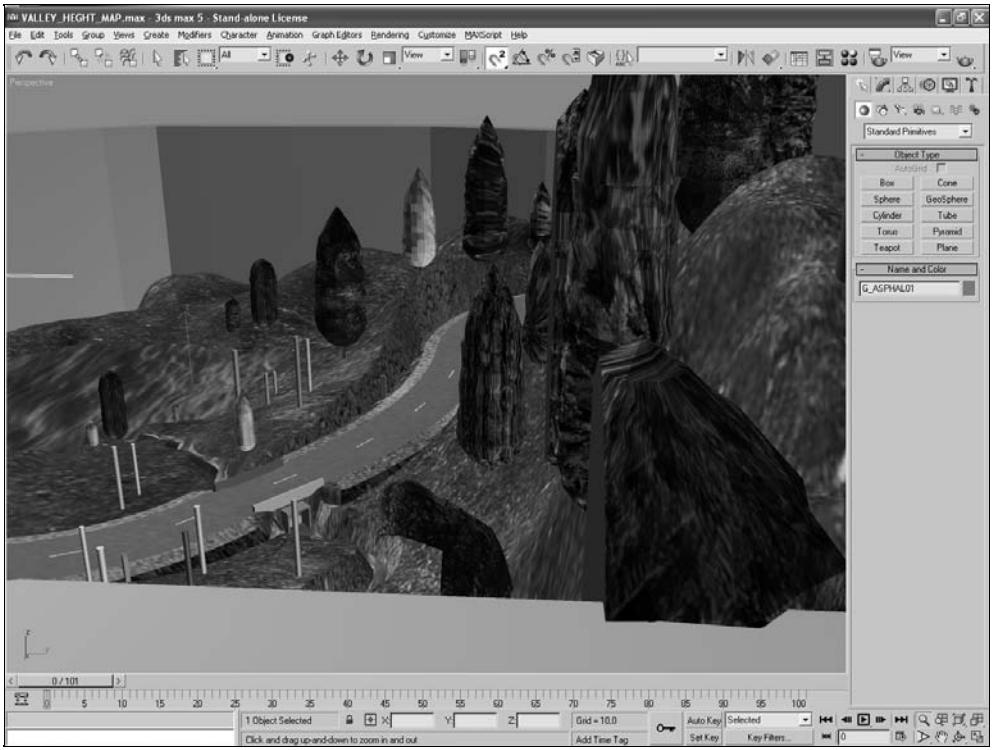


Рис. 8.5. Модифицированная модель долины.
Обратите внимание на то, что долина окружена высокими стенами,
а у многих деревьев кроны укорочены или вообще отсутствуют

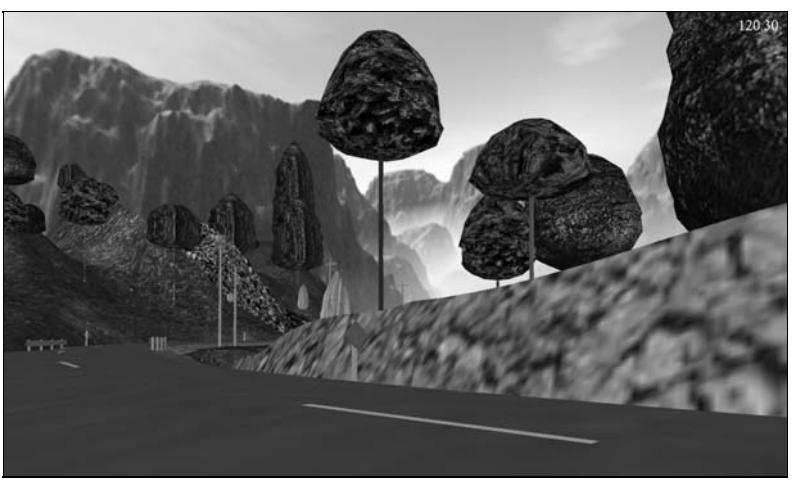


Рис. 8.6. Финальный вариант программы, моделирующей виртуальный мир

Я не буду приводить исходный код модифицированного примера (Ex11), т. к. все изменения, которые были в него внесены, довольно тривиальны: фактически была добавлена только загрузка второй модели из файла формата ASE и изменена одна строка функции `Display`. Как всегда, вы сможете найти исходный код примера на CD-диске. Кроме того, для улучшения внешнего вида программы я добавил в пример Ex11 рисование окружающей среды долины: неба и далеких гор. Код, отвечающий за рисование окружающей среды, был взят из примера Ex10 *главы 4*. Скриншот финальной версии программы приведен на рис. 8.6.

Заключение

В этой главе мы рассмотрели использование внеэкранных буферов пикселов `pbuffer`, представляющих собой специальную область видеопамяти, в которую программист может выводить изображение. Также было рассмотрено расширение ARB_render_texture, позволяющее непосредственно использовать содержимое буфера `pbuffer` в качестве текстуры, без промежуточного копирования. Так как создание и использование буфера `pbuffer` средствами WGL — довольно трудоемкая задача, в состав NVIDIA OpenGL SDK входит класс `PBuffer`, инкапсулирующий всю работу с `pbuffer`. В этой главе мы модифицировали класс `PBuffer`, добавив в него поддержку расширения ARB_render_texture. В заключение мы создали программу моделирования виртуального мира, использующую `pbuffer` для определения высоты местности под наблюдателем.

Глава 9

Сжатые текстуры

В настоящее время в программах, работающих с трехмерной графикой, как правило, используются 32-битные текстуры высокого разрешения (512×512 и выше). Однако за такое высокое качество текстур приходится расплачиваться увеличением объема памяти, занимаемого такими текстурами. В результате этого производителям видеокарт пришлось изобретать различные технологии хранения текстур большого объема. В настоящее время в видеокартах, как правило, одновременно реализуются два противоположных подхода к хранению больших текстур.

- Хранение текстур в оперативной памяти компьютера. Для реализации этого подхода был разработан специальный высокоскоростной интерфейс AGP (Accelerated Graphics Port), позволяющий видеокарте напрямую обмениваться информацией с оперативной памятью. Это позволяет увеличить скорость обмена между видеопамятью видеокарты и основной оперативной памятью. К сожалению, каких-либо приемлемых результатов в этой области пока не достигнуто: самый быстрый на сегодня интерфейс между видеокартой и процессором AGP8X способен передавать данные с *пиковой* скоростью около 2,1 Гбайт/с, в то время как видеокарта NVIDIA GeForce FX 5900 Ultra обладает пропускной способностью 27 Гбайт/с! Поэтому нет ничего удивительного в том, что хранение текстур в оперативной памяти приводит к катастрофическому падению производительности.
- Хранение текстур в видеопамяти. Реализация данного подхода требует значительного увеличения объема видеопамяти. В настоящее большинство видеокарт оснащается видеопамятью объемом от 64 до 256 Мбайт. Хотя этот объем кажется довольно внушительным, не следует забывать о том, что при разрешении экрана 1280×1024×32 размер кадрового буфера может превышать 10 Мбайт. А при использовании полноэкранного сглаживания (FSAA 4x) с двойной буферизацией вообще достигать астрономического размера в 40 Мбайт. В этом случае у видеокарты, имеющей объем видеопамяти 64 Мбайт для хранения текстур свободными останутся лишь 24 Мбайт видеопамяти. А ведь кроме текстур в видеопамяти могут храниться еще и

буферы `pbuffer` Использование большего объема видеопамяти (128 или 256 Мбайт) снижает остроту проблемы, но такие видеокарты, к сожалению, пока не получили широкого распространения на российском рынке.

Как видно, не один из подходов *пока* не решил проблему использования текстур высокого разрешения. И хотя в ближайшем будущем, эта проблема, возможно, будет решена путем простого наращивания объема видеопамяти (уже анонсированы видеокарты с объемом видеопамяти 512 Мбайт), в настоящее время проблема нехватки видеопамяти остается одной из самых острых.

В результате производителям видеокарт пришлось решать задачу увеличения качества текстур без одновременного увеличения объема видеопамяти. Выход был найден довольно быстро — фирмы S3 и 3dfx предложили форматы сжатия текстур S3TC и FXT1, которые позволяют сжимать текстуры в 6 и 8 раз соответственно (имеется в виду максимальная степень сжатия). Естественно, оба этих метода реализуют сжатие с потерей информации. Подобный метод сжатия используется форматом JPG. Однако в большинстве случаев качество сжатых текстур с увеличенной детализацией лучше качества несжатых текстур того же объема.

9.1. Расширение ARB_texture_compression

OpenGL, начиная с версии 1.3, поддерживает работу со сжатыми текстурами с помощью расширения ARB_texture_compression. Само по себе расширение ARB_texture_compression не умеет работать с каким-либо определенным форматом сжатых текстур. Вместо этого оно предоставляет программисту интерфейс для работы со сжатыми текстурами, который не привязан к какому-либо определенному формату сжатых текстур. Поэтому приложение, работающее со сжатыми текстурами и использующее это расширение, сможет нормально работать на любой видеокарте, поддерживающей сжатие текстур. При этом на видеокартах Voodoo 4 и Voodoo 5 будет автоматически использоваться формат сжатия текстур FXT1, а на GeForce и Radeon — формат S3TC.

Для сжатия текстуры при загрузке в видеопамять необходимо указать в качестве внутреннего формата текстуры константу `GL_COMPRESSED_RGB_ARB` или `GL_COMPRESSED_RGBA_ARB`. Для текстуры без альфа-канала указывается константа `GL_COMPRESSED_RGB_ARB`. А для текстуры с альфа-каналом указывается константа `GL_COMPRESSED_RGBA_`. После этого драйвер OpenGL сам выбирает оптимальный (с его точки зрения) внутренний формат для загружаемой текстуры.

В принципе, драйвер может по каким-то своим соображениям выбрать формат без сжатия для некоторых Mipmap-уровней текстуры, или, иными словами, не сжать текстуру. Для проверки того, что требуемый Mipmap-

уровень текстуры был действительно сжат, необходимо выполнить команду `glGetTexLevelParameteriv` с параметром `GL_TEXTURE_COMPRESSED_ARB`. Если она возвратила ненулевое значение, то сжатие прошло успешно, в противном случае произошла ошибка. Если Mipmap-уровень текстуры был сжат, то приложение может получить размер сжатого Mipmap-уровня при помощи команды `glGetTexLevelParameteriv` с параметром `GL_TEXTURE_IMAGE_SIZE_ARB`.

Внимание

Функция `glGetTexLevelParameteriv` с параметром `GL_TEXTURE_IMAGE_SIZE_ARB` корректно работает только со сжатыми Mipmap-уровнями.

Для демонстрации использования расширения ARB_texture_compression я переписал пример Ex04 из *главы 4* с использованием этого расширения, добавив в него автоматическое сжатие текстуры при загрузке с последующим приближенным определением коэффициента сжатия текстуры (Ex01).

Изменения в основном затронули обработчик консольной команды `load_texture` (листинг 9.1).

Листинг 9.1

```
bool onConsole(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
       if (Cmd=="load_texture")
       {
...
              pTextureImage=read(Params[0]);
              if (!pTextureImage)
              {
                     console->add(string("Can't open file
'")+Params[0]+string("'"));
                     return false;
              };

              texture.bind();
// Загружаем сжатую текстуру в видеопамять и сжимаем ее
              gluBuild2DMipmaps(texture.target, GL_COMPRESSED_RGB_ARB,
pTextureImage->width, pTextureImage->height,
                     pTextureImage->format, GL_UNSIGNED_BYTE,
pTextureImage->pixels);
```

```
// Проверяем, был ли сжат нулевой Mipmap-уровень
              int compressed;
              glGetTexLevelParameteriv(texture.target, 0,
GL_TEXTURE_COMPRESSED_ARB, &compressed);
// Если нулевой уровень был сжат
              if (compressed)
              {
                     console->add("Texture has been properly
compressed");
// Оцениваем приблизительный размер несжатой текстуры исходя из того, что
// большинство видеокарт используют внутренний формат GL_RGBA8 для
// несжатых текстур
                     int uncompressed_size=pTextureImage->
width*pTextureImage->height*4;
                     char str[200];
                     sprintf(str, "Uncompress image size: %d",
uncompressed_size);
                     console->add(&str[0]);
                     int compressed_size;
// Определяем размер нулевого Mipmap-уровня
                     glGetTexLevelParameteriv(texture.target, 0,
GL_TEXTURE_IMAGE_SIZE_ARB, &compressed_size);
                     sprintf(str, "Compress image size: %d",
compressed_size);
                     console->add(&str[0]);
// Определяем коэффициент сжатия текстуры
                     sprintf(str, "Compression ratio: %f",
float(uncompressed_size)/compressed_size);
                     console->add(&str[0]);
              }
              else
                     console->add("Texture has not been compressed");

              console->add(string("Texture \"")+Params[0]+string("\" has
been loaded"));
              return true;
       }
       return false;
}
```

На рис. 9.1 приведен внешний вид окна примера Ex01 сразу после запуска программы.

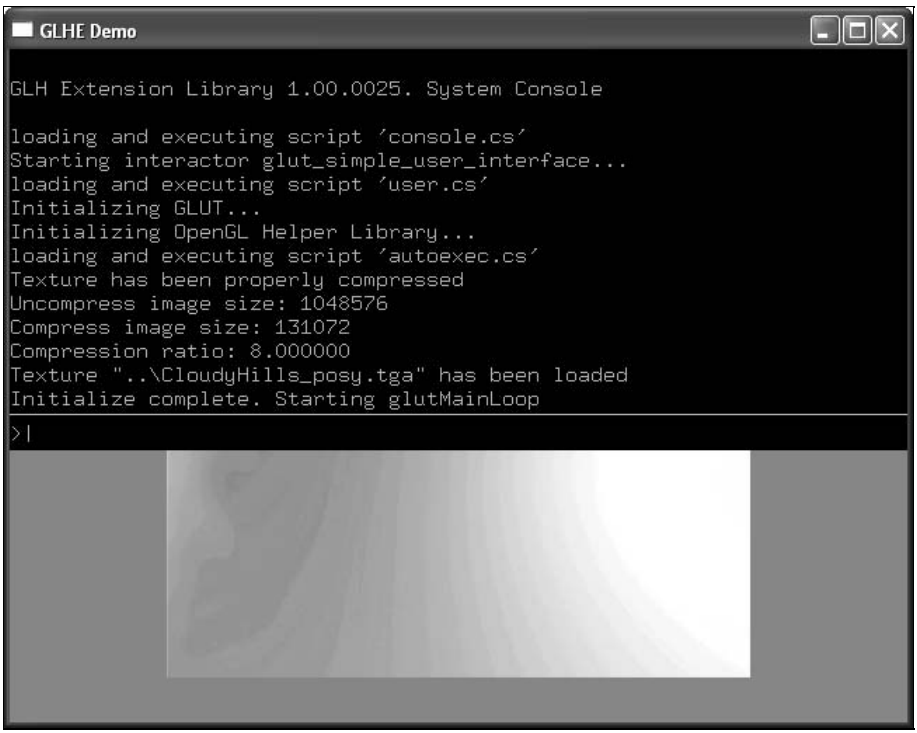


Рис. 9.1. Пример Ex01

Приложение может управлять качеством сжатия текстуры при помощи нового параметра команды `glHint` — `GL_TEXTURE_COMPRESSION_HINT_ARB`. Если этот параметр равен `GL_FASTEST`, драйвер постарается минимизировать размер сжатой текстуры, жертвуя качеством. Если же параметр `GL_TEXTURE_COMPRESSION_HINT_ARB` равен `GL_NICEST`, то драйвер постарается сохранить качество текстуры.

Однако часто реальные приложения нуждаются в более гибких средствах управления качеством сжатых текстур, чем средства, предоставляемые командой `glHint`. К сожалению, расширение ARB_texture_compression не предоставляет таких средств. Это связано с тем, что расширение ARB_texture_compression — это всего лишь интерфейс между прикладной программой и расширениями OpenGL, реализующими компрессию текстур. А алгоритм компрессии текстур в общем случае может быть любым: S3TC, FXT1, JPEG, LZW, GIF и т. д. Следовательно, любая попытка сделать рас-

ширение ARB_texture_compression более низкоуровневым сузит область его применения.

В результате программисту не остается иного выбора, кроме как изучать низкоуровневые расширения сжатия текстур. В настоящее время самым распространенным форматом сжатия текстур является S3TC (его поддерживали абсолютно все видеокарты, выпускаемые на момент написания книги), мы рассмотрим расширение EXT_texture_compression_s3tc, предоставляющее программисту расширенные средства для управления качеством сжатия текстур с использованием формата S3TC.

9.2. Расширение EXT_texture_compression_s3tc

Непосредственная работа с текстурами формата S3TC осуществляется с использованием расширения EXT_texture_compression_s3tc. В настоящее время это расширение поддерживает четыре формата текстур, которые приведены в табл. 9.1.

Таблица 9.1. *Форматы текстур, поддерживающиеся расширением EXT_texture_compression_s3tc*

Формат текстуры	Поддержка альфа-канала	Степень сжатия
GL_COMPRESS_RGB_S3TC_DXT1_EXT	Нет	6:1*
GL_COMPRESS_RGBA_S3TC_DXT1_EXT	Да (1 бит)	8:1
GL_COMPRESS_RGBA_S3TC_DXT3_EXT	Да (4 бита)	4:1
GL_COMPRESS_RGBA_S3TC_DXT5_EXT	Да (3 бита с интерполяцией)	4:1

* Так как большинство видеокарт используют вместо формата GL_RGB8 формат GL_RGBA8, реальная степень сжатия равна 8:1.

Примечание

Если вы раньше работали с DirectX, то наверняка заметили, что расширение EXT_texture_compression_s3tc не поддерживает форматы текстур DXT2 и DXT4, имеющиеся в DirectX. Это связано с тем, что эти форматы не имеют несжатых аналогов в OpenGL. В результате их поддержка потребует введения нового несжатого формата текстур RGBA_PREMULTIPLIED_ALPHA, что будет просто пустой тратой времени.

Для того чтобы понять особенности различных форматов S3TC, необходимо немного разбираться в их внутренней структуре. Поэтому в следующем раз-

деле мы коротко рассмотрим основы алгоритма компрессии S3TC и различных форматов сжатых текстур.

9.2.1. Алгоритм компрессии S3TC и форматы сжатых текстур S3TC

Принцип работы алгоритма S3TC-компрессии текстур можно описать следующим образом. Сначала текстура разбивается на блоки 4×4 текселя. В каждом блоке выбираются два наиболее характерных текселя (обычно наиболее яркий и наиболее темный). Эти тексели сохраняются в неизменном виде. Для остальных текселей запоминаются коэффициенты интерполяции, для хранения которых отводится очень маленькое число бит (обычно 2—3 бита).

Чтобы лучше понять алгоритм компрессии, рассмотрим процесс компрессии блока 4×4 текселя, изображенного на рис. 9.2, с использованием формата GL_COMPRESSED_RGB_S3TC_DXT1_EXT.

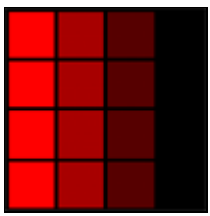


Рис. 9.2. Текстура из DirectX 8 SDK

Сначала выбираются два наиболее характерных цвета, которые запоминаются в 16-битном формате. В нашем случае этими цветами будут ярко-красный и черный (цвет 0 и цвет 1). В связи с тем, что человеческий глаз наиболее чувствителен к зеленому цвету, для хранения каналов R, G и B отводятся 5, 6 и 5 бит соответственно (5 + 6 + 5 = 16). Для хранения коэффициентов интерполяции каждого текселя отводится 2 бита. Но т. к. 2 бита явно не хватит для хранения двух коэффициентов интерполяции, вместо самих коэффициентов интерполяции будут храниться индексы из табл. 9.2. Такая таблица обычно хранится в ROM видеокарты.

Таблица 9.2. *Индексы интерполяции для формата GL_COMPRESSED_RGB_S3TC_DXT1_EXT*

Индекс (в двоичном виде)	Коэффициент перед цветом 0	Коэффициент перед цветом 1
00	1	0
01	0	1

Таблица 9.2 *(окончание)*

Индекс (в двоичном виде)	Коэффициент перед цветом 0	Коэффициент перед цветом 1
10	2/3	1/3
11	1/3	2/3

Например, если индекс текущего текселя равен 10, то его цвет будет рассчитываться по формуле: 2 / 3 × (цвет 0) + 1 / 3 × (цвет 1).

В итоге мы получаем два 16-битных цвета и 16 2-битных индексов. Для хранения этой информации необходимо 16 × 2 + 2 × 16 = 64 бита. В то же время несжатое 24-битное изображение требует для хранения 24 × 16 = 384 бита. Следовательно, мы получаем сжатие в 6 раз. Но платой за это является то, что в блоке 4×4 текселей не может быть больше четырех разных цветов, т. е. алгоритм производит сжатие с потерей информации.

Формат GL_COMPRESSED_RGBA_S3TC_DXT1_EXT отличается от предыдущего формата поддержкой работы с текстурами, содержащими 1 бит прозрачности (т. е. текстура может быть либо непрозрачной, либо 100% прозрачной). Такие текстуры очень удобны для создания текстур-трафаретов. Это достигается путем небольшой модификации таблицы индексов (табл. 9.3).

Таблица 9.3. *Индексы интерполяции для формата GL_COMPRESSED_RGBA_S3TC_DXT1_EXT*

Индекс (в двоичном виде)	Коэффициент перед цветом 0	Коэффициент перед цветом 1	Итоговый альфа-канал (0..1)
00	1	0	1
01	0	1	1
10	?	?	1
11	?	?	0

Как видно, индексу 11 теперь соответствует полностью прозрачная текстура. Но теперь в блоке 4×4 текселей не может быть больше трех разных цветов.

Из всего вышесказанного можно сделать два вывода. Во-первых, формат GL_COMPRESSED_RGBA_S3TC_DXT1_EXT *никогда* не следует использовать для сжатия полупрозрачных текстур, т. к. это приведет к сильному искажению альфа-канала. Во-вторых, формат сжатия GL_COMPRESS_RGBA_S3TC_DXT1_EXT уступает по качеству формату GL_COMPRESS_RGB_S3TC_DXT1_EXT.

Формат GL_COMPRESS_RGBA_S3TC_DXT3_EXT предназначен для хранения полупрозрачных текстур. В этом случае, каждый блок размером 4×4 текселя состоит из двух частей: блока альфа-компонентов текселей и 64-х битного блока с RGB-компонентами текселей.

В блоке альфа-каналов хранятся 4 старших бита альфа-канала каждого текселя, т. е. этот блок занимает $4 \times 16 = 64$ бита. Из этого можно сделать вывод, что формат GL_COMPRESS_RGBA_S3TC_DXT3 по качеству передачи прозрачности текстуры полностью аналогичен RGBA-текстурам формата (4 − 4 − 4 − 4). В итоге весь фрагмент размером 4×4 текселя занимает в памяти $64 + 64 = 128$ бит. В то же время несжатый 32-битный блок размером 4×4 текселя требует для хранения $32 \times 16 = 512$ бит. Следовательно, степень сжатия в этом случае равна $512 / 128 = 4$.

Формат сжатия текстур GL_COMPRESSED_RGBA_S3TC_DXT5_EXT отличается от GL_COMPRESSED_RGBA_S3TC_DXT3_EXT лишь тем, что блок альфа-канала хранится теперь в сжатом виде, причем алгоритм сжатия очень напоминает алгоритм сжатия RGB-блока. Сначала запоминаются альфа-каналы двух текселей, имеющих минимальную и максимальную прозрачность. Для остальных текселей запоминаются 3-битные индексы коэффициентов интерполяции (табл. 9.4).

Таблица 9.4. *Индексы интерполяции для формата GL_COMPRESSED_RGBA_S3TC_DXT5_EXT*

Индекс (в двоичном виде)	Коэффициент перед альфа-каналом 0	Коэффициент перед альфа-каналом 1
000	1	0
001	0	1
010	6/7	1/7
011	5/7	2/7
100	4/7	3/7
101	3/7	4/7
110	2/7	5/7
111	1/7	6/7

В блоке альфа-каналов в этом случае также хранится $16 \times 2 + 3 \times 16 = 64$ бита, т. е. степень сжатия полностью аналогична формату сжатия GL_COMPRESSED_RGBA_S3TC_DXT3_EXT.

После этого небольшого поверхностного обзора форматов компрессии попробуем ответить на вопрос, какой формат подходит для каких случаев.

Если мы сжимаем RGB-изображение, то наш единственный выбор — формат GL_COMPRESS_RGB_S3TC_DXT1_EXT. Хотя текстуры поверхностей реальных объектов сжимаются с нормальным качеством, не стоит пытаться сжимать текстуры, содержащие плавные цветовые переходы (например, текстуры неба или бликов от фар), или пестрые текстуры с большим разбросом цветов соседних текселей (например, текстуру, состоящую из случайных точек). Также не стоит забывать о том, что форматы сжатия текстур S3TC изначально разрабатывались для сжатия текстур высокого разрешения, поэтому попытка сжатия текстур низкого разрешения ни к чему хорошему не приведет.

Существуют два варианта сжатия изображения с одним битом прозрачности (спрайты и т. д). Если изображение довольно монотонное, то можно попробовать использовать формат GL_COMPRESSED_RGBA_S3TC_DXT1_EXT, в противном случае оптимальным выбором будут форматы GL_COMPRESSED_RGBA_S3TC_DXT3_EXT или GL_COMPRESSED_RGBA_S3TC_DXT5_EXT.

При сжатии полупрозрачного изображения выбор сделать не так уж просто. Если в пределах одного блока размером 4×4 текселя возможны хаотичные "скачки" прозрачности в широком диапазоне, то лучшим выбором скорее всего будет формат GL_COMPRESSED_RGBA_S3TC_DXT3_EXT, в противном случае — формат GL_COMPRESSED_RGBA_S3TC_DXT5_EXT.

Если же тип текстуры заранее не известен, то лучше всего использовать формат GL_COMPRESSED_RGBA_S3TC_DXT5_EXT, который в подавляющем большинстве случаев дает наилучшее качество изображения.

9.2.2. Использование расширения EXT_texture_compression_s3tc

Работа с расширением EXT_texture_compression_s3tc почти не отличается от работы с расширением ARB_texture_compression. Расширение EXT_texture_compression_s3tc просто предоставляет программисту четыре новых формата текстур:

- GL_COMPRESSED_RGB_S3TC_DXT1_EXT;
- GL_COMPRESSED_RGBA_S3TC_DXT1_EXT;
- GL_COMPRESSED_RGBA_S3TC_DXT3_EXT;
- GL_COMPRESSED_RGBA_S3TC_DXT5_EXT.

Эти форматы используются аналогично форматам GL_COMPRESSED_RGB_ARB или GL_COMPRESSED_RGBA_ARB расширения ARB_texture_compression.

Для демонстрации использования этого расширения мы перепишем пример Ex01 с его использованием. Но сначала нам надо решить несколько вопро-

сов. Во-первых, нам необходимо научить функцию `texImage2D` библиотеки NV_UTIL_EXT загружать текстуру в видеопамять с использованием требуемого внутреннего формата. Первое, что приходит в голову, — добавить в функцию `texImage2D` новый параметр: идентификатор внутреннего формата. К сожалению, у этого подхода есть один существенный недостаток. Дело в том, что внутренний формат текстуры часто зависит от количества цветовых компонентов текстуры. Например, для сжатия обычной RGB-текстуры оптимальнее всего использовать формат GL_COMPRESSED_RGB_S3TC_DXT1_EXT, а для текстуры с альфа-каналом — GL_COMPRESSED_RGBA_S3TC_DXT5_EXT. Следовательно, приложение должно самостоятельно определять требуемый формат пиксела перед каждым вызовом функции `texImage2D`, что увеличит размер исходного кода.

Гораздо лучше заставить функцию `texImage2D` брать всю информацию о форматах текстур из одномерного массива, в котором каждому элементу с индексом *i* будет соответствовать внутренний формат для текстуры с *i* компонентами цвета (листинг 9.2).

Листинг 9.2

```
// По умолчанию для текстуры с тремя цветовыми компонентами
// используется формат GL_RGB, а для текстуры с четырьмя компонентами —
// формат GL_RGBA
int INTERNAL_FORMATS[5]={0, 0, 0, GL_RGB, GL_RGBA};
bool texImage2D(tga::tgaImage* image, tex_object* tex_obj, bool bMip-
map=true)
{
        assert(image);
        GLenum target=tex_obj->target;

// Обратите внимание: вместо image->components используется
// INTERNAL_FORMATS[image->components]
        if (bMipmap)
                gluBuild2DMipmaps(target, INTERNAL_FORMATS
[image->components], image->width, image->height, image->format,
                GL_UNSIGNED_BYTE, image->pixels);
        else
                glTexImage2D(target, 0, INTERNAL_FORMATS
[image->components], image->width, image->height, 0, image->format,
                GL_UNSIGNED_BYTE, image->pixels);
        return true;
}
```

В этом случае для того, чтобы заставить функцию `texImage2D` использовать другой формат пикселов, достаточно изменить элементы массива `INTERNAL_FORMATS`. Но мы можем пойти еще дальше, предоставив пользователю возможность самостоятельно выбирать формат текстур, используемый функцией `texImage2D`. Для этого добавим в библиотеку NV_UTIL_EXT поддержку двух консольных команд: `set_texture_format_rgb` (задает формат RGB-текстуры) и `set_texture_format_rgba` (задает формат RGBA-текстуры). В листинге 9.3 приведен исходный текст обработчика этих консольных команд.

Листинг 9.3

```
bool NVUtilExtCmdProc(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
// Если текущая команда - set_texture_format_rgb или
// set_texture_format_rgba
       if ((Cmd=="set_texture_format_rgb") ||
(Cmd=="set_texture_format_rgba"))
       {
// Если нет параметров, показываем текущий формат пикселов
              if (Params.empty())
              {
// Определяем формат в зависимости от команды
                     int format;
                     if (Cmd=="set_texture_format_rgb")
                            format=INTERNAL_FORMATS[3];
                     else
                            format=INTERNAL_FORMATS[4];
// Расшифровываем код формата
                     switch(format)
                     {
                     case GL_RGB:
                            console->add("GL_RGB");
                            break;
                     case GL_RGBA:
                            console->add("GL_RGBA");
                            break;
                     case GL_RGB4:
                            console->add("GL_RGB4");
                            break;
```

```
                    case GL_RGBA4:
                        console->add("GL_RGBA4");
                        break;
                    case GL_RGB5:
                        console->add("GL_RGB5");
                        break;
                    case GL_RGB5_A1:
                        console->add("GL_RGB5_A1");
                        break;
                    case GL_RGB8:
                        console->add("GL_RGB8");
                        break;
                    case GL_RGBA8:
                        console->add("GL_RGBA8");
                        break;
                    case GL_COMPRESSED_RGB_ARB:
                        console->add("GL_COMPRESSED_RGB_ARB");
                        break;
                    case GL_COMPRESSED_RGBA_ARB:
                        console->add("GL_COMPRESSED_RGBA_ARB");
                        break;
                    case GL_COMPRESSED_RGB_S3TC_DXT1_EXT:
                        console->
add("GL_COMPRESSED_RGB_S3TC_DXT1_EXT");
                        break;
                    case GL_COMPRESSED_RGBA_S3TC_DXT1_EXT:
                        console->
add("GL_COMPRESSED_RGBA_S3TC_DXT1_EXT");
                        break;
                    case GL_COMPRESSED_RGBA_S3TC_DXT3_EXT:
                        console->
add("GL_COMPRESSED_RGBA_S3TC_DXT3_EXT");
                        break;
                    case GL_COMPRESSED_RGBA_S3TC_DXT5_EXT:
                        console->
add("GL_COMPRESSED_RGBA_S3TC_DXT5_EXT");
                        break;
                    default:
                        console->add("Unknown");
                        break;
```

```
                }
                return true;
        }
// Если параметр равен "/?", выводим справку по командам
        if(Params[0]=="/?")
        {
                if (Cmd=="set_texture_format_rgb")
                {
                        console->add("sets format of rgb-texture");
                        console->add("");
                        console->add("set_texture_format_rgb
<format>");
                }
                else
                {
                        console->add("sets format of rgba-texture");
                        console->add("");
                        console->add("set_texture_format_rgba
<format>");
                }
                console->add("where");
                console->add("  <format> is one of following
values");
                console->add("      RGB for GL_RGB");
                console->add("      RGBA for GL_RGBA");
                console->add("      RGB4 for GL_RGB4");
                console->add("      RGBA4 for GL_RGBA4");
                console->add("      RGB5 for GL_RGB5");
                console->add("      RGB5_A1 for GL_RGB5_A1");
                console->add("      RGB8 for GL_RGB8");
                console->add("      RGBA8 for GL_RGBA8");
                console->add("      COMPRESSED_RGB for
GL_COMPRESSED_RGB_ARB");
                console->add("      COMPRESSED_RGBA for
GL_COMPRESSED_RGBA_ARB");
                console->add("      DXT1 for
GL_COMPRESSED_RGB_S3TC_DXT1_EXT");
                console->add("      DXT1A for
GL_COMPRESSED_RGBA_S3TC_DXT1_EXT");
                console->add("      DXT3 for
GL_COMPRESSED_RGBA_S3TC_DXT3_EXT");
```

```
                    console->add("      DXT5 for
GL_COMPRESSED_RGBA_S3TC_DXT5_EXT");
                    return true;
             }
// Иначе - устанавливаем соответствующий формат
             int components;
// Определяем, для какого числа цветовых компонентов пользователь хочет
// установить формат
             if (Cmd=="set_texture_format_rgb")
                    components=3;
             else
                    components=4;
// Устанавливаем формат в зависимости от параметра
             if (Params[0]=="RGB")
             {
                    INTERNAL_FORMATS[components]=GL_RGB;
                    return true;
             }
             if (Params[0]=="RGBA")
             {
                    INTERNAL_FORMATS[components]=GL_RGBA;
                    return true;
             }
             if (Params[0]=="RGB4")
             {
                    INTERNAL_FORMATS[components]=GL_RGB4;
                    return true;
             }
             if (Params[0]=="RGBA4")
             {
                    INTERNAL_FORMATS[components]=GL_RGBA4;
                    return true;
             }
             if (Params[0]=="RGB5")
             {
                    INTERNAL_FORMATS[components]=GL_RGB5;
                    return true;
             }
             if (Params[0]=="RGB5_A1")
```

```
        {
                INTERNAL_FORMATS[components]=GL_RGB5_A1;
                return true;
        }
        if (Params[0]=="RGB8")
        {
                INTERNAL_FORMATS[components]=GL_RGB8;
                return true;
        }
        if (Params[0]=="RGBA8")
        {
                INTERNAL_FORMATS[components]=GL_RGBA8;
                return true;
        }
        if (Params[0]=="COMPRESSED_RGB")
        {
                INTERNAL_FORMATS[components]=GL_COMPRESSED_RGB_ARB;
                return true;
        }
        if (Params[0]=="COMPRESSED_RGBA")
        {
INTERNAL_FORMATS[components]=GL_COMPRESSED_RGBA_ARB;
                return true;
        }
        if (Params[0]=="DXT1")
        {
INTERNAL_FORMATS[components]=GL_COMPRESSED_RGB_S3TC_DXT1_EXT;
                return true;
        }
        if (Params[0]=="DXT1A")
        {
INTERNAL_FORMATS[components]=GL_COMPRESSED_RGBA_S3TC_DXT1_EXT;
                return true;
        }
        if (Params[0]=="DXT3")
        {
```

```
        INTERNAL_FORMATS[components]=GL_COMPRESSED_RGBA_S3TC_DXT3_EXT;
                        return true;
                }
                if (Params[0]=="DXT5")
                {

        INTERNAL_FORMATS[components]=GL_COMPRESSED_RGBA_S3TC_DXT5_EXT;
                        return true;
                }
                console->add(string("Unknown or unsupported pixel
format:")+ Params[0]);
                return false;
        }


        return false;
}
```

Заставлять пользователя вручную регистрировать обработчик команд очень не гуманно. Поэтому мы создадим функцию, выполняющую автоматическую регистрацию консольных команд библиотеки NV_UTIL_EXT в заданном экземпляре консоли (листинг 9.4).

Листинг 9.4

```
void nv_util_ext_register_console_cmds(glut_console* console)
{
        console->add("NVIDIA Utility Library Extension 1.00.0009");
        console->addCmd("set_texture_format_rgb", "set_texture_format_rgb
<format>", 0, NVUtilExtCmdProc);
// Регистрируем консольные команды
        console->addCmd("set_texture_format_rgba",
"set_texture_format_rgba <format>", 0, NVUtilExtCmdProc);
// Выполняем скрипт из файла nv_util_ext.cs, в котором могут находиться
// команды установки формата текстур по умолчанию
        console->processCmd("exec nv_util_ext.cs");
}
```

Теперь для загрузки текстуры из файла image.tga с последующим сжатием с использованием формата DXT1 пользователю достаточно выполнить следующие команды:

```
set_texture_format_rgb DXT1
load_texture image.tga
```

По идее, в примере Ex02 пользователь должен иметь возможность быстро изменять режим компрессии путем простого нажатия клавиши. Самая простая реализация такой задумки — создать набор скриптов, изменяющих текущий формат текстуры с последующей перезагрузкой текстуры из файла, после чего привязать созданные скрипты к горячим клавишам (например, <F1>—<F12>). Перезагрузка производится новой командой `reload_texture`. А привязка скриптов к горячим клавишам — командой `bind`. Создание новой консольной команды `reload_texture`, повторяющей вызов последней команды `load_texture`, продемонстрировано в листинге 9.5.

Листинг 9.5

```
bool onConsole(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
// Если команда load_texture
       if (Cmd=="load_texture")
       {
// Пропущено
...
// Запоминаем имя загруженного файла (используется командой
// reload_texture)
filename=Params[0];
              glutPostRedisplay();
              return true;
       }

// Если команда reload_texture
       if (Cmd=="reload_texture")
       {
// Если параметр равен "/?", то выводим справку
              if ((Params.size()==1) && (Params[0]=="/?"))
              {
                     console->add("reloads texture from file");
                     console->add("");
                     console->add("reload_texture");
                     return true;
              }
// Вызываем команду load_texture со старыми параметрами
              console->processCmd(string("load_texture ")+filename);
```

```
                return true;
        }

        return false;
}
```

Теперь остается только поместить в файл autoexec.cs исходный код скриптов переключения режимов компрессии текстур, и программа будет готова (листинг 9.6).

Листинг 9.6.

```
; Загружаем несжатую текстуру, устанавливая в качестве внутреннего
; формата GL_RGB/GL_RGBA
begin_script load_texture_rgb
text_out "Current texture format: RGB"
set_title RGB
set_texture_format_rgb RGB
set_texture_format_rgba RGBA
reload_texture
end_script

; Далее - по аналогии
begin_script load_texture_rgb5
text_out "Current texture format: RGB5"
set_title RGB5
set_texture_format_rgb RGB5
set_texture_format_rgba RGB5_A1
reload_texture
end_script

begin_script load_texture_rgb8
text_out "Current texture format: RGB8"
set_title RGB8
set_texture_format_rgb RGB8
set_texture_format_rgba RGBA8
reload_texture
end_script
```

```
begin_script load_texture_compressed_rgb
text_out "Current texture format: COMPRESSED_RGB"
set_title COMPRESSED_RGB
set_texture_format_rgb COMPRESSED_RGB
set_texture_format_rgba COMPRESSED_RGBA
reload_texture
end_script

begin_script load_texture_compressed_rgba
text_out "Current texture format: COMPRESSED_RGBA"
set_title COMPRESSED_RGBA
set_texture_format_rgb COMPRESSED_RGBA
set_texture_format_rgba COMPRESSED_RGBA
reload_texture
end_script

begin_script load_texture_dxt1
text_out "Current texture format: DXT1"
set_title DXT1
set_texture_format_rgb DXT1
set_texture_format_rgba DXT5
reload_texture
end_script

begin_script load_texture_dxt1a
text_out "Current texture format: DXT1A"
set_title DXT1A
set_texture_format_rgb DXT1A
set_texture_format_rgba DXT1A
reload_texture
end_script

begin_script load_texture_dxt3
text_out "Current texture format: DXT3"
set_title DXT3
set_texture_format_rgb DXT3
set_texture_format_rgba DXT3
reload_texture
end_script
```

```
begin_script load_texture_dxt5
text_out "Current texture format: DXT5"
set_title DXT5
set_texture_format_rgb DXT5
set_texture_format_rgba DXT5
reload_texture
end_script

bind F3 "run_script load_texture_rgb"
bind F4 "run_script load_texture_rgb5"
bind F5 "run_script load_texture_rgb8"
bind F6 "run_script load_texture_compressed_rgb"
bind F7 "run_script load_texture_compressed_rgba"
bind F8 "run_script load_texture_dxt1"
bind F9 "run_script load_texture_dxt1a"
bind F10 "run_script load_texture_dxt3"
bind F11 "run_script load_texture_dxt5"

load_texture ..\CloudyHills_posy.tga
run_script load_texture_rgb
```

С помощью этого примера я провел небольшое исследование изменения производительности программы при использовании сжатых текстур. В качестве текстуры использовалось изображение с разрешением 2048×2048, а разрешение экрана было установлено в 640×480 без использования FSAA. Результаты измерений приведены в табл. 9.5.

Таблица 9.5. *Результаты измерений производительности при использовании сжатых текстур*

Видеокарта	Производительность, FPS			
	GL_RGB8	GL_RGB5	DXT1	DXT5
NVIDIA GeForce2 MX 32 MB AGP	348	421	455	428
NVIDIA GeForce FX 5800 Ultra	2440	3044	3120	3058
ATI Radeon 9700 Pro	2049	2668	3146	3025

Из результатов измерений следует, что сжатые текстуры значительно быстрее 32-битных и 16-битных текстур даже тогда, когда программа не испытывает проблем с нехваткой видеопамяти. Это связано с тем, что сжатые текстуры занимают очень мало места в видеопамяти. Это приводит к тому, что их использование значительно снижает нагрузку на пропускную шину данных между графическим процессором и видеопамятью. Иными словами, если быстродействие приложения упирается в пропускную способность видеопамяти, то использование сжатых текстур поднимет его производительность. В противном случае влияние сжатых текстур на производительность будет минимальным.

Для демонстрации последнего утверждения я добавил в пример Ex11 из *главы 6* поддержку сжатых текстур, после чего измерил изменение производительности (Ex03). Результаты измерений приведены в табл. 9.6.

Таблица 9.6. *Изменение производительности примера Ex03 при использовании сжатых текстур*

Видеокарта	Производительность, FPS	
	GL_RGB8	DXT1
NVIDIA GeForce2 MX 32MB AGP	63	68
NVIDIA GeForce FX 5800 Ultra	101	107
ATI Radeon 9700 Pro	113	114

Результаты получились вполне предсказуемые: в примере Ex03 используется сцена, состоящая из большого количества полигонов (общее количество полигонов, обрабатываемых за один кадр, — более 130 000), в то же время общий объем текстур сцены не превышает 24 Мбайт (23 текстуры с разрешением 512×512). Следовательно, пример Ex03 больше нагружает GPU видеокарты, а не шину видеопамяти. Использование сжатых текстур обеспечило небольшой прирост производительности (около 5%) лишь на видеокартах с мощным GPU и с недостаточно большой пропускной способностью видеопамяти по отношению к GPU (GeForce2 MX, GeForce FX). Если же на видеокарте установлена очень быстрая видеопамять по отношению к GPU (Radeon 9700 Pro), то использование сжатых текстур практически не скажется на производительности примера Ex03 (прирост менее 1%) — пропускная способность видеопамяти не будет использоваться на полную мощность из-за недостаточной мощности GPU.

Из всего вышесказанного следует, что использование сжатых текстур всегда оказывает положительное влияние на производительность программы. Но платой за это является снижение качества изображения. Для оценки снижения качества изображения я провел небольшое исследование качества авто-

компрессии текстур тремя различными видеокартами[1]: GeForce2 MX, GeForce FX 5800 Ultra и ATI Radeon 9700 Pro. Результаты исследования приведены ниже. Я намеренно не привожу скриншоты, т. к. из-за черно-белой печати они все равно не дадут представления об изменении качества изображения.

Начнем с хорошего. Все видеокарты довольно неплохо справляются с компрессией текстур земли, дерева, травы и аналогичных текстур, не содержащих характерных мелких элементов и плавных переходов цвета. А вот с текстурами, имеющими плавные переходы цвета (неба, воды и т. д.) ситуация обстоит значительно хуже.

NVIDIA GeForce2 MX

Первое, что бросается в глаза при анализе качества сжатия текстур у GeForce 2 MX, — отвратительное качество автокомпрессии текстур с использованием формата GL_COMPRESSED_RGB_S3TC_DXT1_E0XT. В большинстве случаев качество сжатой текстуры намного хуже качества 16-битных текстур. Причина этого явления в следующем — при распаковке текстур формата GL_COMPRESSED_RGB_S3TC_DXT1_EXT видеокарта выполняет промежуточные вычисления с 16-битным текселем. А поскольку большинство коэффициентов, используемых при расчетах, — бесконечные периодические дроби, то 16-битная точность вычислений резко снижает итоговую точность расчетов. В результате при компрессии текстур, содержащих плавные переходы цвета, мы наблюдаем резкие переходы между цветами (бандинг).

К счастью, это явление наблюдается только при использовании форматов сжатия текстур GL_COMPRESSED_RGB_S3TC_DXT1_EXT и GL_COMPRESSED_RGBA_S3TC_DXT1_EXT. При распаковке текстур форматов GL_COMPRESSED_RGBA_S3TC_DXT3_EXT и GL_COMPRESSED_RGBA_S3TC_DXT5_EXT видеокарта выполняет вычисления с 32-битными текселями. В результате чего у программиста не остается иного выбора, кроме как использовать вместо формата GL_COMPRESSED_RGBA_S3TC_DXT1_EXT форматы GL_COMPRESSED_RGBA_S3TC_DXT3_EXT или GL_COMPRESSED_RGBA_S3TC_DXT5_EXT, имеющие вдвое меньшие коэффициенты сжатия, и соответственно, более низкую производительность.

При этом на видеокартах GeForce2 не имеет смысла использовать сжатый формат GL_COMPRESSED_RGB, т. к. он всегда соответствует формату GL_COMPRESSED_RGBA_S3TC_DXT1_EXT. Более того, ситуация с форматом GL_COMPRESSED_RGB даже более драматична, чем с форматом GL_COMPRESSED_RGBA_S3TC_DXT1_EXT. Дело в том, что новые вер-

[1] Более подробную информацию о качестве автокомпрессии вы сможете найти в [20] и [21].

сии драйверов NVIDIA позволяют форсировать использование формата GL_COMPRESSED_RGBA_S3TC_DXT3_EXT вместо GL_COMPRESSED_RGBA_S3TC_DXT1_EXT при помощи ключа `S3TCQuality` в секции OpenGL. Так вот, этот ключ не оказывает никакого влияния на формат текстур GL_COMPRESSED_RGB, в результате чего пользователь никак не сможет повлиять на качество распаковки текстур.

NVIDIA GeForce FX 5800 Ultra

Видеокарты GeForce FX порадовали отличным качеством компрессии текстур: даже при компрессии текстур с плавными цветовыми переходами качество изображения практически не изменялось. Поэтому в большинстве случаев программист может смело сжимать текстуры с использованием форматов GL_COMPRESSED_RGB и GL_COMPRESSED_RGBA, не беспокоясь о качестве изображения.

ATI Radeon 9700 Pro

Качество автокомпрессии у ATI Radeon 9700 Pro находится где-то по середине между GeForce 2 и GeForce FX. С одной стороны, качество компрессии текстур довольно хорошее, но с другой — оно все-таки не настолько хорошее, чтобы можно было сжимать все текстуры без разбору.

В ходе исследования качества компрессии текстур у ATI Radeon 9700 Pro было обнаружено немного странное поведение видеокарты при использовании форматов текстуры GL_RGB и GL_RGBA. Как известно, эти форматы просто указывают на то, что в текстуре присутствуют соответственно три или четыре цветовых компонента, не задавая при этом фиксированного формата текселя текстуры. Сам же внутренний формат текстуры определяется драйвером исходя из настроек. Например, если пользователь установил в настройках драйвера максимальное качество изображения, то драйвер, скорее всего, выберет в качестве внутреннего формата текстуры GL_RGB8 или GL_RGBA8. Если же драйвер настроен на достижение максимальной производительности в ущерб качеству, то он может преобразовать текстуры в 16-битные, или даже сжать их.

Видеокарты ATI ведут себя более агрессивно: даже при настройках драйвера `High Quality` (высшее качество) они могут преобразовать текстуры формата GL_RGB/GL_RGBA в 16-битные или сжать их. И хотя корпорация ATI утверждает, что драйвер уменьшает размер текстуры только в том случае, если это никак не скажется на качестве, в действительности ситуация обстоит намного хуже. Например, драйвер может ни с того, ни с сего преобразовать текстуру неба в 16-битный формат, в результате чего на небе появится сильный бандинг. Выход из данной ситуации — всегда задавать явным образом требуемый формат текстуры, не полагаясь на драйвер.

Выводы

Как видно, практически каждый ускоритель требует к себе индивидуального подхода. Пытаться учесть особенности работы всех существующих ускорителей в программе — дело не благодарное, и вряд ли реализуемое. Гораздо лучше просто предоставить пользователю возможность самому выбирать внутренний формат текстуры. При этом текстуры желательно разделить на несколько групп (текстуры пола/стен, неба, карты теней и т. д.) и позволить пользователю по отдельности настраивать формат сжатия каждой группы текстур.

Для того чтобы грамотно использовать мощный инструмент, надо хорошо знать как его сильные, так и слабые стороны. Давайте попробуем составить требования к текстуре, которая сжималась бы с наихудшим качеством:

- Текстура должна быть невысокого разрешения.
- Текстура должна содержать плавный переход цветов.
- Каждый соседний пиксель текстуры должен быть своего цвета.
- Цвета пикселов должны быть подобраны таким образом, чтобы не один из цветов внутри блока 4×4 не мог быть получен путем интерполяции двух соседних цветов.

Остановимся на последнем пункте. Ранее говорилось, что в процессе компрессии текстуры видеокарта выбирает два цвета внутри текстуры, а остальные цвета вычисляются по формуле $C = C1 \times k1 + k2 \times C2$. Можно заметить, что коэффициенты k1 и k2 связаны соотношением $k2 = 1 - k1$, т. е. формулу вычисления цвета можно переписать следующим образом: $C = C1 \times k1 + (1 - k1) \times C2$. Но т. к. каждый цвет состоит из трех компонентов, видеокарте приходится решать систему из трех уравнений с одним неизвестным:

$$R = R1 \times k1 + (1 - k1) \times R2;$$

$$G = G1 \times k1 + (1 - k1) \times C2;$$

$$B = B1 \times k1 + (1 - k1) \times B2.$$

А такая система, как известно, не всегда имеет решение. Следовательно, нам необходимо подобрать цвета C, C1 и C2 таким образом, чтобы эта система уравнений не имела решения, или, иными словами, чтобы ни один из цветов внутри блока 4×4 не мог быть получен путем интерполяции двух соседних цветов.

Для того чтобы получить такую текстуру, я написал небольшую программу на Borland Delphi 7 (Ex04). Ниже приведен фрагмент кода, отвечающего за генерацию текстуры:

```
for j:=0 to Image1.Picture.Graphic.Height-1 do
 for i:=0 to Image1.Picture.Graphic.Width-1 do
```

```
   begin
    Image1.Picture.Bitmap.Canvas.Pixels[i, j]:=
      RGB(Trunc((1.0-i/Image1.Picture.Graphic.Width)*255),
          Trunc((i/Image1.Picture.Graphic.Width)*0.5+((1.0-
j/Image1.Picture.Graphic.Width)*0.5)*255),
          Trunc(j/Image1.Picture.Graphic.Height*255));
   end;
```

При помощи этой программы была создана текстура 32×32 (рис. 9.3), которая затем была сжата с использованием формата DXT1.

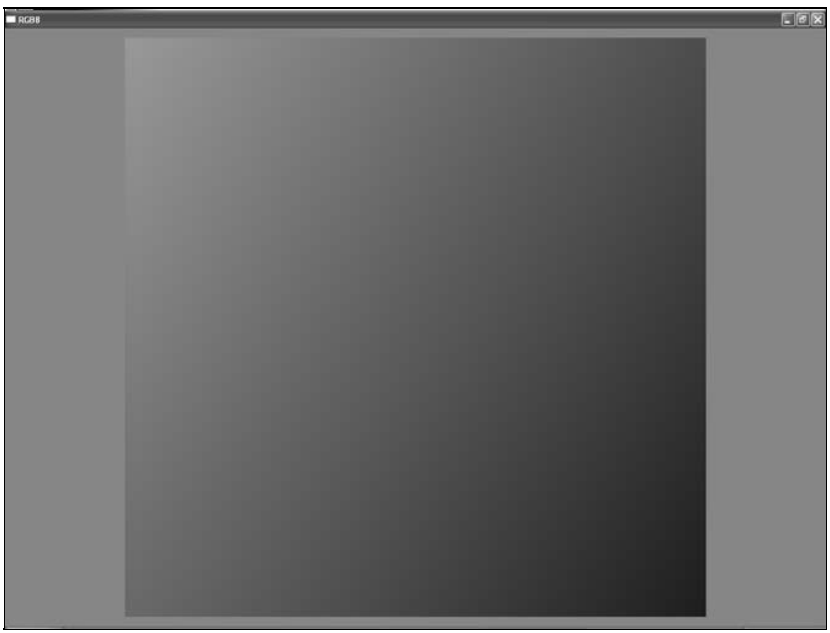


Рис. 9.3. Исходная текстура

Результат компрессии изображен на рис. 9.4.

После сжатия качество текстуры ухудшилось катастрофически — изображение фактически распалось на отдельные квадраты. И это при том, что скриншот был получен на видеокарте GeForce FX 5800 Ultra. На видеокарте GeForce2 при использовании GL_COMPRESS_RGB_S3TC_DXT1_EXT качество сжатия будет еще хуже. Для того чтобы увидеть причину появления этих квадратов, необходимо выключить фильтрацию текстур (рис. 9.5).

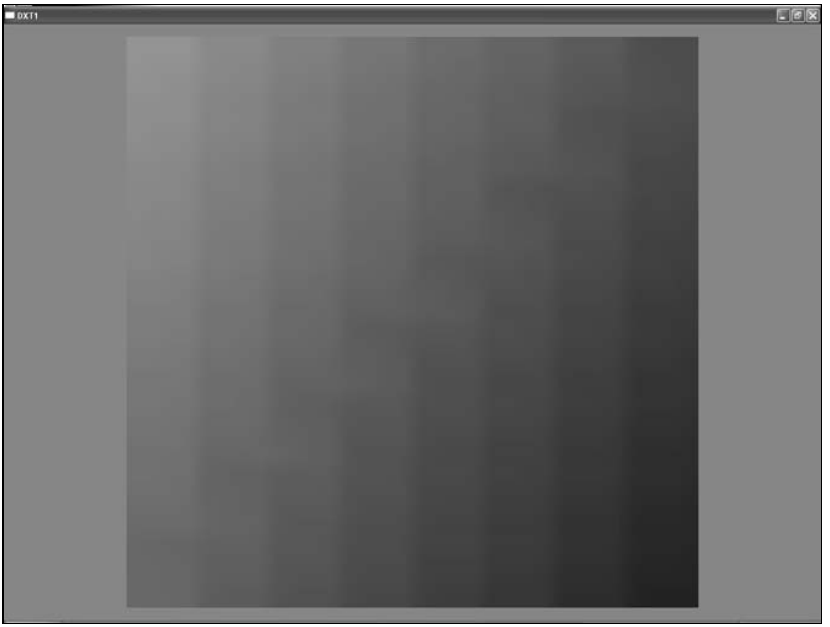


Рис. 9.4. Сжатая текстура

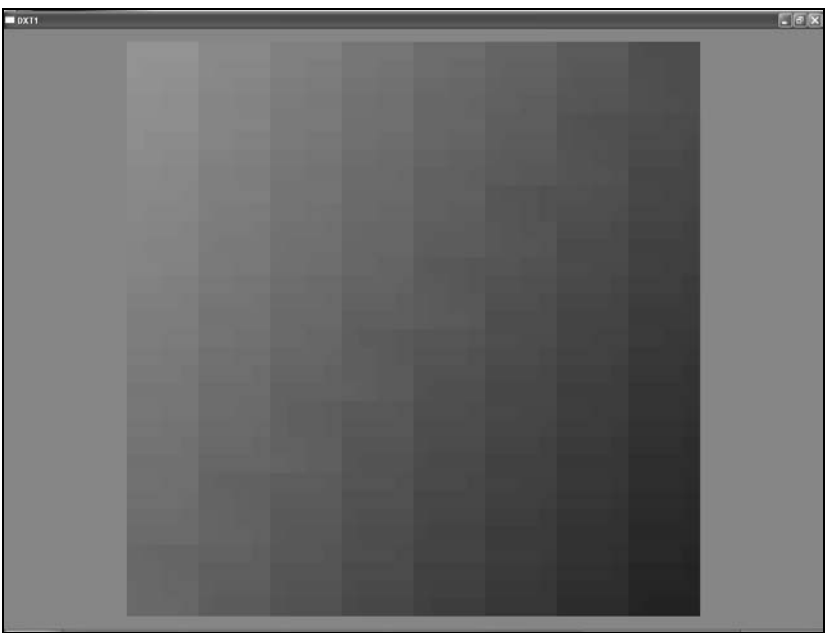


Рис. 9.5. Сжатая текстура с выключенной фильтрацией

Как видно, "квадраты" — это ничто иное, как блоки текселей размером 4×4, причем каждый блок состоит только из четырех разных цветов. Самое интересное, что эта текстура переводится в 16-битный формат с минимальной потерей качества.

9.3. Сохранение сжатых текстур на диске

Хотя сжатые текстуры и занимают меньше места в видеопамяти, на диске они хранятся в несжатом виде. Поэтому у разработчика может возникнуть естественное желание хранить сжатые текстуры на диске в сжатом виде для экономии свободного места. Кроме того, это позволит уменьшить время, затрачиваемое на загрузку текстур, т. к. компрессия большого объема текстур может занимать довольно много времени (до нескольких десятков секунд).

Для того чтобы сохранить сжатую текстуру на диске, программист должен выполнить следующие действия (подразумевается, что сжатая текстура уже хранится в видеопамяти):

1. Выполнить команду `glGetTexLevelParameteriv` с параметром `GL_TEXTURE_COMPRESSED_ARB`, чтобы убедится, что текстура хранится в сжатом виде.
2. Выполнить команду `glGetTexLevelParameteriv` с параметром `GL_TEXTURE_INTERNAL_FORMAT` для определения формата сжатия текстуры. Эту величину надо будет сохранить, т. к. она понадобится при распаковке текстур.

Для каждого Mipmap-уровня:

1. Определить и сохранить на диске размер текстуры при помощи команды `glGetTexLevelParameteriv` с параметром `GL_TEXTURE_IMAGE_SIZE_ARB`.
2. При помощи команды `glGetCompressedTexImageARB` получить образ сжатой текстуры и сохранить его на диске.

Загрузка текстуры из файла выполняется в точности наоборот:

1. Загрузка сохраненных параметров текстуры.
2. Загрузка каждого Mipmap-уровня сжатой текстуры в видеопамять с использованием новой команды `glCompressedTexImage2DARB`.

Команда `glCompressedTexImage2DARB` предназначена для загрузки образов *сжатых* текстур в видеопамять:

```
void glCompressedTexImage2DARB(enum target, int level,enum
internalformat, sizei width, sizei height, int border, sizei imageSize,
const void *data);
```

где:

- `target` — целевая текстура;
- `level` — номер уровня детализации (Mipmap-уровня);
- `internalformat` — внутренний формат текстуры;
- `width` — ширина текстуры;
- `height` — высота текстуры;
- `border` — ширина границы;
- `imageSize` — размер образа сжатой текстуры;
- `data` — указатель на область памяти, в которой хранится образ сжатой текстуры.

Для того чтобы сделать все вышесказанное более понятным, мы добавим в пример Ex02 возможность сохранять и считывать сжатые текстуры с диска (Ex05). Для хранения сжатых текстур на диске я разработал свой собственный простой формат сжатых текстур — SCI (S3 Compression Image). Его структура приведена в табл. 9.7.

Таблица 9.7. *Структура файла формата SCI*

Смещение от начала файла	Размер (байт)	Содержание
0	4	Строка "SCI" с нулевым символом на конце
4	4	Длина изображения
8	4	Ширина изображения
12	4	Формат сжатой текстуры
16	4	Размер изображения нулевого Mipmap-уровня
20	(Размер изображения нулевого Mipmap-уровня)	Нулевой Mipmap-уровень
20 + (Размер изображения нулевого Mipmap-уровня)	4	Размер 1-го Mipmap-уровня
...	...	...

Для сохранения сжатой текстуры на диске мы добавим в программу новую консольную команду: `save_texture`. Загрузка сжатой текстуры из файла будет осуществляться при помощи старой консольной команды: `load_texture`, в которую мы добавим возможность работы со сжатыми текстурами.

Следовательно, для того, чтобы сжать файл eburg.jpg с использованием формата GL_COMPRESS_RGB_S3TC_DXT1_EXT и сохранить результат в файл eburg.sci, необходимо выполнить три команды:

```
set_texture_format_rgb DXT1
load_texture eburg.jpg
save_texture eburg.sci
```

А для того чтобы повторно загрузить уже сжатый файл eburg.sci, надо выполнить команду:

```
load_texture eburg.sci
```

В листинге 9.7 приведен исходный код измененного обработчика консольных команд.

Листинг 9.7

```
bool onConsole(void* Object, string Cmd, vector<string> Params,
glut_console* console)
{
// Команда load_texture
       if (Cmd=="load_texture")
       {
              if (Params.size()==0)
                     return false;
// Справка
              if (Params[0]=="/?")
              {
                     console->add("loads texture from file");
                     console->add("");
                     console->add("load_texture <filename>");
                     console->add("where <filename> is file with
texture");
                     return true;
              }
// Удаляем старое изображение
              if (pTextureImage)
              {
                     if (pTextureImage->pixels)
                     {
                            delete[] pTextureImage->pixels;
                            pTextureImage->pixels=0;
```

```
                        }
                        delete pTextureImage;
                        pTextureImage=0;
                }
// Активизируем текстуру
                texture.bind();
// Определяем расширение файла
                int pos=Params[0].find_last_of(".");
                string ext="";
                if ((pos==-1) || (pos==Params[0].size()-1))
                {
                        console->add("Please, specify a  file name with
extension");
                };
                ext=Params[0].substr(pos+1, Params.size()-pos);

// Если расширение равно SCI
                if (Upper(ext)==string("SCI"))
                {
// Выделяем место для структуры с информацией о текстуре
                        pTextureImage=new tga::tgaImage;
                        pTextureImage->pixels=0;
// Открываем файл для чтения
                        ifstream file(Params[0].c_str(), ios::binary);
// Считываем идентификатор текстуры
                        char str[4];
                        file.read((char *) &str[0], sizeof(str));
// Если он не равен SCI, то текстура повреждена
                        if (strcmp(&str[0],"SCI"))
                        {
                                console->add("Bad sci-file");
                                return false;
                        };
// Считываем размеры текстуры
                        int width;
                        int height;
                        file.read((char *) &width, sizeof(width));
                        file.read((char *) &height, sizeof(height));
                        pTextureImage->width=width;
                        pTextureImage->height=height;
```

```
// Считываем внутренний формат текстуры
                    GLint internalFormat;
                    file.read((char *) &internalFormat,
sizeof(internalFormat));
// Выводим информацию о внутреннем формате текстуры
                    switch (internalFormat)
                    {
                    case GL_COMPRESSED_RGB_S3TC_DXT1_EXT:
                           console->add("DXT1 format");
                           break;
                    case GL_COMPRESSED_RGBA_S3TC_DXT3_EXT:
                           console->add("DXT3 format");
                           break;
                    case GL_COMPRESSED_RGBA_S3TC_DXT5_EXT:
                           console->add("DXT5 format");
                           break;
                    default:
                           console->add("Error: unsupported format");
                           return false;
                    }
// Определяем количество цветовых компонентов в текстуре
                    if
(internalFormat==GL_COMPRESSED_RGB_S3TC_DXT1_EXT)
                           pTextureImage->components=3;
                    else
                           pTextureImage->components=4;
// Загружаем все Mipmap-уровни
                    int LOD=0
// Пока размер и высота не равны 0
                    while (!((width==0) && (height==0)))
                    {
// Если высота или размер равны 0, то устанавливаем их равными 1
                           if (width==0)
                                  width=1;
                           if (height==0)
                                  height=1;
// Считываем размер сжатого Mipmap-уровня
                           int compressed_size;
```

```
                                file.read((char *) &compressed_size,
sizeof(compressed_size));
// Выделяем место для хранения Mipmap-уровня
                                char* compressedImage=new
char[compressed_size];
// Считываем текущий Mipmap-уровень
                                file.read(compressedImage, compressed_size);
                                glCompressedTexImage2DARB(texture.target,
LOD, internalFormat, width, height,

0, compressed_size, compressedImage);
                                console->addGLErrorInfo();
// Удаляем изображение Mipmap-уровня
                                delete[] compressedImage;
// Вычисляем размеры следующего Mipmap-уровня
                                LOD++;
                                width/=2;
                                height/=2;
                        }
// Текстура не может быть пережата с использование другого формата
                        bCanBeRecompress=false;

                        console->add(string("texture ")+Params[0]+string("
have been loaded"));
                }
                else
                {
// Иначе — загружаем текстуру как обычно
                        pTextureImage=read(Params[0]);
                        if (!pTextureImage)
                        {
                                console->add(string("Can't open file
'")+Params[0]+string("'"));
                                return false;
                        };
// Текстура может быть пережата
                        bCanBeRecompress=true;
// Загружаем текстуру в видеопамять
                        texImage2D(pTextureImage, &texture);
                }
```

```
// Выводим информацию о текстуре
              int compressed;
              glGetTexLevelParameteriv(texture.target, 0,
GL_TEXTURE_COMPRESSED_ARB, &compressed);
              if (compressed)
              {
                     console->add("Texture has been properly
compressed");
                     int uncompressed_size=pTextureImage->
width*pTextureImage->height*4;
                     char str[200];
                     sprintf(str, "Uncompress image size: %d",
uncompressed_size);
                     console->add(&str[0]);
                     int compressed_size;
                     glGetTexLevelParameteriv(texture.target, 0,
GL_TEXTURE_IMAGE_SIZE_ARB, &compressed_size);
                     sprintf(str, "Compress image size: %d",
compressed_size);
                     console->add(&str[0]);
                     sprintf(str, "Compression ratio: %f",
float(uncompressed_size)/compressed_size);
                     console->add(&str[0]);
              }
              else
                     console->add("Texture has not been compressed");
              console->add(string("Texture \"")+Params[0]+string("\" has
been loaded"));
              filename=Params[0];
              glutPostRedisplay();
              return true;
       }
// Повторяем вызов предыдущей команды load_texture
       if (Cmd=="reload_texture")
       {
              if ((Params.size()==1) && (Params[0]=="/?"))
              {
                     console->add("reloads texture from file");
                     console->add("");
                     console->add("reload_texture");
                     return true;
              }
```

```
// Выполняется только в том случае, если текстура может быть пережата
              if (bCanBeRecompress)
                     console->processCmd(string("load_texture
")+filename);

              return true;
       }
// Сохраняем текстуру в файл
       if (Cmd=="save_texture")
       {
// Справка по команде
              if ((Params.size()==0) || (Params[0]=="/?"))
              {
                     console->add("Saves texture.");
                     console->add("");
                     console->add("savetexture <filename>");
                     console->add("where <filename> is sci-file");
              }
              else
              {
// Если текстура не загружена
                     if (pTextureImage==0)
                     {
// Выводим сообщение об ошибке
                            console->add("Texture image is not loaded");
                     }
                     else
                     {
// Проверяем, сжата ли текстура
                            int compressed;
                            glGetTexLevelParameteriv(texture.target, 0,
GL_TEXTURE_COMPRESSED_ARB, &compressed);
// Если текстура не сжата
                            if (!compressed)
                            {
// Выводим сообщение об ошибке
                                   console->add("Cannon save uncompress
texture");
                                   return false;
                            }
```

```
// Открываем файл для записи
                        ofstream file(Params[0].c_str(),
ios::binary);
// Если ошибка, выводим сообщение
                        if (!file)
                        {
                                console->add(string("Could not create
file: ")+Params[0]);
                                return false;
                        }

// Записываем идентификатор текстуры
                        char str[4]="SCI";
                        file.write((char *) &str[0], sizeof(str));

// Записываем размеры текстуры
                        int width=pTextureImage->width;
                        int height=pTextureImage->height;
                        file.write((char *) &width, sizeof(width));
                        file.write((char *) &height,
sizeof(height));

// Записываем внутренний формат текстуры
                        GLint internalFormat;
                        glGetTexLevelParameteriv(texture.target, 0,
GL_TEXTURE_INTERNAL_FORMAT, &internalFormat);
                        file.write((char *) &internalFormat,
sizeof(internalFormat));

// Перебираем Mipmap-уровни
                        int LOD=0;
                        while (!((width==0) && (height==0)))
                        {
// Если размер равен 0, изменяем его на единицу
                                if (width==0)
                                        width=1;
                                if (height==0)
                                        height=1;
```

```
// Получаем размер текущего Mipmap-уровня
                                    int compressed_size;
              glGetTexLevelParameteriv(texture.target, LOD,
GL_TEXTURE_IMAGE_SIZE_ARB, &compressed_size);
                                    console->addGLErrorInfo();
                                    file.write((char *) &compressed_size,
sizeof(compressed_size));
// Выделяем место для хранения Mipmap-уровня
                                    char* compressedImage=new
char[compressed_size];
// Получаем изображение текущего Mipmap-уровня
       glGetCompressedTexImageARB(texture.target, LOD, compressedImage);
                                    console->addGLErrorInfo();
// Записываем его в файл
                                    file.write(compressedImage, com-
pressed_size);
// Освобождаем память
                                    delete[] compressedImage;
// Вычисляем параметры следующего Mipmap-уровня
                                    LOD++;
                                    width/=2;
                                    height/=2;
                              }
                              console->add(string("texture
")+Params[0]+string(" have been saved"));
                        }
              }
              return true;
       }

       return false;
}
```

Тем не менее разработка своего собственного формата для хранения сжатых текстур — не самое лучшее решение проблемы. В большинстве случаев гораздо лучше пользоваться каким-нибудь распространенным форматом. Поэтому в следующем разделе мы рассмотрим формат DDS, который практически является стандартом хранения сжатых текстур.

9.4. Использование файлов формата DDS

Большинство реальных программ хранят сжатые текстуры в файлах формата DDS (Direct Draw Surface). Как видно из названия, этот формат изначально предназначен для приложений, использующих API DirectX. Но, несмотря на это, он хорошо уживается и с OpenGL. Формат DDS обладает следующими преимуществами по сравнению с другими распространенными форматами графических файлов (BMP, JPG, TGA, GIF, TIFF): возможностью хранения в одном файле текстуры вместе со всеми Mipmap-уровнями, возможностью сохранения внутри файла набора двумерных текстур, и, наконец, поддержкой форматов сжатых текстур S3TC (DXT1, DXT2, DXT3, DXT4 и DXT5). Кстати, на CD-диске в каталоге \NVIDIA SDK\Textures имеется библиотека текстур формата DDS, занимающая более 200 Мбайт.

Существует множество утилит для создания DDS-файлов. В этой книге для создания сжатых файлов мы воспользуемся комплектом утилит DXT TOOLS 5.32 корпорации NVIDIA, которые вы сможете найти на CD-диске в каталоге NVIDIA.SDK\DXT_TOOLS_v5.32. В этот комплект утилит входят следующие средства:

- Thumb Nail Viewer — средство для просмотра содержимого DDS-файлов в проводнике Windows в режиме эскиза страниц;
- plug-in для Adobe Photoshop, добавляющий поддержку файлов формата DDS;
- plug-in для Adobe Photoshop, позволяющий преобразовывать карту высот в карту нормалей (мы будем использовать этот plug-in в *главе 10*);
- Command Line Compression Tools — средства для работы со сжатыми текстурами из командой строки;
- DXT Compression/Decompression Library — библиотека, позволяющая читать изображения из файлов формата DDS и сохранять (!) изображения в файлы формата DDS.

Рассмотрим эти утилиты подробнее.

9.4.1. Thumb Nail Viewer

Эта утилита встраивается в проводник Windows, позволяя просматривать содержимое файлов формата DDS непосредственно из проводника. Для установки утилиты запустите файл install.bat из каталога \DXT_TOOLS_v5.32\ ThumbNailViewer.

Теперь запускайте проводник и переходите в каталог, содержащий файлы формата DDS. Остается только выбрать в меню **Вид** пункт **Эскизы страниц**, и на экране появится уменьшенная копия рисунка, хранящегося в файле (рис. 9.6).

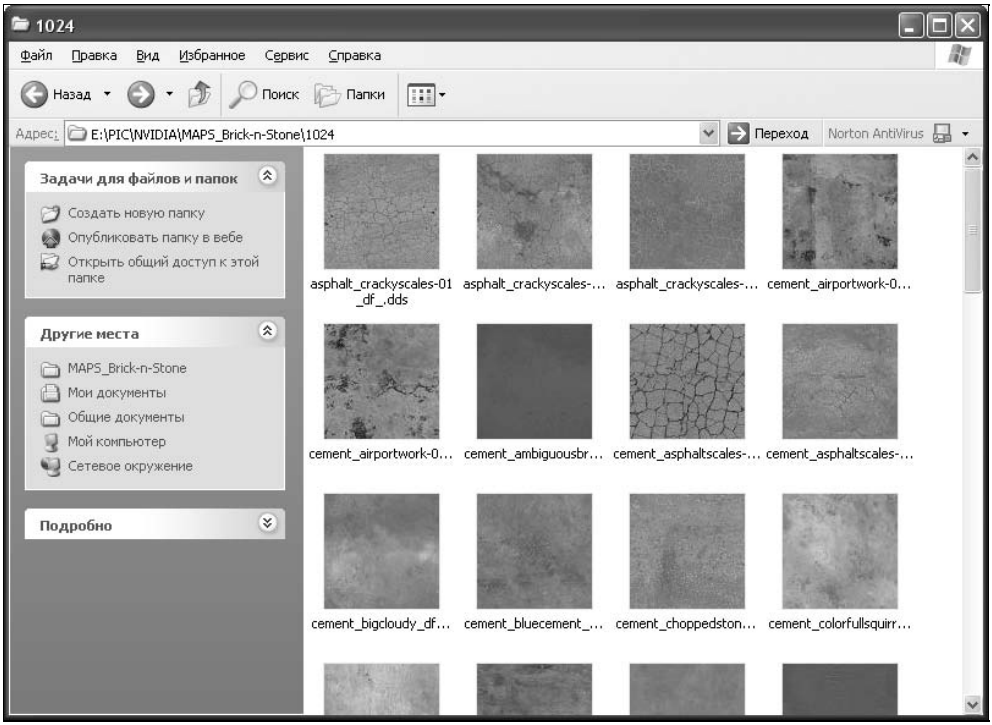


Рис 9.6. Просмотр содержимого файлов формата DDS из проводника

9.4.2. Adobe PhotoShop DXT Compression

Для установки этой утилиты просто скопируйте файл dds.8bi из каталога \DXT_TOOLS_v5.32\Photoshop Plugins\dds.8bi в подкаталог Plug-Ins каталога Photoshop. После этого Photoshop сможет открывать и сохранять файлы формата DDS. При сохранении файла формата DDS на экране будет появляться диалоговое окно (рис. 9.7), в котором пользователь может выбрать формат текстуры (группа **Save Format**), необходимость генерации Mipmap-уровней (группа **MIP maps**) и параметры фильтрации Mipmap-уровней (группа **MIP maps generation**).

Кроме того, данная утилита позволяет визуально оценить потери качества при использовании различных форматов сжатия путем нажатия кнопки **3D Preview**. После нажатия этой кнопки утилита создаст мозаичную текстуру, каждый элемент которой будет содержать уменьшенную копию изображения, сжатого с использованием своего формата (рис. 9.8).

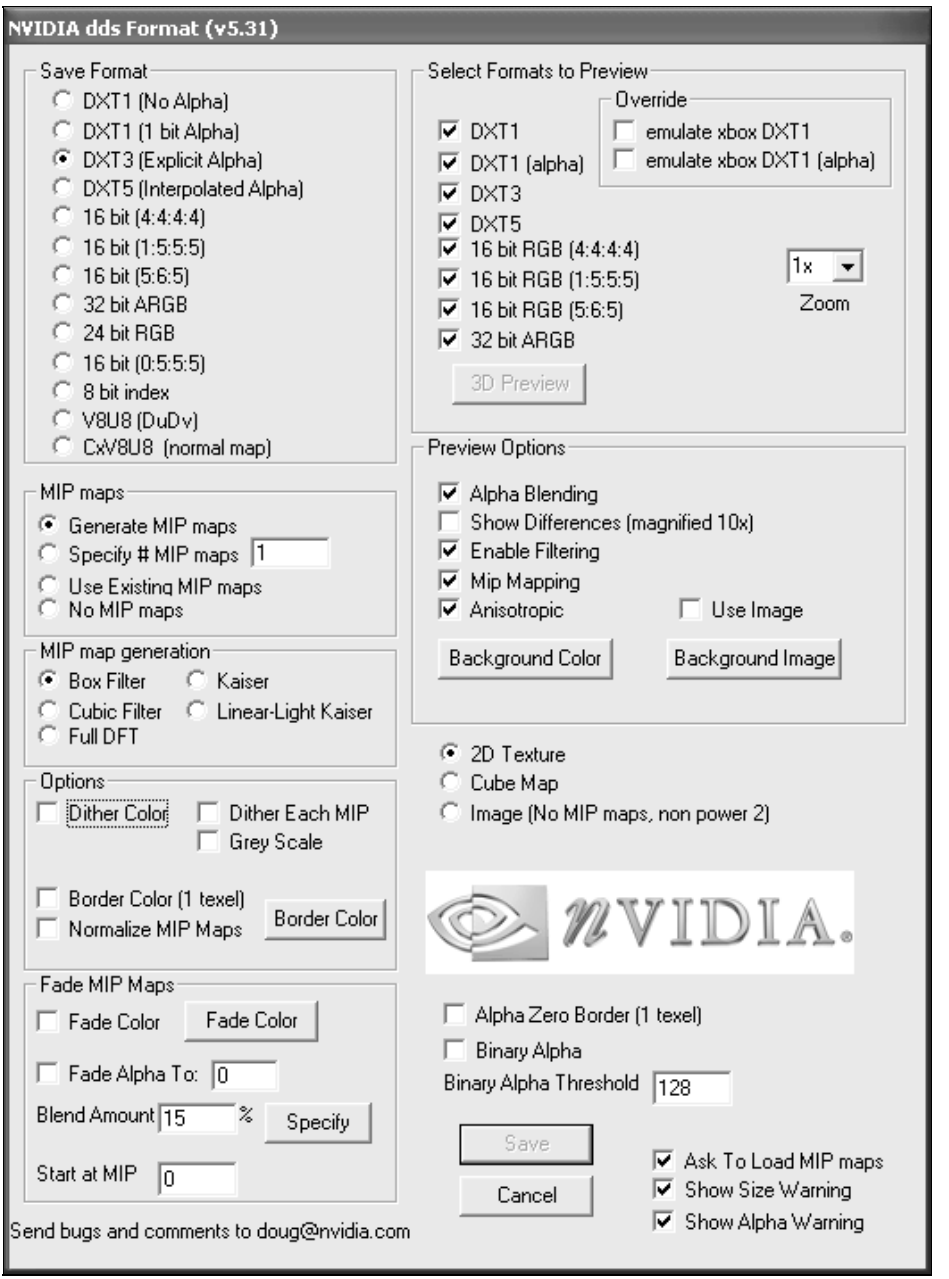


Рис. 9.7. Диалогового окна
Adobe PhotoShop DXT Compression

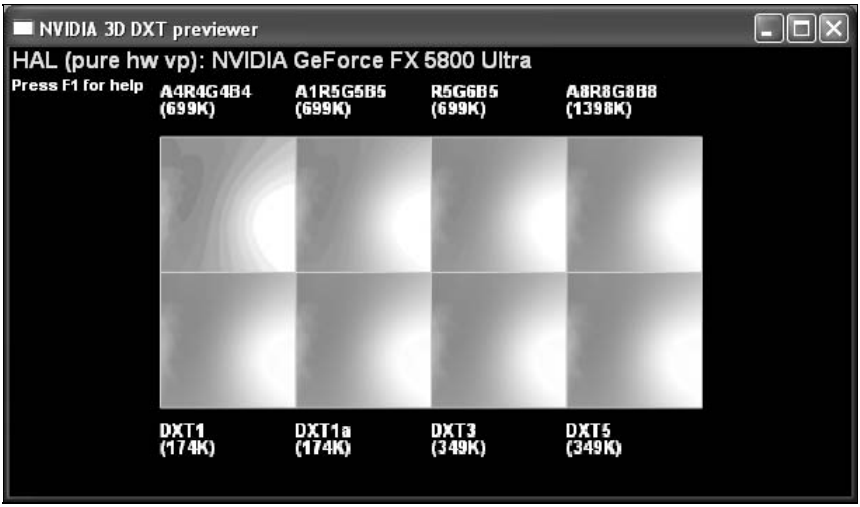


Рис. 9.8. Окно предварительного просмотра Adobe PhotoShop DXT Compression

9.4.3. Утилиты командной строки

Мы рассмотрим две утилиты, работающие из командной строки: `nvDXT` и `readDXT`. Первая утилита используется для преобразования файлов форматов TGA, BMP, GIF, PPM, TIF, CEL в файлы формата DDS. Хотя данная утилита имеет множество параметров, чаще всего используется строка следующего формата:

```
nvdxt.exe -file "имя файла" {-формат текстуры}
```

Список поддерживаемых форматов текстур приведен в табл. 9.8.

Таблица 9.8. *Форматы текстур утилиты* `nvDXt`

Ключ	Формат
`-dxt1c`	DXT1 без альфа-канала
`-dxt1a`	DXT1 с альфа-каналом
`-dxt3`	DXT3
`-dxt5`	DXT5
`-u1555`	Несжатый формат 5 : 5 : 5 : 1 бит на канал (R : G : B : A)
`-u4444`	Несжатый формат 4 : 4 : 4 : 4 бит на канал
`-u565`	Несжатый формат 5 : 6 : 5 бит на канал

Таблица 9.8 *(окончание)*

Ключ	Формат
`-u8888`	Несжатый формат 8 : 8 : 8 : 8 бит на канал
`-u888`	Несжатый формат 8 : 8 : 8 бит на канал
`-u555`	Несжатый формат 5 : 5 : 5 бит на канал

Например, для преобразования файла wood.tga в wood.dds с использованием формата DXT5 нужно воспользоваться командой `nvdxt.exe -file WOOD.TGA -dxt5`.

Другая утилита (`readDXT`), преобразующая файл формата DDS в файл test.tga, может быть полезна для редактирования файлов формата в редакторах, не поддерживающих формат DXT. Синтаксис утилиты `readDXT` очень простой: `readDXT <имя файла>`.

9.4.4. Загрузка сжатых текстур из файлов формата DDS

Как говорилось в начале *разд. 9.4*, в состав NVIDIA DXT Tools входит библиотека DXT Compression/Decompression Library, позволяющая приложению работать с файлами формата DDS. Библиотека DXT Compression/Decompression Library обладает очень широкими возможностями:

- чтение изображений из файлов формата DDS;
- возможность программной распаковки сжатых текстур, позволяющая использовать формат DDS на видеокартах без аппаратной поддержки формата S3TC. Кроме того, программная распаковка текстур, как правило, значительно качественнее аппаратной, хотя и намного медленнее;
- сохранение изображения в файлы формата S3TC и TGA;
- возможность программного сжатия текстур с использованием форматов S3TC, позволяющая сохранять изображения в файлы формата S3TC на видеокартах, без аппаратной поддержки S3TC. Кроме того, программное сжатие текстур значительно качественнее аппаратного, хотя и намного медленнее последнего.

Платой за эту гибкость является относительная сложность использования библиотеки DXT Compression/Decompression Library. При этом все возможности библиотеки нужны лишь разработчикам графических редакторов и программ-конвертеров. Остальным же разработчикам необходима лишь поддержка операции чтения файлов формата DDS.

Поэтому в состав NVIDIA OpenGL SDK входит "облегченная" библиотека для работы с файлами формата DDS — NV_DDS, которая значительно

проще в использовании. Главный недостаток библиотеки NV_DDS заключается в том, что она умеет только загружать текстуры из файлов формата DDS. Но для нас этот недостаток не является существенным — все остальные операции над файлами формата DDS намного проще выполнять средствами утилит пакета DXT Tools.

В этой книге вместо библиотеки NV_DDS из состава NVIDIA OpenGL SDK мы будем использовать новую версию библиотеки NV_DDS, входящую в состав NVIDIA Cg Toolkit. Для подключения к проекту библиотеки NV_DDS необходимо скопировать в каталог с программой файлы nv_dds.h и nv_dds.cpp, которые находятся на CD-диске в каталоге \NVIDIA SDK\ NV_DDS. Затем в список исходных файлов (Source Files) проекта необходимо добавить файл nv_dds.cpp (рис. 9.9), а в текст программы — директиву `#include " nv_dds.h"`.

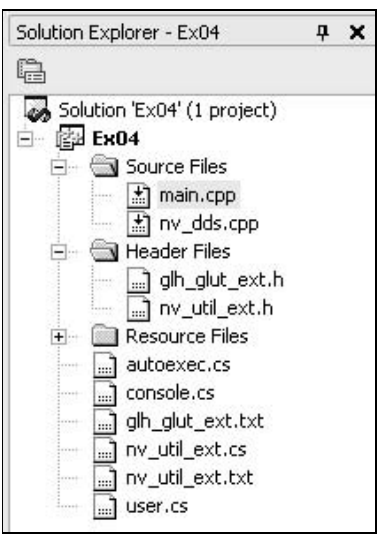


Рис. 9.9. Окно **Solution Explorer** проекта, использующего библиотеку NV_DDS

В основе библиотеки NV_DDS лежит класс `CSurface`, инкапсулирующий понятие "изображение":

```
class CSurface
{
    public:
// Создаем пустое изображение
        CSurface();
```

```
// Создаем пустое изображение с инициализацией некоторых полей класса
        CSurface(int w, int h, int d, int imgsize);
// Конструктор копирования
        CSurface(const CSurface &copy);
// Оператор присваивания (по сути - тот же конструктор копирования)
        CSurface &operator= (const CSurface &rhs);
        virtual ~CSurface();
// Возвращаем указатель на массив пикселов изображения
        operator char*();
// Создаем новое изображение (с выделением памяти для массива пикселов)
        void create(int w, int h, int d, int imgsize);
// Выгружаем изображение из памяти
        void clear();
// Ширина изображения
        int width;
// Высота изображения
        int height;
// Количество байт на пиксел
        int depth;
// Размер текстуры (занимаемая память в байтах)
        int size;

private:
// Указатель на массив пикселей текстуры
        char *pixels;
};
```

Следующий уровень абстракции — это текстура, которая представляет собой совокупность основного изображения (0-й Mipmap-уровень) и массива изображений Mipmap-уровней (формат DDS позволяет хранить в одном файле текстуру со всеми Mipmap-уровнями). Для работы с текстурами в библиотеке NV_DDS используется класс `CTexture`, содержащий внутри себя подобъект класса `CSurface`. При этом, подобъект класса `CSurface` используется для работы с основным изображением, а большинство остальных методов и полей класса `CTexture` — для работы с массивом Mipmap-уровней:

```
class CTexture
{
    public:
// Создаем пустую текстуру
        CTexture();
```

```
// Создаем пустую текстуру с разрешением w × h × d и размером imgSize.
// Mipmap-уровни не создаются
        CTexture(int w, int h, int d, int imgSize);
// Конструктор копирования
        CTexture(const CTexture &copy);
// Оператор присваивания
        CTexture &operator= (const CTexture &rhs);
        ~CTexture();
// Ширина текстуры
        int width;
// Высота текстуры
        int height;
// Количество бит на пиксел
        int depth;
// Размер текстуры
        int size;
// Вектор Mipmap-уровней текстуры
        vector<CSurface> mipmaps;
    private:
// Изображение
        CSurface image;
    };
```

Вершиной иерархии классов библиотеки NV_DDS является класс `CDDSImage`, инкапсулирующий собственно понятие DDS-файла. Так как формат DDS позволяет хранить в одном файле набор из нескольких текстур, имеющих одинаковое разрешение и формат, класс `CDDSImage` хранит не одну текстуру, а целый массив текстур (массив экземпляров класса `CTexture`):

```
class CDDSImage
{
    public:
        CDDSImage();
        ~CDDSImage();

// Возвращает текстуру с индексом index
        CTexture &operator[](int index);
// Загружаем изображение из файла filename. Если flipImage=true, то
// изображение зеркально отражается относительно оси y
        bool load(string filename, bool flipImage = true);
```

```
// Выгружаем текстуру из памяти
        void clear();

// Загружаем одномерную текстуру в видеопамять
        bool upload_texture1D();
// Загружаем двухмерную текстуру в видеопамять
        bool upload_texture2D(int imageIndex = 0, GLenum target =
GL_TEXTURE_2D);
// Загружаем NPOTD-текстуру в видеопамять
        bool upload_textureRectangle();
// Загружаем трехмерную текстуру в видеопамять
        bool upload_texture3D();
// Загружаем кубическую текстуру в видеопамять(будут рассмотрены далее)
        bool upload_textureCubemap();
// Число цветовых компонентов в текстуре
        int components;
// Содержит ли DDS-файл сжатые текстуры?
        bool compressed;
// Содержит ли DDS-файл кубические текстуры? (сжатые текстуры будут
// рассмотрены в главе 9)
        bool cubemap;
// Содержит ли DDS-файл трехмерные текстуры?
        bool volume;
// Формат пикселов текстур, содержащихся в DDS-Файле
        int format;

        private:
// Мы не будем рассматривать раздел private, т. к. он не используется
// большинством пользователей библиотеки NV_DDS
}
```

Для демонстрации использования библиотеки NV_DDS я добавил в пример Ex05 возможность чтения текстур из файлов формата DDS (Ex06). Для этого в обработчик консольной команды `load_texture` был добавлен следующий фрагмент кода (листинг 9.8).

Листинг 9.8

```
bool onConsole(void* Object, string Cmd, vector<string> Params,
glut_console* console)
```

```
{
       if (Cmd=="load_texture")
       {
...
// Если расширение файла - DDS
                     if (Upper(ext)==string("DDS"))
                     {
// Создаем структуру для хранения информации о файле
                           pTextureImage=new tga::tgaImage;
                           pTextureImage->pixels=0;
// Создаем объект для работы с файлом формата DDS
                           CDDSImage image;

// Загружаем изображение из файла
                           if (!image.load(Params[0]))
                           {
                                  console->add("Bad file or file not
found");
                                  return false;
                           }

// Заполняем поля структуры pTextureImage
                           pTextureImage->components=image.components;
                           pTextureImage->format=image.format;
                           pTextureImage->width=image[0].width;
                           pTextureImage->height=image[0].height;
// Загруженное изображение не может быть пережато с использованием
// другого формата
                           bCanBeRecompress=false;
// Загружаем в видеопамять нулевой Mipmap-уровень
                           glCompressedTexImage2DARB(GL_TEXTURE_2D, 0,
image.format, image[0].width, image[0].height, 0, image[0].size, im-
age[0]);
// Загружаем в видеопамять остальные Mipmap-уровни
                           for (unsigned int i = 0; i
< image[0].mipmaps.size(); i++)
                           {
```

```
                                glCompressedTexImage2DARB(GL_TEXTURE_2D,
i+1, image.format, image[0].mipmaps[i].width, image[0].mipmaps[i].height,
0, image[0].mipmaps[i].size, image[0].mipmaps[i]);
                            }
...
}
```

Даже после беглого анализа этого фрагмента исходного кода примера Ex06 становится ясно, что у примера имеется один очень существенный недостаток — он корректно работает только со сжатыми текстурами формата DDS. Для исправления этого недостатка необходимо научить программу отличать сжатые форматы текстур от несжатых. Причем для загрузки сжатых текстур программа должна использовать команду `glCompressedTexImage2DARB`, а для загрузки несжатых текстур — команду `glTexImage2D`.

Такая модификация существенно усложнит программу. К счастью, класс `CDDSImage` имеет метод `upload_texture2D`, выполняющий загрузку двухмерной текстуры формата DDS в видеопамять. Следовательно, мы можем исправить недостаток примера Ex06 простой заменой кода загрузки текстуры в видеопамять вызовом метода `upload_texture2D` (Ex07) (листинг 9.9).

Листинг 9.9

```
                    if (Upper(ext)==string("DDS"))
                    {
                            pTextureImage=new tga::tgaImage;
                            pTextureImage->pixels=0;
                            CDDSImage image;

                            if (!image.load(Params[0]))
                            {
                                    console->add("Bad file or file not
found");
                                    return false;
                            }

                            pTextureImage->components=image.components;
                            pTextureImage->format=image.format;
                            pTextureImage->width=image[0].width;
                            pTextureImage->height=image[0].height;
```

```
                bCanBeRecompress=false;

                image.upload_texture2D();
        }
```

Программа стала значительно проще.

9.4.5. Добавление поддержки текстур файлов DDS в библиотеку ASE Reader

Так как пакет 3D-моделирования 3D Studio MAX 5 имеет встроенную поддержку файлов формата DDS, то создать 3D-модель, использующую текстуры формата DDS, не составит большого труда. А вот с экспортом таких моделей возникнут проблемы — библиотека ASE Reader умеет загружать текстуры только из файлов форматов BMP, TGA и JPG. Единственный способ исправить это недоразумение — "научить" библиотеку ASE Reader использовать формат DDS.

Библиотека ASE Reader не содержит собственных средств для чтения изображений из графических файлов. Вместо этого она использует функцию `read` библиотеки NV_UTIL_EXT. Следовательно, нам необходимо просто добавить в функцию `read` поддержку работы с файлами формата DDS. Но это, к сожалению, практически невозможно. Дело в том, что функция `read` использует для хранения изображения структуру `tgaImage`, в которой не предусмотрено полей для хранения изображений Mipmap-уровней. Для классических графических файловых форматов это ограничение не является актуальным. Большинство форматов, в частности, BMP, TGA и JPG, не поддерживают хранение Mipmap-уровней в одном файле с изображением. Кроме того, генерация Mipmap-уровней для несжатых форматов является тривиальной задачей, которая с легкостью выполняется командой `gluBuild2DMipmaps` или расширением SGIS_generate_mipmap.

А вот с файлами формата DDS ситуация обстоит совершенно по-другому: формат DDS умеет хранить в одном файле все Mipmap-уровни текстуры. Следовательно, если мы хотим воспользоваться структурой `tgaImage`, то нам придется игнорировать Mipmap-уровни. Но поскольку быстрая генерация Mipmap-уровней для сжатого изображения не возможна, нам придется пожертвовать и Mipmap-фильтрацией, что катастрофически ухудшит качество изображения.

Из всего вышесказанного следует, что структура `tgaImage` не подходит для хранения сжатых изображений. Но что же делать? Создавать новую продвинутую структуру, способную хранить Mipmap-уровни, или научить функцию `read` работать со классом `CDDSImage`? Для начала необходимо ответить на вопрос: "Что нам надо?" Нам необходимо загружать изображения из файлов

форматов BMP, TGA, JPG и DDS в *видеопамять*. То есть в нашем случае структуры `tgaImage` и `CDDSImage` представляют собой не что иное, как буфер для хранения промежуточной информации. Следовательно, нам необходима функция, принимающая в качестве параметра имя файла с последующей его загрузкой в видеопамять. При этом функция должна возвращать информацию о параметрах загруженного файла, которая может использоваться программой для оценки качества текстуры, объема занимаемой видеопамяти и т. д. Для хранения этой информации вполне подойдет структура `tgaImage`.

Написать такую функцию не составит труда: фактически функция должна выполнять следующие операции:

1. Определить расширение файла.
2. Если расширение файла dds, то работать с ним при помощи класса `CDDSImage`.
3. Иначе — работать с файлом, используя комбинацию функций `read` и `texImage2D`.

В листинге 9.10 приведен исходный текст функции `ReadAndTexImage2D`, выполняющей эти операции.

Листинг 9.10

```
// Загружаем текстуру из файла в видеопамять
// filename - имя файла
// info - адрес структуры tgaImage, в которую будет занесена информация
// об основных параметрах файла
// Если загрузка прошла успешно, возвращает true, иначе - false
bool ReadAndTexImage2D(string filename, tga::tgaImage* info=0)
{
// Определяем расширение файла
        int pos=filename.find_last_of(".");
        string ext="";
        if ((pos==-1) || (pos==filename.size()-1))
                return 0;
        ext=filename.substr(pos+1, filename.size()-pos);
// Если расширение DDS
        if (Upper(ext)=="DDS")
        {
// Загружаем текстуру из файла
                CDDSImage image;
```

```
            if (!image.load(filename))
                  return false;
// Загружаем текстуру в видеопамять
            image.upload_texture2D();

// Если адрес не равен 0, заполняем структуру tgaImage
            if (info)
            {
                  info->format=image.format;
                  info->components=image.components;
                  info->width=image[0].width;
                  info->height=image[0].height;
                  info->pixels=0;
            }
      }
      else
      {
// Загружаем текстуру из файла
            tga::tgaImage* image=read(filename);
            if (!image)
                  return false;
// Загружаем текстуру в видеопамять
            tex_object_2D tex_2d;
            texImage2D(image, &tex_2d);
// Удаляем массив пикселов (теперь он только занимает лишнее место)
            if (image->pixels)
            {
                  delete[] image->pixels;
                  image->pixels=0;
            }

            if (info)
            {
// Копируем в область памяти, на которую показывает указатель info,
// информацию об изображении
                  *info=*image;
            };
// Освобождаем память
```

```
            clear(image);
        }
        return true;
}
```

Но поскольку библиотека формата ASE Reader умеет читать текстуры из ZIP-архива, функция `ReadAndTexImage2D` также должна уметь это делать. Для этого достаточно написать еще один вариант функции `ReadAndTexImage2D`, которая распаковывает файл из архива во временный каталог Windows, после чего вызывает оригинальную функцию `ReadAndTexImage2D` (листинг 9.11).

Листинг 9.11

```
// Загружаем текстуру из файла в видеопамять
// zip_filename - имя архива
// inzip_filename - имя файла внутри архива
// info - адрес структуры tgaImage, в которую будет занесена информация
// об основных параметрах файла
// Если загрузка прошла успешно, возвращает true, иначе - false
bool ReadAndTexImage2D(string zip_filename, string inzip_filename,
tga::tgaImage* info=0)
{
        int pos=inzip_filename.find_last_of('.');
        if ((pos==-1) || (pos==inzip_filename.size()-1) || (pos==0))
                return false;

        string file_ext=inzip_filename.substr(pos+1, in-
zip_filename.size()-pos-1);

// Распаковываем файл в буфер
        unsigned char* buffer;
        unsigned int buffer_size;
        buffer=unzip::open(zip_filename.c_str(), inzip_filename.c_str(),
&buffer_size);
        if (!buffer)
                return false;
// Получаем имя временного файла
        char szTmpPath[MAX_PATH+1];
        char szTmpFile[MAX_PATH+1];
```

```
        GetTempPath(MAX_PATH, szTmpPath);
        GetTempFileName(szTmpPath, "img", 0, szTmpFile);
// Копируем содержимое буфера во временный файл
        string tmp_filename=szTmpFile;
        tmp_filename=tmp_filename.substr(0, tmp_filename.size()-
3)+file_ext;
        ofstream ofile(tmp_filename.c_str(), ios::binary);
        ofile.write((char*) buffer, buffer_size);
        ofile.close();
        delete[] buffer;
// Вызываем оригинальную функцию ReadAndTexImage2D
        bool result=ReadAndTexImage2D(tmp_filename, info);
// Удаляем временные файлы
        DeleteFile(tmp_filename.c_str());
        DeleteFile(szTmpFile);
// Возвращаем результат оригинальной функции ReadAndTexImage2D
        return result;
}
```

Теперь остается только немного подкорректировать метод `load_texture` класса `CAseTextureManager`, и новая версия библиотеки ASE Reader будет готова (листинг 9.12).

Листинг 9.12

```
CAseTexture* CAseTextureManager::load_texture(string zip_filename, string
inzip_filename)
{
        string short_inzip_filename=inzip_filename;
        int pos=inzip_filename.find_last_of("\\");
        if (pos!=-1)
        {
                if (pos!=short_inzip_filename.size()-1)
                        short_inzip_filename=inzip_filename.substr(pos+1,
inzip_filename.size()-pos-1);
                else
                        return 0;
        }
        string fullname=zip_filename+"::"+short_inzip_filename;
```

```
        for (list<CAseTexture>::iterator itor=textures.begin();
itor!=textures.end(); itor++)
        {
                if (itor->filename==fullname)
                {
                        itor->count++;
                        return &(*itor);
                }
        }


//-------------------------------
// Измененный фрагмент
//-------------------------------
// Создаем новую текстуру
        CAseTexture* pTexture=new CAseTexture;
// Счетчик текстур равен 1
        pTexture->count=1;
// Текстура будет POTD-текстурой
        pTexture->pTextureObj=new tex_object_2D;
// Делаем текстуру активной и задаем параметры фильрации
        pTexture->pTextureObj->bind();
        pTexture->pTextureObj->parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        pTexture->pTextureObj->parameter(GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
// Выделяем место под структуру tgaImage
        tga::tgaImage* image=new tga::tgaImage;
// Загружаем текстуру из памяти
        if (ReadAndTexImage2D(zip_filename, short_inzip_filename, image))
        {
                ReadAndTexImage2D(inzip_filename, image);
                if (!image)
                {
                        delete pTexture;
                        return false;
                }
                else
                        pTexture->filename=inzip_filename;
        }
```

```
        else
                pTexture->filename=fullname;
// Запоминаем ширину и высоту текстуры
        pTexture->width=image->width;
        pTexture->height=image->height;
//-------------------------------
// Конец изменений
//-------------------------------
        textures.push_back(*pTexture);
        return pTexture;
}
```

Для проверки работоспособности библиотеки ASE Reader я перекомпилировал пример Ex10 из *главы 4* (летящий самолет) с использованием новой версии библиотеки (Ex08). Кроме того, текстура поверхности земли grass2.jpg была заменена текстурой AERIAL_Rivers_df.dds из библиотеки текстур NVIDIA, которая находится на CD (рис. 9.10).

Рис. 9.10. Самолет, летящий над местностью, на которую наложена сжатая текстура формата DDS разрешением 1024×1024

Заключение

В этой главе было рассмотрено использование сжатых текстур в OpenGL-программах, позволяющих сократить объем занимаемой видеопамяти, а также немного увеличить быстродействие программы. Расплатой за это является некоторое ухудшение качества изображения.

Сначала мы рассмотрели расширение ARB_texture_compression, являющееся высокоуровневым интерфейсом между прикладной программой и низкоуровневыми расширениями OpenGL. Данный интерфейс реализует работу со сжатыми текстурами. Однако из-за своей "высокоуровневости" расширение ARB_texture_compression предоставляет программисту очень ограниченные средства по управлению качеством сжатия текстур. Поэтому мы изучили низкоуровневое расширение EXT_texture_compression_s3tc, предоставляющее программисту возможность сжатия текстур с использованием формата S3TC. Этот формат поддерживается всеми современными видеокартами.

Кроме того, мы рассмотрели возможность сохранения сжатых текстур на диске, а также их загрузки из файлов формата DDS. И в конце главы мы модифицировали библиотеку ASE Reader, добавив в нее поддержку загрузки текстур из файлов формата DDS.

Глава 10

Кубические текстурные карты

В предыдущих главах мы использовали три вида текстур: одномерные (GL_TEXTURE_1D), двумерные (GL_TEXTURE_2D) и двумерные текстуры размера, не кратного степени 2 (GL_TEXTURE_RECTANGLE_NV). Наложение этих текстур обычно сводится к "натягиванию" плоского изображения на трехмерный объект. В этом случае каждой вершине объекта ставится в соответствие определенный тексель текстуры, т. е. текстура привязывается к вершинам объекта, а при закраске объекта между вершинами объекта производится интерполяция текстурных координат.

Хотя такое простое текстурирование идеально подходит для наложения текстур на трехмерные объекты, оно не является панацеей от всех бед компьютерной графики. Существуют задачи, которые нельзя решить этим методом. Одной из таких задач является наложение окружающей среды (environment mapping) — моделирование зеркальных поверхностей без использования метода трассировки лучей.

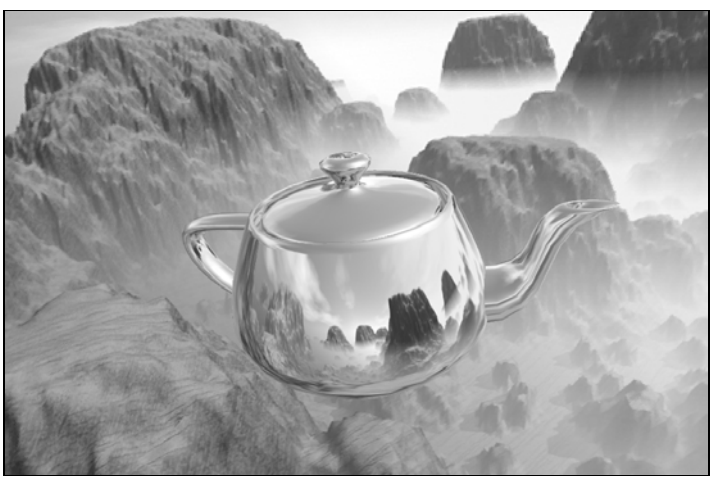


Рис. 10.1. Картина неизвестного художника "чайник в горах"

В качестве примера такой поверхности на рис. 10.1 изображен отполированный до зеркального блеска металлический чайник, летящий над горами и отражающий местность, находящуюся вокруг него.

Плоские зеркальные поверхности можно моделировать с помощью буфера трафарета или обычных двухмерных текстур. Но если объект имеет неплоскую форму, то здесь начинаются проблемы. Дело в том, что отражение нельзя представить в виде плоской статичной текстуры, "привязанной" к вершинам объекта, т. к. вид отражения зависит как от направления взгляда, так и от формы объекта.

10.1. Наложение окружающей среды с использованием сферических карт

В принципе, для моделирования нашего полированного чайника можно использовать сферические текстурные карты, поддерживаемые всеми версиями OpenGL. Как мне кажется, во многих книгах механизм сферических карт описан не достаточно полно, и я собираюсь восполнить этот пробел.

Сферическая карта текстуры представляет собой обычную двухмерную текстуру, в которую занесена вся 360-градусная панорама вокруг объекта. На рис. 10.2 приведена сферическая карта окружающей среды, наложенная на "чайник в торговом зале", изображенный на рис. 10.3.

Рис. 10.2. Сферическая карта текстуры для чайника, показного на рис. 10.3

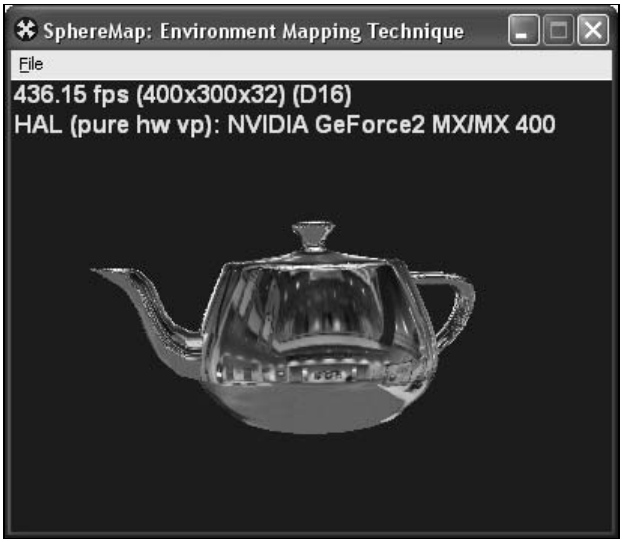


Рис. 10.3. "Чайник в торговом зале".
Скриншот из примера SphereMap, входящего в состав DirectX 8 SDK

Для получения совершенной сферической текстурной карты окружающей среды произвольного объекта необходимо взять большую серебряную сферу и поместить ее в окружающую среду на место объекта. После этого необходимо сфотографировать сферу камерой, расположенной на бесконечном расстоянии от объекта, с объективом, имеющим бесконечное фокусное расстояние.

Но поскольку этот метод очень тяжело реализовать в реальных условиях, на практике карта сферической текстуры создается путем совмещения положения камеры с положением объекта и последующим получением снимков окружающей среды в шести разных направлениях (слева, справа, снизу, сверху, спереди и сзади). Затем полученные изображения накладываются на виртуальную полусферу. Впоследствии эта полусфера заносится в текстуру.

Рассмотрим пример. Пусть у нас имеется зеркальный чайник, вокруг которого летает множество объектов (рис. 10.4).

Для моделирования отражения нам потребуется создать сферическую текстуру. Поместим камеру в центр чайника и сориентируем ее в направлении "вправо", после чего нарисуем всю сцену, но без чайника. Скопируем полученное изображение в текстуру. Повторим эти действия для оставшихся пяти направлений.

Таким образом, мы получим шесть двухмерных текстур. Теперь необходимо нарисовать полусферу, поверхность которой поделена на шесть разных фрагментов, каждому из которых соответствует своя текстура (рис. 10.5).

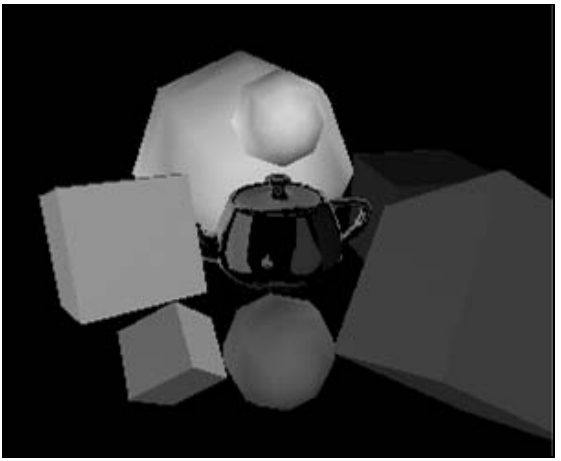


Рис. 10.4. Слайд из доклада [22].
Полированный чайник, в котором отражаются остальные объекты

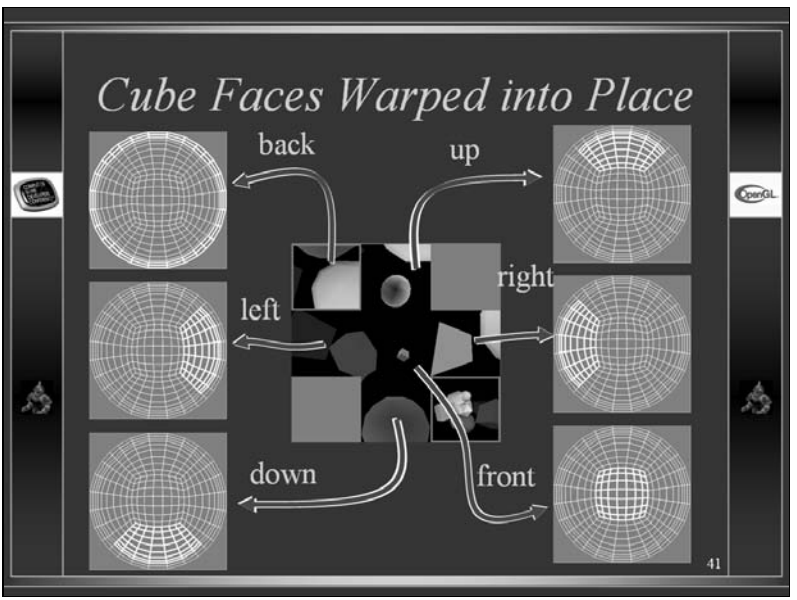


Рис. 10.5. Слайд из доклада [22].
Показывает соответствие между частями виртуальной полусферы и снимками с разных сторон объекта

Остается только скопировать изображение полученной сферы в текстуру, и мы получим готовую сферическую текстурную карту (рис. 10.6).

Рис. 10.6. Слайд из доклада [22]. Полученная сферическая текстура

Для того чтобы наложить текстуру на объект, необходимо включить режим генерации сферических текстурных координат с помощью следующей последовательности команд:

```
glTexGenf(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP)
glTexGenf(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP)
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

В справочной системе OpenGL приводятся выражения, с помощью которых OpenGL выполняет генерацию текстурных координат. Но что стоит за этими формулами?

Предположим, что мы накладываем сферическую текстурную карту окружающей среды на объект, изображенный на рис. 10.7. Для того чтобы узнать, какой фрагмент окружающей среды отражается в точке P на поверхности объекта, необходимо определить два луча. Первый луч располагается от глаза наблюдателя до поверхности объекта (вектор *u*), а второй — от поверхности объекта до фрагмента окружающей среды (вектор *r*).

Отраженный луч, определяемый вектором *r*, продолжается до пересечения с гипотетической поверхностью, которая содержит текстуру. Если предположить, что размеры объекта значительно меньше размеров окружающей его гипотетической сферы с текстурой, то можно считать, что отраженный луч выходит непосредственно из центра сферы. Следовательно, направление вектора *r* может быть непосредственно использовано для указания на тек-

сель текстуры, отражающийся в точке P. Иными словами, вектор *r* однозначно определяет пару текстурных координат *s* и *t*. Для определения этих текстурных координат в OpenGL используется следующее выражение (его вывод находится в [9]):

$$(s,t) = (\frac{1}{2} \cdot (\frac{r_x}{p} + 1), \frac{1}{2} \cdot (\frac{r_y}{p} + 1)) ,$$

где:

$$p = \sqrt{r_x{}^2 + r_y{}^2 + (r_z + 1)^2} .$$

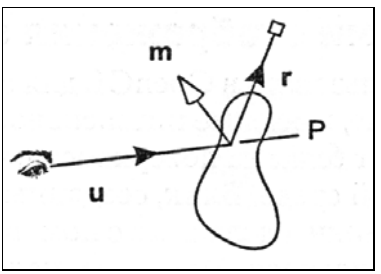


Рис. 10.7. Определение направления отраженного луча[1]

Хотя сферические текстурные карты позволяют моделировать довольно реалистичное отражение, они имеют ряд существенных недостатков.

Во-первых, сферическая текстура зависит от позиции наблюдателя. Следовательно, если наблюдатель будет перемещаться в пространстве, то нам придется динамически изменять сферическую текстуру. А это накладная операция — необходимо визуализировать сцену с шести разных позиций, после чего выполнить еще один проход визуализации для формирования сферической текстуры.

Во-вторых, сферические текстурные карты плохо моделируют плоские поверхности, состоящие из небольшого количества полигонов. Дело в том, что OpenGL вычисляет текстурные координаты только в вершинах объекта, в то время как текстурные координаты в промежуточных точках определяются путем интерполяции. Так как при наложении сферических текстурных карт текстурные координаты (*s*, *t*) вычисляются по очень сложным нелинейным зависимостям, текстурные координаты соседних пикселов объекта связаны между собой нелинейно. Следовательно, для корректного наложения сферической текстурной карты объект должен состоять из достаточно большого количества полигонов. Если же мы, к примеру, попытаемся наложить сфе-

[1] Рисунок взят из [9].

рическую текстурную карту на протяженную плоскость, составленную из двух треугольников, то отражение получится искаженным. Причем, чем больше расстояние между вершинами объекта, тем выше погрешность. Если вы сравните рис. 10.3 и рис. 10.8, то увидите, что отражение на боку чайника является вполне корректным, в то время как на дне чайника отражение просматривается, словно через лупу. Это явление связано с тем, что боковая поверхность чайника состоит из значительно большего количества полигонов, чем его дно.

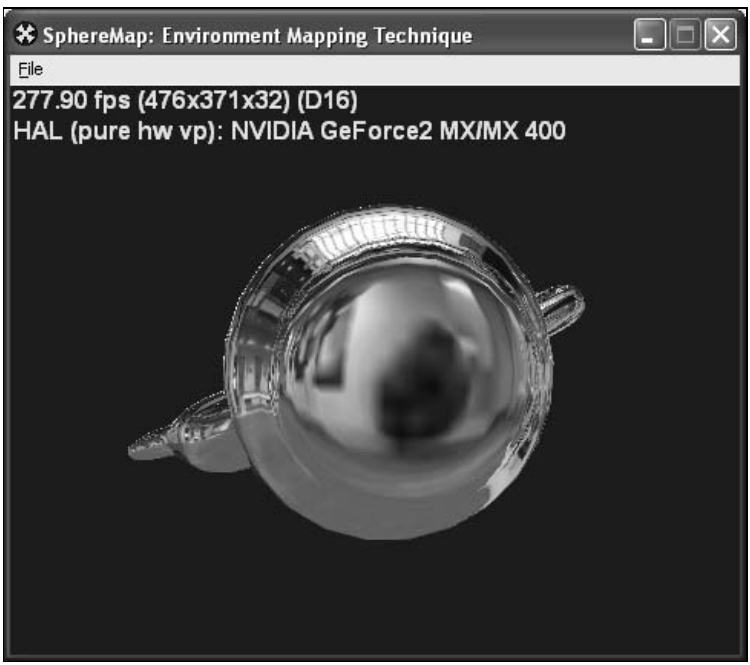


Рис. 10.8. "Чайник в торговом зале".
Скриншот из примера SphereMap, входящего в состав DirectX 8 SDK.
Обратите внимание на искаженное отражение на дне чайника

Третий недостаток сферических текстурных карт выражается в появлении артефактов на краях силуэта объекта, которые выражаются появлением хаотичных мусорных точек, бегущих вдоль краев объекта. Это явление связано с тем, что текстурные координаты на краях сферической карты очень чувствительны к направлению нормали объекта, в результате чего любой небольшой поворот объекта вызывает скачкообразное изменение текстурных координат.

Четвертым недостатком сферических текстурных карт является то, что они используют не весь объем текстуры. Квадратная текстура имеет площадь

$4\times r^2$ пикселов (r — радиус полусферы), а сферическая — $\pi\times r^2$, следовательно, КПД использования текстуры равен $\frac{\pi \cdot r^2}{4 \cdot r^2} = 78.5\%$. Это приводит к неэффективному использованию видеопамяти, которой, как известно, всегда не хватает.

Из всего вышесказанного можно сделать вывод, что сферические текстуры не очень хорошо подходят для моделирования зеркальных объектов. Конечно, их можно использовать для создания эффекта хромированных объектов, в которых отражается только смутное пятно, но если мы хотим создавать реалистичные динамические отражения, то сферические текстурные карты нам не подойдут.

10.2. Наложение окружающей среды с использованием кубических текстурных карт

Давайте еще раз внимательно посмотрим на процесс создания сферической текстурной карты, описанный выше. Представим, что мы находимся в центре зеркального объекта. Для того чтобы получить полную панораму окружающей среды вокруг объекта, нам надо сфотографировать виртуальным фотоаппаратом пространство вокруг него. Если наш фотоаппарат будет обладать углом зрения в 90 градусов, то для покрытия всего пространства вокруг объекта будет достаточно шести фотографий, на которых изображено пространство справа, слева, сверху, снизу, спереди и сзади объекта. Если из этих фотографий сложить куб, то на его внутренних гранях будет изображена 360-градусная панорама вокруг объекта. В качестве примера на рис. 10.9 изображены грани куба, кодирующие пространство вокруг чайника, изображенного на рис. 10.1.

Такое виртуальное фотографирование очень легко реализовать: для этого достаточно создать буфер `pbuffer`, настроить угол обзора, положение и направление камеры при помощи команд `gluPerspective(90, 1, ...)` и `gluLookAt`, после чего нарисовать сцену в буфере `pbuffer` с последующим копированием его содержимого в текстуру.

Эту последовательность действий необходимо повторить для остальных направлений. Далее мы начинаем формировать сферическую текстуру... Стоп! А зачем нам ее формировать? Ведь шести изображений, которые мы только что получили, вполне достаточно для формирования отражения: каждому пикселу на этих изображениях соответствует свой уникальный вектор, направленный из центра зеркального объекта. Этот вектор может выступать в роли вектора отражения. При этом множество всех пикселов шести изобра-

жений покрывает множество всех возможных направлений векторов отражений — все шесть изображений, сложенных вместе, образуют кубическую текстуру.

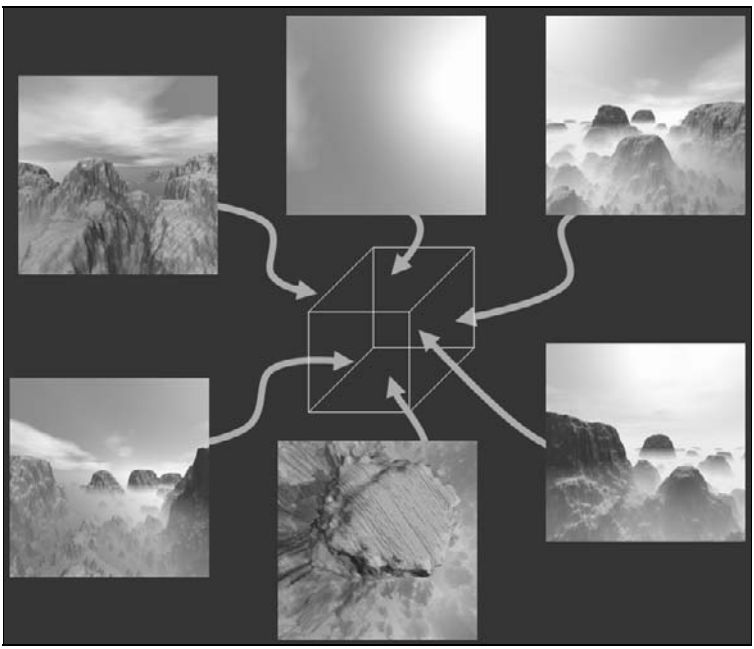


Рис. 10.9. Грани куба, кодирующие 360-градусную панораму вокруг чайника

Для того, чтобы создать отражение с использованием кубической текстуры, ее необходимо наложить на объект с использованием *кубического текстурирования*. Кубическое текстурирование осуществляется путем доступа к текселю текстуры, состоящей из шести квадратных двухмерных текстур, посредством трехмерного вектора направления[2].

Следовательно, нам необходимо сгенерировать текстурные координаты для наложения кубической текстуры. Но стандартный OpenGL умеет одновременно накладывать не более одной текстуры на полигон. Следовательно, нам надо объединить все эти текстуры в одну большую текстуру. А ведь это приведет к падению быстродействия. Кроме того, размеры текстуры в OpenGL должны быть кратны степени 2. Это приведет к тому, что мы сможем задействовать площадь объединенной текстуры только на 75% (рис. 10.10), следовательно, эффективность использования видеопамяти будет даже ниже, чем у сферических текстур. А если мы воспользуемся

[2] Определение взято из [2].

NPOTD-текстурами, то рискуем сильно проиграть в скорости. Вот если бы в OpenGL можно было использовать 6 квадратных текстур как одну.

Рис. 10.10. Один из вариантов размещения кубической текстуры в двухмерной текстуре типа `GL_TEXTURE_2D`

Причина отсутствия в базовой версии OpenGL поддержки кубических текстур связана с тем, что на момент ее появления ни одна серийная рабочая станция не поддерживала их. И только спустя 5 лет после создания OpenGL появились массовые геометрические процессоры, поддерживающие работу с кубическими текстурами. "Первой ласточкой" стал известный GPU, который должен был "изменить мир" в начале сентября 1999 года — GeForce 256. Следствием выхода GeForce 256 явилось создание расширения EXT_texture_cube_map, впоследствии переименованного в ARB_texture_cube_map, позволившее программистам использовать кубические текстуры в своих программах.

10.2.1. Расширение ARB_texture_cube_map

Расширение ARB_texture_cube_map предоставляет программисту новый тип текстуры — `GL_TEXTURE_CUBE_MAP_ARB`, состоящей из шести квадратных текстур. Использование большинства команд OpenGL (`glEnable`, `glDisable`, `glTexParameter`, `glBindTexture`) при работе с кубическими текстурами практически ничем не отличается от использования этих команд с двухмерными текстурами. Единственное отличие заключается в том, что в качестве целевой текстуры указывается `GL_TEXTURE_CUBE_MAP_ARB`.

А вот с командами, осуществляющими доступ к изображению кубических карт (`glTexImage2D`, `glGetTexImage`) все намного сложнее. Дело в том, что эти команды OpenGL могут одновременно работать только с одной двухмерной текстурой, в то время как кубическая текстура состоит из шести двухмерных текстур. Для разрешения этого противоречия расширение ARB_texture_cube_map предоставляет программисту шесть новых целевых текстур:

1. `GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB`;
2. `GL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB`;
3. `GL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB`;

4. GL_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB;
5. GL_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB;
6. GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB.

Дополнительная информация

Константы целевых текстур связаны между собой следующим соотношением:

GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB + 5 = GL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB + 4 = GL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB + 3 = GL_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB + 2 = GL_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB + 1 = GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB.

Это соотношение значительно облегчает написание циклов, перебирающих все грани кубической текстуры.

Каждая из этих целевых текстур соответствует определенной стороне гипотетического куба, центр которого совпадает с центром некоторой системы координат, оси которой параллельны осям глобальной системы координат, а стороны куба выровнены вдоль осей этой системы координат. Таким образом, каждая ось системы координат проходит сквозь середину определенной грани куба. Ось +X проходит сквозь грань, которой соответствует целевая текстура GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB. Ось −X проходит сквозь грань, которой соответствует текстура GL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB. Ось +Y проходит сквозь грань с текстурой GL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB и т. д. Ориентация осей локальной системы координат каждой грани показана на рис. 10.11.

Обратите внимание

Обратите внимание, что начало системы текстурных координат каждой грани кубической текстуры находится в верхнем левом углу. Но мы помним, что в OpenGL начало системы координат обычно находится в нижнем левом углу. Это было сделано для облегчения разработки программ, поддерживающих одновременно API OpenGL и DirectX (в DirectX начало системы координат расположено в верхнем левом углу).

Доступ к кубической текстуре осуществляется с использованием трехмерного вектора направления. К примеру, вектор (3, 1.5, 0.9) соответствует текселю стороны +X с координатами (0.5, 0.3) (рис. 10.12).

В OpenGL вектор направления задается с использованием команд семейства glTexCoord3f, причем текстурные координаты *s*, *t*, *r* соответствуют компонентам x, y и z. При растеризации изображения OpenGL выполняет попиксельную интерполяцию текстурных координат, после чего каждому пикселу объекта ставится в соответствие определенный тексель кубической текстуры. Каким образом OpenGL определяет грань кубической текстуры и двумерные текстурные координаты, которым соответствует данный трехмерный вектор?

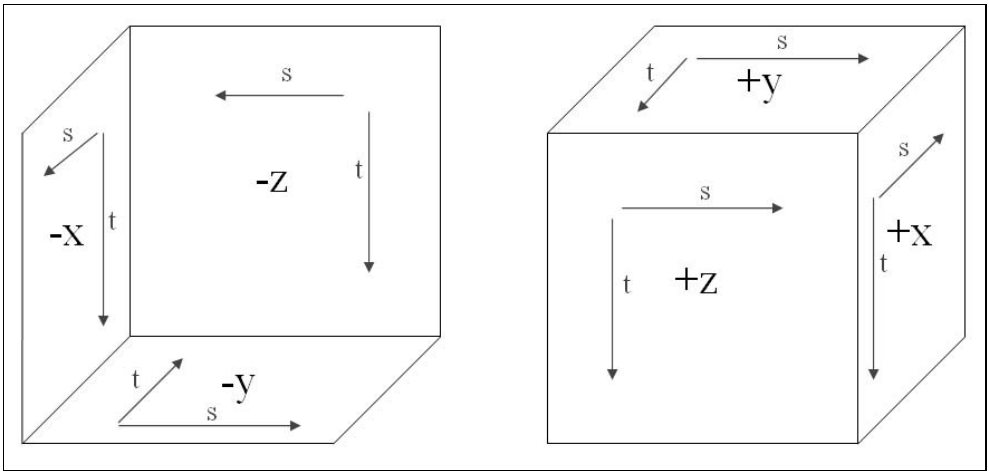


Рис. 10.11. Соответствие идентификаторов сторонам куба (знак "+" означает POSITIVE, знак "−" означает NEGATIVE) и координатные системы текстур для каждой грани[3]

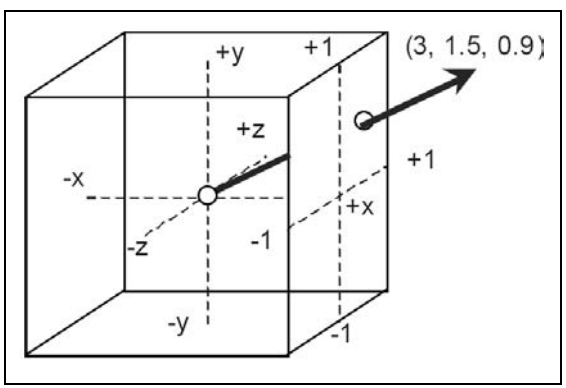


Рис. 10.12. Ненормализованный вектор направления (3, 1.5, 0.9) "протыкает" плоскость +X в точке с координатами (0.5, 0.3). Внимание! Это не текстурные координаты OpenGL. Рисунок взят из статьи [25]

Сначала OpenGL определяет направление координаты, имеющей наибольшее значение. Обычно такое направление называют главным направлением оси (major axis direction). Получив главное направление оси, OpenGL определяет из табл. 10.1 грань кубической текстуры и значения вспомогательных

[3] Рисунок взят из файла \doc\misc\cubemap_image_orientation.ppt из состава NVIDIA OpenGL SDK.

параметров `sc`, `tc` и `ma`, которые используются при вычислении текстурных координат:

$$s = \frac{\frac{sc}{|ma|} + 1}{2};$$

$$t = \frac{\frac{tc}{|ma|} + 1}{2}.$$

Таблица 10.1. *Определение целевой текстуры и параметров* `sc`, `tc` *и* `ma` *на основе главного направления оси*

Главное направление оси	Целевая текстура	sc	tc	ma
+*rx*	GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB	−*rz*	−*ry*	*rx*
−*rx*	GL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB	+*rz*	−*ry*	*rx*
+*ry*	GL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB	+*rx*	+*rz*	*ry*
−*ry*	GL_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB	+*rx*	−*rz*	*ry*
+*rz*	GL_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB	+*rx*	−*ry*	*rz*
−*rz*	GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB	−*rx*	−*ry*	*rz*

В качестве примера мы попробуем определить двухмерные текстурные координаты текселя, которому соответствует трехмерный вектор направления (3, 1.5, 0.9).

После небольшого анализа координат вектора приходим к выводу, что наибольшее значение, равное 3, имеет проекция вектора на ось *x*. Следовательно, главным направлением оси будет +*rx*. Из табл. 10.1 мы определяем, что главному направлению оси +*rx* соответствует целевая текстура `GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB`, а параметры `sc`, `tc` и `ma`, соответственно равны −0.9, −1.5, 3. Остается только подставить эти значения в формулы для вычисления текстурных координат:

$$s = \frac{\frac{-0.9}{|3|} + 1}{2} = 0{,}35;$$

$$t = \frac{\frac{-1.5}{|3|} + 1}{2} = 0{,}25.$$

Итак, вектор (3, 1.5, 0.9) соответствует текселю с текстурными координатами (*s*, *t*), находящемуся на грани целевой текстуры `GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB`.

А теперь попробуйте ответить на вопрос: можно ли программно реализовать расширение ARB_texture_cube_map на видеокартах без поддержки этого расширения путем помещения всех шести граней кубической текстуры в одну текстуру? Ответ — нет. Дело в том, что расширение ARB_texture_cube_map попиксельно преобразует трехмерный вектор в текстурные координаты, в то время как с использованием стандартного OpenGL мы сможем реализовать такое преобразование лишь на уровне вершин. В этом заключается фундаментальное отличие кубических текстурных карт расширения ARB_texture_cube_map от сферических текстурных карт OpenGL.

Хотя расширение позволяет задавать трехмерный вектор направления вручную, в большинстве случаев разумнее всего позволить ему генерировать их автоматически. Расширение ARB_texture_cube_map поддерживает три режима автоматической генерации текстурных координат.

- `GL_REFLECTION_MAP_ARB`. Рассчитывает для каждой вершины вектор отражения на основе вектора взгляда наблюдателя и вектора нормали. На рис. 10.7 это векторы *r*, *u* и *m* соответственно. Значение полученного вектора в системе координат наблюдателя присваивается текстурным координатам (*s*, *t*, *r*). Этот режим генерации текстур обычно используется для наложения кубических карт окружающей среды.
- `GL_NORMAL_MAP_ARB`. Заносит в текстурные координаты (*s*, *t*, *r*) значение вектора нормали.
- `GL_EYE_LINEAR`. Заносит в текстурные координаты (*s*, *t*, *r*) вектор взгляда наблюдателя на вершину в системе координат наблюдателя.

В остальном использование кубических текстур ничем не отличается от стандартных текстур OpenGL.

Теперь мы знаем достаточно информации для написания нашей первой программы, использующей кубические текстуры. Наша программа будет рисовать на экране полированный чайник, в котором отражается горная местность (рис. 10.13) (Ex01).

Для начала нам надо написать код, загружающий кубическую текстуру в память. Один из вариантов — загружать вручную шесть двухмерных текстур, являющихся частями одной большой двухмерной текстуры. Этот метод не является самым лучшим подходом — при изменении названия текстуры нам придется изменить шесть имен файлов. Поэтому мы будем использовать унифицированные имена файлов, в которых хранятся грани кубической текстуры. Название каждого файла, в котором хранится грань кубической текстурной карты, будет иметь следующий формат: "имя кубической тексту-

ры" + "суффикс". При этом "суффикс" может принимать одно из следующих значений: "_posx", "_posy", "_posz", "_negx", "_negy" или "_negz". Я думаю, что комментировать значения суффиксов не имеет смысла. Для примера я приведу имена файлов кубической текстуры, которую использует наш чайник (текстуры взяты из NVIDIA SDK):

CloudyHills_posx.jpg

CloudyHills_negx.jpg

CloudyHills_posy.jpg

CloudyHills_negy.jpg

CloudyHills_posz.jpg

CloudyHills_negz.jpg

Изображения граней данной кубической карты можно увидеть на рис. 10.9.

Рис. 10.13. Полированный чайник.
Горы исчезли, но их отражение осталось

Таким образом, если мы знаем только имя файла без суффикса, то сгенерировать остальные имена файлов не составит труда. Тем не менее генерация имен файлов для граней кубической текстуры — довольно утомительное за-

нятие, особенно если необходимо загрузить несколько кубических текстур. Поэтому мы создадим новую функцию `read`, которая позволяет за один раз загрузить все грани кубической текстуры.

Но сначала нам нужно решить, где мы будем хранить информацию о гранях кубической текстуры. Структура `tga::tgaImage` для этой цели не подходит — она умеет хранить только одно изображение. Следовательно, нам придется создать новую структуру для хранения информации о кубических текстурах. Но существует и более простой, хотя и менее красивый, способ — использовать массив из шести структур `tga::tgaImage`:

```
typedef tga::tgaImage* cubemapImages[6];
```

Функция `read` должна удовлетворять следующим критериям:

- принимать в качестве параметра имя файла;
- возвращать структуру `cubemapImages` с информацией о загруженной кубической карте;
- в случае удачной загрузки функция должна возвратить значение `true`, иначе — `false`;
- если загрузка файла не удалась, то функция должна возвратить имя файла грани текстуры, с которым возникли проблемы, — это облегчит поиск ошибок.

Заголовок функции, удовлетворяющий этим требованиям, приведен ниже:

```
bool read(string filename, cubemapImages& images, string* info=0)
```

Как видно из этого объявления, функция принимает следующие параметры:

- `filename` — имя кубической текстуры;
- `images` — ссылка на массив кубических текстур;
- `info` — указатель на строку, в которую будет занесена информация об ошибках.

Функция возвращает `true` при успешной загрузке текстуры, и `false` — в случае ошибки.

Исходный текст функции с подробными комментариями приведен в листинге 10.1.

Листинг 10.1

```
// Вспомогательная функция, которая освобождает память, занятую
// структурой cubemapImages
void clear(cubemapImages& images)
{
// Перебираем все грани кубической текстуры
```

```
        for (int i=0; i<6; i++)
        {
// Если указатель не равен 0 — освобождаем память
                if (images[i])
                {
                        if (images[i]->pixels)
                        {
                                delete[] images[i]->pixels;
                        }
                        delete images[i];
                        images[i]=0;
                }
        }
}
bool read(string filename, cubemapImages& images, string* info=0)
{
// Если передан указатель на строку, в которую будет заноситься
// информация, — очищаем ее
        if (info)
// Очищаем строку
                info->clear();
// Массив префиксов. Обратите внимание, что порядок префиксов совпадает с
// со списком идентификаторов GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB ...
// GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB, отсортированным по возрастанию
        string file_id[6]={"_posx", "_negx", "_posy", "_negy", "_posz",
"_negz"};

// Разделяем файл на имя и расширение
        size_t pos=filename.find_last_of('.');
        if ((pos==-1) || (pos==filename.size()-1) || (pos==0))
                return false;
// Расширение
        string file_ext=filename.substr(pos+1, filename.size()-pos-1);
// Имя
        string file_base=filename.substr(0, pos);
        string current_filename;

// Освобождаем память (если хранилось старое изображение)
        clear(images);
```

```
// Перебираем все грани
        for (int i=0; i<6; i++)
        {
// Формируем имя файла текущей грани кубической текстуры
                current_filename=file_base+file_id[i]+'.'+file_ext;
// Загружаем грань в память
                images[i]=read(current_filename.c_str());
// Если загрузка грани не удалась
                if (!images[i])
                {
// Заносим в info сообщение об ошибке и выходим из программы
                        if (info)
                                *info=string("Could not load image:
")+current_filename;
                        return false;
                }
        }
        return true;
}
```

Помимо всего прочего, нам очень не помешала бы вспомогательная функция, загружающая кубическую карту, хранящуюся в типе-массиве `cubemapImages`, в видеопамять. Эта функция должна принимать массив с информацией о гранях кубической текстуры и флаг, указывающий на необходимость генерации Mipmap-уровней. В случае удачной загрузки кубических текстур в видеопамять функция должна возвращать `true`, иначе — `false`. Исходный текст такой функции (`texImageCubemap`), удовлетворяющей всем этим критериям, приведен в листинге 10.2.

Листинг 10.2

```
bool texImageCubemap(cubemapImages& images, bool bMipmap=true)
{
// Перебираем все грани кубической карты
        for (int i=0; i<6; i++)
        {
// Если грань существует
                if (images[i] && (images[i]->pixels))
                {
```

```
// Если необходимо генерировать Mipmap-уровни
                    if (bMipmap)
// Генерируем Mipmap-уровни
       gluBuild2DMipmaps(GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB+i,
images[i]->components, images[i]->width,
                                                    images[i]->height,
images[i]->format, GL_UNSIGNED_BYTE,
                                                    images[i]->pixels);
                    else
// Загружаем текстуры без генерации Mipmap-уровней
              glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB+i, 0,
images[i]->components, images[i]->width,
                                                  images[i]->height, 0,
images[i]->format,
                                                  GL_UNSIGNED_BYTE,
images[i]->pixels);
              }
              else
// Иначе - ошибка
                    return false;
       }


// Фиксируем координаты s, t в диапазоне [1/(2*N)..2-1/(2*N)], где
// N - размер текстуры в направлении фиксации. Если этого не сделать, то
// на стыках граней кубической текстурной карты появятся артефакты
       glTexParameterf(GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
       glTexParameterf(GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);


       return true;
}
```

Обратите внимание, что мы задаем параметры оболочки (wrap parameters) только для текстурных координат *s* и *t*, в то время как текстурные координаты для кубических текстурных карт задаются при помощи трех координат *s*, *t* и *r*. Это связано с тем, что параметры оболочки применяются к кубическим текстурным координатам только после их преобразования в двухмерные текстурные координаты *s*, *t*. То есть задание параметров оболочки для текстурной координаты *r* будет избыточным. Тем не менее большинство разработчиков предпочитают "перестраховаться", задавая, на всякий случай, параметры оболочки и для координаты *r*.

Теперь мы можем переходить к написанию кода, осуществляющего загрузку кубической карты из файла с последующим наложением на чайник (листинг 10.3).

Листинг 10.3

```
...
// Дисплейный список для чайника
display_list list0;
// Массив указателей на изображения граней кубической текстуры
cubemapImages images={0, 0, 0, 0, 0, 0};
// Идентификатор кубической текстурной карты
GLuint texture;

// Инициализация OpenGL
bool Init()
{
// Проверяем поддержку необходимых расширений
       if (!glh_init_extensions("GL_ARB_texture_cube_map"))
       {
              console.add(string("Unsupported extensions:
")+glh_get_unsupported_extensions());
              console.exit(-1);
              return false;
       }

// Настраиваем параметры OpenGL
       glEnable(GL_DEPTH_TEST);
       glEnable(GL_LIGHTING);
       glEnable(GL_LIGHT0);

// Загружаем кубическую текстурную карту
       string info;
       if (!read("..\\CloudyHills.jpg", images, &info))
       {
// Если произошла ошибка, выводим сообщение в консоль и завершаем
// программу
              console.add(info);
              console.exit(-1);
```

```
            return false;
        }

// Получаем свободный идентификатор текстуры
        glGenTextures(1, &texture);
// Делаем полученную текстуру текущей
        glBindTexture(GL_TEXTURE_CUBE_MAP_ARB, texture);

// Загружаем кубическую карту в видеопамять
        texImageCubemap(images);

// Выгружаем кубическую карту из основной памяти (она нам больше не
// нужна)
        clear(images);

// Задаем режим наложения текстуры и ее фильтрации
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
        glTexParameteri(GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        glTexParameteri(GL_TEXTURE_CUBE_MAP_ARB, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);

// Создаем дисплейный список
        list0.new_list(GL_COMPILE);
// Устанавливаем занесение в текстурные координаты вектора отражения
            glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,
GL_REFLECTION_MAP_ARB);
            glTexGeni(GL_T, GL_TEXTURE_GEN_MODE,
GL_REFLECTION_MAP_ARB);
            glTexGeni(GL_R, GL_TEXTURE_GEN_MODE,
GL_REFLECTION_MAP_ARB);

// Включаем автогенерацию текстурных координат
            glEnable(GL_TEXTURE_GEN_S);
            glEnable(GL_TEXTURE_GEN_T);
            glEnable(GL_TEXTURE_GEN_R);

// Включаем наложение кубических текстурных карт
            glEnable(GL_TEXTURE_CUBE_MAP_ARB);
```

```
// Рисуем чайник
              glutSolidTeapot(2);

// Восстанавливаем настройки OpenGL
              glDisable(GL_TEXTURE_CUBE_MAP_ARB);

              glDisable(GL_TEXTURE_GEN_S);
              glDisable(GL_TEXTURE_GEN_T);
              glDisable(GL_TEXTURE_GEN_R);
       list0.end_list();

       glClearColor(0.5, 0.5, 0.75, 1);

       return true;
}

// Обновляем содержимое экрана
void Display()
{
// Очищаем экран
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Активизируем кубическую карту
       glBindTexture(GL_TEXTURE_CUBE_MAP_ARB, texture);
// Рисуем сцену (чайник)
       list0.call_list();
}
...
```

В начале этого раздела уже говорилось о том, что использование большинства команд OpenGL (`glEnable`, `glDisable`, `glTexParameter`, `glBindTexture`) с кубическими текстурами аналогично их использованию с одномерными или двухмерными текстурами. Следовательно, на основе класса `tex_object` можно легко создать новый класс, инкапсулирующий работу с кубическими текстурами. В составе библиотеки OpenGL Helper Library (GLH) уже имеется такой класс:

```
class tex_object_cube_map : public tex_object
{ public: tex_object_cube_map() : tex_object(GL_TEXTURE_CUBE_MAP_ARB) {}
};
```

Для демонстрации использования класса `tex_object_cube_map` я переписал пример Ex01 с использованием этого класса (Ex02). Мы не будем рассматривать этот пример, т. к. изменения в примере Ex02 являются чисто косметическими.

10.2.2. Загрузка кубических текстур из файлов формата DDS

В предыдущем разделе для хранения кубических текстурных карт на диске мы использовали обычные форматы графических файлов (JPG, TGA, BMP и т. д.). Каждая грань кубической текстуры хранилась в отдельном файле, в результате чего для хранения одной кубической текстуры использовались шесть файлов.

Но хранить одну кубическую карту в нескольких разных файлах не совсем удобно. Гораздо рациональнее хранить все грани кубической текстуры в одном файле. К счастью, существует формат DDS, позволяющий хранить в одном файле несколько изображений с Mipmap-уровнями. Так как в *главе 9*, мы уже рассматривали использование формата DDS для хранения сжатых текстур, мы можем сразу перейти к практическому использованию формата DDS для хранения кубических текстурных карт. В этом разделе мы модифицируем пример Ex02, добавив в него возможность загружать кубические текстурные карты из файлов формата DDS.

Для начала мы должны объединить все грани кубической текстуры из примера Ex02 в один файл (CloudyHills.dds). Для этой цели мы воспользуемся утилитой командной строки `nvdxt` из состава DXT TOOLS 5.32:

```
nvdxt -cubemap<имя создаваемого DDS-файла> -list<список файлов с гранями кубической текстуры> -<формат текстуры>
```

Итак, сначала нам необходимо создать текстовый файл (CloudyHills.lst) со списком файлов, в которых хранятся грани кубической текстуры. Утилита `nvdxt` предполагает, что грани перечисляются в следующем порядке: +X, −X, +Y, −Y, +Z, −Z. В нашем случае файл CloudyHills.lst будет содержать следующие строки:

```
CloudyHills_posx.jpg
CloudyHills_negx.jpg
CloudyHills_posy.jpg
CloudyHills_negy.jpg
CloudyHills_posz.jpg
CloudyHills_negz.jpg
```

Затем выполняем из командной строки следующую команду

```
nvdxt -cubemap CloudyHills.dds -list CloudyHills.lst -u888
```

и получаем сообщение: "CloudyHills_posx.jpg, Can't open input file 'CloudyHills_posx.jpg'[GBM failed to read file]". Это сообщение означает, что утилита nvdxt почему-то не смогла открыть файл CloudyHills_posx.jpg. Выход из данной ситуации — преобразовать файлы с гранями кубической карты из формата JPG в другой графический формат, поддерживаемый утилитой nvdxt, например, TGA (эту операцию можно выполнить при помощи ACDSee или Photoshop). После этого корректируем файл CloudyHills.lst и еще раз пытаемся сгенерировать файл CloudyHills.dds. На этот раз генерация файла проходит без приключений, и мы, наконец, получаем готовый файл формата DDS, содержащий кубическую карту окружающей среды.

Теперь нам необходимо обучить нашу программу загружать кубические карты из файлов формата DDS в видеопамять. Для этого воспользуемся библиотекой NV_DDS, рассмотренной в *главе 8*. К счастью, работа с кубическими текстурами средствами этой библиотеки практически ничем не отличается от работы с обычными двухмерными текстурами: главное отличие заключается в том, что для загрузки текстур в видеопамять используется метод upload_textureCubemap класса CDDSImage (вместо метода upload_texture2D). Исходный текст этого метода с подробными комментариями приведен в листинге 10.4.

Листинг 10.4

```
// Загружаем кубическую текстуру в видеопамять
bool CDDSImage::upload_textureCubemap()
{
// Проверка текстуры на корректность
    assert(valid);
    assert(cubemap);
    assert(images.size() == 6);

// Текущая целевая текстура (т. е. текущая грань)
    GLenum target;

// Перебираем все грани кубической текстуры
    for (int n = 0; n < 6; n++)
    {
// Определяем целевую текстуру для текущей грани
        target = GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB+n;
// Загружаем ее в видеопамять
        if (!upload_texture2D(n, target))
```

```
            return false;
    }

    return true;
}
```

Итак, нам остается только немного модифицировать фрагменты кода примера Ex02, отвечающие за загрузку кубических текстур (листинг 10.5).

Листинг 10.5

```
...
// Объект
CDDSImage image;

bool Init()
{
...
        if (!image.load("..\\CloudyHills.dds"))
        {
                console.add("CloudyHills.dds: Bad file or file not
found");
                console.exit(-1);
                return false;
        }

        reflect_texture.bind();

        if (!image.upload_textureCubemap())
        {
                console.add("CloudyHills.dds: Can't load texture to
VRAM");
                console.exit(-1);
                return false;
        };

// Задаем режим наложения текстуры
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
// Настраиваем параметры фильтрации
        reflect_texture.parameter(GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
```

```
        reflect_texture.parameter(GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        reflect_texture.parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        reflect_texture.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
...
```

Запускаем полученную программу на выполнение и получаем какое-то, мягко говоря, странное отражение (рис 10.14). После внимательного изучения отражения обнаруживается, что у кубической текстурной карты почему-то переставлены местами нулевые Mipmap-уровни граней +Y и −Y. Но наша программа не выполняет никаких действий с гранями кубической текстуры. Следовательно, ошибка спряталась где-то внутри библиотеки NV_DDS.

Рис. 10.14. Некорректное отражение в чайнике из-за ошибки в библиотеке NV_DDS

Но как ее найти? Ведь исходный код библиотеки NV_DDS занимает более 1000 строк! Давайте внимательно изучим ситуацию. В нашей программе используются только два метода класса `CDDSImage`: `load` и `upload_textureCubemap`. Чуть выше мы уже внимательно изучали исходный текст метода `upload_textureCubemap` и не нашли в нем ничего подозрительного. Следовательно, ошибка, скорее всего, затаилась где-то внутри метода

`load`. Но где? Метод `load` занимает более 150 строк, поэтому внимательное изучение его исходного кода — задача не из приятных. Но в этом и нет необходимости: ошибка заключается в том, что метод `load`, по-видимому, некорректно обрабатывает две грани (+Y и −Y) кубической текстуры. Следовательно, нам необходимо проанализировать код функции на наличие строк `GL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB` и `GL_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB`, или найти упоминания цифр 2 и 3. Потратив некоторое время на блуждание по тексту функции, обнаруживаем следующий "подозрительный" фрагмент кода, расположенный в конце функции `load`:

```
if (cubemap && (flipImage))
{
        CTexture tmp;
        tmp = images[3];
        images[3] = images[2];
        images[2] = tmp;
}
```

Как видно, этот код меняет местами грани +Y и −Y кубической текстуры. Причем, т. к. оператор "=" класса `CTexture` не выполняет копирования Mipmap-уровней, этот код, в действительности, переставляет местами только нулевые Mipmap-уровни. Похоже, это именно то, что мы ищем. После удаления (или комментирования) этого фрагмента кода программа начинает работать нормально.

Возможно, у вас возник вопрос: откуда этот фрагмент кода оказался внутри библиотеки NV_DDS? Если честно — не знаю. По-видимому, это просто "рудиментарный" фрагмент кода, который создатели библиотеки NV_DDS просто забыли удалить из нее после очередной модификации.

В заключение раздела давайте попробуем ответить на такой вопрос — стоит ли использовать формат DDS для хранения кубических текстур? Итак, формат DDS имеет следующие плюсы:

- все грани кубической текстуры хранятся внутри одного файла;
- формат DDS поддерживает сжатие текстур по методу S3TC.

Но, как всегда, есть и минусы:

- несжатые файлы формата DDS имеют очень большой объем (на уровне файлов формата BMP);
- формат поддерживает только сжатие по методу S3TC, который очень сильно уступает формату JPG по степени сжатия, а формату TGA — по качеству изображения;
- кубические карты формата DDS очень неудобно редактировать.

Поэтому в примерах этой книги мы будем хранить кубические текстуры в файлах форматов JPG и TGA.

10.2.3. Моделирование отражения с использованием статических кубических текстурных карт

В предыдущих разделах мы создали несколько программ, выполняющих наложение окружающей среды с использованием кубических текстур. Правда, у всех этих программ есть общий недостаток. Несмотря на то, что в отражении на глянцевом чайнике видна горная местность, сам чайник находится на простом синем фоне (рис. 10.13). В реальной жизни такое невозможно — в зеркальном объекте всегда видно отражение *окружающей среды*. Следовательно, для того, чтобы изображение на рис. 10.13 стало корректным, должно выполнятся одно из двух условий:

- в чайнике должен отражаться синий фон;
- чайник должен парить над горной местностью.

Моделирование отражения синего фона в чайнике — не очень интересная задача, поэтому мы попытаемся воссоздать горную местность, отраженную в чайнике.

На первый взгляд эта задача кажется неразрешимой. Но давайте вспомним, каким образом расширение ARB_texture_cube_map формирует отражение на поверхности чайника. Фактически OpenGL помещает каждую вершину объекта в центр виртуального куба, после чего определяет на основе векторов направления взгляда наблюдателя (вектор u) и нормали (вектор m) направление вектора отражения (вектор r). Следовательно, для того чтобы отражение в чайнике согласовывалось с окружающей средой, мы можем изобразить окружающую среду с использованием куба, на грани которого наложены изображения граней кубической текстуры. При этом должны выполняться два условия:

- центр куба должен совпадать с положением центра объекта, иначе изображение не будет физически корректным;
- куб должен иметь достаточно большой размер. Чем больше размер куба, тем меньше нестыковка между отражением и окружающей средой.

Итак, нам необходимо нарисовать большой куб, на грани которого наложены изображения с граней кубической текстуры. Вроде бы все просто — на основе рис. 10.11 можно рассчитать текстурные координаты граней куба и наложить на них текстуры. Но если вы раньше никогда не работали с кубическими текстурами, то вы, скорее всего, запутаетесь в текстурных координатах для сторон куба. Поэтому я советую создать кубическую карту, имеющую грани различных цветов с цветными пометками на краях, и отлаживать

приложение с использованием этой кубической карты. Например, во время отладки этого примера я использовал кубическую карту отражения, показанную на рис. 10.15.

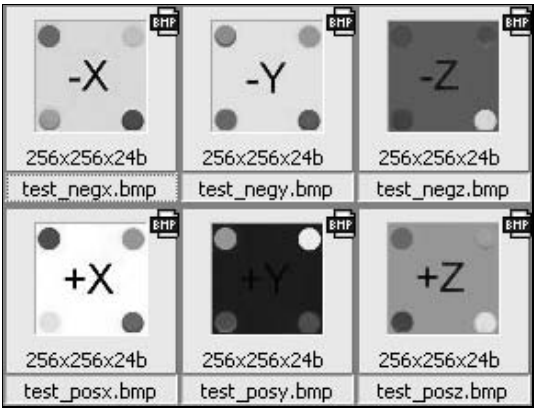


Рис. 10.15. Грани тестовой кубической текстуры

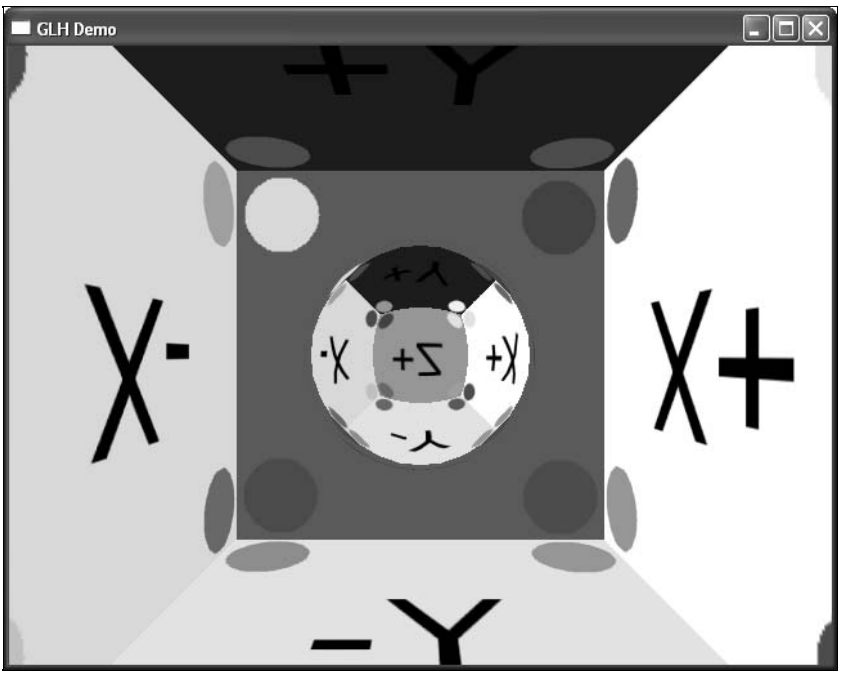


Рис. 10.16. Тестовый пример для проверки на корректность соотношения между окружающей средой и ее отражением в объекте

Кроме того, для удобства сравнения отражения окружающей среды в объекте с реальной окружающей средой во время отладки приложения я использовал в качестве зеркального объекта шар, а размеры куба были немного уменьшены (рис. 10.16).

В листинге 10.6 приведен исходный код примера Ex04, моделирующий окружающую среду зеркального объекта с использованием куба.

Листинг 10.6

```
// Дисплейный список, выводящий на экран глянцевый объект
display_list list0;
// Грани кубической карты
cubemapImages images={0, 0, 0, 0, 0, 0};
// Кубическая текстура отражения
tex_object_cube_map reflect_texture;
// Текстуры граней куба
tex_object_2D walls_textures[6];
// Дисплейный список, рисующий куб, моделирующий окружающую среду
display_list walls;
// Размер куба окружающей среды. Во время отладки равен 5, в финальной
// версии — 50
const float offset=5;
// Инициализация программы
bool Init()
{
// Инициализируем расширение
        if (!glh_init_extensions("GL_ARB_texture_cube_map"))
        {
                console.add(string("Unsupported extensions:
")+glh_get_unsupported_extensions());
                console.exit(-1);
                return false;
        }
// Настраиваем параметры OpenGL
        glEnable(GL_DEPTH_TEST);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
// Загружаем кубическую карту
        string info;
```

```
        if (!read("..\\test.bmp", images, &info))
        {
                console.add(info);
                console.exit(-1);
                return false;
        }
// Загружаем кубическую текстуру в видеопамять
        reflect_texture.bind();

        texImageCubemap(images);
// Настраиваем параметры текстуры
        reflect_texture.parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        reflect_texture.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Создаем дисплейный список рисования зеркального объекта
        list0.new_list(GL_COMPILE);
                glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,
GL_REFLECTION_MAP_ARB);
                glTexGeni(GL_T, GL_TEXTURE_GEN_MODE,
GL_REFLECTION_MAP_ARB);
                glTexGeni(GL_R, GL_TEXTURE_GEN_MODE,
GL_REFLECTION_MAP_ARB);

                glEnable(GL_TEXTURE_GEN_S);
                glEnable(GL_TEXTURE_GEN_T);
                glEnable(GL_TEXTURE_GEN_R);

                reflect_texture.bind();
                reflect_texture.enable();

                glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_REPLACE);

                glutSolidTeapot(2);
// В отладочном режиме рисуем сферу
                glutSolidSphere(2, 32, 32);
                reflect_texture.disable();
                glDisable(GL_TEXTURE_GEN_S);
                glDisable(GL_TEXTURE_GEN_T);
```

```
                glDisable(GL_TEXTURE_GEN_R);
        list0.end_list();
// Создаем двумерные текстуры для 6-ти граней куба
        for (int i=0; i<6; i++)
        {
                walls_textures[i].bind();


                gluBuild2DMipmaps(walls_textures[i].target, images[i]->
components, images[i]->width,
                                                images[i]->height,
images[i]->format, GL_UNSIGNED_BYTE,
                                                images[i]->pixels);
                walls_textures[i].parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
                walls_textures[i].parameter(GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
                walls_textures[i].parameter(GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
                walls_textures[i].parameter(GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
        }
// Создаем дисплейный список, выводящий на экран куб, моделирующий
// окружающую среду
        walls.new_list(GL_COMPILE);
// Включаем отсечение невидимых граней и выключаем расчет освещения
                glEnable(GL_CULL_FACE);
                glCullFace(GL_BACK);
                glDisable(GL_LIGHTING);
// Задаем режим наложения текстур
                glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_REPLACE);
// Рисуем грань куба, на которую наложена текстура с грани -Z кубической
// текстуры
                walls_textures[5].bind();
                walls_textures[5].enable();
                glBegin(GL_QUADS);
                        glTexCoord2f(1, 1);
                        glVertex3f(-offset, -offset, -offset);

                        glTexCoord2f(0, 1);
                        glVertex3f(offset, -offset, -offset);
```

```
                glTexCoord2f(0, 0);
                glVertex3f(offset, offset, -offset);

                glTexCoord2f(1, 0);
                glVertex3f(-offset, offset, -offset);
        glEnd();
        walls_textures[5].disable();
// Рисуем грань куба, на которую наложена текстура с грани +Z кубической
// текстуры
        walls_textures[4].bind();
        walls_textures[4].enable();
        glBegin(GL_QUADS);
                glTexCoord2f(0, 0);
                glVertex3f(-offset, offset, offset);

                glTexCoord2f(1, 0);
                glVertex3f(offset, offset, offset);

                glTexCoord2f(1, 1);
                glVertex3f(offset, -offset, offset);

                glTexCoord2f(0, 1);
                glVertex3f(-offset, -offset, offset);
        glEnd();
        walls_textures[4].disable();
// Рисуем грань куба, на которую наложена текстура с грани -Y кубической
// текстуры
        walls_textures[3].bind();
        walls_textures[3].enable();
        glBegin(GL_QUADS);
                glTexCoord2f(0, 1);
                glVertex3f(-offset, -offset, -offset);

                glTexCoord2f(0, 0);
                glVertex3f(-offset, -offset, offset);

                glTexCoord2f(1, 0);;
                glVertex3f(offset, -offset, offset);
```

```
                glTexCoord2f(1, 1);
                glVertex3f(offset, -offset, -offset);
        glEnd();
        walls_textures[3].disable();
// Рисуем грань куба, на которую наложена текстура с грани +Y кубической
// текстуры
        walls_textures[2].bind();
        walls_textures[2].enable();
        glBegin(GL_QUADS);
                glTexCoord2f(1, 0);
                glVertex3f(offset, offset, -offset);

                glTexCoord2f(1, 1);;
                glVertex3f(offset, offset, offset);

                glTexCoord2f(0, 1);
                glVertex3f(-offset, offset, offset);

                glTexCoord2f(0, 0);
                glVertex3f(-offset, offset, -offset);
        glEnd();
        walls_textures[2].disable();
// Рисуем грань куба, на которую наложена текстура с грани -X кубической
// текстуры
        walls_textures[1].bind();
        walls_textures[1].enable();
        glBegin(GL_QUADS);
                glTexCoord2f(0, 0);
                glVertex3f(-offset, offset, -offset);

                glTexCoord2f(1, 0);
                glVertex3f(-offset, offset, offset);

                glTexCoord2f(1, 1);
                glVertex3f(-offset, -offset, offset);

                glTexCoord2f(0, 1);
                glVertex3f(-offset, -offset, -offset);
        glEnd();
```

```
            walls_textures[1].disable();
// Рисуем грань куба, на которую наложена текстура с грани +X кубической
// текстуры
            walls_textures[0].bind();
            walls_textures[0].enable();
            glBegin(GL_QUADS);
                  glTexCoord2f(1, 1);
                  glVertex3f(offset, -offset, -offset);

                  glTexCoord2f(0, 1);
                  glVertex3f(offset, -offset, offset);
                  glTexCoord2f(0, 0);
                  glVertex3f(offset, offset, offset);
                  glTexCoord2f(1, 0);
                  glVertex3f(offset, offset, -offset);
            glEnd();
            walls_textures[0].disable();
            glDisable(GL_CULL_FACE);
            glEnable(GL_LIGHTING);
      walls.end_list();
// Выгружаем изображения граней кубической текстуры из памяти
      clear(images);
      glClearColor(0.5, 0.5, 0.75, 1);
      return true;
}
void Display()
{
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Рисуем куб с окружающей средой
      walls.call_list();
// Рисуем зеркальный объект
      list0.call_list();
}
```

Поработав некоторое время с примером Ex04, вы очень скоро обнаружите у него один существенный недостаток: отражение не зависит от положения наблюдателя, в результате чего стоит только наблюдателю немного изменить свою позицию в пространстве, как отражение тут же перестает быть физически корректным (рис. 10.17).

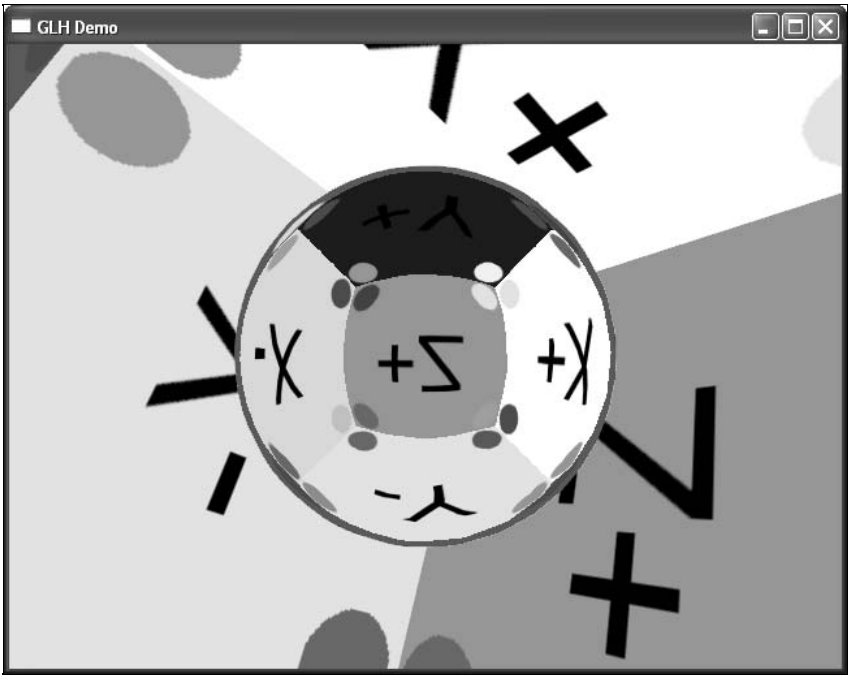


Рис. 10.17. Некорректное моделирование зеркального объекта в примере Ex04 — отражение не зависит от положения наблюдателя

В этом нет ничего удивительного: OpenGL при генерации текстурных координат в режиме `GL_REFLECTION_MAP_ARB` не имеет никакой информации о том, как расположены грани кубической текстуры относительно наблюдателя. Поэтому OpenGL предполагает, что взгляд наблюдателя все время направлен вдоль оси Z в направлении грани −Z. В результате в центре шара все время находится отражение грани +Z.

Один из вариантов решения этой проблемы сводится к динамическому обновлению кубической текстуры при каждом изменении положения наблюдателя. Правда для этого придется каждый раз заново визуализировать все шесть граней кубической текстуры, что отрицательно повлияет на производительность.

Но давайте ответим на вопрос, каким образом изменение положения наблюдателя влияет на отражение? Для этого попробуйте мысленно "покрутиться" вокруг стеклянного шара. Как при этом будет вести себя отражение? Если вы будете поворачиваться влево, отражение окружающей среды будет поворачиваться вправо, если же вы повернетесь вправо, то отражение повернется влево, и т. д. Иными словами, отражение всегда поворачивается в противоположную сторону.

Итак, нам необходимо научиться поворачивать отражение, формируемое с использованием кубической текстурной карты. Выше говорилось о том, что OpenGL в режиме `GL_REFLECTION_MAP_ARB` рассчитывает для каждой вершины вектор направления отражения взгляда наблюдателя, компоненты которого (`x`, `y`, `z`) затем заносятся в текстурные координаты (*s*, *t*, *r*). При этом, OpenGL предполагает, что каждая вершина расположена в центре виртуального куба, образованного гранями кубической карты. Следовательно, поворот отражения сводится к простому повороту векторов направлений, хранящихся в текстурных координатах вершин. В OpenGL аффинные преобразования над текстурными координатами выполняются при помощи текстурной матрицы `GL_TEXTURE_MATRIX`, которая по умолчанию равна единичной матрице.

Думаю, сейчас вы уже сами догадались, какие операции нам необходимо выполнить для корректного поворота отражения при повороте наблюдателя.

1. Получить текущую матрицу модели.
2. Вычислить обратную матрицу к матрице модели.
3. Обнулить все коэффициенты обратной матрицы, отвечающие за перенос вдоль осей *x*, *y* и *z*.
4. Умножить текстурную матрицу на полученную матрицу.

Рис. 10.18. Искажение отражения при необнуленных коэффициентах матрицы, отвечающих за перемещение

Остановимся на 3 пункте. Что произойдет, если мы не обнулим у обратной матрицы коэффициенты, отвечающие за перенос? Да ничего хорошего! Дело в том, что OpenGL в режиме `GL_REFLECTION_MAP_ARB` рассчитывает направление отраженного вектора взгляда. Чему равен модуль этого вектора — никто не знает (это зависит от конкретной реализации OpenGL). Поэтому результат умножения вектора отражения на матрицу переноса попросту непредсказуем. Следовательно, если вы не обнулите коэффициенты, отвечающие за перенос вдоль осей *x*, *y* и *z*, то с вероятностью 99% получите вместо отражения смутное пятно (рис. 10.18).

После всего, что мы выяснили, исправить пример Ex04 не составит большого труда. В листинге 10.7 приведен исходный текст исправленной функции `Display` (Ex05).

Листинг 10.7

```
void Display()
{
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Рисуем окружающую среду
       walls.call_list();
       glMatrixMode(GL_TEXTURE);
// Сохраняем текстурную матрицу
       glPushMatrix();
// Получаем матрицу модели
       matrix4f m=get_matrix(GL_MODELVIEW_MATRIX);
// Вычисляем обратную матрицу
       m=m.inverse();
// Обнуляем коэффициенты матрицы, отвечающие за перенос вдоль осей x,
// y и z
       m(0,3)=0;
       m(1,3)=0;
       m(2,3)=0;
// Умножаем текстурную матрицу на рассчитанную матрицу
       glMultMatrixf(&m(0, 0));
       glMatrixMode(GL_MODELVIEW);
// Рисуем объект
       list0.call_list();
// Восстанавливаем текстурную матрицу
       glMatrixMode(GL_TEXTURE);
```

```
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
}
```

Теперь наша программа работает нормально (рис. 10.19). Однако в ней есть небольшой изъян. Дело в том, что программа рассчитывает обратную матрицу "в лоб" (с использованием метода `inverse` класса `matrix4f`). А это, как известно, очень трудоемкая операция.

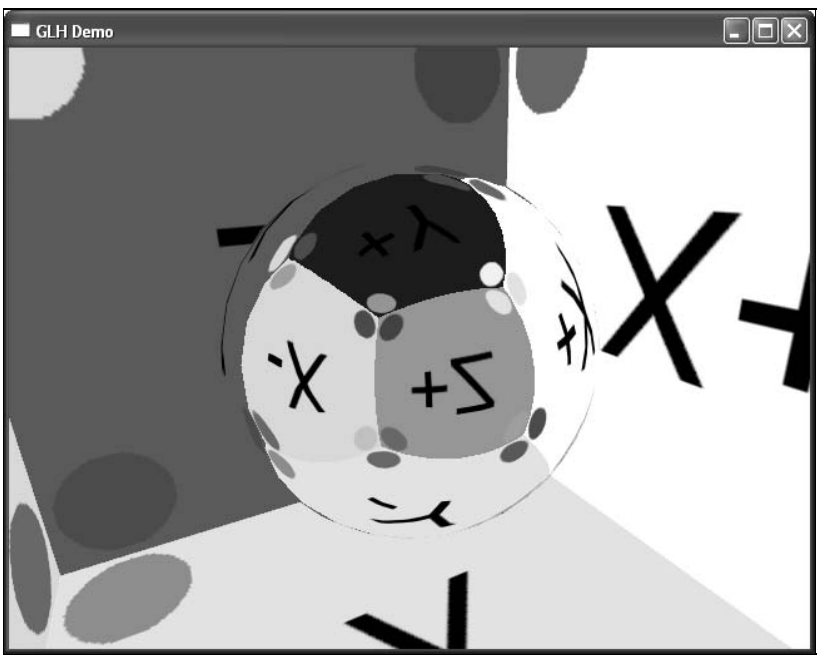


Рис. 10.19. Корректное отражение

Однако в нашем случае матрицу модели рассчитывает интерактор `glut_simple_user_interactor`, который имеет метод быстрого вычисления матрицы обратного преобразования — `get_inverse_transform()`. В результате, заменив в функции `Display` строки

```
matrix4f m=get_matrix(GL_MODELVIEW_MATRIX);
m=m.inverse();
```

на строку

```
matrix4f m=user.get_inverse_transform();
```

мы получим ощутимую прибавку к производительности.

Теперь попробуйте немного проиграться с параметрами тесселяции сферы, чтобы изучить зависимость между качеством отражения и количеством полигонов в объекте. Очень скоро вы заметите, что уменьшение числа полигонов значительно искажает отражение. Например, плавные кривые превращаются в ломаные прямые (рис. 10.20). Как такое может быть? Ведь в начале *разд. 10.2* говорилось о том, что при наложении кубических карт OpenGL *попиксельно* преобразует трехмерные векторы направлений в текстурные координаты, в результате чего кубические карты значительно лучше подходят для текстурирования малополигональных объектов, чем сферические карты. Следовательно, изменение количества полигонов в меньшую сторону не должно отрицательно сказываться на качестве отражения.

Рис. 10.20. Зеркальный шар, состоящий из 512 треугольников (16×16×2). Обратите внимание, что у отражения кривые линии превратились в ломанные

Все дело в том, что OpenGL в режиме `GL_REFLECTION_MAP_ARB` рассчитывает векторы направления отражения взгляда наблюдателя на уровне вершин, а значения векторов в промежуточных пикселах получаются путем простой интерполяции. Иными словами, когда OpenGL переводит трехмерные векторы направления в двухмерные текстурные координаты, он работает с приближенными значениями трехмерных векторов. При этом, чем больше размер полигона, тем больше погрешность (и соответственно, тем сильнее искажения). Тем не менее интерполяция трехмерных векторов направления намного

предпочтительнее простой интерполяции двухмерных текстурных координат, используемой OpenGL при наложении сферических текстурных карт.

В заключение нам остается только заменить шар чайником, тестовую кубическую карту кубической картой горной местности, а также увеличить размер куба с 5 до 50, и мы получим чайник, парящий над горами (см. рис. 10.1).

10.2.4. Моделирование динамического отражения

В предыдущем разделе мы изучили моделирование отражения с использованием статических кубических карт. Хотя этот метод хорошо походит для моделирования отражения статической окружающей среды, он совершенно не подходит для моделирования отражений на глянцевых объектах, находящихся в динамической окружающей среде: любое изменение окружающей среды объекта приводит к изменению кубической текстурной карты. Следовательно, для моделирования отражения динамической окружающей среды нам необходимо научиться генерировать в реальном времени кубические карты на основе окружающей среды объекта.

Создание кубической текстуры осуществляется методом "фотографирования" пространства вокруг объекта виртуальной камерой с углом зрения 90 градусов. Для покрытия всего пространства вокруг объекта будет достаточно шести фотографий: справа, слева, сверху, снизу, спереди и сзади объекта. Иными словами, код создания кубической текстуры должен иметь примерно следующую структуру:

1. Настраиваем параметры перспективной проекции командой вида `gluPerspective(90, 1, …)`.
2. Помещаем камеру в центр объекта, на который будет наложена кубическая карта, и настраиваем направление взгляда камеры в нужном направлении (эту операцию можно выполнить с использованием команды `gluLookAt`).
3. Рисуем окружающую среду объекта.
4. Копируем полученное изображение в соответствующую грань кубической текстуры.
5. Повторяем шаги 2—4 для остальных граней кубической текстурной карты.

Все вроде бы очень просто. Но если вы никогда не писали подобные приложения, то вы, скорее всего, запутаетесь в настройках параметров камеры на шаге 2: единственный способ проверить правильность генерации кубической текстурной карты — наложить полученную текстуру на объект. А глядя на неправильно сгенерированное отражение, довольно трудно определить, в чем ошибка. Мы ведь даже не можем с уверенностью сказать, какая текстурная грань соответствует данному фрагменту отражения.

Поэтому для облегчения процесса работы с динамическими кубическими картами мы должны разработать средства для отображения плоской развертки кубической текстурной карты на экране. Один из вариантов такой развертки приведен на рис. 10.21.

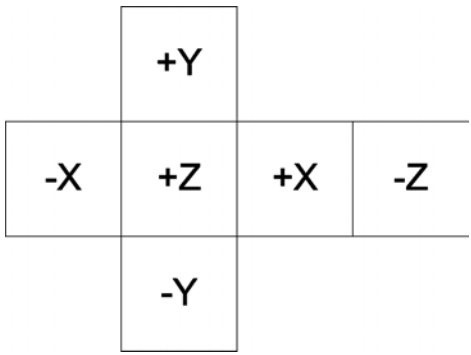


Рис. 10.21. Двухмерная развертка кубической карты

Давайте попробуем сформулировать требования, которым должен удовлетворять наш инструментарий.

1. Простота подключения к проекту. Добавление вывода развертки кубической карты не должно требовать модификации более 10 строк.
2. Развертка кубической карты должна выводиться поверх существующего изображения.
3. Программист должен иметь возможность задать размер и положение развертки кубической карты на экране.
4. Пользователь должен иметь возможность включать/отключать показ развертки кубической карты прямо из консоли.
5. Любая модификация кубической карты должна тут же приводить к модификации отображаемой развертки.

Следствием из первого пункта является необходимость инкапсуляции всего кода инструментария внутри интерактора, который мы назовем `glut_cubemap`. Этот интерактор должен иметь поля, в которые пользователь будет заносить размер и положение развертки кубической карты. При этом, размер и позиция должны указываться в относительных координатах (от 0 до 1), в противном случае (при использовании абсолютных координат) изменение размеров окна потребует модификации положения развертки кубической карты.

Четвертый пункт означает, что интерактор должен иметь обработчик консольных команд. Нашему интерактору вполне хватит поддержки единствен-

ной консольной команды `show_cubemap`, принимающей в качестве параметра целое число от 0 до 2. Ноль выключает показ развертки кубической текстуры, единица включает вывод развертки кубической текстуры, а двойка включает отображение поверх развертки кубической текстуры идентификатора граней (для удобства пользователя). Как должны выглядеть идентификаторы граней? Первое, что приходит в голову, — это то, что идентификаторы должны представлять собой надписи вида +X, −X, +Y, −Y, +Z и −Z поверх соответствующих граней. Какого цвета должны быть эти надписи? Ведь белая надпись будет сливаться с белым фоном, а черная — с черным. А поскольку содержимое динамической текстуры непрерывно меняется, цвет текстуры под надписью тоже будет изменяться. Выходом из сложившегося противоречия заключается заливка экрана в окрестностях надписей однотонным фоном. К примеру, можно организовать вывод белых надписей на черном фоне.

В последнем (пятом) пункте говорится о том, что интерактор `glut_cubemap` должен оперативно реагировать на любое изменение в кубической текстурной карте. Иными словами, интерактор должен формировать развертку кубической карты на основе кубической текстурной карты, идентификатор которой будет передаваться интерактору в качестве параметра.

Итак, перед нами возникает задача построения развертки кубической текстурной карты в реальном времени. Фактически эта задача сводится к выделению из кубической текстурной карты отдельных граней с последующим их наложением на квадратные полигоны. Казалось бы, чего здесь сложного? Расширение ARB_texture_cube_map предоставляет программисту доступ к изображениям граней кубической текстуры при помощи команды `glGetTexImage`. Следовательно, мы можем создать при помощи связки команд `glGetTexImage` и `glTexImage2D` набор из шести двухмерных текстур, соответствующих граням кубических текстурных карт, а затем наложить их на квадратные полигоны. Однако у этого способа есть один существенный недостаток — передача изображений из кубической текстуры в двухмерные текстуры осуществляется через медленную оперативную память. А т. к. построение развертки будет осуществляться при рисовании каждого кадра, этот недостаток приведет к уменьшению производительности программы в несколько раз.

А может быть стоит попытаться наложить кубическую текстуру на квадратный полигон таким образом, чтобы на экране была видна только требуемая грань кубической текстуры? Что надо сделать, чтобы в квадратном полигоне, состоящем из четырех вершин, было видно отражение только грани +Z кубической текстуры? Правильно. Задать текстурные координаты в вершинах квадрата таким образом, чтобы в углах квадрата отражались углы грани +Z кубической текстурной карты. Так как углам грани +Z соответствуют трехмерные векторы направлений (−1, −1, 1), (1, −1, 1), (1, 1, 1) и (−1, 1, 1),

код рисования такого квадратного полигона будет иметь примерно следующий вид:

```
glBegin(GL_QUADS);
       glTexCoord3f(-1, -1, 1);
       glVertex2f(-1, -1);

       glTexCoord3f(1, -1, 1);
       glVertex2f(1, -1);

       glTexCoord3f(1, 1, 1);
       glVertex2f(1, 1);

       glTexCoord3f(-1, 1, 1);
       glVertex2f(-1, 1);
glEnd();
```

Код вывода остальных текстурных граней будет отличаться лишь текстурными координатами углов граней.

Думаю, после всего вышесказанного написание интерактора `glut_cubemap` не составит большого труда. Исходный код этого интерактора приведен в листинге 10.8.

Листинг 10.8

```
class glut_cubemap:public glut_interactor
{
public:
// Указатель на кубическую текстурную карту
       tex_object_cube_map* cube_map;
// Ширина кубической карты (ширина экрана=1)
       float sx;
// Высота кубической карты (высота экрана=1)
       float sy;
// Координата левого нижнего угла кубической карты (ось x)
       float tx;
// Координата левого нижнего угла кубической карты (ось y)
       float ty;
// Показывать ли кубическую карту на экране
       bool bVisible;
```

```
// Указатель на интерактор консоли, в котором будут зарегистрированы
// консольные команды
       glut_console* console;
// Ширина одного квадрата с гранью кубической карты в пикселах
       int pixel_square_sx;
// Высота одного квадрата с гранью кубической карты в пикселах
       int pixel_square_sy;
// Показывать ли идентификаторы граней
       bool bShowMarks;
       glut_cubemap(glut_console* aconsole=0);
       void DrawText(string str);
       virtual void display();
       static bool stdCmdProc(void* Object, string Cmd, vector<string>
Params, glut_console* console);
};


//Обработчик консольных команд
bool glut_cubemap::stdCmdProc(void* Object, string Cmd, vector<string>
Params, glut_console* console)
{
       assert(console);
       assert(Object);


// Получаем указатель на экземпляр интерактора
       glut_cubemap* cubemap=(glut_cubemap*) Object;


// Если текущая команда — show_map
       if (Cmd=="show_cubemap")
       {
// Если нет параметров — выводим информацию о текущем состоянии
// интерактора
              if (Params.empty())
              {
                     if (cubemap->bVisible)
                            if (cubemap->bShowMarks)
                                   console->add("Net of cubemap is
visible with marks");
                            else
                                   console->add("Net of cubemap is
visible");
```

```
                else
                        console->add("Bet of cubemap is hidden");
                return true;
        }

// Если параметр равен /?, то выводим справку по команде
        if (Params[0]=="/?")
        {
                console->add("shows net of cubemap texture");
                console->add("");
                console->add("show_cubmap <visible>");
                console->add("where");
                console->add("  <visible> is 0 when net of cubemap
is hidden");
                console->add("               1, when net of cubmap
is visible");
                console->add("               2, when net of cubmap
is visible with marks");
                return true;
        }
// Если параметр равен 0, то отключаем показ развертки
        if (Params[0]=="0")
        {
                cubemap->bVisible=false;
                glutPostRedisplay();
                return true;
        }
// Если параметр равен 1, то включаем показ развертки
        if (Params[0]=="1")
        {
                cubemap->bVisible=true;
                cubemap->bShowMarks=false;
                glutPostRedisplay();
                return true;
        }

// Если параметр равен 2, то включаем показ идентификатора граней
        if (Params[0]=="2")
        {
```

```
                        cubemap->bVisible=true;
                        cubemap->bShowMarks=true;
                        glutPostRedisplay();
                        return true;
                }
        }

        return false;
}

/*
Конструктор aconsole  - указатель на консоль, используемую для вывода
сообщений и обработки команд
*/
glut_cubemap::glut_cubemap(glut_console* aconsole)
{
        assert(aconsole);

// По умолчанию кубическая карта отсутствует
        cube_map=0;
// Размеры развертки кубической карты
        sx=1;
        sy=1;
// Положение развертки
        tx=0;
        ty=0;
// По умолчанию развертка невидима
        bVisible=false;
        console=aconsole;
// Идентификаторы граней невидимы
        bShowMarks=false;

// Регистрируем консольную команду
        console->addCmd("show_cubemap", "shows net of cubemap texture",
this, stdCmdProc);
// Выполняем скрипт из файла cubemap.cs
        console->processCmd("exec cubemap.cs");
}
```

```
// Обработчик события display
void glut_cubemap::display()
{
// Если развертка видна и задана кубическая текстура
       if (cube_map && bVisible)
       {
// Определяем размеры окна
              int window_width=glutGet(GLUT_WINDOW_WIDTH);
              int window_height=glutGet(GLUT_WINDOW_HEIGHT);
// Вычисляем размеры квадратов развертки в пикселах
              pixel_square_sx=window_width*sx/4.0;
              pixel_square_sy=window_height*sy/3.0;
// Инициализируем дисплейные списки консоли с символами GLUT
              console->initCharLists();

              glPushAttrib(GL_ALL_ATTRIB_BITS);
// Отключаем тексты глубины и освещения
              glDisable(GL_DEPTH_TEST);
              glDisable(GL_LIGHTING);
// Задаем базовое смещение для дисплейных списков
              glListBase(console->TimesRoman24ListBase);

// Активизируем кубическую карту
              cube_map->bind();
// Задаем режим наложения текстуры
              glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
GL_REPLACE);

// Задаем ортогональную проекцию
              glMatrixMode(GL_PROJECTION);
              glPushMatrix();
              glLoadIdentity();
              gluOrtho2D(-1, 1, -1, 1);
// Обнуляем матрицу модели
              glMatrixMode(GL_MODELVIEW);
              glPushMatrix();
              glLoadIdentity();
```

```
// Рисуем квадрат развертки кубической текстуры (грань +Z)
// Совмещаем видовое окно с положением квадрата
              glViewport(pixel_square_sx+tx*window_width,
pixel_square_sy+ty*window_height, pixel_square_sx, pixel_square_sy);
              cube_map->enable();
// Рисуем квадрат размером во все видовое окно
              glBegin(GL_QUADS);
                     glTexCoord3f(-1, -1, 1);
                     glVertex2f(-1, -1);

                     glTexCoord3f(1, -1, 1);
                     glVertex2f(1, -1);

                     glTexCoord3f(1, 1, 1);
                     glVertex2f(1, 1);

                     glTexCoord3f(-1, 1, 1);
                     glVertex2f(-1, 1);
              glEnd();
// Выводим идентификатор грани
              DrawText("+Z");


// Рисуем развертку грани -X
              glViewport(tx*window_width,
pixel_square_sy+ty*window_height, pixel_square_sx, pixel_square_sy);
              glBegin(GL_QUADS);
                     glTexCoord3f(-1, -1, -1);
                     glVertex2f(-1, -1);

                     glTexCoord3f(-1, -1, 1);
                     glVertex2f(1, -1);

                     glTexCoord3f(-1, 1, 1);
                     glVertex2f(1, 1);

                     glTexCoord3f(-1, 1, -1);
                     glVertex2f(-1, 1);
              glEnd();
              DrawText("-X");
```

```
// Рисуем развертку грани +X
              glViewport(2*pixel_square_sx+tx*window_width,
pixel_square_sy+ty*window_height, pixel_square_sx, pixel_square_sy);
              glBegin(GL_QUADS);
                     glTexCoord3f(1, -1, 1);
                     glVertex2f(-1, -1);

                     glTexCoord3f(1, -1, -1);
                     glVertex2f(1, -1);

                     glTexCoord3f(1, 1, -1);
                     glVertex2f(1, 1);

                     glTexCoord3f(1, 1, 1);
                     glVertex2f(-1, 1);
              glEnd();
              DrawText("+X");


// Рисуем развертку грани -Z
              glViewport(3*pixel_square_sx+tx*window_width,
pixel_square_sy+ty*window_height, pixel_square_sx, pixel_square_sy);
              glBegin(GL_QUADS);
                     glTexCoord3f(1, -1, -1);
                     glVertex2f(-1, -1);

                     glTexCoord3f(-1, -1, -1);
                     glVertex2f(1, -1);

                     glTexCoord3f(-1, 1, -1);
                     glVertex2f(1, 1);

                     glTexCoord3f(1, 1, -1);
                     glVertex2f(-1, 1);
              glEnd();
              DrawText("-Z");


// Рисуем развертку грани +Y
              glViewport(pixel_square_sx+tx*window_width,
2*pixel_square_sy+ty*window_height, pixel_square_sx, pixel_square_sy);
```

```
            glBegin(GL_QUADS);
                  glTexCoord3f(-1, 1, 1);
                  glVertex2f(-1, -1);

                  glTexCoord3f(1, 1, 1);
                  glVertex2f(1, -1);

                  glTexCoord3f(1, 1, -1);
                  glVertex2f(1, 1);

                  glTexCoord3f(-1, 1, -1);
                  glVertex2f(-1, 1);
            glEnd();
            DrawText("+Y");

// Рисуем развертку грани -Y
            glViewport(pixel_square_sx+tx*window_width,
ty*window_height, pixel_square_sx, pixel_square_sy);

            glBegin(GL_QUADS);
                  glTexCoord3f(-1, -1, -1);
                  glVertex2f(-1, -1);

                  glTexCoord3f(1, -1, -1);
                  glVertex2f(1, -1);

                  glTexCoord3f(1, -1, 1);
                  glVertex2f(1, 1);

                  glTexCoord3f(-1, -1, 1);
                  glVertex2f(-1, 1);
            glEnd();
            DrawText("-Y");

// Восстанавливаем настройки OpenGL
            glPopMatrix();
            glMatrixMode(GL_PROJECTION);
            glPopMatrix();
            glMatrixMode(GL_MODELVIEW);
```

```
                cube_map->disable();
                glPopAttrib();
        }
}



/* Рисуем белую строку на черном фоне
        str - строка текста
*/
void glut_cubemap::DrawText(string str)
{
// Если включен режим показа идентификатора
        if (bShowMarks)
        {
// Определяем ширину текста в пикселах
                int PixelWidth=0;
                const char *c=str.c_str();
                while (*c)
                {
                PixelWidth+=glutBitmapWidth(GLUT_BITMAP_TIMES_ROMAN_24,
*c);
                        c++;
                }

// Определяем ширину текста в относительных координатах
                float TextWidth=2.0/(pixel_square_sx)*PixelWidth;
// Определяем высоту текста в относительных координатах (высота символов
// используемого шрифта равна 24 пиксела)
                float TextHeight=2.0/(pixel_square_sy)*24.0;

// Выключаем наложение кубических карт
                cube_map->disable();
// Рисуем черный прямоугольник в центре квадрата с разверткой грани
// кубической текстуры. Размер прямоугольника чуть-чуть больше размера
// выводимой строки символов
                glColor3f(0.0, 0.0, 0.0);
```

```
                glRectf(-TextWidth*1.2/2.0, -TextHeight*1.2/2.0,
TextWidth*1.2/2.0, TextHeight/2.0);

                glColor3f(1.0, 1.0, 1.0);
// Задаем позицию вывода строки символов таким образом, чтобы строка
// оказалась в центре грани
                glRasterPos2f(-TextWidth/2.0, -TextHeight/2.0);
// Выводим строку
                glCallLists(str.length(), GL_UNSIGNED_BYTE, str.c_str());
                cube_map->enable();
        }
};
```

Для того чтобы протестировать интерактор `glut_cubemap`, мы добавим в пример Ex05 возможность вывода развертки кубической текстуры на экран (Ex06). Для этого нам потребуется добавить в файл main.cpp буквально несколько строк:

```
...
// Интерактор glut_cubemap
glut_cubemap cubemap(&console);
...
int main(int argc, char* argv[])
{
...
// Интерактор cubemap будет показывать развертку кубической текстуры
// reflect_texture
        cubemap.cube_map=&reflect_texture;
// Размер текстурной развертки будет составлять 50% от размеров окна
// вдоль осей x и y
        cubemap.sx=0.5;
        cubemap.sy=0.5;

// Задаем размер окна
        glut_add_interactor(&user);
        glut_add_interactor(&cb);
        glut_add_interactor(&cubemap);
        glut_add_interactor(&console);
        glut_add_interactor(&swapbuffers);
...
}
```

Как видно, для того, подключить интерактор `glut_cubemap` к программе, чтобы нам потребовалось добавить всего 5 строк! Внешний вид окна полученной программы изображен на рис. 10.22.

Рис. 10.22. Внешний вид двухмерной развертки кубической текстурной карты

Теперь мы можем поместить в пакетный файл cubemap.cs, автоматически выполняемый после создания интерактора `glut_cubemap` небольшой скрипт, позволяющий пользователю циклически изменять режим отображения развертки кубической текстуры простым нажатием клавиши <F3>:

```
begin_script show_cubemap_script
bind F3 "run_script show_cubemap_marks_script"
show_cubemap 1
end_script

begin_script hide_cubemap_script
bind F3 "run_script show_cubemap_script"
show_cubemap 0
end_script
```

```
begin_script show_cubemap_marks_script
bind F3 "run_script hide_cubemap_script"
show_cubemap 2
end_script

bind F3 "run_script show_cubemap_script"
```

Теперь мы можем приступать к созданию программ, использующих динамические кубические текстурные карты. Для начала мы перепишем пример Ex06, заменив статические текстурные карты на динамические (Ex07). Так как в начале этого раздела мы уже подробно рассматривали алгоритм генерации динамических текстурных карт, я сразу приведу листинг с наиболее важными фрагментами этой программы (листинг 10.9).

Листинг 10.9

```
// Создаем буфер pbuffer разрешением 256×256
PBuffer pbuffer0(256, 256, GLUT_RGB | GLUT_DEPTH | GLUT_SINGLE);
// Идентификатор главного окна программы
GLuint WinID;
// Инициализируем OpenGL
bool Init()
{
// Инициализируем необходимые расширения
       if (!glh_init_extensions("GL_ARB_texture_cube_map
GL_SGIS_generate_mipmap WGL_ARB_pbuffer WGL_ARB_pixel_format"))
       {
              console.add(string("Unsupported extensions:
")+glh_get_unsupported_extensions());
              console.exit(-1);
              return false;
       }
// Получаем идентификатор главного окна
       WinID=glutGetWindow();
// Инициализируем буфер pbuffer с собственным контекстом, но разделяемыми
// списками
       pbuffer0.Initialize(false, true);
       glEnable(GL_DEPTH_TEST);
// Загружаем кубическую карту в оперативную память
       string info;
```

```
        if (!read("..\\cloudyhills.jpg", images, &info))
        {
                console.add(info);
                console.exit(-1);
                return false;
        }
// Создаем кубическую текстуру
        reflect_texture.bind();
// Включаем автоматическую генерацию Mipmap-уровней
        reflect_texture.parameter(GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
// Настраиваем параметры Mipmap-фильтрации
        reflect_texture.parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        reflect_texture.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Задаем оболочки для текстурных координат S и T
        reflect_texture.parameter(GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        reflect_texture.parameter(GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
// Выделяем в видеопамяти место для хранения граней кубической текстурной
// карты. Сама кубическая текстурная карта будет создана позже
        for (int i=GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB;
i<=GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB; i++)
                glTexImage2D(i, 0, GL_RGB, pbuffer0.GetWidth(),
pbuffer0.GetHeight(), 0, GL_RGB, GL_UNSIGNED_BYTE, 0);
// Создаем дисплейные списки чайников и граней куба окружающей среды
(соответственно list0 и walls)
...     glEnable(GL_DEPTH_TEST);
        string info;
        if (!read("..\\cloudyhills.jpg", images, &info))
        {
                console.add(info);
                console.exit(-1);
                return false;
        }
        reflect_texture.bind();
        reflect_texture.parameter(GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
        reflect_texture.parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        reflect_texture.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        reflect_texture.parameter(GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        reflect_texture.parameter(GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

```
       for (int i=GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB;
i<=GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB; i++)
              glTexImage2D(i, 0, GL_RGB, pbuffer0.GetWidth(),
pbuffer0.GetHeight(), 0, GL_RGB, GL_UNSIGNED_BYTE, 0);
...
}
// Обновляем кубическую текстурную карту окружающей среды
void MakeCubemapTexture()
{
// Получаем матрицу модели
       matrix4f m=get_matrix(GL_MODELVIEW_MATRIX);
// Обнуляем у нее коэффициенты, отвечающие за перенос. Иными словами,
// выделяем из нее матрицу поворота
       m(0,3)=0.0;
       m(1,3)=0.0;
       m(2,3)=0.0;
// Если произошла потеря буфера pbuffer, создаем его заново
       pbuffer0.HandleModeSwitch();
// Делаем контекст буфера pbuffer текущим
       pbuffer0.MakeCurrent();
// Создаем видовое окно размером во весь буфер pbuffer
       glViewport(0, 0, pbuffer0.GetWidth(), pbuffer0.GetHeight());
// Устанавливаем параметры матрицы проекции
       glMatrixMode(GL_PROJECTION);
       glLoadIdentity();
       gluPerspective(90, 1, 0.1, 100);
       glMatrixMode(GL_MODELVIEW);
// Генерируем грань +X кубической текстуры
// Очищаем экран
       glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Задаем позицию и направление взгляда наблюдателя
       glLoadIdentity();
       gluLookAt(0, 0, 0, 1, 0, 0, 0, -1, 0);
// Умножаем матрицу модели буфера pbuffer на текущую матрицу поворота
// главного окна
       glMultMatrixf(&m(0, 0));
// Рисуем окружающую среду
       walls.call_list();
```

```
// Копируем полученное изображение в грань +X кубической текстуры
        reflect_texture.bind();
        glCopyTexSubImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB, 0, 0, 0,
0, 0, pbuffer0.GetWidth(), pbuffer0.GetHeight());
// Генерируем грань -Y кубической текстуры
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        gluLookAt(0, 0, 0, -1, 0, 0, 0, -1, 0);
        glMultMatrixf(&m(0, 0));
        walls.call_list();
        reflect_texture.bind();
        glCopyTexSubImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB, 0, 0, 0,
0, 0, pbuffer0.GetWidth(), pbuffer0.GetHeight());
// Генерируем грань + Y кубической текстуры
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        gluLookAt(0, 0, 0, 0, 1, 0, 0, 0, 1);
        glMultMatrixf(&m(0, 0));
        walls.call_list();
        reflect_texture.bind();
        glCopyTexSubImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB, 0, 0, 0,
0, 0, pbuffer0.GetWidth(), pbuffer0.GetHeight());
// Генерируем грань -Y кубической текстуры
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        gluLookAt(0, 0, 0, 0, -1, 0, 0, 0, -1);
        glMultMatrixf(&m(0, 0));
        walls.call_list();
        reflect_texture.bind();
        glCopyTexSubImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB, 0, 0, 0,
0, 0, pbuffer0.GetWidth(), pbuffer0.GetHeight());
// Генерируем грань +Z кубической текстуры
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        gluLookAt(0, 0, 0, 0, 0, 1, 0, -1, 0);
        glMultMatrixf(&m(0, 0));
        walls.call_list();
        reflect_texture.bind();
        glCopyTexSubImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB, 0, 0, 0,
0, 0, pbuffer0.GetWidth(), pbuffer0.GetHeight());
```

```
// Генерируем грань -Z кубической текстуры
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        gluLookAt(0, 0, 0, 0, 0, -1, 0, -1, 0);
        glMultMatrixf(&m(0, 0));
        walls.call_list();
        reflect_texture.bind();
        glCopyTexSubImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB, 0, 0, 0,
0, 0, pbuffer0.GetWidth(), pbuffer0.GetHeight());
// Переключаемся в главное окно программы
        glutSetWindow(WinID);
}
// Обновляем содержимое экрана
void Display()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Обновляем кубическую текстурную карту
        MakeCubemapTexture();
// Рисуем глянцевый чайник
        list0.call_list();
// Рисуем окружающую среду вокруг чайника
        walls.call_list();
}
```

Пример Ex07 внешне практически не отличается от примера Ex06. Однако если вы выведете на экран развертку кубической текстурной карты и попробуете повертеть изображение при помощи мыши, то увидите, что синхронизация отражения с окружающей средой осуществляется путем честного пересчета кубической текстурной карты вместо коррекции текстурной матрицы (рис. 10.23). Тем не менее использование динамических текстур в примере Ex07 приводит к бессмысленному усложнению программы и падению производительности. Поэтому еще раз повторюсь: никогда не используйте динамические кубические текстурные карты там, где можно обойтись простыми статическими текстурами.

Так как пример Ex07 генерирует кубические текстурные карты окружающей среды на лету, мы можем легко добавить в сцену подвижные объекты, например, несколько разноцветных чайников, летающих вокруг глянцевого чайника. Для этого нам придется написать новую функцию рисования окружающей среды `DrawOutdoor`, а также немного модифицировать функции

`MakeCubemapTexture` и `Display`, заменив строку `walls.call_list()` на вызов функции `DrawOutdoor` (листинг 10.10).

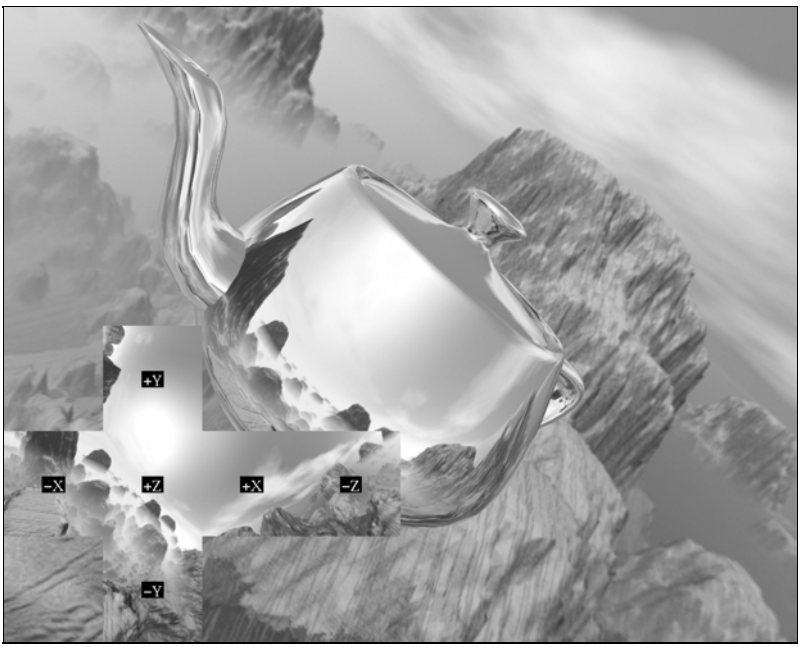


Рис. 10.23. Внешний вид примера Ex07. Обратите внимание, что поворот сцены привел к изменению кубической текстурной карты

Листинг 10.10

```
// Дисплейный список рисования чайника
display_list teapot;
// Структура с информацией о чайнике
struct Teapot
{
// Цвет чайника
       vec3 color;
// Положение чайника
       vec3 pos;
// Поворот чайника
       float rotate;
};
```

```
// Количество чайников
const TeapotCount=6;
// Массив с информацией о чайниках
Teapot Teapots[TeapotCount];
// Время запуска программы
GLuint StartTick;
// Инициализируем OpenGL
bool Init()
{
...
// Создаем дисплейный список чайника
       teapot.new_list(GL_COMPILE);
              glutSolidTeapot(2.0);
       teapot.end_list();
// Перебираем все чайники
       for (unsigned int i=0; i<TeapotCount; i++)
       {
// Задаем позицию чайника относительно оси y
              Teapots[i].pos.y=0;
// Генерируем случайный цвет чайника, пока сумма компонентов R + G + B не
// будет больше 1 (чтобы не было чайников темных цветов)
              do
              {
                     Teapots[i].color.x=nv_random()/2.0+0.5;
                     Teapots[i].color.y=nv_random()/2.0+0.5;
                     Teapots[i].color.z=nv_random()/2.0+0.5;
              }
              while
(Teapots[i].color.x+Teapots[i].color.y+Teapots[i].color.z<1.0);
       }
...
// Запоминаем момент запуска программы
       StartTick=GetTickCount();
       return true;
}
// Рисуем окружающую среду вокруг чайника
void DrawOutdoor()
{
```

```
// Настраиваем параметры OpenGL
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        glDisable(GL_CULL_FACE);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_COLOR_MATERIAL);
        glEnable(GL_DEPTH_TEST);
// Перебираем все чайники
        for (unsigned int i=0; i<TeapotCount; i++)
        {
                glPushMatrix();
// Задаем положение и цвет чайника
                glTranslatef(Teapots[i].pos.x, Teapots[i].pos.y, Tea-
pots[i].pos.z);
                glColor3fv(&Teapots[i].color.x);
                glRotatef(Teapots[i].rotate, 0, 1.0, 0);
// Рисуем чайник
                teapot.call_list();
                glPopMatrix();
        }
// Рисуем куб с окружающей средой
        walls.call_list();
        glPopAttrib();
}
// Обновляем кубическую текстурную карту окружающей среды
void MakeCubemapTexture()
{
// Выделяем матрицу поворота
        matrix4f m=get_matrix(GL_MODELVIEW_MATRIX);
        m(0,3)=0.0;
        m(1,3)=0.0;
        m(2,3)=0.0;
// Активизируем буфер pbuffer
        pbuffer0.HandleModeSwitch();
        pbuffer0.MakeCurrent();
// Настраиваем параметры видового окна и матрицы проекции
        glViewport(0, 0, pbuffer0.GetWidth(), pbuffer0.GetHeight());
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
```

```
        gluPerspective(90, 1, 0.1, 100);
        glMatrixMode(GL_MODELVIEW);
// Рисуем грань +X кубической текстурной карты
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        gluLookAt(0, 0, 0, 1, 0, 0, 0, -1, 0);
        glMultMatrixf(&m(0, 0));
// Рисуем окружающую среду чайника
        DrawOutdoor();
        reflect_texture.bind();
        glCopyTexSubImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB, 0, 0, 0,
0, 0, pbuffer0.GetWidth(), pbuffer0.GetHeight());
...
}
void Display()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Обновляем кубическую текстуру
        MakeCubemapTexture();
// Рисуем глянцевый чайник
        list0.call_list();
// Рисуем окружающую среду
        DrawOutdoor();
}
...
// Обновляем положение чайников в моменты бездействия программы
void Idle()
{
// Определяем время работы программы в секундах
        float t=float(GetTickCount()-StartTick)/3000.0;
// Перебираем все чайники
        for (unsigned int i=0; i<TeapotCount; i++)
        {
// Определяем новое положение чайника
                Teapots[i].pos.x=10.0*sin(t);
                Teapots[i].pos.z=10.0*cos(t);
// Определяем угол поворота чайника
                Teapots[i].rotate=(t*180.0/nv_pi);
```

```
// Переходим к следующему чайнику
                t+=2.0*nv_pi/float(TeapotCount);
        }
        glutPostRedisplay();
}
```

Программа вроде бы работает нормально: разноцветные чайники отражаются в нашем глянцевом чайнике, а их отражение послушно следует за ними. Однако по отражению чайника почему-то периодически "пробегают" две-три полосы, делящие чайник на две-три зоны, имеющие различные яркости. При рассмотрении развертки кубической карты становится ясно, что это явление возникает только тогда, когда отражение цветного чайника переходит из одной грани кубической текстуры в другую (рис. 10.24).

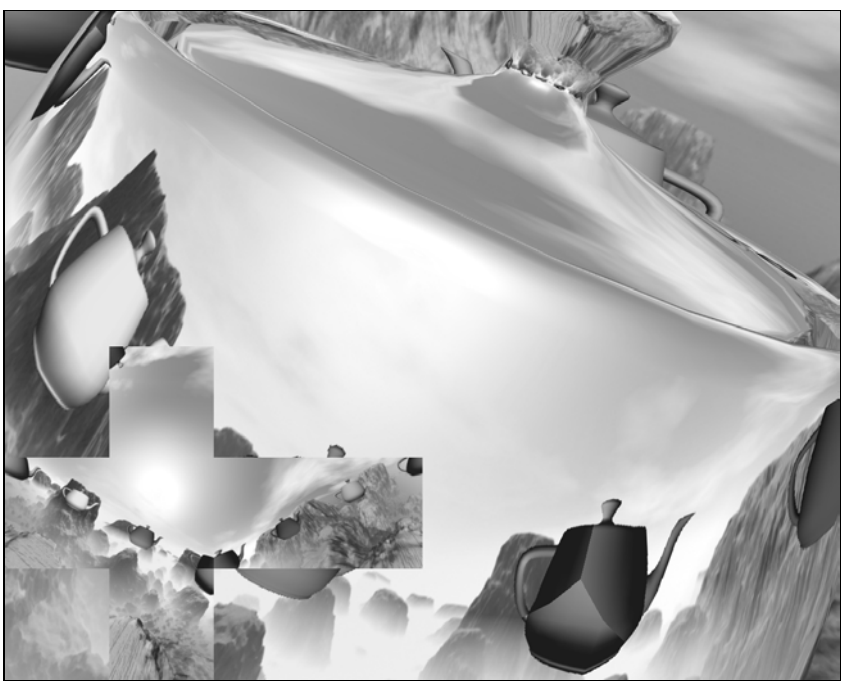


Рис. 10.24. Разделение отражения цветного чайника на зоны разной яркости при переходе отражения из одной грани кубической текстуры в другую (пример Ex08)

С чем это может быть связано? При создании кубической текстуры мы поворачиваем камеру в различных направлениях при помощи команды `gluLookAt`. Если быть более точным, эта команда изменяет матрицу модели

таким образом, что мы видим нужный фрагмент сцены, т. е. двигается сцена, а не камера. Казалось бы — какая разница, ведь результат все равно будет один и тот же. Но это не так — данное утверждение истинно только в случае, когда сцена перемещается целиком, со всеми объектами. Однако в нашей программе это условие не соблюдается — у нас на сцене имеется один объект, который не меняет положение при вращении сцены. Это источник света (`GL_LIGHT0`).

По умолчанию координаты этого источника света `GL_LIGHT0` равны (0, 0, 1, 0), т. е. источник света бесконечно удален вдоль оси *z*. Координаты источника света задаются командой `glLight` с параметром `GL_POSITION`, после чего они умножаются на текущую матрицу модели и заносятся в память видеокарты. После этого источник света всегда находится на одном и том же месте до следующего вызова функции `glLight` с параметром `GL_POSITION`. Иными словами, команда `gluLookAt`, вызываемая после команды `glLight`, не оказывает влияния на положение источника света.

Из всего вышесказанного следует вывод: при вращении сцены командой `gluLookAt` источник света остается на месте, следовательно, для наблюдателя это выглядит так, будто бы источник света двигается по сцене вслед за камерой. Следовательно, в каждой грани кубической текстуры источник света имеет различное положение, что приводит к резкому перепаду яркости на стыке между гранями. Вообще, это явление происходило и раньше (в примере Ex07), но мы его просто не замечали, т. к. рисовали грани куба, симулирующие окружающую среду, с выключенным освещением.

Выход из данной ситуации — изменение положения источника света вместе со сценой. Этот прием демонстрируется в примере Ex09. Ниже приведен фрагмент измененной функции `DrawOutdoor`, которая теперь автоматически изменяет позицию источника света при повороте сцены (листинг 10.11).

Листинг 10.11

```
// Параметры источников света
// Позиции источников света
vec4f light0_pos(1, 0, 0.1, 0);
vec4f light1_pos(-1, -1, 0.5, 0);
vec4f light2_pos(-1, 0.5, -0.3, 0);
// Рассеянная составляющая источника света
vec4f light_ambient(0.3, 0.3, 0.3, 0);
// Диффузная составляющая источников света
vec4f light0_diffuse(0.8, 0.6, 0.6, 0);
vec4f light1_diffuse(0.2, 0.2, 0.4, 0);
vec4f light2_diffuse(0.4, 0.6, 0.4, 0);
```

```
// Рисуем окружающую среду
void DrawOutdoor()
{
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        glDisable(GL_CULL_FACE);
// Включаем источники света
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_LIGHT1);
        glEnable(GL_LIGHT2);
// Включаем цветные материалы
        glEnable(GL_COLOR_MATERIAL);
// Включаем тест глубины
        glEnable(GL_DEPTH_TEST);
// Задаем параметры диффузной составляющей источников света
        glLightfv(GL_LIGHT0, GL_DIFFUSE, &light0_diffuse.v[0]);
        glLightfv(GL_LIGHT1, GL_DIFFUSE, &light1_diffuse.v[0]);
        glLightfv(GL_LIGHT2, GL_DIFFUSE, &light2_diffuse.v[0]);
// Задаем параметры рассеянной составляющей света
        glLightfv(GL_LIGHT0, GL_AMBIENT, &light_ambient.v[0]);
// Задаем позицию источников света
        glLightfv(GL_LIGHT0 ,GL_POSITION, &light0_pos.v[0]);
        glLightfv(GL_LIGHT1 ,GL_POSITION, &light1_pos.v[0]);
        glLightfv(GL_LIGHT2 ,GL_POSITION, &light2_pos.v[0]);
...
// Рисуем чайники и куб окружающей среды
}
```

Теперь отражение на глянцевом чайнике выглядит вполне нормально (рис. 10.25).

Однако производительность программы нормальной не назовешь — даже на NVIDIA GeForce FX 5800 Ultra количество кадров в секунду не превышает 10. С чем это связано? В процессе рисования одного кадра наша программа выводит $1 + 6 + 6 \times 6 = 43$ чайника (1 глянцевый чайник + 6 цветных чайников + 6 граней кубической текстурной карты, каждая из которых содержит по 6 цветных чайников). Учитывая, что каждый чайник состоит из 3136 полигонов *(см. разд. 8.2.1)*, получается, что наша сцена содержит более 138 000 полигонов! Не слишком ли это много для такой простой программы? Надо уменьшить количество полигонов в сцене. Но как?

Рис. 10.25. Корректное отражение цветных чайников в глянцевом чайнике

Хотя наша программа и выводит на экран 43 чайника, в действительности на экране одновременно видно примерно 1 + 6 + 6 = 13 чайников (1 глянцевый чайник + 6 цветных чайников + 6 чайников в отражении[4]). Следовательно, нам надо встроить в программу отсечение невидимых чайников, не попадающих в пирамиду видимости камеры. Такая операция очень просто организуется с использованием классов `psAABB` и `psCamera`, проверяющих попадание прямоугольных оболочек объекта в пирамиду видимости *(см. разд. 7.2.3)*. Обновленный вариант программы приведен в листинге 10.12 (Ex10).

Листинг 10.12

```
// Прямоугольная оболочка чайника
psAABB bound;
// Определена ли прямоугольная оболочка чайника
bool bBuildBound=false;
// Определяем прямоугольную оболочку чайника
psAABB BuildBoundBox()
```

[4] Если часть чайников находится на стыке кубических граней, то число чайников в отражении будет больше 6.

```
{
...
}
// Рисуем окружающую среду
void DrawOutdoor()
{
       glPushAttrib(GL_ALL_ATTRIB_BITS);
// Настраиваем параметры OpenGL
       glDisable(GL_CULL_FACE);
       glEnable(GL_LIGHTING);
       glEnable(GL_LIGHT0);
       glEnable(GL_LIGHT1);
       glEnable(GL_LIGHT2);
       glEnable(GL_COLOR_MATERIAL);
       glEnable(GL_DEPTH_TEST);
// Настраиваем параметры освещения
       glLightfv(GL_LIGHT0, GL_DIFFUSE, &light0_diffuse.v[0]);
       glLightfv(GL_LIGHT1, GL_DIFFUSE, &light1_diffuse.v[0]);
       glLightfv(GL_LIGHT2, GL_DIFFUSE, &light2_diffuse.v[0]);
       glLightfv(GL_LIGHT0, GL_AMBIENT, &light_ambient.v[0]);
       glLightfv(GL_LIGHT0 ,GL_POSITION, &light0_pos.v[0]);
       glLightfv(GL_LIGHT1 ,GL_POSITION, &light1_pos.v[0]);
       glLightfv(GL_LIGHT2 ,GL_POSITION, &light2_pos.v[0]);
// Объект — камера
       psCamera camera;
// Перебираем
       for (unsigned int i=0; i<TeapotCount; i++)
       {
              glPushMatrix();
// Задаем положение чайника
              glTranslatef(Teapots[i].pos.x, Teapots[i].pos.y, Tea-
pots[i].pos.z);
              glRotatef(Teapots[i].rotate, 0, 1.0, 0);
// Формируем параметры камеры исходя из значений матриц проекции и модели
              camera.psGetPositionGL();
// Видна ли прямоугольная оболочка в камере?
              if (!camera.psTest(bound))
              {
```

```
// Если да, рисуем чайник на экране
                    glColor3fv(&Teapots[i].color.x);
                    teapot.call_list();
                }
                glPopMatrix();
        }
// Рисуем куб окружающей среды
        walls.call_list();
        glPopAttrib();
}
// Обновляем содержимое экрана
void Display()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        if (!bBuildBound)
        {
                bound=BuildBoundBox();
                bBuildBound=true;
        }
// Обновляем содержимое экрана
        MakeCubemapTexture();
        list0.call_list();
        DrawOutdoor();
}
```

Из табл. 10.2 видно, что незначительная модификация исходного текста примера привела к удвоению производительности.

Таблица 10.2. *Сравнение производительности примеров Ex09 и Ex10*

Видеокарта	Производиельность (FPS)	
	Ex09	Ex10
NVIDIA GeForce FX 5800 Ultra	11	21
NVIDIA GeForce2 MX 400 32MB	11	12
ATI Radeon 9700 Pro	7	18

10.2.5. Использование расширения ARB_render_texture для работы с кубическими текстурами

Как вы, наверное, заметили, все примеры из предыдущего раздела формируют кубическую текстуру в два прохода.

1. Формирование изображения грани кубической карты в буфере `pbuffer`.
2. Копирование полученного изображения из буфера `pbuffer` в кубическую текстурную карту с использованием команды `glCopyTexSubImage2D`.

С другой стороны, в *разд. 8.3* мы рассмотрели расширение ARB_render_texture, позволяющее программисту непосредственно использовать содержимое буфера `pbuffer` в качестве текстурной карты. Это позволяет избавиться от ненужного копирования текстуры из буфера `pbuffer` в видеопамять, а также немного сэкономить дефицитную видеопамять.

Вам может показаться, что изображения кубических текстурных карт можно формировать с использованием расширения ARB_render_texture аналогично двухмерным текстурам: для этого надо создать 6 буферов `pbuffer`, в каждый из которых будет занесена своя грань кубической текстуры. Но это не так. Вот почему: для показа кубической текстуры нам надо будет одновременно связать с целевой текстурой `GL_TEXTURE_CUBE_MAP_ARB` все 6 ее граней, т. е. связать с контекстом все 6 буферов `pbuffer`. А этого нельзя сделать — команда `wglBindTexImageARB` может единовременно связать с целевой текстурой только один буфер `pbuffer`.

Для разрешения этого противоречия расширение ARB_texture_cube_map дополняет расширения WGL_ARB_pbuffer и WGL_ARB_render_texture новыми возможностями.

- Атрибут `WGL_TEXTURE_TARGET_ARB` функции `wglCreatePbufferARB` теперь может принимать новое значение `WGL_TEXTURE_CUBE_MAP_ARB`, указывающее на то, что буфер `pbuffer` должен содержать изображения 6 граней кубической текстуры.
- Функция `wglSetPbufferAttribARB` обзавелась новым атрибутом `WGL_CUBE_MAP_FACE_ARB`, задающим текущую грань кубической карты буфера `pbuffer`, которую визуализирует OpenGL.

Таким образом, обновление изображения кубической текстурной карты, хранящейся в буфере `pbuffer`, осуществляется следующим образом:

1. Делаем текущим контекст буфера `pbuffer` (команда `wglMakeCurrent`).
2. Освобождаем `pbuffer` (команда `wglReleaseTexImageARB`).

3. Выбираем для визуализации грань кубической текстурной карты +X (команда `wglSetPbufferAttribARB` с атрибутом `WGL_CUBE_MAP_FACE_ARB` равным `WGL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB`).
4. Визуализируем изображение грани +X
5. Повторяем шаги 3 и 4 для граней −X, +Y, −Y, +Z и −Z.
6. Привязываем изображение текстурной карты к целевой текстуре `GL_TEXTURE_CUBE_MAP_ARB` (команда `wglBindTexImageARB`).

Однако работать с буфером `pbuffer` средствами OpenGL — задача не из приятных — одна инициализация буфера `pbuffer` занимает около 100 строк. Поэтому в *главе 8* мы разработали класс `PBuffer`, инкапсулирующий всю работу с буфером `pbuffer`. Но т. к. класс `PBuffer` не умеет работать с кубическими текстурными картами, мы немного модифицируем его, добавив поддержку кубических текстурных карт.

Для этого необходимо добавить в метод `Initialize` поддержку нового значения атрибута `WGL_TEXTURE_TARGET_ARB` команды `wglCreatePbufferARB` — `WGL_TEXTURE_CUBE_MAP_ARB`, а также определить новый битовый флаг `PBUFFER_TEXTURE_CUBE_MAP_ARB`, указывающий на то, что буфер `pbuffer` будет использоваться для хранения изображений граней кубической текстурной карты (листинг 10.13).

Листинг 10.13

```
const PBUFFER_TEXTURE_RGB=64;
const PBUFFER_TEXTURE_RGBA=128;
const PBUFFER_TEXTURE_1D=256;
const PBUFFER_TEXTURE_2D=512;
const PBUFFER_TEXTURE_MIPMAP=1024;
const PBUFFER_TEXTURE_CUBE_MAP_ARB=2048;
...
void PBuffer::Initialize(bool sharecontexts, bool sharelists)
{
...
// Список целочисленных атрибутов буфера pbuffer
       int properties[MAX_ATTRIBS*2];
       int niattribs=0;
// Обнуляем список атрибутов
    for ( int i = 0; i < 2*MAX_ATTRIBS; i++ )
    {
```

```
            properties[i] = 0;
    }
// Если указан флаг PBUFFER_TEXTURE_RGB, в буфере pbuffer будет храниться
// текстура формата RGB
      if (mode & PBUFFER_TEXTURE_RGB)
      {
            properties[2*niattribs]=WGL_TEXTURE_FORMAT_ARB;
            properties[2*niattribs+1]=WGL_TEXTURE_RGB_ARB;
            niattribs++;
      }
// Если указан флаг PBUFFER_TEXTURE_RGBA, в буфере pbuffer будет
// храниться текстура формата RGBA
      if (mode & PBUFFER_TEXTURE_RGBA)
      {
            properties[2*niattribs]=WGL_TEXTURE_FORMAT_ARB;
            properties[2*niattribs+1]=WGL_TEXTURE_RGBA_ARB;
            niattribs++;
      }
// Если указан флаг PBUFFER_TEXTURE_1D, в буфере pbuffer будет храниться
// одномерная текстура
      if (mode & PBUFFER_TEXTURE_1D)
      {
            properties[2*niattribs]=WGL_TEXTURE_TARGET_ARB;
            properties[2*niattribs+1]=WGL_TEXTURE_1D_ARB;
            niattribs++;
      }
// Если указан флаг PBUFFER_TEXTURE_2D, в буфере pbuffer будет храниться
// двухмерная текстура
      if (mode & PBUFFER_TEXTURE_2D)
      {
            properties[2*niattribs]=WGL_TEXTURE_TARGET_ARB;
            properties[2*niattribs+1]=WGL_TEXTURE_2D_ARB;
            niattribs++;
      }
// Если указан флаг PBUFFER_TEXTURE_CUBE_MAP_ARB, в буфере pbuffer будет
// храниться кубическая текстура
      if (mode & PBUFFER_TEXTURE_CUBE_MAP_ARB)
      {
            properties[2*niattribs]=WGL_TEXTURE_TARGET_ARB;
```

```
                properties[2*niattribs+1]=WGL_TEXTURE_CUBE_MAP_ARB;
                niattribs++;
        }
// Если указан флаг PBUFFER_TEXTURE_MIPMAP, в буфере pbuffer будет
// храниться текстура вместе со всеми Mipmap-уровнями
        if (mode & PBUFFER_TEXTURE_MIPMAP)
        {
                properties[2*niattribs]=WGL_MIPMAP_TEXTURE_ARB;
                properties[2*niattribs+1]=GL_TRUE;
                niattribs++;
        }
// Создаем pbuffer
    buffer = wglCreatePbufferARB( hdc, format, width, height,
properties);
...
};
```

Кроме того, для обеспечения комфортной работы с буфером `pbuffer` не помешает инкапсулировать команду `wglSetPbufferAttribARB`, используемую для выбора текущей грани кубической текстуры (листинг 10.14).

Листинг 10.14

```
// Битовые флаги-идентификаторы граней кубических текстур
const PBUFFER_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB=1;
const PBUFFER_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB=2;
const PBUFFER_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB=4;
const PBUFFER_TEXTURE_CUBE_MAP_POSITIVE_X_ARB=8;
const PBUFFER_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB=16;
const PBUFFER_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB=32;

// Задаем для визуализации грань кубической текстурной карты, хранящейся
// в pbuffer.
//     Attrib - набор битовых флагов
void PBuffer::SetPbufferAttribARB(const int Attribs)
{
// Массив атрибутов грани
        int niattribs=0;
        int iattributes[2*MAX_ATTRIBS];
```

```
        for ( int i = 0; i < 2*MAX_ATTRIBS; i++)
                iattributes[i]=0;

// Если битовый флаг равен PBUFFER_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB,
// делаем текущей грань -X
        if (Attribs & PBUFFER_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB)
        {
                iattributes[niattribs*2]=WGL_CUBE_MAP_FACE_ARB;

        iattributes[niattribs*2+1]=WGL_TEXTURE_CUBE_MAP_NEGATIVE_X_ARB;
                niattribs++;
        }
// Если битовый флаг равен PBUFFER_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB,
// делаем текущей грань -Y
        if (Attribs & PBUFFER_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB)
        {
                iattributes[niattribs*2]=WGL_CUBE_MAP_FACE_ARB;

        iattributes[niattribs*2+1]=WGL_TEXTURE_CUBE_MAP_NEGATIVE_Y_ARB;
                niattribs++;
        }
// Если битовый флаг равен PBUFFER_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB,
// делаем текущей грань -Z
        if (Attribs & PBUFFER_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB)
        {
                iattributes[niattribs*2]=WGL_CUBE_MAP_FACE_ARB;

        iattributes[niattribs*2+1]=WGL_TEXTURE_CUBE_MAP_NEGATIVE_Z_ARB;
                niattribs++;
        }
// Если битовый флаг равен PBUFFER_TEXTURE_CUBE_MAP_POSITIVE_X_ARB,
// делаем текущей грань +X
        if (Attribs & PBUFFER_TEXTURE_CUBE_MAP_POSITIVE_X_ARB)
        {
                iattributes[niattribs*2]=WGL_CUBE_MAP_FACE_ARB;

        iattributes[niattribs*2+1]=WGL_TEXTURE_CUBE_MAP_POSITIVE_X_ARB;
                niattribs++;
        }
```

```
// Если битовый флаг равен PBUFFER_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB,
// делаем текущей грань +Y
        if (Attribs & PBUFFER_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB)
        {
                iattributes[niattribs*2]=WGL_CUBE_MAP_FACE_ARB;

        iattributes[niattribs*2+1]=WGL_TEXTURE_CUBE_MAP_POSITIVE_Y_ARB;
                niattribs++;
        }
// Если битовый флаг равен PBUFFER_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB,
// делаем текущей грань +Z
        if (Attribs & PBUFFER_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB)
        {
                iattributes[niattribs*2]=WGL_CUBE_MAP_FACE_ARB;

        iattributes[niattribs*2+1]=WGL_TEXTURE_CUBE_MAP_POSITIVE_Z_ARB;
                niattribs++;
        }
// Задаем текущую грань кубической текстурной карты
        wglSetPbufferAttribARB(buffer, &iattributes[0]);
}
```

Располагая таким инструментом, как класс `PBuffer`, нам не составит большого труда добавить в пример Ex10 поддержку расширения ARB_render_texture (Ex11). Для этого фактически необходимо только изменить параметр у конструктора объекта `pbuffer0`, а также немного "подправить" функцию `MakeCubemapTexture` (листинг 10.15).

Листинг 10.15

```
// Создаем pbuffer для хранения кубических текстурных карт
PBuffer pbuffer0(512, 512, GLUT_RGB | GLUT_DEPTH | GLUT_SINGLE |
PBUFFER_TEXTURE_CUBE_MAP_ARB | PBUFFER_TEXTURE_RGB |
PBUFFER_TEXTURE_MIPMAP);


// Обновляем кубическую текстурную карту
void MakeCubemapTexture()
{
// Определяем матрицу поворота
        matrix4f m=get_matrix(GL_MODELVIEW_MATRIX);
        m(0,3)=0.0;
```

```
        m(1,3)=0.0;
        m(2,3)=0.0;
// Если pbuffer потерян, восстанавливаем его
        pbuffer0.HandleModeSwitch();
// Делаем текущим контекст pbuffer
        pbuffer0.MakeCurrent();
// Освобождаем pbuffer
        reflect_texture.bind();
        pbuffer0.ReleaseTexImage();
// Настраиваем видовое окно и матрицу проекции
        glViewport(0, 0, pbuffer0.GetWidth(), pbuffer0.GetHeight());
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(90, 1, 0.1, 100);
        glMatrixMode(GL_MODELVIEW);
// Обновляем грань +X кубической текстурной карты
// Делаем текущей грань +X
        pbuffer0.SetPbufferAttribARB(PBUFFER_TEXTURE_CUBE_MAP_POSITIVE_X_
ARB);
// Выводим изображение грани
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        gluLookAt(0, 0, 0, 1, 0, 0, 0, -1, 0);
        glMultMatrixf(&m(0, 0));
        DrawOutdoor();
// Принудительно выполняем все ранее вызванные команды. Если этого не
// сделать, то изображение грани кубической текстуры, скорее всего, будет
// сформировано с ошибками
        glFlush();
// Обновляем грань -X кубической текстурной карты
        pbuffer0.SetPbufferAttribARB(PBUFFER_TEXTURE_CUBE_MAP_NEGATIVE_X_
ARB);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        gluLookAt(0, 0, 0, -1, 0, 0, 0, -1, 0);
        glMultMatrixf(&m(0, 0));
        DrawOutdoor();
        glFlush();
```

```
// Обновляем грань +Y кубической текстурной карты
        pbuffer0.SetPbufferAttribARB(PBUFFER_TEXTURE_CUBE_MAP_POSITIVE_Y_
ARB);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        gluLookAt(0, 0, 0, 0, 1, 0, 0, 0, 1);
        glMultMatrixf(&m(0, 0));
        DrawOutdoor();
        glFlush();
// Обновляем грань -Y кубической текстурной карты
        pbuffer0.SetPbufferAttribARB(PBUFFER_TEXTURE_CUBE_MAP_NEGATIVE_Y_
ARB);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        gluLookAt(0, 0, 0, 0, -1, 0, 0, 0, -1);
        glMultMatrixf(&m(0, 0));
        DrawOutdoor();
        glFlush();
// Обновляем грань +Z кубической текстурной карты
        pbuffer0.SetPbufferAttribARB(PBUFFER_TEXTURE_CUBE_MAP_POSITIVE_Z_
ARB);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        gluLookAt(0, 0, 0, 0, 0, 1, 0, -1, 0);
        glMultMatrixf(&m(0, 0));
        DrawOutdoor();
        glFlush();
// Обновляем грань -Z кубической текстурной карты
        pbuffer0.SetPbufferAttribARB(PBUFFER_TEXTURE_CUBE_MAP_NEGATIVE_Z_
ARB);
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity();
        gluLookAt(0, 0, 0, 0, 0, -1, 0, -1, 0);
        glMultMatrixf(&m(0, 0));
        DrawOutdoor();
        glFlush();
// Переключаемся в главное окно
        glutSetWindow(WinID);
        reflect_texture.bind();
```

```
// Привязываем изображение из буфера pbuffer к текущему текстурному
// объекту
        pbuffer0.BindTexImage();
}
```

В заключение раздела мы улучшим пример Ex11, сделав все цветные чайники глянцевыми (рис. 10.26).

Рис. 10.26. Семь глянцевых чайников

В принципе, мы можем моделировать отражения на всех семи чайниках с использованием динамических кубических текстурных карт. Однако на практике это приведет к падению производительности в несколько раз. А если учесть, что пример Ex11 даже на ускорителе NVIDIA GeForce FX 5800 Ultra работает со скорость около 20 FPS, то становится ясно, что такой метод нам не подходит.

Следовательно, нам придется найти компромисс между качеством отражения и производительностью. Например, мы можем использовать динамическую кубическую текстурную карту только для центрального чайника, а на остальные чайники накладывать статическую кубическую текстурную карту. В результате корректное отражение будет только в центральном чайнике, а в

остальных чайниках будет отражаться только статическая окружающая среда (т. е. горная местность без чайников). В листинге 10.16 приведены наиболее важные фрагменты измененных функций `Init` и `DrawOutdoor`.

Листинг 10.16

```
...
// Изображения граней кубической карты
cubemapImages images={0, 0, 0, 0, 0, 0};
// Динамическая кубическая текстура, накладываемая на центральный чайник
tex_object_cube_map reflect_texture;
// Статическая кубическая текстура, накладываемая на цветные чайники
tex_object_cube_map static_reflect_texture;
// Текстуры граней куба окружающей среды
tex_object_2D walls_textures[6];
...
bool Init()
{
...
// Загружаем кубическую текстурную карту
       string info;
       if (!read("..\\cloudyhills.jpg", images, &info))
       {
              console.add(info);
              console.exit(-1);
              return false;
       }
// Создаем текстурные объекты динамической кубической текстуры
       reflect_texture.bind();
       reflect_texture.parameter(GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
       reflect_texture.parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
       reflect_texture.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
       reflect_texture.parameter(GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
       reflect_texture.parameter(GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
// Создаем текстурный объект статической кубической текстуры
       static_reflect_texture.bind();
       static_reflect_texture.parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
```

```
        static_reflect_texture.parameter(GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
        static_reflect_texture.parameter(GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
        static_reflect_texture.parameter(GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);
// Загружаем статическую кубическую текстурную карту
        texImageCubemap(images);
// Создаем двухмерные текстуры для граней куба
        for (int i=0; i<6; i++)
        {
                walls_textures[i].bind();


                walls_textures[i].parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
                walls_textures[i].parameter(GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
                walls_textures[i].parameter(GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);
                walls_textures[i].parameter(GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);


                gluBuild2DMipmaps(walls_textures[i].target, images[i]->
components, images[i]->width,
                                                images[i]->height,
images[i]->format, GL_UNSIGNED_BYTE,
                                                images[i]->pixels);
        }
...
}


// Рисуем окружающую среду
void DrawOutdoor()
{
// Настраиваем параметры OpenGL и источники света
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        glDisable(GL_CULL_FACE);
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_LIGHT1);
        glEnable(GL_LIGHT2);
```

```
        glEnable(GL_COLOR_MATERIAL);
        glEnable(GL_DEPTH_TEST);

        glLightfv(GL_LIGHT0, GL_DIFFUSE, &light0_diffuse.v[0]);
        glLightfv(GL_LIGHT1, GL_DIFFUSE, &light1_diffuse.v[0]);
        glLightfv(GL_LIGHT2, GL_DIFFUSE, &light2_diffuse.v[0]);
        glLightfv(GL_LIGHT0, GL_AMBIENT, &light_ambient.v[0]);
        glLightfv(GL_LIGHT0 ,GL_POSITION, &light0_pos.v[0]);
        glLightfv(GL_LIGHT1 ,GL_POSITION, &light1_pos.v[0]);
        glLightfv(GL_LIGHT2 ,GL_POSITION, &light2_pos.v[0]);
        psCamera camera;
// Привязываем текстурный объект к целевой текстуре
        static_reflect_texture.bind();
// Включаем наложение текстуры
        static_reflect_texture.enable();
// Задаем режим наложения текстуры
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_ADD);
// Задаем режим генерации текстурных координат
        glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_ARB);
        glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_ARB);
        glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_ARB);
// Включаем генерацию текстурных координат
        glEnable(GL_TEXTURE_GEN_S);
        glEnable(GL_TEXTURE_GEN_T);
        glEnable(GL_TEXTURE_GEN_R);
// Задаем текстурную матрицу для корректного наложения отражения
// (Эта матрица является общей для всех цветных чайников)
        glMatrixMode(GL_TEXTURE);
        glPushMatrix();
        glLoadIdentity();
// Получаем текущую матрицу модели
        matrix4f m=get_matrix(GL_MODELVIEW_MATRIX);
// Вычисляем обратную матрицу
        m=m.inverse();
// Обнуляем коэффициенты матрицы, отвечающие за перенос
        m(0,3)=0.0;
        m(1,3)=0.0;
        m(2,3)=0.0;
```

```
// Умножаем текущую текстурную матрицу на полученную
        glMultMatrixf(&m(0, 0));
        glMatrixMode(GL_MODELVIEW);
// Рисуем вращающиеся разноцветные чайники
        for (unsigned int i=0; i<TeapotCount; i++)
        {
                glPushMatrix();
                glTranslatef(Teapots[i].pos.x, Teapots[i].pos.y, Tea-
pots[i].pos.z);
                glRotatef(Teapots[i].rotate, 0, 1.0, 0);
                camera.psGetPositionGL();
                if (!camera.psTest(bound))
                {
                        glColor3fv(&Teapots[i].color.x);
                        teapot.call_list();
                }
                glPopMatrix();
        }

// Восстанавливаем настройки OpenGL
        glMatrixMode(GL_TEXTURE);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
        static_reflect_texture.disable();
        glDisable(GL_TEXTURE_GEN_S);
        glDisable(GL_TEXTURE_GEN_T);
        glDisable(GL_TEXTURE_GEN_R);
// Рисуем куб с окружающей средой
        walls.call_list();
        glPopAttrib();
}
```

Для моделирования красивых ярких разноцветных чайников мы накладываем на них текстуру отражения в режиме `GL_ADD`. Однако если оставить код, отвечающий за генерацию цвета чайников без изменений, чайники будут смотреться слишком яркими. Кроме того, не исключен вариант, когда на экране будут находиться несколько чайников одного цвета. Все это влияет на внешний вид программы не самым лучшим образом, поэтому для ис-

правления этих недостатков мы немного модифицируем фрагмент функции Init, отвечающий за генерацию цветов чайников:

```
       bool test;
// Перебираем все чайники
       for (int i=0; i<TeapotCount; i++)
       {
              Teapots[i].pos.y=0;
              do
              {
// Генерируем случайный цвет чайника
                     Teapots[i].color.x=nv_random()/2.0+0.5;
                     Teapots[i].color.y=nv_random()/2.0+0.5;
                     Teapots[i].color.z=nv_random()/2.0+0.5;
                     test=true;
// Проверяем, нет ли других чайников похожего цвета
                     for (int j=0; j<i-1; j++)
                            if (fabs((Teapots[i].color.x-
Teapots[j].color.x)*
                                          (Teapots[i].color.y-
Teapots[j].color.y)*
                                              (Teapots[i].color.z-
Teapots[j].color.z))<0.001)
                            {
                                   test=false;
                                   break;
                            };
// Является ли цвет слишком ярким
                     if
(Teapots[i].color.x+Teapots[i].color.y+Teapots[i].color.z>0.7)
                            test=false;
// Является ли цвет слишком темным
                     if
(Teapots[i].color.x+Teapots[i].color.y+Teapots[i].color.z<0.2)
                            test=false;
// Перебираем цвета до тех пор, пока не найдем подходящий
              } while (!test);
       }
```

Теперь разноцветные чайники стали значительно красивее. Тем не менее пример Ex12 можно еще улучшить.

Вот только небольшой список возможных улучшений:

- предусмотреть возможность отключать автоматическую генерацию Mipmap-уровней, значительно снижающую производительность старых видеокарт;
- предоставить пользователям возможность управлять частотой обновления динамической кубической текстурной карты. Например, обновление кубической карты только через кадр в большинстве случаев приводит к полукратному приросту производительности, что очень актуально для устаревших видеокарт;
- обновлять только те грани кубической текстуры, изображение которых действительно изменилось. Это особенно актуально при включенной автоматической генерации Mipmap-уровней.

Думаю, вам не составит большого труда самостоятельно добавить эти возможности в пример Ex12.

В этой книге мы уже много раз убеждались в том, что для эффективного использования любой технологии необходимо знать не только ее достоинства, но и недостатки. Поэтому давайте попробуем разобраться, насколько соответствует отражение на центральном чайнике настоящему, физически корректному отражению. С первого взгляда кажется, что все вроде бы нормально: чайник как чайник, в нем отражается окружающая среда, включая остальные глянцевые чайники. Но действительно ли в нем отражается то, что надо?

Давайте внимательно рассмотрим окрестности носика чайника (рис. 10.27).

Первое, что бросается в глаза, — в корпусе чайника не отражается его собственный носик. В этом нет ничего удивительного — кубические карты в принципе не могут использоваться для моделирования эффекта самоотражения, т. к. во время формирования кубической карты сам глянцевый объект не визуализируется. Следствием этого является физическая некорректность отражения в тех частях объекта, в которых должен наблюдаться эффект самоотражения. Например, мы видим отражение гор и цветных чайников на внутренней поверхности носика центрального чайника, в то время как они принципиально не могут там отражаться.

Итак, кубические карты не могут моделировать эффект самоотражения, или, иными словами, они подходят для моделирования только тех глянцевых объектов, на поверхности которых не наблюдается эффекта самоотражения. Такому условию удовлетворяют только выпуклые фигуры. Следовательно, кубические карты могут физически корректно моделировать отражения только на выпуклых объектах. Если же объект не является выпуклым, то его необходимо "разрезать" на несколько выпуклых объектов, и на каждый из них наложить собственную кубическую текстуру.

Рис. 10.27. Носик глянцевого чайника

Однако не каждый геометрический объект может быть разбит на небольшое число выпуклых объектов. Примером такого объекта является тор: чем больше детализация тора, тем на большее количество выпуклых объектов он будет разбит. Следствием этого является неприспособленность кубических текстур для моделирования отражения на торе: в этом случае количество выпуклых объектов и, соответственно, кубических текстурных карт превзойдет все разумные пределы.

Итак, мы пришли к выводу, что отражение на чайнике не является физически корректным из-за того, что чайник не является выпуклым объектом. Но единственная ли это причина? Будет ли отражение физически корректным, если мы заменим чайник, например, на сферу (рис. 10.28)?

Несмотря на то, что "странности", связанные с отсутствием эффекта самоотражения, пропали, отражение на шаре выглядит каким-то неестественным. Что-то не так: отражения чайников какие-то неестественно маленькие — в реальности отражения чайников должны быть, как минимум, в два раза больше. Но с чем это связано? Все очень просто: при формировании изображения кубической текстурной карты мы "фотографируем" окружающую среду из *центра* шара. Иным словами, кубическая текстурная карта, накладываемая на наш большой шар, сформирована для шарика бесконечно малого размера. А поскольку в нашем случае размер шара примерно равен расстоянию от поверх-

ности шара до чайника, то при формировании отражения OpenGL считает, что расстояние между поверхностью шара и чайником в два раза больше, чем на самом деле. Поэтому в том, что отражения чайников в два раза меньше, чем в реальности, нет ничего удивительного. В принципе этот эффект влияет и на отражение куба с окружающей средой (горной местности). Но из-за того, что размеры шара на порядок меньше размеров куба с окружающей средой, отражение куба практически не искажается.

Рис. 10.28. Глянцевая сфера

Из всего этого можно сделать следующий вывод: кубические текстурные карты подходят для формирования физически корректного отражения только тех объектов, расстояние до которых значительно больше геометрических размеров глянцевого объекта.

В заключение я хотел бы еще раз напомнить вам о том, что OpenGL формирует отражение окружающей среды на основе нормалей объекта. Поэтому если нормали к объекту рассчитаны неверно, то и отражение будет выглядеть некорректно. К сожалению, наглядным примером того, как не надо рассчитывать нормали, является чайник библиотеки GLUT. На рис. 10.29 крупным планом показано дно центрального глянцевого чайника примера Ex12.

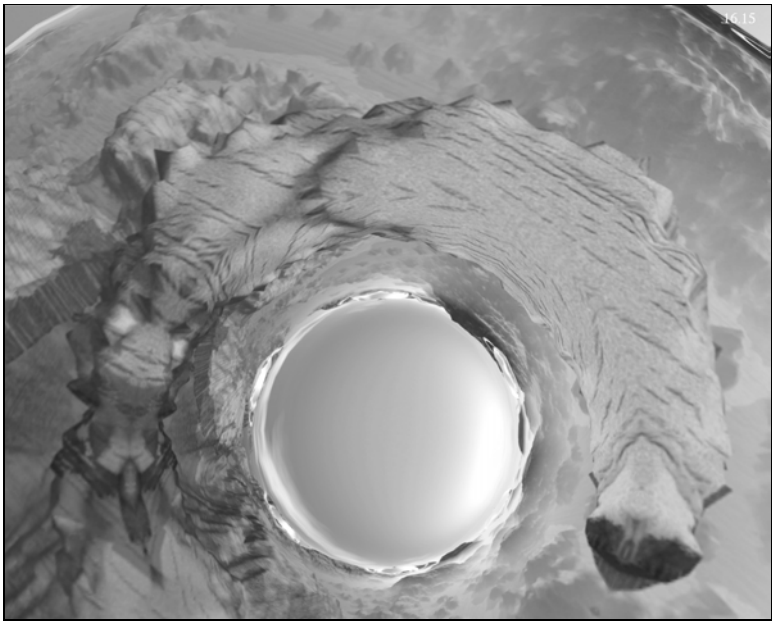


Рис. 10.29. Дно центрального глянцевого чайника

Первое, что бросается в глаза, это наличие в центре дна непонятной "черной дыры", в которой отражаются небо и цветные чайники. Они в принципе не могут там отражаться — это противоречит всем законам оптики (но не OpenGL). Все дело в том, что из-за ошибки GLUT нормаль в центре чайника повернута на 180 градусов. А поскольку эта нормаль имеет примерно такое же направление, как и нормали на крышке чайника, нет ничего удивительного в том, что мы видим в центре дна чайника отражение, аналогичное тому, что находится на его крышке.

Итак, пора подвести итоги. Глянцевые чайники примера Ex12, формируемые с использованием кубических карт, имеют мало общего с реальностью. Что же нам делать — отказаться от формирования отражения с использованием кубических карт? Конечно же, нет! Кубические карты позволяют значительно улучшить внешний вид программы, выводя его совершенно на другой уровень. Просто следует помнить о том, что кубические текстурные карты используются не для моделирования физически корректного отражения с учетом всех законов оптики. Задача кубических карт — создать у пользователя иллюзию того, что он видит отражение окружающей среды, а на сколько оно соответствует действительности — это совершенно не важно.

Кубические текстурные карты — это очень мощный инструмент, который может использоваться не только для формирования отражений, но и для многих других целей. В следующем разделе мы рассмотрим пример нетра-

диционного использования кубических текстур для закраски объектов по методу Фонга в реальном времени.

10.3. Нетрадиционное использование кубических карт на примере закраски методом Фонга

Как известно, OpenGL использует вершинную модель освещения, называемую также интерполяционным закрашиванием по методу Гуро. Эта модель освещения довольно сносно моделирует освещение, когда интенсивность освещения изменяется линейно вдоль поверхности объекта, т. е. тогда, когда источник света находится на большом расстоянии от объекта и не создает бликов. В противном случае будет наблюдаться неприятное явление "дрожания" блика при его движении вдоль поверхности объекта — яркость и форма блика будут зависеть от того, как далеко он находится от ближайшей вершины объекта. А если блик не будет "касаться" ни одной вершины, то он вообще не будет виден.

Для демонстрации всего вышесказанного мы напишем небольшую программу, рисующую на экране полированный чайник, освещаемый тремя разноцветными источниками света (рис. 10.30).

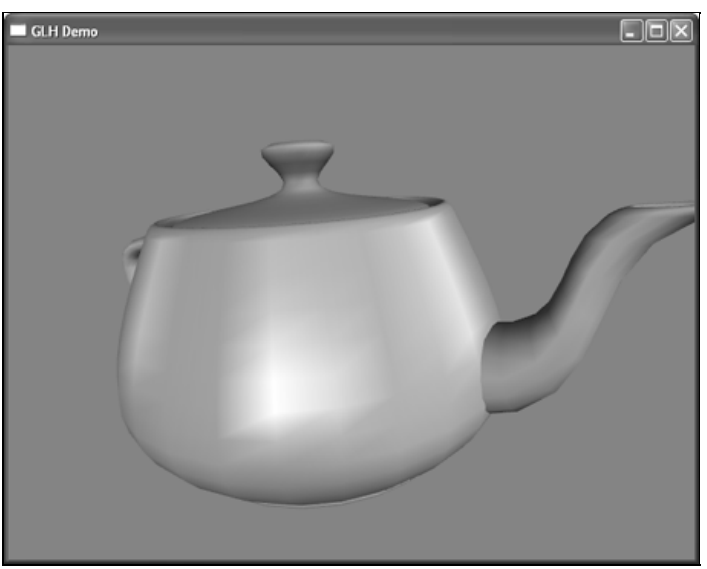


Рис. 10.30. Полированный чайник, освещенный тремя разноцветными источниками света. Обратите внимание, что бликов практически не видно — в их окрестностях отчетливо вырисовываются стыки полигонов

Так как этот пример (Ex13) не содержит ничего необычного, я сразу приведу исходный текст программы с подробными комментариями (листинг 10.17).

Листинг 10.17

```
...
// Цвета источников света
vec4f light0_color(0.7, 0.7, 0.7, 1);
vec4f light1_color(0.0, 0.5, 0.2, 1);
vec4f light2_color(0.3, 0.1, 0.5, 1);
// Положения источников света
vec4f light0_pos(0, 0, 1, 0);
vec4f light1_pos(8, 3, 1, 0);
vec4f light2_pos(-10, 2, 1, 0);
// Глянцевитость материала
GLfloat mat_shininess=128;
// Цвет зеркальных бликов материала
vec4f mat_specular(1.0, 1.0, 1.0, 1);
// Инициализация OpenGL
void Init()
{
       glEnable(GL_DEPTH_TEST);
// Включаем все три источника света
       glEnable(GL_LIGHTING);
       glEnable(GL_LIGHT0);
       glEnable(GL_LIGHT1);
       glEnable(GL_LIGHT2);
// Задаем положение источников света
       glLightfv(GL_LIGHT0, GL_POSITION, &light0_pos[0]);
       glLightfv(GL_LIGHT1, GL_POSITION, &light1_pos[0]);
       glLightfv(GL_LIGHT2, GL_POSITION, &light2_pos[0]);
// Задаем диффузные составляющие источников света
       glLightfv(GL_LIGHT0, GL_DIFFUSE,  &light0_color[0]);
       glLightfv(GL_LIGHT1, GL_DIFFUSE,  &light1_color[0]);
       glLightfv(GL_LIGHT2, GL_DIFFUSE,  &light2_color[0]);
// Задаем зеркальные составляющие источников света
       glLightfv(GL_LIGHT0, GL_SPECULAR,  &light0_color[0]);
       glLightfv(GL_LIGHT1, GL_SPECULAR,  &light1_color[0]);
       glLightfv(GL_LIGHT2, GL_SPECULAR,  &light2_color[0]);
```

```
// Задаем параметры материала
        glMaterialfv(GL_FRONT_AND_BACK, GL_SHININESS, &mat_shininess);
        glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, &mat_specular[0]);
        glClearColor(0.5, 0.5, 0.75, 1);
}
void Display()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Рисуем чайник
        glutSolidTeapot(2);
}
```

Как видно из рис. 10.30, блики на поверхности чайника практически отсутствуют — вместо них в окрестностях поверхности чайника, там, где должны быть блики, видны лишь яркие границы полигонов. Причина этого явления уже обсуждалась выше — закраска по методу Гуро в принципе не позволяет моделировать реалистичные блики.

Выходом из этой ситуации является использование иной модели освещения, к примеру, закраски по методу Фонга, когда интерполяция цвета между вершинами объекта заменяется интерполяцией направления нормалей. Но в этом случае освещение придется рассчитывать для каждого пиксела объекта, что отрицательно скажется на быстродействии. Однако если источник света статический, то задачу можно сильно упростить.

Так как яркость бликов зависит исключительно от направления отраженного вектора взгляда наблюдателя от поверхности, мы можем построить таблицу зависимости яркостей бликов от направления вектора отражения. Для хранения такой таблицы идеально подходит кубическая текстура.

Теперь давайте подумаем, что произойдет, если мы включим наложение кубической текстурной карты и автоматическую генерацию текстурных координат (режим `GL_REFLECTION_MAP_ARB`). В этом случае ускоритель будет автоматически находить тексель текстуры, на который указывает отраженный вектор взгляда наблюдателя от поверхности. А ведь это именно то, что нам нужно. Обратите внимание на одну деталь. Координаты кубической текстуры рассчитываются только в вершинах объекта, а в остальных точках экрана они лишь интерполируются. Но поскольку эти координаты задают вектор отражения, такая интерполяция означает интерполяцию направления вектора отражения, что в свою очередь может трактоваться как интерполяция направления вектора нормали. То есть мы фактически выполняем закраску по методу Фонга.

Из всего вышесказанного получаем следующий вывод: поместив таблицу зависимости яркости блика от направления вектора отражения в кубиче-

скую текстурную карту, мы сможем выполнить закраску методом Фонга без использования трудоемких вычислений. Дело за небольшим — создать такую кубическую карту.

В нашем случае OpenGL рассчитывает цвет блика по следующей формуле[5]:

$$color = light\_color \cdot \cos\alpha^{shininess},$$

где:

- *color* — итоговый цвет точки (трехмерный или четырехмерный вектор);
- *light_color* — цвет источника света (трехмерный или четырехмерный вектор);
- cosα — косинус угла между вектором отражения взгляда наблюдателя от поверхности объекта и вектором, направленным из текущей точки на источник света. Если cosα меньше 0, то OpenGL полагает его равным 0;
- *shininess* — коэффициент глянцевитости объекта.

Написание функции расчета цвета блика по этой формуле не составит большого труда.

```
// Краткое описание параметров функции
// dir - вектор отражения взгляда наблюдателя от поверхности объекта
// light_vec - вектор, направленный из текущей точки объекта на источник
// света
// color - цвет источника света
// shininess - коэффициент глянцевитости объекта
vec4f calc_color(vec3f dir, vec4f light_vec, vec4f color, float
shininess)
{
// Нормализуем векторы
       dir.normalize();
       light_vec.normalize();
// Находим косинус угла между векторами
       float cosa=dir.dot(vec3f(&light_vec[0]));
// Если косинус меньше 0, считаем его равным 0
       if (cosa<0)
              cosa=0;
// Рассчитываем цвет точки
       vec4f result= color* powf(cosa, shininess);
       return result;
}
```

[5] Приведен очень упрощённый вариант формулы расчёта освещения. Полную формулу, используемую OpenGL при расчете освещения, можно найти в [6] или [9].

Формирование кубической текстурной карты осуществляется путем перебора всех текселей кубической текстуры с последующим вызовом функции `calc_color` для каждого текселя. В принципе мы могли бы сами написать функцию, выполняющую эти действия. Однако зачем изобретать велосипед — в составе библиотеки OpenGL Helper Library (GLH) уже имеется инструментарий для генерации кубических текстур, расположенный в файле glh_cube_map.h.

В основе этого инструментария есть функция `make_cube_map`, генерирующая кубическую текстурную карту на основе экземпляра пользовательского класса. После генерации кубической карты функция `make_cube_map` автоматически заносит ее в видеопамять с использованием команды `glTexImage2D`.

```
template <class FunctionOfDirection>
void make_cube_map(FunctionOfDirection & f, GLenum internal_format,
                   int size, int level = 0)
```

где:

- `FunctionOfDirection` — пользовательский класс, реализующий расчет цвета текселя кубической текстурной карты на основе трехмерного вектора направления;
- `f` — экземпляр класса `FunctionOfDirection`;
- `internal_format` — внутренний формат текстуры;
- `size` — размер текстуры в текселях;
- `level` — номер генерируемого Mipmap-уровня.

Пользовательский класс `FunctionOfDirection` представляет собой произвольный класс, в котором обязательно должны присутствовать следующие компоненты:

```
struct user_calc
{
// Определение пользовательского типа, который будет использоваться для
// хранения цветовых компонентов текселей текстуры. В нашем случае это
// GLfloat
       typedef GLfloat Type;
// Количество цветовых компонентов в текстуре (например, 3)
       int components;
// Внутренний формат текселей текстуры (например, GL_FLOAT)
       GLenum type;
// Формат изображения текстуры
       GLenum format;
```

```
// Оператор (), принимающий в качестве параметра трехмерный вектор
// направления и указатель на массив Type[components], в который
// необходимо занести рассчитанный цвет текселя для данного направления
        void operator() (const vec3f & v, Type * t)
        {
        }
// Прочие поля и методы класса (могут быть любыми)
};
```

Итак, нам необходимо создать свой собственный класс, рассчитывающий цвет текселя кубической карты в зависимости от вектора трехмерного направления. Иными словами, нам просто необходимо написать обыкновенную оболочку для функции `calc_color` (листинг 10.18).

Листинг 10.18

```
struct calc_lighting_cubemap
{
// Определение пользовательского типа, который будет использоваться для
// хранения цветовых компонентов текселей текстуры. В нашем случае это
// GLfloat
        typedef GLfloat Type;
// Количество цветовых компонентов в текстуре
        int components;
// Внутренний формат текселей текстуры
        GLenum type;
// Формат изображения текстуры
        GLenum format;
// Глянцевитость материала
        GLenum shininess;
// Конструктор, принимающий в качестве параметра глянцевитость материала
        calc_lighting_cubemap(float shine)
        {
                components=3;
                type=GL_FLOAT;
                format=GL_RGB;
                shininess=shine;
        }
```

```
// Оператор (). Для упрощения программы, используются глобальные
// переменные light0_pos, light0_color и т. д.
       void operator() (const vec3f & v, Type * t)
       {
// Рассчитываем цвет текселя как сумму цветов бликов трех источников
// света
              vec4f color=calc_color(v, light0_pos, light0_color,
shininess)+
                                     calc_color(v, light1_pos,
light1_color, shininess)+
                                     calc_color(v, light2_pos,
light2_color, shininess);
// Возвращаем результат
              t[0]=color[0];
              t[1]=color[1];
              t[2]=color[2];
       }
};
```

Теперь нам остается немного подправить текст главного модуля (main.cpp) примера Ex13, и программа будет готова (листинг 10.19).

Листинг 10.19

```
...
// Кубическая текстурная карта с бликами
tex_object_cube_map specular_cubemap;
// Параметры источников света и материала
vec4f light0_color(0.7, 0.7, 0.7, 1);
vec4f light1_color(0.0, 0.5, 0.2, 1);
vec4f light2_color(0.3, 0.1, 0.5, 1);
vec4f light0_pos(0, 0, 1, 0);
vec4f light1_pos(8, 3, 1, 0);
vec4f light2_pos(-10, 2, 1, 0);
GLfloat mat_shininess=128;
vec4f mat_specular(1.0, 1.0, 1.0, 1);
// Функция расчета освещения calc_color
vec4f calc_color(vec3f dir, vec4f light_vec, vec4f color, float
shininess)
```

```
{
...
}
// Класс-оболочка для функции calc_color
struct calc_lighting_cubemap
{
...
};
// Инициализирует OpenGL
void Init()
{
        glEnable(GL_DEPTH_TEST);
        glPixelStorei(GL_PACK_ALIGNMENT, 1);
        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
// Включаем расчет освещения
        glEnable(GL_LIGHTING);
        glEnable(GL_LIGHT0);
        glEnable(GL_LIGHT1);
        glEnable(GL_LIGHT2);
// Настраиваем параметры источников света
        glLightfv(GL_LIGHT0, GL_POSITION, &light0_pos[0]);
        glLightfv(GL_LIGHT1, GL_POSITION, &light1_pos[0]);
        glLightfv(GL_LIGHT2, GL_POSITION, &light2_pos[0]);
        glLightfv(GL_LIGHT0, GL_DIFFUSE,  &light0_color[0]);
        glLightfv(GL_LIGHT1, GL_DIFFUSE,  &light1_color[0]);
        glLightfv(GL_LIGHT2, GL_DIFFUSE,  &light2_color[0]);
        glClearColor(0.5, 0.5, 0.75, 1);
// Создаем кубическую текстуру
        specular_cubemap.bind();
// Настраиваем параметры фильтрации и т. д.
        specular_cubemap.parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        specular_cubemap.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        specular_cubemap.parameter(GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        specular_cubemap.parameter(GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
// Включаем автоматическую генерацию Mipmap-уровней
        specular_cubemap.parameter(GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
// Создаем экземпляр класса calc_lighting_cubemap для расчета кубической
// карты бликов
        calc_lighting_cubemap calc_specular_cubemap(mat_shininess);
```

```
// Создаем кубическую текстуру разрешением 256×256
        make_cube_map(calc_specular_cubemap, GL_RGB, 256, 0);
}
// Обновляем содержимое экрана
void Display()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Задаем режим наложения текстуры
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_ADD);
// Включаем автоматическую генерацию текстурных координат отражения
        glEnable(GL_TEXTURE_GEN_S);
        glEnable(GL_TEXTURE_GEN_T);
        glEnable(GL_TEXTURE_GEN_R);
        glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_ARB);
        glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_ARB);
        glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_ARB);
// Включаем наложение кубической текстурной карты
        specular_cubemap.bind();
        specular_cubemap.enable();
// Рисуем чайник
        glutSolidTeapot(2);
// Восстанавливаем старые параметры
        specular_cubemap.disable();
        glDisable(GL_TEXTURE_GEN_S);
        glDisable(GL_TEXTURE_GEN_T);
        glDisable(GL_TEXTURE_GEN_R);
}
```

Хотя в примере Ex14 используется комбинированное освещение, разница в качестве изображения по сравнению с примером Ex13 видна невооруженным глазом (рис. 10.31). В примере Ex14 диффузное освещение рассчитывается средствами OpenGL (т. е. с использованием закраски Гуро), а блики — при помощи кубической текстурной карты (т. е. по Фонгу).

Однако мы можем пойти еще дальше, полностью отказавшись от расчета освещения средствами OpenGL. В примере Ex14 OpenGL рассчитывает диффузное освещение по следующей формуле:

$$color = light\_color \cdot \cos\alpha,$$

где:

- *color* — итоговый цвет точки (трехмерный или четырехмерный вектор);

- *light_color* — цвет источника света (трехмерный или четырехмерный вектор);
- cosα — косинус угла между вектором нормали и вектором, направленным из текущей точки на источник света. Если cosα меньше 0, то OpenGL полагает его равным 0.

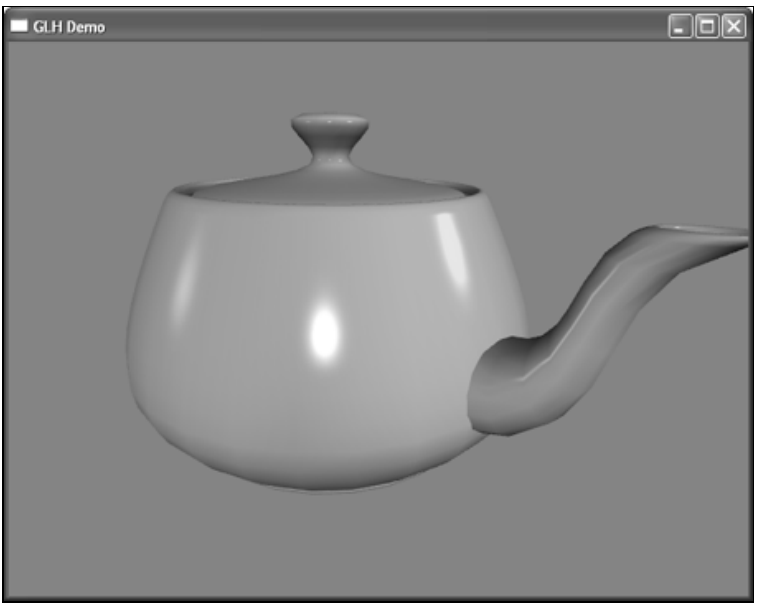


Рис. 10.31. Полированный чайник, освещенный тремя цветными источниками света. Используется комбинированное освещение

Таким образом, создание кубической текстурной карты диффузного освещения отличается от бликов только тем, что cosα не возводится в степень. Следовательно, мы можем генерировать кубическую текстурную карту с использованием класса `calc_lighting_cubemap`, положив коэффициент глянцевитости `shininess` равным 1. Кроме того, перед наложением кубической текстурной карты необходимо включить автоматическую генерацию текстурных координат в режиме `GL_NORMAL_MAP_ARB` вместо режима `GL_REFLECTION_MAP_ARB`. В листинге 10.20 приведены наиболее важные фрагменты примера Ex15, выполняющего полную закраску по методу Фонга.

Листинг 10.20

```
// Инициализация OpenGL
void Init()
```

```
{
        glEnable(GL_DEPTH_TEST);
        glPixelStorei(GL_PACK_ALIGNMENT, 1);
        glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
        glClearColor(0.5, 0.5, 0.75, 1);
// Формируем кубическую текстурную карту бликов
        specular_cubemap.bind();
        specular_cubemap.parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        specular_cubemap.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        specular_cubemap.parameter(GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        specular_cubemap.parameter(GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        specular_cubemap.parameter(GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
        calc_lighting_cubemap calc_specular_cubemap(mat_shininess);
        make_cube_map(calc_specular_cubemap, GL_RGB, 256, 0);
// Формируем кубическую текстурную карту с диффузным освещением
        diffuse_cubemap.bind();
        diffuse_cubemap.parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
        diffuse_cubemap.parameter(GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        diffuse_cubemap.parameter(GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        diffuse_cubemap.parameter(GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        diffuse_cubemap.parameter(GL_GENERATE_MIPMAP_SGIS, GL_TRUE);
        calc_lighting_cubemap calc_diffuse_cubemap(1.0);
        make_cube_map(calc_diffuse_cubemap, GL_RGB, 256, 0);
}
void Display()
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// Задаем режим наложения текстуры
        glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);

// Включаем автоматическую генерацию текстурных координат
        glEnable(GL_TEXTURE_GEN_S);
        glEnable(GL_TEXTURE_GEN_T);
        glEnable(GL_TEXTURE_GEN_R);
// Задаем режим автогенерации GL_NORMAL_MAP_ARB для наложения
// кубической текстурной картой с диффузным освещением
        glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP_ARB);
```

```
        glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP_ARB);
        glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_NORMAL_MAP_ARB);
// Рисуем чайник с кубической текстурной картой диффузного освещения
        diffuse_cubemap.bind();
        diffuse_cubemap.enable();
        glutSolidTeapot(2);
        diffuse_cubemap.disable();
// Второй проход. Рисуем чайник с кубической текстурной картой бликов
// поверх первого чайника с диффузной кубической текстурной картой
        glDepthFunc(GL_LEQUAL);
        glBlendFunc(GL_ONE, GL_ONE);
        glEnable(GL_BLEND);
        glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_ARB);
        glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_ARB);
        glTexGeni(GL_R, GL_TEXTURE_GEN_MODE, GL_REFLECTION_MAP_ARB);
        specular_cubemap.bind();
        specular_cubemap.enable();
        glutSolidTeapot(2);
        specular_cubemap.disable();
        glDepthFunc(GL_LESS);
        glDisable(GL_BLEND);
        glDisable(GL_TEXTURE_GEN_S);
        glDisable(GL_TEXTURE_GEN_T);
        glDisable(GL_TEXTURE_GEN_R);
}
```

Качество изображения еще немного улучшилось, правда, за это улучшение пришлось заплатить высокую цену — теперь чайник визуализируется за два прохода, что соответственным образом влияет на производительность (разумеется, не в лучшую сторону). Поэтому в большинстве случаев разумнее всего ограничиться закраской по методу Фонга только бликов, оставив диффузное освещение на откуп OpenGL.

В заключение раздела попробуйте ответить на простой вопрос: всегда ли кубические текстурные карты позволяют моделировать физически корректные блики на объектах? Ответ — нет. Так как наложение кубических текстурных карт бликов практически ничем не отличается от наложения кубических текстурных карт отражения. Методы наложения кубических текстурных карт бликов и отражения имеют общие недостатки. В частности, при формировании бликов с использованием кубических текстурных карт ускоритель не может оценить расстояние от поверхности объекта до источ-

ника света. В результате чего кубические текстурные карты могут корректно формировать отражения только от бесконечно удаленных источников. В противном случае блики уже не будут физически корректными, однако внешний вид этих бликов будет на порядок лучше, чем у физически корректных бликов, рассчитанных средствами базовой версии OpenGL. Что важнее в конкретном случае: физическая корректность или красота изображения — решать вам.

10.4. Экспорт из 3D Studio MAX материалов, использующих текстурные карты отражения reflect/refract

В этом разделе мы научимся экспортировать из 3D Studio MAX сцены, содержащие материалы, использующие текстурные карты отражения reflect/refract.

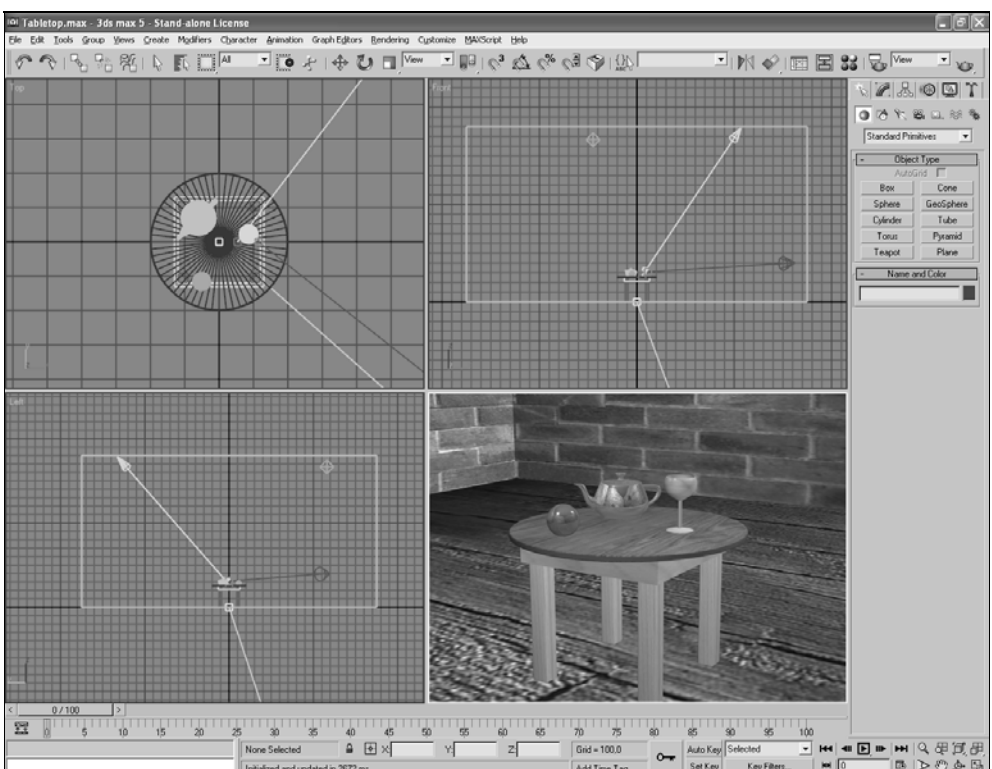


Рис. 10.32. Сцена с тремя глянцевыми объектами

В качестве примера мы попробуем экспортировать небольшую сцену, состоящую из небольшой комнаты, в которой находится стол с расположенными на нем тремя глянцевыми объектами: чайником, сферой и стаканом (рис. 10.32). Отражения моделируются при помощи текстурной карты reflect/refract. Вы можете найти эту сцену на CD, прилагаемом к книге, в каталоге \Examples\Ch10-Cubemap\3DSMAX_MODELS\Room.

Экспортировать саму сцену с использованием библиотеки ASE Reader не составит большого труда. А вот с экспортом материалов, содержащих карты отражения, все намного сложнее. Дело в том, что 3D Studio MAX помещает в файл формата ASE только информацию о том, что материал содержит текстурную карту отражения. При этом сама карта отражения не экспортируется. В результате мы оказываемся перед выбором:

- разработать свой инструментарий, генерирующий кубические текстурные карты для материалов reflect/refract, хранящихся в файле формата ASE;
- использовать средства, встроенные в 3D Studio MAX.

Мы не будем изобретать велосипед и воспользуемся инструментарием из 3D Studio MAX. Рассмотрим процесс генерации кубической текстурной карты на примере сферы.

Для начала необходимо открыть редактор материалов (клавиша <M>) и выделить материал сферы **Ball** (рис. 10.33).

Откройте текстурную карту отражения refract/reflect и щелкните по кнопке **To File** (рис. 10.34).

В открывшемся диалоговом окне введите имя файла, в который будет сохранена кубическая карта (например, ball.tga) и нажмите кнопку **Save**. Остается только нажать на кнопку **Pick Object And Render Maps** и выбрать объект, на который будет наложен этот материал. После этого 3D Studio MAX автоматически сгенерирует шесть файлов: ball_RT.tga, ball_LF.tga, ball_BK.tga, ball_FR.tga, ball_UP.tga и ball_DN.tga, содержащие изображения граней +X, −X, +Y, −Y, +Z и −Z кубической текстуры. Аналогичным образом получаем кубические текстуры для материала чайника и стакана.

Итак, у нас есть изображения граней кубических текстур. Но откуда мы узнаем, какие файлы соответствуют текстурной карте отражения данного материала? Ведь в описании материала указывается только то, что данный материал использует текстурную карту отражения и коэффициент прозрачности отражения. Про то, где и как хранится кубическая карта, там нет никакой информации.

Поэтому мы должны принять некоторое соглашение о соответствии между параметрами материала и названиями файлов, в которых хранится кубическая текстура. Наша программа будет всегда полагать, что кубическая текстура хранится в файле формата DDS, имя которого совпадает с именем материала. Например, кубическая карта окружающей среды материала ball будет храниться в файле ball.dds.

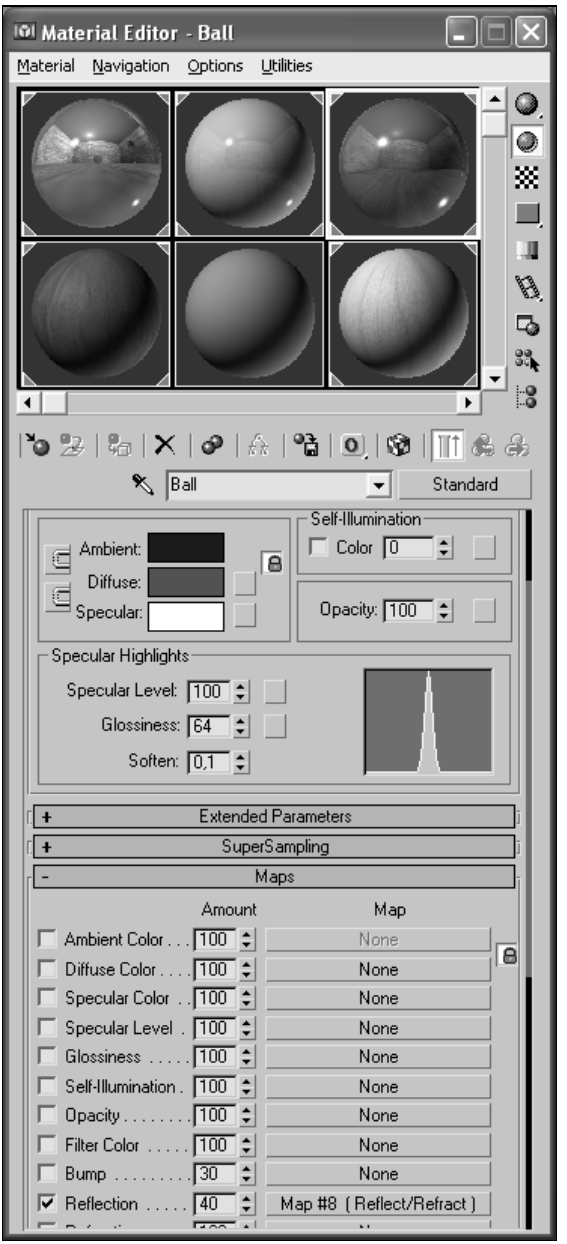


Рис. 10.33. Редактор материалов. Материал ball

Рис. 10.34. Редактор материалов. Карта reflect/refract материала ball

Следовательно, нам необходимо объединить файлы ball_RT.tga, ball_LF.tga, ball_BK.tga, ball_FR.tga, ball_UP.tga и ball_DN.tga в один файл ball.dds. Эта операция легко выполняется с использованием утилиты NV_DXT *(см. разд. 10.2.2)*. Однако здесь есть подвох — система координат, используемая 3D Studio MAX, не совпадает с системой координат, используемой OpenGL. Поэтому перед упаковкой всех граней в файл формата DDS, нам придется повернуть некоторые грани, а именно:

- ball_rt.tga. Повернуть на 90° против часовой стрелки;
- ball_lf.tga. Повернуть на 90° по часовой стрелке;
- ball_fr.tga. Повернуть на 180° против часовой стрелки.

Так как эти действия придется повторять при каждом изменении сцены, нам очень не помешало бы написать пакетный файл, автоматизирующий эти операции (листинг 10.21). Этот файл (ball.bat) находится на CD в каталоге \Examples\Ch10-Cubemap\3DSMAX_MODELS\Room. Там же находятся пакетные файлы для генерации кубических текстурных карт бокала и чайника (PolishedGold.bat и Teapot.bat).

Листинг 10.21

```
rem Поворачиваем изображение из файла ball_rt.tga на 90° против часовой
rem стрелки, результат заносится в файл tmp_rt.tga
rem Для поворота изображения используется распространенная программа
rem IrfanView
i_view32.exe ball_rt.tga /rotate_l /convert=tmp_rt.tga
rem Поворачиваем изображение из файла ball_lt.tga на 90° по часовой
rem стрелке, результат заносится в файл tmp_lt.tga
i_view32.exe ball_lf.tga /rotate_r /convert=tmp_lf.tga
rem Копируем изображение из файла ball_bk.tga в файл tmp_bk.tga
copy /b ball_bk.tga tmp_bk.tga
rem Поворачиваем изображение из файла ball_fr.tga на 180° против часовой
rem стрелки, результат заносится в файл tmp_fr.tga
i_view32.exe ball_fr.tga /rotate_l /convert=tmp_fr.tga
i_view32.exe tmp_fr.tga /rotate_l /convert=tmp_fr.tga
rem Копируем изображение из файла ball_up.tga в файл tmp_up.tga
copy /b ball_up.tga tmp_up.tga
rem Копируем изображение из файла ball_dn.tga в файл tmp_dn.tga
copy /b ball_dn.tga tmp_dn.tga
rem Объединяем изображения из файлов tmp_??.tga в файл ball.dds
rem Список файлов с гранями кубической карты находится в файле tmp.lst:
```

```
rem tmp_rt.tga
rem tmp_lf.tga
rem tmp_bk.tga
rem tmp_fr.tga
rem tmp_up.tga
rem tmp_dn.tga
nvdxt -cubemap ball.dds -list tmp.lst -u888
// Удаляем временные файлы
del tmp_rt.tga
del tmp_lf.tga
del tmp_bk.tga
del tmp_fr.tga
del tmp_up.tga
del tmp_dn.tga
```

В завершение остается только упаковать файл описания сцены (room.ase) со всеми текстурами, включая карты освещения, в единый ZIP-архив (room.zip).

Теперь мы можем приступать к модификации библиотеки ASE Reader, которая фактически сводится к четырем шагам:

1. Создать функцию `read`, загружающую файлы формата DDS из ZIP-архива.
2. Добавить в метод `CAseTextureManager::load_texture` дополнительный параметр `target`, указывающий на то, как трактовать содержимое файлов формата DDS. Если `target=GL_TEXTURE_2D`, то метод считает, что в DDS-файле хранится двухмерная текстура, в противном случае — кубическая.
3. Научить метод `CAseModel::LoadModel` определять исходя из названия материала имя файла DDS, в котором хранится текстура.
4. Добавить в метод `CAseModel::DrawModel` корректный вывод отражений при использовании кубических текстур.

Необходимость первого шага обусловлена тем, что библиотека NV_DDS не умеет загружать DDS-файлы из ZIP-архивов. Поэтому мы напишем небольшую функцию `read`, которая извлекает DDS-файл из архива, сохраняет его на диске, загружает в видеопамять с использованием библиотеки NV_DDS и, наконец, удаляет временный файл (листинг 10.22).

Листинг 10.22

```
// Загружаем файл формата DDS (inzip_filename) из ZIP-архива
// (zip_filename) и заносим информацию в объект dds. Если загрузка прошла
// успешно, возвращает false, иначе — true
```

```
bool read(string zip_filename, string inzip_filename, CDDSImage& dds)
{
// Определяем расширение файла
       size_t pos=inzip_filename.find_last_of('.');
       if ((pos==-1) || (pos==inzip_filename.size()-1) || (pos==0))
              return false;
       string file_ext=inzip_filename.substr(pos+1, in-
zip_filename.size()-pos-1);
// Извлекаем файл из ZIP-архива
       unsigned char* buffer;
       unsigned int buffer_size;
       buffer=unzip::open(zip_filename.c_str(), inzip_filename.c_str(),
&buffer_size);
       if (!buffer)
              return 0;
// Получаем свободное имя временного файла и путь к временному каталогу
// Windows
       char szTmpPath[MAX_PATH+1];
       char szTmpFile[MAX_PATH+1];
       GetTempPath(MAX_PATH, szTmpPath);
       GetTempFileName(szTmpPath, "img", 0, szTmpFile);

// Сохраняем извлеченный файл во временный каталог Windows
       string tmp_filename=szTmpFile;
       tmp_filename=tmp_filename.substr(0, tmp_filename.size()-
3)+file_ext;
       ofstream ofile(tmp_filename.c_str(), ios::binary);
       ofile.write((char*) buffer, buffer_size);
       ofile.close();
       delete[] buffer;
// Загружаем DDS-файл с использованием библиотеки NV_DDS
       bool result=dds.load(tmp_filename);
// Удаляем временный файл
       DeleteFile(tmp_filename.c_str());
       DeleteFile(szTmpFile);
       return result;
}
```

Теперь мы должны научить метод `CAseTextureManager::load_texture` отличать DDS-файлы с двухмерными изображениями от DDS-файлов с кубиче-

скими текстурами. Это можно сделать путем добавления дополнительного параметра `target`, указывающего на то, как трактовать содержимое файлов формата DDS. Если `target=GL_TEXTURE_2D`, то метод будет считать, что в DDS-файле хранится двухмерная текстура, в противном случае — кубическая. Исходный текст модифицированного метода `CAseTextureManager::load_texture` приведен в листинге 10.23.

Листинг 10.23

```
CAseTexture* CAseTextureManager::load_texture(string zip_filename, string
inzip_filename, int target)
{
// Определяем короткое имя файла (без пути)
       string short_inzip_filename=inzip_filename;
       int pos=inzip_filename.find_last_of("\\");
       if (pos!=-1)
       {
              if (pos!=short_inzip_filename.size()-1)
                     short_inzip_filename=inzip_filename.substr(pos+1,
inzip_filename.size()-pos-1);
              else
                     return 0;
       }
       string fullname=zip_filename+"::"+short_inzip_filename;
// Проверяем, был ли этот файл загружен ранее
       for (list<CAseTexture>::iterator itor=textures.begin();
itor!=textures.end(); itor++)
       {
              if (itor->filename==fullname)
              {
// Если этот файл уже загружен, увеличиваем счетчик ссылок и возвращаем
// указатель на загруженную текстуру
                     itor->count++;
                     return &(*itor);
              }
       }
// Создаем новую текстуру
       CAseTexture* pTexture=new CAseTexture;
       pTexture->count=1;
```

```
// Создаем текстурный объект в зависимости от значения параметра target
       if (target==GL_TEXTURE_2D)
              pTexture->pTextureObj=new tex_object_2D;
       else
              pTexture->pTextureObj=new tex_object_cube_map;
// Настраиваем параметры текстуры
       pTexture->pTextureObj->bind();
       pTexture->pTextureObj->parameter(GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);
       pTexture->pTextureObj->parameter(GL_TEXTURE_MAG_FILTER,
GL_LINEAR);
// Если файл надо трактовать как двухмерную текстуру
       if (target==GL_TEXTURE_2D)
       {
// Создаем временный объект image
              tga::tgaImage* image=new tga::tgaImage;
// Пытаемся загрузить текстуру из ZIP-файла
              if (ReadAndTexImage2D(zip_filename, short_inzip_filename,
image))
              {
// Если загрузка не удалась, пытаемся загрузить текстуру, используя
// полное имя файла
                     ReadAndTexImage2D(inzip_filename, image);
                     if (!image)
                     {
                            delete pTexture;
                            return false;
                     }
                     else
                            pTexture->filename=inzip_filename;
              }
              else
                     pTexture->filename=fullname;
// Сохраняем ширину и высоту изображения
              pTexture->width=image->width;
              pTexture->height=image->height;
              delete [] image;
       }
       else
```

```
        {
                CDDSImage dds;
// Пытаемся загрузить текстуру из ZIP-файла
                if (!read(zip_filename, short_inzip_filename, dds))
                {
// Если загрузка не удалась, пытаемся загрузить текстуру, используя
// полное имя файла
                        if(!dds.load(inzip_filename))
                        {
                                delete pTexture;
                                return false;
                        }
                        else
                                pTexture->filename=inzip_filename;
                }
                else
                        pTexture->filename=fullname;
// Загружаем текстуру в видеопамять
                dds.upload_textureCubemap();
// Сохраняем ширину и высоту изображения
                pTexture->width=dds[0].width;
                pTexture->width=dds[0].height;
        }
// Добавляем текстуру в список текстур
        textures.push_back(*pTexture);
// Возвращаем указатель на текстуру
        return pTexture;
}
```

Далее необходимо "научить" метод `CAseModel::LoadModel` определять, исходя из названия материала, имя файла DDS, в котором хранится кубическая текстурная карта. Как мы условились, кубическая текстура хранится в файле формата DDS, имя которого совпадает с именем материала. Следовательно, если в файле формата ASE отсутствует имя текстурной карты отражения, метод `CAseModel::LoadModel` получает имя файла кубической текстурной карты путем простого добавления к названию материала строки .dds. Но здесь есть один подвох: при сохранении названия материала в файл формата ASE, 3D Studio MAX заключает его в кавычки. Поэтому, чтобы получить из названия материала имя файла, удовлетворяющее требованиям Win32, из

него необходимо предварительно удалить кавычки. Наиболее существенные фрагменты метода `CAseModel::LoadModel` приведены в листинге 10.24.

Листинг 10.24

```
bool CAseModel::LoadModel(char *filename, char *modelname)
{
...
// Если определена карта отражения
                              if (mat->map_reflect.size()!=0)
                              {
                                      string bitmap;
                                      int target;
// Если задан файл изображения карты отражения
                                      if (mat->map_reflect[0]->bitmap)
                                      {
// Считаем, что карта отражения является сферической текстурной картой
                                              target=GL_TEXTURE_2D;
                                              bitmap=mat->map_reflect[0]->
bitmap;
                                      }
                                      else
                                      {
// Считаем, что карта отражения является кубической текстурной картой

       target=GL_TEXTURE_CUBE_MAP_ARB;
// Определяем название файла текстурной карты
                                              bitmap=string(mat->
name)+".dds";
// Удаляем из имени файла все кавычки
       bitmap.erase(remove(bitmap.begin(), bitmap.end(), '"'),
bitmap.end());
                                      }
// Загружаем текстуру отражения в видеопамять
       data[n].reflect_texture=TextureManager.load_texture(filename,
bitmap, target);
// Если произошла ошибка, выводим сообщение об ошибке
                                      if (data[n].reflect_texture==0)
                                      {
                                              errors.push_back(string("Error
during loading material ")+mat->name+" for object "+
                                                      (*data[n].name)+" :");
```

```
                                        errors.push_back(string("can't
load file ")+filename+"::"+bitmap);
                                    }
// Определяем прозрачность отражения
                                    data[n].reflect_color[0]=1.0;
                                    data[n].reflect_color[1]=1.0;
                                    data[n].reflect_color[2]=1.0;
                                    data[n].reflect_color[3]=mat-
>map_reflect[0]->amount;
                                }
...
}
```

И, наконец, нам надо научить метод `CAseModel::DrawModel` различать сферические и кубические текстурные карты. Казалось бы, что может быть проще? При использовании сферических текстурных карт надо включать режим генерации текстурных координат `GL_SPHERE_MAP`, а при использовании кубических карт — `GL_REFLECTION_MAP_ARB`. Но не стоит забывать, что для корректного наложения статической кубической текстурной карты методу `CAseModel::DrawModel` необходимо знать текущее положение наблюдателя. Казалось бы, достаточно добавить в метод `CAseModel::DrawModel` дополнительный параметр, содержащий матрицу, которая будет присваиваться матрице текстуры `GL_TEXTURE` перед выводом отражения. Однако в этом случае вызов метода `CAseModel::DrawModel` нельзя будет размещать внутри дисплейного списка, что приведет к значительному снижению производительности. Выход из данной ситуации — задание матрицы текстуры вручную. Чтобы изменение текстурной матрицы не повлияло на все текстуры объекта, за исключением кубических текстурных карт, в начало метода `CAseModel::DrawModel` необходимо вставить следующие строки:

```
glMatrixMode(GL_TEXTURE);
// Сохраняем текущую матрицу текстуры в стеке
glPushMatrix();
// Замещаем текущую матрицу единичной
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
```

А вывод кубической текстурной карты необходимо обрамить следующим кодом:

```
glMatrixMode(GL_TEXTURE);
// Восстанавливаем матрицу текстуры
glPopMatrix();
```

```
glMatrixMode(GL_MODELVIEW);
// Рисуем объект с кубической текстурной картой
...
// Помещаем пользовательскую текстурную матрицу в стек и замещаем
// текстурную матрицу единичной
glMatrixMode(GL_TEXTURE);
glPushMatrix();
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
```

Теперь мы в силах модифицировать метод `CAseModel::DrawModel`. Это не составит большого труда (листинг 10.25).

Листинг 10.25

```
void CAseModel::DrawModel(bool textured)
{
       glPushAttrib(GL_ALL_ATTRIB_BITS);
       glEnable(GL_COLOR_MATERIAL);
glPushMatrix();
// Сохраняем матрицу текстуры и заменяем ее единичной
       glMatrixMode(GL_TEXTURE);
       glPushMatrix();
       glLoadIdentity();
       glMatrixMode(GL_MODELVIEW);
// Если сцена загружена
       if (data)
// Перебираем все объекты сцены
              for (int n = 0; n < numChunks; n++)
              {
// Визуализируем диффузную составляющую материала
...
// Если у материала имеется текстура отражения
                     if (textured && data[n].reflect_texture)
                     {
// Настраиваем параметры OpenGL для наложения отражения поверх диффузной
// составляющей материала
                            glBlendFunc(GL_SRC_ALPHA,
GL_ONE_MINUS_SRC_ALPHA);
```

```
                              glEnable(GL_BLEND);
                              glDepthMask(false);
                              glDepthFunc(GL_LEQUAL);
                              glTexEnvi(GL_TEXTURE_ENV,
GL_TEXTURE_ENV_MODE, GL_MODULATE);
                              data[n].reflect_texture->pTextureObj->
bind();
                              data[n].reflect_texture->pTextureObj->
enable();
                              glColor4fv(&data[n].reflect_color[0]);
// Если отражение задано кубической текстурой
                              if (data[n].reflect_texture->pTextureObj->
target==GL_TEXTURE_CUBE_MAP_ARB)
                              {
// Включаем автоматическую генерацию текстурных координат отражения
                                    glEnable(GL_TEXTURE_GEN_S);
                                    glEnable(GL_TEXTURE_GEN_T);
                                    glEnable(GL_TEXTURE_GEN_R);
                                    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,
GL_REFLECTION_MAP_ARB);
                                    glTexGeni(GL_T, GL_TEXTURE_GEN_MODE,
GL_REFLECTION_MAP_ARB);
                                    glTexGeni(GL_R, GL_TEXTURE_GEN_MODE,
GL_REFLECTION_MAP_ARB);
// Восстанавливаем пользовательскую матрицу текстуры
                                    glMatrixMode(GL_TEXTURE);
                                    glPopMatrix();
                                    glMatrixMode(GL_MODELVIEW);
                              }
                              else
                              {
// Включаем автоматическую генерацию сферических текстурных координат
                                    glEnable(GL_TEXTURE_GEN_S);
                                    glEnable(GL_TEXTURE_GEN_T);
                                    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE,
GL_SPHERE_MAP);
                                    glTexGeni(GL_T, GL_TEXTURE_GEN_MODE,
GL_SPHERE_MAP);
                              }
// Рисуем текущий объект
                              glDrawElements(GL_TRIANGLES,
data[n].numfaces*3, GL_UNSIGNED_INT, data[n].indices);
```

```
// Если накладывается кубическая текстура
                              if (data[n].reflect_texture->pTextureObj->
target==GL_TEXTURE_CUBE_MAP_ARB)
                              {
// Сохраняем матрицу текстуры и заменяем ее единичной
                                    glMatrixMode(GL_TEXTURE);
                                    glPushMatrix();
                                    glLoadIdentity();
                                    glMatrixMode(GL_MODELVIEW);
                              }
// Восстанавливаем параметры OpenGL по умолчанию
                              data[n].reflect_texture->pTextureObj->
disable();
                              glDisable(GL_TEXTURE_GEN_S);
                              glDisable(GL_TEXTURE_GEN_T);
                              glDisable(GL_TEXTURE_GEN_R);
                              glDisable(GL_BLEND);
                              glDepthMask(true);
                              glDepthFunc(GL_LESS);
                        }
                        glPopMatrix();
                        }
                  }
// Восстанавливаем атрибуты, матрицы текстуры и модели
        glPopAttrib();
        glMatrixMode(GL_TEXTURE);
        glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
        glPopMatrix();
}
```

Для демонстрации использования новой версии библиотеки ASE Reader я перекомпилировал пример Ex08 *главы 4*, заменив модель долины valley.ase на модель комнаты room.ase (Ex16). Внешний вид окна программы приведен на рис. 10.35.

Как видно, пример Ex16 корректно выводит на экран сцену, содержащую глянцевые объекты. Причем в каждом глянцевом объекте полностью отражается вся окружающая среда объекта, включая другие глянцевые объекты. В общем, определить, что изображение на рис. 10.35 визуализировалось с использованием OpenGL, а не 3D Studio MAX, сможет только профессионал.

Рис. 10.35. Изображение из примера Ex16

Заключение

В этой главе мы научились моделировать глянцевые объекты с использованием сферических и кубических текстурных карт. Параллельно были рассмотрены различные варианты хранения кубических текстур на диске, как с использованием классических графических форматов (TGA, JPG), так и формата DDS. Кроме того, был затронут вопрос нетрадиционного использования кубических текстурных карт для моделирования закраски методом Фонга в реальном времени. В заключение мы рассмотрели экспорт из 3D Studio MAX материалов, использующих текстурные карты отражения reflect/refract.

Заключение

Ну, вот и все. Надеюсь, книга принесла вам пользу и удовольствие, но возможно у вас появились замечания. Обо всех найденных ошибках и опечатках обязательно сообщайте на мой e-mail: **gsaf@sura.ru**.

Несмотря на солидный объем, это всего лишь вводное руководство, которое охватывает только небольшую часть таких огромных тем, как NVIDIA OpenGL SDK и OpenGL. Так, например, в этой книге не рассмотрены такие важные темы, как мультитекстурирование, шейдеры и системы частиц (спрайтов). Однако все эти темы, в принципе, нереально уместить в одной книге такого размера: одна только краткая спецификация расширений OpenGL, поддерживаемых видеокартами NVIDIA GeForceFX, занимает более 1100 страниц. Тем не менее, после прочтения этой книги вы легко сможете продолжить самостоятельное изучение этих тем.

Ниже приведены несколько полезных ссылок на интернет-сайты, содержащие огромное количество информации по OpenGL и связанным с ней темам:

- **www.opengl.org** — главный сайт любого программиста, использующего OpenGL;
- **www.nvidia.com/developer** — сайт поддержки разработчиков, использующих продукты корпорации NVIDIA. Содержит огромное количество документации и утилит, большая часть из которых также подходит для реализации OpenGL других производителей;
- **www.ati.com/developer** — сайт поддержки разработчиков, использующих продукты корпорации ATI. Несмотря на то, что сайт ориентирован в первую очередь на DirectX, на нем имеется довольно много информации по OpenGL;
- **www.gamedev.ru** — сайт, посвященный разработке игр. Имеется огромное количество информации по OpenGL;
- **www.ixbt.com** — хотя этот сайт и не ориентирован на профессиональных программистов, на нем периодически появляются интересные статьи, которые также будут полезны разработчикам. Кроме того, на сайте имеется ряд интересных форумов, включая форум разработчиков компьютерных игр.

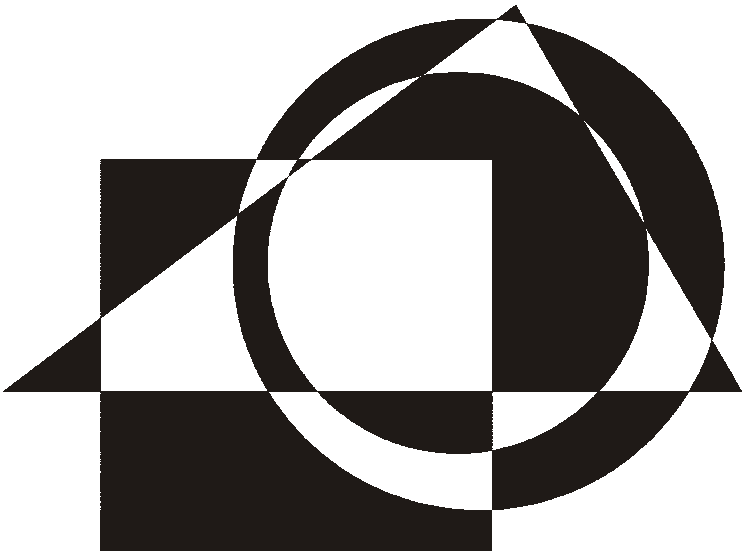
После того как книга была написана, у меня родилась идея написать ее продолжение. В настоящее время я работаю над двумя книгами.

- Во втором издании этой книги я собираюсь рассмотреть новую версию NVIDIA OpenGL SDK, которая теперь называется Cg Toolkit. В круг

вопросов принятых к рассмотрению попадут новые расширения OpenGL версий 1.2 и выше, которые не вошли в эту книгу. Среди них EXT_texture3D, ARB_multitexture, GL_ARB_texture_env_combine, GL_ARB_texture_env_dot3 и др. В конце книги планируется привести пример разработки полноценного аркадного симулятора самолета.

- Другая книга будет посвящена шейдерам. Скорее всего, в ней будут рассматриваться язык шейдеров OpenGL 2.0 (GLslang) и инструментарий корпорации ATI (RenderMonkey) для написания и отладки шейдеров. Но поскольку на момент написания книги будущее программных продуктов ATI и NVIDIA видится очень туманным, то в структуру книги, скорее всего, еще будут внесены существенные изменения.

Если у вас имеются какие-либо пожелания к материалу будущих книг, обязательно сообщите на мой e-mail: **gsaf@sura.ru**.



ЧАСТЬ III

Приложения

Приложение 1

Таблица расширений, поддерживаемых видеокартами корпорации NVIDIA

Название расширения	Тип GPU				
	Riva128	RivaTNT	NV1x	NV2x	NV3x
ARB_depth_texture			Em	R25+	X
ARB_fragment_program			Em	Em	X
ARB_imaging		R10	R10	X	X
ARB_multisample				X	X
ARB_multitexture		X	X	X	X
ARB_point_parameters		R35	R35	R35	X
ARB_shadow			Em	R25+	X
ARB_texture_border_clamp			Em	X	X
ARB_texture_compression			X	X	X
ARB_texture_cube_map			X	X	X
ARB_texture_env_add		X	X	X	X
ARB_texture_env_combine			X	X	X
ARB_texture_env_crossbar					
ARB_texture_env_dot3			X	X	X
ARB_texture_mirrored_repeat		R40	R40	R40	X
ARB_transpose_matrix		X	X	X	X
ARB_vertex_program			R40+	R40+	X

(продолжение)

Название расширения	Тип GPU				
	Riva128	**RivaTNT**	**NV1x**	**NV2x**	**NV3x**
ARB_window_pos		R40	R40	R40	X
EXT_abgr	X	X	X	X	X
EXT_bgra	X	X	X	X	X
EXT_blend_color			X	X	X
EXT_blend_func_separate			Em	Em	X
EXT_blend_minmax			X	X	X
EXT_blend_subtract			X	X	X
EXT_clip_volume_hint			R20+		
EXT_compiled_vertex_array		X	X	X	X
EXT_draw_range_elements		R20	R20	R20	X
EXT_fog_coord		X	X	X	X
EXT_multi_draw_arrays		R25	R25	R25	X
EXT_packed_pixels	X	X	X	X	X
EXT_paletted_texture			X	X	X
EXT_point_parameters	X	X	X	X	X
EXT_rescale_normal			X	X	X
EXT_secondary_color		X	X	X	X
EXT_separate_specular_color		X	X	X	X
EXT_shadow_funcs			Em	R25+	X
EXT_shared_texture_palette			X	X	X
EXT_stencil_wrap	X	X	X	X	X
EXT_texture3D		sw	sw	X	X
EXT_texture_compression_s3tc			X	X	X
EXT_texture_cube_map			X	X	X
EXT_texture_edge_clamp		X	X	X	X
EXT_texture_env_add		X	X	X	X
EXT_texture_env_dot3			X	X	X

(продолжение)

Название расширения	Тип GPU				
	Riva128	RivaTNT	NV1x	NV2x	NV3x
EXT_texture_env_combine		X	X	X	X
EXT_texture_filter_anisotropic			X	X	X
EXT_texture_lod_bias		R10	X	X	X
EXT_texture_object	X	X	X	X	X
EXT_vertex_array	X	X	X	X	X
EXT_vertex_weighting		X	X	X	
KTX_buffer_region	X	X	X	X	X
HP_occlusion_test			Em	R25	X
IBM_texture_mirrored_repeat		X	X	X	X
NV_blend_square		X	X	X	X
NV_copy_depth_to_color			Em	R20	X
NV_depth_clamp			Em	R25+	X
NV_evaluators		R10	R10	X	
NV_fence			X	X	X
NV_float_buffer			Em	Em	X
NV_fog_distance		X	X	X	X
NV_half_float			Em	Em	X
NV_light_max_exponent		X	X	X	X
NV_multisample_filter_hint				X	X
NV_occlusion_query			Em	R25	X
NV_packed_depth_stencil		R10+	R10+	R10+	X
NV_pixel_data_range			R40	R40	X
NV_point_sprite			R35+	R25	X
NV_primitive_restart			Em	Em	X
NV_register_combiners			X	X	X
NV_register_combiners2			Em	X	X
NV_texgen_emboss			X		

(продолжение)

Название расширения	Тип GPU				
	Riva128	RivaTNT	NV1x	NV2x	NV3x
NV_texgen_reflection	X	X	X	X	X
NV_texture_compression_vtc			Em	X	X
NV_texture_env_combine4		X	X	X	X
NV_texture_expand_normal			Em	Em	X
NV_texture_rectangle			X	X	X
NV_texture_shader			Em	X	X
NV_texture_shader2			Em	X	X
NV_texture_shader3			Em	R25	X
NV_vertex_array_range			X	X	X
NV_vertex_array_range2			R10	R10	X
NV_vertex_program			R10	X	X
NV_vertex_program1_1			R25	R25	X
NV_vertex_program2			Em	Em	X
S3_s3tc			X	X	X
SGIS_generate_mipmap			R10	X	X
SGIS_multitexture		X	X	X	
SGIS_texture_lod			X	X	X
SGIX_depth_texture			Em	X	X
SGIX_shadow			Em	X	X
WGL_ARB_buffer_region		X	X	X	X
WGL_ARB_extensions_string		X	X	X	X
WGL_ARB_multisample				X	X
WGL_ARB_pixel_format		R10	R10	X	X
WGL_ARB_pbuffer		R10	R10	X	X
WGL_ARB_render_texture			R25	R25	X
WGL_EXT_extensions_string		X	X	X	X
WGL_EXT_swap_control		X	X	X	X

(окончание)

Название расширения	Тип GPU				
	Riva128	**RivaTNT**	**NV1x**	**NV2x**	**NV3x**
WGL_NV_render_depth_texture			Em	R25	X
WGL_NV_render_texture_ rectangle			R25	R25	X
WIN_swap_hint	X	X	X	X	X

Условные обозначения:

- X — поддерживается;
- sw — поддерживается в режиме программной растеризации. Иными словами, использование этого расширения приведет к резкому падению производительности;
- Em — аналогично sw, но поддерживается только в режиме эмуляции;
- Rxx — поддержка появилась только начиная с драйверов Detonator версии xx;
- Rxx+ — поддержка появилась только спустя некоторое время после выхода соответствующей версии драйверов.

Приложение 2

Таблица расширений, поддерживаемых видеокартами корпорации ATI

Название расширения	Тип GPU			
	Radeon	Radeon	Radeon	Radeon
	7200	7000	8500	9500
	7500		9000	9600
			9100	9700
			9200	9800
Расширения корпорации ATI				
GL_ATI_draw_buffers				X
GL_ATI_element_array			X	X
GL_ATI_envmap_bumpmap	X	X	X	X
GL_ATI_fragment_shader			X	X
GL_ATI_map_object_buffer	X		X	X
GL_ATI_pn_triangles			X	X
GL_ATI_separate_stencil				X
GL_ATI_texture_env_combine3	X	X	X	X
GL_ATI_texture_float				X
GL_ATI_texture_mirror_once	X	X	X	X
GL_ATI_vertex_array_object	X		X	X

(продолжение)

GL_ATI_vertex_attrib_array_object			X	X
GL_ATI_vertex_streams	X	X	X	X
GL_ATIX_texture_env_combine3	X	X	X	X
GL_ATIX_texture_env_route	X	X	X	X
GL_ATIX_vertex_shader_output_point_size	X	X	X	X
WGL_ATI_pixel_format_float				X
Расширения, одобренные комитетом ARB				
GL_ARB_depth_texture				X
GL_ARB_fragment_program				X
GL_ARB_fragment_shader				X
GL_ARB_multitexture	X	X	X	X
GL_ARB_multisample				X
GL_ARB_occlusion_query	X		X	X
GL_ARB_point_parameters			X	X
GL_ARB_point_sprite				X
GL_ARB_shader_objects				X
GL_ARB_shading_language_100				X
GL_ARB_shadow				X
GL_ARB_shadow_ambient				X
GL_ARB_texture_border_clamp	X	X	X	X
GL_ARB_texture_compression	X	X	X	X
GL_ARB_texture_cube_map	X	X	X	X
GL_ARB_texture_env_add	X	X	X	X
GL_ARB_texture_env_combine	X	X	X	X
GL_ARB_texture_env_crossbar	X	X	X	X
GL_ARB_texture_env_dot3	X	X	X	X
GL_ARB_texture_mirrored_repeat	X	X	X	X

(продолжение)

GL_ARB_transpose_matrix	X	X	X	X
GL_ARB_vertex_blend	X	X	X	X
GL_ARB_vertex_buffer_object	X		X	X
GL_ARB_vertex_program			X	X
GL_ARB_vertex_shader				X
GL_ARB_window_pos	X	X	X	X
WGL_ARB_extensions_string	X	X	X	X
WGL_ARB_make_current_read	X	X	X	X
WGL_ARB_multisample				X
WGL_ARB_pbuffer	X (not NT4)	X (not NT4)	X (not NT4)	X (not NT4)
WGL_ARB_pixel_format	X	X	X	X
WGL_ARB_render_texture	X (not NT4)	X (not NT4)	X (not NT4)	X (not NT4)
Другие расширения				
GL_EXT_abgr	X	X	X	X
GL_EXT_bgra	X	X	X	X
GL_EXT_blend_color			X	X
GL_EXT_blend_func_separate			X	X
GL_EXT_blend_minmax			X	X
GL_EXT_blend_subtract			X	X
GL_EXT_clip_volume_hint	X	X	X	X
GL_EXT_compiled_vertex_array	X	X	X	X
GL_EXT_draw_range_elements	X	X	X	X
GL_EXT_fog_coord	X	X	X	X
GL_EXT_multi_draw_arrays	X	X	X	X
GL_EXT_packed_pixels	X	X	X	X
GL_EXT_point_parameters	X	X	X	X
GL_EXT_rescale_normal	X	X	X	X

(продолжение)

GL_EXT_secondary_color	X	X	X	X
GL_EXT_separate_specular_color	X	X	X	X
GL_EXT_shadow_funcs				X
GL_EXT_stencil_wrap	X	X	X	X
GL_EXT_texgen_reflection	X	X	X	X
GL_EXT_texture3D	X	X	X	X
GL_EXT_texture_compression_s3tc	X	X	X	X
GL_EXT_texture_cube_map	X	X	X	X
GL_EXT_texture_edge_clamp	X	X	X	X
GL_EXT_texture_env_add	X	X	X	X
GL_EXT_texture_env_combine	X	X	X	X
GL_EXT_texture_env_dot3	X	X	X	X
GL_EXT_texture_filter_anisotropic	X	X	X	X
GL_EXT_texture_lod_bias	X	X	X	X
GL_EXT_texture_object	X	X	X	X
GL_EXT_texture_rectangle	X	X	X	X
GL_EXT_vertex_array	X	X	X	X
GL_EXT_vertex_shader			X	X
GL_HP_occlusion_test	X	X	X	X
GL_KTX_buffer_region			X	X
GL_NV_blend_square	X	X	X	X
GL_NV_occlusion_query	X	X	X	X
GL_NV_point_sprite				X
GL_NV_texgen_reflection	X	X	X	X
GL_S3_s3tc	X	X	X	X
GL_SGI_color_matrix	X	X	X	X
GL_SGI_texture_edge_clamp	X	X	X	X
GL_SGIS_generate_mipmap	X	X	X	X

(окончание)

GL_SGIS_multitexture	X	X	X	X
GL_SGIS_texture_border_clamp	X	X	X	X
GL_SGIS_texture_lod	X	X	X	X
GL_SUN_multi_draw_arrays	X	X	X	X
GL_WIN_swap_hint	X	X	X	X
WGL_EXT_extensions_string	X	X	X	X
WGL_EXT_swap_control	X	X	X	X

Приложение 3

Описание компакт-диска

Компакт-диск, прилагаемый к книге, содержит восемь каталогов:

- ATI SDK — заголовочные файлы для реализации OpenGL корпорации ATI;
- BugLayerUtils — библиотека BuglsayerUtils из книги [8] (библиотека имеется на CD-ROM). Эта библиотека изменяет макрос `assert` стандартной библиотеки C++, после чего он начинает выводить множество дополнительной информации, а также исправляет ряд ошибок в Visual C++;
- DOC — документация, на которую имеются ссылки в тексте книги;
- Drivers — драйверы для видеокарт корпораций ATI и NVIDIA, используемые при подготовке примеров для этой книги;
- Examples — проекты примеров книги, сгруппированные по главам;
- NVIDIA SDK — NVIDIA SDK 5.21 с несколькими обновлениям, а также другой инструментарий NVIDIA:
  - DXT_TOOLS_v5.32 — набор утилит NVIDIA для работы с файлами формата DDS;
  - Extensions — библиотеки GLH_GLUT_EXT и NV_UTIL_EXT, расширяющие функциональность библиотеки OpenGL Helper Library и NV_UTIL соответственно;
  - NV_DDS — обновленная версия библиотеки NV_DDS, позволяющая загружать текстуры из файлов формата DDS;
  - Textures — библиотека текстур высокого разрешения формата DDS;
  - Update — обновленные файлы NVIDIA OpenGL SDK;
- `psCamera` — вспомогательные классы проверки попадания прямоугольной оболочки объекты в видовой объем камеры, созданные Петром Поповым;
- Setup — файлы инсталляции примеров.

Инсталляция примеров книги осуществляется путем запуска файла setup.exe из каталога Setup. Процесс инсталляции примеров состоит из четырех этапов.

1. Установка на компьютер NVIDIA SDK 5.21, включающего в себя NVIDIA OpenGL SDK.
2. Установка исправлений для NVIDIA OpenGL SDK. На этом этапе часть файлов NVIDIA OpenGL SDK заменяются исправленными вариантами из папки Update.
3. Копирование примеров книги на жесткий диск.
4. Регистрация путей к каталогам NVIDIA OpenGL SDK и примеров книги в Visual Studio .NET 2002 путем модификации файла VCComponents.dat.

Хотя все эти действия при желании можно выполнить и вручную, установка примеров книги с использованием инсталлятора занимает значительно меньше времени.

Список литературы и источников в Интернете

1. Краснов М. OpenGL. Графика в проектах Delphi. — СПб.: БХВ — Санкт-Петербург, 2000. — 352 с.
2. Эйнджел Э. Интерактивная компьютерная графика. Вводный курс на базе OpenGL, 2-е. изд.: Пер. с англ. — М.: Издательский дом "Вильямс", 2001. — 592 с. — Парал. тит. англ.
3. Черносвитов А. Visual C++ 7: Учебный курс. — СПб.: Питер, 2001. — 528 с.
4. Бартеньев О. В. Графика OpenGL: программирование на Фортране. Диалог-МИФИ, 2000. — 368 с.
5. Тихомиров Ю. В. OpenGL. Программирование трехмерной графики. — 2-е. изд. — БХВ — Санкт-Петербург, 2002. — 304 с.
6. Ву Мейсон, Нейдер Джеки, Девис Том, Шрайнер Дейв. OpenGL. Официальное руководство программиста: Пер. с англ./Мейсон Ву, Джеки Нейдер, Том Девис, Дейв Шрайнер. — СПб.: ООО "ДиаСофтЮП", 2002. — 592 с.
7. Тейксейра С., Пачеко К. Borland Delphi 6. Руководство разработчика: Пер. с англ. — М.: Издательский дом "Вильямс", 2002. — 1120 с. — Парал. тит. англ.
8. Роббинс Дж. Отладка приложений: Пер. с англ. — СПб.: БХВ — Санкт-Петербург, 2001. — 512 с.
9. Хилл Ф. OpenGL. Программирование компьютерной графики. Для профессионалов. — СПб.: Питер, 2002. — 1088 с.
10. Дьяконов В. П. Справочник по расчетам на микрокалькуляторах. — 3-е изд., перераб. и доп. — М.: Наука. Гл. ред. физ.-мат. лит., 1989. — 464 с.
11. Coordinate Systems, Vectors, Planes FAQ (**ftp://ftp.netcom.com/pub/he/hexapod/index.html**).
12. The Matrix and Quaternions FAQ (**ftp://ftp.netcom.com/pub/he/hexapod/index.html**)
13. Сергей Ваткин. Кватернионы в программировании игр (**www.gamedev.ru**).
14. Маров М. Эффективная работа: 3ds max 4. — СПб.: Питер, 2001. — 864 с.

15. Маров М. Энциклопедия 3D Studio MAX 3. — СПб.: Питер, 2000. — 1184 с.
16. Белл Дж. 3D Studio MAX R2.5. Спецэффекты и дизайн: Пер. с англ. — М.: Диалектика, 1998. — 464 с. — Парал. тит. англ.
17. Белл Дж. А. и др. 3D Studio MAX R3. Спецэффекты и дизайн: Пер. с англ. — Уч. пос. — М.: Издательский дом "Вильямс", 2000. — 400 с. — Парал. тит. англ.
18. Mark Kilgard. All About OpenGL Extensions (**www.gamedev.org**).
19. Mark Kilgard. Improving Shadows and Reflections via the Stencil Buffer (**www.nvidia.com**).
20. Андрей Воробьев и др. 3DGiToги (**www.ixbt.com**).
21. Алексей Берилло. Обзор форматов текстурного сжатия S3TC и FXT1 и сравнение с 16-битными текстурами (**www.ixbt.com**).
22. Mark Kilgard. Слайды доклада "Real-time Environment Reflections with OpenGL" на GDC'99 (**www.nvidia.com/developer**).
23. NVIDIA Cube Map OpenGL Tutorial (**www.nvidia.com/developer**).
24. NVIDIA OpenGL Extension Specifications (**www.nvidia.com/developer**).
25. Mark Kilgard. A Practical and Robust Bump-mapping Technique for Today's GPUs (**www.nvidia.com/developer**).
26. Cass Everitt. Anisotropic Texture Filtering in OpenGL (**www.nvidia.com**).
27. Андрей Воробьев, Александр Медведев. Обзор "NVIDIA GeForce3" (**www.ixbt.com**).
28. Андрей Воробьев, Александр Медведев. ATI RADEON 9700 Pro 128MB (**www.ixbt.com**).
29. ATI OpenGL Extension Support (**www.ati.com/developer**).
30. Christopher Oat. Rendering to an off-screen buffer with WGL_ARB_pbuffer (**www.ati.com/developer**).
31. Ashu Rege. Слайды доклада "Occlusion (HP and NV Extensions)" на GDC 2002 (**www.nvidia.com/developer**).
32. Chris Wynn. Слайды доклада "OpenGL Render-to-Texture" (**www.nvidia.com/developer**).

Предметный указатель

Д

И

К

M

O

П

Р

С

Т

У

Ф

(окончание рубрики см. на с. 716)

Ц

Э