DirectX и C++
искусство программирования

МИХАИЛ ФЛЕНОВ

- Графические эффекты в демонстрационных роликах
- Пиксельные и вершинные шейдеры
- Оптимизация графики
- 2D- и 3D-эффекты
- Современные графические технологии

+CD

Михаил Фленов

DirectX и C++
искусство программирования

Санкт-Петербург
«БХВ-Петербург»
2006

УДК 681.3.06
ББК 32.973.26-018.2
Ф69

Фленов М. Е.

Ф69 DirectX и C++. Искусство программирования. — СПб.: БХВ-Петербург, 2006. — 384 с.: ил.

ISBN 5-94157-831-8

Рассмотрено программирование графических эффектов на языке C++ с использованием популярной библиотеки DirectX. На занимательных практических примерах показано, как создавать различные визуальные эффекты (реалистичный огонь, электрические разряды, зеркала и др.), используемые при разработке демонстрационных роликов (Demoscene). Пошагово описано применение основных методов и интерфейсов DirectX. Показано, как написать оптимальный и эффективный программный код. Большое внимание уделено технологии использования вершинных и пиксельных шейдеров для создания реалистичных изображений. Компакт-диск, прилагаемый к книге, содержит листинги примеров из книги и дополнительную информацию по DirectX.

Для программистов

УДК 681.3.06
ББК 32.973.26-018.2

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Юрий Рожко*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн обложки	*Инны Тачиной*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 26.02.06.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 30,96.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

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

ISBN 5-94157-831-8

Оглавление

Предисловие

Данная книга посвящена разработке демо-роликов с использованием C++ и DirectX. Чтобы понять, что именно мы будем рассматривать и чему учиться, нужно пояснить, что такое демо-ролик (Demo) и демо-сцена (Demoscene). *Демо-ролик* — это небольшая программа или вставка кода в программе, которая показывает графические эффекты с возможным звуковым сопровождением. *Демо-сцена* — это целая культура создания роликов, которая определяет определенные правила и законы. Существуют даже целые *демо-вечеринки* (Demo Party) и конкурсы, где любая команда может представить на суд зрителей свой ролик. Да, именно команда. В настоящее время уже очень сложно создать действительно впечатляющий ролик в одиночку. Для этого нужно как минимум три профессионала в своей области: программист, художник и музыкант.

Но все эти определения понятия *демо* нельзя назвать полными, потому что это что-то большее, и чтобы понять это, необходимо окунуться в историю и увидеть, как зародилась эта культура и как она развивалась. Только тогда вы сможете понять дух этой действительно уникальной культуры, и что именно будет рассматриваться в данной книге.

О чем эта книга

Итак, данная книга посвящена созданию графической части демо-роликов, созданию эффектов и формированию видеоряда, т. е. мы будем рассматривать только программирование. Рисование текстур и создание музыки выходит за рамки книги даже потому, что я просто не умею рисовать и не смогу вас этому научить ☺. Но несмотря на это, мы будем использовать уже созданные текстуры и звук, дабы научиться синхронизировать все это в пространстве экрана и во времени проигрывания демо-ролика.

Несмотря на то, что в большинстве примеров будет использоваться графический ускоритель, некоторые расчеты мы будем производить самостоятельно, дабы повысить скорость или просто насладиться программированием графики.

Чтобы наши демо-ролики не выглядели слишком убогими и устаревшими, в качестве основы я выбрал достаточно мощный DirectX 9. Это значит, что не каждый компьютер сможет воспроизвести такие ролики и вам понадобится графический ускоритель ATI 9XXX-серии или GeForce FX и выше. Я думаю, что к моменту выхода этой книги такие ускорители и даже мощнее уже будут стоять на большинстве компьютеров и для вас это не станет большой проблемой.

Конечно же, некоторые ролики можно создать и с более старой версией DirectX, что позволит им работать на более старых компьютерах и даже без графического ускорителя, но я все же предпочитаю больше следовать прогрессу, а не стоять на месте. Девятая версия DirectX довольно сильно отличается от 8-й, поэтому мы рассмотрим все функции достаточно подробно. Это будет полезно тем, кто никогда еще не работал с DirectX или работал, но со старой версией.

Поскольку основная цель демо-ролика показать возможности программиста и компьютера, то немалую часть книги мы уделим оптимизации. Некоторые участки кода будут упрощены, дабы код был читаемым, но все же, оптимизации будет уделено достаточно большое внимание. Только благодаря ей вы и сможете создать что-то впечатляющее. Если же на экране отображается простая движущаяся сфера, которая постоянно тормозит, то это никого не впечатлит. Но если сцена будет сложной и при этом она станет отображаться с частотой в 60 кадров в секунду, то это уже искусство.

Я всегда говорю, что если вы создадите программу, то вы программист, но если вы создаете ее лучше других, то вы хакер. Мы будем учиться искусству хакеров, т. е. создавать демо-ролики лучше других.

Для понимания книги вам достаточно только знания языка программирования C++. Технологию DirectX мы будем рассматривать с самых основ, но правда только то, что будет необходимо для материала книги. Рассмотреть абсолютно все возможности просто нереально, уж слишком много накопилось их у DirectX. Если же вы хотите познакомиться с DirectX более глубоко, то без специализированной книги, MSDN (Microsoft Development Network) и файлов помощи вам не обойтись.

С одной стороны, данная работа может позиционироваться как руководство (manual) по созданию эффектов, а с другой стороны, посредством этой книги вы будете с помощью интересных примеров изучать DirectX. Сложность эффектов, которые мы будем рассматривать, будет расти постепенно. Первые примеры могут показаться вам слишком простыми, но эффектность сцены

иногда как раз и кроется в простоте. Если же объединить несколько простых эффектов, то может получиться что-то вообще сногсшибательное.

Благодарности

В каждой своей книге я благодарю тех, кто помогает мне в работе. Не устану благодарить своих родных и близких (жену, детей, родителей и т. д.), которые ежедневно окружают меня и терпят мои исчезновения в виртуальной реальности. Да, иногда я теряюсь в реальном мире, и компьютерный мир становится основным. Даже когда я ложусь спать, мне сняться операторы того языка программирования, за которым просидел весь день, и графические линии сцены, которая создавалась сегодня.

Не устану благодарить и редакцию Gemeland, а именно журналов "Хакер" и "Хакер Спец", с которыми я сотрудничаю уже долгое время. В последнее время мне все меньше удается писать статьи для этих журналов, но я всегда с теплыми чувствами относился, и буду относиться к этим ребятам. Как бы не ругали журнал "Хакер", я считаю, что на данный момент это единственный реально компьютерный журнал, направленный на массового читателя и профессионалов и отражающий реальную цифровую жизнь.

Я заметил, что во всех предыдущих книгах я обошел вниманием одного человека, который помогает в создании практически всех книг — Лозовского Александра, выпускающего редактора журналов "Хакер" и "Хакер Спец". Именно его рецензии вы можете наблюдать на обратной стороне книги.

Хочу поблагодарить своих виртуальных друзей ☺ и особенно ребят из VR-team, которые помогают мне в создании и управлении сайтом **http://www.vr-online.ru**.

И как всегда, отдельная благодарность вам, за то, что купили эту книгу, и надеюсь, что она вам понравится. Мы постарались сделать все необходимое, чтобы книга была интересной и никто не пожалел потраченных денег.

Если у вас возникли вопросы по книге или просто пожелания, то я всегда открыт к общению. Пишите мне на **horrific@vr-online.ru** или лучше заходите на форум сайта **http://www.vr-online.ru**. Отвечать на почту мне удается далеко не всегда, а вот на форум я захожу каждый день, и по возможности стараюсь помочь всем, у кого возникли сложности или вопросы.

ГЛАВА 1

Введение в Demo и DirectX

Прежде чем окунуться в великолепный мир программирования, давайте немного поговорим о демо-сцене и DirectX. Познакомившись с историей Demo, мы постараемся ощутить дух развития всей сцены. История сцены достаточно интересна и я рекомендую прочитать ее в любом случае.

Далее нас ждет введение и теория DirectX. Вот эту часть можно пропустить, если вы уже имеете опыт программирования графики в Windows. В данной книге под графикой мы будем понимать именно DirectX, а не GDI (Graphic Device Interface, интерфейс графических устройств), если явно не указана используемая графическая технология.

1.1. История демо-сцены

Первые ролики для платформы PC содержали только видеоэффекты и не всегда имели звуковое сопровождение, потому что это были 80-е годы, и не каждый PC-совместимый компьютер имел звуковую карту. В те времена производители устанавливали только один маленький динамик PC Speaker, возможности которого ограничивались писком. Но впоследствии звук стал неотъемлемой частью демо, и программисты умудрялись создать шедевры даже на пищалке PC Speaker, а когда звуковая карта стала устанавливаться практически в каждый компьютер, качество звука значительно повысилось.

Здесь меня могут упрекнуть в искажении фактов, ведь даже в 80-е годы были компьютеры, которые могли воспроизводить достаточно качественный по сравнению с PC Speaker звук. Да, были платформы типа Amiga, которые также оказали на демо-сцену серьезное влияние, но мы рассматриваем именно PC-платформу, а она не была предназначена для игр и графики, и изначально здесь все ограничивалось ASCII-графикой и примитивным звуком.

Первые ролики создавали в основном крэкеры (взломщики программ) или профессиональные программисты (хакеры). Крэкеры создавали небольшие ролики, которые содержали информацию о самом крэкере или группе. Такие демо-ролики вставлялись во взломанные программы, и любой пользователь мог увидеть, благодаря кому программа стала бесплатной. Эти демо-ролики были максимально простыми, потому что основным их требованием было — минимальное количество места при максимальном эффекте (производительности и красоты). Именно крэкеров считают основателями культуры демосцены, по крайней мере на платформе PC.

Программисты расширили эту идею и стали создавать более сложные деморолики, которые стали существовать как отдельные программы. Цель таких роликов — поразить пользователя и показать невиданные возможности простого PC-совместимого компьютера.

В середине 90-х я увидел демо-ролик Second Reality от группы Future Crew, который был создан в 1993 году (рис. 1.1). После его просмотра у меня осталось не передаваемое словами ощущение восторга. На 386-м компьютере с частотой 40 МГц я смог увидеть действительно впечатляющие эффекты и великолепный звук. Это уже был целый видеоклип, в котором графика великолепно гармонировала со звуком, а главное — я поразился, как такой слабенький компьютер может рассчитывать такие сложные сцены в реальном времени.

Рис. 1.1. Скриншот ролика Second Reality

Я не могу сказать, что именно группа Future Crew стала новатором в новом поколении демо, но для меня и многих других именно эта "дема" перевернула жизнь. С тех пор я увлекся графикой и графическими эффектами и с тех времен я стал усиленно интересоваться этой культурой. Теперь демо-ролик — это не просто шанс программисту показать свои способности, но и продемонстрировать другим новые, невиданные возможности компьютера. Ролик Second Reality сочетал в себе умопомрачительную по тем временам 2D- и 3D-графику в сочетании с великолепной музыкой, и при этом не использовалось никаких графических ускорителей.

Период с 1993-го по 1995-й дал демо-сцене новый виток гонки за 3D-эффектами. Появление процессора Pentium позволило усложнить графику и создавать новые эффекты, которые рассчитывались в реальном времени, но при этом еще не использовалось никаких ускорителей, без которых мы сейчас не можем представить себе ни одну графическую игру или даже графическую программу.

Да, разрешение экрана в демо-роликах было не таким высоким и благодаря этому экономилось большое количество процессорного времени. И все же, для создания таких эффектов требовались немалые усилия, знания и умения программистов. Даже при небольшом разрешении ролики смотрелись очень эффектно, потому что в те времена стандартом был монитор 14 дюймов, а о 15 или 17 дюймов приходилось только мечтать, потому что стоили они очень много. Это сейчас мы сидим за 17-дюймовыми "трубами" или панелями и тут без графического ускорителя не обойтись, иначе при разрешении 320×200 точек на экране будет изображение из больших квадратов, а не точек.

Так уж повелось в нашей жизни, что программисты в большинстве случаев не умеют рисовать (у меня тоже проблемы с художеством), поэтому в создание демо-сцен стали включаться художники и музыканты. Теперь, для того чтобы поразить пользователя и тем более победить на конкурсе, необходима была полная гармония графических эффектов, текстур, звука и видеоряда. Именно поэтому над созданием роликов стали трудиться целые команды.

С увеличением мощности компьютеров усложнялись и демо-ролики. Этому способствовало и появление новых графических технологий, упрощающих программирование, таких как OpenGL и DirectX, а также появление графических ускорителей. Демо-ролики уже перестали быть двумерными и все больше покоряли третье измерение.

Некоторые программисты совершенствовались и шли в ногу со временем, а некоторые противились этому. Таким образом, демо-сцена стала разделяться на несколько основных направлений:

- классические демо-ролики, размер исполняемого файла которых не превышает 4 или 64 Кбайт и при этом не используется никаких графических технологий и ускорителей. В таких демо для создания сцены может использоваться только математика, и все расчеты должны производиться в реальном времени;
- 2D, 3D или смешанная демо-сцена;
- Demo с использованием графического ускорителя и технологии OpenGL;
- Demo с использованием графического ускорителя и технологии DirectX.

Самое сложное, на мой взгляд, это первое. Создать демо-ролик с хорошим эффектом в 4 Кбайт очень непросто. Конечно, задача усложняется, если использовать Windows, но и в MS-DOS эта задача не такая уж простая. И все же, виртуозы своего дела умудряются и на такие шедевры. Например, на сайте **http://www.scene.org** можно найти "демку" fr026, точнее сказать, это графический эффект. Размер исполняемого COM-файла — 34 байта, а исходный код на языке ассемблера умещается на одной странице (листинг 1.1).

Листинг 1.1. Исходный код "демки" fr026

```
; fr-026: 34b mul+cycle (uc 6.22 size optimizing compo entry)
; 1st place
; code: ryg (original concept) & kb (additional opcode crunching)
;
; rules were to write a "colorcycling" (no real colorcycling was
; required, only colorcycling-like animation) effect that display
; a x * y-ish pattern somewhere on the screen. the palette was
; required to look like the original version.

org  256

  mov  al, 13h  ; ...load al with 13h (mode number)
  int  10h      ; set mode 13h

  mov  bp, 320  ; screen is 320 pixels wide

x les  ax, [bx] ; es = end of mem (9fffh), ax = 20cdh
  cwd           ; => dx = 0 (for idiv)
  mov  ax, di   ; get dest address
  idiv bp       ; div by width => ax=y coord, dx=x coord
  imul dx       ; mul x by y
  stosb         ; store color in vram
  mov  dx, 3c9h ; dac data
  bt   cx, 6    ; bit 6 of cx => carry flag
  salc          ; set al from carry flag
  xor  ax, cx   ;=> all this is functionally equivalent to something like
                ; "test al, 64" / "jz skip" / "not al" / "skip: blah"
                ; which should be somewhat easier understandable,
                ; but bigger.
  out  dx, al   ; write grayscale pixel values
  out  dx, al
  out  dx, al
  loop x        ; pixel loop.
  loop x        ; skip cx = 0 to get the cycling effect!
```

```
; that's all there is to it.
; this is the 7th (or 8th) version; the first try was
; 49 bytes, from then on we successively made it smaller.
```

В листинге код эффекта показан как есть, но если из него убрать все комментарии, то его размер уменьшится почти в два раза.

Отдельной веткой шло развитие Amigo и ZX Spectrum демо-сцены, но мы эту тему опускаем в связи с досрочной смертью платформы, поэтому данное направление не рассматриваем.

Создание роликов с использованием графического ускорителя также могло разделяться на несколько направлений, в которых могло быть ограничение на размер исполняемого файла или не было этого ограничения.

В настоящее время разновидностей демо-сцены намного больше, здесь и Flash-демо, и создание видеороликов в графических пакетах, таких как 3ds Max, цифровые картинки, полностью созданные на компьютере, или даже просто цифровая музыка. Описать процесс создания всех разновидностей демо-сцены невозможно, потому что они безграничны. Можно ограничиться только теми направлениями, которые стали стандартом для показа на демовечеринках и конкурсах, но даже для рассмотрения этой темы нужна книга размером с "Войну и мир". Именно поэтому я ограничился только демороликами с использованием ускорителей и технологии DirectX, да и как я уже сказал, из меня художник и музыкант не получился, и для изучения этой стороны демо-сцены лучше почитать специализированную литературу.

Может показаться, что демо-сцена — это всего лишь видеоклип на какую-то тему и под определенную музыку. Если под словом "клип" понимать то, что мы видим на телеканале MTV, то к демо-сцене можно отнести единичные творения. Из российских клипов к сцене с натяжкой относятся клипы Глюкозы. Аниматоры тут серьезно постарались и работы получаются законченными и интересными, а по поводу стиля музыки — это дело вкуса, и в демосцене принимаются любые направления, но желательно использовать компьютерный звук.

Demo — это больше, чем просто звук и видео — это гармония, это отражение души, настроения и состояния авторов. Одна только музыка является отражением души человека, и по пристрастиям человека можно определить даже характер, который может быть, в принципе, изменчивым. Например, в детстве я любил песенки Шаинского, затем техно, рэп, Happy Hardcore, рок, а в последнее время опустился до попсы. Видимо жизнь стало попсой, а иногда и попой (мягко говоря) ☺.

Самое сложное в процессе создания демо-ролика — программирование, потому что для того, чтобы поразить зрителя, необходимо создать нечто неве-

роятно потрясающее и удивительное. Необходимо на компьютере с малой мощностью выдавать такие эффекты, которые только анонсируются производителями видеоускорителей и появятся в аппаратной поддержке лишь через несколько лет. Конечно, для этого необходимо очень хорошо разбираться в математике, особенно в геометрии, тригонометрии, матрицах и некоторых других областях. Мы же постараемся не опускаться до такой "низости", а воспользуемся возможностями современных видеочипов. Даже этого будет достаточно, чтобы реализовать интересные эффекты, и поразить зрителя.

Некоторые задаются вопросом — как разработчики демо-роликов умудряются создать графику лучше, чем в играх? Когда мы поняли, что такое ролик, можно легко получить ответ. Просто игра и ролик — это разные вещи. В роликах не нужен AI (Artificial Intelligence, искусственный интеллект), отображение графики идет по линейному сюжету, что значительно упрощает программирование. Получается, что у Demo меньше производственных расходов, и они могут выполнять больше вычислений в тот момент, когда игры обрабатывают AI-монстров.

Дополнительную информацию о демо-сцене можно найти на сайтах:

- **http://www.scene.org** — наверно самый крупный сервер демо-сцены. На этом сервере находится очень много домашних страничек различных демо-групп, где они выкладывают свои последние работы. А на FTP-сервере **ftp://ftp.scene.org** можно найти работы большинства некогда и ныне знаменитых групп и авторов одиночек. Если хотите посмотреть работы профессиональных групп, то советую заглянуть на **ftp://ftp.scene.org/pub /demos/groups/**;
- **http://www.demoscene.ru** — крупнейший российских портал о демо-сцене.

В качестве "контрольного выстрела" предлагаю запустить программу из папки Demo/kkrieger, размещенной на прилагаемом компакт-диске. Это 3D-игра в стиле Quake, которая занимает со всеми ресурсами менее 100 Кбайт (соответствующий скриншот показан на рис. 1.2). Да, она маленькая и ее можно пройти за день, но графика достаточно приемлемого качества, а размер поражает. Попробуйте создать что-нибудь хоть немного приближенное.

Если вас впечатлила эта игра, то рекомендую заглянуть на сайт разработчиков (**http://www.theprodukkt.com**), где можно найти еще несколько красивых демонстрационных программ.

Во время создания игры .kkrieger разработчики использовали множество методов оптимизации. Я не видел исходного кода и не прогонял ее через отладчик, поэтому могу только догадываться, что они сжали исполняемый файл чем-нибудь вроде PECompact, не использовали растровые текстуры, а генерировали их на лету (создание текстуры кодом отнимет меньше места) и ис-

пользовали в качестве эффектов звук минимального качества. Одни только звуковые эффекты не могут занимать менее 100 Кбайт, если их сделать CD-качества.

Рис. 1.2. Скриншот игры .kkrieger

И все же, для своего размера и при таком качестве, получается действительно шедевр, который еще не раз прогремит на всю ИТ-вселенную, если разработчики не бросят этот проект. С другой стороны, если они начнут работать в сторону повышения качества звука и изображений, то размер программы станет сильно увеличиваться и тогда проект потеряет "изюминку". Поэтому игре лучше оставаться такой, как она есть, ведь именно малый ее размер делает работу разработчиков действительно гениальной, и поэтому их можно смело называть хакерами.

1.2. Введение в DirectX

Со времен, когда MS Windows еще не был операционной системой (ОС), а только надстройкой для MS-DOS, в качестве программной основы для работы с графикой использовался интерфейс GDI (Graphic Device Interface, интерфейс графических устройств). На то время это была действительно удачная технология, с помощью которой можно было работать с любой видеокартой. На платформе PC было слишком большое разнообразие видеочипов с различными возможностями, и GDI предоставлял универсальный способ доступа к видеофункциям. Эта технология до сих пор используется в Windows, но в значительно переработанном виде.

Универсальность — это хорошо, но производительность видео оставляла желать лучшего. Когда я впервые увидел игру Doom, то поразился, почему игра может создавать сложнейшие сцены на компьютере с 386-м процессором, а Windows не может? Конечно же, разрешение игры ниже, но и сцены трехмерного мира намного сложнее. Производительность GDI — это черепаха по сравнению с прямым доступом к памяти. Основная проблема GDI кроется в том, что ни одно приложение не может получить прямой доступ к видеокарте и видеобуферу, иначе очень сложно будет реализовать многооконную систему, да и универсальность добавляет ложку дегтя и возможности хорошего видеочипа используются не на все 100%.

Из-за медленной графической системы платформа Windows оказалась непригодной для создания игр. Да, на самом деле, были разработки, ускоряющие графику, но разогнать черепаху даже с помощью установки вентилятора на панцире невозможно.

Чтобы решить проблему скорости, возникла необходимость в создании принципиально новой технологии, которая решала бы три основные проблемы:

1. Приложение должно иметь возможность получить прямой доступ к видеопамяти.
2. Возможности видеокарты нужно использовать на все 100%, тем более что не за горами было появление видеоускорителей, которые сейчас стали нормой, а в настоящее время функциями ускорителя уже наделили и видеокарты.
3. Приложение должно получать максимальные ресурсы процессора. ОС Windows многозадачная система, и процессор разделяется между многими процессорами, а в играх это неприемлемо.

Все эти проблемы достаточно эффективно решаются с помощью DirectX, которому предоставляются максимально возможные ресурсы компьютера и прямой доступ к видеобуферу, если приложение работает в полноэкранном режиме. На мой взгляд, это наиболее простая задача. Если процесс занимает весь экран, то ему можно отдать намного больше ресурсов, чем если приложение работает в окне. В оконном режиме ОС должна прорисовывать все, что находится в фоне, а именно графика является более слабым местом программы.

Наиболее сложная задача — использовать максимальные возможности видеочипа. На PC-совместимые компьютеры сейчас устанавливают чипы NVIDIA GeForce, ATI Radeon, Matrox и др. и все они несовместимы между собой. Одна видеокарта поддерживает одни возможности, а другая — совершенно другие. К тому же, есть еще видеочипы, графические возможности которых минимальны (например, чипы от Intel). Как же добиться универсальности? Можно реализовать в DirectX только те возможности, которые

есть во всех видеокартах, но что-то подобное уже было в GDI, и производительность оказалась минимальной. Можно реализовать то, что посчитаем нужным, но тогда программы будут работать не на всех компьютерах и возникнет множество проблем с видеокартами.

Хороших решений проблемы универсальности всего два:

1. *Предоставить интерфейс ко всем необходимым возможностям*. Если возможность поддерживается видеочипом аппаратно, то использовать ее, если же нет, — реализовывать ее программно.
2. *Реализовать то, что хочется, и для совместимости заставить разработчиков следовать спецификации*.

Изначально фирма Microsoft выбрала первый вариант, потому что видеоускорители стоили дорого и установлены были далеко не на всех компьютерах. Движок DirectX мог работать в двух режимах:

1. *HAL* (Hardware Abstraction Layer, уровень аппаратных абстракций) — этот уровень задействуется, если видеочип поддерживает необходимые функции аппаратно.
2. *HEL* (Hardware Emulation Layer, уровень эмуляции аппаратуры) — этот уровень задействуется, когда необходимая функция не поддерживается аппаратно.

Программный уровень (HEL) в DirectX 9 лучше не использовать в конечных приложениях. Вы можете включать его только на этапе разработки приложения. Простые возможности ускорения графики встроены в большинство современных видеочипов. При эмуляции сложных эффектов будет слишком большая нагрузка на центральный процессор, который может уже не справиться с необходимыми расчетами, и вывод графики станет крайне медленным.

На мой взгляд, DirectX — это одна из лучших технологий для игр. В настоящее время у нее только один серьезный конкурент — OpenGL, но возможности DirectX больше, потому что охватывают не только графику, но и звук, и сеть, а значит, для написания игр у DirectX есть все необходимое.

1.3. Установка и настройка DirectX

Для разработки графических приложений необходимо установить для Visual Studio пакет DirectX SDK (Software Development Kit, пакет разработчика приложений). Стандартная поставка среды разработки Visual Studio ничего не знает о функциях DirectX, поэтому необходимо сообщить о существовании этих функций. Для этого используются заголовочные файлы .h и библиотеки .lib, которые как раз и устанавливаются вместе с пакетом SDK.

Инсталляция DirectX SDK не составит проблем, потому что тут достаточно только запустить программу установки, согласиться с лицензионным соглашением и нажимать кнопку **Next**, пока установка не будет завершена. Единственное — запомните директорию, куда вы установили SDK.

Но установки недостаточно. Во время инсталляции DirectX SDK на ваш компьютер только скопируются необходимые файлы, а нужно еще сообщить среде разработки, где их искать. Рассмотрим настройку на примере Visual Studio .NET. Запускаем среду разработки и выбираем меню **Tools | Options** (Инструменты | Опции). Перед вами откроется окно настроек среды разработки. С левой стороны окна будет расположено дерево разделов настроек. В разделе **Projects** (Проекты) выбираем **VC++ Directories** (Каталог VC++) и в правой части окна видим настройки каталогов (рис. 1.3).

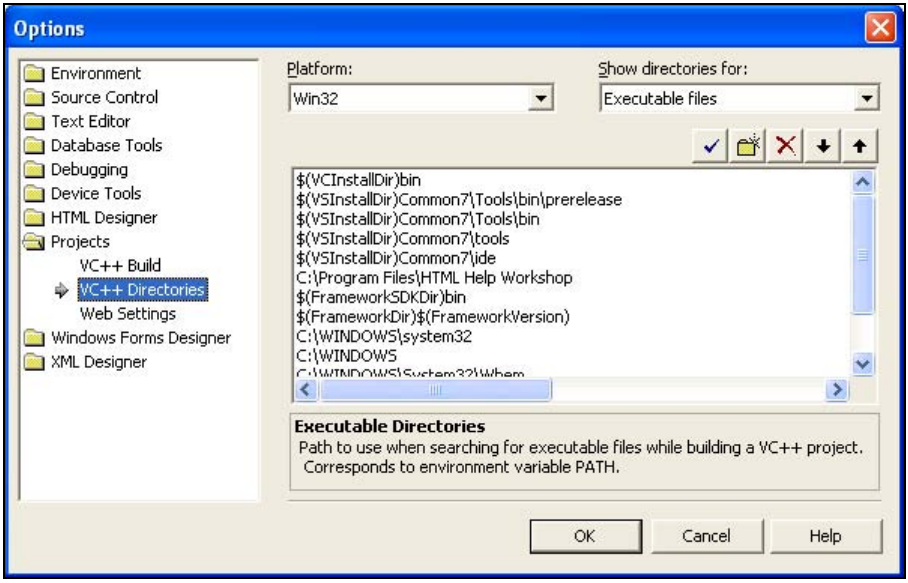

Рис. 1.3. Окно настройки каталогов в Visual Studio .NET

Для начала укажем каталог, где находятся заголовочные файлы .h. Они копируются в папку \Include каталога, в который вы установили DirectX SDK. В выпадающем списке **Show directories for** (Показать каталоги для) выберите **Include files** (Подключаемые файлы). Теперь нажмите кнопку **New line** (Новая строка) или клавиши <Ctrl>+<Ins> и в списке каталогов появится новая строка. Щелкните по кнопке с тремя точками и выберите каталог, где у вас находятся заголовочные файлы.

Теперь укажем каталог, где находятся библиотечные файлы .lib. Они копируются в папку \lib каталога, в который вы установили DirectX SDK. В выпа-

дающем списке **Show directories for** (Показать каталоги для) выберите **Library files** (Библиотечные файлы). Добавляем новую строку тем же методом, что и для заголовочных файлов, и выбираем каталог, в котором у вас находятся библиотечные файлы.

Теперь Visual Studio будет готова компилировать проекты, но не сможет собрать исполняемый файл. Для этого каждому проекту, который будет использовать DirectX, нужно явно указать необходимые библиотеки. Чтобы увидеть, как указать библиотеки, создайте новый проект и выберите меню **Project | Properties** (Проект | Свойства). Не пугайтесь, если перед именем пункта меню **Properties** (Свойства) будет имя вашего проекта, это нормально. Перед вами откроется окно настройки проекта, которое схоже с настройками среды разработки. Слева также будет находиться дерево разделов свойств.

В дереве свойств выбираем раздел **Configuration Properties | Linker | Input** (Свойства конфигурации | Сборщик | Входящие). Перед вами откроется окно, как на рис. 1.4.

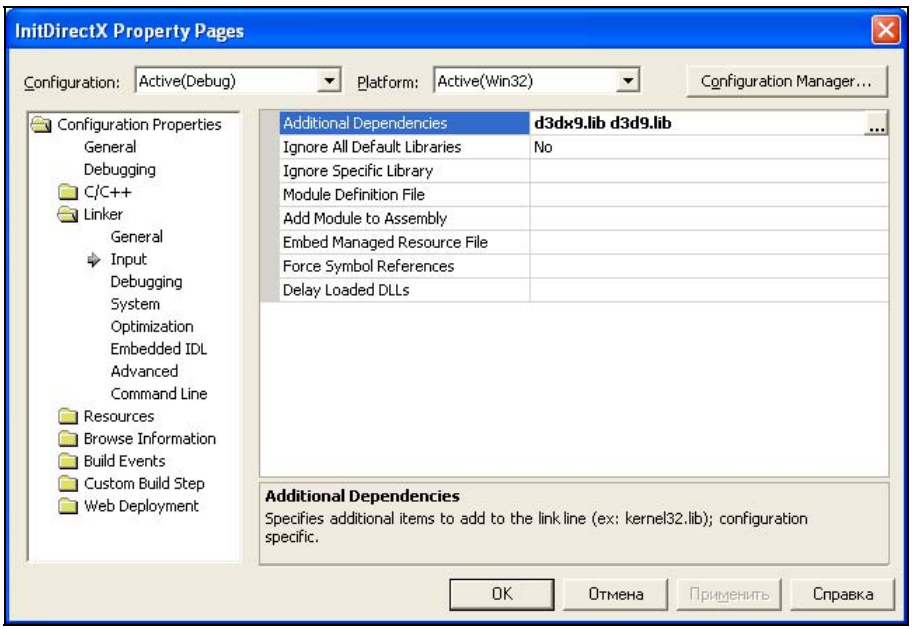

Рис. 1.4. Окно настройки свойств проекта в Visual Studio .NET

В строке **Additional Dependencies** (Дополнительные зависимости) необходимо указать библиотеки, которые нужно подключить во время сборки проекта. Выделите эту строку и щелкните по кнопке с тремя точками. Перед вами появится окно, в котором можно указать дополнительные библиотеки (рис. 1.5).

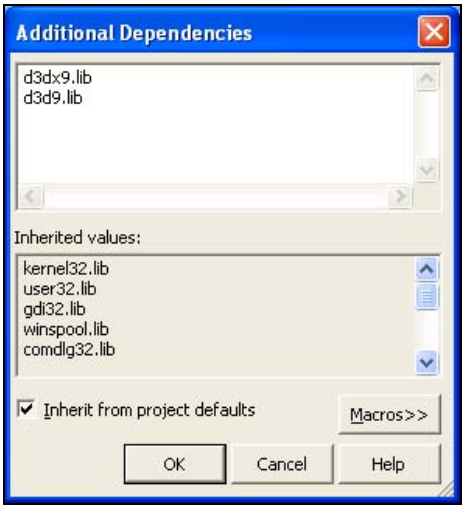

Рис. 1.5. Окно добавления библиотек

Для большинства проектов из данной книги необходимо как минимум указать библиотеки d3dx9.lib и d3d9.lib. Укажите каждую из них в отдельной строке и нажмите **ОК** для сохранения изменений.

Добавить библиотеки нужно для обеих конфигураций: Release и Debug. Последовательно выберите в выпадающем меню **Configuration** (Конфигурация) оба пункта и добавьте модули. Помимо этого, в разделе **Configuration Properties | C\C++ | Precompiled Header** (Свойства конфигурации | C\C++ | Предварительно скомпилированные заголовочные файлы) для обеих конфигураций установите в параметре **Create/Use precompiled Header** (Создавать/Использовать предварительно скомпилированные заголовочные файлы) параметр **Automatically Generate (/YX)** (Автоматически генерировать).

Помните, что пути, которые мы прописали в свойствах Visual Studio, нужно устанавливать только один раз, а библиотеки в свойствах проекта прописываются для каждого нового проекта.

Вот теперь среда разработки Visual Studio знает, где искать заголовочные и библиотечные файлы, а сборщик знает, какие дополнительные библиотеки необходимо использовать во время сборки проекта.

1.4. Введение в оптимизацию

Я всегда говорю, что если вы написали хорошую программу, то вы программист, а если вы сделали ее лучше других, то вы хакер. Как можно сделать программу лучше других? Один из способов — сделать ее быстрее и не требовательной к ресурсам компьютера. Демо-сцена получила такую популяр-

ность именно благодаря оптимизации, которая позволяет выжимать из компьютера все его соки, поэтому мы не можем обойти этот вопрос стороной.

Оптимизация в графике — достаточно сложный процесс, и о различных методах повышения производительности определенных алгоритмов мы будем говорить еще не один раз. Основа любой оптимизации — алгоритм. Если задачу можно решить несколькими способами и выбрать наиболее медленный, то затраты на оптимизацию будут выше, чем эффект от нее. Намного результативнее переписать код на более эффективный алгоритм.

В этом разделе мы опишем основные методы, которым будем следовать при написании кода. Эти методы относятся не только к графике, но и к любым другим приложениям. Об оптимизации можно также почитать в книгах "Программирование на C++ глазами хакера" [1] и "Программирование на Delphi глазами хакера" [2].

Глядя на создаваемые сейчас программы (особенно программистами одиночками), складывается впечатление, что программисты просто забыли про оптимизацию. Программисты наверно думают, что раз их творение в виде исходного кода никто не увидит, то можно писать что угодно. С этой точки зрения программы с открытым исходным кодом имеют большое преимущество, потому что они намного чище и иногда намного быстрее. Создавая код, мы ленимся его оптимизировать не только с точки зрения размера, но и с точки зрения скорости. Глядя на такие вещи, хочется ругаться матом, вот только программа от этого лучше не станет.

Хакеры, однако, тоже далеко не ушли. Если раньше, глядя на программиста или хакера, создавался образ прокуренного, заросшего и немытого молодого человека, то сейчас это цифровое существо, залитое пивом "Балтика" по самые уши, за которого все выполняют машины. Вам медсестра в поликлинике не говорила, что у вас вместо крови одно только пиво льется? Нет, разумеется, я ничего против пива не имею, я и сам его люблю, но надо же и меру знать!

Не надо тратить большие деньги на модернизацию компьютера!!! Начните прежде улучшения с себя. Давайте оптимизируем свою работу и то, что мы делаем, и тогда компьютер заработает намного быстрее.

Итак, рассмотрим законы, которым мы будем следовать при написании кода.

ЗАКОН № 1

Оптимизировать можно все. Даже там, где вам кажется, что все и так работает быстро, можно сделать еще быстрее.

Это действительно так. И этот закон очень сильно проявляется в программировании. Идеального кода не существует. Даже простую операцию сложения

2 + 2 тоже можно оптимизировать (например, использовать сдвиг). Чтобы достичь максимального результата, нужно действовать последовательно и желательно в том порядке, в котором мы будем рассматривать законы.

Помните, что любую задачу можно решить хотя бы двумя способами (или больше), и ваша задача выбрать наилучший метод, который обеспечит желаемую производительность и универсальность.

ЗАКОН № 2

Первое, с чего нужно начинать, — это с поиска самых слабых и медленных мест.

Зачем начинать оптимизацию с того, что и так работает достаточно быстро! Если вы будете оптимизировать сильные места, то можете встретить неожиданные конфликты. Да и эффект будет минимален.

Тут же я вспоминаю пример из своей собственной жизни. Где-то в 1996 году меня посетила одна невероятная идея — написать собственную игру в стиле Doom. Я не собирался ее делать коммерческой, а хотел только потренировать свои мозги на сообразительность. Четыре месяца невероятного труда, и нечто похожее на движок уже было готово. Я создал один голый уровень, по которому можно было перемещаться, и с чувством гордости побежал по коридорам.

Никаких монстров, дверей и атрибутики на нем не было, а тормоза ощущались достаточно значительные. Тут я представил себе, что будет, если добавить монстров и атрибуты, да еще и наделить все это AI... Вот тут чувство собственного достоинства поникло. Кому нужен движок, который при разрешении 320×200 (тогда это было круто!) в голом виде тормозит со страшной силой? Вот именно...

Понятное дело, что мой виртуальный мир нужно было оптимизировать. Целый месяц я бился над кодом и вылизывал каждый оператор моего движка. Результат — мир стал прорисовываться на 10% быстрей, но тормозить не перестал. И тут я увидел самое слабое место — вывод на экран. Мой движок просчитывал сцены достаточно быстро, а пробоиной был именно вывод изображения. Тогда еще не было шины AGP, и я использовал простую PCI-видеокарту от S3 с 1 Мбайтом памяти.

Пара часов колдовства, и я выжал из PCI все возможное. Откомпилировав движок, я снова загрузился в свой виртуальный мир. Одно нажатие клавиши "вперед", и я очутился у противоположной стены. Никаких тормозов, сумасшедшая скорость просчета и моментальный вывод на экран.

Как видите, моя ошибка была в том, что вначале я неправильно определил слабое место своего движка. Я месяц потратил на оптимизацию математики и

что в результате? Мизерные 10% прироста в производительности. Но когда я реально нашел слабое звено, то смог повысить производительность в несколько раз.

Именно поэтому я говорю, что надо начинать оптимизировать только со слабых мест. Если вы ускорите работу самого слабого звена вашей программы, то, может быть, и не понадобится ускорять другие места. Вы можете потратить дни и месяцы на оптимизацию сильных сторон и увеличить производительность только на 10% (что может оказаться недостаточным), или несколько часов на совершенствование слабой части и получить улучшение в 10 раз!

ЗАКОН № 3

Следующим шагом вы должны разобрать все операции по косточкам и выяснить, где происходят регулярно повторяющиеся операции. Начинать оптимизацию нужно именно с них.

Опять начнем рассмотрение этого закона с программирования. Допустим, что у вас есть следующий код (приведена просто логика, а не реальная программа):

1. `А:=А*2`;
2. `Б:=1`;
3. `Х:=Х+Б`;
4. `Б:=Б+1`;
5. `Если Б<100, то перейти на шаг 3.`

Любой программист скажет, что здесь слабым местом является первая строка, потому что там используется умножение. Это действительно так. Умножение всегда выполняется дольше, и если заменить его на сложение (`А:=А+А`) или еще лучше на сдвиг, то вы выиграете пару тактов процессорного времени. Но это только пара тактов и для процессора это будет незаметно.

Теперь посмотрите еще раз на наш код. Больше ничего не видите? А я вижу. В этом коде используется цикл: "Пока Б<100, то будет выполняться операция Х:=Х+Б". Это значит, что процессору придется выполнить 100 переходов с шага 5 на шаг 3. А это уже не мало. Как можно здесь что-то оптимизировать? Очень легко. Здесь у нас внутри цикла выполняется две строки: 3 и 4. А что если мы внутри цикла размножим их 2 раза:

2. `Б:=1`;
3. `Х:=Х+Б`;
4. `Б:=Б+1`;

5. `X:=X+Б;`

6. `Б:=Б+1;`

7. `Если Б<50, то перейти на шаг 3.`

Здесь мы разложили цикл на более маленький. Вторую и третью операцию мы повторили два раза. Это значит, что за один проход цикла выполняются два раза строки 3 и 4, и только после этого произойдет переход на строку 3, чтобы повторить операцию. Такой цикл уже нужно повторить только 50 раз (потому что за один раз выполняется два действия). Это значит, что мы сэкономили 50 операций переходов. Неплохо? А это уже несколько сотен тактов процессорного времени.

А что если внутри цикла написать строки 2 и 3 десять раз. Это значит, что за один проход цикла строки 2 и 3 будут вычисляться 10 раз, и мне понадобится повторить такой цикл только 10 раз, чтобы получить в результате 100. А это уже экономия 90 операций переходов.

Недостаток этого подхода — увеличился код нашей программы, зато повысилась скорость, и очень значительно. Этот подход очень хорош, но им не стоит злоупотреблять. С одной стороны, увеличивается скорость, а с другой — размер. А большой размер — это враг любой программы. Поэтому надо находить золотую середину.

ЗАКОН № 4

(Этот закон — расширение предыдущего)

Оптимизировать одноразовые операции — это только потеря времени. Сто раз подумай, прежде чем начать мучиться с редкими операциями людей, которые ленятся что-нибудь делать.

Так вот именно здесь вы можете проявлять свою врожденную леность в полном объеме. В данном случае крутым считается не тот, кто целый день промучился и ничего не добился, а тот, кто выполнил свою работу наиболее быстро и эффективно. И эти две вещи путать нельзя.

ЗАКОН № 5

Нужно знать "внутренности" компьютера и принципы его работы. Чем лучше вы знаете, каким образом компьютер будет выполнять ваш код, тем лучше вы сможете его оптимизировать.

Этот закон относится только к программированию. Тут трудно привести полный набор готовых решений, но некоторые приемы я постараюсь описать.

- Старайтесь поменьше использовать вычисления с плавающей запятой. Любые операции с целыми числами выполняются в несколько раз быстрее.
- Операции умножения и тем более деления также выполняются достаточно долго. Если вам нужно умножить какое-то число на 3, то для процессора будет легче три раза сложить одно и то же число, чем выполнить умножение. Хотя современные процессоры работают с умножением уже достаточно быстро и разница уже не так заметна.

  А как же тогда экономить на делении? Вот тут нужно знать математику. У процессора есть такая операция, как сдвиг. Вы должны знать, что процессор "думает" в двоичной системе, и числа в компьютере хранятся именно в ней. Например, число 198 для процессора будет выглядеть как 11000110. Теперь посмотрим, как работают операции сдвига.

  Сдвиг вправо — если сдвинуть число 11000110 вправо на одну позицию, то последняя цифра исчезнет и останется только 1100011. Теперь введите это число в калькулятор и переведите его в десятичную систему. Ваш результат должен быть 99. Как видите — это ровно половина числа 198. Вывод: когда вы сдвигаете число вправо на одну позицию, то вы делите его на 2.

  Сдвиг влево — возьмем то же самое число 11000110. Если сдвинуть его влево на одну позицию, то с правой стороны освободится место, которое заполняется нулем — 110001100. Теперь переведите это число в десятичную систему. Должно получиться 396. Что оно вам напоминает? Это 198 умноженное на 2.

  Вывод: *когда вы сдвигаете число вправо, то вы делите его на 2; когда сдвигаете влево, то умножаете его на 2*. Так что используйте эти сдвиги везде, где возможно, потому что сдвиги работают в несколько раз быстрее умножения и деления и даже сложения и вычитания.
- При создании процедуры не обременяйте их большим количеством входных параметров. Перед каждым вызовом процедуры ее параметры помещаются в специальную область памяти (стек), а после входа изымаются оттуда. Чем больше параметров, тем больше расходы на общение со стеком.

  Тут же нужно сказать, что вы должны действовать аккуратно и с самими параметрами. Не вздумайте пересылать процедурам переменные, которые могут содержать данные большого объема в чистом виде. Лучше передать адрес ячейки памяти, где хранятся данные, а внутри процедуры работать с этим адресом. Вот представьте себе ситуацию, когда вам нужно передать текст размером одного тома "Войны и мир"... Перед входом в процедуру программа попытается загнать все это в стек. Если вы не увидете его переполнение, то задержка точно будет значительная.

- В самых критичных моментах (как, например, вывод на экран) можно воспользоваться языком ассемблера. Даже встроенный в Delphi или C++ ассемблер намного быстрее штатных функций языка. Ну а если скорость в каком-то месте уж слишком критична, то код ассемблера можно вынести в отдельный модуль. Там его нужно откомпилировать с помощью компилятора TASM или MASM и подключить к своей программе.

  Ассемблер достаточно быстрая и компактная вещь, но писать довольно большой проект только на нем — это очень сложно. Поэтому я не советую им увлекаться, а использовать его только в самых критичных для скорости местах.

ЗАКОН № 6

Для сложных расчетов можно заготовить таблицы с заранее рассчитанными результатами и потом использовать эти таблицы в реальном режиме времени.

Когда появился первый Doom, игровой мир поразился качеству графики и скорости работы. Это действительно был шедевр программистской мысли, потому что компьютеры того времени не могли рассчитывать трехмерную графику в реальном времени. В те годы еще даже и не думали о 3D-ускорителях, и видеокарты занимались только отображением информации и не выполняли никаких дополнительных расчетов.

Как же тогда программистам игры Doom удалось создать трехмерный мир? Секрет прост, как и все в этом мире. Игра не просчитывала сцены, а все сложные математические расчеты были рассчитаны заранее и занесены в отдельную базу, которая запускалась при старте программы. Конечно же, занести все возможные результаты нельзя, поэтому база хранила основные результаты. Когда нужно было получить расчет значения, которого не было в заранее рассчитанной таблице, то бралось наиболее приближенное число. Таким образом, Doom получил отличную производительность и достаточное качество 3D-картинки.

ЗАКОН № 7

Лишних проверок не бывает.

Чаще всего оптимизация может привести к нестабильности исполняемого кода, потому что для увеличения производительности некоторые убирают ненужные на первый взгляд проверки. Запомните, что ненужных проверок не бывает! Если вы думаете, что какая-то нестандартная ситуация может и не возникнуть, то она не возникнет только у вас. У пользователя, который будет

использовать вашу программу, может произойти все, что угодно. Он непременно нажмет на то, на что не нужно или введет неправильные данные.

Обязательно делайте проверки всего того, что вводит пользователь. Делайте это сразу же, и не ждите, когда введенные данные понадобятся.

Не выполняйте проверки в цикле, а выносите их за его пределы. Любые лишние операторы `if` внутри цикла очень сильно влияют на производительность, поэтому по возможности проверки нужно делать до или после цикла.

Циклы — это слабое место любой программы, поэтому оптимизацию надо начинать именно с них и стараться не вставлять в них лишние проверки. Внутри циклических операций не должно выполняться ничего лишнего — ведь это будет повторено много раз!

ЗАКОН № 8

Не переусердствуйте с оптимизацией. Слишком большие затраты на ускорение выполнения кода могут свести на нет затраченные усилия. Ставьте перед собой реальные цели и задачи. Если у вас не получилось оптимизировать ваш код до необходимой степени, то нет смысла продолжать долгие попытки и мучения. Возможно, будет легче найти совершенно другой способ решения проблемы.

Не всегда получается добиться идеала, потому что оптимизация скорости и оптимизация качества чаще всего противоположные вещи. Лучше всего это заметно на примере программирования графики. Чтобы сцена (например, в компьютерных играх) выводилась на экран быстрее, можно сделать приближенные, но быстрые расчеты. Из-за этого компьютер быстро производит расчеты, изображение получается невысокого качества. Для повышения качества нужно больше времени, поэтому очень часто приходится выбирать что-то одно.

В графических редакторах жертвуют скоростью, потому что тут не требуются расчеты реального времени. А вот в играх жертвовать приходится качеством, иначе играть будет невозможно, и вам просто не заплатят.

ЗАКОН № 9

Не доверяйте оптимизатору компилятора.

Да, современные компиляторы обладают хорошими средствами оптимизации, но я не думаю, что самый лучший оптимизатор сможет поднять производительность более чем на 10%. Действительно, в графических программах каждый процент может обеспечить необходимые ресурсы, которые позволят графике отображаться без задержек, и все же надеяться на это не стоит.

Если же у вас сам исходный код не оптимизирован, то ни один автоматический оптимизатор не сможет сделать его лучше. Чтобы поднять производительность, сначала нужно оптимизировать логику, просмотреть все циклы, т. е. улучшить код самостоятельно. Только в том случае, когда все это выполнено, и вы не видите больше вариантов для повышения скорости, вот тогда можно воспользоваться автоматической оптимизацией.

У каждого компилятора есть набор ключей, с помощью которых можно включить ту или иную оптимизацию. Попробуйте поиграть с параметрами и посмотреть на результат. Обязательно тестируйте создаваемый двоичный код, бывают случаи (редко, но бывают), что оптимизация снижает скорость работы.

Еще один вариант повышения скорости — сменить компилятор. Можно взять компилятор более новой или старой версии. Иногда новые компиляторы добавляют к результирующему коду много "мусора". Этим страдают визуальные средства разработки от Borland. Дабы упростить разработку и ускорить компиляцию, в двоичный файл попадает слишком много лишнего кода, который может никогда не понадобиться. Тем же самым страдает и Visual C++, если использовать библиотеки MFC (Microsoft Foundation Classes, библиотека базовых классов Microsoft). Именно поэтому лучше отказаться от использования объектного кода, а применять только стандартный C или Паскаль без единого объекта.

Иногда может помочь и смена компилятора. Каждый компилятор создает свой собственный код, и скорость может быть разной. Существует множество тестов на производительность результирующего кода, но лично я никогда им не доверял. Лучше всегда доверяться своему опыту, потому что все зависит от конкретной задачи. Лично мне больше нравится код, создаваемый компилятором Intel, но это личное предпочтение, потому что чаще всего приходится использовать компилятор от Microsoft, как более распространенный.

Если вы хотите добиться максимальной производительности, то вам я все же рекомендую обратить внимание на компилятор от Intel. Дело в том, что он обновляется достаточно часто, и именно в нем в первую очередь появляются функции и методы использования новых возможностей процессоров. Например, если воспользоваться старым компилятором от любого производителя, то он скорей всего ничего не будет знать о современных возможностях процессора, таких как SSE (Streaming SIMD[1] Extension), а может даже и MMX (MultiMedia eXtension). Современные наборы инструкций направлены на повышение производительности программного обеспечения, так почему же этим не воспользоваться?

[1] SIMD — Single Instruction Multiple Data.

В ближайшее время нас ждет наводнение многоядерных процессоров, а чтобы задействовать эти возможности, необходимо, чтобы о них знал компилятор. А кто знает о новинках лучше, чем компилятор от производителя железа? А если этот производитель еще и лидер на рынке домашних систем (на которые мы ориентируемся) и является законодателем мод, то такому компилятору, как минимум, можно доверять.

С другой стороны, мы ориентируемся на ОС Microsoft Windows и используем технологию DirectX, которая также принадлежит этому производителю. Так почему же не воспользоваться компилятором этого производителя? Вот так мы и встали перед выбором — что же использовать? Как я уже сказал, не стоит обращать внимание на тесты, пусть и независимые. Все мы знаем, как эти тесты производятся. Доверяйте только своему опыту и попробуйте оба варианта.

1.5. Инициализация Direct3D

Теперь мы готовы рассмотреть инициализацию DirectX. Начнем с Direct3D (графическая часть Microsoft DirectX), потому что он на сегодняшний момент более актуален и ему мы будем уделять больше внимания. Чтобы разговор был интереснее, напишем пример, а потом познакомимся с его возможностями.

Итак, создайте пустое приложение **File | New | Project** (Файл | Новый | Проект) и в появившемся окне выберите в дереве **Project Type** (Тип проекта) пункт **Visual C++ Projects | Win32** (Проекты Visual C++ | Win32). Чтобы наши программы были небольшими и быстрыми, мы не будем использовать MFC, поэтому выбираем пункт **Win32 Project** (Проект Win32) (рис. 1.6). В поле **Name** (Имя) укажите имя проекта, а в поле **Location** (Месторасположение) — путь, по которому будет сохранен проект.

После нажатия кнопки **OK** перед нами открывается мастер создания приложения. В разделе **Application Settings** (Настройки приложения) (рис. 1.7) выберите тип приложения, установив флажок **Windows application** (Приложение Windows) в группе **Application type** (Тип приложения), а все остальные флажки должны быть сняты.

Теперь наше базовое приложение готово. Не забудьте в свойствах проекта подключить библиотеки.

Теперь подчистим проект от лишнего. Для начала необходимо убрать из меню пункт **Help** (Справка), а в коде обработчик для него.

Идеальное место для инициализации Direct3D — сразу после создания и отображения окна в функции `InitInstance`. Надо только определиться, как будет

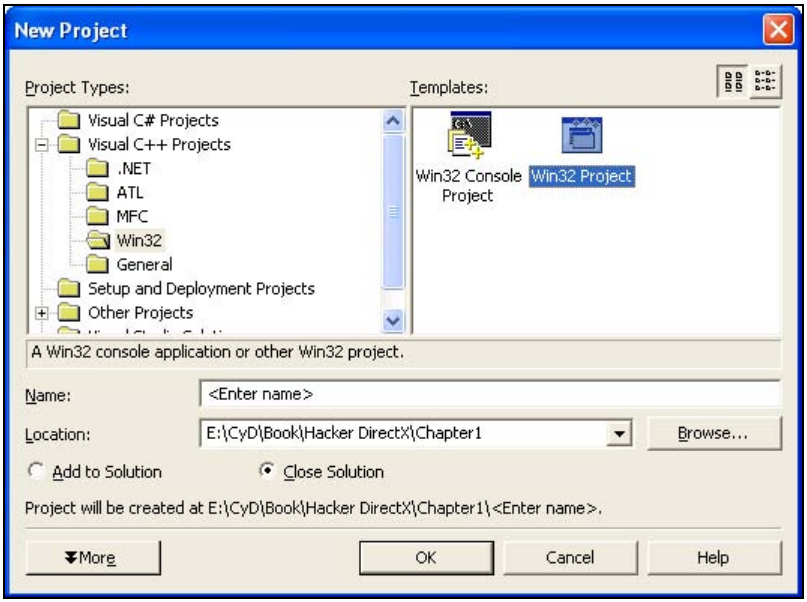

Рис. 1.6. Окно создания нового проекта

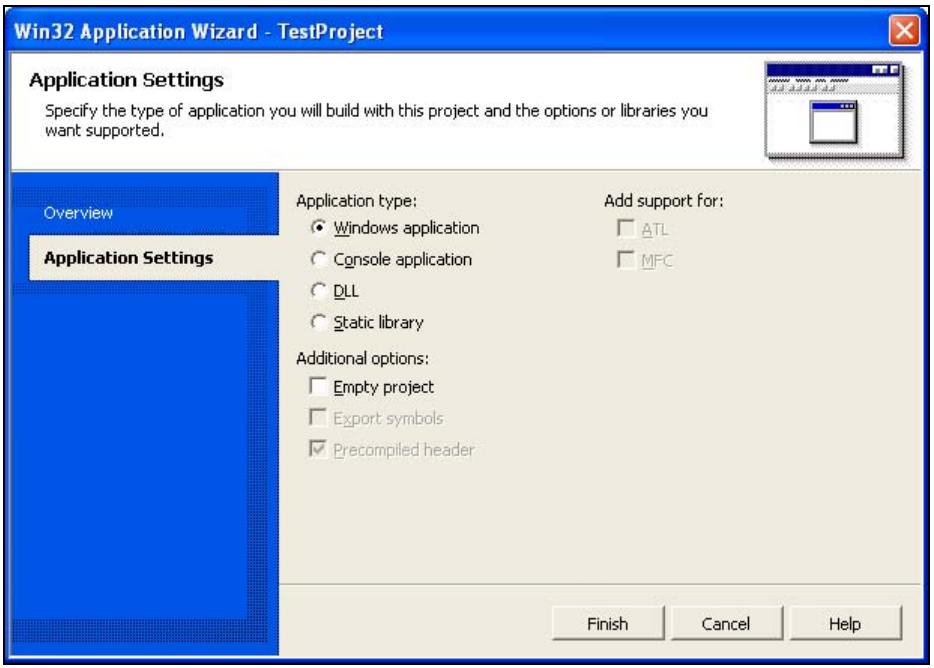

Рис. 1.7. Мастер создания приложения

происходить инициализация. Поскольку это одноразовая операция, которая будет выполняться только при старте приложения, то можно написать отдельную универсальную функцию, и ни коем образом не экономить на проверках корректности выполнения операций.

Так как нам придется написать несколько приложений и код инициализации Direct3D будет идентичным, давайте создадим модуль, в котором и будет наша функция инициализации. Создайте файлы для исходного кода dxfunc.cpp и заголовочный файл dxfunc.h. Содержимое заголовочного файла можно увидеть в листинге 1.2.

Листинг 1.2. Содержимое заголовочного файла dxfunc.h

```
#ifndef _DXFUNC_H_
#define _DXFUNC_H_

#include "d3d9.h"
#include "d3dx9.h"

// Функция инициализации Direct3D
HRESULT DX3DInit (IDirect3D9 **ppiD3D9,
      IDirect3DDevice9 **ppiD3DDevice9,
      HWND hWnd,
      DWORD iWidth,
      DWORD iHeight,
      BOOL bFullScreen
      );

#endif
```

В этом файле мы объявляем функцию `DX3DInit` со следующими параметрами:

- `ppiD3D9` типа `IDirect3D9` — указатель на указатель для хранения интерфейса `IDirect3D` 9-й версии;
- `ppiD3DDevice9` типа `IDirect3DDevice9` — указатель на указатель для хранения интерфейса устройства `IDirect3DDevice` 9-й версии;
- `hWnd` — окно, в котором будет происходить отображение Direct3D-сцены;
- `iWidth` и `iHeight` — желаемая ширина и высота окна;
- `bFullScreen` — параметр, указывающий на то, нужно ли выводить программу в полноэкранном режиме. Если да, то сразу будет происходить переключение на весь экран, иначе пользователь увидит запрос на выбор нужного режима. Чаще всего создаются именно полноэкранные приложения, без поддержки оконного режима. Дело в том, что работа в этих двух

режимах происходит немного по-разному, поэтому если этот параметр уже равен `bFullScreen`, то оконный режим предлагаться не будет.

Очень интересными являются первые два параметра. Это указатели на указатели. Почему именно так? Нам нужно проинициализировать Direct3D и сохранить в этих переменных указатели на интерфейсы. По идее, мы должны были бы передать просто указатели на переменные. Но тут есть проблема — дело в том, что через эти параметры будут передаваться переменные, которые должны инициализироваться в функции. Но переменные, переданные через параметры, нельзя изменять. Точнее сказать, изменять их можно сколько угодно, но после выхода из функции их значение восстановится на исходное. Это связано с тем, что переменные копируются в стек, и внутри функции изменения будут происходить с копией.

Эффективное решение проблемы — использовать указатель, на указатель — `**Переменная`, где `*Переменная` — это переменная для хранения указателя на интерфейс. Если изменить это значение, то на выходе оно восстановится. Но если изменить значение, на которое указывает указатель, то оно не будет сохранено. Дело в том, что в этом случае мы изменяем не переменную в стеке, а значение в памяти, на которое указывает указатель.

Как все запутано. Рассмотрим процесс передачи параметров на примере. Допустим, что в основном модуле мы объявляем переменную `*pD3D` для хранения указателя на интерфейс `IDirect3D9`, который в изначальном состоянии равен нулю. Если передать эту переменную напрямую в функцию, то там в переменную будет записан адрес интерфейса Direct3D, но после выхода адрес будет снова равен нулю.

Если передать указатель на `*pD3D`, то сам указатель `**pD3D` изменяться не будет и не нужно. Зато изменится значение переменной `*pD3D`, и это значение не может сброситься.

Давайте сразу же напишем в нашем шаблоне DirectX-приложения вызов функции инициализации, а потом уже напишем и разберем саму функцию. Для этого в основном модуле программы объявим две переменные для хранения создаваемых интерфейсов на `IDirect3D9` и устройство `IDirect3DDevice9`:

```
IDirect3D9       *pD3D       = NULL;
IDirect3DDevice9 *pD3DDevice = NULL;
```

Вызов функции инициализации пишем в функции `InitInstance` после создания окна, но перед отображением и перерисовкой главного окна:

```
// Вызов функции инициализации
// Если результат работы функции не равен S_OK,
// то произошла ошибка
```

```
if (DX3DInit(&pD3D, &pD3DDevice, hWnd, 800, 600, FALSE)!=S_OK)
{
   MessageBox(hWnd, "Ошибка инициализации DirectX",
       "Error", MB_OK);
   return FALSE;
}

// Отображение и обновление окна
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
```

Почему необходимо вызывать функцию именно здесь? Дело в том, что мы пишем универсальную функцию инициализации, которая будет запрашивать с помощью окна диалога, необходимо ли инициализироваться в полноэкранном режиме или нет. Вы должны учитывать, что все окна диалога после инициализации будут отображаться неверно, если вообще покажутся на экране. Эту проблему решить не сложно, просто нужно вывести окно до каких-либо инициализаций.

Но есть еще одна проблема — зачем отображать и перерисовывать окно, когда его инициализация не завершена? Это бессмысленно, ведь во время инициализации Direct3D изменяются параметры окна, и потом снова потребуется перерисовка 3D-сцены. Пусть это и одноразовые, но все же бессмысленные затраты процессорного времени на предварительную прорисовку.

Если ваша программа будет работать как в оконном, так и в полноэкранном режиме, то это станет только плюсом. Например, у меня ноутбук с широким экраном, и некоторые графические программы в полноэкранном режиме работают не очень хорошо. Проблему решает оконный режим, потому что видеокарта у меня достаточно мощная и позволяет использовать его. Но если полноэкранного режима в программе не предусмотрено, то приходится смотреть на картинку, немного растянутую по горизонтали.

Теперь посмотрим на содержимое файла dxfunc.cpp (листинг 1.3) и соответственно функции `DX3DInit`. В файле у нас находится только подключение заголовочного файла dxfunc.h и непосредственно сама функция инициализации.

Листинг 1.3. Содержимое файла dxfunc.cpp

```
#include "dxfunc.h"

// Функция инициализации Direct3D
HRESULT DX3DInit(IDirect3D9 **ppiD3D9,
              IDirect3DDevice9 **ppiD3DDevice9,
```

```
                HWND hWnd,
                DWORD iWidth,
                DWORD iHeight,
                BOOL bFullScreen)
{
  // Инициализация
  if((*ppiD3D9 = Direct3DCreate9(D3D_SDK_VERSION)) == NULL)
    return E_FAIL;

  // Заполняем основные параметры
  D3DPRESENT_PARAMETERS d3dpp;
  ZeroMemory(&d3dpp, sizeof(d3dpp));

  d3dpp.BackBufferWidth = iWidth;
  d3dpp.BackBufferHeight = iHeight;

  // Запрос на отображение в полноэкранном режиме
  int  iRes;
  if (!bFullScreen)
     iRes=MessageBox(hWnd, "Use fullscreen mode?",
             "Screen", MB_YESNO | MB_ICONQUESTION);
  else
      iRes = IDYES;


  if(iRes == IDYES)
   {
      ////////////////////////////////////////////////////////
      // Полноэкранный режим
      ////////////////////////////////////////////////////////
      // Установка параметров полноэкранного режима
      d3dpp.BackBufferFormat = D3DFMT_R5G6B5;
      d3dpp.SwapEffect       = D3DSWAPEFFECT_FLIP;
      d3dpp.Windowed         = FALSE;
      d3dpp.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;
      d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_DEFAULT;
  }
 else
  {
      ////////////////////////////////////////////////////////
      // Оконный режим
      ////////////////////////////////////////////////////////
      RECT wndRect;
      RECT clientRect;
```

```
      GetWindowRect(hWnd, &wndRect);
      GetClientRect(hWnd, &clientRect);

      int iWinWidth  = iWidth + (wndRect.right-wndRect.left)  -
            (clientRect.right-clientRect.left);
      int iWinHeight = iHeight + (wndRect.bottom-wndRect.top) -
            (clientRect.bottom-clientRect.top);

      MoveWindow(hWnd, wndRect.left, wndRect.top,
              iWinWidth, iWinHeight, TRUE);

      // Получить формат пиксела
      D3DDISPLAYMODE d3ddm;
      (*ppiD3D9)->GetAdapterDisplayMode(D3DADAPTER_DEFAULT, &d3ddm);

      // Установка параметров
      d3dpp.BackBufferFormat = d3ddm.Format;
      d3dpp.SwapEffect       = D3DSWAPEFFECT_DISCARD;
      d3dpp.Windowed         = TRUE;
  }

  // Создать 3D-устройство
  DWORD Flags= D3DCREATE_MIXED_VERTEXPROCESSING |
    D3DCREATE_MULTITHREADED;

  HRESULT hRes;
  if(FAILED(hRes = (*ppiD3D9)->CreateDevice(
     D3DADAPTER_DEFAULT,
     D3DDEVTYPE_HAL, hWnd, Flags,
     &d3dpp, ppiD3DDevice9)))
     return hRes;

  // Установить перспективу
  float Aspect =
      (float)d3dpp.BackBufferWidth / (float)d3dpp.BackBufferHeight;

  D3DXMATRIX matProjection;
  D3DXMatrixPerspectiveFovLH(&matProjection, D3DX_PI/4.0f,
      Aspect, 10.0f, 1000.0f);
  (*ppiD3DDevice9)->SetTransform(D3DTS_PROJECTION, &matProjection);

  return S_OK;
}
```

Давайте рассмотрим содержимое листинга и познакомимся с функциями инициализации Direct3D.

Первое, что мы делаем, — создаем экземпляр интерфейса `IDirect3D9` с помощью функции `Direct3DCreate9` и сохраняем результат в переменной `*ppiD3D9`:

```
if((*ppiD3D9 = Direct3DCreate9(D3D_SDK_VERSION)) == NULL)
  return E_FAIL;
```

Этот интерфейс используется для создания объектов Direct3D и для настройки окружения. В качестве параметра необходимо передать константу `D3D_SDK_VERSION`. Эта константа описана в заголовочном файле d3d9.h и ее значение увеличивается в каждой версии (на данный момент у меня константа равна 31). По значению константы DirectX определяет, является ли ваш заголовочный файл корректным. Если значение не будет совпадать, то функция вернет ошибку. Проблему решит перекомпиляция программы с корректным заголовочным файлом.

Далее заполняется структура типа `D3DPRESENT_PARAMETERS`, которая определяет параметры представления. Прежде чем использовать структуру, мы заполнили ее нулевыми значениями, чтобы незаполненные поля использовали значение по умолчанию:

```
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory(&d3dpp, sizeof(d3dpp));
```

В общем виде структура выглядит следующим образом:

```
typedef struct _D3DPRESENT_PARAMETERS_ {
    UINT                    BackBufferWidth;
    UINT                    BackBufferHeight;
    D3DFORMAT               BackBufferFormat;
    UINT                    BackBufferCount;

    D3DMULTISAMPLE_TYPE     MultiSampleType;

    D3DSWAPEFFECT           SwapEffect;
    HWND                    hDeviceWindow;
    BOOL                    Windowed;
    BOOL                    EnableAutoDepthStencil;
    D3DFORMAT               AutoDepthStencilFormat;
    DWORD                   Flags;

    UINT                    FullScreen_RefreshRateInHz;
    UINT                    FullScreen_PresentationInterval;
} D3DPRESENT_PARAMETERS;
```

Чтобы вы лучше понимали, какие параметры мы устанавливали и зачем, давайте рассмотрим поля этой структуры:

- `BackBufferWidth` и `BackBufferHeight` — ширина и высота заднего буфера. Если вы используете полноэкранный режим, то значения параметров должны быть равны размерам устанавливаемого разрешения. Если используется оконный режим, то значения параметров могут быть нулевыми и в этом случае буфер установлен в соответствии с размерами клиентской части окна, которое будет использоваться для вывода Direct3D-графики;
- `BackBufferFormat` — формат пиксела. Если вы используете оконный режим, то формат должен соответствовать текущему установленному значению в настройках Windows, потому что в оконном режиме нельзя поменять формат пиксела для отдельного окна. В случае с полноэкранным режимом мы можем использовать любое из возможных значений. Наиболее часто используются следующие значения (устаревшие форматы в 8 бит, редко используемые опустим):
  - `D3DFMT_R8G8B8` — 24 бит, по 8 бит на цвет (красный, зеленый и голубой);
  - `D3DFMT_A8R8G8B8` — 32 бит, по 8 пикселов на канал (красный, зеленый, голубой и альфа);
  - `D3DFMT_R5G6B5` — 16 бит, 5 бит на красный, 6 бит на зеленый и 5 бит на голубой;
  - `D3DFMT_X1R5G5B5` — 16 бит, по 5 бит на каждый цвет и 1 бит зарезервирован;
  - `D3DFMT_A1R5G5B5` — 16 бит, по 5 бит на каждый цвет и 1 бит зарезервирован для альфа канала;
- `BackBufferCount` — количество задних буферов. Здесь можно указать значение от 0 до 3, только 0 указывать нет смысла, потому что это значение будет трактоваться как 1;
- `MultiSampleType` — определяет тип переключения буферов. Этот параметр имеет смысл только при использовании эффекта переключения (параметр `SwapEffect`) `D3DSWAPEFFECT_DISCARD`;
- `SwapEffect` — тип переключения буферов. Здесь можно указать одно из следующих значений:
  - `D3DSWAPEFFECT_DISCARD` — после переключения буфера содержимое заднего буфера сбрасывается. Иногда это бывает очень удобно. При использовании других режимов буферы меняются местами без сброса содержимого;
  - `D3DSWAPEFFECT_FLIP` — переключение буфера, при котором задний буфер становится первичным без необходимости копировать данные.

Этот метод наиболее быстрый, но может использоваться только при полноэкранном режиме;

- D3DSWAPEFFECT_COPY — копирование заднего буфера на передний. Параметр может использоваться при наличии одного заднего буфера. Во время копирования не происходит синхронизации с вертикальным обновлением экрана VSYNC, поэтому без дополнительных проверок может возникнуть эффект бегущей полосы;
- D3DSWAPEFFECT_COPY_VSYNC — копирование заднего буфера на передний с вертикальной синхронизацией;
- D3DSWAPEFFECT_FORCE_DWORD — не используется;

- hDeviceWindow — окно, которое будет использоваться для вывода графики. Если параметр равен нулю, то применяется активное окно. При полноэкранном режиме используется все окно;
- Windowed — если параметр равен TRUE, то будет использоваться оконный режим, иначе — полноэкранный;
- EnableAutoDepthStencil — если параметр равен TRUE, то DirectX будет автоматически управлять буфером глубины;
- AutoDepthStencilFormat — формат буфера глубины. Здесь могут быть такие же значения, как у рассмотренного ранее параметра BackBufferFormat;
- Flags — дополнительные флаги;
- FullScreen_RefreshRateInHz — частота обновления экрана, используется только в полноэкранном режиме. Параметр должен содержать значение, которое будет поддерживаться данным компьютером. Для оконного приложения в этом параметре должно быть значение или константа D3DPRESENT_RATE_DEFAULT (значение по умолчанию), что одно и то же;
- FullScreen_PresentationInterval — интервал представления в полноэкранном режиме. Здесь можно указать одно из следующих значений:
  - D3DPRESENT_INTERVAL_DEFAULT — интервал по умолчанию;
  - D3DPRESENT_INTERVAL_IMMEDIATE — начать отображение немедленно, драйвер не будет дожидаться периода вертикального восстановления (vertical retrace period);
  - D3DPRESENT_INTERVAL_ONE — ожидать период вертикального восстановления, но отображение не может происходить чаще, чем обновляется экран;
  - D3DPRESENT_INTERVAL_TWO, D3DPRESENT_INTERVAL_THREE и D3DPRESENT_INTERVAL_FOUR — ожидать период вертикального восстановления, но

отображение не может происходить чаще каждого второго, третьего или четвертого обновления экрана соответственно.

Сразу же после заполнения структуры `d3dpp` нулями мы заполняем основные свойства, которые будут идентичны для полноэкранного и оконного режимов, а именно — ширина и высота заднего буфера:

```
d3dpp.BackBufferWidth = iWidth;
d3dpp.BackBufferHeight = iHeight;
```

Здесь мы задаем ширину и высоту буфера. Далее запрашиваем у пользователя с помощью окна `MessageBox`, необходимо ли отобразить окно в полноэкранном режиме? Если да, то продолжаем заполнять структуру параметрами, которые должны использоваться в полноэкранном режиме:

```
d3dpp.BackBufferFormat = D3DFMT_R5G6B5;
d3dpp.SwapEffect       = D3DSWAPEFFECT_FLIP;
d3dpp.Windowed         = FALSE;
d3dpp.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;
d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_DEFAULT;
```

В качестве формата пиксела выбран `D3DFMT_R5G6B5`, при котором 16 бит для хранения цвета делятся в следующем соотношении: красный — 5 бит, зеленый — 6 бит и голубой — 5 бит. Шестнадцать бит позволяют использовать 65 535 цветов, а этого достаточно, чтобы создавать реалистичные эффекты, и при этом производительность будет довольно высокая.

Можно было бы выбрать 24 бита, где на каждый цвет выделяется по 8 бит (1 байт). В этом случае работать с цветами было бы проще, потому что машинные инструкции плохо функционируют с битами, и больше любят оперировать с байтом. Однако, выбрав 24 бита, работа станет медленнее, чем при 16 битах. Конечно же, если использовать 8 бит, то скорость будет еще выше, но мы не будем опускаться так низко, потому что 256 цветов — это слишком мало для графического приложения.

Помимо формата пиксела, для переключения буфера будем использовать метод `D3DSWAPEFFECT_FLIP`, который является самым быстрым, но может применяться только в полноэкранном режиме. Свойство `Windowed` устанавливаем в `FALSE`, потому что это полноэкранный режим.

И самое важное — устанавливаем частоту смены экрана в значение по умолчанию `D3DPRESENT_RATE_DEFAULT`. Для электронно-лучевых мониторов минимальной частотой, которая не будет создавать мерцания, является 85 Гц. Но не вздумайте указывать это значение, потому что у ЖК мониторов 60 Гц, как правило, является максимумом и приемлемым для глаз. Лучше выбрать значение по умолчанию или определить, какие режимы поддерживает монитор, и предложить пользователю выбрать оптимальное значение самому.

В случае с оконным режимом мы сначала должны изменить размеры окна так, чтобы они соответствовали выбранным размерам. Для этого определяем текущее положение окна и с помощью функции `MoveWindow` устанавливаем новые размеры:

```
RECT wndRect;
RECT clientRect;
GetWindowRect(hWnd, &wndRect);
GetClientRect(hWnd, &clientRect);

int iWinWidth  = iWidth + (wndRect.right-wndRect.left)  -
   (clientRect.right-clientRect.left);
int iWinHeight = iHeight + (wndRect.bottom-wndRect.top) -
   (clientRect.bottom-clientRect.top);

MoveWindow(hWnd, wndRect.left, wndRect.top,
        iWinWidth, iWinHeight, TRUE);
```

Обратите внимание, что мы корректируем размер окна с учетом клиентской области. Дело в том, что функция `MoveWindow` установит размер всего окна, а клиентская область окажется меньше желаемой. Чтобы именно клиентская область имела размеры 800×600, нужно увеличить эти размеры на ширину оборки, меню и других элементов управления, способных уменьшить размер клиентской области.

Единственная проблема, которую мы не учитываем в данном коде, — это то, что не проверяются возможные выходы окна за пределы экрана. Если окно выйдет за пределы, то у Direct3D могут возникнуть проблемы.

Поскольку у нас будет использоваться оконный режим, то мы не можем выбрать формат пиксела, который захочется, поэтому придется использовать тот формат, который сейчас установлен в Windows. Давайте определим, что же сейчас установлено. Для этого используется метод `GetAdapterDisplayMode` интерфейса `IDirect3D`. Этому методу нужно передать два параметра:

- адаптер, информацию о котором нужно получить. Мы будем использовать первичный адаптер, а для ссылки на него можно использовать константу `D3DADAPTER_DEFAULT`;
- переменную типа `D3DDISPLAYMODE` — это структура, в которую будет записан результат работы.

После выполнения метода `GetAdapterDisplayMode` текущий формат пиксела можно определить через свойство `Format` структуры `D3DDISPLAYMODE`. Следующий код показывает пример вызова метода `GetAdapterDisplayMode`:

```
D3DDISPLAYMODE d3ddm;
(*ppiD3D9)->GetAdapterDisplayMode(D3DADAPTER_DEFAULT,
    &d3ddm);
```

А вот теперь уже устанавливаем параметры представления, которые специфичны для оконного режима:

```
d3dpp.BackBufferFormat = d3ddm.Format;
d3dpp.SwapEffect       = D3DSWAPEFFECT_DISCARD;
d3dpp.Windowed         = TRUE;
```

Здесь мы устанавливаем формат пиксела, эффект переключения между буферами и указываем на необходимость использования оконного режима. Для оконных приложений оптимальным будет применение переключения между буферами `D3DSWAPEFFECT_DISCARD`. Однако вы должны учитывать, что задний буфер при этом сбрасывается.

Далее снова идет код, который идентичен для обоих режимов. Мы сформировали такую большую структуру представления ради создания устройства Direct3D. В этот момент будут созданы дополнительные буферы, и если установлен полноэкранный режим, то произойдет переключение режима в соответствии с выбранными параметрами.

Для создания Direct3D-устройства используется метод `CreateDevice` интерфейса `IDirect3D`, который мы инициализировали в самом начале. В общем виде метод выглядит следующим образом:

```
HRESULT CreateDevice(
    UINT Adapter,
    D3DDEVTYPE DeviceType,
    HWND hFocusWindow,
    DWORD BehaviorFlags,
    D3DPRESENT_PARAMETERS *pPresentationParameters,
    IDirect3DDevice9** ppReturnedDeviceInterface
);
```

Давайте рассмотрим параметры:

- `Adapter` — номер адаптера, который мы хотим использовать. В большинстве случаев достаточно использовать первичный адаптер и на него же будет ссылаться константа `D3DADAPTER_DEFAULT`;
- `DeviceType` — тип устройства. Здесь можно указать одно из следующих значений:
  - `D3DDEVTYPE_HAL` — использовать аппаратные возможности. Этот параметр является более предпочтительным, потому что мы уже договорились о направленности на современные видеокарты, которые аппаратно поддерживают DirectX 9;

- D3DDEVTYPE_REF — использовать программные возможности, но если есть возможность, можно применять специализированные инструкции процессора;
- D3DDEVTYPE_SW — использовать программное устройство;

- hFocusWindow — окно, которое будет использоваться для вывода графики. Это должно быть верхнее (поверх всех) или полноэкранное окно;
- BehaviorFlags — дополнительные параметры создания устройства;
- pPresentationParameters — это указатель на структуру типа D3DPRESENT_PARAMETERS, которую мы заполняли ранее;
- ppReturnedDeviceInterface — указатель на указатель на переменную типа IDirect3DDevice9, в которую будет сохранен интерфейс созданного устройства.

Пример вызова метода в нашей функции DX3DInit:

```
HRESULT hRes;
if(FAILED(hRes = (*ppiD3D9)->CreateDevice(
        D3DADAPTER_DEFAULT,
        D3DDEVTYPE_HAL, hWnd, Flags,
        &d3dpp, ppiD3DDevice9)))
        return hRes;
```

Устройство создано, и мы готовы к работе, но еще желательно настроить параметры камеры. Для этого используется функция D3DXMatrixPerspectiveFovLH, которая в общем виде выглядит следующим образом:

```
D3DXMATRIX* D3DXMatrixPerspectiveFovLH(
  D3DXMATRIX* pOut,
  FLOAT fovy,
  FLOAT Aspect,
  FLOAT zn,
  FLOAT zf
);
```

Рассмотрим параметры этой функции:

- pOut — указатель на структуру D3DXMATRIX, через которую мы получим результат работы;
- fovy — угол обзора;
- Aspect — соотношение сторон;
- zn — расстояние до ближней плоскости отсечения. Все, что находится ближе, считается невидимым;

- `zf` — расстояние до дальней плоскости отсечения. Все, что находится дальше, считается невидимым.

На рис. 1.8 графически показаны параметры этой функции. В точке P находится камера и из этой точки мы смотрим на мир. Все, что ближе плоскости ZN или дальше ZF, мы видеть не можем. Угол между двумя линиями взора, направленными под углом вниз и вверх, — это и есть значение `fovy`.

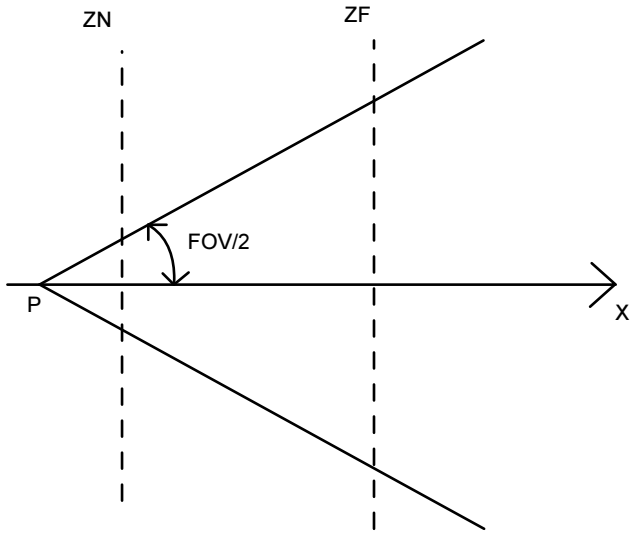

Рис. 1.8. Представление

Теперь необходимо установить созданную матрицу устройству `IDirect3DDevice`. Для этого используется метод `SetTransform`:

```
HRESULT SetTransform(
    D3DTRANSFORMSTATETYPE State,
    CONST D3DMATRIX *pMatrix
);
```

Тут имеется два параметра:

- `State` — состояние. Для изменения представления этот параметр должен быть равен константе `D3DTS_PROJECTION`. В этом параметре можно также указывать одну из следующих констант:
  - `D3DTS_VIEW` — установить матрицу просмотра;
  - `D3DTS_TEXTUREn` — установить матрицу n-й текстуры, где n может изменяться от 0 до 7;
- `pMatrix` — указатель на устанавливаемую матрицу.

Указатель на устройство у нас сохранен в переменной ppiD3DDevice9, а значит, вызов функции будет выглядеть следующим образом:

```
(*ppiD3DDevice9)->SetTransform(D3DTS_PROJECTION,
   &matProjection);
```

Теперь функцию инициализации можно считать завершенной. Попробуйте запустить программу и убедиться, что она работает. Пока что вы можете увидеть только переключение в полноэкранный режим и установку размеров. В остальном функция будет выглядеть так же, как и любое другое приложение.

Сразу же рассмотрим, как можно освободить выделенные ресурсы. То что мы берем, нужно обязательно отдавать, и ресурсы компьютера тут не являются исключением, поэтому эту тему нужно рассмотреть сразу, а не в самом конце.

Для освобождения интерфейсов Direct3D используется метод Release. Сначала необходимо вызвать этот метод, а потом обнулить переменную. При этом освобождение ресурсов должно идти в обратном созданию порядке. В нашем приложении сначала был создан интерфейс IDirect3D9, а затем IDirect3DDevice9. Это значит, что сначала нужно уничтожить IDirect3DDevice9, а потом уже IDirect3D9. Следующий код показывает, как можно освободить интерфейсы, запрошенные в нашем примере:

```
pD3DDevice->Release();
pD3DDevice= NULL;
pD3D->Release();
pD3D = NULL;
```

Но вызывать сразу метод Release опасно, ведь если интерфейс не был создан (возникла ошибка во время инициализации или просто интерфейс не был востребован), то произойдет ошибка. Это никогда не может быть приятным, поэтому перед вызовом Release необходимо проверить существование интерфейса:

```
if (pD3DDevice)
 {
   pD3DDevice->Release();
   pD3DDevice= NULL;
 }
if (pD3D)
 {
  pD3D->Release();
  pD3D = NULL;
 }
```

Для того чтобы не было лишних проблем, лучше будет во время объявления переменных интерфейсов DirectX сразу же указывать, что они изначально равны нулю:

```
IDirect3D9 *pD3D = NULL;
IDirect3DDevice9 *pD3DDevice = NULL;
```

Для упрощения жизни можно написать макрос, который абсолютно не повлияет на скорость работы программы, но я не буду этого делать.

Примечание

Исходный код примера можно увидеть в каталоге \Chapter1\InitDirect3D на прилагаемом компакт-диске. Файл dxfunc.cpp можно найти в каталоге \Common.

1.6. Инициализация DirectDraw

С помощью Direct3D можно создавать не только 3D-сцены, но и трехмерные изображения. Да это не так уж и сложно, но не всегда эффективно использовать тяжеловесный Direct3D, когда вся сцена просчитывается вами самостоятельно, а DirectX нужен только для получения прямого доступа к видеопамяти. В этих случаях проще обратиться к старому, проверенному временем DirectDraw и его интерфейсу IDirectDraw, который специально предназначен для работы с 2D-графикой.

Даже несмотря на то, что интерфейс IDirectDraw уже давно не развивается, все необходимые библиотеки и заголовочные файлы сохранились, и вы можете их применять. На первый взгляд, в DirectDraw инициализация сложнее, но иногда затраты окупаются простотой дальнейшего использования.

Давайте создадим базовое приложение, которое будет инициализировать DirectDraw. Для этого создаем новое пустое приложение, как и в *разд. 1.5*. Удалите из него диалоговое окно **О программе** и соответствующий код, чтобы уменьшить код программы. Теперь добавим в свойствах проекта необходимые библиотеки, а нам понадобятся dxguid.lib и ddraw.lib.

Функцию инициализации DirectDraw вынесем в отдельный модуль. Для этого создадим заголовочный файл ddfunc.h и файл кода ddfunc.cpp. Заголовочный файл будет содержать описания используемых функций, а текст его показан в листинге 1.4.

Листинг 1.4. Заголовочный файл ddfunc.h

```
#ifndef _DDFUNC_H_
#define _DDFUNC_H_
```

```
#include "ddraw.h"

// Функция инициализации Direct3D
HRESULT DXDDInit(IDirectDraw7 **ppiDD,
        IDirectDrawSurface7 **primsurf,
        IDirectDrawSurface7 **backsurf,
        HWND hWnd,
        DWORD iWidth,
        DWORD iHeight,
        WORD iColors,
        BOOL &bFullScreen
   );

#endif
```

Инициализацию DirectDraw будем проводить по аналогии с Direct3D, т. е. может быть создание как полноэкранного режима, так и оконного. В DirectDraw намного более явная разница между этими режимами.

Итак, в заголовочном файле у нас объявлена функция `DXDDInit`, которая принимает следующие параметры:

- `ppiDD` — указатель на создаваемый интерфейс `IDirectDraw7`. Обратите внимание, что здесь используется 7-я версия. Да, в заголовочном файле ddraw.h именно 7-я версия является последней;
- `primsurf` — указатель на создаваемую первичную поверхность типа `IDirectDrawSurface7`;
- `backsurf` — указатель на создаваемую поверхность заднего плана типа `IDirectDrawSurface7`;
- `hWnd` — окно, в котором будет происходить вывод графики;
- `iWidth` и `iHeight` — ширина и высота окна или разрешения соответственно;
- `iColors` — желаемое количество цветов;
- `bFullScreen` — если этот параметр равен значению `TRUE`, то необходимо создать полноэкранное приложение.

Теперь посмотрим на сам код инициализации, который представлен в листинге 1.5.

Листинг 1.5. Код инициализации DirectDraw

```
HRESULT DXDDInit(IDirectDraw7 **ppiDD,
    IDirectDrawSurface7 **primsurf,
    IDirectDrawSurface7 **backsurf,
```

```
    HWND hWnd,
    DWORD iWidth,
    DWORD iHeight,
    BOOL &bFullScreen)
{
 // Запрос на отображение в полноэкранном режиме
 int  iRes;
 if ((*bFullScreen)==FALSE)
  iRes=MessageBox(hWnd, "Use fullscreen mode?", "Screen",
         MB_YESNO | MB_ICONQUESTION);
 else
  iRes = IDYES;

 HRESULT hr;
 hr = DirectDrawCreateEx(NULL, (VOID**)ppiDD, IID_IDirectDraw7, NULL);
 if(FAILED(hr))
   return hr;

 if(iRes == IDYES)
  {
   ////////////////////////////////////////////////////////
   // Полноэкранный режим
   ////////////////////////////////////////////////////////
   *bFullScreen=FALSE;
   SetWindowLong(hWnd, GWL_STYLE, WS_POPUP);
   (*ppiDD)->SetCooperativeLevel(hWnd, DDSCL_EXCLUSIVE |
              DDSCL_FULLSCREEN | DDSCL_ALLOWMODEX );
   (*ppiDD)->SetDisplayMode(iWidth, iHeight, iColors, 0, 0);
  }
 else
  {
   ////////////////////////////////////////////////////////
   // Оконный режим
   ////////////////////////////////////////////////////////
   *bFullScreen=TRUE;
   RECT wndRect;
   RECT clientRect;
   GetWindowRect(hWnd, &wndRect);
   GetClientRect(hWnd, &clientRect);
   int iWinWidth  = iWidth + (wndRect.right-wndRect.left)  -
        (clientRect.right-clientRect.left);
   int iWinHeight = iHeight + (wndRect.bottom-wndRect.top) -
        (clientRect.bottom-clientRect.top);
```

```
   MoveWindow(hWnd, wndRect.left, wndRect.top,
        iWinWidth, iWinHeight, TRUE);

   (*ppiDD)->SetCooperativeLevel(hWnd, DDSCL_NORMAL);
  }

DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);

if(iRes == IDYES)
 {
  desc.dwFlags =  DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
  desc.dwBackBufferCount = 1;
  desc.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE | DDSCAPS_FLIP |
        DDSCAPS_COMPLEX;
 }
else
 {
  desc.dwFlags =  DDSD_CAPS;
  desc.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;
 }

hr=(*ppiDD)->CreateSurface(&desc, primsurf, 0);
if (hr!=DD_OK)
 return FALSE;

if(iRes == IDYES)
 {
  DDSCAPS2 surfcaps;
  ZeroMemory(&surfcaps, sizeof(surfcaps));
  surfcaps.dwCaps = DDSCAPS_BACKBUFFER;
  hr=(*primsurf)->GetAttachedSurface(&surfcaps, backsurf);
  if (hr!=DD_OK)
    return FALSE;

   ClearSurface(*primsurf, 0);
   ClearSurface(*backsurf, 0);
  }
 else
  {
   ZeroMemory(&desc, sizeof(desc));
   desc.dwSize = sizeof(desc);
```

```
    desc.dwFlags = DDSD_WIDTH | DDSD_HEIGHT | DDSD_CAPS;
    desc.dwWidth = iWidth;
    desc.dwHeight = iHeight;
    desc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN | DDSCAPS_VIDEOMEMORY;

    // Создание поверхности
    HRESULT r=(*ppiDD)->CreateSurface(&desc, backsurf, 0);
    if (r!=DD_OK)
        return 0; // Ошибка создания поверхности
   }
  return S_OK;
}
```

Теперь давайте посмотрим, что же здесь происходит. В самом начале мы запрашиваем у пользователя необходимость запуска в полноэкранном режиме. Потом создаем интерфейс `IDirectDraw7`. Если бы мы захотели организовать интерфейс самой первой версии DirectDraw, то для его создания нужно было бы использовать функцию `DirectDrawCreate`, а для получения более поздних версий применяется расширенный вариант этой функции `DirectDrawCreateEx`:

```
HRESULT WINAPI DirectDrawCreateEx(
   GUID FAR * lpGuid,
   LPVOID  *lplpDD,
   REFIID  iid,
   IUnknown FAR *pUnkOuter);
```

Здесь имеется четыре параметра:

- `lpGuid` — глобальный идентификатор GUID, отображающий драйвер, который необходимо создать. Если в этом параметре указать значение `NULL`, то будет использоваться активный драйвер дисплея. Вы также можете указать константу `DDCREATE_EMULATIONONLY` (использовать эмуляцию видеовозможностей) или `DDCREATE_HARDWAREONLY` (не эмулировать возможности, которые не поддерживаются видеокартой);
- `lplpDD` — указатель на переменную, в которую будет записан адрес созданного `IDirectDraw` интерфейса;
- `iid` — версия создаваемого интерфейса IDirectDraw. Здесь можно указать одну из следующих констант: `IID_IDirectDraw`, `IID_IDirectDraw2`, `IID_IDirectDraw4` или `IID_IDirectDraw7`. Как видите, номера версий идут не последовательно;
- `pUnkOuter` — указатель, который используется для будущей совместимости. Этот параметр должен быть равен значению `NULL`, иначе произойдет ошибка.

Далее нам необходимо установить уровень контроля над видеокартой, и если выбран полноэкранный режим, то переключить видеорежим, поэтому код разделяется в зависимости от выбранного режима.

Если выбран полный экран, то выполняется следующий код:

```
SetWindowLong(hWnd, GWL_STYLE, WS_POPUP);
(*ppiDD)->SetCooperativeLevel(hWnd, DDSCL_EXCLUSIVE |
                    DDSCL_FULLSCREEN | DDSCL_ALLOWMODEX );
(*ppiDD)->SetDisplayMode(iWidth, iHeight, iColors, 0, 0);
```

В первой строке мы устанавливаем с помощью WinAPI-функции `SetWindowLong` стиль окна `WS_POPUP`, который убирает все оборки и заголовок окна, оставляя только клиентскую часть. Зачем это нужно? После переключения в полный экран, в отличие от Direct3D, в DirectDraw остаются все оборки, меню и заголовок, а это абсолютно нам не нужно.

В следующей строке устанавливается уровень контроля видеокартой с помощью метода `SetCooperativeLevel`. У этой функции два параметра:

- `hWnd` — окно, которому предоставляется контроль;
- `dwLevel` — уровень контроля. Здесь можно указать одно из следующих значений:
  - `DDSCL_EXCLUSIVE` — эксклюзивный доступ, который доступен только для полноэкранных приложений;
  - `DDSCL_NORMAL` — нормальный режим;
  - `DDSCL_ALLOWMODEX` — позволяет использовать режимы дисплея ModeX;
  - `DDSCL_ALLOWREBOOT` — разрешить сочетание клавиш <Ctrl>+<Alt>+<Del> во время работы в полноэкранном режиме;
  - `DDSCL_FULLSCREEN` — эксклюзивный режим, при котором вы отвечаете за содержимое первичного буфера.

Последняя строчка кода устанавливает необходимый видеорежим с помощью метода `SetDisplayMode`. Здесь имеется пять параметров:

- ширина экрана;
- высота экрана;
- глубина цвета;
- частота смены кадров;
- не используется и должен быть равен нулю.

Далее идет код, который производит инициализацию окна для оконного режима. Здесь мы так же, как и при инициализации Direct3D, определяем и из-

меняем размеры окна. Далее устанавливаем нормальный режим контроля видеокартой:

```
(*ppiDD)->SetCooperativeLevel(hWnd, DDSCL_NORMAL);
```

В принципе, на этом инициализация DirectDraw будет завершена. Остальной код касается создания поверхностей, которые нужно формировать отдельно. Если при создании устройства Direct3D задний буфер организуется автоматически, то здесь это отдельная песня.

Но прежде чем создавать поверхность, нужно заполнить необходимыми параметрами структуру типа `DDSURFACEDESC2`. Структура достаточно сложная и рассматривать ее полностью мы не будем. Подробную информацию можно получить в файле помощи по DirectX, а рассмотрим только то, что использовали мы.

Итак, сначала объявляем переменную `desc` типа `DDSURFACEDESC2` и обнуляем ее содержимое, чтобы не заполненные нами поля содержали нулевое значение:

```
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
```

Теперь необходимо записать в поле `dwSize` корректный размер структуры:

```
desc.dwSize = sizeof(desc);
```

В параметре `dwFlags` нужно указать с помощью констант поля, которые мы хотим изменить. При полноэкранном режиме будем изменять опции (константа `DDSD_CAPS`) и количество задних буферов (`DDSD_BACKBUFFERCOUNT`). В случае с оконным режимом изменяются только опции.

Количество задних буферов нужно указывать в поле `dwBackBufferCount` и для полноэкранного режима в большинстве случаев достаточно одного заднего буфера. Для оконного режима может быть только первичный буфер, поэтому тут мы ничего не указываем.

В качестве опций в параметре `ddsCaps.dwCaps` используем следующее:

- `DDSCAPS_PRIMARYSURFACE` — необходимо создать первичную поверхность (применяется при обоих режимах);
- `DDSCAPS_COMPLEX` — поверхности должны быть связаны;
- `DDSCAPS_FLIP` — использовать переключение буферов.

Теперь мы готовы вызвать метод `CreateSurface` для создания поверхности. У этого метода три параметра:

- указатель на структуру типа `DDSURFACEDESC2`, которую мы недавно заполнили;

- указатель на переменную первичной поверхности;
- не используется и должен быть равен нулю.

Если мы выбрали полноэкранный режим, то помимо первичной поверхности у нас будет еще и буфер заднего плана. Для получения указателя на нее используется следующий код:

```
DDSCAPS2 surfcaps;
ZeroMemory(&surfcaps, sizeof(surfcaps));

surfcaps.dwCaps = DDSCAPS_BACKBUFFER;

hr=(*primsurf)->GetAttachedSurface(&surfcaps, backsurf);
if (hr!=DD_OK)
   return FALSE;
```

Сначала заполняется структура типа `DDSCAPS2`. Она достаточно сложная, поэтому мы рассмотрим пока только то, что используется в этом коде. Для начала переменная заполняется нулями. Потом в свойство `dwCaps` заносится константа `DDSCAPS_BACKBUFFER`, которая означает, что нам необходимо получить указатель на задний буфер. Вот и все, что нам нужно.

Теперь, чтобы получить указатель на вторую поверхность, необходимо использовать метод `GetAttachedSurface` интерфейса первичной поверхности. В качестве параметров метод получает указатель на заполненную структуру типа `DDSCAPS2` и указатель на переменную `IDirectDrawSurface7`, куда будет записан указатель на буфер заднего плана.

Если у нас выбран оконный режим, то вторая поверхность не будет сформирована и ее придется создать вручную. Конечно же, можно обойтись и без нее, но в 90% случаев удобно иметь поверхность заднего плана, пусть она и не будет связана с первичной. Следующий код создает несвязанную поверхность:

```
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);
desc.dwFlags = DDSD_WIDTH | DDSD_HEIGHT | DDSD_CAPS;
desc.dwWidth = iWidth;
desc.dwHeight = iHeight;
desc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |
    DDSCAPS_VIDEOMEMORY;

// Создание поверхности
HRESULT r=(*ppiDD)->CreateSurface(&desc, backsurf, 0);
if (r!=DD_OK)
    return 0; // Ошибка создания поверхности
```

Сначала мы обнуляем уже существующую переменную `desc`, которая имеет тип `DDSURFACEDESC2` и призвана описывать поверхность. В параметр `dwFlags` заносим константы `DDSD_WIDTH`, `DDSD_HEIGHT` и `DDSD_CAPS`, которые означают, что в структуре будут заполнены поля ширины, высоты и свойств поверхности соответственно. Далее, идет заполнение соответствующих полей.

В поле свойств (`ddsCaps.dwCaps`) мы указываем два параметра: `DDSCAPS_OFFSCREENPLAIN` и `DDSCAPS_VIDEOMEMORY` (поверхность должна быть в видеопамяти, чтобы копирование в первичную поверхность происходило максимально быстро). Но видеопамять не резиновая, и поэтому по возможности необходимо использовать системную память и параметр `DDSCAPS_SYSTEMMEMORY`.

Параметр `DDSCAPS_OFFSCREENPLAIN` означает, что поверхность не относится к какому-либо специализированному типу (не является первичной, вторичной, z-буфером, альфа и т. д.). Это просто поверхность, с которой вы можете работать как с буфером памяти для любых собственных нужд.

В коде листинга 1.5 вы можете заметить еще функцию `ClearSurface`, которая находится в файле ddfunc.cpp в каталоге \Common на компакт-диске, но пока мы не будем ее рассматривать, чтобы не усложнять себе жизнь. В следующей главе мы восполним этот пробел.

Теперь, для инициализации DirectDraw достаточно объявить три переменные. Одна из них должна иметь тип `IDirectDraw7` для хранения указателя на интерфейс DirectDraw. И две переменные типа `IDirectDrawSurface7` для хранения интерфейсов первичной и вторичной поверхностей. Например:

```
IDirectDraw7 *ppiDD;
IDirectDrawSurface7 *primsurf;
IDirectDrawSurface7 *backsurf=NULL;
BOOL bWindowed=FALSE;
```

Теперь вызов функции инициализации DirectDraw может выглядеть следующим образом:

```
DXDDInit(&ppiDD, &primsurf, &backsurf,
    hWnd, 800, 600, 16, &bWindowed);
```

Где написать вызов этой функции? Идеальное место — после создания окна, но до ее отображения (до вызова `ShowWindow` и `UpdateWindow`) в функции `InitInstance`. Например:

```
// Код Создание окна

DXDDInit(&ppiDD, &primsurf, &backsurf,
    hWnd, 800, 600, 16, &bWindowed);

ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
```

Самое интересное, что при вызове полноэкранного приложения, вызов WinAPI-функций `ShowWindow` и `UpdateWindow` не является необходимым, но желательным.

Примечание

Исходный код примера можно увидеть в каталоге \Chapter1\InitDirectDraw, а файл ddfunc.cpp — в каталоге \Common.

1.7. Освобождение ресурсов

Освобождать ресурсы крайне необходимо. Это не просто хороший тон в программировании, но и показывает, что вы не программист-однодневка, который учился двоечником на задней парте, а профессионал. Вы же после еды моете посуду и убираете мусор, так почему же не убрать за своей программой, очистив "загаженную" память?

Ресурсы DirectX — это указатели на интерфейсы, которые освобождаются методом `Release`. После освобождения указатель необходимо установить в нулевое значение.

Освобождение необходимо производить в обратном созданию порядке. Например, если вы только проинициализировали DirectDraw, то у вас есть интерфейсы двух поверхностей и интерфейс DirectDraw. Порядок освобождения следующий:

1. Уничтожить вторичную поверхность.
2. Уничтожить первичную поверхность.
3. Уничтожить DirectDraw.

Последовательность должна быть именно такой, потому что эти интерфейсы создавались в порядке обратном этому. Если же сначала уничтожить первичную поверхность, то указатель на вторую станет не действительным, ведь эти две поверхности связаны между собой.

Идеальным местом для очистки интерфейсов является событие `WM_DESTROY`. Следующий пример демонстрирует очистку ресурсов DirectDraw:

```
case WM_DESTROY:
 if (backsurf) {backsurf->Release();backsurf=0;}
 if (primsurf) {primsurf->Release();primsurf=0;}
 if (ppiDD) {ppiDD->Release();ppiDD=0;}
 PostQuitMessage(0);
 break;
```

Старайтесь не забывать уничтожать объекты. Я по правде и сам иногда, добавляя создание нового интерфейса, забываю описать его уничтожение.

ГЛАВА 2

Основные функции DirectX

В этой главе мы познакомимся с основными функциями, которые нам понадобятся в будущем. При этом функции инициализации, которые мы уже рассмотрели в предыдущей главе, мы опустим.

Помимо этого, мы опустим тему использования WinAPI-функций GDI в DirectDraw. Если вы уже знакомы с этой возможностью, то советую забыть, а если не знали, то лучше и не знать, чтобы не было соблазна воспользоваться ими. Дело в том, что если вы воспользуетесь WinAPI-функцией рисования линии на поверхности DirectDraw, то никакая оптимизация не сможет сделать вашу программу быстрее, чем рисование без WinAPI-функций.

Поскольку у DirectDraw нет возможности рисовать простые фигуры, поэтому приходится вспоминать математику. Но если обратиться к WinAPI, то смысл от использования DirectDraw падает практически до нуля. В последующих главах мы еще поговорим о методах рисования от простых до сложных фигур в DirectX, а до этого нам еще нужно узнать, как работать с поверхностями в DirectDraw и как формировать 3D-фигуры в Direct3D. Вот этим мы сейчас и займемся.

Для начала заметим, что функции и методы DirectDraw в случае удачной работы возвращают константу `DD_OK`. Чаще всего мы будем просто сравнивать результат с этой константой, не углубляясь в природу ошибки, потому что без проверок нельзя, а углубляться в них — потеря производительности. Более подробную информацию о возможных ошибках вы можете почерпнуть в файле помощи.

2.1. Загрузка картинки в DirectDraw

Мы уже научились инициализировать DirectDraw, но пока приложение еще ничего не отображает, смысла от него мало. DirectDraw настолько прост, что

предоставляет нам только прямой доступ к памяти и возможность копирования поверхностей. В качестве бонуса есть возможность применять GDI-функции, но мы договорились их не использовать из-за их медлительности. Однако в этой главе нам придется нарушить это правило, но только для того, чтобы первый пример не был слишком большим.

Итак, давайте научимся загружать картинку и отображать ее. Для этого создайте новый проект, добавьте к нему наш модуль ddfunc и воспользуйтесь функцией инициализации DirectDraw. При этом инициализация должна сразу создавать полноэкранный режим без возможности оконного режима, работа с которым отличается, и его поддержку мы добавим чуть позже.

Загрузка картинок может происходить заранее. Например, в играх картинки загружаются на этапе загрузки уровня, а в демонстрационных программах на этапе старта программы. В этот момент мы еще можем пожертвовать скоростью. К тому же может потребоваться загрузка множества картинок, поэтому напишем универсальную функцию в нашем модуле ddfunc.cpp.

В заголовочном файле ddfunc.h добавляем описание функции:

```
IDirectDrawSurface7 *LoadBMPToSurface(LPCTSTR filename,
    IDirectDraw7 *ppiDD);
```

Функция получает два параметра: имя файла, ресурс которого нужно загрузить, и указатель на интерфейс `IDirectDraw7`, который понадобится для создания поверхности, куда будет загружена картинка. В качестве результата функция возвращает указатель на созданную поверхность.

Сам код функции показан в листинге 2.1.

Листинг 2.1. Загрузка картинки в поверхность

```
IDirectDrawSurface7 *LoadBMPToSurface(LPCTSTR filename,
              IDirectDraw7 *ppiDD)
{
  IDirectDrawSurface7 *surf;
  HANDLE hBmp;
  BITMAP bm;

  // Загрузка картинки из файла
  hBmp = LoadImage(0, filename, IMAGE_BITMAP, 0, 0,
         LR_LOADFROMFILE | LR_CREATEDIBSECTION);
  if (hBmp==0)
    return 0;
  GetObject(hBmp, sizeof(bm), &bm);
```

```
  // Подготавливаем структуру для создания поверхности
  DDSURFACEDESC2 desc;
  ZeroMemory(&desc, sizeof(desc));
  desc.dwSize = sizeof(desc);
  desc.dwFlags = DDSD_WIDTH | DDSD_HEIGHT | DDSD_CAPS;
  desc.dwWidth = bm.bmWidth;
  desc.dwHeight = bm.bmHeight;
  desc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN | DDSCAPS_SYSTEMMEMORY;

  // Создание поверхности
  HRESULT r=ppiDD->CreateSurface( &desc, &surf, 0 );
  if (r!=DD_OK)
    return 0; // Ошибка создания поверхности

  // Копирование содержимого картинки на поверхность
  HDC hdcImage;
  hdcImage = CreateCompatibleDC(0);
  if (hdcImage == 0)
    return 0;
  SelectObject(hdcImage, hBmp);

  HDC hdc;
  surf->GetDC(&hdc);
  StretchBlt(hdc, 0, 0, bm.bmWidth, bm.bmHeight, hdcImage,
           0, 0, bm.bmWidth, bm.bmHeight, SRCCOPY);
  surf->ReleaseDC(hdc);

  DeleteDC(hdcImage);
  DeleteObject(hBmp);
  return surf;
}
```

Загрузка картинки происходит с помощью WinAPI-функции `LoadImage`. Затем получаем информацию об объекте с помощью WinAPI-функции `GetObject`, чтобы узнать ширину и высоту картинки и использовать эту информацию для создания поверхности соответствующих размеров.

Далее, заполняем структуру `DDSURFACEDESC2` размерами создаваемой поверхности (они соответствуют размерам картинки) и параметры поверхности, которую можно хранить и в системной памяти (параметр `DDSCAPS_SYSTEMMEMORY`).

Пока что ничего нового и все это мы уже видели. А вот далее начинается самое интересное. Для упрощения задачи будем копировать картинку через WinAPI-функции работы с графикой. Да, это медленно, но пока не будем

слишком сильно усложнять задачу. Функции WinAPI используют контексты устройства HDC (Handle to Device Context), поэтому необходимо:

1. Создать контекст устройства и поместить туда картинку.
2. Получить контекст поверхности, в которую будем копировать картинку.
3. Скопировать изображение с помощью WinAPI-функции `BitBlt` или `StretchBlt`.

Следующий код создает контекст HDC и помещает в него картинку:

```
HDC hdcImage;
hdcImage = CreateCompatibleDC(0);
if (hdcImage == 0)
    return 0;
SelectObject(hdcImage, hBmp);
```

Чтобы получить контекст DirectDraw-поверхности, используется WinAPI-функция `GetDC`. В качестве единственного параметра методу необходимо передать переменную типа HDC, куда будет записан контекст:

```
HDC hdc;
surf->GetDC(&hdc);
```

Теперь достаточно скопировать данные из одного контекста в другой:

```
StretchBlt(hdc, 0, 0, bm.bmWidth, bm.bmHeight, hdcImage,
           0, 0, bm.bmWidth, bm.bmHeight, SRCCOPY);
```

После использования полученный контекст поверхности необходимо уничтожить. Для этого применяется метод `ReleaseDC`:

```
surf->ReleaseDC(hdc);
```

Это самый простой вариант загрузки картинки. Как мы уже говорили, загрузка происходит в такие моменты, когда скоростью можно и пренебречь и даже можно довериться WinAPI. Но для игр и продвинутых демонстрационных роликов иногда требуется загружать десятки мегабайт графики, и если мы напишем быструю функцию, то ее использование позволит повысить скорость переходов между уровнями игры и загрузки программы в несколько раз.

Когда мы рассмотрим все необходимые функции DirectDraw, попробуйте сами оптимизировать загрузку картинок и реализовать их без использования WinAPI.

Теперь посмотрим, как можно воспользоваться функцией для загрузки картинки:

```
imagesurf = LoadBMPToSurface("desktop.bmp", ppiDD);
```

В этом примере мы вызываем функцию `LoadBMPToSurface`, указав в качестве первого параметра имя файла, где расположена картинка, а второй параметр — это указатель на интерфейс `IDirectDraw`.

В качестве результата функция возвращает нам интерфейс поверхности, в которую загружена картинка. Удобство функции в том, что она сама создает поверхность, поэтому код загрузки выглядит в виде одной строчки.

2.2. Отображение картинок в DirectDraw

Давайте разберемся, как можно отобразить картинку. Это позволит нам убедиться, что DirectDraw инициализируется верно, а картинка загружается на поверхность правильно. Идеальным местом для рисования будет событие `WM_PAINT`, как и для любого другого Windows-приложения, ведь если это событие сгенерировано, значит, мы обязаны перерисовать окно.

Следующий пример показывает, как в полноэкранном режиме отобразить загруженную в *разд. 2.1* картинку:

```
case WM_PAINT:
  hdc = BeginPaint(hWnd, &ps);

  if ((imagesurf!=NULL) && (!bWindowed))
   {
     backsurf->BltFast(0, 0, imagesurf, NULL, DDBLTFAST_WAIT);
     primsurf->Flip(NULL, DDFLIP_WAIT);
   }

  EndPaint(hWnd, &ps);
  break;
```

Код отображения мы написали между функциями `BeginPaint` и `EndPaint`, которые сгенерировали мастер во время создания приложения. Сначала мы проверяем, существует ли поверхность с картинкой и является ли текущий режим полноэкранным. Если да, то копируем картинку в задний буфер с помощью функции `BltFast`:

```
backsurf->BltFast(0, 0, imagesurf, NULL, DDBLTFAST_WAIT);
```

У интерфейса поверхности существует два метода копирования данных одной поверхности в другую: `Blt` и `BltFast`. Первый обладает большими возможностями, поэтому работает медленнее, а второй метод проще и быстрее, потому что не поддерживает возможности масштабирования копируемых данных. Давайте посмотрим оба этих метода.

2.2.1. Метод *Blt*

Первый метод `Blt` выглядит следующим образом:

```
HRESULT Blt(
  LPRECT lpDestRect,
  LPDIRECTDRAWSURFACE lpDDSrcSurface,
  LPRECT lpSrcRect,
  DWORD dwFlags,
  LPDDBLTFX lpDDBltFX
  );
```

Рассмотрим параметры этого метода:

- `lpDestRect` — область Rect в поверхности-приемнике, в которую необходимо скопировать данные;
- `lpDDSrcSurface` — поверхность, из которой необходимо копировать;
- `lpSrcRect` — область Rect в поверхности-источнике, которую необходимо скопировать;
- `dwFlags` — флаги, которые определяют дополнительные параметры копирования. Флагов очень много, поэтому рассматривать их будем постепенно, по мере надобности. Чтобы увидеть все параметры, обратитесь к файлу помощи, возможно, это поможет, если нет проблем с английским языком ☺;
- `lpDDBltFX` — указатель на структуру `DDBltFX`, в которой задаются дополнительные параметры и эффекты копирования.

Помимо того, что функция позволяет масштабировать полученные данные, она также может и обрезать данные, если часть изображения выходит за пределы окна. Если вас интересует производительность, то никогда не пользуйтесь этой функцией, если вам не нужно масштабировать данные. Отсечение невидимых данных можно сделать и самостоятельно.

2.2.2. Метод *BltFast*

Метод `BltFast` проще, это видно сразу же по его общему виду:

```
HRESULT BltFast(
  DWORD dwX,
  DWORD dwY,
  LPDIRECTDRAWSURFACE lpDDSrcSurface,
  LPRECT lpSrcRect,
  DWORD dwTrans
  );
```

Здесь имеется пять параметров:

- `dwX` и `dwY` — определяют левую и верхнюю позицию, в которую нужно скопировать содержимое поверхности;
- `lpDDSrcSurface` — поверхность, из которой нужно скопировать данные;
- `lpSrcRect` — область Rect в поверхности-источнике, которую необходимо скопировать;
- `dwTrans` — определяет тип копирования. Здесь можно указать одно или сочетание из следующих значений:
  - `DDBLTFAST_DESTCOLORKEY` — прозрачное копирование, при котором необходимо учитывать ключ прозрачности поверхности назначения;
  - `DDBLTFAST_NOCOLORKEY` — нормальное копирование без специфичных возможностей;
  - `DDBLTFAST_SRCCOLORKEY` — прозрачное копирование, при котором необходимо учитывать ключ прозрачности поверхности источника;
  - `DDBLTFAST_WAIT` — если в текущий момент происходит прорисовка, то без указания этого флага вы увидите ошибку `DDERR_WASSTILLDRAWING`. Укажите этот параметр, чтобы не видеть этой ошибки ☺. В этом случае DirectDraw дождется, когда прорисовка будет доступной и корректно прорисует поверхность. Чаще всего этот параметр необходим, когда вы выводите данные на первичную поверхность.

2.2.3. Переключение поверхностей

После прорисовки данных во вторичном буфере мы переключаем поверхности. В этот момент вторичная поверхность становится первичной и все, что мы нарисовали во вторичной поверхности, отображается на экране. Для переключения поверхностей используется метод `Flip`:

```
primsurf->Flip(NULL, DDFLIP_WAIT);
```

В общем виде метод выглядит следующим образом:

```
HRESULT Flip(
  LPDIRECTDRAWSURFACE lpDDSurfaceTargetOverride,
  DWORD dwFlags
  );
```

Первый параметр — указатель на интерфейс поверхности, на которую нужно переключиться. Этот параметр эффективен, когда у вас более двух связанных поверхностей, т. е. более одного заднего буфера. В нашем случае только один

задний буфер, поэтому переключение может быть только на него, и этот параметр нулевой.

Второй параметр — это флаги переключения. Здесь можно указать только одно значение: DDFLIP_WAIT. Если в текущий момент происходит прорисовка, то без указания этого флага вы увидите ошибку DDERR_WASSTILLDRAWING. Если указать во втором параметре DDFLIP_WAIT, то в этом случае DirectDraw дождется, когда переключение станет доступным, и корректно выполнит операцию.

Примечание

Исходный код примера находится в каталоге \Chapter2\FastBlt на прилагаемом компакт-диске.

2.2.4. Примеры копирования поверхностей

Давайте посмотрим, как можно вывести изображение на весь экран. Если картинка меньше, чем разрешение экрана, то ее нужно будет масштабировать, а это умеет делать метод Blt. В листинге 2.2 представлен пример, который показывает, как можно воспользоваться этим методом для масштабирования нашей поверхности с картинкой на весь экран.

Листинг 2.2. Пример копирования содержимого поверхности с масштабированием

```
case WM_PAINT:
 hdc = BeginPaint(hWnd, &ps);

 if ((imagesurf!=NULL) && (!bWindowed))
  {
   RECT dst;
   dst.left=dst.top=0;
   dst.right=iWidth;
   dst.bottom=iHeight;

   DDBLTFX bltfx;
   ZeroMemory(&bltfx, sizeof(bltfx));
   bltfx.dwSize=sizeof(bltfx);

   backsurf->Blt(&dst, imagesurf, 0, DDBLT_WAIT, &bltfx);
   primsurf->Flip(NULL, DDFLIP_WAIT);
  }
```

В этом примере мы объявили переменную `dst` типа `RECT`, в которую впоследствии заполняются размеры окна. Теперь переменную `bltfx` типа `DDBLTFX`. Эта переменная должна передаваться в последнем параметре. Конечно же, мы не используем дополнительных параметров и, по идее, можно было бы обойтись и без этого параметра, а указать ноль, но мы подготовили почву для следующего примера.

После всех подготовительных действий мы вызываем метод `Blt` со следующими параметрами:

```
backsurf->Blt(&dst, imagesurf, 0, DDBLT_WAIT, &bltfx);
```

Первый параметр задает параметры области приема. В нашем случае область приема равна всему окну, поэтому изображение будет растянуто. Третий параметр — это область, которую нужно взять из источника. У нас указан ноль, а значит будет взята вся область.

Примечание

Исходный код примера находится в каталоге \Chapter2\BitBlt на прилагаемом компакт-диске.

2.2.5. Использование метода *Blt* для очистки поверхности

Теперь вспомним о существовании в нашем модуле ddfunc функции `ClearSurface`, которая должна очищать поверхность. Код функции представлен в листинге 2.3.

Листинг 2.3. Очистка поверхности

```
BOOL ClearSurface(LPDIRECTDRAWSURFACE7 surf, int iColor)
{
       if (surf==0)
              return FALSE;

       DDBLTFX bltfx;
       ZeroMemory( &bltfx, sizeof(bltfx) );
       bltfx.dwSize = sizeof(bltfx);
       bltfx.dwFillColor = iColor;

       HRESULT r;
       r=surf->Blt(0, 0, 0, DDBLT_COLORFILL | DDBLT_WAIT, &bltfx );
       if (r==DD_OK)
              return TRUE;
```

```
        else
                return FALSE;
}
```

Давайте рассмотрим, что мы указали в каждом из параметров методу `Blt` (общий вид этого метода мы рассматривали в *разд. 2.2.1*):

- `lpDestRect` — область закраски в поверхности-приемнике. В нашем случае здесь ноль, а значит, будет закрашена вся поверхность;
- `lpDDSrcSurface` — поверхность из которой необходимо копировать. Поскольку у нас не будет копирования поверхности, а только закраска всей поверхности указанным цветом, то этот параметр равен нулю;
- `lpSrcRect` — область копирования в поверхности-источнике. У нас этой поверхности нет, а значит, и параметр равен нулю;
- `dwFlags` — флаги. Помимо уже знакомого нам параметра ожидания возможности прорисовки (`DDBLT_WAIT`), здесь указан флаг `DDBLT_COLORFILL`, который означает, что необходимо заполнить поверхность определенным цветом. Сам цвет задается в структуре, которую мы указываем в последнем параметре метода `Blt`;
- `lpDDBltFX` — структура `DDBltFX`, у которой в параметре `dwFillColor` задан цвет, которым будет заполнена поверхность. Помимо этого, в структуре должен быть корректно заполнено поле `dwSize`, где указывается размер структуры.

Таким образом, метод `Btl` можно использовать не только для копирования содержимого поверхностей.

В листинге 2.4 представлен пример, в котором весь экран в цикле перекрашивается разными цветами.

Листинг 2.4. Перекрашивание экрана

```
case WM_PAINT:
  hdc = BeginPaint(hWnd, &ps);

  if ((imagesurf!=NULL) && (!bWindowed))
   {
    DDBLTFX bltfx;
    ZeroMemory( &bltfx, sizeof(bltfx) );
    bltfx.dwSize = sizeof(bltfx);
    HRESULT r;

    for (int i=0; i<1000; i++)
     {
      bltfx.dwFillColor = i;
```

```
      r=backsurf->Blt(0, 0, 0, DDBLT_COLORFILL | DDBLT_WAIT, &bltfx );
      primsurf->Flip(NULL, DDFLIP_WAIT);
     }
   }

 EndPaint(hWnd, &ps);
 break;
```

В этом примере по событию `WM_PAINT` запускается цикл от 0 до 1000, в котором очищается экран. В качестве цвета берется значение счетчика. Таким образом, цвет экрана будет переходить от черного до синего.

Единственный недостаток этого примера — выход из программы затруднен, потому что если система сгенерирует 10 событий прорисовки подряд, то программе придется перерисовать экран 10 000 раз, поэтому программа среагирует на комбинацию клавиш <Ctrl>+<F4> не сразу, а после выполнения всех прорисовок.

Примечание

Исходный код примера находится в каталоге \Chapter2\ClearBlt на прилагаемом компакт-диске.

2.2.6. Прозрачное копирование

Теперь поговорим о том, как можно копировать данные с учетом прозрачности. Допустим, что у нас есть рисунок вертолета (рис. 2.1) и мы хотим отобразить сам вертолет поверх какого-то изображения без черного фона.

Рис. 2.1. Изображение вертолета с черным фоном, который должен быть прозрачен при копировании

Черный фон выбран не случайно, ведь этот цвет определяется нулевым значением. Как мы уже выяснили в *разд. 1.5* (когда изучали инициализацию DirectDraw), формат пиксела в DirectDraw может быть разным, а значит, на поверхности один пиксел может описываться 1, 2, 3 или даже 4-мя байтами. Если же в качестве прозрачного будет использоваться красный цвет, то в зависимости от формата пиксела число байт может отличаться.

Если вы разрабатываете приложение, которое будет работать только в полноэкранном режиме с определенным форматом пиксела, то это не такая уж страшная проблема и можно рассчитать число заранее. Главное использовать один и тот же цвет прозрачности во всех картинках. А если это оконное приложение, то тут мы не можем повлиять на формат пиксела, разве что принудить пользователя установить нужное значение в свойствах рабочего стола Windows, но это не есть хорошо.

Проблема очень элегантно решается использованием в качестве прозрачного черного цвета, а там, где нужен черный непрозрачный, использовать цвет 000001. Большинство людей просто не заметит, что перед ним не истинно черный цвет, а немного (на единичку сдвинутый) серый.

Для задания прозрачного цвета у поверхности есть метод `SetColorKey`. У этого метода имеется два параметра: флаги (для задания прозрачности будем использовать здесь `DDCKEY_SRCBLT`) и структура типа `DDCOLORKEY`, определяющая цвет, который будет применяться в качестве прозрачного.

Прозрачный цвет необходимо указать только один раз и лучше это сделать на этапе загрузки картинки в поверхность, а не на этапе использования. Давайте добавим в пример, который мы рассматривали в предыдущем разделе, загрузку еще одной картинки с изображением вертолета и отображения ее с учетом прозрачности. В листинге 2.5 показан код загрузки поверхностей и задания прозрачного цвета.

Листинг 2.5. Загрузка изображения и задание прозрачности

```
// Загрузка изображения фона
imagesurf = LoadBMPToSurface("desktop.bmp", ppiDD);
// Загрузка изображения вертолета
imagesurf2 = LoadBMPToSurface("vertol.bmp", ppiDD);

// Подготовка структуры, описывающей прозрачность
DDCOLORKEY cData;
cData.dwColorSpaceLowValue = 0;
cData.dwColorSpaceHighValue = 0;

// Установка прозрачного цвета
imagesurf2->SetColorKey(DDCKEY_SRCBLT, &cData);
```

С загрузкой изображения все понятно. Тут ничего нового нет. После этого объявляется переменная `cData` типа структуры `DDCOLORKEY`. Эта структура выглядит следующим образом:

```
typedef struct _DDCOLORKEY{
  DWORD dwColorSpaceLowValue;
  DWORD dwColorSpaceHighValue;
} DDCOLORKEY,FAR *LPDDCOLORKEY;
```

Поле `dwColorSpaceLowValue` определяет минимальное значение цвета, которое будет использоваться в качестве прозрачного, а `dwColorSpaceHighValue` — максимальное значение. В нашем случае диапазон не нужен и прозрачным будет только черный цвет со значением 0, поэтому в оба поля заносим именно 0.

Далее, вызываем метод `SetColorKey` у поверхности, куда загружено изображение вертолета. В качестве первого параметра указываем флаг `DDCKEY_SRCBLT` (должен использоваться цвет прозрачности источника) и заполненную структуру типа `DDCOLORKEY`.

Теперь посмотрим, как можно отобразить поверхность с учетом прозрачности. Если просто вызвать метод `Blt` или `BltFast`, как мы это делали ранее, то ничего хорошего не произойдет. Черный фон вертолета будет также скопирован. Чтобы методы использовали выбранный нами цвет прозрачности, необходимо указать флаг `DDBLT_KEYSRC` для метода `Blt` и флаг `DDBLTFAST_SRCCOLORKEY` для метода `BltFast`. Следующий пример показывает, как копировать прозрачную картинку с помощью метода `Blt`:

```
backsurf->Blt(&dst, imagesurf2, 0,
   DDBLT_KEYSRC | DDBLT_WAIT, 0);
```

Эту строку кода необходимо добавить после того, как мы отобразили изображение фона. Но поскольку мы не используем тут масштабирования и не учитываем выходы за пределы окна, то давайте лучше воспользуемся более быстрым методом — `BltFast`, который также умеет копировать изображения с учетом прозрачности.

```
backsurf->BltFast(100, 100, imagesurf2, 0,
    DDBLTFAST_SRCCOLORKEY | DDBLTFAST_WAIT);
```

Результат работы примера показан на рис. 2.2. Надеюсь, что возможности полиграфии позволят вам увидеть, что фон вертолета не был скопирован, а поверх рисунка с моим котом по кличке Чикист виден только вертолет без черного окружения.

Примечание

Исходный код примера находится в каталоге \Chapter2\TransparentBlt на прилагаемом компакт-диске.

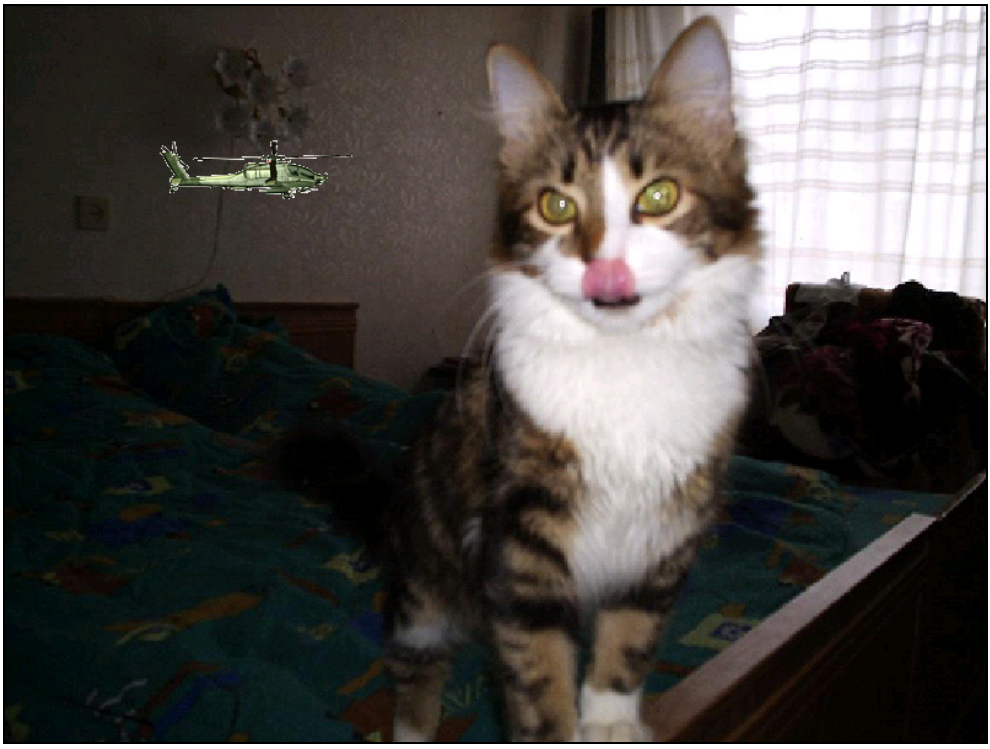

Рис. 2.2. Результат копирования изображения с учетом прозрачности

2.3. Оконные приложения

Теперь поговорим о том, как выводить изображение при использовании оконного режима. Пока что мы жестко отрезали этот режим и не использовали его. Давайте посмотрим, как происходит здесь вывод графики, и сделаем возможным работу нашего приложения, созданного в предыдущем разделе, как в полном экране, так и в окне.

Для начала обратимся к функции `InitInstance`, где создается окно. Изменим начальное значение переменной `bWindowed` на `FALSE`, чтобы перед пользователем появлялся запрос на запуск в полном экране, и он мог выбрать оконный режим. Остальная часть инициализации не изменится.

Теперь переходим к коду прорисовки окна, который выполняется в обработчике события `WM_PAINT`. Модифицированная версия с учетом работы в оконном режиме показана в листинге 2.6.

Листинг 2.6. Прорисовка окна с учетом оконного режима

```
case WM_PAINT:
 hdc = BeginPaint(hWnd, &ps);

 if (imagesurf!=NULL)
  {
   // Область копирования
   RECT dst;
   dst.left=dst.top=0;
   dst.right=iWidth;
   dst.bottom=iHeight;

   // Копируем картинку 1
   backsurf->Blt(&dst, imagesurf, 0, DDBLT_WAIT, 0);

   // Копируем картинку 2
   backsurf->BltFast(100, 100, imagesurf2, 0,
        DDBLTFAST_SRCCOLORKEY | DDBLTFAST_WAIT);
  }

 if (bWindowed)
  {
   RECT clientRect;
   POINT p;
   p.x=p.y=0;
   ClientToScreen(hWnd, &p);
   GetClientRect(hWnd, &clientRect);
   OffsetRect(&clientRect, p.x, p.y);

   primsurf->Blt(&clientRect, backsurf, 0, DDBLT_WAIT, 0);
  }
 else
  primsurf->Flip(NULL, DDFLIP_WAIT);

 EndPaint(hWnd, &ps);
 break;
```

Вспоминаем функцию инициализации DirectDraw, которую мы рассматривали в *разд. 1.5* и использовали в этом примере. Во время инициализации в любом случае создается первичная и вторичная поверхности, только в случае с оконным режимом, причем вторичная поверхность не связана и не может использоваться переключение. Поэтому мы можем рисовать на заднем буфере

что угодно, разница будет только в переносе данных заднего буфера на передний.

В новом варианте рисования вывод данных на задний буфер будет происходить вне зависимости от режима. Задний буфер полностью подготавливается, но перед переключением с помощью метода `Flip` мы проверяем, какой сейчас режим. Если программа работает в полноэкранном режиме, то переключение буферов происходит посредством метода `Flip`, иначе выполняется следующий код:

```
RECT clientRect;
POINT p;
p.x=p.y=0;
ClientToScreen(hWnd, &p);
GetClientRect(hWnd, &clientRect);
OffsetRect(&clientRect, p.x, p.y);

primsurf->Blt(&clientRect, backsurf, 0, DDBLT_WAIT, 0);
```

Сначала мы вызываем WinAPI-функцию `ClientToScreen`, чтобы определить левую верхнюю позицию окна. Результат будет записан в переменную `p` (она имеет тип `POINT`, поля которой изначально обнулены).

Далее, определяем размеры клиентской области с помощью WinAPI-функции `GetClientRect` и увеличиваем полученные размеры с помощью `OffsetRect`. После этого в переменной `clientRect` будут координаты окна на экране.

Зачем все это нужно? Дело в том, что даже в оконном режиме мы получаем доступ ко всему экрану и первичный буфер содержит данные всего экрана, а перерисовать мы должны только свое окно. Для этого-то мы и определяем координаты клиентской области, которую нужно обновить.

Теперь вызываем знакомый нам метод `Blt` и перерисовываем в первичном буфере область своего окна содержимым вторичного буфера. Вторичный же буфер у нас имеет размеры клиентской области, что упрощает код.

В принципе приложение готово к использованию, но оно обладает одним недостатком — изображение будет прорисовываться всегда. Попробуйте запустить программу и потом переключиться на другое окно или свернуть свое приложение. Окно вашей программы свернется, но рисунок останется на экране. Проблема опять связана с тем, что мы рисуем на экране, на который указывает первичный буфер, а не в окне. Перед прорисовкой необходимо убедиться в том, что наше окно активно и только тогда рисовать. Если же окно не активно, то и рисования не должно быть. Я бы добавил в начало кода прорисовки следующую проверку:

```
if (GetActiveWindow()!=hWnd)
  return 0;
```

Теперь можно запустить программу и убедиться, что все работает корректно.

Примечание

Исходный код примера находится в каталоге \Chapter2\Windowed на прилагаемом компакт-диске.

2.4. Контроль области отображения

Областью отображения нашего приложения является окно. Если в полноэкранном режиме мы выйдем за пределы экрана хотя бы на один пиксел, то изображение просто не будет показано вообще. В оконном приложении за пределы экрана убегают реже, намного чаще выскакивают за пределы окна.

Давайте возьмем пример из предыдущего раздела и исправим его так, чтобы вывод изображения вышел за пределы окна. Например, найдите строку, где мы используем WinAPI-функцию `OffsetRect`, с помощью которой определяем положение клиентской области окна и увеличим смещение на 40 пикселов:

```
OffsetRect(&clientRect, p.x+40, p.y);
```

В результате изображение выйдет за пределы окна, что не совсем корректно (рис. 2.3). Хотя нет, это совсем не корректно, вернее полный ужас. Ни один пользователь не будет доволен такому результату, поэтому необходимо както решать эту проблему и обрезать часть изображения, которая выходит за пределы окна. Это можно сделать вручную (о чем мы поговорим в следующей главе), а можно довериться возможностям DirectDraw. Сейчас рассмотрим второй вариант.

В DirectX есть очень интересный интерфейс — `IDirectDrawClipper` (интерфейс отсечения), который может помочь нам решить проблему выхода за пределы окна. К тому же, он очень прост в использовании, хотя и немного отбирает производительности у приложения, во время вывода графики на экран.

Для создания интерфейса необходимо вызвать метод `CreateClipper` интерфейса `IDirectDraw`. У этого метода три параметра:

- `dwFlags` — флаги, которые пока не используются, и этот параметр должен быть равен нулю;
- `lplpDDClipper` — указатель на указатель на переменную типа `IDirectDrawClipper`. О как красиво! В эту переменную будет сохранен результат работы;
- `pUnkOuter` — параметр не используется и должен быть равен нулю.

Создав экземпляр интерфейса `IDirectDrawClipper`, необходимо связать его с окном, выходы за пределы которого нужно отслеживать. Для этого вызываем метод `SetHWnd`, которому нужно передать два параметра: флаг, который не используется и должен быть равен нулю, а также идентификатор окна.

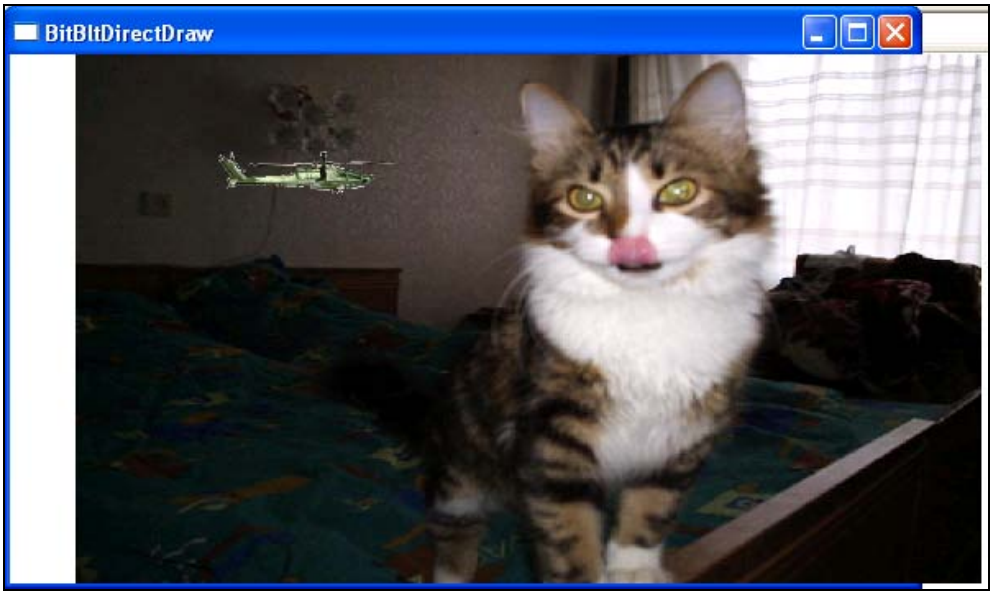

Рис. 2.3. Эффект выхода изображения за пределы окна

Теперь связываем первичную поверхность с объектом отсечения. Для этого у интерфейса поверхности есть метод `SetClipper`. У него только один параметр — объект отсечения.

Задний буфер отсекать не нужно, потому что это слишком сильно повлияет на производительность. В заднем буфере мы формируем будущее изображение, и во время каждого копирования данных объект отсечения будет проверять области. Уж лучше проверку реализовать в первичном буфере, чтобы отсечение произошло во время копирования данных из заднего буфера на передний. Таким образом, интерфейс отсечения сработает только один раз.

В виде кода все вышесказанное выглядит следующим образом:

```
ppiDD->CreateClipper(0, &ddClipper, 0);
ddClipper->SetHWnd(0, hWnd);
primsurf->SetClipper(ddClipper);
```

Добавьте этот код после создания поверхностей в примере, который мы создали в прошлой главе, и запустите программу. Даже если поверхность будет выходить за пределы окна, все лишнее автоматически отрежется.

Примечание

Исходный код примера находится в каталоге \Chapter2\Clipper на прилагаемом компакт-диске.

2.5. Прямой доступ к видеопамяти

Теперь нам предстоит затронуть самую важную и самую интересную область DirectDraw — прямой доступ к видеопамяти. Вспоминаются времена MS-DOS, когда без прямого доступа программировать игры и графику (в том числе и Demo) было просто невозможно. Да, программирование через прямой доступ достаточно сложно, но без него добиться необходимой скорости иногда просто нельзя.

Для получения указателя на видеопамять поверхности (если поверхность находится в системной памяти, то на системную память) используется метод `Lock` интерфейса поверхности. В общем виде метод выглядит следующим образом:

```
HRESULT Lock(
  LPRECT lpDestRect,
  LPDDSURFACEDESC lpDDSurfaceDesc,
  DWORD dwFlags,
  HANDLE hEvent
  );
```

Давайте посмотрим параметры, которые необходимо передать этому методу:

- `lpDestRect` — область Rect, которую необходимо заблокировать;
- `lpDDSurfaceDesc` — указатель на структуру типа `DDSURFACEDESC`, через которую мы получим информацию о блокированной поверхности;
- `dwFlags` — флаги, используемые при блокировании. Здесь можно указать одно из следующих значений:
  - `DDLOCK_WAIT` — если блокировка не может быть выполнена (например, происходит рисование на поверхности), то метод `Lock` будет ожидать освобождения поверхности и возможности блокировки;
  - `DDLOCK_EVENT` — если поверхность не может быть заблокирована, то в момент освобождения поверхности будет сгенерировано событие, которое указано в последнем параметре. Если существует несколько вызовов методов `Lock`, ожидающих освобождения поверхности, то события будут генерироваться по методу FIFO (First In First Out, первый вошел — первый вышел);

- DDLOCK_SURFACEMEMORYPTR — (это значение по умолчанию) указывает на необходимость вернуть правильный указатель на верхний левый угол прямоугольника экрана;

❒ hEvent — событие, которое должно быть сгенерировано, когда поверхность будет готова к блокированию.

После вызова метода Lock поверхность блокируется и не может перемещаться в памяти. Желательно указывать область, которую необходимо заблокировать, чтобы не блокировать всю поверхность.

Старайтесь блокировать поверхность только на короткий промежуток времени. Не стоит вызывать метод Lock, а потом выполнять какие-то расчеты, не работая с памятью поверхности. Старайтесь сначала выполнить расчеты, а потом уже блокировать.

Следующий пример показывает, как можно заблокировать поверхность:

```
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);

if (backsurf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
{
 // Поверхность заблокирована и здесь мы можем с ней работать
 // Указатель на память поверхности находится в desc.lpSurface
}
```

В этом примере мы сначала подготавливаем структуру DDSURFACEDESC2, через которую нам будет возвращена информация о поверхности, и указатель на память. В большинстве случаев мы будем работать только с двумя полями этой структуры: lpSurface, которое указывает на память поверхности и lPitch — количество байт в одной строке. Длина строки зависит от ширины экрана и размера пиксела и даже от видеокарты, поэтому без поля lPitch очень часто просто не обойтись.

Как же поле lPitch может зависеть о видеокарты? Допустим, что для ускорения графики видеокарта работает с поверхностями не побайтно, а сразу по 4 байта (DWORD). В этом случае ширина поверхности должна быть кратной 4. А если нам нужна ширина в 10 байт, то поверхность автоматически расширяется до 12, хотя работаем и отображаем мы только первые 10 байт. Остальные 2 используются только для кратности и могут содержать любой мусор, но они существуют. Таким образом, логическая ширина поверхности, которую вы запросили, может отличаться от физической, которая реально выделена. Если вы рассчитаете ширину строки как размер пиксела, умноженный на ширину

затребованной поверхности, то результат может оказаться не таким, как вы ожидали, и строки будут заполняться со смещением. Поэтому для расчета положения в памяти следующей строки обязательно используйте параметр `lPitch`.

Если прямой доступ больше не нужен, то необходимо разблокировать поверхность. Для этого используется метод `Unlock`, которому можно передать указатель на поверхность, которую надо разблокировать:

```
backsurf->Unlock(desc.lpSurface);
```

Почему "можно" указывать параметр? Да потому что необязательно. Параметр необходим, когда вы блокируете несколько участков одной и той же поверхности, а если всю сразу, то можно указать в этом параметре 0.

Вы должны учитывать, что когда поверхность заблокирована, то нельзя использовать отладчики, потому что ядро Windows в этот момент останавливается.

Давайте создадим интересный пример, который будет использовать прямой доступ к поверхности и выводить анимацию. В качестве анимации применим движущуюся градиентную заливку. Для этого создадим новое приложение и добавим в него инициализацию DirectDraw, как мы это делали ранее.

Теперь изменим главный цикл обработки сообщений, как это показано в листинге 2.7. Чтобы уменьшить листинг и сэкономить место в нем, не приводится код отображения заднего буфера на первичной поверхности при оконном режиме.

Листинг 2.7. Анимация в цикле обработки сообщений

```
int c=0;
while (true)
{
 // Проверяем, есть ли в очереди сообщения
 if (PeekMessage(&msg, NULL, NULL, NULL, PM_NOREMOVE))
 {
  // Сообщения есть, поэтому обрабатываем первое из них
  if (!GetMessage(&msg, NULL, NULL, NULL)) break;
  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
  {
   TranslateMessage(&msg);
   DispatchMessage(&msg);
  }
 }
```

```
 // Подготавливаем структуру для блокировки поверхности
 DDSURFACEDESC2 desc;
 ZeroMemory(&desc, sizeof(desc));
 desc.dwSize = sizeof(desc);

 // Блокируем поверхность заднего буфера
 if (backsurf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
 {
  // Получаем указатель на память поверхности
  BYTE* dst = (BYTE *)desc.lpSurface;

  // Цикл рисования поверхности
  for (int i=0; i<iHeight; i++)
   FillMemory(dst + (i*desc.lPitch), desc.lPitch, i+c);
  c++;

  // Разблокируем поверхность
  backsurf->Unlock(0);
 }

 // Переключаем вторичную поверхность на первичную
 if (bWindowed)
 {
  // Прорисовка поверхности для оконного режима
  ...
  ...
 }
 else
  primsurf->Flip(NULL, DDFLIP_WAIT);
}
```

Запустите пример и посмотрите результат его работы. На рис. 2.4 вы можете увидеть примерный результат, но рисунок не может передать анимации, поэтому лучше увидеть все своими глазами, прежде чем мы начнем разбирать листинг. А разбирать здесь есть что, потому что отображения графики в главном цикле обработки сообщений вместо события `WM_PAINT` мы будем достаточно часто использовать в будущих проектах.

В самом начале мы объявляем переменную `c` типа `int`, которая будет отвечать за анимацию, а точнее, начальный цвет, с которого будет строиться градиент. Далее запускается бесконечный цикл. Этот цикл можно прервать только оператором `break`:

```
while (true)
```

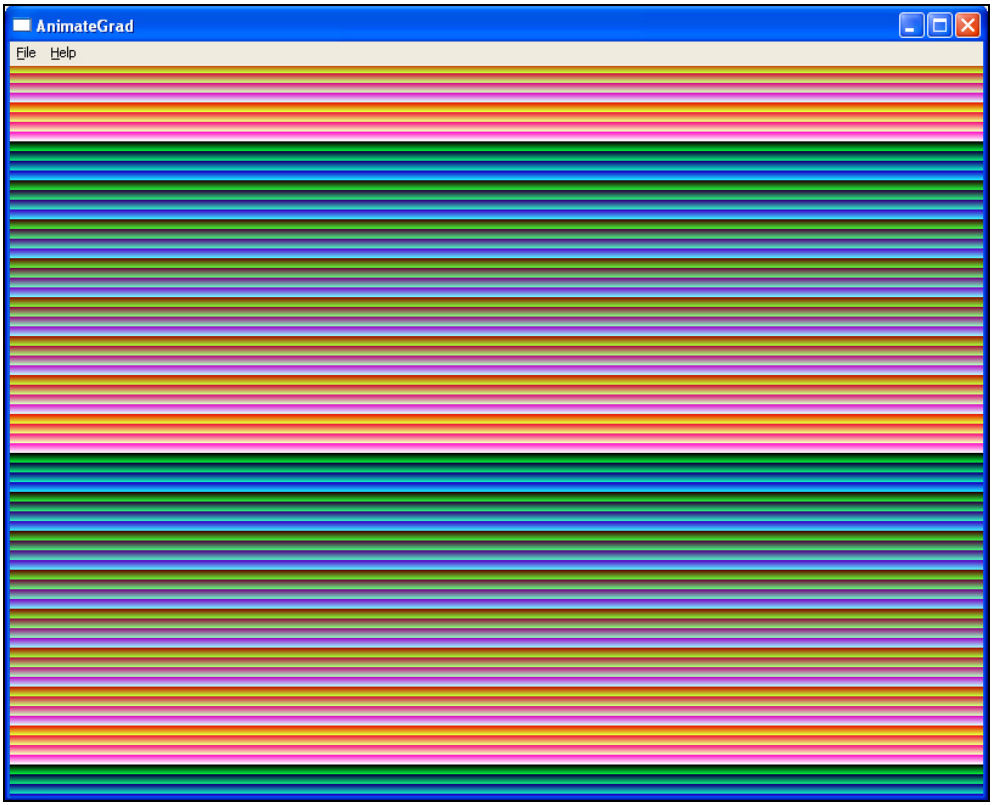

Рис. 2.4. Пример анимации через прямой доступ к поверхности

Внутри цикла идет модифицированный обработчик сообщений:

```
if (PeekMessage(&msg, NULL, NULL, NULL, PM_NOREMOVE))
{
  if (!GetMessage(&msg, NULL, NULL, NULL)) break;
  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
}
```

Обработчик по умолчанию использует для получения сообщений функцию `GetMessage`. Если для приложения есть сообщения, то функция возвращает первое сообщение из очереди. Если ничего нет, то функция "замораживает" работу основного потока, в котором работает приложение, и ожидает сообщения. Это нас не устраивает. Нам необходимо отлавливать сообщения, а

если их нет, то продолжать работу. Именно поэтому и приходится модифицировать обработчик событий.

Сначала с помощью WinAPI-функции `PeekMessage` мы проверяем, есть ли для нашего приложения сообщения. При этом последний параметр равен `PM_NOREMOVE`, т. е. не нужно удалять сообщение из очереди. Если сообщения есть, то функция вернет значение `TRUE` и продолжит работу, иначе `FALSE` и работа программы продолжится. Теперь нет никакой заморозки.

Если сообщения нет, то мы забираем его с помощью функции `GetMessage`. Если эта функция вернула ложь, то произошел выход из программы и цикл обработки сообщений необходимо прерывать. Если результат `true`, то тут уже идет классический разбор сообщения с помощью функций `TranslateMessage` и `DispatchMessage`.

Теперь начинается сам вывод графики. Сначала подготавливаем структуру типа `DDSURFACEDESC2`, которая понадобится нам для блокировки поверхности.

Далее происходит блокировка поверхности заднего буфера, дабы получить указатель на его память:

```
if (backsurf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
```

Если блокировка прошла удачно, то для удобства сохраняем указатель на память в переменной `dst`, которая будет иметь тип указателя на массив `BYTE` чисел:

```
BYTE* dst = (BYTE *)desc.lpSurface;
```

Теперь запускается цикл, который будет выполняться от нуля до количества строк в поверхности. На каждом этапе цикла мы будем заполнять отдельную строку определенным цветом. Цвет определяется счетчиком `i`.

Для заполнения строки цветом будем использовать функцию `FillMemory`, которая заполняет указанную память определенным числом:

```
FillMemory(dst + (i*desc.lPitch), desc.lPitch, i+c);
```

Первый параметр определяет начало памяти, где находится текущая строка для закраски. Второй параметр — размер строки, который берется из параметра `lPitch` структуры `desc`. Последний параметр — это число, которым нужно заполнять память, и это будет цвет. Цвет определяем числом `i` плюс корректировочное число `c`, благодаря которому и достигается эффект анимации.

Далее все уже нам знакомо, потому что тут идет вывод содержимого заднего буфера на передний.

Результат работы программы сильно зависит от формата пиксела. Поскольку мы заполняем цвет не попиксельно, а побайтно, не обращая внимания на размер одного пиксела.

Если вы запустили приложение и увидели результат работы, то у вас может создаться впечатление, что анимация слишком медленная. Да, мы использовали прямой доступ и выполнили достаточно простые операции, а движение не такое уж и быстрое, даже на очень быстром компьютере. А если графика будет сложнее, то анимация станет еще медленнее? Нет. Дело в том, что графическая программа не может работать быстрее частоты развертки монитора. Если монитор обновляется 60 раз в секунду, то приложение не может обновлять его чаще. Если в текущий момент происходит отображение одной поверхности, а мы уже сформировали другую поверхность заднего буфера, то до момента переключения придется подождать.

Если же посмотреть на нагрузку процессора, то она невысокая, и он способен сформировать более сложную графику без ущерба производительности.

Примечание

Исходный код примера находится в каталоге \Chapter2\AnimateGrid на прилагаемом компакт-диске.

2.6. Формат пиксела

Для прямого доступа к поверхности очень важно знать, уметь определять и использовать текущий формат пиксела. На самом деле, если нам понадобится максимальная производительность, то желательно привязаться только к одному формату и использовать его, потому что для поддержки разных форматов потребуется универсальный код, который будет переполнен проверками. И все же, универсальность иногда нужна, и никуда от нее не деться, поэтому нельзя обойти эту тему стороной.

Если сделать программу, которая будет поддерживать любой формат пиксела, то это придаст вам больше авторитета в глазах пользователя. С помощью формата пиксела пользователь сможет выбирать качество изображения и соответственно скорость работы программы, которая, безусловно, зависит от формата. При этом код немного увеличивается, но это стоит того. Дабы упростить себе жизнь, мы будем использовать для 16-битного режима формат пиксела R5G6B5, как более распространенный и используемый по умолчанию.

Единственное, что мы не будем рассматривать вовсе, — это 8-битный формат, как устаревший и не удовлетворяющий качеству изображения. Да, и из 8 битов умельцы умудрялись создавать шедевры, и тут нельзя не поклониться демо-сценаристам 80-х годов и программистам таких игр, как Doom и ей подобным. Это действительно шедевры, но нынешние времена требуют большего.

Для получения формата пиксела поверхности используется метод интерфейса поверхности `GetPixelFormat`. Этому методу передается только один параметр — указатель на структуру `DDPIXELFORMAT`. У этой структуры четыре поля, которые нам пригодятся:

- `dwRGBBitCount` — количество бит на пиксел;
- `dwRBitMask`, `dwGBitMask` и `dwBBitMask` — эти три поля определяют маску для каждого из цветов — красного, зеленого и голубого соответственно.

Следующий пример показывает, как определить формат пиксела:

```
DDPIXELFORMAT ftPixel;
ZeroMemory(&ftPixel, sizeof(ftPixel));
ftPixel.dwSize=sizeof(ftPixel);

backsurf->GetPixelFormat(&ftPixel);
```

По умолчанию для 16-битного режима, который мы использовали ранее, будет создан формат пиксела R5G6B5, и для него мы получим следующие значения масок цвета:

- `dwRGBBitCount` = 2;
- `dwRBitMask` = 1111100000000000 или число 63488;
- `dwGBitMask` = 0000011111100000 или число 2016;
- `dwBBitMask` = 0000000000011111 или число 31.

Как сформировать 16-битный цвет, если у нас он представлен 24 битами (по байту на каждую составляющую)? Это достаточно просто и несложно для процессора, если использовать сдвиг. Посмотрим на следующую формулу:

Голубой /8 + (Зеленый/4 << 5) + (Красный/8 << 11)

Каждая составляющая делится на 8 или четыре. Почему? Дело в том, что в 24-битном формате для хранения красного используется 8 бит (максимум 256, т. к. $2^8 = 256$), а в 16-битном только 5 бит и максимум равен 32 возможным значениям ($2^5 = 32$). Это значит, что число значение цвета 255 в 24-битном представлении цвета должно стать 31 для 16-битного, т. е. значение цвета должно уменьшиться в 8 раз (256/32). У зеленого цвета максимум 64 значения, поэтому тут значение цвета делим на 4 (256/64).

Помимо приведения мы должны произвести и смещение битов и установить каждый цвет в свою позицию. Голубой цвет начинается с нулевой позиции, поэтому его изменять не нужно, и оставляем на месте. Зеленый цвет начинается с 6 бита, а значит, его значение нужно сдвинуть на 5 бит. Красная составляющая начинается с 12 бита, а значит, сдвигаем его на 11 бит и складываем результат всех трех параметров.

Давайте напишем пример анимации, в котором цвет будет определяться в зависимости от текущего формата пиксела. Для этого немного модифицируем написанный в *разд. 2.5* пример. Для начала добавим до главного цикла обработки сообщений определение формата пиксела. Формат задается во время инициализации и после этого не изменяется, поэтому определить его нужно заранее.

Код блокировки и заполнения поверхности показан в листинге 2.8. После этого кода добавьте переключение поверхностей, и программа будет готова.

Листинг 2.8. Заполнение поверхности цветом, в зависимости от формата

```
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);

if (backsurf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
{
 BYTE iBlue=0;
 BYTE iGreen=0;
 BYTE iRed=255;

 // Заполнить поверхность для 16-битного режима
 if (ftPixel.dwRGBBitCount==16)
 {
  WORD* dst = (WORD *)desc.lpSurface;

  for (int i=0; i<iHeight; i++)
  {
   for (int j=0; j<desc.lPitch; j++)
     *(WORD *)(dst+j) = (WORD)(iRed/8+c | (iGreen/4+i << 5) |
           ((iBlue/8+j) << 11));
   dst=dst+desc.lPitch;
  }
 }

 // Заполнить поверхность для 24-битного режима
 if (ftPixel.dwRGBBitCount==24)
 {
  BYTE* dst = (BYTE *)desc.lpSurface;

  for (int i=0; i<iHeight; i++)
  {
   for (int j=0; j<desc.lPitch; j++)
```

```
    {
      *(BYTE *)(dst+j) = iRed+j;
      j++;
      *(BYTE *)(dst+j) = iGreen+i;
      j++;
      *(BYTE *)(dst+j) = iBlue+c;
    }
   dst=dst+desc.lPitch;
  }
 }
 c++;
 backsurf->Unlock(0);
}
```

Здесь мы заполняем вручную всю поверхность вдоль и поперек побайтно. Только так можно производить заполнение произвольным цветом.

Внимание!

В данном примере реализованы лишь 16- и 24-битные форматы пиксела. Если же пример запустить с 32-битным цветом, то в результате на экране будет мусор, потому что в этом случае поверхность не будет заполняться. Если вы используете оконные приложения и прямой доступ к памяти, то необходимо реализовывать поддержку всех возможных форматов пиксела, иначе программа будет работать не корректно.

Результат работы программы показан на рис. 2.5.

Теперь посмотрим, как достигается такой результат. После блокирования поверхности объявляется три переменные: `iBlue`, `iGreen` и `iRed`, для хранения значений цветов. Это единственно необходимое подготовительное действие.

Далее код делится на две части для формата пиксела 16 и 24 бит. Из-за отличий в представлении данных приходится формировать картинку по-разному. Если поверхность 16-битная, то выполняется следующий код:

```
WORD* dst = (WORD *)desc.lpSurface;

for (int i=0; i<iHeight; i++)
{
  for (int j=0; j<desc.lPitch; j++)
     *(WORD *)(dst+j) = (WORD)(iBlue/8+c | (iGreen/4+i << 5) |
                     ((iRed/8+j) << 11));

  dst=dst+desc.lPitch;
}
```

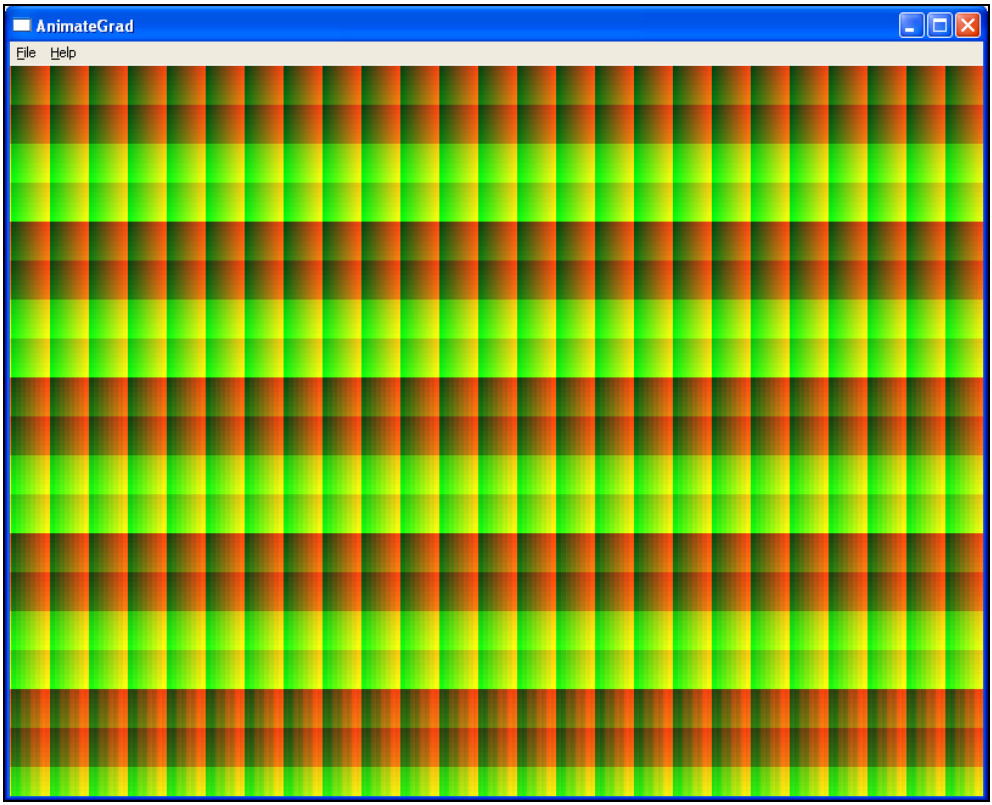

Рис. 2.5. Результат работы программы

В данном случае мы воспринимаем поверхность, как массив из `WORD`-чисел. В результате, наша задача состоит в том, чтобы просто заполнить каждый элемент этого массива, и поверхность будет готова.

Далее идет код заполнения поверхности для 24-битного формата пиксела:

```
if (ftPixel.dwRGBBitCount==24)
 {
  BYTE* dst = (BYTE *)desc.lpSurface;

  for (int i=0; i<iHeight; i++)
  {
   for (int j=0; j<desc.lPitch; j++)
   {
     *(BYTE *)(dst+j) = iRed+j;
     j++;
     *(BYTE *)(dst+j) = iGreen+i;
```

```
      j++;
      *(BYTE *)(dst+j) = iBlue+c;
    }
    dst=dst+desc.lPitch;
  }
 }
```

Тут приходится бороться с одним неудобством — цвет представлен 24 битами, а процессор привык работать с числами, кратными 2, т. е. 8 (BYTE), 16 (WORD), 32 (DWORD). Из-за этого приходится воспринимать пиксел как три числа типа BYTE и заполнять поверхность каждой составляющей цвета в отдельности.

Пример получается достаточно универсальным и быстрым. Конечно же, можно было использовать маски и написать более удобное решение в виде одного цикла, но в этом случае понизилась бы скорость заполнения поверхности. В данном же случае для каждого формата пиксела будет свой код заполнения поверхности, и он может быть оптимизирован именно для этого формата. Но при этом, усложняется поддержка кода и увеличивается размер программы. В чем сложность поддержки? Если придется изменить логику заполнения, то потребуется и корректировать каждый цикл заполнения в отдельности.

В этом примере мы не контролируем размер числа в переменной c, поэтому оно может превышать формат пиксела и выходить за пределы (перекрывать биты других составляющих цвета). Это сделано нарочно, потому что в результате мы получаем достаточно интересный эффект плавного перехода цвета.

Примечание

Исходный код примера находится в каталоге \Chapter2\PixelFormat на прилагаемом компакт-диске.

2.7. Потеря поверхностей

В *разд. 2.2* мы создали приложение, которое выводит на экран изображения. Попробуйте запустить его в полном экране и потом переключиться на другое приложение с помощью комбинации клавиш <Alt>+<Tab>. В ответ на это, приложение DirectDraw будет свернуто на панель задач. Теперь попробуйте вернуться в приложение DirectDraw. В лучшем случае вы увидите белый экран, а в худшем — мусор. Почему и как с этим бороться? Попробуем разобраться.

Когда мы переходим в полноэкранный режим и получаем эксклюзивные права на видео, то ОС позволяет нам делать с видеопамятью все, что угодно. Но

когда управление возвращается ОС, то она не может контролировать, что мы натворили в памяти, и просто отбирает все, чтобы прорисовать рабочий стол и окна на нем. Из-за этого в нашей программе остаются ссылки на поверхности в видеопамяти, но эти ссылки могут указывать на мусор, который натворила ОС, когда получила права на выполнения. Такая ситуация называется *потерей поверхностей* и восстанавливать все придется вручную собственными ручками.

Чтобы узнать, потеряна ли поверхность, можно вызвать метод интерфейса поверхности `IsLost`. Метод вернет истину, если указатель показывает на мусор. В этом случае для восстановления поверхности нужно вызвать сначала метод `Restore` для восстановления памяти. Если поверхность содержит какие-то данные, то эти данные будут потеряны и их придется перерисовать. Тут нужно учитывать, что потеряются и загруженные картинки, а значит, их придется перезагружать заново.

Поскольку операция восстановления поверхностей происходит достаточно редко, ведь не каждую же секунду происходит переключение между DirectDraw и рабочим столом, то можно написать отдельную функцию, которая будет все проверять и восстанавливать.

Некоторые программисты любят перед каждой прорисовкой проверять необходимость восстановления поверхности. В принципе, достаточно проверить первичную поверхность, потому что она теряется первой. Если она "жива", то все остальные точно будут живы. Поэтому функция может выглядеть следующим образом:

```
void RestoreSurfaces()
{
 if (primsurf->IsLost())
  {
    primsurf->Restore();

    if (backsurf->IsLost())
      backsurf->Restore();

    // Проверка и восстановление других поверхностей
  }
}
```

Но вызывать метод перед каждой прорисовкой слишком глупо. Как мы уже знаем, один вызов функции и возврат из нее требует процессорного времени, да и внутри функции будет проверка метода `IsLost()`.

Намного более эффективным методом я считаю просто проверить результат первого вызова метода `Blt`. Если во время прорисовки поверхности произошла ошибка `DDERR_SURFACELOST`, то поверхность потеряна. Только в этом случае

будет вызываться функция восстановления поверхностей. Таким образом, в коде отображения графики, который выполняется достаточно часто, добавляется всего одна операция проверки `if`, а вызов функции происходит только если это необходимо.

Давайте реализуем эту логику на примере, который мы рассмотрели в *разд. 2.2*. Для начала находим первый вызов метода `Blt` и добавляем проверку:

```
if (backsurf->Blt(&dst, imagesurf, 0, DDBLT_WAIT,
         0)==DDERR_SURFACELOST)
{
  RestoreSurfaces();
  backsurf->Blt(&dst, imagesurf, 0, DDBLT_WAIT, 0);
}
```

Если результат вызова `Blt` равен `DDERR_SURFACELOST`, то сначала вызывается функция `RestoreSurfaces`, а потом повторяется попытка прорисовки, уже на восстановленных поверхностях.

Функция `RestoreSurfaces` для данного примера показана в листинге 2.9. В ней мы уже уверены, что какие-то (а может и все) поверхности потеряны, поэтому проверяем каждую из них.

Листинг 2.9. Функция восстановления поверхностей

```
void RestoreSurfaces()
{
       if (primsurf->IsLost())
              primsurf->Restore();

       if (backsurf->IsLost())
              backsurf->Restore();

       if (imagesurf->IsLost())
       {
              imagesurf->Release();
              imagesurf = LoadBMPToSurface("desktop.bmp", ppiDD);
       }

       if (imagesurf2->IsLost())
       {
              imagesurf2->Release();
              imagesurf2 = LoadBMPToSurface("vertol.bmp", ppiDD);
       }
}
```

Обратите внимание, что поверхности `imagesurf` и `imagesurf2`, которые содержат картинки для восстановления, сначала уничтожаются, а потом вызывается функция `LoadBMPToSurface` для повторной загрузки изображения.

То, что поверхности с картинками приходится уничтожать и восстанавливать заново — является недостатком нашей функции `LoadBMPToSurface`. Чтобы решить эту проблему, можно перегрузить ее и создать еще один вариант, который в качестве параметра будет получать указатель на поверхность. Новый вариант функции будет выглядеть, как это показано в листинге 2.10.

Листинг 2.10. Функция загрузки изображения с проверкой существования поверхности

```
IDirectDrawSurface7 *LoadBMPToSurface(IDirectDrawSurface7 **ddsurf,
   LPCTSTR filename, DirectDraw7 *ppiDD)
{
 IDirectDrawSurface7 *surf;
 surf=*ddsurf;
 HANDLE hBmp;
 BITMAP bm;

 // Загрузка картинки из файла
 hBmp = LoadImage(0, filename, IMAGE_BITMAP, 0, 0,
         LR_LOADFROMFILE | LR_CREATEDIBSECTION);
 if (hBmp==0)
   return 0;
 GetObject(hBmp, sizeof(bm), &bm);

 if (surf==NULL)
 {
  // Подготавливаем структуру для создания поверхности
  ...

  // Создание поверхности
  ...
 }

// Загрузка картинки
...
```

Теперь, на этапе загрузки программы и изображений мы можем использовать первый вариант функции, а на этапе восстановления второй. В этом случае восстановление будет выглядеть так:

```
if (imagesurf2->IsLost())
{
 imagesurf2->Restore();
 imagesurf2 = LoadBMPToSurface(&imagesurf2,
        "vertol.bmp", ppiDD);
}
```

В этом случае мы только восстанавливаем поверхность без создания заново, что немного быстрее. Сама функция `LoadBMPToSurface` проверит, если переданный указатель поверхности существует, то она будет использовать уже существующую поверхность, иначе будет создана новая.

Примечание

Исходный код примера находится в каталоге \Chapter2\Lost на прилагаемом компакт-диске.

2.8. Определение поддерживаемых режимов

До этого момента все программы, которые мы писали, были привязаны к заранее определенному разрешению. Мы даже не проверяли, поддерживается ли данный режим видеокартой и монитором. Проверку мы опустили, потому что использовалось разрешение 800×600 с глубиной цвета в 16 бит, которое поддерживается большинством видеокарт и мониторов. В настоящее время такие параметры даже являются минимумом для Windows XP, поэтому можно быть уверенным, что компьютер будет поддерживать нужный нам режим.

Но такое решение не является оптимальным. Чем выше выбрать разрешение, тем более качественно будет выглядеть картинка. К тому же, некоторые мониторы могут плохо отображать информацию при неоптимальном для данной модели разрешении. В настоящее время широкое распространение получают широкоформатные экраны, для которых классическое разрешение 800×600 может оказаться растянутым по горизонтали, что вызовет ненужные искажения.

Вполне логично научить нашу программу определять параметры, которые поддерживает данный компьютер, и давать пользователю возможность самому выбрать необходимые параметры в зависимости от экрана и мощности процессора и видеосистемы.

Чтобы определить, какие видеорежимы поддерживает система, у интерфейса `IDirectDraw` есть метод `EnumDisplayModes`, который выглядит следующим образом:

```
HRESULT EnumDisplayModes(
  DWORD dwFlags,
  LPDDSURFACEDESC lpDDSurfaceDesc,
  LPVOID lpContext,
  LPDDENUMMODESCALLBACK lpEnumCallback
  );
```

В данном случае имеется 4 параметра:

- `dwFlags` — флаги, которые сейчас не используются, и параметр должен быть равен нулю;
- `lpDDSurfaceDesc` — структура типа `DDSURFACEDESC`, по полям которой определяется тип режимов, получаемых нами. Если в этой структуре заполнить поле глубины цвета, то будут отображены режимы только с этой глубиной;
- `lpContext` — перечисление происходит через вызов функции, которой будет передаваться структура с описанием найденного режима и пользовательская переменная. Через этот параметр вы можете задать любое значение, передаваемое в функцию обратного вызова;
- `lpEnumCallback` — здесь нам нужно указать функцию обратного вызова, которая будет вызываться в ответ на найденный поддерживаемый режим.

Функция обратного вызова должна выглядеть следующим образом:

```
HRESULT WINAPI dmFound(LPDDSURFACEDESC2 desc, LPVOID p)
{
}
```

Первый параметр — структура типа `DDSURFACEDESC2`, в которой описывается информация о найденном режиме. Во втором параметре мы получаем то же самое значение, что было указано в третьем параметре (`lpContext`) метода `EnumDisplayModes`.

Давайте создадим приложение, которое позволит перед запуском выбирать режим. Для этого нам понадобятся следующие глобальные переменные:

```
struct dsModes
{
  DWORD width, height, depth;
};
dsModes dmList[100];
int dsModesTotal=0;
```

Для удобства хранения информации о найденных режимах мы ввели структуру `dsModes`, которая будет хранить ширину, высоту и глубину цвета. Затем

мы объявляем список из структур в 100 элементов и переменную `dsModesTotal`, которая должна определять количество режимов.

Теперь создадим диалоговое окно, которое будет отображать список найденных режимов. В окне нам понадобятся только кнопки **OK** и **Cancel**, а также компонент ListBox (рис. 2.6).

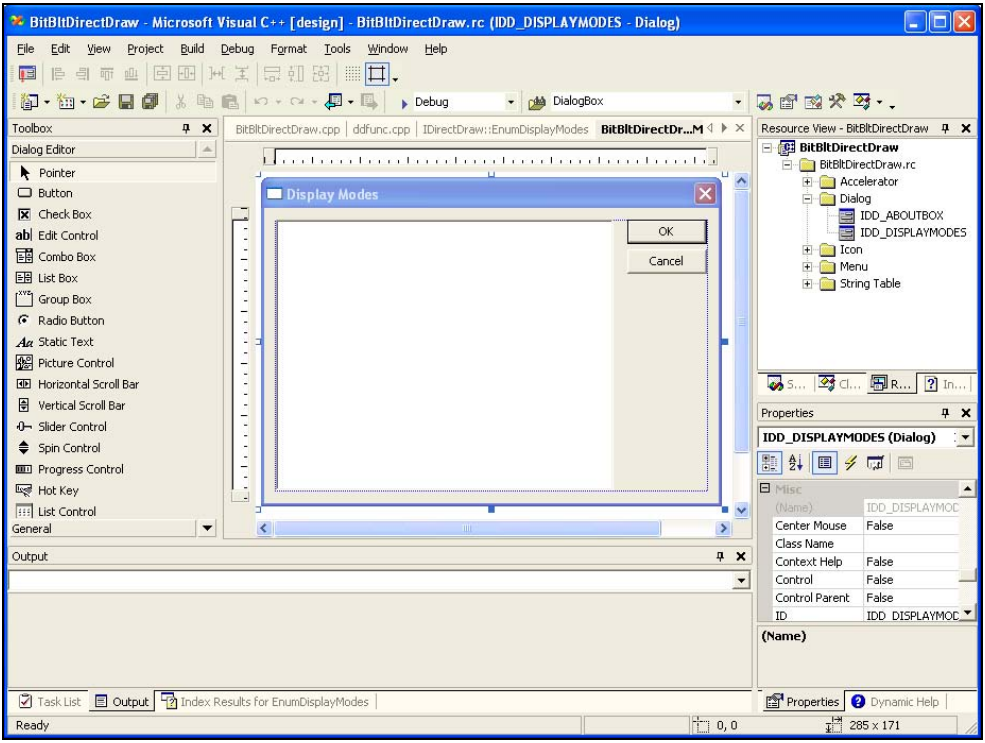

Рис. 2.6. Пример окна для отображения найденных ресурсов

При создании окна будьте внимательны — по умолчанию список ListBox создается с отсортированными элементами, а это не есть хорошо, потому что невозможно будет сопоставить список из ListBox со списком режимов в списке из переменной `dmList`. Чтобы не встретиться с этой проблемой, обязательно отключите сортировку.

Для этого окна нам понадобится и функция, которая будет обрабатывать события:

```
LRESULT CALLBACK DisplayModes(HWND, UINT, WPARAM, LPARAM);
```

Саму функцию мы рассмотрим чуть позже, а сейчас разберем код создания окна диалога. Его нужно сделать до того, как будет проинициализирован DirectDraw. Пример кода отображения окна представлен в листинге 2.11.

Листинг 2.11. Пример отображения окна выбора режима

```
// Временно создаем интерфейс IDirectDraw
HRESULT hr;
hr = DirectDrawCreateEx(NULL, (VOID**)&ppiDD, IID_IDirectDraw7, NULL);
if(FAILED(hr))
  return 0;

// Отображение окна и получение выбранного элемента
int nItemSelected=DialogBox(hInst, (LPCTSTR)IDD_DISPLAYMODES,
         hWnd, (DLGPROC)DisplayModes);
if (nItemSelected==65535)
  return FALSE;

bWindowed=TRUE;
iWidth=dmList[nItemSelected].width;
iHeight=dmList[nItemSelected].height;

// Уничтожение временно созданного DirectDraw
if (ppiDD)
{
  ppiDD->Release;
  ppiDD=NULL;
}
```

В самом начале мы временно получаем интерфейс `IDirectDraw`, чтобы можно было получить список доступных режимов. После получения списка переменную можно уничтожить, потому что она будет проинициализирована в нашей функции инициализации DirectDraw — `DXDDInit`.

Далее, отображаем диалоговое окно с помощью WinAPI-функции `DialogBox`. В качестве результата функция будет возвращать выделенный элемент или 65 535 (если бы результат был бы знаковым, то это число соответствовало бы –1), если пользователь ничего не выбрал или нажал кнопку отмены. Если режим не выбран, то прерываем функцию инициализации со значением `FALSE`, чтобы завершить работу программы. Если режим выбран, то определяем его параметры через список `dmList`.

В листинге 2.12 показан код функции обработки сообщений, окна выбора видеорежима.

Листинг 2.12. Функция обработки сообщений, окна выбора режима

```
LRESULT CALLBACK DisplayModes(HWND hDlg, UINT message,
  WPARAM wParam, LPARAM lParam)
```

```
{
 HWND hWndListBox;
 switch (message)
 {
  case WM_INITDIALOG:
    hWndListBox = GetDlgItem(hDlg, IDC_LIST1);
    ppiDD->EnumDisplayModes(0, 0, hWndListBox, dmFound);
    return TRUE;

  case WM_COMMAND:
    if (LOWORD(wParam) == IDOK || LOWORD(wParam) == IDCANCEL)
    {
      if (LOWORD(wParam) == IDOK)
      {
        HWND hWndListBox = GetDlgItem(hDlg, IDC_LIST1);
        wParam = (int)SendMessage( hWndListBox, LB_GETCURSEL, 0, 0 );
      }
      else
        wParam = 65535;

      EndDialog(hDlg, LOWORD(wParam));
      return TRUE;
    }
    break;
 }
 return FALSE;
}
```

По событию `WM_INITDIALOG` (во время инициализации диалогового окна) выполняется два действия:

1. С помощью функции `GetDlgItem` определяется указатель на элемент управления ListBox, который необходимо заполнить доступными режимами. Мы делаем это заранее, чтобы потом не пришлось определять каждый раз при нахождении очередного режима.
2. Вызывается метод `EnumDisplayModes` для начала поиска режимов. Первый и второй параметры равны нулю, а значит, будут отображаться все типы режимов. Третий параметр (произвольное значение) будет передавать функции обратного вызова идентификатор элемента управления ListBox. Последний параметр — название функции обратного вызова `dmFound`.

По событию `WM_COMMAND` мы проверяем, была ли нажата кнопка **OK** или **Cancel**, и если да, то нам необходимо закрыть окно. Кроме того, если нажата

кнопка **ОК**, то мы определяем текущий выделенный элемент в списке выбора режимов, чтобы вернуть это значение:

```
wParam = (int)SendMessage( hWndListBox, LB_GETCURSEL, 0, 0 );
```

Функция обратного вызова `dmFound` показана в листинге 2.13.

Листинг 2.13. Функция обратного вызова определения доступных видеорежимов

```
HRESULT WINAPI dmFound(LPDDSURFACEDESC2 desc, LPVOID p)
{
 if (desc->ddpfPixelFormat.dwRGBBitCount==16)
 {
  TCHAR szTemp[MAX_PATH];
  _sntprintf(szTemp, MAX_PATH-1, TEXT("%d x %d - %d"),
       desc->dwWidth,
       desc->dwHeight,
       desc->ddpfPixelFormat.dwRGBBitCount);

  SendMessage((HWND)p, LB_ADDSTRING, 0, (LPARAM) TEXT(szTemp));

  dmList[dsModesTotal].width  = desc->dwWidth;
  dmList[dsModesTotal].height = desc->dwHeight;
  dmList[dsModesTotal].depth  = desc->ddpfPixelFormat.dwRGBBitCount;
  dsModesTotal++;
 }

 return DDENUMRET_OK;
}
```

Через параметр `desc` мы получаем структуру, в которой хранится описание найденного видеорежима. В этой структуре нас будут интересовать следующие поля:

- `dwWidth` — ширина экрана;
- `dwHeight` — высота экрана;
- `ddpfPixelFormat.dwRGBBitCount` — глубина цвета.

Второй параметр функции — это произвольный параметр, в котором мы передали идентификатор списка выбора, куда нужно добавить найденный режим.

В данном примере я добавляю режим только в том случае, если глубина цвета равна 16 (параметр `ddpfPixelFormat.dwRGBBitCount` структуры `desc` должен

быть равен 16). Конечно же, ограничить вывод определенными видеорежимами мы могли бы с помощью второго параметра при вызове метода `EnumDisplayModes`, но я не стал этого делать.

Сначала все эти значения формируются в строку следующего вида:

```
Ширина × Высота — Глубина цвета
```

Эта строка добавляется в список выбора ListBox.

После этого информация о найденном режиме добавляется в массив `dmList`.

В качестве результата функция обратного вызова возвращает константу `DDENUMRET_OK`, это означает, что обработка прошла успешно и можно получать следующий видеорежим.

Пример окна выбора режима представлен на рис. 2.7.

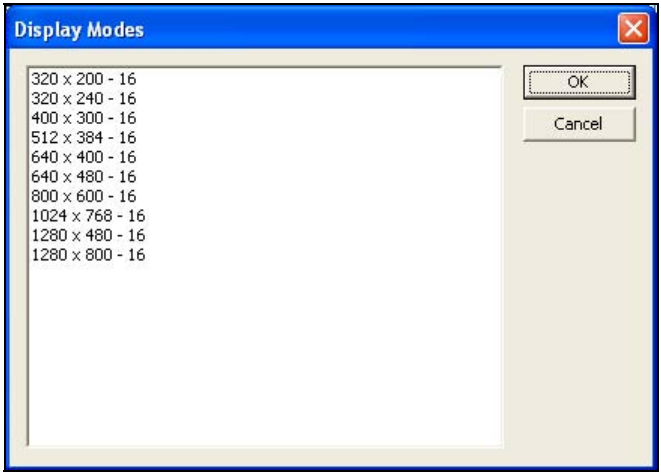

Рис. 2.7. Пример результата работы окна выбора видеорежимов

Попробуйте запустить программу и убедиться, что поддерживаемые видеорежимы отображаются, и программа правильно их выбирает.

Если вы позволяете пользователю выбирать видеорежимы, то будьте осторожны при задании частоты развертки при вызове метода `SetDisplayMode`. Дело в том, что разные режимы могут поддерживать разную частоту, а некоторые вообще позволяют использовать частоту только по умолчанию, поэтому в примерах к данной книге мы будем применять именно значение по умолчанию, т. е. у функции `SetDisplayMode` четвертый параметр будет равен нулю.

Интерфейс DirectDraw позволяет определить поддерживаемую частоту экрана для каждого видеорежима, но мы опустим эту тему, чтобы сэкономить место.

Примечание

Исходный код примера находится в каталоге \Chapter2\Modes на прилагаемом компакт-диске.

2.9. Отображение в Direct3D

Начнем знакомство с Direct3D с рассмотрения того, как можно очищать устройство и отображать сцену. Проект инициализации, который мы рассмотрели в *разд. 1.5*, ничего не делал. Единственное, что мы могли увидеть, — белый экран. Давайте посмотрим, как можно закрасить экран и отобразить сцену на экране.

Как и в случае с DirectDraw, создание сцены происходит в заднем буфере и только потом происходит переключение на передний план. Единственное различие — в Direct3D можно не заботиться, в каком именно буфере происходит рисование.

Основной интерфейс Direct3D — это `IDirect3DDevice9`. Именно у него есть необходимые для отображения методы. Так, например, для очистки используется метод `Clear`, который выглядит следующим образом:

```
HRESULT Clear(
    DWORD Count,
    const D3DRECT *pRects,
    DWORD Flags,
    D3DCOLOR Color,
    float Z,
    DWORD Stencil
);
```

- `Count` — количество прямоугольников. Метод может закрашивать сразу несколько прямоугольников, указатель на которые должен быть во втором параметре;
- `pRects` — указатель на структуру, которая будет определять закрашиваемый прямоугольник. Если параметр равен нулю, то закрашивается вся область. Структура должна иметь вид `D3DRECT`:

```
typedef struct _D3DRECT {
    LONG x1, y1;
    LONG x2, y2;
} D3DRECT;
```

- `Flags` — параметры закраски. Здесь можно указать одно из следующих значений:

- D3DCLEAR_TARGET — очистить отображение. Значение цвета указано в параметре Color;
- D3DCLEAR_STENCIL — очистить stencil-буфер (буфер трафарета). Значение, которым будет заполнен буфер, указано в параметре Stencil;
- D3DCLEAR_ZBUFFER — очистить буфер глубины. Значение очистки находится в параметре Z;

- Color — цвет, которым нужно заполнить буфер отображения;
- Z — новое значение, которое будет помещено в буфер глубины. Значение параметра может быть от 0.0 (ближайшее расстояние) до 1.0 (дальнее расстояние);
- Stencil — значение, которым будет заполнен каждый буфер трафарета.

Посмотрим пример очистки экрана синим цветом:

```
RECT  r;
SetRect(&r, 0, 0, iWidth, iHeight);
pD3DDevice->Clear(1, (D3DRECT*)&r, D3DCLEAR_TARGET,
       D3DCOLOR_XRGB(0,0,255), 1.0, 0);
```

Обратите внимание, как в четвертом параметре мы передаем цвет. Для этого используется макрос D3DCOLOR_XRGB, которому необходимо передать три составляющих цвета, а на выходе получаем цвет в формате D3D. В Direct3D есть еще несколько макросов, упрощающих создание цвета в нужном формате:

- D3DCOLOR_RGBA — создание цвета в формате красный, зеленый, синий и альфа. Все эти четыре составляющие должны передаваться в качестве параметров в такой же последовательности;
- D3DCOLOR_ARGB — создание цвета в формате альфа, красный, зеленый и синий.

Самое интересное — это посмотреть, как выглядят макросы, и стоит ли их использовать. Быть может, есть способ лучше. Макрос D3DCOLOR_ARGB имеет следующий вид:

```
#define D3DCOLOR_ARGB(a,r,g,b) \
    ((D3DCOLOR)((((a)&0x00)<<24)|(((r)&0x00)<<16)|
    (((g)&0x00)<<8)|((b)&0x00)))
```

Макрос возвращает тип D3DCOLOR, который на самом деле является числом-размером в два слова (DWORD). В этом числе в первый байт заносится значение альфа, во второй значение красного цвета и т. д. Получается, что макрос просто формирует цвет в 32- или 4-байтном битном виде и возвращает его.

Все остальные макросы просто вызывают макрос `D3DCOLOR_ARGB` с необходимыми параметрами. Поскольку все это макросы, то не стоит беспокоиться о вызовах, как в случае с функциями, потому что вызовов нет (код макроса просто вставляется в то место, где мы его использовали), и потери скорости тут не будет. С другой стороны, код программы может оказаться перенасыщенным одним и тем же кодом формирования D3D-цвета.

Даже если написать этот код по событию `WM_PAINT`, то ничего пока не произойдет, и экран останется еще белым. Чтобы созданная сцена (в данном случае закраска) отобразилась, необходимо вызвать метод `Present`, который выглядит следующим образом:

```
HRESULT Present(
    CONST RECT *pSourceRect,
    CONST RECT *pDestRect,
    HWND hDestWindowOverride,
    CONST RGNDATA *pDirtyRegion
);
```

Давайте рассмотрим параметры этого метода.

- `pSourceRect` — область копирования. Параметр должен быть равен нулю, если вы не использовали во время создания тип переключения поверхностей `D3DSWAPEFFECT_COPY`. Если указан — 0, то копироваться будет вся поверхность отображения;
- `pDestRect` — область приемника, куда нужно скопировать данные. Этот параметр также должен быть равен нулю, если во время создания интерфейса IDirect3D не использовался параметр `D3DSWAPEFFECT_COPY`;
- `hDestWindowOverride` — указатель на окно, где нужно отобразить сцену;
- `pDirtyRegion` — прямоугольники определяют минимальный набор пикселов, которые должны быть обновлены. Остальная часть сцены не обновляется. Параметр используется, только если при создании применялся параметр `D3DSWAPEFFECT_COPY`.

При полноэкранном режиме лучше все параметры оставить равными нулю, т. е. отображение будет выглядеть следующим образом:

```
pD3DDevice->Present(NULL, NULL, 0, NULL);
```

Примечание

Исходный код примера находится в каталоге \Chapter2\D3Dclear на прилагаемом компакт-диске.

2.10. Примитивы Direct3D

С основами DirectDraw мы покончили, и если нам что-то еще понадобится, то мы рассмотрим это по мере надобности. А сейчас переходим к Direct3D, который является более перспективным.

Если DirectDraw не использует примитивов, возлагая их создание на программистов, а только дает нам интерфейсы для быстрого доступа к памяти, то создание трехмерных сцен без этого невозможно. В Direct3D есть три примитива — точка, отрезок и треугольник. Все остальные примитивы можно построить с помощью этих трех составляющих. Конечно же, основной примитив, с которым мы будем работать, — это треугольник, но рассмотрим все.

Почему в Direct3D в качестве основы используется именно треугольник, и нет более сложных фигур, упрощающих создание графики? Дело в том, что сами видеоускорители при формировании графики оперируют именно треугольниками, поэтому логичнее будет, если программа будет "разговаривать" с устройством на одном языке и оперировать одними и теми же данными.

Работа с Direct3D значительно отличается от того, что мы рассматривали, поэтому давайте писать пример и одновременно знакомиться с функциями, которые используются. Просто без толку нагрузить ваш мозг функциями без реального примера будет слишком накладно.

Не будем слишком упрощать себе задачу, а сразу же в качестве примера начнем писать приложение, вращающее куб. Каждая сторона его должна быть окрашена в свой цвет, чтобы вы могли увидеть, что это куб. Дело в том, что у нас пока не будет освещения, без которого монолитно окрашенная фигура будет казаться плоской. Разный цвет каждой грани позволит увидеть объемность фигуры без освещения.

Для начала создадим приложение и инициализируем Direct3D, как мы это делали раньше. Благо, для этого достаточно вызвать только одну функцию `DX3DInit`, которую мы написали в *главе 1*. После инициализации должна вызываться функция, которая будет инициализировать все необходимое для создания куба. Назовем эту функцию `CreateQuad` и вызовем ее после инициализации. Функция показана в листинге 2.14.

Листинг 2.14. Функция инициализации куба

```
void CreateQuad()
{
  // Массив вершин куба
  const sVertex svVortexList[] = {
```

```
      // Грань 1
  -0.5f, -0.5f, -0.5f, 0x00ff0000,
  -0.5f,  0.5f, -0.5f, 0x00ff0000,
   0.5f, -0.5f, -0.5f, 0x00ff0000,
   0.5f,  0.5f, -0.5f, 0x00ff0000,

      // Грань 2
   0.5f, -0.5f, -0.5f, 0x0000ff00,
   0.5f,  0.5f, -0.5f, 0x0000ff00,
   0.5f, -0.5f,  0.5f, 0x0000ff00,
   0.5f,  0.5f,  0.5f, 0x0000ff00,

      // Грань 3
   0.5f, -0.5f,  0.5f, 0x000000ff,
   0.5f,  0.5f,  0.5f, 0x000000ff,
  -0.5f, -0.5f,  0.5f, 0x000000ff,
  -0.5f,  0.5f,  0.5f, 0x000000ff,

      // Грань 4
  -0.5f, -0.5f,  0.5f, 0x00ffff00,
  -0.5f,  0.5f,  0.5f, 0x00ffff00,
  -0.5f, -0.5f, -0.5f, 0x00ffff00,
  -0.5f,  0.5f, -0.5f, 0x00ffff00,

      // Грань 5
  -0.5f,  0.5f, -0.5f, 0x0000ffff,
  -0.5f,  0.5f,  0.5f, 0x0000ffff,
   0.5f,  0.5f, -0.5f, 0x0000ffff,
   0.5f,  0.5f,  0.5f, 0x0000ffff,

      // Грань 6
   0.5f, -0.5f, -0.5f, 0x00ff00ff,
   0.5f, -0.5f,  0.5f, 0x00ff00ff,
  -0.5f, -0.5f, -0.5f, 0x00ff00ff,
  -0.5f, -0.5f,  0.5f, 0x00ff00ff,
};

void * pBuf;

// Создание буфера вершин
HRESULT hRes = pD3DDevice->CreateVertexBuffer(sizeof(sVertex) *
      iVertsNum,  0 , D3DFVF_XYZ | D3DFVF_DIFFUSE,
      D3DPOOL_DEFAULT, &vBuffer, 0);
```

```
  if (FAILED(hRes))
    return;

  // Блокировка буфера вершин и заполнение
  hRes = vBuffer->Lock(0, sizeof(sVertex) * iVertsNum, &pBuf, 0);
  if (FAILED(hRes))
    return;
  memcpy(pBuf, svVortexList, sizeof(sVertex) * iVertsNum);
  vBuffer->Unlock();

  // Массив индексов построения треугольника
  const unsigned short Idxes[]={
    0,1,2,2,1,3,
    4,5,6,6,5,7,
    8,9,10,10,9,11,
    12,13,14,14,13,15,
    16,17,18,18,17,19,
    20,21,22,22,21,23,
  };

  // Создание буфера индексов
  hRes = pD3DDevice->CreateIndexBuffer(sizeof(short) * iIdxNum,
       D3DUSAGE_WRITEONLY, D3DFMT_INDEX16, D3DPOOL_DEFAULT,
       &iBuffer, 0);
  if (FAILED(hRes))
    return;

  // Блокировка буфера индексов и заполнение
  hRes = iBuffer->Lock(0, sizeof(short) * iIdxNum, &pBuf, 0);
  if (FAILED(hRes))
    return;
  memcpy(pBuf, Idxes, sizeof(short) * iIdxNum);
  iBuffer->Unlock();
}
```

Листинг достаточно большой, хотя половину в нем занимает описание координат вершин треугольников, используемых для построения куба, и описание индексов. Давайте по частям рассмотрим весь листинг.

2.10.1. Описание фигуры

Сначала мы объявляем массив с именем `svVortexList` типа `sVertex`. Что такое `sVertex`? Это структура, которая описывает точку и состоит из трех вещест-

венных чисел — координат точки и двойное целое число, описывающее цвет точки:

```
struct sVertex
{
  float x, y, z;
  DWORD color;
};
```

Добавьте описание этой структуры в начало модуля, чтобы не было проблем с компилятором.

Построить куб из точек или отрезков достаточно сложно. Это сколько же примитивов понадобится, чтобы залить каждую поверхность! Единственный вариант, который остается, — использовать закрашенные треугольники. Как же нам строить куб из треугольников? Для этого нужно нарисовать 6 граней, каждая из которых будет состоять из двух треугольников (рис. 2.8).

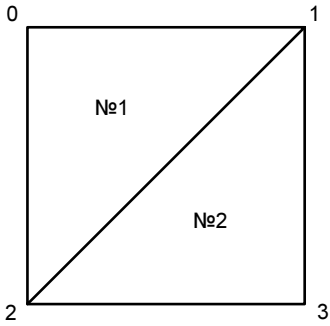

Рис. 2.8. Грань куба из двух треугольников

Получается, что для описания каждой грани необходимо четыре точки. Дабы не сильно мучиться с общими ребрами, будем считать, что все они независимы. В этом случае нам понадобятся 24 точки (6 граней по 4 точки на каждую грань). Вот эти грани мы и создаем. Каждая вершина описывается у нас структурой `sVertex`, а значит, описание точки выглядит примерно следующим образом:

```
-0.5f, -0.5f, -0.5f, 0x00ff0000,
```

Здесь у нас четыре значения: координаты точки X, Y и Z и цвет. Цвет задаем в формате AARRGGBB, где по одному байту выделяется альфа-каналу, красному, зеленому и голубому.

В нашем случае все вершины одной грани куба окрашиваются одним и тем же цветом. Но это не есть обязательно. Каждая вершина треугольника может иметь собственный цвет. В этом случае общий цвет треугольника будет в ви-

де градиента, в котором цвет одной вершины будет плавно переходить в цвет другой. Жаль, что черно-белая полиграфия книги не сможет передать этой красоты, но когда мы закончим пример и увидим реальный прямоугольник, попробуйте поиграть с цветами вершин. Результат может быть очень интересным и возможно, вы воспользуетесь этим приемом для создания какого-либо эффекта.

Чтобы в будущем проще было общаться с массивом, введем константу с именем `iVertsNum` и равную 24.

2.10.2. Буфер вершин

В Direct3D для хранения вершин применяется специальный буфер. В Direct3D для работы с этим буфером используется интерфейс `IDirect3DVertexBuffer9`. Чтобы отобразить фигуру, необходимо заполнить буфер вершинами и указать, какие примитивы будут использоваться и в какой последовательности строить фигуру.

Посмотрим, как создается этот буфер. Для этого используется метод `CreateVertexBuffer`, который выглядит следующим образом:

```
HRESULT CreateVertexBuffer(
    UINT Length,
    DWORD Usage,
    DWORD FVF,
    D3DPOOL Pool,
    IDirect3DVertexBuffer9** ppVertexBuffer,
    HANDLE* pHandle
);
```

Здесь имеются следующие параметры:

- `Length` — длина буфера в байтах. Параметр используется для буферов вершин формата Flexible Vertex Format (FVF, гибкий формат вершин);
- `Usage` — флаги, которые определяют, как будет использоваться буфер. Параметров много и для получения дополнительной информации обратитесь к справочному файлу. Мы используем флаг `D3DUSAGE_WRITEONLY`, который означает, что буфер будет применяться только для записи, т. е. сформировали точки буфера, записали и не читаем из буфера. Это позволит Direct3D оптимизировать память для лучшего использования и формирования сцены;
- `FVF` — по флагам этого параметра Direct3D определяет, как выглядит описание каждой точки. В нашем случае это координаты и цвет. Такого флага нет, но можно указать сразу два флага:

- D3DFVF_XYZ — точка описывается координатами X, Y и Z;
- D3DFVF_DIFFUSE — в описании точки указан рассеянный цвет;

- Pool — флаг, определяющий, где должен быть расположен буфер. Здесь можно указать одно из следующих значений:
  - D3DPOOL_DEFAULT — определяется системой по умолчанию;
  - D3DPOOL_MANAGED — при необходимости происходит автоматическое копирование буфера в память, доступную аппаратной части, и автоматически резервируется в системной памяти, а значит, не сможет произойти потери памяти;
  - D3DPOOL_SYSTEMMEM — использовать системную память;
  - D3DPOOL_SCRATCH — использовать системную память, но при этом, устройство не может обратиться к этой памяти;
  - D3DPOOL_FORCE_DWORD — не используется;
- ppVertexBuffer — указатель на буфер с вершинами;
- pHandle — не используется и должен быть равен нулю.

Теперь посмотрите, с какими параметрами и как мы вызываем этот метод в нашем примере:

```
HRESULT hRes = pD3DDevice->CreateVertexBuffer(
  sizeof(sVertex) * iVertsNum,
  D3DUSAGE_WRITEONLY,
  D3DFVF_XYZ | D3DFVF_DIFFUSE,
  D3DPOOL_DEFAULT,
  &vBuffer,
  0);
```

2.10.3. Работа с буфером вершин

Работа с буфером вершин похожа на прямой доступ к поверхности. Сначала мы блокируем буфер с помощью метода Lock, заполняем буфер значениями вершин и затем разблокируем его. Как и при работе с поверхностью, блокировка буфера должна быть максимально короткой по времени. Метод Lock выглядит следующим образом:

```
HRESULT Lock(
    UINT OffsetToLock,
    UINT SizeToLock,
    VOID **ppbData,
    DWORD Flags
);
```

Здесь имеются следующие четыре параметра:

- `OffsetToLock` — смещение от начала буфера, откуда нужно блокировать данные;
- `SizeToLock` — размер блокируемых данных;
- `**ppbData` — указатель на буфер, где хранятся данные буфера вершин;
- `Flags` — флаги.

Теперь посмотрим, как мы вызываем метод блокировки:

```
hRes = vBuffer->Lock(0, sizeof(sVertex) * iVertsNum, &pBuf, 0);
```

Первый параметр у нас равен нулю, а значит, блокировка будет с самого начала. Второй параметр равен длине буфера (размер одной точки умножается на количество точек). Третий параметр — это указатель на буфер типа `void`, а последний — флаги, которые мы не используем.

Теперь просто копируем значение всех точек в буфер с помощью WinAPI-функции `memcpy`. Сразу после копирования буфер больше не нужен, поэтому освобождаем его с помощью метода `Unlock`.

2.10.4. Буфер индексов вершин

Теперь необходимо указать последовательность точек, из которых будут строиться наши треугольники. Для этого используется буфер индексов. Конечно, мы могли определить каждый треугольник собственными значениями и не мучаться с индексами, а просто заполнить их числами от 0 до количества вершин в треугольниках, но хоть немного же надо сэкономить вершин и упростить работу видеокарте.

В нашем буфере вершин для описания двух треугольников, описывающих одну грань, используется всего четыре вершины. Как это достигается? Посмотрите на рис. 2.9, где описывается построение первой грани. Чтобы построить этот рисунок, необходимо сначала от точки 0 нарисовать отрезок к точке 1, потом от точки 1 к точке 2. После этого Direct3D замыкает точку 2 и точку 0, и полученный треугольник закрашивается указанными цветами.

Далее, строим второй треугольник. Для этого можно использовать уже существующие в буфере точки 1 и 2, и добавляем только точку 3. При этом мы должны указать в буфере индексов, что треугольник должен строиться с помощью линий от точки 2 к точке 1 и затем от точки 1 к точке 3.

Последовательность указания точек при построении треугольника очень важна, потому что по ней определяется, какая сторона будет видима, а какая нет. Закрашивается только видимая сторона, а обратная остается белой. Видимая будет та сторона, точки которой строятся по часовой стрелке. В нашем случае обе стороны построены по часовой стрелке.

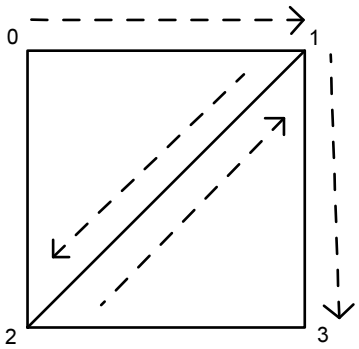

Рис. 2.9. Последовательность построения грани из треугольников

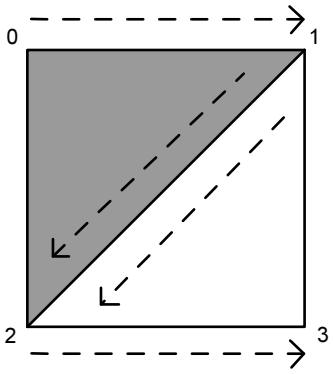

Рис. 2.10. Построение треугольников в разном направлении

Давайте посмотрим на рис. 2.10. Здесь треугольник 1 строится по вершинам в следующей последовательности: 0-1-2. Получается построение по часовой стрелке, а значит, перед нами передняя сторона, которая будет закрашена, поэтому левый верхний треугольник закрашен серым цветом. Второй треугольник строится в последовательности 1-2-3. Это построение выполняется уже против часовой стрелки, а значит, перед нами обратная сторона треугольника, а видимая сейчас сторона треугольника не будет закрашена, а останется белой. В правом нижнем треугольнике будет закрашенной сторона, которая нам сейчас не видна, потому что если развернуть этот треугольник, то с другой стороны построение окажется по часовой стрелке.

Следующий массив чисел определяет индексы, по которым будут строиться треугольники:

```
const unsigned short Idxes[]={
  0,1,2,2,1,3,
  4,5,6,6,5,7,
  8,9,10,10,9,11,
  12,13,14,14,13,15,
  16,17,18,18,17,19,
  20,21,22,22,21,23,
};
```

Для удобства восприятия, я разместил в каждой строке по 6 индексов, которые описывают два треугольника или одну грань.

Фигура куба выбрана не зря, потому что у нее длина всех ребер одинаковая и проще представлять себе в пространстве по числам получаемую фигуру. Попробуйте пройтись хотя бы по первым четырем вершинам и мысленно представить, где она находится в пространстве.

Для хранения индексов применяется специальный индексный буфер. В Direct3D для работы с индексами используется интерфейс `IDirect3DIndexBuffer9`. Работа с ним схожа на работу с буфером вершин. Сначала необходимо создать интерфейс с помощью метода `CreateIndexBuffer` интерфейса `IDirect3DDevice9`. Этот метод выглядит следующим образом:

```
HRESULT CreateIndexBuffer(
    UINT Length,
    DWORD Usage,
    D3DFORMAT Format,
    D3DPOOL Pool,
    IDirect3DIndexBuffer9** ppIndexBuffer,
    HANDLE* pHandle
);
```

Рассмотрим параметры этого метода:

- `Length` — необходимая длина буфера. В нашем случае длина будет равна количеству индексов, умноженному на размер каждого, а размер индекса равен размеру целого числа `short`;
- `Usage` — параметры использования. В данном случае ничего указывать не будем. Поскольку в буфер мы намереваемся только писать, поэтому можно было бы указать параметр `D3DUSAGE_WRITEONLY`, который мы уже видели при рассмотрении создания буфера вершин;
- `Format` — формат индекса или размер каждого элемента массива. В нашем случае индексы описаны числами `short` (16 бит), а значит, в этом параметре нужно указать флаг `D3DFMT_INDEX16`. Если бы у нас было много элементов в массиве, то пришлось бы использовать число большего размера (32-битное `int`) и тогда в этом параметре надо было указывать `D3DFMT_INDEX32`;
- `Pool` — флаг. Возможные значения этого параметра идентичны одноименному параметру метода `CreateVortexBuffer`;
- `ppIndexBuffer` — указатель на переменную, которая будет указывать на созданный интерфейс буфера индексов;
- `pHandle` — не используется и должен быть равен нулю.

Вот так мы создаем буфер индексов:

```
hRes = pD3DDevice->CreateIndexBuffer( sizeof(short) * iIdxNum,
       0, D3DFMT_INDEX16, D3DPOOL_DEFAULT, &iBuffer, 0);
```

Теперь посмотрим, как происходит заполнение индексного буфера. Для этого сначала нужно заблокировать буфер, затем заполнить и освободить бу-

фер индексов. Для блокировки используется метод `Lock` интерфейса `IDirect3DIndexBuffer9`. Параметры этого метода идентичны одноименному из интерфейса `IDirect3DVertexBuffer9`. Логика использования также идентична, поэтому просто посмотрите на код заполнения буфера индексами, и все станет ясно без дополнительных комментариев.

2.10.5. Точка просмотра

Для удобства, наш куб находится в координатах от –0,5 до +0,5 по всем осям, т. е. центр расположен в нулевой точке. Но после инициализации Direct3D, в качестве *матрицы отображения* используется единичная матрица, при которой текущее положение камеры находится также в нулевой точке. Это значит, что мы будем смотреть на мир из центра куба и ничего, кроме невидимых сторон этого куба, не увидим, а невидимые стороны белые, поэтому и сцена окажется пустой.

Чтобы увидеть куб, можно подкорректировать координаты куба, но для первого примера лучше не усложнять координаты, поэтому мы изменим положение камеры, точки, из которой и куда мы будем смотреть на мир. Вот тут возникают проблемы у тех, кто не знаком с высшей математикой и матрицами, поэтому придется потратить пару минут на дополнительные пояснения.

Что такое матрица? Нам достаточно знать, что это двумерный массив чисел. В Direct3D используется матрица размером 4×4, поэтому массив будет выглядеть следующим образом:

```
A11  A12  A13  A14
A21  A22  A23  A24
A31  A32  A33  A34
A41  A42  A43  A44
```

В Direct3D эта матрица объявлена как `D3DMATRIX` и имеет следующий вид:

```
typedef struct _D3DMATRIX {
    union {
        struct {
            float        _11, _12, _13, _14;
            float        _21, _22, _23, _24;
            float        _31, _32, _33, _34;
            float        _41, _42, _43, _44;

        };
        float m[4][4];
    };
} D3DMATRIX;
```

Каждый элемент этой матрицы — это число, которое имеет определенный смысл. Мы пока будем работать только с числами A41, A42 и A43 — это X-, Y- и Z-координаты точки расположения камеры. Значения остальных параметров должны быть равны единичной матрице:

```
1  0  0  0
0  1  0  0
0  0  1  0
X  Y  Z  0
```

Параметры камеры мы уже задали во время инициализации Direct3D с помощью функции `SetTransform`. Тогда первый параметр был равен `D3DTS_PROJECTION`, а теперь с помощью этой же функции мы должны задать координаты расположения камеры, и тут первый параметр должен быть равен `D3DTS_VIEW`, а второй — это устанавливаемая матрица.

Итак, после формирования куба вызываем функцию `SetView`, которая будет отодвигать камеру по оси Z на 5 единиц, т. е. элемент матрицы A43 будет равен 5. Вот как эта функция выглядит в виде кода:

```
void SetView()
{
  D3DMATRIX View = {
    1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0,
    0, 0, 5, 1,
  };
  pD3DDevice->SetTransform(D3DTS_VIEW, &View);
}
```

Вот теперь мы сможем увидеть наш куб, поэтому можно переходить к его отображению.

Сделаем небольшое отступление и немного глубже затронем тему матрицы. Нет, слишком глубоко уходить не будем, но посмотрим еще, как можно поворачивать сцену. Если необходимо повернуть сцену на угол b по часовой стрелке вокруг оси Z, то нужно установить следующую матрицу:

```
D3DMATRIX View = {
  cos(b), -sin(b), 0, 0,
  sin(b),  cos(b), 0, 0,
  0, 0, 1, 0,
  X, Y, Z, 1,
};
```

Обратите внимание, что в одной матрице мы устанавливаем угол поворота и положение объекта. Если необходимо произвести поворот против часовой стрелки, то возьмите отрицательное значение угла.

Третья строка осталась единичной, но она отвечает за Z-координату и вокруг этой оси тоже можно крутить объект. Просто пока опустим эту строчку, потому что в двумерном представлении жить проще.

Да, для поворота приходится выполнять тригонометрические функции, но это того стоит. Особенно если необходимо не только произвести разворот, но и масштабировать объект. Допустим, что по оси X надо масштабировать сцену или объект на значение SX, а по оси Y на значение SY. В этом случае матрица преобразований будет выглядеть следующим образом:

```
D3DMATRIX View = {
  SX*cos(b), SX*-sin(b), 0, 0,
  SY*sin(b), SY*cos(b), 0, 0,
  0, 0, 1, 0,
  X, Y, Z, 1,
};
```

Если же нужно повернуть сцену вокруг оси X, то используем матрицу следующего вида:

```
D3DMATRIX View = {
  1, 0, 0, 0,
  0, cos(b), -sin(b), 0,
  0, sin(b),  cos(b), 0,
  X, Y, Z, 1,
};
```

Для поворота вокруг оси Y используем матрицу:

```
D3DMATRIX View = {
  cos(b), 0, -sin(b), 0,
  0, 1, 0, 0,
  sin(b),  0, cos(b), 0,
  X, Y, Z, 1,
};
```

Если необходимо просто масштабировать сцену, то используем следующую матрицу:

```
D3DMATRIX View = {
  SX, 0, 0, 0,
  0, SY, 0, 0,
```

```
  0, 0, SZ, 0,
  X, Y, Z, 1,
};
```

Эти матрицы преобразований получаются из математических операций над матрицами. Я бы с удовольствием дал пару уроков математики для тех, кто не знаком с этой темой, но лучше не тратить на это время. Достаточно просто знать матрицы, которые я привел, и этого будет достаточно.

2.10.6. Отображение

Как и в случае с DirectDraw, рисование будет происходить по событию `WM_PAINT`. Код отображения можно увидеть в листинге 2.15.

Листинг 2.15. Код отображения сцены

```
case WM_PAINT:
  // Очистка поверхности
  RECT  r;
  SetRect(&r, 0, 0, iWidth, iHeight);
  pD3DDevice->Clear(1, (D3DRECT*)&r, D3DCLEAR_TARGET,
    D3DCOLOR_XRGB(255,255,255), 1.0, 0);

  // Подготовка изображения в заднем буфере
  pD3DDevice->BeginScene();
  DrawScene();
  pD3DDevice->EndScene();

  // Отображение
  pD3DDevice->Present(NULL, NULL, 0, NULL);
  break;
```

В самом начале мы очищаем содержимое буфера, закрашивая его белым цветом. Затем вызывается метод `BeginScene` интерфейса `IDirect3DDevice9`. Этот метод всегда должен вызываться до формирования сцены, а после завершения создания сцены необходимо вызвать метод `EndScene`. После вызова метода `EndScene` мы отображаем сформированную сцену с помощью метода `Present`.

2.10.7. Вывод буфера вершин

Теперь необходимо познакомиться с функцией `DrawScene`. Ее можно увидеть в листинге 2.16.

Листинг 2.16. Функция `DrawScene`

```
void DrawScene()
{
  // Выставляем матрицу для объекта в мировых координатах
  ViewAngle+=0.01;
  float b = ViewAngle++;

  // Определение матрицы мира
  D3DMATRIX World = {
    cos(b)*cos(b), cos(b)*sin(b), sin(b), 0,
    -sin(b), cos(b), 0, 0,
    -sin(b)*cos(b), -sin(b)*sin(b), cos(b), 0,
    0, 0,  0, 1,
  };

  // Устанавливаем матрицу мира
  pD3DDevice->SetTransform(D3DTS_WORLD, &World);

  // Определяем вершинный формат
  pD3DDevice->SetFVF( D3DFVF_XYZ | D3DFVF_DIFFUSE);

  // Устанавливаем вершинный буфер
  pD3DDevice->SetStreamSource( 0, vBuffer, 0, sizeof(sVertex) );

  // Устанавливаем индексный буфер
  pD3DDevice->SetIndices(iBuffer);

  // Вывод объекта
  pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
     0, 0, iVertsNum, 0, iPointsNum);
}
```

В самом начале рассчитываем число, от которого будет определяться угол поворота, на который нужно повернуть куб. Для этого введем переменную `ViewAngle`. Эта переменная будет автоматически увеличиваться после каждого обновления экрана. Результат расчетов заносится в переменную `b`, используемую для хранения угла, на который нужно повернуть куб.

Чтобы повернуть куб, применим *мировую матрицу* (World Matrix). Мы уже познакомились с матрицами проекции и отображения, а теперь давайте познакомимся с мировой. Она также имеет размеры 4×4 и схожа с матрицей отображения, только влияет не на точку, из которой мы смотрим на мир, а на сам виртуальный мир.

```
D3DMATRIX World = {
  cos(b)*cos(b), cos(b)*sin(b), sin(b), 0,
  -sin(b), cos(b), 0, 0,
  -sin(b)*cos(b), -sin(b)*sin(b), cos(b), 0,
  0, 0,  0, 1,
};
```

Для установки матрицы используется уже знакомый нам метод `SetTransform`, только первый параметр должен быть равен `D3DTS_WORLD`, а второй — указатель на устанавливаемую матрицу мира.

Теперь у нас все готово для отображения созданного куба на устройстве Direct3D. Для этого нужно указать нашему интерфейсу `IDirect3DDevice9`, что необходимо использовать созданный нами ранее буфер вершин и индексов. Для этого сначала с помощью метода `SetFVF` задаем формат описания вершин в нашем буфере. Этой функции нужно передать в качестве параметра флаги, которые определят формат вершины. У нас вершины состоят из координат X, Y и Z и рассеянного цвета, поэтому в качестве параметра функции нужно указать два флага: `D3DFVF_XYZ` и `D3DFVF_DIFFUSE`.

Далее, указываем вершинный буфер с помощью метода `SetStreamSource`. Этот метод выглядит следующим образом:

```
HRESULT SetStreamSource(
    UINT StreamNumber,
    IDirect3DVertexBuffer9 *pStreamData,
    UINT OffsetInBytes,
    UINT Stride
);
```

Здесь имеются следующие четыре параметра:

- `StreamNumber` — номер потока данных, который может изменяться от 0 до максимального количества потоков минус 1;
- `pStreamData` — указатель на буфер вершин;
- `OffsetInBytes` — смещение в байтах от начала в буфере вершин, начиная с которого необходимо начать копирование;
- `Stride` — шаг, или размер вершины.

В нашем случае вызов выглядит следующим образом:

```
pD3DDevice->SetStreamSource(0, vBuffer, 0, sizeof(sVertex));
```

После этого указываем интерфейсу `IDirect3DDevice9` буфер индексов с помощью метода `SetIndices`. У этого метода только один параметр — указатель на буфер индексов.

Установив все необходимые буферы, нужно их прорисовать с помощью метода `DrawIndexedPrimitive`. Этот метод выглядит следующим образом:

```
HRESULT DrawIndexedPrimitive(
    D3DPRIMITIVETYPE Type,
    INT BaseVertexIndex,
    UINT MinIndex,
    UINT NumVertices,
    UINT StartIndex,
    UINT PrimitiveCount
);
```

Здесь имеются следующие параметры:

- `Type` — тип примитивов, которые нужно использовать. В данном случае можно указать одно из перечисленных значений:
  - `D3DPT_POINTLIST` — в буфере находятся координаты точек. Каждая координата — это отдельная точка;
  - `D3DPT_LINELIST` — в буфере находятся координаты линий. Каждая пара координат в буфере образует линию;
  - `D3DPT_LINESTRIP` — в буфере находятся координаты связанных линий, при этом конец первой линии является началом следующей;
  - `D3DPT_TRIANGLELIST` — в буфере находятся координаты треугольников. Каждые три точки определяют треугольник;
  - `D3DPT_TRIANGLESTRIP` — в буфере находятся связанные треугольники, т. е. последний отрезок первого треугольника является началом следующего;
  - `D3DPT_TRIANGLEFAN` — треугольники формируются на основе смежных сторон;
- `BaseVertexIndex` — смещение от начала буфера, где находится первый индекс;
- `MinIndex` — минимальный индекс точки в буфере, который должен отображаться;
- `NumVertices` — количество вершин;
- `StartIndex` — расположение индекса в буфере индексов, начиная с которого нужно начинать чтение;
- `PrimitiveCount` — количество примитивов, которые нужно создать. В нашем случае будет 12 треугольников, поэтому желательно создать константу, которую легко можно будет редактировать, в случае увеличения или уменьшения количества примитивов.

В параметре `Type` мы видим, что каждый тип используемого примитива должен содержать в буфере данные в определенном порядке. Например, для примитива `D3DPT_TRIANGLELIST` каждые три точки в буфере образуют треугольник. Но в нашем буфере каждый следующий треугольник может использовать точки предыдущего треугольника, как это происходит при применении типа `D3DPT_TRIANGLESTRIP`. Это достигается за счет использования индекса. Если бы индекс был пустой, то каждый следующий треугольник `D3DPT_TRIANGLELIST` не смог бы использовать координаты предыдущего.

На этом пример закончен и его можно запускать для просмотра результата. На рис. 2.11 можно увидеть результат работы. Надеюсь, что полиграфия позволит увидеть куб.

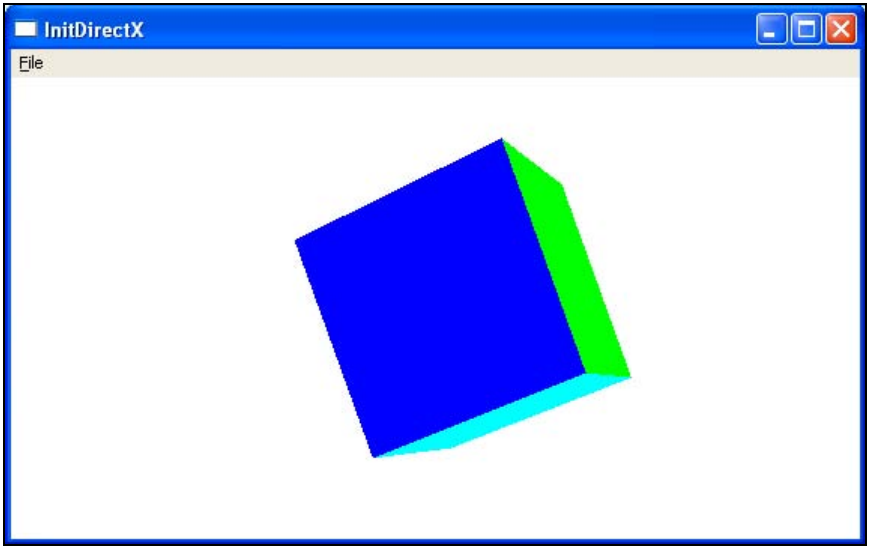

Рис. 2.11. Построение треугольников в разном направлении

Чтобы увидеть, как треугольники образуют наш куб, можно включить отображение куба в проволочном виде, когда треугольники не закрашиваются. Для этого добавьте следующую строку кода перед вызовом метода `DrawIndexedPrimitive`:

```
pD3DDevice->SetRenderState(D3DRS_FILLMODE, D3DFILL_WIREFRAME);
```

В результате на экране будет построен прямоугольник, показанный на рис. 2.12.

Примечание

Исходный код примера находится в каталоге \Chapter2\D3Dtest на прилагаемом компакт-диске.

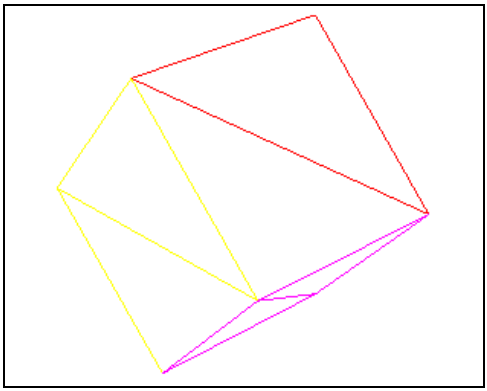

Рис. 2.12. Построение треугольников в разном направлении

2.11. Mesh

Создавать поверхность из вершин достаточно накладно, и чем больше точек используется в сцене, тем больше ресурсов компьютера необходимо для ее формирования. К тому же, создавать формы посредством точек достаточно сложно и утомительно без удобных средств.

Как же тогда создаются 3D-игры с высокой детализацией? В этом нам помогают графические пакеты типа 3D Studio MAX, с помощью которого удобно создавать 3D-модели. Полученную модель можно сохранить в файле с расширением X. Этот файл является родным для DirectX и может быть легко загружен с помощью одной только функции в объект mesh, с которым достаточно удобно работать, а скорость отображения остается достаточно высокой. Конечно, чем сложнее mesh-объект, тем больше ресурсов понадобится для отображения, поэтому не стоит пренебрегать вершинами во время создания объекта в графическом пакете.

Что же такое mesh? В переводе на русский язык (я думаю, что для многих читателей этот язык родной) это слово означает сеть. Действительно, объекты, которые мы будем использовать, выглядят как сеть, и это вы увидите на реальном примере. Конечно же, для построения самой сетки будут использоваться в качестве базового примитива треугольники, но на это можно закрыть глаза.

Для примера с помощью 3D Studio MAX я создал нехитрую фигуру, чем-то напоминающую тарелку НЛО (рис. 2.13), и сохранил в файле с расширением X. Именно ее мы будем загружать и отображать в нашем окне.

Давайте познакомимся с mesh-объектами и научимся с ними работать. Помимо этого, мы познакомимся с таким понятием, как освещение, без которого невозможно увидеть реальный цвет объекта.

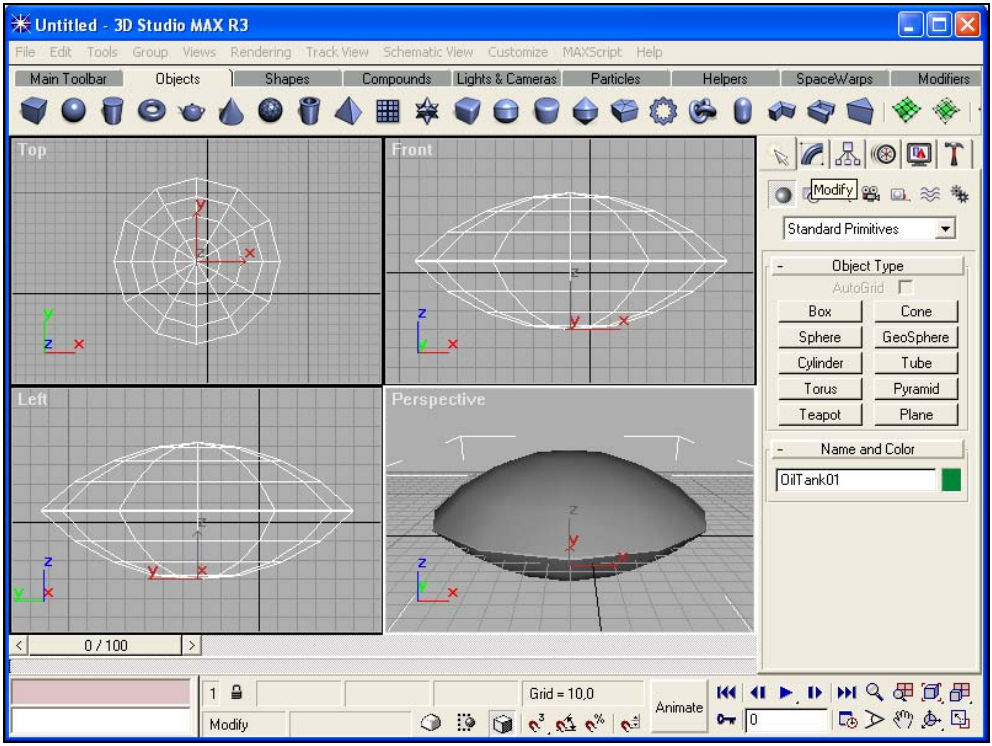

Рис. 2.13. Тарелка НЛО, которую мы будем загружать

Итак, создадим новое приложение и добавим вызов функции инициализации Direct3D. Сразу же загрузим наш объект. Конечно, для загрузки X-файла достаточно только функции `D3DXLoadMeshFromX`, но помимо объекта в файле может быть еще и информация о материалах и о текстурах, которые также нужно отделить от объекта для дальнейшего использования, поэтому все не так уж и просто.

2.11.1. Загрузка X-файла

Для упрощения задачи я создал функцию `LoadMash` в модуле `dxfunc`, чтобы вы могли ее использовать в будущем в своих собственных проектах. Код функции показан в листинге 2.17.

Листинг 2.17. Код функции загрузки X-файла в mesh

```
DWORD LoadMesh (char *filename,
    IDirect3DDevice9 *ppiD3DDevice9,
    ID3DXMesh **ppMesh,
```

```
    LPDIRECT3DTEXTURE9 **pMeshTextures,
    char *texturefilename,
    D3DMATERIAL9 **pMeshMaterials)
{
 LPD3DXBUFFER pD3DXMtrlBuffer;
 DWORD dwNumMaterials;

 D3DXLoadMeshFromX(filename, D3DXMESH_SYSTEMMEM, ppiD3DDevice9,
       NULL, &pD3DXMtrlBuffer, NULL, &dwNumMaterials, ppMesh);

 D3DXMATERIAL* d3dxMaterials =
          (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();

 // Инициализируем материалы и текстуры
(*pMeshTextures) = new LPDIRECT3DTEXTURE9[dwNumMaterials];
(*pMeshMaterials) = new D3DMATERIAL9[dwNumMaterials];

 for( DWORD i=0; i<dwNumMaterials; i++ )
 {
  // Копируем материал
  (*pMeshMaterials)[i] = d3dxMaterials[i].MatD3D;

  // Создаем текстуру
  if( FAILED(D3DXCreateTextureFromFile(ppiD3DDevice9,
         d3dxMaterials[i].pTextureFilename, &(*pMeshTextures)[i])))
    if( FAILED(D3DXCreateTextureFromFile(ppiD3DDevice9,
           texturefilename, &(*pMeshTextures)[i])))
     (*pMeshTextures)[i] = NULL;
 }
 return dwNumMaterials;
}
```

Давайте посмотрим, что у нас здесь происходит. В функцию передаются следующие параметры:

- `filename` — имя загружаемого X-файла;
- `ppiD3DDevice9` — указатель на интерфейс `IDirect3DDevice9`;
- `ppMesh` — переменная типа `ID3DXMesh`. Этот интерфейс используется для хранения и работы с mesh-поверхностями;
- `pMeshTextures` — указатель на указатель интерфейса `IDirect3DTexture9`. Этот интерфейс используется для хранения текстур;
- `texturefilename` — указатель на имя файла текстуры по умолчанию. Имена текстур могут храниться прямо в X-файле, но если там нет текстуры, то

будем использовать файл из этого параметра. Если уже и здесь текстуры не будет, тогда не будем обтягивать объект текстурой;

- `pMeshMaterials` — структура типа `D3DMATERIAL9`, которая определяет материал объекта.

В качестве результата функция будет возвращать количество загруженных объектов. Дело в том, что объектов может быть несколько, ведь фигуру человека слишком сложно создать из одной сетки. Вполне логично сделать каждую часть тела из отдельной сетки, и все они будут храниться в одном X-файле.

Давайте сразу же посмотрим на структуру `D3DMATERIAL9`, потому что она пригодится нам в будущем:

```
typedef struct _D3DMATERIAL9 {
   D3DCOLORVALUE Diffuse;  // рассеянный цвет
   D3DCOLORVALUE Ambient;  // внешняя среда
   D3DCOLORVALUE Specular; // цвет отражения
   D3DCOLORVALUE Emissive; // цвет излучения
   float Power;            // число, определяющее резкость отражения
} D3DMATERIAL9;
```

Параметры структуры не сложные, поэтому вы легко сможете разобраться с ними по комментариям.

Возвращаемся к коду функции загрузки X-файла. Сначала мы объявляем две переменные:

- `pD3DXMtrlBuffer` типа `LPD3DXBUFFER` — это буфер, в котором будут храниться используемые в файле материалы;
- `dwNumMaterials` типа `DWORD` для хранения количества загруженных сеток.

Теперь производим непосредственно загрузку с помощью функции `D3DXLoadMeshFromX`. Эта функция выглядит следующим образом:

```
HRESULT D3DXLoadMeshFromX(
    LPCTSTR pFilename,
    DWORD Options,
    LPDIRECT3DDEVICE9 pDevice,
    LPD3DXBUFFER* ppAdjacency,
    LPD3DXBUFFER* ppMaterials,
    LPD3DXBUFFER* ppEffectInstances,
    DWORD* pNumMaterials,
    LPD3DXMESH* ppMesh
);
```

Посмотрим параметры функции:

- `pFilename` — имя загружаемого файла;
- `Options` — опции, которые должны использоваться при загрузке. Опций достаточно много, но мы пока будем применять только `D3DXMESH_SYSTEMMEM`, которая означает, что загрузка происходит в системную память;
- `pDevice` — указатель на интерфейс `IDirect3DDevice9`;
- `ppAdjacency` — указатель на буфер для смежных граней;
- `ppMaterials` — указатель на буфер для материалов;
- `ppEffectInstances` — указатель на буфер для экземпляров эффектов;
- `pNumMaterials` — через этот параметр будет получено количество загруженных материалов;
- `ppMesh` — непосредственно указатель на объект сетки.

2.11.2. Материалы и текстуры

Загрузив данные, нужно разобраться с материалами и при необходимости загрузить текстуры, которые сейчас находятся в буфере `pD3DXMtrlBuffer`. Сначала определим указатель на данные буфера с помощью метода `GetBufferPointer` (`pD3DXMtrlBuffer->GetBufferPointer()`). По этому указателю хранятся данные типа `D3DXMATERIAL`, поэтому введем переменную такого типа и сохраним в ней полученный указатель:

```
D3DXMATERIAL* d3dxMaterials =
    (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
```

Теперь мы знаем количество материалов, поэтому инициализируем массивы для их хранения:

```
(*pMeshMaterials) = new D3DMATERIAL9[dwNumMaterials];
(*pMeshTextures) = new LPDIRECT3DTEXTURE9[dwNumMaterials];
```

Первая строка выделяет память для хранения `dwNumMaterials`-материалов, а вторая строка для хранения такого же количества текстур. Это не совсем этичное решение, которое может привести к сверхрасходу памяти, но иногда им можно воспользоваться. Если ваша фигура состоит из 10 сеток и ни одна из них не использует текстуры, то выделенная для хранения текстур память будет потеряна для общества.

Теперь запускаем цикл, который будет перебирать все материалы от 0 до значения `dwNumMaterials`. В цикле мы сначала копируем материал в i-й элемент массива `pMeshMaterials`.

Затем нужно загрузить картинку, используемую в качестве текстуры. Имя файла текстуры находится в свойстве `pTextureFilename` материала `d3dxMaterials[i]`. Для загрузки картинки применим функцию `D3DXCreateTextureFromFile`. Мы ее еще не использовали, поэтому посмотрим, как она выглядит:

```
HRESULT D3DXCreateTextureFromFile(
    LPDIRECT3DDEVICE9 pDevice,
    LPCTSTR pSrcFile,
    LPDIRECT3DTEXTURE9 *ppTexture
);
```

Здесь имеются следующие три параметра:

- `pDevice` — указатель на интерфейс `IDirect3DDevice9`, с которым должна быть связана текстура;
- `pSrcFile` — имя файла;
- `ppTexture` — указатель на интерфейс типа `IDirect3DTexture9`, куда будет загружена текстура.

Если во время загрузки произошла ошибка, то пытаемся загрузить файл, который был указан в качестве параметра функции. Если же и тут произошла ошибка, то текстуры не будет.

2.11.3. Точка осмотра сцены

Теперь у нас проинициализирован Direct3D и загружен необходимый объект. Все эти действия можно делать после создания, но до отображения окна. Но не будем торопиться переходить к отображению, а отодвинем камеру немного назад и настроим освещение, без которого материалы будут отображаться черным цветом.

Я создал довольно большой объект, поэтому приходится отодвигать камеру достаточно далеко, аж на 300 единиц.

```
D3DMATRIX View = {
    1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0,
    0, 0, 300, 1,
};
pD3DDevice->SetTransform(D3DTS_VIEW, &View);
```

2.11.4. Освещение

И последнее, что мы сделаем перед началом отображения, — настроим освещение. С освещением мы также не работали, поэтому рассмотрим эту тему. Следующий код показывает действия, которые будем выполнять в нашей программе:

```
D3DLIGHT9 light;
ZeroMemory(&light, sizeof(D3DLIGHT9));
light.Type = D3DLIGHT_DIRECTIONAL;
light.Direction = D3DXVECTOR3(0.5f, 0.0f, 0.5f);
light.Diffuse.r = light.Diffuse.g = light.Diffuse.b =
    light.Diffuse.a = 1.0f;
pD3DDevice->SetLight(0, &light);
pD3DDevice->LightEnable(0, TRUE);
```

Давайте посмотрим, что тут происходит. Для работы с цветом используется структура `D3DLIGHT9`, поэтому мы объявляем переменную `light` этого типа. Всю структуру рассматривать нет смысла, потому что у нее аж 13 параметров — не счастливое число, хотя я и не суеверный. Лучше рассмотрим только то, что нам понадобится, и что будет использоваться в данном примере.

Перед использованием всех структур мы обнуляем их содержимое. Освещение не является исключением, поэтому обнулим и его, чтобы незаполненные поля были равны нулю и применялись значения по умолчанию.

В параметре `Type` указывается тип света. Мы будем использовать направленный свет, поэтому в данном параметре укажем флаг `D3DLIGHT_DIRECTIONAL`. Помимо этого можно применить следующие типы освещения:

- `D3DLIGHT_POINT` — точечный;
- `D3DLIGHT_SPOT` — прожектор.

После этого устанавливаем направление света в свойстве `Direction`. В качестве вектора будет результат `D3DXVECTOR3`, которому в виде параметра передаются координаты направления вектора. Следующим этапом устанавливаем цвет в свойстве `Diffuse`. Цвет освещения будет белый, поэтому во все составляющие цвета записываем значение 1.

Освещение готово и его можно установить интерфейсу `IDirect3DDevice9` с помощью функции `SetLight`. Этому методу передается два параметра: индекс лампочки и указатель на структуру `D3DLIGHT9`, которая содержит данные об устанавливаемом свете. Каждая новая лампочка должна иметь свой индекс. Если указать индекс уже существующей лампочки, то старые данные будут заменены новыми.

Новое освещение создано и связано с устройством, и чтобы его включить, используем метод LightEnable. Этому методу нужно также указать два параметра: индекс лампочки, состояние которой мы хотим изменить, и новое значение состояния (TRUE, чтобы включить лампочку, и FALSE, чтобы выключить).

2.11.5. Отображение сетки

Отображение будет происходить по событию WM_PAINT. Код здесь такой же, как и в примере из *разд. 2.10*, т. е. очистка экрана, начало сцены рисования, вызов функции DrawScene, завершение сцены, отображение. Все это мы уже рассматривали, поэтому сэкономим место в книге, а разберем функцию DrawScene, которая полностью переработана для отображения сетки. Эта функция показана в листинге 2.18.

Листинг 2.18. Функция отображения сетки

```
void DrawScene()
{
 // Определяем матрицу мира для вращения сцены
 ViewAngle+=0.01f;
 float b = ViewAngle;

 D3DMATRIX World = {
   cos(b)*cos(b), cos(b)*sin(b), sin(b), 0,
   -sin(b), cos(b), 0, 0,
   -sin(b)*cos(b), -sin(b)*sin(b), cos(b), 0,
   0, 0,  0, 1,
 };
 pD3DDevice->SetTransform(D3DTS_WORLD, &World);

 // Уберите комментарий со следующей строки,
 // чтобы увидеть сетку в проволочном виде
 // pD3DDevice->SetRenderState(D3DRS_FILLMODE, D3DFILL_WIREFRAME);

 // Включить освещение
 pD3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE);

 // Назначить каждому элементу материал и отобразить сетку
 for (DWORD i=0; i<dwNumMaterials; i++)
 {
   pD3DDevice->SetMaterial(&pMeshMaterials[i]);
```

```
    if (pMeshTextures[i])
      pD3DDevice->SetTexture(0, pMeshTextures[i]);

    pMesh->DrawSubset(i);
  }
}
```

Этот пример будет вращать сцену, поэтому вначале, как и в примере из *разд. 2.10*, изменяем параметры мировых координат для вращения сцены. После этого разрешаем использовать освещение. Действительно, мы добавили лампы, включили их, но без особого разрешения освещение не будет работать. Чтобы Direct3D начал использовать лампы, нужно вызвать метод `SetRenderState`, указав в качестве первого параметра константу `D3DRS_LIGHTING`, а в качестве второго — значение `TRUE`.

Теперь запускаем цикл перебора сеток. В этом цикле с помощью методов `SetMaterial` и `SetTexture` устанавливаем интерфейсу `IDirect3DDevice9`

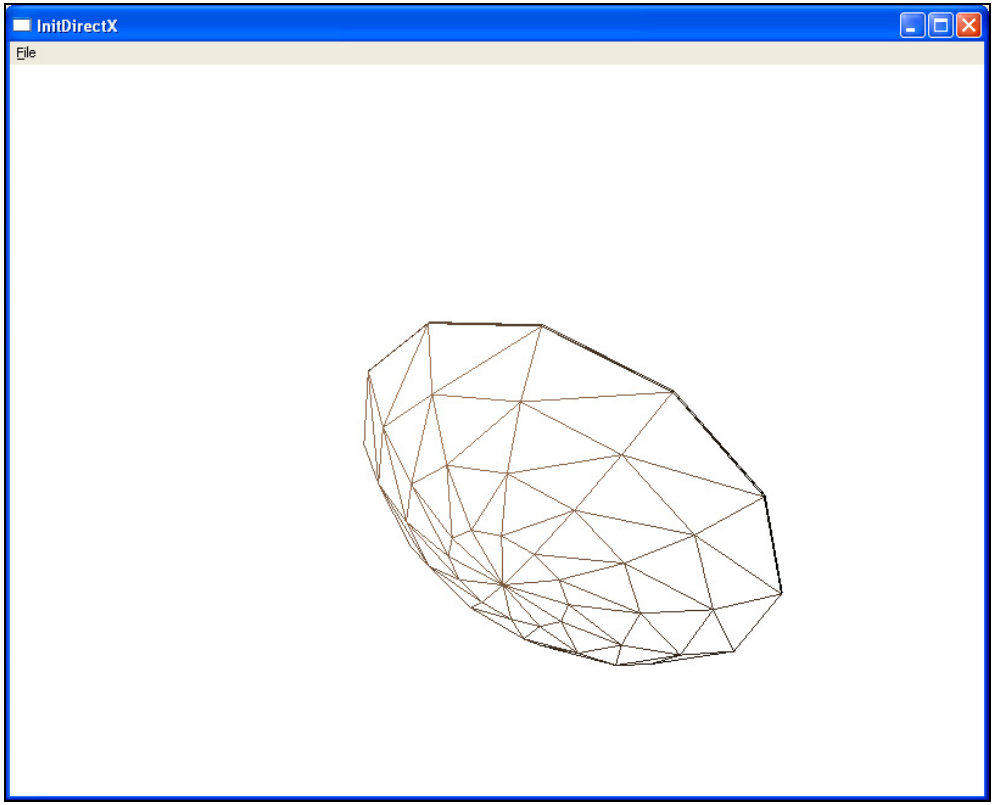

Рис. 2.14. Отображение объекта в виде сетки

материал и текстуру. Когда текстуры готовы, отображаем сетку с помощью метода `DrawSubset`, которому в качестве параметра нужно передать индекс отображаемой сетки.

Можете запустить программу и посмотреть на результат работы. В функции `DrawScene` есть строка кода, которая устанавливает просмотр сцены в проволочном виде. Включите его и посмотрите на сетку в проволочном виде. Сравните результат с фигурой, которая отображалась в 3D Studio MAX на рис. 2.13. В 3D Studio MAX сетка создана из прямоугольников, но в Direct3D такого примитива нет, поэтому здесь сетка создана из треугольников (рис. 2.14), т. е. каждый прямоугольник разбит на два треугольника.

Примечание

Исходный код примера находится в каталоге \Chapter2\D3Dmesh на прилагаемом компакт-диске.

2.12. Синхронизация

Теперь мы рассмотрели все необходимые основы и пора бы перейти к написанию реальных примеров эффектов. Но нам еще необходимо рассмотреть один интересный вопрос — синхронизацию. Дело в том, что демо-ролик это не только видео, но и звук и обе эти составляющие должны работать синхронно. Никому не понравиться, если графика будет опережать видео или наоборот.

Несмотря на то, что звук мы не анализируем, о синхронизации забыть нельзя. Если просто выводить графику, не задумываясь о синхронизации, то на вашем компьютере все может выглядеть вполне нормально. А вот если запустить ролик на другом компьютере, который будет работать быстрее и чаще генерировать и выводить кадры, то видео начнет опережать и гармония потеряется. Необходимо явно что-то делать.

Привязаться к музыке достаточно сложно, ведь ее воспроизведение идет параллельно и трудно определить, какой отрывок сейчас воспроизводится. Можно поставить на звуковом потоке определенные метки, но это решение подойдет только для демо-ролика и не решит проблему полностью, потому что задержки все равно будут, или просто нужно будет создать слишком много меток.

2.12.1. Синхронизация задержками

Более качественный вариант — привязаться ко времени. Для этого введем целочисленную глобальную переменную с именем `LastTickCount`:

```
int LastTickCount;
```

В этой переменной будет храниться время последнего отображения графики. После запуска и инициализации программы сохраняем в этой переменной текущее время с помощью WinAPI-функции `GetTickCount`:

```
LastTickCount = GetTickCount();
```

Эта функция возвращает количество миллисекунд, прошедших с момента старта ОС. Теперь, во время отображения графики, необходимо проверить, сколько времени прошло с момента отображения последнего кадра. Например, если у вас за отображение отвечает функция `DrawScene`, то код будет выглядеть следующим образом:

```
void DrawScene()
{
  int ThisTickCount = GetTickCount();

  if (ThisTickCount-LastTickCount>5)
  {
   // Отображение графики

   LastTickCount = ThisTickCount;
  }
}
```

В самом начале функции определяем текущее время. Затем проверяем, если с момента последнего отображения прошло больше 5 мс, то необходимо сформировать новый кадр, иначе ничего делать не нужно.

После формирования кадра сохраняем в переменной `LastTickCount` значение времени, которое мы получили в самом начале функции `DrawScene`.

Вместо функции `GetTickCount` можно использовать и `timeGetTime`, которая выполняет ту же задачу.

2.12.2. Синхронизация временем

Синхронизация задержками позволяет затормозить графику, если компьютер обрабатывает видеоданные быстрее, чем звук. А что если компьютер медленнее? В этом случае произойдет обратный эффект — звук начнет опережать видеопоток. Что делать в этом случае? Заставить пользователя понизить разрешение? Хороший выход, но и этого может не хватить. Можно написать большими буквами системные требования, которые будут не ниже вашего компьютера, но это тоже не выход. Однако менее производительный компьютер сможет сформировать видеопоток чуть медленнее и при этом потеряет всего пару кадров в секунду, что не сильно повлияет на результат.

Вопрос остается в том, как выкинуть лишние кадры, которые не успевают сформироваться за указанный промежуток времени, например, за время звучания определенного музыкального отрывка, чтобы не потерять гармонию. Тут необходимо немного модифицировать алгоритм и привязать формирование сцены не к частоте кадров, а ко времени.

В некоторых случаях, создавая анимацию, мы не будем привязываться ко времени, а просто сформируем сцену в момент безделья программы. Таким образом, синхронизация будет отсутствовать, потому что нет звука и не с чем синхронизировать.

Как же работает привязка анимации ко времени? Допустим, что в деморолике есть определенный отрывок музыки длиной в 4 сек, за время звучания которого нужно переместить изображение самолета из одного угла в другой. Алгоритм отображения будет следующим:

1. В начале звучания отрывка запоминаем текущее время.
2. На каждом этапе цикла определяем текущее время, вычитаем из него начальное время и получаем время, прошедшее с начала запуска анимации.
3. Если прошло более 4 сек, то отображать что-то уже поздно, если меньше, то отображаем объект. Положение объекта определяется по формуле:

```
положение = start_point + (end_point - start_point) *
    ((t-t_start)/t_length)
```

Необходимо небольшое пояснение:

- `start_point` — точка, из которой должно начинаться движение;
- `end_point` — точка, в которую происходит движение;
- `t` — текущее время;
- `t_start` — время, в которое движение начато;
- `t_length` — продолжительность движения.

Теперь посмотрим на формулу по частям, причем с самого конца:

```
((t-t_start)/t_length)
```

Здесь определяется, сколько времени прошло. Из текущего времени отнимается начальное время и делится на продолжительность движения. Например, движение начато в 10 сек с момента старта ОС, сейчас прошло 12, а движение должно происходить 4 сек:

$$((12 - 10) / 4) = 0.5$$

Получается, что прошло половина времени, а значит, объект должен находиться посередине. Давайте рассмотрим формулу до конца, и убедимся, что она покажет на середину.

Двигаемся дальше. Для упрощения жизни смотрим на движение в одной плоскости. Допустим, что начальное положение объекта равно 10 точек, а конечное — 20. В этом случае формула будет выглядеть так:

$$10 + (20 - 10)*((12 - 10) / 4) = 15$$

Как видите, формула показала, что если прошла половина необходимого времени, то объект должен находиться посередине.

Если же привязать анимацию ко времени, то она будет корректно отображаться на любом компьютере. Только если компьютер быстрый, то движение будет происходить максимально плавным, а на медленном компьютере будут заметны рывки и явные пропадания кадров из анимации. В любом случае, какой бы ни был компьютер, самолет пройдет необходимое расстояние ровно за 4 сек.

В данном случае мы рассмотрели формулу, измеряя время в секундах, но не забывайте, что функция `GetTickCount` возвращает время в миллисекундах.

2.12.3. Пример синхронизации временем 2D-графики

Давайте рассмотрим синхронизацию временем на примере, чтобы увидеть, как это происходит в виде кода. Для начала создаем проект и добавляем инициализацию DirectDraw, как мы это делали уже не раз.

Для реализации примера нам понадобится одна переменная и три константы:

```
double tStartAnimTime;
const int AnimTime = 4000;
const int StartPos = 20;
const int EndPos = 600;
```

В переменной `tStartAnimTime` будет храниться время начала анимации, от которого мы будем отталкиваться. Константа `AnimTime` определяет время, за которое изображение самолета должно переместиться из левого угла в правый. Не забываем, что время задается в миллисекундах, а значит, если нам требуется 4 сек, то значение времени нужно умножить на 1000. Константы `StartPos` и `EndPos` определяют левую и правую позиции самолета, т. е. откуда и докуда нужно долететь.

Теперь осталось только долететь. Лишь бы только не вылететь в трубу, а именно действительно долететь. Сначала модифицируем главный цикл обработки сообщений, как показано в листинге 2.19.

Листинг 2.19. Цикл обработки сообщений

```
StartAnim();
// Main message loop:
while (true)
{
 if (PeekMessage(&msg, NULL, NULL, NULL, PM_NOREMOVE))
 {
  if (!GetMessage(&msg, NULL, NULL, NULL)) break;
  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
  {
   TranslateMessage(&msg);
   DispatchMessage(&msg);
  }
 }

 if (GetActiveWindow()!=hWnd)
  continue;

 DrawScene();
}
```

В самом начале вызывается метод `StartAnim`, который инициализирует анимацию. Поскольку наш пример слишком прост, то и этот метод будет состоять только из одной строчки кода:

```
tStartAnimTime = GetTickCount();
```

В случае с более сложной анимацией здесь может потребоваться большее количество действий, а пока нам ничего другого не нужно.

Далее идет сам обработчик сообщений. В *разд. 2.5* мы уже видели такой метод обработки сообщений, когда программа не блокируется. На каждом этапе цикла вызывается метод `DrawScene`. Этот метод показан в листинге 2.20.

Листинг 2.20. Функция формирования сцены

```
void DrawScene()
{
 // Область копирования
 RECT dst;
 dst.left=dst.top=0;
 dst.right=iWidth;
 dst.bottom=iHeight;
```

```
 // Копируем картинку 1
 backsurf->Blt(&dst, imagesurf, 0, DDBLT_WAIT, 0);

 // Определяем текущее время
 double tCurrentTime = GetTickCount();

 // Если прошло время анимации, то начнем с начала
 if ((tCurrentTime-tStartAnimTime)>AnimTime)
  StartAnim();

 // Определяем позицию самолета
 double iPos=StartPos+(EndPos-StartPos)*
           ((tCurrentTime-tStartAnimTime)/AnimTime);

 // Копируем картинку 2
 backsurf->BltFast(iPos, 100, imagesurf2, 0,
            DDBLTFAST_SRCCOLORKEY | DDBLTFAST_WAIT);

 // Отображение заднего буфера на переднем
 RECT clientRect;
 POINT p;
 p.x=p.y=0;
 ClientToScreen(hWnd, &p);
 GetClientRect(hWnd, &clientRect);
 OffsetRect(&clientRect, p.x, p.y);
 primsurf->Blt(&clientRect, backsurf, 0, DDBLT_WAIT, 0);
}
```

В принципе, листинг снабжен подробными комментариями и, мне кажется, все понятно и без дополнительного описания. Попробуйте запустить пример, и с секундомером в руках замерить время движения самолета. Оно должно быть примерно 4 сек (плюс/минус несколько миллисекунд, которые могут быть затрачены на последнее отображение картинки).

В данном случае самолет должен двигаться плавно от одного края до другого, потому что ресурсов даже слабого компьютера должно хватить на анимацию.

Теперь попробуйте добавить в код этой функции задержку в 100 мс:

```
Sleep(100);
```

Это можно сделать в любом месте, лишь бы задержка происходила при каждом вызове функции `DrawScene`. После этого запустите пример и снова убедитесь, что самолет преодолевает необходимое пространство за то же время,

только в этом случае движение происходит рывками. Дело в том, что за указанное время компьютер сможет отобразить самолет максимум 40 раз, т. е. за один шаг самолет должен переместиться на 14,5 пикселов (разность между конечной и начальной позиции, разделенная на количество отображений, т. е. (600 – 20)/40 = 14,5).

Примечание

Исходный код примера находится в каталоге \Chapter2\Sinchronize2D на прилагаемом компакт-диске.

2.12.4. Пример синхронизации временем 3D-графики

Теперь посмотрим, как выполняется синхронизация во времени при отображении 3D-графики. Тут работа по трансформации объектов происходит с матрицами, поэтому желательно знание того, как работать с ними. Я постараюсь описать все как можно проще и доступнее для всех, но все же эта книга не курс математики.

Формула расчета движения с учетом времени та же самая, поэтому сразу перейдем к рассмотрению примера. Давайте заставим 3D-объект, который мы создали в *разд. 2.11*, двигаться снизу вверх и движение его должно происходить ровно за 4 сек, вне зависимости от производительности компьютера и видеосистемы.

Начнем с того, что определим начальную и конечную мировые матрицы отображения объекта:

```
D3DMATRIX Start_View = {
  1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 0, 1,
};

D3DMATRIX End_View = {
  1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, 1000, 0, 1,
};
```

Первая матрица соответствует единичной. Во второй матрице координата Y будет равна 1000, т. е. движение осуществляется по оси Y от нуля до 1000.

Помимо этого, нам понадобятся переменная `tStartAnimTime` для хранения времени начала анимации и константа `AnimTime` для хранения времени движения объекта:

```
double tStartAnimTime;
const int AnimTime = 4000;
```

Теперь посмотрим, как рассчитать формулу. Вот тут нужно быть очень аккуратным. Для начала следует понять, как происходит сложение и вычитание. Во время сложения каждый элемент матрицы складывается с соответствующим элементом в другой матрице. Это так же, как сложить элементы двух массивов.

Во время вычитания происходит вычитание каждого элемента второй матрицы из первой. Например, если вычесть из матрицы `End_View` матрицу `Start_View`, мы получим следующее:

```
D3DMATRIX End_View = {
  0, 0, 0, 0,
  0, 0, 0, 0,
  0, 0, 0, 0,
  0, 1000, 0, 0,
};
```

Обратите внимание, что все элементы, кроме координаты по оси Y, стали равны нулю. Давайте вспомним формулу:

```
положение = start_point + (end_point - start_point) *
    ((t-t_start)/t_length)
```

Именно такое вычитание матриц происходит в скобках. Если следовать правилам математики, то теперь должно происходить перемножение на результат вычислений: `((t-t_start)/t_length)`. Во время умножения на число (не на матрицу), число умножается на все элементы матрицы. У нас после вычитания только один элемент не нулевой и именно он изменится после перемножения на число.

Последнее действие, которое будет выполнено, — сложение с начальной матрицей. Начальная матрица является единичной, а значит, в результате мы получим следующую матрицу:

```
D3DMATRIX End_View = {
  1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, Y, 0, 1,
};
```

где `Y` — новое значение положения объекта.

Формула расчета положения в зависимости от времени достаточно хороша, но ей нужно пользоваться аккуратно. Любое изменение последовательности расчета может привести к нежелательным последствиям. Например, если сначала рассчитать формулу `start_point + (end_point - start_point)`, а потом умножить ее на результат формулы `((t-t_start)/t_length)`, то итог будет неожиданным. Давайте посмотрим, что именно будет не так.

Посмотрим, что происходит после расчета `start_point + (end_point - start_point)`. Сначала выполняется вычитание, и мы увидим нулевую матрицу, где заполнено только одно значение — координата по оси Y. Теперь прибавляем начальную матрицу, т. е. единичную, и получаем в результате:

```
D3DMATRIX End_View = {
  1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, 1000, 0, 1,
};
```

Если же теперь умножить эту матрицу на число N (результат расчета `((t-t_start)/t_length)`), то итоговая матрица будет следующей:

```
D3DMATRIX End_View = {
  1*N, 0, 0, 0,
  0, 1*N, 0, 0,
  0, 0, 1*N, 0,
  0, 1000*N, 0, 1*N,
};
```

Как видите, изменится не только координата по оси Y, но и все единицы будут умножены на число N. Последовательность расчетов очень важна, и при работе с матрицами нельзя забывать об этом.

А вот теперь посмотрим на функцию `DrawScene`, которая показана в листинге 2.21.

Листинг 2.21. Функция отображения 3D-анимации в зависимости от времени

```
void DrawScene()
{
 double tCurrentTime = GetTickCount();

 if ((tCurrentTime-tStartAnimTime)>AnimTime)
  StartAnim();
```

```
 // Расчет матрицы
 D3DXMATRIX World = D3DXMATRIX(Start_View)+(D3DXMATRIX(Start_View) -
       D3DXMATRIX(End_View))*((tCurrentTime-tStartAnimTime)/AnimTime);

 // Настройка отображения
 pD3DDevice->SetTransform(D3DTS_WORLD, &World);
 pD3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE);

 // Отображение сетки
 for (DWORD i=0; i<dwNumMaterials; i++)
 {
  pD3DDevice->SetMaterial(&pMeshMaterials[i]);

  if (pMeshTextures[i])
   pD3DDevice->SetTexture(0, pMeshTextures[i]);

   pMesh->DrawSubset(i);
  }
}
```

Как видите, математика с матрицами практически ничем не отличается от математики с числами. Единственное отличие — каждая матрица должна заключаться в скобки, а перед ней нужно указывать, что это матрица типа `D3DXMATRIX`. В остальном логика расчета и отображения нам уже знакома, и тут просто нечего больше добавить.

Чтобы проверить корректность синхронизации, можно снова добавить вызов функции `Sleep`, чтобы задержать формирование сцены и сетка снова перескакивала по несколько пикселов, дабы успеть за 4 сек перебежать в конечную точку.

Как видите, синхронизация в третьем измерении не намного сложнее двух измерений, а в чем-то даже проще.

Примечание

Исходный код примера находится в каталоге \Chapter2\Sinchronize3D на прилагаемом компакт-диске.

ГЛАВА 3

Оптимизация в DirectX

В *разд. 1.4* мы говорили об основах оптимизации. Эти основы касаются практически любого типа программ, но графические программы не любые. У них есть свои нюансы, и с ними необходимо считаться.

В этой главе нам предстоит познакомиться с некоторыми приемами оптимизации графических приложений, а именно DirectDraw и Direct3D.

Я всегда расхваливаю DirectX за ее гениальные решения множества проблем, но большинство интерфейсов DirectX слишком универсальны, и иногда лучше обойтись без них, т. е. через блокировки и прямой доступ. Так, например, в *разд. 3.1* мы увидим, как произвести закраску поверхности без использования DirectDraw функций, а через прямой доступ.

В этой главе нас ожидают некоторые алгоритмы быстрого решения типичных задач. Большинство из них помогут нам в будущем, когда мы будем писать более законченные приложения.

3.1. Оптимизация графики

Оптимизация графики — это достаточно интересный процесс, потому что здесь, помимо общих принципов оптимизации программ, существует еще множество приемов. Например, чтобы оптимизировать размер картинки, можно ее сжать классическим архиватором. Размер файла уменьшится, но коэффициент сжатия зависит от самой картинки, например, фотографии с большим количеством цветов сжимаются очень плохо.

Человеческий глаз не способен различить два оттенка цвета, если в них отличается на единицу только одна составляющая. Получается, что если объединить близлежащие точки с небольшим отличием в оттенке и заменить их одним цветом, то файл сожмется больше, а пользователь ничего и не заметит.

Такое сжатие называется *компрессией с потерей качества*. Тут нужно добиваться компромисса между качеством изображения и качеством сжатия и это достигается путем выбора максимальной разницы между оттенками, которые можно объединить. Ведь если будут объединены два цвета, разница которых незаметна глазу, то качество пострадает не сильно. Но если отличие заметно, то качество теряется.

Где это используется? Конечно же в файлах изображений, которые может загружать программа. Чем меньше файл, тем быстрее его можно загрузить в память с жесткого диска, который является самым слабым звеном компьютера. После этого, процессор восстановит изображение, и если алгоритм сжатия не сложный, то это может быть выполнено быстрее, чем загружать с диска не сжатые данные.

Количество используемых цветов влияет и на требуемые ресурсы. Чем больше цветов мы используем, тем больше нужно памяти для хранения изображений и поверхности и больше надо процессорного времени, чтобы копировать всю эту память между поверхностями. Предположим, например, что у нас есть картинка размером в 100×100 пикселов. Если используется глубина цвета в 1 байт, то для хранения изображения понадобится 10000 байт памяти. Поскольку такая картинка сможет содержать максимум 256 цветов, а это очень мало, и поэтому желательно использовать минимум 2 байта, когда количество цветов будет равно 65535. Этого уже достаточно для хранения более качественного изображения, но в этом случае понадобится 20000 байт памяти, что в два раза больше, а значит, и копирование данных будет требовать от процессора в два раза больше ресурсов.

Если же выбрать глубину цвета в 24 бита, то тут уже понадобится в 3 раза больше ресурсов, хотя количество цветов будет исчисляться 16777216. Вот и думай после этого, что выбрать — качество или скорость. Наша задача выбрать оптимальный вариант, который позволит получить лучшее соотношение скорости и качества.

Размер изображений также играет немаловажную роль. Ведь если мы выбрали разрешение экрана в 800×600 пикселов при 16-битном цвете, то для хранения поверхности понадобится $800 \times 600 \times 2 = 960000$ байт памяти. Это почти 1 Мбайт! А если изображение будет 1024×768, то тут уже объем необходимой поверхности памяти составит 1572864 байт. Снова увеличение ресурсов в два раза, а значит, и падение производительности во время копирования содержимого поверхностей.

Оптимизация графики на первоначальном этапе — это борьба выбора между качеством и скоростью. В настоящее время стандартом стали ЖК мониторы с диагональю 15 дюймов, и чтобы картинка на них выглядела приемлемо, необходимо использовать разрешение 800×600, а лучше 1024×768, поэтому мы чаще всего будем использовать именно эти значения.

Но неплохо было бы сделать возможность выбора необходимого разрешения. Например, у меня широкоформатный ноутбук и идеалом для него является разрешение 1280×800, а разрешения 800×600 и 1024×768 выглядят растянуто, что очень неудобно. Поэтому в данном случае можно предложить пользователю самому выбирать необходимое разрешение в зависимости от имеющихся ресурсов.

Для указания глубины цвета мы чаще всего будем использовать 16 бит, что позволяет добиться приемлемого качества при минимуме затрат. Можно предложить пользователю выбирать и глубину цвета, но тогда придется писать слишком универсальный код, который понизит производительность. Чтобы не терять в скорости, лучше все же привязаться к определенной глубине экрана.

Двумерная графика может формироваться двумя способами: математическим и спрайтовым. В случае математического способа, поверхность заполняется через прямой доступ по определенному алгоритму. Некоторые из алгоритмов создания эффектов мы рассмотрим в *главе 4*. При использовании математики скорость формирования сцены, помимо глубины цвета и разрешения, очень сильно зависит от используемого алгоритма.

В случае со спрайтовым формированием сцены, дополнительным фактором торможения является количество спрайтов, потому что для вывода каждого из них необходимо выполнить операцию `Blt` или `BltFast`. Чем больше обращений, тем больше потеря, потому что тут проявляется эффект цикла — приходится много раз вызывать одну и ту же функцию, во время вызова несколько раз параметры поднимаются в стеке (а у `Blt` их немало) и происходит вызов удаленной функции, а также куча лишних проверок со всеми вытекающими последствиями. Иногда функцию `Blt` даже лучше заменить на копирование данных через прямой доступ к памяти и WinAPI-функцию `memcpy`.

Потери данных при спрайтовом выводе могут быть и неоправданными. Допустим, что у нас экран размером в 800×600 пикселов, как показано на рис. 3.1. Все, что закрашено черным цветом, — это оборка, которая статична и не изменяется, а в белой области данные формируются динамически, причем полностью. Эту задачу можно решить следующим образом: перед формированием кадра вывести на экран изображение 800×600 с оборкой, а потом сформировать среднюю часть. Но зачем это делать, если оборка статична? Не лучше ли один раз вывести оборку, а потом перекрашивать только центральную часть, которая изменяется? Конечно же лучше, и таким образом вы сэкономите целую операцию `Blt`, которой приходится копировать большой объем информации.

Если экран программы всякий раз формируется полностью, то тогда и не стоит перед формированием сцены очищать его. Все равно каждый пиксел будет закрашен и от нашей заливки ничего не останется. Другое дело, когда вы рисуете движение звезд на черном небе. В этом случае действительно удобнее закрасить экран черным, а потом поверх неба нарисовать звезды. Но если на экране присутствует поверхность земли (например, из 600 пикселов в высоту — нижние 200 это земля, а верхние 400 это черное небо), то закрашивать нужно только ту часть, где есть небо и нет смысла очищать весь экран.

Рис. 3.1. Схема экрана программы

На этом общие сведения оставим в покое и перейдем к более серьезным задачам. Дальнейший материал этой главы будет показывать алгоритмы быстрого решения типичных задач, с которыми вы можете столкнуться в будущих проектах. Все же быстрый алгоритм позволит повысить скорость работы без потерь качества, что лучше любой оптимизации кода.

3.2. Быстрая закраска поверхности

В *разд. 2.2.5* для очистки содержимого поверхности мы использовали метод `Blt`. Его производительность нельзя назвать идеальной, потому что:

- метод слишком сложен и его код не может быть оптимальным;
- до вызова метода необходимо объявить структуру типа `DDBLTFX`, заполнить ее нулями, указать размер и цвет закраски;
- на вызов метода `Blt` требуется время, потому что у него достаточно много параметров и их необходимо поместить в стек.

Все это затраты времени, которые нельзя назвать идеальными, и ради заливки цветом использование метода `Blt` оказывается слишком неэффективным. Более оптимальным вариантом будет применение блокировки и закраски вручную. Алгоритм закраски в этом случае следующий:

1. Заблокировать поверхность.
2. Заполнить полученную память необходимыми значениями.
3. Разблокировать поверхность.

Самый простой способ закраски — когда используется черный цвет, потому что этот цвет равен нулю, и достаточно воспользоваться функцией `ZeroMemory`, чтобы заполнить всю поверхность памяти черным цветом. В листинге 3.1 можно увидеть пример, который показывает, как это можно реализовать.

Листинг 3.1. Быстрое заполнение поверхности нулями

```
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);

if (backsurf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
{
  BYTE* dst = (BYTE *)desc.lpSurface;
  ZeroMemory(dst, desc.lPitch*iHeight);
  backsurf->Unlock(0);
}
```

Сначала мы формируем структуру `desc` типа `DDSURFACEDESC2`, которая понадобится при блокировке поверхности. Далее происходит блокировка поверхности, и если все прошло удачно, то мы получаем указатель на память поверхности и заполняем ее нулями с помощью функции `ZeroMemory`. Для определе-

ния размера заполняемой области мы умножаем поле `lPitch` (содержит количество байт между двумя строками) на количество строк. Сразу же после заполнения освобождаем поверхность, дабы блокировка не была долгой.

Если необходимо произвести заполнение поверхности не черным цветом, то можно воспользоваться функцией `FillMemory`, например:

```
FillMemory(dst, desc.lPitch*iHeight, 20);
```

Но тут есть одна проблема — функция заполняет память указанным числом, размером в байт. Именно размерность и является проблемой. Если у нас используется 8-битный цвет, то каждая точка будет заполнена указанным числом, в данном случае это 20 (14h).

Но в наше время такой глубины цвета мало и в большинстве случаев используется минимум 16 бит, т. е. 2 байта, когда каждая точка имеет цвет 1414h. Результат зависит от формата пиксела R5G6B5 или R5G5B5, поэтому если пользователь может выбрать формат пиксела самостоятельно, то результат будет разным.

Если же у нас будет 24-битный формат пиксела, то каждая точка окрашивается цветом 141414h, т. е. максимум, чего мы можем добиться — любого оттенка серого цвета от белого до черного.

Результат использования функции `FillMemory` зависит от формата пиксела и не позволяет получить любой цвет заполнения.

Для работы с памятью есть еще WinAPI-функция `memset`, которая заполняет память указанным числом размером в `int`, но и это не очень хороший выход, потому что привязка к размеру заполняемого символа усложняет жизнь. Вот если бы у нас была возможность задавать размерность заполняемого символа, вот это была бы жизнь!

Если же необходимо заполнить поверхность произвольным цветом, то можно запустить цикл и заполнить поверхность вручную, без специализированных функций. Например, если у вас используется 16-битный формат пиксела, то заполнение поверхности красным цветом может выглядеть, как показано в листинге 3.2.

Листинг 3.2. Заполнение поверхности красным цветом по 2 байта

```
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);

if (backsurf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
{
  WORD* dst = (WORD *)desc.lpSurface;
```

```
  int count=desc.lPitch*iHeight;
  while (count--)
  {
    *(WORD*)dst = (WORD)63488;
    dst = (WORD *)dst + 2;
  }
  backsurf->Unlock(0);
}
```

В данном случае мы запускаем цикл, в котором каждое слово (WORD) заполняется указанным цветом. Число 63 488 соответствует красному цвету при формате пиксела R5G6B5.

Но и это еще не все. Мы можем оптимизировать этот пример еще больше и повысить производительность в два раза. Как мы уже говорили, самое слабое звено любой программы — это циклы. В данном случае цикл простой, но он выполняется много раз, а если быть точнее, то при 16-битном цвете выполнение будет происходить число раз определяемое по следующей формуле: 2 (байта) × ширина поверхности × высота поверхности. Для поверхности размером 800×600 будет 960 000 обращений к памяти и переходов в цикле. Можно сократить эти операции в два раза и для этого достаточно заполнять сразу по 2 байта памяти. Следующий пример демонстрирует это (листинг 3.3).

Листинг 3.3. Заполнение поверхности любым цветом по 4 байта за цикл

```
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);

if (backsurf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
{
 DWORD* dst = (DWORD *)desc.lpSurface;

 int count=desc.lPitch*iHeight;
 while (count=count-2)
 {
  *(DWORD *)dst = (DWORD)4160813056;
  dst = (DWORD *)dst + 1;
 }

 backsurf->Unlock(0);
}
```

В этом примере мы перемещаем сразу по 2 байта за один шаг цикла, а число 4 160 813 056 (F800F800h) соответствует двум идущим подряд числам 63 488 (F800h).

Можно еще повысить скорость работы, если переписать этот пример на языке ассемблера, но мы не будем опускаться до такой "низости программирования", потому что у нас основной язык это C, и не будем лишний раз уходить за его рамки.

Скорость — это хорошо, но мы потеряли универсальность. Данная функция заполнения поверхности привязана к формату пиксела в 2 байта. Если ваше приложение будет использовать разные форматы, то придется написать несколько вариантов заполнений и код для правильного определения цвета, в зависимости от размера пиксела.

Примечание

Исходный код примера находится в каталоге \Chapter3\FastClear на прилагаемом компакт-диске.

3.3. Рисование линий

Для работы с графикой очень часто приходится рисовать линии, но в DirectDraw нет функций для рисования примитивов, как в WinAPI и ее библиотеки GDI. Если при программировании с помощью GDI у вас есть все необходимое для рисования линий, прямоугольников и кругов, то в DirectDraw ничего подобного не заложено. Вы конечно же можете получить контекст рисования DirectDraw поверхности и использовать стандартный WinAPI-набор функций, но от этого скорость вывода только замедлится, и поэтому мы договорились забыть о том, что обращение к GDI вообще возможно.

Благодаря отсутствию функций рисования примитивов, для создания простой линии придется немного попотеть. Давайте напишем универсальную и очень быструю функцию, которая позволит вывести на поверхность линию. Раз линии должны выводиться очень быстро, то для этого будем использовать прямой доступ к памяти. В принципе, это единственный способ нарисовать что-либо на экране, не считая копирования в поверхность подготовленной картинки. Но поскольку линии бывают разные и заранее подготовить изображения всех возможных вариантов нереально, то придется поработать ручками.

Для начала давайте определимся, что линию всегда будем рисовать сверху вниз. В этом случае нам нужно будет всегда увеличивать координату по оси Y (или оставлять неизменной, если линия расположена горизонтально). Это немного упростит задачу, но для универсальности кода придется вначале проверять координаты, и если `Y1` будет больше чем `Y2`, то нужно менять координаты местами.

Давайте посмотрим на код, который рисует линию, и на его примере представим алгоритм. Так мы сэкономим время и место в книге. Функция рисования линии показана в листинге 3.4.

Листинг 3.4. Функция рисования линии

```
void DrawLine(IDirectDrawSurface7* surf, int X1, int Y1,
              int X2, int Y2, BYTE r, BYTE g, BYTE b)
{
 WORD c = (WORD)(b/8 | (g/4 << 5) | (r/8 << 11));

 int NS;

 // Если первая точка выше, то меняем их местами,
 // чтобы всегда рисовать сверху вниз
 if (Y2<Y1)
 {
  NS=Y1;
  Y1=Y2;
  Y2=NS;

  NS=X1;
  X1=X2;
  X2=NS;
 }

// Блокировка поверхности
 DDSURFACEDESC2 desc;
 ZeroMemory(&desc, sizeof(desc));
 desc.dwSize = sizeof(desc);

 if (surf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
 {
  BYTE* dst = (BYTE *)desc.lpSurface;
  int DX=abs(X2-X1);
  int DY=abs(Y2-Y1);
  int iOffset=Y1*desc.lPitch+X1*2;

  // Рисуем слева направо
  if (X1<=X2)
  {
   //Рисуем слева направо и X изменяется чаще
   if (DX>DY)
```

```
    {
     NS=DX % 2;
     for (int i=0; i<=DX; i++)
     {
      *(WORD *)(dst+iOffset)=c;
      iOffset=iOffset+2;
      NS=NS-DY;
      if (NS<0)
      {
       iOffset=iOffset+desc.lPitch;
       NS=NS+DX;
      }
     }
   }
   else
   {
    //Рисуем слева направо и Y изменяется чаще
    NS=DY % 2;
    for (int i=0; i<DY; i++)
    {
     *(WORD *)(dst+iOffset)=c;
     iOffset=iOffset+desc.lPitch;
     NS=NS-DX;
     if (NS<0)
     {
      iOffset=iOffset+2;
      NS=NS+DY;
     }
    }
  }
 }
 else
 // Рисуем справа налево
 {
  //Рисуем справа налево и X изменяется чаще
  if (DX>DY)
  {
   NS=DX % 2;
   for (int i=0; i<=DX; i++)
   {
    *(WORD *)(dst+iOffset)=c;
    iOffset=iOffset-2;
    NS=NS-DY;
```

```
    if (NS<0)
    {
     iOffset=iOffset+desc.lPitch;
     NS=NS+DX;
    }
   }
  }
  else
  {
   //Рисуем справа налево и Y изменяется чаще
   NS=DY % 2;
   for (int i=0; i<DY; i++)
   {
    *(WORD *)(dst+iOffset)=c;
    iOffset=iOffset+desc.lPitch;
    NS=NS-abs(DX);
    if (NS<0)
    {
     iOffset=iOffset-2;
     NS=NS+DY;
    }
   }
  }
 }
 surf->Unlock(0);
}
```

Для начала определимся с переменными, которые нам понадобятся:

- `X1`, `Y1`, `X2`, `Y2` — все это целые числа, в которых хранятся координаты двух точек (начало и конец линии). Значения этих переменных мы получаем через параметры функции;
- `DX`, `DY` — длина отрезка по оси X и Y. Переменная `DX` — это разность между `X2` и `X1`, а переменная `DY` — это `Y2` минус `Y1`;
- `iOffset` — здесь будем хранить указатель на точку в видеопамяти, куда мы будем выводить следующую точку;
- `NS` — переменная-счетчик, определяющая изменение редко изменяемой координаты. Алгоритм построен так, что сначала определяется ось, по которой изменение происходит чаще. Например, если отрезок имеет координаты точек (0,0) и (100,10), то по горизонтали будет 100 точек, а по вертикали 10, т. е. чаще будем рисовать линию вправо, и иногда вниз. Значение переменной `NS` будет определять, когда нужно опуститься вниз.

Прежде чем рисовать отрезок, нужно еще проверить соотношение координат. Если координата `Y2` больше `Y1`, то следует поменять точки местами и рисовать в обратном направлении. Это необходимо, чтобы не писать код рисования снизу вверх, а только сверху вниз.

Когда все подготовительные расчеты выполнены, можно блокировать поверхность и начинать выводить линию. Для этого определяем положение первой точки в видеопамяти и заносим это значение в переменную `iOffset`. В результате, сначала умножаем координату `Y1` на ширину строки (`desc.lPitch`), чтобы найти в видеопамяти строку `Y1`, а затем просто добавляем к результату координату `X1`.

Весь код делится на четыре части, в зависимости от направления отрезка:

```
if (X1<=X2)
{
  // Рисуем слева направо
  if (DX>DY)
  {
    //Рисуем слева направо и X изменяется чаще
  }
  else
  {
     //Рисуем слева направо и Y изменяется чаще
  }
}
else
  // Рисуем справа налево
 {
     if (DX>DY)
     {
       //Рисуем справа налево и X изменяется чаще
     }
     else
     {
      //Рисуем справа налево и Y изменяется чаще
     }
}
```

Рассмотрим вариант, когда значение переменной `X1` меньше `X2`, т. е. отрезок направлен вправо и ширина больше, чем длина (`if (DX>DY)`):

```
NS=DX % 2;
for (int i=0; i<=DX; i++)
{
 *(WORD *)(dst+iOffset)=c;
```

```
 iOffset=iOffset+2;
 NS=NS-DY;
 if (NS<0)
 {
  iOffset=iOffset+desc.lPitch;
  NS=NS+DX;
 }
}
```

В переменную-счетчик `NS` заносим результат деления ширины строки на 2. Эта корректировка необходима, чтобы отрезок в начале и в конце был более гладким. Идеально гладким он не будет, потому что мы не станем использовать сглаживание, и все же небольшая корректировка не помешает.

Теперь запускаем цикл, который должен выполняться, от 0 до значения ширины линии, потому что она больше. На каждом этапе цикла по адресу `iOffset` выводим код цвета и увеличиваем значение смещения `iOffset` на 2 (размер глубины цвета), чтобы указатель `iOffset` показывал на следующую точку в памяти.

Далее, вычитаем из счетчика `NS` значение `DY`, и если счетчик становится меньше нуля, то нужно не просто сдвинуться вправо, а перейти на следующую строку (опуститься на один пиксел по вертикали). Для этого увеличиваем переменную `iOffset` на значение ширины поверхности (`desc.lPitch`) и увеличиваем счетчик `NS` на значение ширины строки, чтобы заново начать отсчет следующего спуска по вертикали.

Теперь посмотрим, что происходит, если отрезок более вытянут по вертикали:

```
NS=DY % 2;
for (int i=0; i<DY; i++)
{
  *(WORD *)(dst+iOffset)=c;
  iOffset=iOffset+desc.lPitch;
  NS=NS-DX;
  if (NS<0)
  {
      iOffset=iOffset+2;
      NS=NS+DY;
  }
}
```

Код очень похож на тот, что мы уже рассматривали. Но в данном случае линия больше направлена вниз, поэтому на каждом этапе цикла мы переходим на следующую строку, т. е. увеличиваем переменную `iOffset` на значение

ширины поверхности (desc.lPitch). Смещение вправо (увеличение смещения на ширину пиксела) происходит только когда счетчик iOffset становится меньше нуля.

Остальные варианты предлагаю рассмотреть самостоятельно. Попробуйте мысленно пройтись по коду и посмотреть, что происходит.

Возможно, эта функция в будущем вам еще пригодится, поэтому я вынес ее код в модуль ddfunc.cpp. Но помните, что она работает только с 16-битным цветом. Если вам понадобится другая глубина, то придется ее немного модифицировать, а может быть лучше будет написать еще один вариант функции, но для 24-битного цвета.

Теперь посмотрим, как можно использовать функцию DrawLine. Для этого по событию WM_PAINT можно написать код, как это показано в листинге 3.5.

Листинг 3.5. Использование функции DrawLine

```
case WM_PAINT:
  hdc = BeginPaint(hWnd, &ps);
  // TODO: Add any drawing code here...

  // Восстанавливаем поверхности, если это необходимо
  if (primsurf->IsLost())
    RestoreSurfaces();

  // Очистка поверхности
  ClearSurface(backsurf, 0);

  DrawLine(backsurf, 10, 10, 220, 20, 255, 0, 0);
  DrawLine(backsurf, 10, 10, 50, 100, 0, 0, 255);
  DrawLine(backsurf, 20, 200, 300, 100, 0, 255, 255);
  DrawLine(backsurf, 500, 500, 100, 300, 255, 0, 255);

  // Переключение поверхности
  primsurf->Flip(NULL, DDFLIP_WAIT);

  EndPaint(hWnd, &ps);
  break;
```

Результат работы этого примера представлен на рис. 3.2.

Поскольку мы ничего не выводим на поверхность с помощью метода Blt, то проверку на потерю поверхностей приходится выполнять методом isLost().

После этого очищаем поверхность уже знакомой нам функцией `ClearSurface` и выводим на заднюю поверхность линии. Все очень просто!

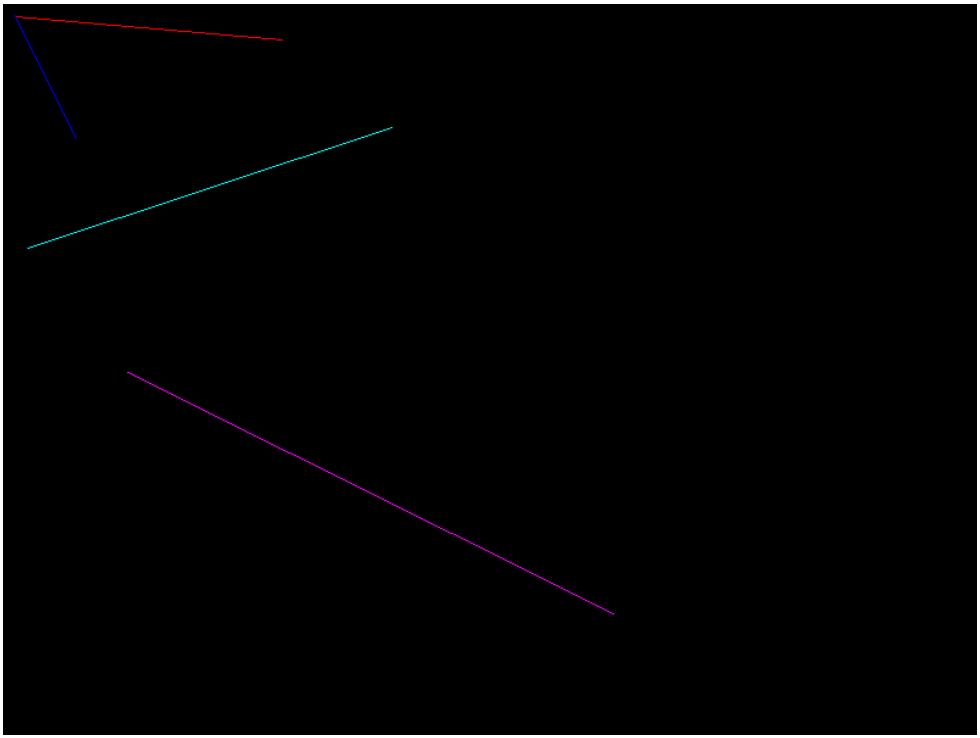

Рис. 3.2. Результат работы программы рисования линий

Алгоритм достаточно быстрый, но нельзя сказать, что оптимальный. Даже простого взгляда на параметры функции достаточно, чтобы увидеть, что их слишком много и их возможно сократить. Например, координаты точек можно передавать через структуру `POINT`. Таким образом, мы сэкономим две переменные, а код не сильно усложнится. Это позволит ускорить вызов функции за счет меньшего количества обращений к стеку. Я же не стал использовать такой оптимизации, потому что это создаст цепочку в виде `point.x`. Любая цепочка тоже является минусом для производительности.

Можно объединить цвет заранее в 2 байта и передавать функции одним числом, и снова будет экономия в два параметра, только без дополнительных затрат и лишних цепочек. Но в этом случае функция будет не масштабируема. Если вы захотите ее сделать универсальной для различных режимов, то возникнут проблемы с цветом. Его придется расширять из 16 бит в 24 или 32. При этом восстановление без потерь будет невозможным, ведь в 16-битном

режиме для каждой составляющей цвета выделяется меньше байт и восстановить все 255 возможных значений для каждой составляющей будет нельзя.

Можно улучшить код, добавив более быструю операцию деления на 2 (заменить ее сдвигом), и я советую сделать это. Я же не стал этого делать, дабы код был более наглядным.

Данный алгоритм работает в 16-битном цвете, поэтому демонстрационный пример будет работать только в полноэкранном режиме, где мы можем контролировать глубину цвета. В оконном режиме это невозможно, но вы можете написать универсальный алгоритм рисования линии.

Примечание

Исходный код примера находится в каталоге \Chapter3\DrawLine на прилагаемом компакт-диске.

3.4. Быстрая загрузка картинок

В *разд. 2.1* мы начали изучение DirectDraw с загрузки картинок. Тогда наши познания этой технологии были слишком малы, поэтому рисование изображения на поверхности происходило через GDI-функцию `StretchBlt`. Скорость этой функции далека от идеала, и поэтому сейчас мы рассмотрим более быструю функцию загрузки битового изображения.

Быстрый вариант функции загрузки назовем `LoadBMPToSurfaceFast`. Ее код показан в листинге 3.6. Функция достаточно большая, но не сложная (все используемые функции вам должны быть знакомы) и очень интересная.

Листинг 3.6. Быстрая загрузка битовой матрицы

```
IDirectDrawSurface7 *LoadBMPToSurfaceFast(IDirectDrawSurface7 **ddsurf,
    LPCTSTR filename, IDirectDraw7 *ppiDD)
{
 int iImageWidth, iImageHeight;

  // Открываем файл картинки для чтения
 HANDLE hFile = CreateFile(filename, GENERIC_READ, 0, NULL,
     OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
 if(hFile == INVALID_HANDLE_VALUE)
  return FALSE;

 // Загружаем заголовок файла
 BITMAPFILEHEADER bmpfilehdr;
 DWORD dwRead;
```

```
if (!ReadFile(hFile, &bmpfilehdr, sizeof(bmpfilehdr), &dwRead, NULL))
 return FALSE;

// Проверяем сигнатуру файла
char* ptr=(char*)&bmpfilehdr.bfType;
if (*ptr!='B' || *++ptr!='M')
 return FALSE;

// Проверяем заголовок изображения
BITMAPINFOHEADER bmpinfohdr;
if(!ReadFile(hFile, &bmpinfohdr, sizeof(bmpinfohdr), &dwRead, NULL))
 return FALSE;

// Для удобства ширину и высоту сохраняем в локальных переменных
iImageWidth=bmpinfohdr.biWidth;
iImageHeight=bmpinfohdr.biHeight;

// Если картинка сжата, то выходим. Сжатие использовать не будем
if (bmpinfohdr.biCompression!=BI_RGB)
 return 0;

// Переменные для работы с поверхностью
IDirectDrawSurface7 *surf=*ddsurf;
DDSURFACEDESC2 desc;
HRESULT hRes;

// Если необходимо, то создаем поверхность
if (surf==NULL)
{
 // Описываем структуру для создания поверхности
 ZeroMemory(&desc, sizeof(desc));
 desc.dwSize = sizeof(desc);
 desc.dwFlags = DDSD_WIDTH | DDSD_HEIGHT | DDSD_CAPS;
 desc.dwWidth = iImageWidth;
 desc.dwHeight = iImageHeight;
 desc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN | DDSCAPS_SYSTEMMEMORY;

 // Создание поверхности
 hRes=ppiDD->CreateSurface( &desc, &surf, 0 );
 if (hRes!=DD_OK)
  return FALSE;
}
```

```
 // Определяем размер картинки
 int iImageSize=bmpinfohdr.biSizeImage;
 if (iImageSize==0)
 iImageSize=((iImageWidth*(bmpinfohdr.biBitCount/8)+3)& ~3)*iImageHeight;

 // Читаем данные изображения в буфер
 BYTE* buf=new BYTE[iImageSize];
 if(!ReadFile(hFile, buf, iImageSize, &dwRead, NULL))
  return FALSE;

 // Будем загружать только 24-битные картинки
 if (bmpinfohdr.biBitCount!=24)
  return FALSE;

 // Копируем изображение
 ZeroMemory( &desc, sizeof(desc) );
 desc.dwSize = sizeof(desc);
 hRes=surf->Lock( 0, &desc, DDLOCK_WAIT | DDLOCK_WRITEONLY, 0 );
 if (hRes!=DD_OK)
  return FALSE;

 int bytesrequired=iImageWidth*3;
 int bytesgiven=(bytesrequired+3) & ~3;
 BYTE* btSurf = (BYTE*)desc.lpSurface;
 BYTE* imagebits = (BYTE*)(&buf[(iImageHeight-1)*bytesgiven]);

 for (int i=0; i<iImageHeight; i++)
 {
  WORD* wdSurf=(WORD*)btSurf;
  RGBTRIPLE* tlImage=(RGBTRIPLE*)imagebits;
  for (int p=0; p<iImageWidth; p++)
  {
   *wdSurf = (WORD)((tlImage->rgbtBlue / 8) |
             ((tlImage->rgbtGreen / 4) << 5) |
             ((tlImage->rgbtRed / 8) << 11));
    tlImage++;
   wdSurf++;
  }
  btSurf += desc.lPitch;
  imagebits -= bytesgiven;
 }
 surf->Unlock( 0 );
 delete [] buf;
 return surf;
}
```

Данная функция работает с графическим файлом напрямую, открывая его для чтения WinAPI-функцией `CreateFile`. Для чтения данных будем использовать WinAPI-функцию `ReadFile`.

Прежде чем читать файл, сначала загрузим заголовок файла и проверим, равна ли сигнатура BM. Именно с этих букв должен начинаться BMP-файл, который мы будем загружать. Если буквы другие, то можно прерывать загрузку, дабы избежать возможные проблемы.

После этого считываем информацию о битовом файле в структуру `BITMAPINFOHEADER`. Из этой структуры нам понадобятся ширина и высота изображения, которые для удобства сохраняются в переменных `iImageWidth` и `iImageHeight`, а также глубина цвета растрового изображения.

Последняя проверка перед загрузкой — есть ли компрессия. Не будем потеть над сжатыми данными, поэтому если компрессия есть, то загружать не будем.

Когда всевозможные проверки выполнены, можно создавать поверхность. Почему мы этого не делали ранее? Чтобы при выходе по ошибке не пришлось уничтожать созданную поверхность, ведь это усложнит код, потому что надо определять — поверхность создавалась в функции или передавалась уже проинициализированной. Именно поэтому сначала лучше проверить корректность файла. Вероятность остальных ошибок минимальна, например, трудно себе представить, что ошибка возникнет при создании поверхности, когда все параметры переданы правильно.

При создании поверхности мы проверяем — является ли переданный указатель действительным. Если да, то поверхность уже существует и незачем ее заново создавать. Если нет, то создаем интерфейс `IDirectDrawSurface7` для хранения изображения с корректными размерами.

Теперь загружаем сами данные битовой картинки:

```
BYTE* buf=new BYTE[iImageSize];
if(!ReadFile(hFile, buf, iImageSize, &dwRead, NULL))
 return FALSE;
```

Здесь мы объявляем байтовый массив `buf`, под который выделяется память для хранения данных всего изображения. Затем в этот массив читаем сами данные.

Для копирования заблокируем поверхность, и в цикле будем загружать данные. Эта функция "заточена" под загрузку 24-битных изображений в 16-битную поверхность. Если вам нужна 24-битная поверхность, то придется написать код самостоятельно. Мне кажется, что этот код будет проще, чем 16-битный формат, да и если вы напишете его, то лишний раз потренируетесь в работе с поверхностями.

Чтобы цикл копирования был проще, введем две переменные, одна из которых будет указывать на текущую точку копируемой памяти поверхности, а другая на копируемую память буфера:

```
BYTE* btSurf = (BYTE*)desc.lpSurface;
BYTE* imagebits = (BYTE*)(&buf[(iImageHeight-1)*bytesgiven]);
```

Цикл загрузки изображения из битового массива в поверхность выглядит следующим образом:

```
for (int i=0; i<iImageHeight; i++)
{
 WORD* wdSurf=(WORD*)btSurf;
 RGBTRIPLE* tlImage=(RGBTRIPLE*)imagebits;
 for (int p=0; p<iImageWidth; p++)
 {
  *wdSurf = (WORD)((tlImage->rgbtBlue / 8) |
           ((tlImage->rgbtGreen / 4) << 5) |
           ((tlImage->rgbtRed / 8) << 11));
  tlImage++;
  wdSurf++;
 }
 btSurf += desc.lPitch;
 imagebits -= bytesgiven;
}
```

Поскольку глубины цвета буферов не совпадают, то цвет каждой точки приходится преобразовывать. Да, это потеря времени, и если бы у нас форматы были одинаковые, то можно было бы воспользоваться функцией `memcpy`, что намного быстрее, а цикл мог бы выглядеть следующим образом:

```
for (int i=0; i<Высота; i++)
{
 memcpy(btSurf, imagebits, ширина картинки в байтах);
 btSurf += desc.lPitch;
 imagebits -= bytesgiven;
}
```

Только для этого должна совпадать и последовательность цветовых компонентов в цвете. Дело в том, что в растровой картинке используется формат RGB, а в DirectX может быть как RGB, так и BGR.

Примечание

Исходный код примера находится в каталоге \Chapter3\FastLoadBMP на прилагаемом компакт-диске.

3.5. Ручной контроль области вывода

В *разд. 2.4* мы познакомились с интерфейсом `IDirectDrawClipper`, позволяющим производить автоматическое отсечение данных, которые выходят за пределы окна. Это действительно мощное средство, но оно универсальное, а значит, не может быть производительным. А ведь проверить область экрана/окна и возможные выходы за его пределы не так уж и сложно.

Когда мы пишем собственный код отсечения, мы можем учитывать вероятные движения отсекаемого объекта, а интерфейс `IDirectDrawClipper` этого знать не может. Например, у нас есть сцена, в которой мы движемся с автоматом в руках (рис. 3.3). Поскольку человеку сложно двигаться не качаясь (особенно бежать), поэтому автомат тоже должен качаться и некоторая его часть в низу экрана должна то подниматься вверх, то опускаться. Не будем сейчас обращать внимание на то, что сцена на рисунке трехмерная, а мы будем говорить о двумерном отсечении.

Рис. 3.3. Сцена движения с автоматом

Автомат может двигаться только вверх-вниз, при этом боковые и верхнюю область экрана он никогда не пересечет, а значит, там нечего отсекать и можно сэкономить лишние проверки, а просматривать только пересечение нижней части.

Еще один пример — допустим, что у нас есть самолет, который движется только слева направо и никогда не пересечет верхнюю или нижнюю область экрана. Если вы захотите, чтобы он плавно появлялся слева и постепенно уходил за правую часть экрана/окна, то без интерфейса отсечения тут не обойтись. Дело в том, что если изображение самолета не будет полностью помещаться на экран, то DirectDraw не выведет его вообще. Если какая-то часть объекта невидима, то необходимо самостоятельно ее обрезать.

Попробуйте сейчас запустить пример из каталога Chapter3\Clipper_bad на прилагаемом компакт-диске. Обратите внимание, что самолет появляется быстро, когда его изображение полностью вписывается в экран, а при приближении к правой стороне экрана быстро исчезает. Давайте напишем это приложение с нуля и рассмотрим, как происходит обрезание. К тому же, пример интересен и тем, что самолет движется и одновременно происходит его анимация — наклон.

На рис. 3.4 показан пример сцены, которую мы будем создавать. В данном случае сцена состоит из двух рисунков — изображения моря с яхтами и движущегося над морем самолета.

Для начала разберемся, как будет создаваться анимация самолета. В 2D-приложениях лучшим вариантом является спрайтовая анимация, которая позволяет добиться достаточно высокой производительности, но избыточна, если посмотреть на необходимость хранить изображения анимируемого объекта в разных ракурсах. Посмотрите на рис. 3.5, где показан самолет в разных ракурсах. Мы будем выводить определенный спрайт в зависимости от состояния самолета.

Каждый спрайт имеет размер 180×90. Всего спрайтов 7 и получается, что общий размер изображения равен 180×630, потому что спрайты выстроены в ряд по вертикали.

Давайте уже перейдем к написанию примера. Создайте приложение, добавьте инициализацию DirectDraw и загрузите изображение фона в поверхность `imagesurf`, а изображение самолета в поверхность `plainsurf`. Для этого воспользуемся быстрой функцией `LoadBMPToSurfaceFast`, которую мы ранее написали в *разд. 3.4*:

```
imagesurf = LoadBMPToSurfaceFast(&imagesurf, "desktop.bmp", ppiDD);
plainsurf = LoadBMPToSurfaceFast(&plainsurf, "plain.bmp", ppiDD);
```

Теперь необходимо задать цвет прозрачности изображения самолета. Из *разд. 2.2* мы знаем, что лучше применять черный цвет, который легко исполь-

зовать при любой глубине цвета. В данном примере цвет прозрачности будем определять по левой верхней точке изображения. Это тоже достаточно простое и эффективное решение. Достаточно только заблокировать поверхность и прочитать первые 1, 2, 3 или 4 байта (зависит от глубины цвета).

Рис. 3.4. Движение самолета по горизонтали

Для анимации нам еще понадобится три переменные:

```
int iPlainLeft=-180;
float iPlainState=0;
float iStateOffset=1;
```

Переменная `iPlainLeft` определяет левую позицию самолета. Изначально она равна –180, потому что это ширина картинки самолета, и с такой позицией он окажется слева за пределами экрана.

`iPlainState` — это переменная, отражающая состояние самолета. Значение этой переменной может изменяться от 0 до 6. Именно столько у нас картинок самолета, а переменная определяет, какую из них отобразить в данный момент.

Последняя переменная `iStateOffset` определяет, в какую сторону и на сколько нужно увеличивать переменную `iPlainState`.

Пример из листинга 3.7 показывает, как может быть в виде кода реализована анимация с ручным контролем области видимости.

Рис. 3.5. Изображение самолета в разных ракурсах

Листинг 3.7. Задание цвета прозрачности через блокировку

```
// Подготавливаем переменные для блокировки
// и задания цвета прозрачности
DDCOLORKEY  cData;
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);

// Блокируем поверхность
HRESULT hRes=plainsurf->Lock( 0, &desc,
            DDLOCK_WAIT | DDLOCK_WRITEONLY, 0 );
if (hRes!=DD_OK)
  return FALSE;

// Определяем указатель на поверхность в виде указателя на WORD
//(потому что у нас 16-битная поверхность)
WORD* btSurf = (WORD*)desc.lpSurface;

// Задаем цвет прозрачности
cData.dwColorSpaceLowValue = *btSurf;
cData.dwColorSpaceHighValue = *btSurf;

// Разблокируем поверхность
plainsurf->Unlock(0);

// Устанавливаем прозрачность
plainsurf->SetColorKey(DDCKEY_SRCBLT, &cData);
```

Код в листинге подробно закомментирован, поэтому вы разберетесь с ним и без дополнительных пояснений. Тем более что все, что здесь используется, нам уже знакомо.

Теперь перейдем к реализации вывода самолета с учетом отсечения. Поскольку у нас пример будет использовать анимацию, то давайте реализуем вывод графики в главном цикле приложения. Код главного цикла показан в листинге 3.8.

Листинг 3.8. Вывод графики с ручным контролем области отображения

```
while (true)
{
 if (PeekMessage(&msg, NULL, NULL, NULL, PM_NOREMOVE))
```

```
{
 if (!GetMessage(&msg, NULL, NULL, NULL)) break;
 if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
 {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
 }
}

if (GetActiveWindow()!=hWnd)
 continue;

if (imagesurf!=NULL)
{
 iPlainLeft++;
 if ((iPlainLeft+180)>800)
  iPlainLeft=iPlainLeft;

 // Расчет области приемника копирования
 RECT dst;
 dst.left=iPlainLeft>0 ? iPlainLeft : 0;
 dst.top=10;
 dst.right=(iPlainLeft+180)<800 ? iPlainLeft+180 : 800;
 dst.bottom=100;

 // Расчет области источника копирования
 RECT src;
 src.left=iPlainLeft>0 ? 0 : abs(iPlainLeft);
 src.top=WORD(iPlainState)*90;
 src.right=(iPlainLeft+180)<800 ? 180 : abs(iPlainLeft-800);
 src.bottom=(WORD(iPlainState)+1)*90;

 iPlainState+=iStateOffset;
 if (iPlainState>7)
   iStateOffset=-0.02;
 if (iPlainState<0)
  iStateOffset=0.02;
 if (iPlainLeft>980)
  iPlainLeft=-180;

 // Copy Surfaces
 backsurf->BltFast(0, 0, imagesurf, 0, 0);
 backsurf->Blt(&dst, plainsurf, &src, DDBLT_KEYSRC | DDBLT_WAIT, 0);
}
```

```
 if (bWindowed)
 {
  RECT clientRect;
  POINT p;
  p.x=p.y=0;
  ClientToScreen(hWnd, &p);
  primsurf->BltFast(p.x, p.y, backsurf, 0, 0);
 }
 else
  primsurf->Flip(NULL, DDFLIP_WAIT);
}
```

Самое интересное здесь — это определение областей, откуда и куда копировать. Давайте сначала посмотрим, как определяется область, откуда идет копирование:

```
RECT src;
src.left=iPlainLeft>0 ? 0 : abs(iPlainLeft);
src.top=WORD(iPlainState)*90;
src.right=(iPlainLeft+180)<800 ? 180 : abs(iPlainLeft-800);
src.bottom=(WORD(iPlainState)+1)*90;
```

В левую позицию помещается нуль, если текущая позиция самолета больше нуля. Это значит, что изображение будет копироваться с самого начала. Если же положение отрицательное, то копирование начинается с позиции `iPlainLeft`, только берется положительное значение. Это значит, если самолет имеет позицию –10, то первые 10 пикселов картинки копироваться не будут.

В верхнюю позицию структуры записывается результат перемножения состояния `iPlainState` на высоту картинки (90). Таким образом, мы находим верхнюю позицию копируемого изображения, в зависимости от состояния.

Правая позиция может обрезаться, поэтому здесь есть интересная логика. Если левая позиция плюс ширина самолета больше ширины экрана, значит, самолет выходит за правую часть экрана и нужно обрезать лишнее. В этом случае правая позиция копирования будет равна абсолютному значению разницы `iPlainLeft` и ширины картинки. Например, если левая позиция равна 700, то `iPlainLeft+180` будет равно 880. Последние 80 пикселов выходят за пределы экрана и их не нужно выводить. Если вычесть из левой позиции (700) значение 800, то получится 100, а значит, область копирования ограничится 100 пикселами, и последние 80 не будут выведены.

С нижней позицией ничего серьезного нет. Просто умножаем состояние самолета, увеличенное на 1 на высоту сплайна (90).

Теперь посмотрим, куда будет копироваться изображение:

```
RECT dst;
dst.left=iPlainLeft>0 ? iPlainLeft : 0;
dst.right=(iPlainLeft+180)<800 ? iPlainLeft+180 : 800;
dst.top=10;
dst.bottom=100;
```

Если левая позиция самолета меньше нуля, то используем нуль, т. е. самолет будет нарисован начиная с левого края окна. Иначе, используем текущее значение левой позиции `iPlainLeft`.

Попробуйте мысленно прикинуть, как определяется правая позиция. Если разберетесь, то можно считать, что с ручным отсеканием невидимых поверхностей вы разобрались.

Верхняя позиция будет равна 10 (на этой высоте будет лететь самолет), а нижняя 100, т. е. верхняя позиция (10) плюс высота картинки (90).

Как видите, ничего сложного в ручном отсечении нет. А главное, оно будет работать быстрее интерфейса `IDirectDrawClipper`, потому что вы проверяете только то, что необходимо и не затрагиваете те стороны окна, которые невозможно пересечь.

Теперь посмотрим, как происходит анимация. На каждом этапе значение левой позиции увеличивается на единицу, обеспечивая движение слева направо. Если значение этой позиции превышает ширину экрана плюс 180, т. е. самолет улетел за правую область экрана, то сбрасываем позицию в –180, чтобы начать движение заново.

После отображения самолета необходимо изменить состояние самолета, чтобы он в движении постепенно наклонялся и выравнивался за счет смены картинок состояния. Для изменения прибавляем к состоянию `iPlainState` значение `iStateOffset`. Изначально у нас значение приращения положительное. Если состояние становится больше 7, то изменяем приращение на отрицательное, чтобы теперь состояние уменьшалось, и самолет постепенно выравнивался.

Попробуйте запустить пример и посмотреть на результат работы. Такие простейшие методы анимации использовали разработчики игр лет 20 назад, но технология не устарела и при двумерном программировании она еще способна показать себя во всей своей красе.

Примечание

Исходный код примера находится в каталоге \Chapter3\Clipper на прилагаемом компакт-диске.

3.6. Оптимизация прямого доступа

Слишком частая и долгая блокировка — это не есть хорошо, и чем больше поверхности проводят времени в заблокированном состоянии, тем больше вероятность возникновения проблем. Необходимо решение, которое позволило бы использовать прямой доступ в любое время и без ограничения.

Один из возможных вариантов решения данной проблемы — выделить буфер в памяти и формировать изображение в нем. Такой буфер может иметь примерно следующий вид:

```
WORD video_buf[800][600];
```

Когда сцена сформирована, то достаточно скопировать содержимое буфера на поверхность. Буфер `video_buf` — это не поверхность, поэтому к ней мы можем обращаться когда угодно. Этот метод оптимизации блокировок мы рассмотрим в *разд. 4.3*. Сейчас же мы разберем более эффектное решение — создание поверхности на основе буфера.

Итак, создайте новый проект и добавьте к нему инициализацию DirectDraw. Теперь объявим буфер (`video_buf`), как двумерный массив размером в 800×600 (соответствует размерам экрана) и типом `WORD` (соответствует глубине цвета).

Теперь добавим глобальную переменную `buf_surface` типа `IDirectDrawSurface7`. Это поверхность, память которой будет указывать на буфер `video_buf`:

```
IDirectDrawSurface7 *buf_surface=NULL;
```

Затем посмотрим, как можно проинициализировать эту поверхность:

```
// Описание структуры для создания поверхности
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);
desc.dwFlags = DDSD_CAPS | DDSD_WIDTH | DDSD_HEIGHT |
         DDSD_LPSURFACE | DDSD_PITCH;
desc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN |
         DDSCAPS_SYSTEMMEMORY;
desc.dwWidth = iWidth;
desc.dwHeight = iHeight;
desc.lpSurface = &video_buf;
desc.lPitch  = iWidth * 2;

// Создание поверхности
ppiDD->CreateSurface(&desc, &buf_surface, 0);
```

Как всегда, перед созданием поверхности подготавливаем структуру типа DDSURFACEDESC2. В параметре dwFlags мы указываем следующие флаги:

- DDSD_CAPS — в параметре ddsCaps.dwCaps структуры DDSURFACEDESC2 будут указаны свойства поверхности. Там мы укажем свойства DDSCAPS_OFFSCREENPLAIN (поверхность не должна иметь специального назначения) и DDSCAPS_SYSTEMMEMORY (поверхность должна располагаться в системной памяти);
- DDSD_WIDTH и DDSD_HEIGHT — в параметрах dwWidth и dwHeight будут указаны размеры необходимой поверхности. В нашем случае они должны соответствовать размерам экрана;
- DDSD_LPSURFACE — в параметре lpSurface будет указатель на память, которая должна использоваться для хранения данных поверхности. В нашем случае это будет указатель на переменную video_buf. Таким образом, поверхность и переменная video_buf будут указывать на одни и те же данные, и мы сможем обращаться к данным через переменную video_buf без каких-либо блокировок;
- DDSD_PITCH — мы явно укажем размер строки через параметр lPitch и он будет равен ширине экрана. Помните, мы говорили о том, что размер строки должен быть кратным 4, чтобы видеокарта могла использовать максимальные возможности аппаратной части. У нас ширина экрана 800, а если умножить на глубину цвета, то получается 1600 байт, что кратно 4, и мы можем явно указать ширину строки, дабы DirectDraw не выдумал ничего другого.

После заполнения всех необходимых параметров структуры DDSURFACEDESC2 создаем поверхность с помощью метода CreateSurface.

Теперь посмотрим, как можно нарисовать что-нибудь на поверхности, через переменную video_buf. Рисование будет происходить по событию WM_PAINT:

```
case WM_PAINT:
 hdc = BeginPaint(hWnd, &ps);
 // TODO: Add any drawing code here...
 FillMemory(&video_buf, sizeof(video_buf), 200);

 for (int i=0; i<iWidth; i++)
 {
  int j= sin(i * 6.28 / 800)*200;
  video_buf[400-j][i] = 6553;
 }

 backsurf->BltFast(0, 0, buf_surface, 0, DDBLTFAST_WAIT);
 primsurf->Flip(NULL, DDFLIP_WAIT);
```

```
EndPaint(hWnd, &ps);
break;
```

Сначала обнуляем поверхность через заполнение буфера `video_buf` числом 200 WinAPI-функцией `FillMemory`. Затем запускается цикл от 0 до ширины экрана для рисования кривых линий, на основе синусоиды. На каждом этапе цикла мы закрашиваем в буфере одну точку.

После этого отображаем содержимое поверхности `buf_surface` на задний буфер с помощью метода `BltFast` и переключаем поверхности.

Попробуйте запустить пример и убедиться, что линии рисуются. При этом мы ни разу не блокировали поверхность, а рисовали через прямой доступ к поверхности. Просто мы знаем указатель на буфер, который используется поверхностью для хранения данных, и этот указатель всегда корректный.

Если ваш алгоритм слишком сложный и требует частой блокировки поверхности или одной блокировки, но достаточно продолжительной по времени, то стоит рассмотреть вариант, когда графика формируется в буфере, а потом копируется в поверхность, или же используйте буфер, как показано в примере, описанном ранее.

Примечание

Исходный код примера находится в каталоге \Chapter3\buffer на прилагаемом компакт-диске.

3.7. Оптимизация 3D-графики

При создании фигуры с помощью Direct3D используйте минимально необходимые размеры. Например, при создании буфера индексов `IDirect3DIndexBuffer9` каждый элемент массива может быть 16- или 32-битным. Если количество индексов не превышает 65 535, то следует использовать 16-битный массив. В данном случае переход на 32 бита будет неоправданным и понесет лишние расходы памяти, которой никогда не бывает много.

Конечно же, на скорость работы влияют и размеры экрана. Чем больше размеры, тем больше ресурсов необходимо для расчетов сцены. Получается, что и здесь мы должны находить разумную середину, т. е. получить необходимое качество изображения и при этом достаточную производительность, чтобы зритель (пользователь) не видел сильных задержек во время формирования сцены.

Минимум необходимого для приемлемого качества — разрешение 800×600, но в *разд. 3.1* мы уже определились, что лучше давать пользователю самому выбирать нужное ему разрешение. Напоминаю, что существуют разные мо-

ниторы — стандартные и широкоэкранные, где разрешение в 800×600 будет выглядеть растянутым.

Не забываем и про глубину цвета. Чтобы качество изображения было достаточным, необходимо использовать глубину цвета в 16 бит. Большие значения (24 или 32 бит) могут дать картинку более высокого качества, но производительность снизится. Посмотрите на рис. 3.3. Эту сцену мы создали в *разд. 3.5*, при этом использовали глубину 16 бит. Поскольку полиграфия не может передать качество цветной картинки, поэтому лучше запустить пример, и вы увидите, что качество 24-битной загруженной картинки не сильно испортилось. Изменения можно заметить, только если долго рассматривать каждую точку.

В Direct3D заметить потерю качества цвета будет еще сложнее, если сцена будет находиться в движении. Все мы видели фильмы на DVD, где качество картинки не соответствует 100%, но вы заметили это несоответствие? Я думаю, что нет. Потерю качества можно увидеть, только если нажать паузу в момент быстросменяющейся смены сцены, например, во время быстрого движения на автомобиле. Лучше будет, если действия будут происходить в темноте (ночью). В этом случае происходит максимальное сжатие и можно даже увидеть разводы на черном фоне ночи.

Подведем итог — для демо-роликов лучше использовать глубину цвета 16 бит и давать пользователю самому выбирать разрешение экрана, в зависимости от которого зритель сможет контролировать качество и скорость изображения.

Но все это мы уже рассматривали в *разд. 3.1*, а здесь только повторили с учетом появления на нашем горизонте третьего измерения. Теперь же поговорим о нюансах, которые касаются именно 3D-роликов.

Когда мы работали с DirectDraw, то вывод графики происходил на основе рисования растровых изображений. Чем больше этих изображений, тем дольше будет формироваться сцена, поэтому в двумерном измерении приходилось оптимизировать количество картинок.

В Direct3D мы формируем сцену с помощью вершин и треугольников. Чем больше их, тем больше времени необходимо движку и видеокарте на формирование сцены. И снова приходится выбирать между количеством и качеством. Посмотрите на рис. 3.6, где показана сфера, созданная из 32 сегментов. Сфера получается гладкой, но количество необходимых треугольников слишком велико.

Можно сократить количество сегментов до 16, т. е. уменьшить их в два раза. В этом случае сокращается количество ресурсов, необходимых для отработки фигуры, но сфера получается угловатой (рис. 3.7). Я думаю, что такое положение дел устроит далеко не многих.

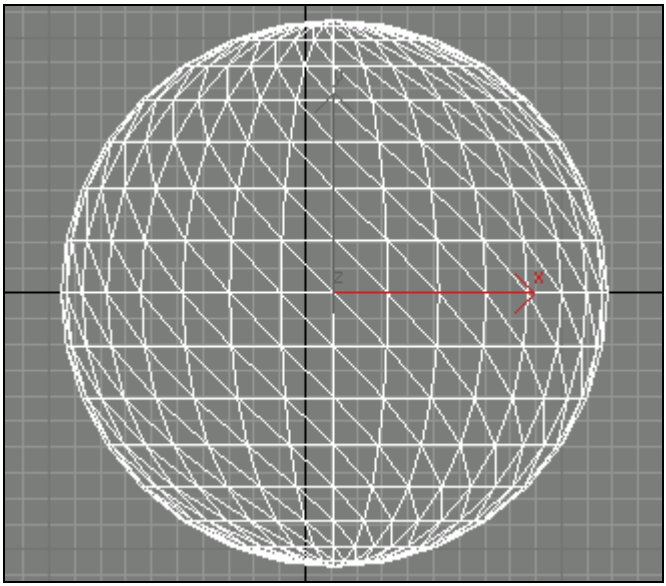

Рис. 3.6. Сфера из 32-х сегментов

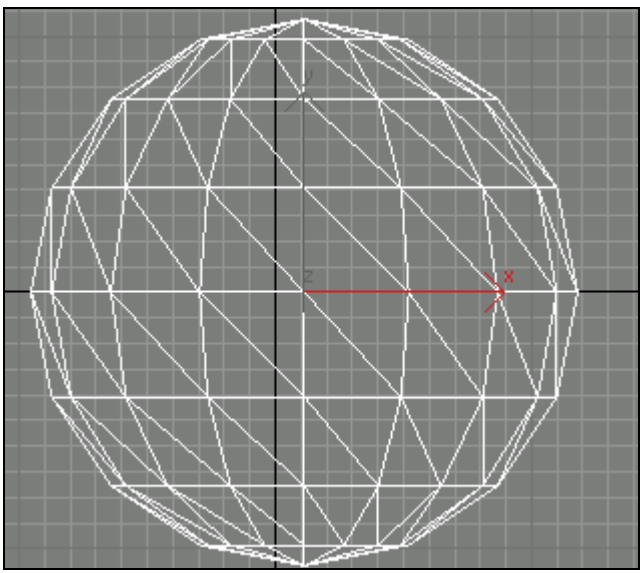

Рис. 3.7. Сфера из 16-ти сегментов

Если же нам нужны сглаженные поверхности, то приходится выбирать между гладкостью и скоростью. В большинстве игр стараются использовать прямоугольные поверхности, ведь для создания четырехугольной плоскости достаточно всего лишь двух треугольников.

Получается, что скорость создания сцены зависит не только от количества объектов на экране — объектов интерьера, освещений, существования теней. На сцену влияют и количество вершин, из которых состоит объект. Здесь необходимо подходить к решению задачи с деликатной осторожностью.

С другой стороны, если создать сферу из 100 сегментов, то это будет явно избыточно, особенно если эта сфера в сцене будет находиться очень далеко и выглядеть мелкой. Пользователь просто не сможет увидеть мелких деталей, которые вы захотите передать. Если у вас есть 3D-редактор, позволяющий контролировать количество сегментов (например, 3D Studio Max), то попробуйте сейчас создать сферу размером во весь экран. Для нормального отображения и получения гладкой поверхности необходимо не менее 30 сегментов. Но если отдалить сферу как можно дальше (чтобы она была диаметром не более десятикопеечной монеты), то будет достаточно и 12 сегментов.

Получается, что при формировании сцены мы можем использовать разное количество сегментов для объектов с разной удаленностью и значительно сэкономить жизнь процессору и видеокарте, а значит, увеличить скорость создания сцены. Мы постараемся в данной книге использовать этот эффект для повышения производительности.

Посмотрите на рис. 3.8, где показаны сферы различного размера. Самая большая сфера создана из 32 сегментов, а остальные из 12. Обратите внимание, что самая маленькая сфера выглядит вполне гладкой, хотя на тех сферах, что побольше, хорошо видны угловатости. Использовать большое количество сегментов на маленьких объектах — излишняя растрата ресурсов. Пример, который формирует эту сцену, можно найти на прилагаемом компакт-диске в каталоге /Chapter3/Sphere.

То же самое касается и текстур. Чем ближе объект, тем лучше видна каждая деталь на текстуре. На самой большой сфере рис. 3.8 можно увидеть, что текстура есть, но на самой дальней сфере вы ничего не увидите. Она настолько мала, что кажется окрашенной в один цвет. Тогда зачем использовать текстуру? Не лучше ли просто окрасить сферу в нужный цвет и пользователь ничего не заметит, особенно если сфера будет в движении.

Чтобы повысить скорость работы с текстурами, можно действовать следующим образом:

- для самых дальних объектов можно вообще не использовать текстуры, а просто окрашивать объект цветом, который будет максимально соответствовать цвету текстуры;
- по мере приближения объекта можно натянуть на него текстуру небольшого размера, например, 16×16. Таким образом, мы заранее масштабируем битовый файл, облегчая жизнь видеокарте;

- для больших объектов, находящихся поблизости, можно использовать более качественные текстуры.

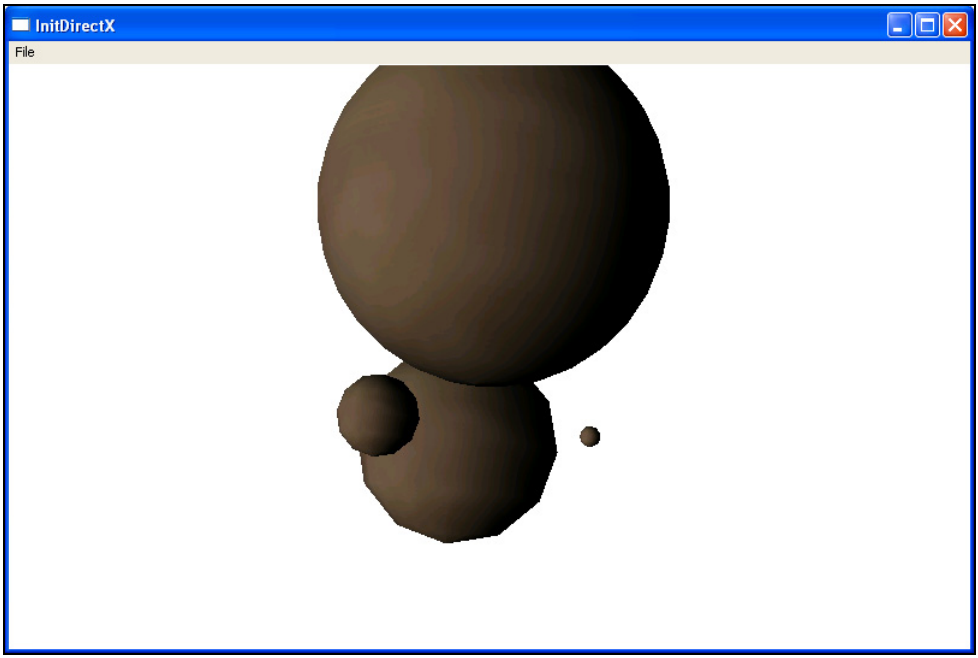

Рис. 3.8. Сферы разного размера и из разного количества сегментов

Если масштабировать текстуры заранее, то качество изображения будет лучше, чем если это сделает DirectX, потому что профессиональный художник сможет сделать необходимые сглаживания, чтобы уменьшенное изображение выглядело приемлемо. Конечно, все возможные варианты текстур подготовить невозможно, но хотя бы несколько вариантов подготовить нужно. Меньшие по размеру текстуры, используемые для дальних объектов, будут обрабатываться быстрее за счет меньшего размера и меньшего коэффициента программного масштабирования. Заранее масштабированные текстуры — это один из немногих трюков, когда мы повышаем и скорость, и качество сцены.

3.8. Функции оптимизации 3D

На большинство внутренних процессов Direct3D мы повлиять не можем, но есть кое-что, на что повлиять можно. Например, с помощью функции `OptimizeInplace` можно задать контроль отображения точек и поверхностей сеток (mesh) для оптимизации производительности. Эта функция в общем виде выглядит следующим образом:

```
HRESULT OptimizeInplace(
    DWORD Flags,
    CONST DWORD *pAdjacencyIn,
    DWORD *pAdjacencyOut,
    DWORD *pFaceRemap,
    LPD3DXBUFFER *ppVertexRemap
);
```

Рассмотрим параметры этой функции:

- `Flags` — параметры, определяющие действия, которые необходимо выполнить для оптимизации. Здесь можно указать следующие значения:
  - `D3DXMESHOPT_COMPACT` — пересортировать поверхности сетки, чтобы убрать неиспользуемые вершины и поверхности;
  - `D3DXMESHOPT_ATTRSORT` — пересортировать поверхности для уменьшения атрибутов;
  - `D3DXMESHOPT_VERTEXCACHE` — пересортировать поверхности для повышения коэффициента эффективности использования кэша вершин;
  - `D3DXMESHOPT_STRIPREORDER` — пересортировать поверхности для увеличения длины близлежащих треугольников;
  - `D3DXMESHOPT_IGNOREVERTS` — оптимизировать только поверхности, не затрагивая вершины;
  - `D3DXMESHOPT_DONOTSPLIT` — пока сортируются атрибуты, не разделять вершины, которые разделяются между группами атрибутов;
  - `D3DXMESHOPT_DEVICEINDEPENDENT` — оптимизировать размер кэша вершин;
- `pAdjacencyIn` — указатель на оптимизированный массив смежных граней;
- `pAdjacencyOut` — указатель на оптимизированный массив смежных граней;
- `pFaceRemap` — указатель на выходной буфер, куда будет помещены новые индексы для каждой поверхности;
- `ppVertexRemap` — указатель на интерфейс `ID3DXBuffer`, который будет содержать новые индексы для каждой вершины.

ГЛАВА 4

2D-эффекты

Мы уже достаточно много говорили о двумерной графике и написали несколько интересных примеров. Но все это было только вступлением к этой главе. Здесь мы будем говорить о эффектах и писать более интересные примеры.

В большинстве случаев хорошая графика связана с математикой, и тут нужно знать не просто операции сложения или вычитания, а геометрию и тригонометрию. Большинство двумерных демо-роликов, получивших популярность и завоевавших признания зрителей, основаны действительно на сложных расчетах.

Ходят легенды, что авторы иногда даже понятия не имеют, что делает алгоритм, потому что во время создания эффекта идет игра с формулами и цифрами. Если во время этой игры получилось что-то красивое, то код оформляется в эффект. Да, иногда такой способ создания эффектов действительно присутствует, но чаще всего эффект рождается в голове программиста, а потом уже переносится в редактор среды разработки в виде кода.

Точно так же некоторые музыканты создают свои шедевры. У кого-то музыка сначала рождается в голове, а потом превращается в ноты, а у кого-то идет игра нот, и если эта игра дает приятную мелодию, то появляется полноценное произведение.

Двумерные эффекты более сложные, потому что все приходится рассчитывать самостоятельно, особенно если хочется создать что-то трехмерное на плоской поверхности. Но зато именно это ценится больше всего среди профессионалов, когда удается сделать 3D-сцену без Direct3D и спрайтовой анимации.

Зрителю, не посвященному в тонкости программирования, абсолютно все равно, как вы создали эффект — с обманом или без. Главное, чтобы было красиво.

4.1. Обман зрения

Программисты очень хитрые и нередко лукавят, дабы сделать программу лучше при минимуме затрат (сил или производительности). Тут же вспоминается игра Doom, которая произвела в свое время фурор в компьютерной графике. Но если посмотреть на ее "внутренности", то вы увидите, что разработчики использовали достаточно большое количество трюков. Например, многие данные не рассчитывались, а брались приближенные значения из заранее подготовленных таблиц результатов.

То что программисты хитрят — это не плохо, а даже хорошо. Как говорится, на войне любые методы хороши, а компьютерная графика и игровая индустрия — это война производителей. В данном случае, главное, чтобы пользователи не страдали и получали продукт должного качества, а хитрости позволяют добиться этого качества.

Ну и самое главное — если вашу хитрость зритель не заметил, то это не хитрость, а отличное решение проблемы. Лично мне все равно, как в игре рассчитываются тени — по заранее подготовленным значениям или математически. Главное, чтобы тени выглядели реально, а как они реализованы, меня уже не волнует.

4.1.1. Градиент

Давайте посмотрим, как можно обмануть зрение зрителя и получить необходимый результат. В *разд. 2.5* мы создавали пример с градиентной заливкой. Для этого мы использовали математику и последовательно заполняли каждый пиксел изображения. Код получился быстрым, но очень нудным. Но есть способ проще. Допустим, что нам необходимо закрасить градиент по горизонтали. Для этого можно создать графический файл размером с экран и просто вывести его. Но это слишком расточительное использование памяти. Содержать картинку такого большого размера — неоправданные расходы. Как же тогда сэкономить?

В данном случае есть очень хитрое решение. Посмотрите на рис. 4.1, где показан классический градиент. Пусть он и черно-белый из-за полиграфии, но это не так уж и страшно, главное, что смысл сохранился. В горизонтальном градиенте все строки одинаковые. Достаточно взять только одну строку и размножить ее на весь экран. А можно и ничего не плодить, а воспользоваться функцией `Blt` и масштабировать одну строку во весь экран. Результат будет один и тот же. Таким образом, мы хитростью упрощаем код, и получаем необходимый результат.

Я думаю, что пример рассматривать не стоит, потому что код заключается в простом масштабировании картинки с помощью `Blt` и правильно подготов-

ленной картинке. Этот пример вы можете найти на прилагаемом компакт-диске.

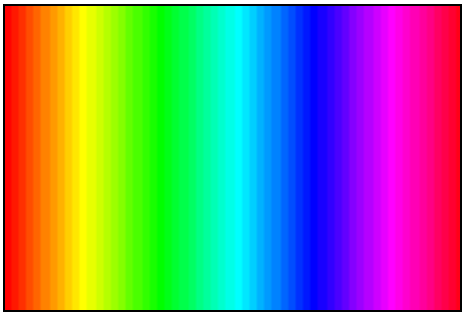

Рис. 4.1. Классический градиент

Примечание

Исходный код примера градиента находится на прилагаемом компакт-диске в каталоге \Chapter4\Gradient.

4.1.2. Мультипликация

Программирование графики с помощью сплайнов тоже является хитростью. С помощью быстрой смены изображения мы заставляем зрителя поверить, что графика живая. Точно так же анимацию создают и мультипликаторы в рисованных мультфильмах.

Спрайтовую анимацию мы сильно затрагивать больше не будем, хотя она используется в демо-роликах до сих пор. Чаще всего, битовые картинки с заранее подготовленным в Photoshop или The GIMP изображением используются в качестве логотипов или вставок простой анимации. Такими вещами сейчас мало кого удивишь, хотя, существуют красивые мультипликационные ролики а-ля Disney. Но качество и красота тут в основном зависят от мастерства художника, который рисует мультипликацию. Как мы уже видели в *разд. 3.5*, программирование спрайтовой анимации не так уж и сложно, поэтому программисту показывать свои умения тут особо негде.

4.2. Линейные эффекты

Самыми простыми, но очень эффективными являются линейные эффекты. Очень часто они основаны на фракталах, которые являются математическими формулами. Давайте посмотрим некоторые эффекты.

4.2.1. Сетка

Первый эффект, который мы построим, показан на рис. 4.2. Это самое простейшее, что можно сделать, но при этом получается достаточно эффектная картинка.

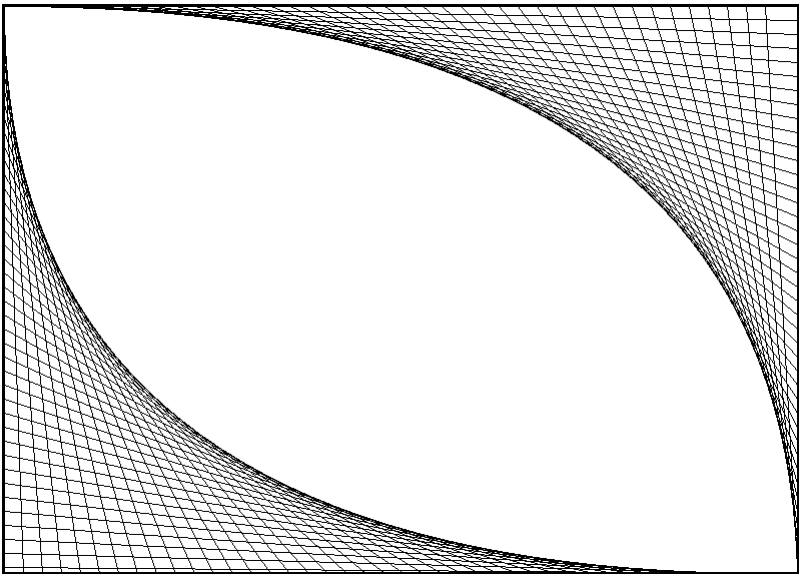

Рис. 4.2. Линейный эффект сетки

Если взглянуть на картинку издалека, то сразу и не поймешь, из чего она сделана. А ведь это всего лишь линии. Присмотритесь, и все станет ясно. Код, который реализует этот эффект, показан в листинге 4.1.

Листинг 4.1. Линейный эффект

```
void DrawScene()
{
  int lines_number=40;
  int dx_offset=iWidth/lines_number;
  int dy_offset=iHeight/lines_number;

  ClearSurface(backsurf, 0);

  for (int i=0; i<lines_number; i++)
  {
    DrawLine(backsurf, i*dx_offset, 0, iWidth, i*dy_offset, 255,0,0);
```

```
    DrawLine(backsurf, 0, i*dy_offset, i*dx_offset, iHeight, 255,0,0);
  }

  primsurf->Flip(NULL, DDFLIP_WAIT);
}
```

Здесь мы просто рисуем линии, постепенно смещая их координаты на каждом этапе цикла одновременно по горизонтали и вертикали.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter4\line_fx.

4.2.2. Спираль квадратов

Усложним задачу. Давайте создадим эффект, который показан на рис. 4.3. Рисунок упрощен и показан только в черно-белом цвете, а в реальном примере мы будем рисовать линии, плавно переходя от черного к какому-нибудь из светлых цветов. При этом фон экрана будет тоже черным, а значит, по краям линии сольются с фоном.

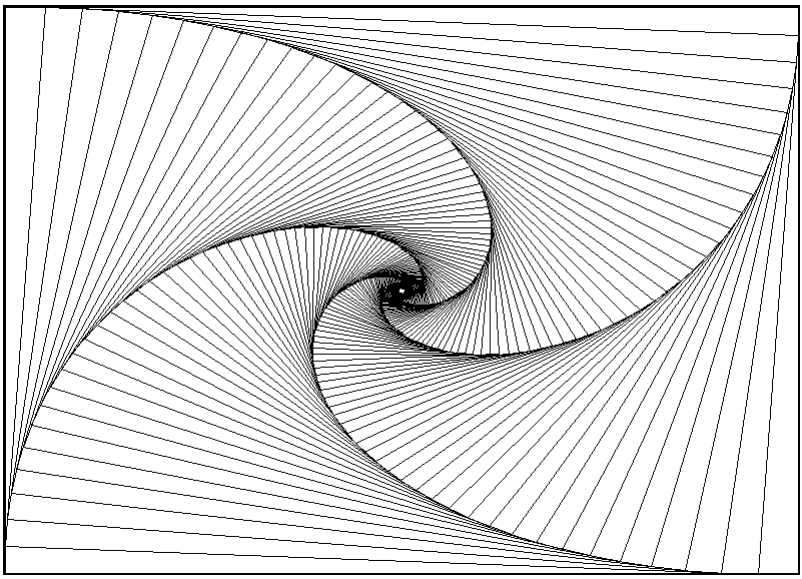

Рис. 4.3. Спираль

Код примера, который реализует спираль, показан в листинге 4.2. Код комментирован, чтобы вы разобрались с ним без дополнительных пояснений. Да

тут и нечего особо пояснять, потому что ничего, кроме математики и рисования линий, нет.

Листинг 4.2. Формирование спирали

```
void DrawScene()
{
 double x1, y1, x2, y2, x3, y3, x4, y4,
       tx1, ty1;

 // Инициализируем переменные
 double p=0.9;
 double q=0.1;
 x1=0; x2=iWidth; x3=iWidth; x4=0;
 y1=0; y2=0; y3=iHeight; y4=iHeight;

 ClearSurface(backsurf, 0);

 for (int i=0; i<100; i++)
 {
  // Рисуем квадрат
  DrawLine(backsurf, x1, y1, x2, y2, 2,i*2,i*2);
  DrawLine(backsurf, x2, y2, x3, y3, 2,i*2,i*2);
  DrawLine(backsurf, x3, y3, x4, y4, 2,i*2,i*2);
  DrawLine(backsurf, x4, y4, x1, y1, 2,i*2,i*2);

  // Расчет положения следующего квадрата
  tx1=p*x1+q*x2;
  ty1=p*y1+q*y2;
  x2=p*x2+q*x3;
  y2=p*y2+q*y3;
  x3=p*x3+q*x4;
  y3=p*y3+q*y4;
  x4=p*x4+q*x1;
  y4=p*y4+q*y1;
  x1=tx1;
  y1=ty1;
 }

 primsurf->Flip(NULL, DDFLIP_WAIT);
}
```

Алгоритм достаточно прост — рисуем квадрат и корректируем координаты каждой вершины квадрата. Для корректировки используются два коэффици-

ента p (равен 0.9) и q (равен 0.1). Эти две переменные в сумме должны быть равны единице, иначе спираль будет не ровной. Чем ближе одна из переменных к единице, а другая к нулю, тем на меньший угол будет поворот квадрата на каждом шаге.

Теперь посмотрим, как используются коэффициенты для расчета координат каждого следующего квадрата:

```
x1=p*x1+q*x2;
```

Для первой координаты X, умножаем текущее значение этой координаты на коэффициент p и прибавляем результат перемножения координаты X следующего отрезка квадрата на коэффициент q. Таким же макаром рассчитываются и Y-координаты вершин.

На первый взгляд, мы рассматриваем достаточно простые сцены. Да, используется какая-то математика, и что-то получается, но это просто. В простоте вся сила и из нее можно выжать максимум. Например, этот пример можно заставить вращать спираль прямоугольников и тогда получится совершенно другой результат.

Давайте посмотрим один из возможных примеров анимации. Для этого введем переменную off, которая будет определять угол поворота спирали. Теперь прорисовку экрана необходимо перенести в основной цикл обработки сообщений и после каждого отображения увеличивать переменную off на 0.01:

```
DrawScene();
off+=0.01;
if (off>1)
  off=0;
```

Помимо увеличения переменной off мы выполняем ее проверку, и если она превышает 1, то обнуляем ее.

Теперь переходим в функцию DrawScene. Перед циклом отображения квадратов корректируем начальное положение координат вершин с учетом текущего значения переменной off:

```
tx1=off*x1+(1-off)*x2;
ty1=off*y1+(1-off)*y2;
x2=off*x2+(1-off)*x3;
y2=off*y2+(1-off)*y3;
x3=off*x3+(1-off)*x4;
y3=off*y3+(1-off)*y4;
x4=off*x4+(1-off)*x1;
y4=off*y4+(1-off)*y1;
```

```
x1=tx1;
y1=ty1;
```

Здесь мы корректируем все вершины точно так же, как это происходит на каждом этапе цикла. Только тут используем для коэффициента одну переменную, ведь значение второго коэффициента это `1-off`. Я сделал это только для того, чтобы не заводить лишнюю глобальную переменную.

А почему же мы раньше не использовали только одну переменную? Дело в том, что коэффициенты `p` и `q`, которые используются внутри цикла, являются локальными, а мне стека не жалко, его много и после выхода из процедуры он все равно очистится. Зато, благодаря наличию двух переменных, не нужно внутри цикла при расчете каждой вершины выполнять операцию `1-коэффициент`. Таким образом, мы экономим в цикле 8 операций, а если умножить это на количество шагов цикла, то экономия составит 800 операций.

Таким образом, вне цикла корректировка вершин не очень оптимизирована, зато внутри цикла лишние потери не нужны и мы их не делаем.

Если этот пример анимировать, то создастся полное впечатление 3D-спирали. Создание 3D-сцены без использования Direct3D является великим искусством хакера и программиста демо-роликов.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter4\square_fx.

4.2.3. Прямоугольный туннель

Еще один интересный вариант создания ощущения 3D-присутствия — туннель. Посмотрите на рис. 4.4, где показан пример множества прямоугольников, один внутри другого. На первый взгляд, картинка простая. Но если заставить ее двигаться (каждый квадрат должен уменьшаться, создавая эффект отдаления каждого из прямоугольников), а между двумя прямоугольниками сделать небольшое пустое пространство, то может возникнуть реальный 3D-эффект.

Чтобы улучшить пример, можно добавить плавное изменение цвета. Чем ближе прямоугольник к центру, тем делать его темнее. Когда мы смотрим в даль, то наиболее отдаленные предметы кажутся размытыми, потому что мы не можем их видеть четко. Когда мы смотрим в неосвещенный туннель, то можем видеть только ближние предметы, а удаленные оказываются в темноте и их видно не будет.

Давайте перейдем к реализации примера. Для начала нам понадобится структура, которая будет описывать каждый прямоугольник:

```
struct QUAD
{
    WORD dwTop, dwLeft, dwRight, dwBottom;
} quad_ent;
```

В принципе можно было бы использовать тип RECT, но вдруг вы захотите наделить каждый прямоугольник своими свойствами (например, делать смещения с разной скоростью по осям или добавить яркость), тогда вам достаточно будет только расширить эту структуру и использовать новые возможности.

Помимо этого, нам понадобится константа, описывающая количество прямоугольников iQuadNumber и массив QuadList из структур QUAD:

```
const int iQuadNumber = 10;
struct QUAD QuadList[iQuadNumber];
```

Теперь нужно проинициализировать массив прямоугольников начальными значениями. Это можно сделать следующим образом:

```
for (int i=0; i<iQuadNumber; i++)
{
 QuadList[i].dwLeft =(iWidth/2)/iQuadNumber*i;
 QuadList[i].dwTop =(iHeight/2)/iQuadNumber*i;
 QuadList[i].dwRight =iWidth-QuadList[i].dwLeft;
 QuadList[i].dwBottom =iHeight-QuadList[i].dwTop;
}
```

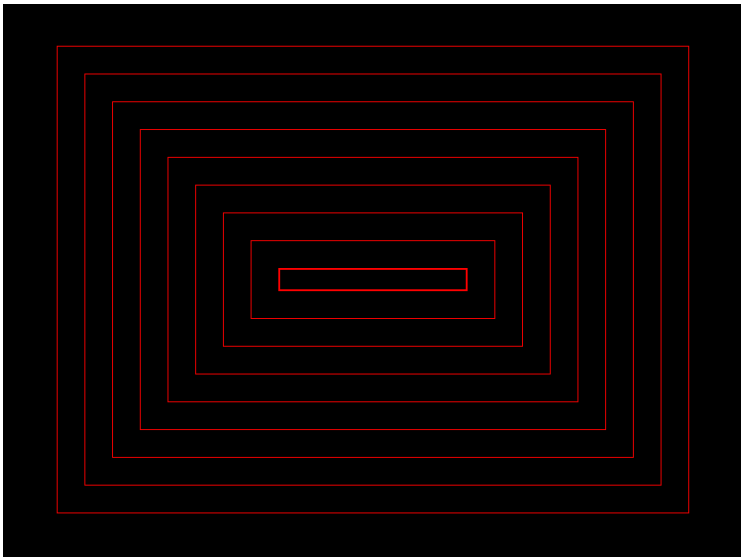

Рис. 4.4. Туннель

Данный код можно написать в любом месте, но главное, до начала отображения, т. е. до появления окна. Я рекомендую выполнить инициализацию массива прямоугольников до инициализации DirectDraw.

Отображение туннеля будем производить в цикле обработки сообщений. Соответствующий код показан в листинге 4.3.

Листинг 4.3. Код отображения прямоугольного туннеля

```
while (true)
{
 if (PeekMessage(&msg, NULL, NULL, NULL, PM_NOREMOVE))
 {
  if (!GetMessage(&msg, NULL, NULL, NULL)) break;
  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
  {
   TranslateMessage(&msg);
   DispatchMessage(&msg);
  }
 }

 if (GetActiveWindow()!=hWnd)
  continue;

 // Очищаем поверхность
 ClearSurface(backsurf, 0);

 for (int i=0; i<iQuadNumber; i++)
 {
  int r;
  if (QuadList[i].dwTop>255)
   r=0;
  else
   r=255-QuadList[i].dwTop;

  // Отображаем прямоугольник с помощью функций DrawLine
  DrawLine(backsurf, QuadList[i].dwLeft, QuadList[i].dwTop,
       QuadList[i].dwRight, QuadList[i].dwTop, r, 0, 0);
  DrawLine(backsurf, QuadList[i].dwLeft, QuadList[i].dwTop,
       QuadList[i].dwLeft, QuadList[i].dwBottom, r, 0, 0);
  DrawLine(backsurf, QuadList[i].dwLeft, QuadList[i].dwBottom,
       QuadList[i].dwRight, QuadList[i].dwBottom, r, 0, 0);
  DrawLine(backsurf, QuadList[i].dwRight, QuadList[i].dwTop,
       QuadList[i].dwRight, QuadList[i].dwBottom, r, 0, 0);
```

```
  // Масштабируем прямоугольник
  QuadList[i].dwTop++;
  QuadList[i].dwLeft++;
  QuadList[i].dwRight--;
  QuadList[i].dwBottom--;

  // Если прямоугольник слишком большой,
  // то инициализируем параметры заново
  if (QuadList[i].dwTop>iHeight/2)
  {
   QuadList[i].dwLeft =(iWidth/2)/iQuadNumber;
   QuadList[i].dwTop =(iHeight/2)/iQuadNumber;
   QuadList[i].dwRight =iWidth-QuadList[i].dwLeft;
   QuadList[i].dwBottom =iHeight-QuadList[i].dwTop;
  }
 }

 // Отображаем задний буфер на переднем
 POINT p;
 p.x=p.y=0;
 ClientToScreen(hWnd, &p);
 primsurf->BltFast(p.x, p.y, backsurf, 0, 0);
}
```

Алгоритм отображения прост до безобразия. А это безобразие заключается в простом запуске цикла, внутри которого с помощью функции `DrawLine` по линиям рисуется прямоугольник. Затем уменьшаем значения свойств структуры, описывающей этот прямоугольник, и проверяем, если размеры слишком малы, то инициализируем поля, чтобы начать движение с начала.

Очень интересно определяется цвет прямоугольника. Если высота больше 255 (это максимум, чему может быть равна одна составляющая цвета), то красную составляющую цвета обнуляем. Если меньше, то из значения 255 вычитаем значение верхней позиции.

Данный эффект легко модифицируется. Например, попробуйте убрать проверку значения верхней позиции, а определять цвет только одной строкой:

```
r=255-QuadList[i].dwLeft;
```

В результате, цвет прямоугольника будет постепенно темнеть, а в центре появятся несколько ярких прямоугольников.

В данном случае при отображении используется только красный цвет, а остальные составляющие равны нулю. Попробуйте изменить эти составляющие, чтобы получить прямоугольники другого цвета.

Если вы запустили пример, то наверно заметили, что создается ощущение движения по туннелю задним ходом. Попробуйте изменить направление, чтобы прямоугольники масштабировались не в сторону уменьшения, а в сторону увеличения.

Вот так вот из простых линий мы создали простой, но интересный эффект. А главное — применялось минимум кода, и пример можно использовать как один из эффектов в 64-килобайтном демо-ролике.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter4\quads.

4.3. Нелинейная графика

Хороший демо-ролик должен смотреться каждый раз так, как будто это совершенно новая работа. Как этого можно добиться? Самый простейший вариант — графика должна строиться нелинейно, т. е. не по заранее созданному вектору, а в зависимости от случайных величин. В этом случае в алгоритм вмешиваются случайные силы, которые делают сцену при каждом просмотре новой. Разумеется, в этом случае сложнее синхронизироваться с музыкальным сопровождением, но небольшие огрехи зритель может и не заметить или даже простить.

Действительно, когда демо-ролик каждый раз смотрится по-разному, пусть и с не большими отличиями, его можно смотреть и смотреть.

4.3.1. Фейерверк

Допустим, что вы решили добавить в свой проект фейерверк. Конечно же, привязывать его к определенному алгоритму будет глупо и без случайных величин тут не обойтись. Давайте попробуем реализовать один из возможных вариантов.

Наш фейерверк будет разбрасывать светящиеся искры из центра экрана, а эти точки в свою очередь должны крутиться по часовой стрелке вокруг центральной точки. Каждая искра фейерверка должна жить своей жизнью на основе случайных чисел, поэтому для описания искр нам понадобится структура следующего вида:

```
struct PARTICLE
{
 WORD x, y;      // Координаты искры
 double angle;   // Угол движения
```

```
 int adjust;     // Изменение угла движения
 int speed;      // Скорость движения
 BYTE r,g,b;     // Цвет
 int life;       // Время жизни искры до затухания
} particle_ent;
```

Для реализации примера нам понадобятся две переменные:

```
const int iParticleNumber = 10000;
struct PARTICLE ParticleList[iParticleNumber];
```

Переменная `iParticleNumber` определяет количество искр, которые генерирует фейерверк, а переменная `ParticleList` — массив искр.

Отображение фейерверка будет происходить в основном цикле обработки сообщений, в котором должна вызываться функция `DrawScene`. Этот метод отображения мы уже использовали не раз. Но до начала цикла нужно еще проинициализировать искры начальными значениями следующим образом:

```
for (int i=0; i<iParticleNumber; i++)
{
  ParticleList[i].speed = rand()%5 + 5;
  ParticleList[i].angle = rand()%360;
  ParticleList[i].adjust=rand()%5 + 5;
  ParticleList[i].x = iWidth/2;
  ParticleList[i].y = iHeight/2;
  ParticleList[i].life=rand()%20;
  ParticleList[i].r=255;
  ParticleList[i].g=0;
  ParticleList[i].b=0;
}
```

Кроме цвета (который изначально равен красному) и начального положения (в начале все искры идут из центра) все поля структуры заполняются на основе случайного значения, генерируемого функцией `rand`. При этом, чтобы скорость и изменение угла не было нулевым, к случайному значению прибавляем число 5, а значит, эти поля будут всегда больше 5.

Теперь посмотрим на функцию `DrawScene`, которая будет рисовать фейерверк. Эта функция показана в листинге 4.4.

Листинг 4.4. Функция отображения фейерверка

```
void DrawScene()
{
 // Подготовка к блокировке поверхности
 ClearSurface(backsurf, 0);
```

```
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);

// Блокировка поверхности
if (backsurf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
{
 BYTE* dst = (BYTE *)desc.lpSurface;

// Цикл рисования искр
for (int i=0; i<iParticleNumber; i++)
 {
  // Рассчитываем новое положение искры
  ParticleList[i].angle+=ParticleList[i].adjust;
  ParticleList[i].x += (WORD)(cos(ParticleList[i].angle/360) *
           ParticleList[i].speed);
  ParticleList[i].y += (WORD)(sin(ParticleList[i].angle/360) *
           ParticleList[i].speed);

  // Уменьшаем время жизни искры
  ParticleList[i].life--;

  // Если искра "умерла" или вышла за пределы экрана,
  // то инициализируем ее заново
  if ((ParticleList[i].life<0) ||
         (ParticleList[i].x<0) ||
         (ParticleList[i].x>=iWidth) ||
         (ParticleList[i].y<0) ||
         (ParticleList[i].y>=iHeight))
  {
   ParticleList[i].life=rand()%200;
   ParticleList[i].x=iWidth/2;
   ParticleList[i].y=iHeight/2;
  }

  // Корректируем цвет искры
  ParticleList[i].g+=ParticleList[i].adjust;
  ParticleList[i].b-=ParticleList[i].adjust;

  // Формируем цвет и отображаем
  WORD c = (WORD)(ParticleList[i].b/8 | (ParticleList[i].g/4 << 5) |
        (ParticleList[i].r/8 << 11));
  *(WORD *)(dst+ParticleList[i].y*desc.lPitch+ParticleList[i].x*2)=c;
 }
```

```
  backsurf->Unlock(0);
  primsurf->Flip(NULL, DDFLIP_WAIT);
 }
}
```

Рисование фейерверка происходит через прямой доступ к поверхности. Поскольку придется выводить на экран достаточно много точек, то блокируем поверхность в самом начале, дабы не делать это на каждом этапе цикла. Да, в этом случае поверхность будет заблокирована достаточно долго, но зато скорость работы цикла рисования повышается за счет одной блокировки, а не 10 000 (столько у нас точек, и каждую из них нужно отобразить).

После блокировки начинается цикл. Сначала мы увеличиваем угол движения искры на значение `adjust`. В зависимости от нового угла и скорости движения искры (параметр `speed`) рассчитываем новое положение.

Теперь нужно узнать — вышла ли точка за пределы экрана. Если да, то происходит инициализация точки заново. Если не сделать этого, то точка окажется не там, где мы ожидали. Дело в том, что при выводе графики через прямой доступ ни о какой автоматической проверке границ экрана не может быть и речи.

Теперь корректируем цвет точки. Он, как и угол движения, корректируется на значение `adjust`. Теперь все готово, и можно отображать точку на поверхности.

Попробуйте запустить пример и посмотреть на результат работы. Первое, что бросается в глаза — точки разбрасываются по сторонам не всегда равномерно. Конечно же, все получается красиво, но необходимо понять, откуда берется этот эффект? Попробуйте запустить программу несколько раз, и вы увидите, что каждый раз направление и форма лучей фейерверка оказывается примерно одинаковой. Проблема в том, что случайные числа, генерируемые функцией `rand`, далеки от случайности. Чтобы получить действительно более реальную случайность, некоторые программисты пишут собственные генераторы случайных чисел.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter4\Firework.

4.3.2. Кислота

Мы уже говорили в *разд. 1.1* о том, что во время создания демо-ролика программисты нередко играют с цифрами и алгоритмами. Давайте попробуем поиграть с фейерверком, который мы создали в *разд. 4.3.1*. Для начала, для

повышения производительности и сокращения времени блокировки поверхности добавим буфер, с которым и будем работать, во время формирования кадра:

```
WORD ScreenBuffer[800][600];
```

Теперь наши частицы будут сначала формироваться в этом буфере, и только потом все перенесем в задний буфер. В этом случае добавляется лишнее копирование памяти, но ведь мы можем переносить не в задний буфер, а сразу на передний, если в буфере `ScreenBuffer` будет храниться законченное изображение кадра. Но если нет особой надобности, и производительность позволяет использовать лишнее копирование, то лучше воспользоваться им. Если ролик работает в полноэкранном режиме, то за счет переключения буферов можно будет избавиться от нежелательных побочных эффектов при обращении к переднему буферу.

Теперь, после расчета каждого пиксела, цвет будет заноситься в буфер `ScreenBuffer` следующим образом:

```
ScreenBuffer[ParticleList[i].x][ParticleList[i].y]=c;
```

При этом блокировку поверхности убираем, а добавляем в конец функции `DrawScene` код из листинга 4.5.

Листинг 4.5. Копирование данных в задний буфер

```
ClearSurface(backsurf, 0);
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);

if (backsurf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
{
 BYTE* dst = (BYTE *)desc.lpSurface;

 for (int x=0; x<iWidth; x++)
  for (int y=0; y<iHeight; y++)
    *(WORD *)(dst+y*desc.lPitch+x*2) = ScreenBuffer[x][y];

  backsurf->Unlock(0);
  primsurf->Flip(NULL, DDFLIP_WAIT);
 }
```

Теперь формирование заднего буфера происходит в конце функции рисования. При этом копирование из `ScreenBuffer` в задний буфер происходит в

цикле, и копируем по 2 байта. Не будем оптимизировать этот код, оставим это в качестве домашнего задания. Если же вы не захотите выполнять это домашнее задание, то никто родителей в школу вызывать не будет ☺.

Пока что мы модифицировали уже существующий пример, но не внесли в него ничего нового. Пора бы сделать что-нибудь интересное. Один из возможных вариантов — добавить еще эффект размытия (Blur). Для этого берется вокруг среднее значения цвета 8-ми точек, окружающих искру. Следующий код показывает, как реализовать это в виде кода:

```
for (int x=2; x<iWidth-1; x++)
 for (int y=2; y<iHeight-1; y++)
 {
  ScreenBuffer[x][y] = (ScreenBuffer[x-1][y-1]
            + ScreenBuffer[x-1][y]
            + ScreenBuffer[x-1][y+1]
            + ScreenBuffer[x][y-1]
            + ScreenBuffer[x][y+1]
            + ScreenBuffer[x+1][y-1]
            + ScreenBuffer[x+1][y]
            + ScreenBuffer[x+1][y+1])/8;
  }
```

Добавьте этот код после формирования сцены в буфере `ScreenBuffer`, но до переноса данных в задний буфер.

Запустите приложение и посмотрите на результат. Я не буду давать снимка экрана, потому что это надо видеть своими глазами, да и черно-белая полиграфия в данном случае уж точно будет бессильна. Результирующий эффект чем-то похож на расплывающуюся по экрану кислоту.

Вот так немного поиграв с одним эффектом, мы получили совершенно новые ощущения. Комбинируя несколько эффектов, позволят вам иногда добиваться отличного результата.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter4\Acid.

4.4. Эффекты с изображениями

Редкий демо-ролик состоит из одной только математики, и в ней обязательно присутствуют битовые изображения. Но если вывести картинку на экран, то это будет слишком просто, поэтому и тут программисты идут на различные хитрости и трюки. Давайте посмотрим на некоторые интересные решения.

Одним из вариантов оригинально вывести картинку — отобразить ее не сразу же, а с помощью определенного эффекта. Например, можно создать эффект движения кисти по кругу, которая и будет рисовать изображение картинки. На рис. 4.5 вы можете увидеть примерный результат после прорисовки кистью одного круга.

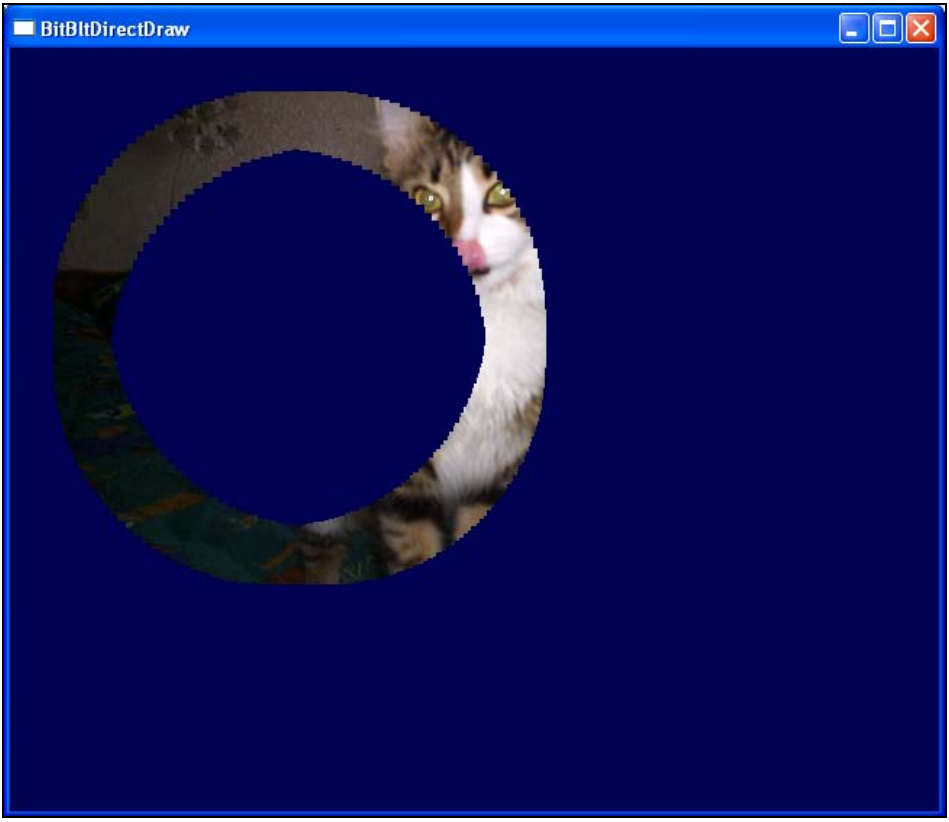

Рис. 4.5. Вывод части изображения по кругу

Теперь посмотрим, как создать подобный эффект с помощью кода. Почему подобный? Чтобы рисунок был наглядным, я уменьшил размер кисти и запустил ее ровно по кругу. Код, который мы рассмотрим, будет использовать кисть большого размера, а во время движения еще и плавное смещение вправо.

Итак, посмотрим на листинг 4.6. Дабы рисование не произошло слишком быстро, и вы насладились бы этим процессом, в данном примере кисть будет смещаться раз в 5 мс. Время мы проконтролируем с помощью функции `GetTickCount` и рисовать станем, только если с момента последней прорисов-

ки (это значение будем хранить в переменной LastTickCount) прошло 5 тиков часиков.

Листинг 4.6. Вывод изображения кистью

```
void DrawScene()
{
 // Определяем время. Если прошло более 5 мс, то рисуем
 int ThisTickCount = GetTickCount();
 if (ThisTickCount-LastTickCount>5)
 {
  // Определяем область картинки, которую нужно вывести
  SetRect(&wrkRect, 100 + (cos(Angle) * 100),
                100 + (sin(Angle) * 100),
                220 + (cos(Angle) * 100),
                220 + (sin(Angle) * 100));

  // Смещаем кисть на один пиксел вправо
  OffsetRect(&wrkRect, offset++, 0);

  // Смещаем кисть на угол 0.1
  Angle = Angle + 0.1;

  // Если пройден полный круг,
  // то уменьшаем значение угла на длину окружности
  if (Angle > 2 * 3.14)
                Angle = Angle - 2 * 3.14;

  // Выводим картинку
  backsurf->Blt(&wrkRect, imagesurf, &wrkRect, DDBLT_WAIT, NULL);

  // Отображаем результат работы
  RECT clientRect;
  POINT p;
  p.x=p.y=0;
  ClientToScreen(hWnd, &p);
  primsurf->BltFast(p.x, p.y, backsurf, 0, 0);

  LastTickCount = GetTickCount();
 }
}
```

Логика отображения проста: у нас есть две поверхности — с картинкой и задний буфер. В зависимости от текущего угла Angle мы определяем область

размером в 120×120 пикселов и переносим кусок изображения из поверхности картинки в поверхность заднего буфера.

Обратите внимание, что для отображения содержимого заднего буфера на переднем плане используется функция `BltFast`, вне зависимости от текущего режима. Таким образом, даже в полноэкранном режиме метод `Flip` не используется, хотя он и быстрее. Почему именно так? Дело в том, что при переключении экранов буферы меняются местами, а поскольку мы их не очищаем, то произойдет эффект мерцания, который будет плохо действовать на глаза. Впрочем, попробуйте воспользоваться методом `Flip` и посмотреть результат. Возможно, что вам этот эффект мерцания понравится. Главное, чтобы мерцание не было слишком долгим, иначе глаза устают. Я, конечно же, не доктор, но может быть, это даже как-то специфически воздействует на глаза или мозг.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter4\multi.

4.4.1. Прозрачность

Существует множество эффектов, с помощью которых можно отобразить картинку, и один из возможных вариантов — прозрачность. В *разд. 3.5* мы создали пример, в котором самолет движется от одного края экрана в другой. Давайте отобразим этот самолет в полупрозрачном виде, а заодно познакомимся с алгоритмом создания этого эффекта.

В рассматриваемом в этом разделе примере мы будем реализовывать прозрачность программно. Разумеется, можно было бы задействовать специальные возможности функции `Blt`, и в этом случае, и тогда отображение происходило бы на аппаратном для видеокарты уровне, но этот вариант подходит не всегда. В случае программного расчета прозрачности можно просчитать не все цветовые составляющие, а только одну или две. Оригинальным вариантом может быть отображение появляющейся картинки по каналам. Сначала нужно прогнать красный канал от полной прозрачности до 100%-й видимости, затем зеленый и, наконец, красный.

Какой бы эффект прозрачности вы не выбрали, возможности программной реализации выше. Но не стоит забывать, что скорость при этом ниже, чем у аппаратной реализации, поэтому желательно работать с небольшими картинками.

Итак, давайте возьмем пример из *разд. 3.5* и заменим метод `Blt`, отображающий самолет, на код из листинга 4.7.

Листинг 4.7. Отображение картинки с учетом прозрачности

```
if (backsurf->Lock(&dst, &desc, DDLOCK_WAIT, 0)==DD_OK)
{
 // Прозрачное копирование
 for (int j=src.top; j<src.bottom; j++)
 {
  for (int i=src.left; i<src.right; i++)
  {
   // Является ли текущая точка прозрачной
   if (baImageBits[i][j]==cImageTransparent)
    continue;

   // Определяем позицию точки в поверхности
   BYTE* wrkPointer = (BYTE *)desc.lpSurface + (j-src.top) * desc.lPitch
+
         (i-src.left) * 2;

   // Определяем каждую составляющую текущей точки поверхности
   WORD old_blue = *(WORD *)wrkPointer & 0x1f;
   WORD old_green = (*(WORD *)wrkPointer >> 5)  & 0x3f;
   WORD old_red = (*(WORD *)wrkPointer >> 11) & 0x1f;

   // Определяем каждую составляющую цвета с учетом прозрачности
   WORD blue  = old_blue+(alpha * ((baImageBits[i][j] & 0x1f) -
          old_blue) >> 8);
   WORD green = old_green+(alpha * (((baImageBits[i][j] >> 5)  & 0x3f) -
          old_green) >> 8);
   WORD red   = old_red+(alpha * (((baImageBits[i][j] >> 11) & 0x1f) -
          old_red) >> 8);

   // Устанавливаем новое значение
   *(WORD *)wrkPointer = blue | ((green) << 5) | ((red) << 11);
   }
 }
 backsurf->Unlock(0);
}
```

Конечно же, вся работа происходит через прямой доступ к поверхности, иначе просто не могло и быть. Обратите внимание, что у метода `Lock` указан первый параметр. Вспоминаем, что это — да ведь это область блокировки. Затем запускается цикл, который перебирает все видимые точки самолета.

Внутри цикла первым делом проверим, а является ли текущая точка абсолютно прозрачной. Помимо полупрозрачности и самолета есть фон, который вообще не нужно выводить. Эту проверку делаем в самом начале, чтобы цикл не выполнялся, и не тратилось лишнее время, ведь именно циклы являются самым слабым местом.

После этого определяем положение текущей точки на поверхности. Но не стоит забывать, что эта точка хранит цвет в 16-битном виде, что неудобно, потому что для расчетов приходится разбирать цвет на составляющие, а потом собирать обратно. Это лишние затраты и недостаток 16 бит, но с ними приходится мериться.

Далее идет расчет нового значения каждой составляющей цвета. Для этого используется формула:

```
Цвет=Цвет1 + (значение_прозрачности * (Цвет2 - Цвет1) >> 8;
```

В данном случае `Цвет1` — это цвет точки на поверхности, куда мы копируем, а `Цвет2` — цвет копируемой точки самолета. Рассчитав все составляющие, их можно копировать на поверхность.

И в прозрачности нет ничего сложного. Запустите пример и убедитесь, что он работает корректно. Чтобы лучше было видно, желательно, чтобы фоновая картинка была многоцветной и не однотонной, поэтому на своем изображении я нарисовал овальный круг красного цвета.

Когда самолет будет лететь, обратите внимание, что в самом начале и в конце скорость полета намного выше. Это связано с тем, что самолет выводится не полностью, а значит, меньше будет шагов у цикла и скорость его работы.

Данный пример лишний раз показал, что именно циклы являются самым слабым местом. При этом у нас получилась не линейная скорость отображения, что отрицательным образом сказывается на сцене. На лицо явная необходимость использования синхронизации отображения.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter4\Transparent.

4.4.2. Линза

Как мы уже убедились, большинство эффектов достаточно просты в реализации. Давайте посмотрим еще один простой, но достаточно красивый эффект — движение по экрану линзы, которая увеличивает изображение, находящееся под линзой.

Квадратная линза

Для примера нам понадобится всего одна поверхность, в которую будет загружена фоновая картинка. Основной код примера спрятан в основном цикле обработки сообщений, и вы можете посмотреть его в листинге 4.8. Для удобства, в листинге приведен весь код обработки сообщений.

Листинг 4.8. Рисования линзы

```
while (true)
{
 // Цикл обработки сообщений
 if (PeekMessage(&msg, NULL, NULL, NULL, PM_NOREMOVE))
 {
  if (!GetMessage(&msg, NULL, NULL, NULL)) break;
  if (!TranslateAccelerator(msg.hwnd, hAccelTable, &msg))
  {
   TranslateMessage(&msg);
   DispatchMessage(&msg);
  }
 }

 if (GetActiveWindow()!=hWnd)
  continue;

 // Отображаем фоновую картинку
 backsurf->BltFast(0, 0, imagesurf, 0, 0);

 // Подготовка структур для отображения линзы
 RECT srcRect, dstRect;
 SetRect(&srcRect, xPos-50, yPos-50, xPos+50, yPos+50);
 SetRect(&dstRect, xPos-100, yPos-100, xPos+100, yPos+100);

 // Отображение линзы
 backsurf->Blt(&dstRect, imagesurf, &srcRect, DDBLT_WAIT, 0);

 // Сдвиг линзы
 xPos=xPos+xInc;
 yPos=yPos+yInc;

 // Проверка на пересечение краев экрана
 if (xPos>iWidth-100)
  xInc=-2;
```

```
 if (yPos>iHeight-100)
  yInc=-1;
 if (xPos<50)
  xInc=2;
 if (yPos<50)
  yInc=1;

 // Отображения заднего буфера на переднем
 POINT p;
 p.x=p.y=0;
 ClientToScreen(hWnd, &p);
 primsurf->BltFast(p.x, p.y, backsurf, 0, 0);
}
```

Отображение сцены происходит после проверки на наличие в очереди сообщений и обработки сообщений. Все начинается с простого вывода фоновой картинки.

Далее, устанавливаем область, которую нужно скопировать, а также куда скопировать. Область источника определяется квадратом, размером в 100 пикселов, а квадрат приемника равен 200 пикселов. Таким образом, при копировании поверхности методу `Blt` придется масштабировать картинку, что и придаст нам эффект линзы.

Чтобы линза не просто стояла на месте, а двигалась, я добавил переменные `xPos` и `yPos` для определения текущей позиции линзы и переменные `xInc` и `yInc` для определения смещения, на которое должно происходить перемещение:

```
int xPos=100, yPos=200;
int xInc=2, yInc=1;
```

После отображения линзы текущее положение увеличивается/уменьшается на значение переменных `xInc` и `yInc`. Если после изменения позиции достигнут край экрана, то направление движения линзы изменяется на противоположное.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter4\Lens_Quad.

Круглая линза

Давайте немного усложним пример, ведь квадратная линза уж слишком проста. Как же сделать линзу лучше? Давайте оформим ее, например, в виде круга. Для этого придется немного подкорректировать пример.

Для начала введем еще одну поверхность zoomsurf, в которую и будем предварительно копировать квадратную линзу, а потом превращать ее в круглую. Как? С помощью простой формулы, которую мы рассмотрим позже, определим, какие точки квадрата выходят за пределы круга, и затем будем закрашивать их черным цветом. После этого достаточно только скопировать поверхность zoomsurf в задний буфер с учетом черного цвета — и круглая линза готова.

Инициализацию поверхности необходимо описать после инициализации DirectDraw следующим образом:

```
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);
desc.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
desc.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN;
desc.dwWidth = 200;
desc.dwHeight = 200;

ppiDD->CreateSurface(&desc, &zoomsurf, 0);
```

В структуре DDSURFACEDESC2 мы указываем, что заполнены поля флагов и размеры. Размеры поверхности соответствуют размерам квадрата описывающего линзу, а в качестве флага указываем DDSCAPS_OFFSCREENPLAIN, т. е. простая поверхность, не имеющая определенного назначения.

После создания поверхности сразу зададим черный цвет в качестве прозрачного. Черный цвет удобен тем, что ноль это всегда черный цвет, вне зависимости от выбранной глубины цвета. Итак, прозрачный цвет задаем следующим образом:

```
DDCOLORKEY cData;
cData.dwColorSpaceLowValue = 0;
cData.dwColorSpaceHighValue = 0;
zoomsurf->SetColorKey(DDCKEY_SRCBLT, &cData);
```

Теперь идем в код создания сцены. Вместо вызова метода Blt, отображающего линзу в заднем буфере, пишем следующий код:

```
RECT srcRect;
SetRect(&srcRect, xPos-50, yPos-50, xPos+50, yPos+50);
// Отображение линзы на поверхности zoomsurf
zoomsurf->Blt(0, imagesurf, &srcRect, DDBLT_WAIT, 0);

// Подготовка к блокировке поверхности
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);
```

```
// Блокировка
if (zoomsurf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
{
 // Закрашиваем все пикселы, которые не попадают в круг,
 // а сам круг остается не тронутым
 BYTE* dst = (BYTE *)desc.lpSurface;
 for (int i=0; i<200; i++)
  for (int j=0; j<200; j++)
   if (((i - 100)*(i - 100)) + ((j - 100)*(j - 100)) > 100 * 100)
      *(WORD *)(dst + j * desc.lPitch + i * 2) = 0;

 zoomsurf->Unlock(0);
}
```

Здесь мы также увеличиваем кусок рисунка в фоне, но результат прорисовывается не в заднем буфере, а на поверхности `zoomsurf`. После этого блокируем поверхность, в которой у нас находится квадратное содержимое пиксела, чтобы можно было просмотреть все пикселы и закрасить все, что не входит в округлость линзы.

Теперь посмотрим, как происходит закрашивание. Для этого запускается цикл от 0 до ширины и от нуля до высоты экрана, чтобы перебрать все пикселы поверхности. На каждом этапе проверяем попадание в окружность по следующей формуле:

```
(i-ширина)^2 + (j-высота)^2
```

Если результат вычисления больше, чем радиус окружности в квадрате (а радиус — это половина длины любой стороны квадрата, ведь все стороны равны), то данная точка выходит за пределы круга.

Вот и все. Попробуйте запустить приложение и посмотреть на результат работы. Вроде бы простой алгоритм, а как эффектно выглядит! Если еще и подобрать хорошую картинку в качестве заднего плана, то результат будет отличным.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter4\Lens_Circle.

Выпуклая линза

Просто круглая линза красива, но не естественна. Увеличительные линзы отображают картинку немного искаженно по краям окружности. Давайте попробуем реализовать нечто подобное.

Код отображения увеличительной линзы с искажением по краям можно увидеть в листинге 4.9. Это очень интересный и удачный алгоритм, который я подсмотрел в одной из демок.

Листинг 4.9. Круглая линза с искажением по краям

```
// Отображаем фоновую картинку
backsurf->BltFast(0, 0, imagesurf, 0, 0);

// Подготовка структуры для блокировки
DDSURFACEDESC2 desc;
ZeroMemory(&desc, sizeof(desc));
desc.dwSize = sizeof(desc);

// Блокировка поверхности с картинкой
if (imagesurf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
{
 // Запоминаем необходимые параметры поверхности
 BYTE* src = (BYTE *)desc.lpSurface;
 int lPitch=desc.lPitch;

 // Обнуляем структуру для блокировки
 ZeroMemory(&desc, sizeof(desc));
 desc.dwSize = sizeof(desc);

 // Блокируем поверхность заднего буфера
 if (backsurf->Lock(0, &desc, DDLOCK_WAIT, 0)==DD_OK)
 {
  BYTE* dst = (BYTE *)desc.lpSurface;
  int s=(rad*rad-circ*circ);

  // Цикл рисования линзы
  for (int y=-rad; y<=rad; y++)
  {
   int yy=y*y;
   for (int x=-rad; x<=rad; x++)
   {
    if (x*x+yy<s)
    {
     double mz = circ / sqrt(rad*rad-double(x*x+yy));
     *(WORD *)(dst+(y+yPos)*desc.lPitch+(xPos+x)*2)=
                *(WORD *)(src+(WORD)(mz*y+yPos)*lPitch+
                 (WORD)(mz*x+xPos)*2);
    }
```

```
    }
   }
   backsurf->Unlock(0);
  }
  imagesurf->Unlock(0);
}

// Определение новой позиции линзы
xPos=xPos+xInc;
yPos=yPos+yInc;
if (xPos>iWidth-100)
 xInc=-2;
if (yPos>iHeight-100)
 yInc=-1;
if (xPos<100)
 xInc=2;
if (yPos<100)
 yInc=1;

// Отображение заднего буфера на переднем
POINT p;
p.x=p.y=0;
ClientToScreen(hWnd, &p);
primsurf->BltFast(p.x, p.y, backsurf, 0, 0);
```

Сначала отображаем фоновый рисунок, поверх которого и будет бегать линза. Далее все будет происходить через прямой доступ к поверхностям. Для этого придется заблокировать поверхности заднего буфера и картинки. Из поверхности картинки мы через прямой доступ будем определять пикселы, которые нужно использовать для формирования линзы на заднем буфере.

Чтобы не расходовать лишнюю память для хранения двух структур типа `DDSURFACEDESC2`, необходимых для блокировки, будем использовать только одну. После блокирования поверхности картинки сохраняем два поля: указатель на память с видеоданными (`lpSurface`) и ширину строки (`lPitch`). После сохранения этих данных структуру можно обнулять для второй блокировки заднего буфера.

Далее следует непосредственно алгоритм расчета, для понимания которого нужно увидеть, что представляют собой переменные `rad` и `circ`. На самом деле, это константы:

```
const int rad=100;
const int circ=50;
```

Первая из них определяет радиус линзы, а вторая — это закругление линзы, от которого зависит искажение. Остальное в алгоритме — это сплошная математика, которую просто необходимо прочувствовать, и дополнительные комментарии излишни.

Оставшийся код — это просто перемещение линзы по экрану и отображение заднего буфера на экране. Все это мы уже видели в предыдущем примере, при отображении квадратной линзы.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter4\Lens_Sphere.

4.5. Фракталы

Я встречал несколько определений фракталов, например, *фрактал* — это объект, который состоит из множества маленьких объектов такой же формы. Очень часто это действительно так, но бывают случаи, когда фрактал состоит из разных объектов. В *разд. 4.2* мы играли линиями, и это тоже можно было отнести к фракталам.

Фракталы — это любимое занятие программистов. В Интернете можно найти множество ресурсов, содержащих различные формулы фракталов. Вам достаточно только реализовать эти формулы в коде, и поэтому мы опустим эту тему.

Если реализовать какой-нибудь быстрый алгоритм и добавить к нему анимацию, то получится хороший демо-ролик. Главное, чтобы прорисовка происходила в реальном времени. Можно изменять цвет фрактала или какие-то коэффициенты (такие формулы очень часто содержат множество коэффициентов), а можно заставить сцену двигаться или вращаться. Самое сложное — найти алгоритм, чтобы он был красивым, сложным и реализовывался в реальном времени. Некоторые формулы требуют больших расчетов, и на оптимизацию, как правило, уходит много времени. Но результат стоит затраченных усилий.

ГЛАВА 5

3D-эффекты

Вот теперь мы подошли к самому интересному — созданию 3D-эффектов. Именно третье измерение позволяет поразить пользователя в максимальной степени. Когда еще не было таких удачных библиотек, как OpenGL и DirectX, создание третьего измерения было вообще шедевром, и если демо-ролик содержал большое количество таких эффектов, то вероятность его победы на конкурсах повышалась.

С помощью современных 3D-библиотек создание третьего измерения упростилось, и мы уже убедились в этом. И все же, хороший 3D-эффект не понизился в цене у зрителя. Даже простые с точки зрения программирования, но впечатляющие с точки зрения зрителя эффекты могут подарить демо-ролику большую популярность, а авторам всемирное признание.

Если посмотреть на тенденцию развития Demo, то в настоящее время наиболее популярными становятся ролики, где массово происходит игра со светом и тенями. А если к этому добавляется прозрачность, то эффект становится еще лучше.

Итак, давайте перейдем к наиболее интересной, на мой взгляд, теме — 3D. Недаром этой теме выделена самая большая по объему часть книги. Изучать эту тему начнем с простых эффектов, и постепенно будем двигаться к более сложным примерам и алгоритмам. Надеюсь, что решения, рассмотренные в этой главе, пригодятся вам в будущем.

5.1. Альфа-смешивание

Одним из эффектов, способных положительно повлиять на восприятие пользователем сцены, является прозрачность. Если сделать это красиво, то зритель всегда оценит это. Давайте попробуем создать два вращающихся объекта: треугольник и квадрат, при этом цвет их будет смешиваться в тех облас-

тях, где фигуры перекрывают друг друга. На рис. 5.1 показан пример сцены, которая должна получиться в результате этого труда.

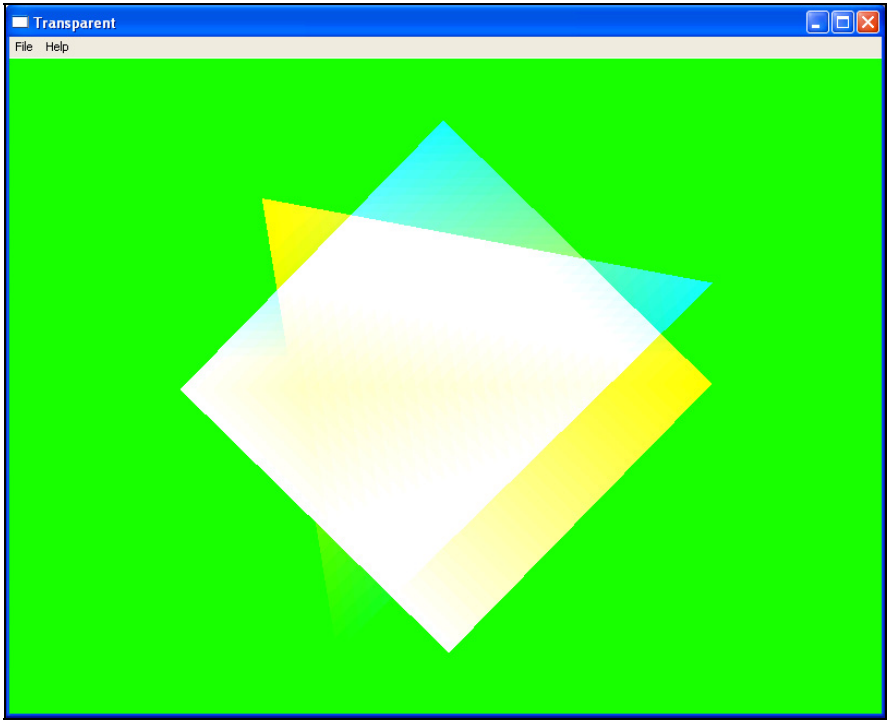

Рис. 5.1. Будущая сцена

Создайте новое приложение и добавьте инициализацию Direct3D. Сцену мы будем формировать через буфер вершин, поэтому нам понадобится структура, описывающая формат вершины, массив вершин и буфер `IDirect3DVertexBuffer9`:

```
struct sVertex
{
  float x, y, z;
  DWORD color;
};

sVertex Triangle[7];
IDirect3DVertexBuffer9 * vTriangleBuffer;
```

Обе фигуры будем хранить в одном буфере. Для отображения треугольника нам понадобятся три вершины, а для прямоугольника достаточно четырех вершин, из которых можно будет сформировать два треугольника.

После инициализации Direct3D заполняем вершины значениями, создаем интерфейс `IDirect3DVertexBuffer9` и переносим значения вершин в созданный буфер. Код инициализации буфера показан в листинге 5.1.

Листинг 5.1. Инициализация и заполнение буфера вершин

```
// Заполняем массив вершин значениями
// Первые три вершины - треугольник
Triangle[0].x=0.7f;
Triangle[0].y=-0.3f;
Triangle[0].z=0.1f;

Triangle[1].x=-0.7f;
Triangle[1].y=-0.3f;
Triangle[1].z=0.1f;

Triangle[2].x=0.0f;
Triangle[2].y=0.7f;
Triangle[2].z=0.1f;

// Следующие четыре вершины - прямоугольник
Triangle[3].x=-0.5f;
Triangle[3].y=-0.5f;
Triangle[3].z=0.1f;

Triangle[4].x=-0.5f;
Triangle[4].y=0.5f;
Triangle[4].z=0.1f;

Triangle[5].x=0.5f;
Triangle[5].y=0.5f;
Triangle[5].z=0.1f;

Triangle[6].x=0.5f;
Triangle[6].y=-0.5f;
Triangle[6].z=0.1f;

// Заполняем вершины цветами
Triangle[0].color = D3DCOLOR_XRGB(0, 0, 255);
Triangle[1].color = D3DCOLOR_XRGB(0, 255, 0);
Triangle[2].color = D3DCOLOR_XRGB(255, 0, 0);
Triangle[3].color = D3DCOLOR_XRGB(255, 0, 255);
Triangle[4].color = D3DCOLOR_XRGB(0, 255, 255);
```

```
Triangle[5].color = D3DCOLOR_XRGB(255, 255, 0);
Triangle[6].color = D3DCOLOR_XRGB(255, 255, 255);

// Создаем интерфейс буфера вершин
void * pBuf;
if (FAILED(pD3DDevice->CreateVertexBuffer(sizeof(sVertex) * 7,
    D3DUSAGE_WRITEONLY, D3DFVF_XYZ | D3DFVF_DIFFUSE, D3DPOOL_DEFAULT,
    &vTriangleBuffer, 0)))
  return FALSE;

// Блокировка и заполнение вершинного буфера
if (FAILED(vTriangleBuffer->Lock(0, sizeof(sVertex) * 7, &pBuf, 0)))
 return FALSE;
memcpy(pBuf, Triangle, sizeof(sVertex) * 7);
vTriangleBuffer->Unlock();

// Функция SetView задает матрицу отображения
SetView();
```

Первые три вершины в массиве `Triangle` описывают треугольник. Остальные четыре — это два треугольника, образующие прямоугольник. При этом треугольники рисуются по вершинам 3, 4, 5 и 3, 5, 6. Далее задается цвет вершин. Обратите внимание, что цвет каждой вершины разный. Это придаст примеру дополнительную эффектность.

Далее создается интерфейс вершин с помощью метода `CreateVertexBuffer`. Для заполнения этого буфера блокируем буфер и переносим в него значения вершин из массива `Triangle` с помощью WinAPI-функции `memcpy`.

В самом конце вызывается функция `SetView`. С ней мы уже познакомились в *главе 2*, но я напомню, как она выглядит, потому что в этой главе мы также будем с ней работать и не раз:

```
void SetView()
{
  D3DMATRIX View = {
    1, 0, 0, 0,
    0, 1, 0, 0,
s    0, 0, 1, 0,
    0, 1, 5, 1,
  };
  pD3DDevice->SetTransform(D3DTS_VIEW, &View);
}
```

Отображение сцены будет происходить по событию WM_PAINT:

```
case WM_PAINT:
 RECT  r;
 SetRect(&r, 0, 0, iWidth, iHeight);
 pD3DDevice->Clear(1, (D3DRECT*)&r, D3DCLEAR_TARGET,
     D3DCOLOR_XRGB(ViewAngle,255, 0), 1.0, 0);

 pD3DDevice->BeginScene();
 DrawScene();
 pD3DDevice->EndScene();

 pD3DDevice->Present(NULL, NULL, 0, NULL);
 break;
```

Этот код нам также знаком по *разд. 2.10* и не раз будет еще использоваться, поэтому вспомним, что здесь происходит. Сначала сцена очищается с помощью метода Clear. В данном примере мы будем рассматривать альфа-смешивание, при котором цвет фона должен быть темным. Если вы выберете белый цвет, то после смешения фигуры будут белыми и сольются с фоном. Между вызовами методов BeginScene и EndScene вызывается функция DrawScene, в которой и происходит формирование сцены.

Листинг 5.2. Функция формирования сцены DrawScene

```
void DrawScene()
{
 // Включаем альфа-смешивание
 pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, 1);
 pD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
 pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);

 // Поворачиваем треугольник
 float b = float((ViewAngle++)%0x1111)/24;

 D3DMATRIX World = {
    cos(b), -sin(b), 0, 0,
    sin(b), cos(b), 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1,
 };
 pD3DDevice->SetTransform(D3DTS_WORLD, &World);
```

```
 // Устанавливаем вершинный формат
 pD3DDevice->SetStreamSource(0, vTriangleBuffer, 0, sizeof(sVertex));
 pD3DDevice->SetVertexShader(NULL);
 pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE);

 // Отображаем треугольник
 pD3DDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);

 // Поворачиваем прямоугольник
 D3DMATRIX World1 = {
    cos(-b), -sin(-b), 0, 0,
    sin(-b), cos(-b), 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1,
 };
 pD3DDevice->SetTransform(D3DTS_WORLD, &World1);

 // Отображаем прямоугольник
 pD3DDevice->DrawPrimitive(D3DPT_TRIANGLEFAN, 3, 2);

 // Отключаем альфа-смешивание
 pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, DWORD(FALSE));
}
```

В самом начале с помощью метода `SetRenderState` устанавливаем параметры формирования сцены, а именно включаем альфа-смешивание. Для этого в качестве первого параметра метода `SetRenderState` указываем значение `D3DRS_ALPHABLENDENABLE`, а для включения смешения во втором параметре указываем 1.

После разрешения смешивания цветов необходимо указать, как это должно происходить. Для этого нужно установить параметры `D3DRS_SRCBLEND` (источник смешения) и `D3DRS_DESTBLEND` (приемник смешения) с помощью метода `SetRenderState`. В нашем случае вершины описаны только с помощью трех составляющих цвета и канал Alpha не использовался, поэтому в качестве примера параметрам `D3DRS_SRCBLEND` и `D3DRS_DESTBLEND` устанавливаем значение `D3DBLEND_ONE`. При этом степень смешивания равна единице. Если какой-то объект сцены не должен быть прозрачным, то можно указать значение `D3DBLEND_ZERO`, т. е. смешения не будет.

Если в качестве смешения используется значение `D3DBLEND_ONE`, то в смешении цветов участвует и фон, поэтому он не должен быть белым. Чтобы эффект был более красивым, во время очистки буфера, в данном примере цвет

фона должен изменяться от черного к красному. Это, безусловно, добавит примеру большей привлекательности.

Теперь можно отображать сцену. Для этого сначала поворачиваем мировую матрицу по часовой стрелке и устанавливаем буфер вершин:

```
pD3DDevice->SetStreamSource(0, vTriangleBuffer, 0, sizeof(sVertex));
pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE);
```

В первой строке устанавливаем буфер вершин. Индексы мы не используем, как и в *разд. 2.10*, поэтому их устанавливать не нужно. Во второй строке указываем формат вершины, чтобы Direct3D знал, в каком формате мы сформировали буфер и что указано.

Теперь можно переходить к отображению фигур. Они у нас находятся в буфере вершин, начиная с нулевой вершины, поэтому отображение будет выглядеть следующим образом:

```
pD3DDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
```

Здесь мы вызываем метод `DrawPrimitive` для отображения примитивов. В качестве базового будет использоваться список треугольников, потому что первый параметр равен `D3DPT_TRIANGLELIST`. Рисовать нужно начиная с нулевой вершины буфера, и всего один треугольник, для которого достаточно трех вершин, и только они будут использоваться. Все остальные вершины в буфере останутся пока не тронутыми.

Теперь поворачиваем мировую матрицу против часовой стрелки. Для этого при указании угла поворота используем то же значение, только отрицательное, и затем можно рисовать прямоугольник:

```
pD3DDevice->DrawPrimitive(D3DPT_TRIANGLEFAN, 3, 2);
```

Здесь мы выводим фигуру, в качестве базы у которой стоят треугольники со смежной стороной (`D3DPT_TRIANGLEFAN`). Прямоугольник в буфере описывается начиная с третьей вершины, поэтому второй параметр равен 3, а для формирования прямоугольника нужно два треугольника, поэтому последний параметр равен 2.

После отображения сцены отключаем смешение, установив параметру `D3DRS_ALPHABLENDENABLE` значение 0 с помощью метода `SetRenderState`.

Попробуйте поиграть смешением и параметрами смешения. Например, в качестве значения прозрачности для источника и приемника можно указать `D3DBLEND_SRCCOLOR`.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Alpha.

5.2. Управление прозрачностью

В *разд. 5.1* мы рассматривали, как работать с прозрачностью. Но мы слишком мало ею управляли, потому что невозможно было задать прозрачность определенной точки. Давайте исправим эту ситуацию.

Если вспомнить, как задается цвет в DirectX, то в 32-битном виде он будет выглядеть как ARGB, т. е. имеет четыре составляющие: степень прозрачности (A) и три значения, задающие цвет (красный — R, зеленый — G и голубой — B). Каждая составляющая отнимает по 8 бит. Мы же задавали цвет в формате XRGB с помощью функции `D3DCOLOR_XRGB`. Эта функция создает цвет из трех составляющих, а в качестве прозрачности задается значение 255, т. е. абсолютно не прозрачный цвет вершины.

Чтобы задать прозрачность, необходимо воспользоваться функцией `D3DCOLOR_ARGB`. Она получает в качестве параметров четыре значения: прозрачность, красный, зеленый и голубой цвет. Прозрачность задается в диапазоне от 0 (полностью прозрачен) до 255 (не прозрачен). Давайте зададим у нашего прямоугольника цвета с учетом прозрачности, т. е. у вершин с индексами от 3 до 6 цвет будет задаваться функцией `D3DCOLOR_ARGB`:

```
Triangle[3].color = D3DCOLOR_ARGB(0, 255, 0, 255);
Triangle[4].color = D3DCOLOR_ARGB(0, 0, 255, 255);
Triangle[5].color = D3DCOLOR_ARGB(200, 255, 255, 0);
Triangle[6].color = D3DCOLOR_ARGB(255, 255, 255, 255);
```

Первые две вершины будут абсолютно прозрачны, потому что у них первый параметр функции `D3DCOLOR_ARGB` равен нулю. У остальных точек прозрачность равна 200 и 255 соответственно.

Теперь, чтобы сцена отображалась корректно с учетом заданной прозрачности, необходимо указать значения свойств источника и приемника прозрачности. Параметру источника `D3DRS_SRCBLEND` зададим прозрачность `D3DBLEND_SRCALPHA`, а `D3DRS_DESTBLEND` укажем `D3DBLEND_INVSRCALPHA`:

```
pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, 1);
pD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
```

Теперь запустите пример и посмотрите на результат работы. Треугольник больше не будет прозрачным, потому что для него степень прозрачности не задается, а вот прямоугольник будет выглядеть достаточно красиво, потому что две вершины полностью прозрачны, а другие нет, и произойдет плавный переход.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Alpha1.

5.3. Экранные координаты

Прежде чем двигаться дальше, давайте затронем тему координат. До сих пор мы использовали мировые координаты, и Direct3D определял положения вершин на основе матриц перемещений, которые мы задавали. Но это не всегда удобно, бывают случаи, когда нужно отобразить объект, на который не будут действовать матрицы, и у нас есть такая возможность.

Интерфейс Direct3D поддерживает два типа формата — *непреобразованный* и *преобразованный*. В первом случае Direct3D подразумевает, что координаты вершин передаются без учета их реального положения на экране и для определения местоположения необходимо воспользоваться преобразованиями матрицами. Именно этот формат мы применяли ранее.

В случае использования преобразованного формата — Direct3D воспринимает координаты вершины в экранных координатах и дополнительных преобразований установленными матрицами не происходит. В данном случае все расчеты по местоположению ложатся на вас.

Если координаты вершин задаются в преобразованном формате `D3DFVF_XYZRHW` (в объявление добавляется еще одно значение — `RHW`), то Direct3D воспринимает координаты как есть. В этом случае значения должны задаваться в экранных координатах, т. е. левая верхняя точка на экране будет иметь координату 0:0, а матрицы преобразования действовать не будут.

Давайте нарисуем треугольник и прямоугольник, которые мы рассматривали в *разд. 5.2* в экранных координатах. Для этого сначала необходимо подкорректировать структуру, которая описывает вершины следующим образом:

```
struct sVertex
{
  float x, y, z, RHW;
  DWORD color;
};
```

Теперь вершина описывается не только координатами и цветом, но и дополнительным значением `RHW`.

Яркой демонстрацией использования такого примера является создание фона. Например, если необходимо нарисовать в фоне определенный рисунок, то можно создать прямоугольник в экранных координатах и натянуть на него

текстуру с нужным рисунком. Но это уже отдельная песня и танцы с бубнами, для озвучивания которых необходимо познакомиться с текстурами.

Теперь посмотрим на инициализацию вершин (листинг 5.3). Количество вершин и объявление массива `Triangle` не изменяется, да и буфер вершин `vTriangleBuffer` имеет все тот же тип интерфейса `IDirect3DVertexBuffer9`.

Листинг 5.3. Инициализация вершин в экранных координатах

```
Triangle[0].x=600.0f;
Triangle[0].y=400.0f;
Triangle[0].z=0.0f;

Triangle[1].x=200.0f;
Triangle[1].y=400.0f;
Triangle[1].z=0.0f;

Triangle[2].x=400.0f;
Triangle[2].y=100.0f;
Triangle[2].z=0.0f;

Triangle[3].x=500.0f;
Triangle[3].y=400.0f;
Triangle[3].z=0.0f;

Triangle[4].x=300.0f;
Triangle[4].y=400.0f;
Triangle[4].z=0.0f;

Triangle[5].x=300.0f;
Triangle[5].y=200.0f;
Triangle[5].z=0.0f;

Triangle[6].x=500.0f;
Triangle[6].y=200.0f;
Triangle[6].z=0.0f;

Triangle[0].color = D3DCOLOR_XRGB(0, 0, 255);
Triangle[1].color = D3DCOLOR_XRGB(0, 255, 0);
Triangle[2].color = D3DCOLOR_XRGB(255, 0, 0);
Triangle[3].color = D3DCOLOR_XRGB(100, 100, 100);
Triangle[4].color = D3DCOLOR_XRGB(100, 100, 100);
Triangle[5].color = D3DCOLOR_XRGB(100, 100, 100);
Triangle[6].color = D3DCOLOR_XRGB(100, 100, 100);
```

```
Triangle[0].RHW=1.0f;
Triangle[1].RHW=1.0f;
Triangle[2].RHW=1.0f;
Triangle[3].RHW=1.0f;
Triangle[4].RHW=1.0f;
Triangle[5].RHW=1.0f;
Triangle[6].RHW=1.0f;

void * pBuf;
if (FAILED(pD3DDevice->CreateVertexBuffer(sizeof(sVertex) * 7,
   D3DUSAGE_WRITEONLY, D3DFVF_XYZRHW | D3DFVF_DIFFUSE, D3DPOOL_DEFAULT,
   &vTriangleBuffer, 0)))
  return FALSE;

// Заполнение вершинного буфера
if (FAILED(vTriangleBuffer->Lock(0, sizeof(sVertex) * 7, &pBuf, 0)))
 return FALSE;
memcpy(pBuf, Triangle, sizeof(sVertex) * 7);
vTriangleBuffer->Unlock();
```

Посмотрите, какие значения заносятся в координаты X и Y вершин. Это положение точек в окне в экранных координатах. Наш экран имеет размер 800×600, а значит, на экране будут видны точки, координаты X которых изменяются в пределах от 0 до 800, а Y от 0 до 600.

Обратите внимание, что Z-координата осталась меньше 1. При использовании преобразованных вершин координата Z определяет значение глубины пиксела в Z-буфере, а не Z-положение координаты. Это очень важно, потому что в преобразованных вершинах не нужна Z-координата. Как мы уже поняли, в этом формате положение определяется в экранных координатах, а экран монитора пока что у нас двумерен.

После задания координат и цвета задается значение для поля `RHW`. В большинстве приложений разработчики просто задают этому полю значении 1. Но что это такое? Это число, которое изменяется от 0 до 1 и чаще всего представляет собой расстояние от точки просмотра до вершины. Для рисования фонового объекта, который будет позади всех, установим в `RHW` значение 1.0.

Теперь, когда мы создаем буфер вершин с помощью метода `CreateVertexBuffer`, в качестве третьего параметра (формат вершины) указываем флаги `D3DFVF_XYZRHW` и `D3DFVF_DIFFUSE`. Флаг `D3DFVF_XYZRHW` сообщает, что точка описывается не только координатами X, Y и Z, но и `RHW`-значением.

Заполнение буфера вершин значениями из массива `Triangle` происходит точно так же, как и при использовании непреобразованных вершин. Точно так же блокируем буфер и копируем в его память содержимое массива.

Переходим к отображению. Тут также отличий нет и нужно использовать уже знакомый нам метод `DrawPrimitive`. Но перед этим необходимо сообщить устройству, что буфер вершин имеет формат `XYZRHW`. Для этого, с помощью метода `SetFVF` задаем флаги `D3DFVF_XYZRHW` и `D3DFVF_DIFFUSE` следующим образом:

```
pD3DDevice->SetFVF(D3DFVF_XYZRHW | D3DFVF_DIFFUSE);
```

В остальном, код отображения объектов в экранных координатах ничем не отличается от отображения объектов на основе непреобразованных вершин.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\RWH.

5.4. Эффекты размытия

Когда объекты движутся слишком быстро, то возникает эффект размытия. Это связано с тем, что зрение человека не успевает переключиться на новое положение объекта. Мы якобы видим очертание объекта в точке `X1`, а он уже сместился в точку `X2`. В воздухе как бы образуется движущийся за объектом шлейф.

Зная, как работать с прозрачностью, создать движущийся шлейф не так уж и сложно. Давайте рассмотрим этот эффект на примере вращающегося треугольника. Для реализации примера нам понадобится массив из 12 вершин:

```
sVertex Triangle[12];
```

После инициализации Direct3D проинициализируем вершины координатами и цветами, как это показано в листинге 5.4, и, конечно же, создадим буфер `IDirect3DVertexBuffer9` и скопируем в него вершины треугольников.

Листинг 5.4. Инициализация массива и буфера вершин

```
// Инициализация Direct3D
if (DX3DInit(&pD3D, &pD3DDevice, hWnd, iWidth, iHeight, FALSE)!=S_OK)
{
 MessageBox(hWnd, "Ошибка инициализации DirectX!", "Error", MB_OK);
 return FALSE;
}

// Заполнение координат
Triangle[0].x=Triangle[3].x=Triangle[6].x=Triangle[9].x=0.7f;
Triangle[0].y=Triangle[3].y=Triangle[6].y=Triangle[9].y=-0.3f;
Triangle[0].z=Triangle[3].z=Triangle[6].z=Triangle[9].z=0.1f;
```

```
Triangle[1].x=Triangle[4].x=Triangle[7].x=Triangle[10].x=-0.7f;
Triangle[1].y=Triangle[4].y=Triangle[7].y=Triangle[10].y=-0.3f;
Triangle[1].z=Triangle[4].z=Triangle[7].z=Triangle[10].z=0.1f;

Triangle[2].x=Triangle[5].x=Triangle[8].x=Triangle[11].x=0.0f;
Triangle[2].y=Triangle[5].y=Triangle[8].y=Triangle[11].y=0.7f;
Triangle[2].z=Triangle[5].z=Triangle[8].z=Triangle[11].z=0.1f;

// Заполнение цветов вершин
Triangle[0].color = Triangle[1].color = Triangle[2].color =
   D3DCOLOR_ARGB(50, 255, 255, 255);
Triangle[3].color = Triangle[4].color = Triangle[5].color =
   D3DCOLOR_ARGB(100, 200, 255, 200);
Triangle[6].color = Triangle[7].color = Triangle[8].color =
   D3DCOLOR_ARGB(200, 100, 255, 100);
Triangle[9].color = Triangle[10].color = Triangle[11].color =
   D3DCOLOR_ARGB(255, 0, 255, 0);

void * pBuf;
if (FAILED(pD3DDevice->CreateVertexBuffer(sizeof(sVertex) * 12,
     D3DUSAGE_WRITEONLY, D3DFVF_XYZ | D3DFVF_DIFFUSE,
     D3DPOOL_DEFAULT, &vTriangleBuffer, 0)))
 return FALSE;

// Заполнение вершинного буфера
if (FAILED(vTriangleBuffer->Lock(0, sizeof(sVertex) * 12, &pBuf, 0)))
  return FALSE;
memcpy(pBuf, Triangle, sizeof(sVertex) * 12);
vTriangleBuffer->Unlock();

// Задание матрицы просмотра
SetView();
```

Обратите внимание, что координаты вершин 0, 3, 6 и 9 одинаковы. Точно так же одинаковы координаты вершин 1, 4, 7, 10 и 2, 5, 8, 11. Таким образом, у нас получается четыре треугольника с одинаковыми координатами.

После задания координат необходимо указать цвета вершин. Вот тут происходит самое интересное. Вершины первого треугольника окрашиваем в белый цвет и устанавливаем им прозрачность равную 50. У следующего треугольника у вершин красную и голубую составляющие делаем светлее (равными 200), а зеленую оставляем не тронутой. Таким образом, треугольник получает светлый оттенок зеленого, и прозрачность 100. Следующий тре-

угольник будет еще зеленее и прозрачность равна 200. Последний треугольник абсолютно зеленый и непрозрачный.

Таким образом, каждый последующий треугольник будет зеленее и менее прозрачным. Когда мы будем рисовать треугольники, то самый верхний будет зеленым и не прозрачным, а все последующие (ниже лежащие) будут постепенно светлеть и будут все больше просвечиваться.

Теперь посмотрим, как происходит вывод треугольников. В листинге 5.5 показан код формирования сцены.

Листинг 5.5. Рисование треугольника с шлейфом

```
void DrawScene()
{
 // Включаем прозрачность
 pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, 1);
 pD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
 pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

 // Определяем положение треугольника
 double b = float((ViewAngle++)%0x1111)/16;

 // Вершинный формат
 pD3DDevice->SetStreamSource(0, vTriangleBuffer, 0, sizeof(sVertex));
 pD3DDevice->SetVertexShader(NULL);
 pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE);

 // Отображение треугольников
 SetWorldView(b);
 pD3DDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
 SetWorldView(b+0.05);
 pD3DDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 3, 1);
 SetWorldView(b+0.1);
 pD3DDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 6, 1);
 SetWorldView(b+0.15);
 pD3DDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 9, 1);

 // Отключаем прозрачность
 pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, 0);
}
```

Эта функция будет вызываться по событию `WM_PAINT`, мы этот метод использовали уже не раз.

В самом начале функции включаем альфа-канал, чтобы Direct3D учитывал прозрачность треугольников. После этого устанавливаем вершинный буфер, в котором хранятся все треугольники.

Теперь все готово к отображению 4-х треугольников с помощью метода `DrawPrimitive`. Но перед отображением каждого треугольника вызывается функция `SetWorldView`, которая выглядит следующим образом:

```
void SetWorldView(double b)
{
  D3DMATRIX World = {
    cos(b), -sin(b), 0, 0,
    sin(b), cos(b), 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1,
  };
  pD3DDevice->SetTransform(D3DTS_WORLD, &World);
}
```

Эта функция получает в качестве параметра угол, на который нужно повернуть мировую матрицу.

Таким образом, перед рисованием каждого следующего треугольника мы поворачиваем матрицу на 0.05 градуса. В результате получается четыре треугольника, каждый из которых немного смещен, а цвет и прозрачность становится все ярче и ярче. Самый верхний (последний треугольник, который мы отображаем) имеет чисто зеленый цвет и непрозрачный.

Запустите этот пример, и вы увидите вращение зеленого треугольника, за которым бежит шлейф (рис. 5.2).

Вот таким вот нехитрым способом мы создали хороший эффект быстрого движения. Можно было обойтись и без прозрачности, а только постепенно делать более яркий цвет, но результат оказался бы не таким эффектным. Прозрачность хоть и отнимает лишние ресурсы процессора, но это стоит того.

В данном примере мы рассмотрели эффект размытия вращения. Если объект движется линейно, без вращения, то каждый следующий треугольник рисовался бы на несколько единиц сдвинутым в противоположную сторону от направления движения. В случае с одновременным движением и вращением необходимо и сдвигать объект, и вращать.

В свойствах мыши ОС Windows вы также можете включить отображение шлейфа, и ОС действует тем же самым методом. Но не стоит злоупотреблять размытием. Если объект движется медленно, то размытия не нужны. В данном случае этот эффект может наоборот навредить восприятию сцены. Если вы играете в игры (я, например, делаю это очень редко, потому что не хватает

времени), то обратите внимание, что эффект размытия используется очень редко. Чаще всего его можно встретить при отображении движения выстрелов или быстро пролетающих объектов, но не более того. За движением людей никаких шлейфов не остается.

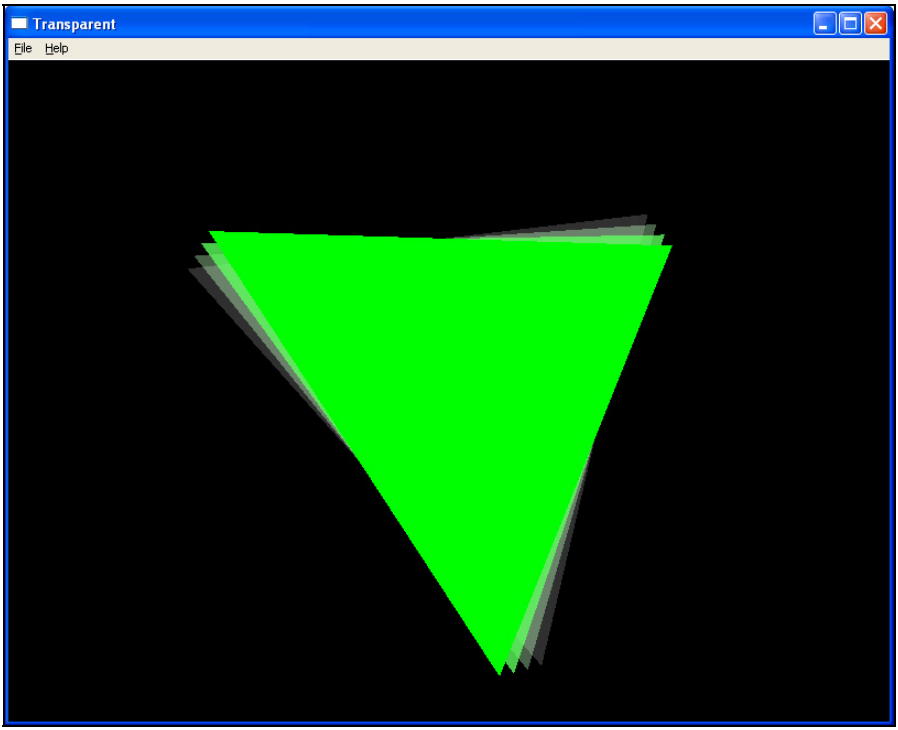

Рис. 5.2. Результат работы программы

В демо-сцене пули летают редко, а вот падающая звезда должна оставлять шлейф. Будьте осторожны при выборе объектов, которые могут иметь шлейф. Все должно быть в меру.

В этом примере мы рассмотрели достаточно простой метод размытия. В DirectX9 реализована очень мощная и удобная технология — *вершинные шейдеры*. С их помощью можно реализовать более качественное размытие, но шейдеры — это отдельная тема.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Alpha2.

5.5. Взрыв на макаронной фабрике

Если вы строите сцену баталий, то выстрелы и взрывы должны уничтожать объекты. Прошли те времена, когда достаточно было просто удалить объект из сцены через некоторое время после уничтожения. Сейчас уже необходимо, чтобы объекты разлетались в клочья, когда в них стреляют из базуки.

Давайте посмотрим простейший пример разрыва в клочья прямоугольника. Прямоугольник будет разрываться на 16 прямоугольников более маленького размера, а значит, можно заранее создать большой прямоугольник из 16-ти маленьких, а во время взрыва заставить маленькие прямоугольники хаотично двигаться. Но это слишком расточительные расходы процессорного времени, когда объект находится в цельном состоянии. Зачем же его рисовать из 64-х вершин, когда можно обойтись четырьмя. Поэтому лучше реализовать следующую логику — если объект целый, то нарисовать его из минимума вершин. Если взорван, то нарисовать большое количество маленьких объектов.

Для реализации примера помимо переменных, необходимых для инициализации Direct3D, нам понадобится структура, описывающая анимацию каждого отдельного прямоугольника после взрыва:

```
struct sQuad
{
  float x, y;       // Координаты прямоугольника
  float xInc, yInc; // Значения приращения координат
  float angleInc;   // Приращение угла
  float angle;      // Угол
};
```

Каждая отдельная часть большого квадрата после взрыва будет не только двигаться, но и вращаться, для этого в структуре есть все необходимые поля.

Помимо структуры нам понадобятся еще следующие переменные:

- `tStartAnimTime` типа `double` — время, когда начата анимация;
- константа `AnimTime` — время, через которое должен произойти взрыв;
- `Triangle` — массив из 8-ми элементов типа структуры, описывающей вершину треугольника. Каждая вершина будет описываться координатами и цветом;
- `Quads` — массив из 16-ти структур типа `sQuad`. Разрыв будет происходить на 16 кусочков, а элементы этой структуры будут описывать положение каждого из кусочков;
- `vTriangleBuffer` типа `IDirect3DVertexBuffer9` — это вершинный буфер.

Почему нам потребуется только 8 элементов в вершинах? Четыре элемента будут описывать прямоугольник, который должен прорисовываться в том

случае, если объект целый. Остальные 4 будут описывать маленький прямоугольник, на который разорвется объект после взрыва.

В листинге 5.6 вы можете увидеть пример инициализации вершин.

Листинг 5.6. Инициализация вершин

```
// Задание вершин маленького прямоугольника
Triangle[0].x=-0.4f;
Triangle[0].y=-0.4f;
Triangle[0].z=0.1f;
Triangle[0].color = D3DCOLOR_XRGB(0, 0, 255);

Triangle[1].x=-0.4f;
Triangle[1].y=-0.2f;
Triangle[1].z=0.1f;
Triangle[1].color = D3DCOLOR_XRGB(0, 0, 255);

Triangle[2].x=-0.2f;
Triangle[2].y=-0.4f;
Triangle[2].z=0.1f;
Triangle[2].color = D3DCOLOR_XRGB(0, 0, 255);

Triangle[3].x=-0.2f;
Triangle[3].y=-0.2f;
Triangle[3].z=0.1f;
Triangle[3].color = D3DCOLOR_XRGB(0, 0, 255);

// Задание вершин большого прямоугольника
Triangle[4].x=-0.4f;
Triangle[4].y=-0.4f;
Triangle[4].z=0.1f;
Triangle[4].color = D3DCOLOR_XRGB(0, 0, 255);

Triangle[5].x=-0.4f;
Triangle[5].y=0.4f;
Triangle[5].z=0.1f;
Triangle[5].color = D3DCOLOR_XRGB(0, 0, 255);

Triangle[6].x=0.4f;
Triangle[6].y=-0.4f;
Triangle[6].z=0.1f;
Triangle[6].color = D3DCOLOR_XRGB(0, 0, 255);
```

```
Triangle[7].x=0.4f;
Triangle[7].y=0.4f;
Triangle[7].z=0.1f;
Triangle[7].color = D3DCOLOR_XRGB(0, 0, 255);

// Здесь необходимо добавить код копирования значения из массива
// вершин в вершинный буфер

// Инициализация структур, описывающих движение отдельных кусочков
for (int i=0; i<4; i++)
 for (int j=0; j<4; j++)
 {
  Quads[i*4+j].x=j*0.2f;
  Quads[i*4+j].y=i*0.2f;
  Quads[i*4+j].xInc=(float)((rand() % 8) / 100.0f-0.05f);
  Quads[i*4+j].yInc=(float)((rand() % 8) / 100.0f-0.05f);
  Quads[i*4+j].angleInc=(float)((rand() % 8) / 50.0f);
  Quads[i*4+j].angle=0;
 }
```

Первые четыре вершины описывают маленький прямоугольник, размером в 0.2×0.2 единицы. Следующие четыре точки описывают прямоугольник в 0.8×0.8 единицы. Это не случайно, ведь первый прямоугольник в 4 раза меньше большого и в большой прямоугольник впишется ровно 16 маленьких. При этом координаты маленького прямоугольника выстроены так, чтобы он вписался в левый верхний угол большого. Это значительно упростит отображение.

На рис. 5.3 показано, как большой прямоугольник будет создаваться из 16-ти маленьких. Чтобы вы увидели результат, я раскрасил вершины маленьких прямоугольников разными цветами.

Самое интересное, что мы описываем координаты только одного маленького прямоугольника — левого нижнего. Все остальные будут создаваться клонированием.

После заполнения массива должно быть создание вершинного буфера и копирование в него данных из сформированного массива. Этот код я опустил, потому что мы его рассматривали уже не раз.

Далее идет заполнение массива `Quads`. Для этого запускается два цикла от 0 до 3. Несмотря на то, что массив `Quads` одномерный, с помощью двух циклов мы с ним работаем как с двумерным. При этом координаты X и Y рассчитываются так, чтобы очередной прямоугольник нарисовался четко в своей ячейке (вспоминаем рис. 5.3, который должен быть где-то рядом с этим абзацем).

Угол поворота изначально равен нулю, чтобы прямоугольники стояли ровно. Значения приращения по осям X и Y, а также приращение угла поворота берутся случайным образом.

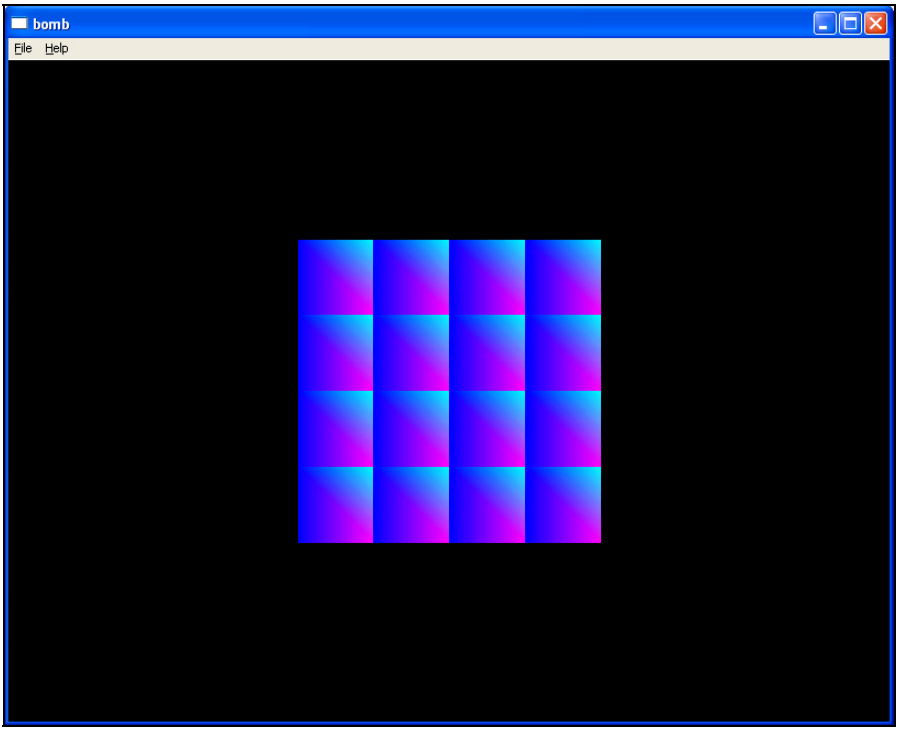

Рис. 5.3. Создание большого прямоугольника из 16-ти маленьких

Теперь посмотрим, как отобразить анимацию взрыва (листинг 5.7).

Листинг 5.7. Анимация взрыва

```
void DrawScene()
{
 // Вершинный формат
 pD3DDevice->SetStreamSource(0, vTriangleBuffer, 0, sizeof(sVertex));
 pD3DDevice->SetVertexShader(NULL);
 pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE);

 double tCurrentTime = GetTickCount();
 if ((tCurrentTime-tStartAnimTime)>AnimTime)
 {
  for (int i=0; i<4; i++)
```

```
   for (int j=0; j<4; j++)
   {
    D3DMATRIX World = {
      cos(Quads[i*4+j].angle), -sin(Quads[i*4+j].angle), 0, 0,
      sin(Quads[i*4+j].angle), cos(Quads[i*4+j].angle), 0, 0,
      0, 0, 1, 0,
      Quads[i*4+j].x, Quads[i*4+j].y, 0, 1,
    };
    pD3DDevice->SetTransform(D3DTS_WORLD, &World);

    pD3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);

    Quads[i*4+j].x+=Quads[i*4+j].xInc;
    Quads[i*4+j].y+=Quads[i*4+j].yInc;
    Quads[i*4+j].angle+=Quads[i*4+j].angleInc;
   }
  }
  else
   pD3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 4, 2);
}
```

Сначала задаем вершинный формат и вершинный буфер, который будет использоваться для рисования. Затем определяем, сколько прошло времени, с момента старта анимации. Если пора взрывать объект, то запускаем два цикла, которые будут рисовать из одного маленького прямоугольника (который описан первыми четырьмя вершинами) один большой. Для этого просто перемещаем мировые координаты в соответствии с положением текущего маленького прямоугольника и выводим в этом месте два треугольника, образующих прямоугольник.

Если еще не настало время взрыва, то рисуем один большой прямоугольник.

Действительно, за счет хранения двух прямоугольников мы расходуем память, да и кода нужно больше. Но зато, если объект целый, мы освобождаем процессор от лишних расчетов.

Запустите пример, и через 4 сек вы увидите как бы взрыв, после которого вихрем будут разбросаны маленькие прямоугольники по экрану. Если добавить синхронизацию и разбить большой прямоугольник на большее количество маленьких кусков (например, размером в 0.1×0.1 единицы), то пример будет выглядеть еще интереснее.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\bomb.

5.6. Текстуры

В *разд. 2.11*, когда рассматривались сетки Mesh, мы немного затронули тему текстур. В данном разделе мы слегка углубимся в эту тему, потому что она позволяет реально повысить качество и эффектность демо-ролика.

Когда мы работаем с 3D-сценой, то нет необходимости рисовать все с помощью примитивов. Каждый примитив — это нагрузка на процессор и видеокарту и не всегда оправданный. Допустим, что вам нужно создать трехмерный объект человека. Вы можете каждую деталь одежды рисовать с помощью 3D-примитивов, а можно просто обтянуть объект такой текстурой, на которой уже нарисованы брюки с карманами и поясом. И такое решение действительно оправдано, если ваш виртуальный друг не будет совать руки в нарисованные карманы или расстегивать ремень, чтобы снять штаны.

Конечно, натянуть на объект рисунок мало. Необходимо правильно позиционировать этот рисунок, да и качество результата в большей степени зависит от качества текстуры. В идеальном случае, вы экономите при создании сцены большое количество вершин и треугольников, а значит, повышаете скорость формирования сцены при минимуме потерь. Хорошую текстуру фотографического качества не заменит никакое количество вершин.

Рассмотрим простой пример, когда текстура позволит вам сэкономить процессорное время. Допустим, что вы хотите построить объект, например, 10-этажный дом. Для его реализации необходимо создать куб, в котором будет множество окон. Каждое окно — это как минимум четыре вершины и два треугольника, образующие прямоугольник, окрашенные в белый цвет. Этот прямоугольник должен символизировать окно. А если сделать еще и подоконник и форточку, то количество вершин резко возрастает.

Таким образом, для создания такого простого объекта понадобится слишком много вершин, ведь в 10-этажном доме будет достаточно много окон. Но если сфотографировать дом с нужным количеством этажей и фотографию натянуть в качестве текстуры на куб, то количество вершин останется равным 8, а сцена будет выглядеть намного лучше, чем создавать дом вершинами по кирпичику, раскрашивая их в разные цвета. А главное — это экономия времени построения сцены.

5.6.1. Простой пример работы с текстурами

Начнем с простейшего примера натягивания прямоугольной текстуры на прямоугольник. Давайте создадим новый проект, и добавим инициализацию Direct3D. Для реализации примера нам понадобится интерфейс, который умеет работать с текстурами, и такой интерфейс называется `IDirect3DTexture9`.

Добавьте глобальную переменную `tQuadTexture`, которая будет являться указателем на данный интерфейс.

Теперь необходимо описать структуру, для хранения координаты вершин прямоугольника, на который будет натягиваться текстура. При описании вершины нам понадобятся координаты этой вершины в пространстве и координаты текстуры, которые должны быть связаны с данной точкой. Цвет вершины описывать нет смысла, потому что он все равно не будет виден под текстурой.

Для описания координаты вершины нужно использовать буфер формата `D3DFVF_XYZ`, а для описания координат текстуры потребуется формат `D3DFVF_TEX1`. Мы уже не раз работали с флагом `D3DFVF_XYZ`, и для него в структуре предусмотрим три параметра, описывающие положение точки по трем координатам: X, Y и Z. Флаг `D3DFVF_TEX1` требует, чтобы в описании вершины было еще два параметра, типа `float`, и это будут X- и Y-координаты (текстура — это двумерный рисунок и Z-координата тут не нужна) точки текстуры, которые будут связаны с данной вершиной. Чтобы переменные координат текстуры не конфликтовали с именами координаты самой вершины, для текстуры используют имена `u` (горизонталь или `x`) и `v` (вертикаль или `y`).

Таким образом, структура, описывающая вершину, должна выглядеть следующим образом:

```
struct sVertex
{
  float x, y, z;
  float u, v;
};
```

Добавьте описание этой структуры в свой проект.

Теперь объявляем переменную `Quad`, которая будет являться массивом из вершин, и переменную `vQuadBuffer` — буфер для хранения вершин:

```
sVertex Quad[4];
IDirect3DVertexBuffer9 * vQuadBuffer;
```

Переходим к инициализации проекта. После инициализации Direct3D добавим код из листинга 5.8.

Листинг 5.8. Инициализация проекта

```
// Заполнение массива вершин
Quad[0].x=-0.5f;
Quad[0].y=-0.5f;
Quad[0].z=0.1f;
```

```
Quad[0].u=0.0f;
Quad[0].v=1.0f;

Quad[1].x=-0.5f;
Quad[1].y=0.5f;
Quad[1].z=0.1f;
Quad[1].u=0.0f;
Quad[1].v=0.0f;

Quad[2].x=0.5f;
Quad[2].y=-0.5f;
Quad[2].z=0.1f;
Quad[2].u=1.0f;
Quad[2].v=1.0f;

Quad[3].x=0.5f;
Quad[3].y=0.5f;
Quad[3].z=0.1f;
Quad[3].u=1.0f;
Quad[3].v=0.0f;

// Создание вершинного буфера
void * pBuf;
if (FAILED(pD3DDevice->CreateVertexBuffer(sizeof(sVertex) * 4,
   D3DUSAGE_WRITEONLY, D3DFVF_XYZ | D3DFVF_TEX1,
   D3DPOOL_DEFAULT, &vQuadBuffer, 0)))
 return FALSE;

// Копирование массива вершин в буфер
if (FAILED(vQuadBuffer->Lock(0, sizeof(sVertex) * 4, &pBuf, 0)))
 return FALSE;
memcpy(pBuf, Quad, sizeof(sVertex) * 4);
vQuadBuffer->Unlock();

// Загрузка текстуры
D3DXCreateTextureFromFile(pD3DDevice, "tex.bmp", &tQuadTexture);

// Установка матрицы просмотра
D3DMATRIX View = {
  1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 2, 1,
 };
pD3DDevice->SetTransform(D3DTS_VIEW, &View);
```

Для каждой вершины треугольника задаются не только координаты расположения, но и координаты текстуры, которые изменяются в пределах от 0 до 1. При этом 0 по оси X соответствует левой крайней точке текстуры, а 1 по оси X — это правая крайняя точка.

Для левой нижней точки прямоугольника мы устанавливаем координаты текстуры 0.0×1.0. Это значит, что с этой вершиной будет связана левая нижняя точка текстуры. Для левой верхней точки устанавливаются координаты 0.0×0.0, т. е. левая верхняя точка текстуры. Таким же образом задаются и остальные координаты вершин и координаты точек текстуры.

После задания значений всех вершин создаем вершинный буфер с помощью метода `CreateVertexBuffer`. Третий параметр, который описывает формат вершин, равен сочетанию флагов `D3DFVF_XYZ` и `D3DFVF_TEX1`, т. е. вершина описывается координатами расположения в пространстве и координатами текстуры. Заполнение вершинного буфера происходит так же, как и для других типов вершин, которые мы использовали ранее.

После создания вершинного буфера вызывается функция `D3DXCreateTextureFromFile`, которая загружает текстуру из указанного файла. С этой функцией мы уже познакомились в *разд. 2.11.2*.

И последнее, что необходимо сделать во время инициализации, — установить матрицу просмотра. Ранее для этого мы использовали отдельную функцию `SetView`, чтобы код был красивее, но в этом проекте я решил отказаться от лишней функции, потому что ее вызов — это неоправданные расходы.

Теперь переходим к рассмотрению процесса отображения объектов с текстурами. Отображать их будем по событию `WM_PAINT`, где будет вызываться функция `DrawScene`. Код этой функции вы можете увидеть в листинге 5.9.

Листинг 5.9. Отображение объектов с текстурами

```
void DrawScene()
{
 float b = float((ViewAngle++)%0x1111)/24;

 // Поворачиваем мировую матрицу по часовой стрелке
 D3DMATRIX World = {
    cos(b), -sin(b), 0, 0,
    sin(b), cos(b), 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1,
  };
 pD3DDevice->SetTransform(D3DTS_WORLD, &World);
```

```
 // Устанавливаем текстуру
 pD3DDevice->SetTexture(0, tQuadTexture);

 // вершинный формат
 pD3DDevice->SetStreamSource(0, vQuadBuffer, 0, sizeof(sVertex));
 pD3DDevice->SetVertexShader(NULL);
 pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_TEX1);

 // Отображаем прямоугольник
 pD3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
}
```

Код отображения объекта с текстурами прост до безобразия. Весь код вам уже знаком, единственное отличие — до отображения прямоугольника с помощью метода `DrawPrimitive` мы устанавливаем загруженную текстуру с помощью метода `SetTexture`.

Запустите пример и в результате вы увидите на экране вращающийся прямоугольник, на котором нарисована текстура из файла tex.bmp. Я в качестве текстуры выбрал изображение белой розы и получил прямоугольник с этой текстурой, как показано на рис. 5.4.

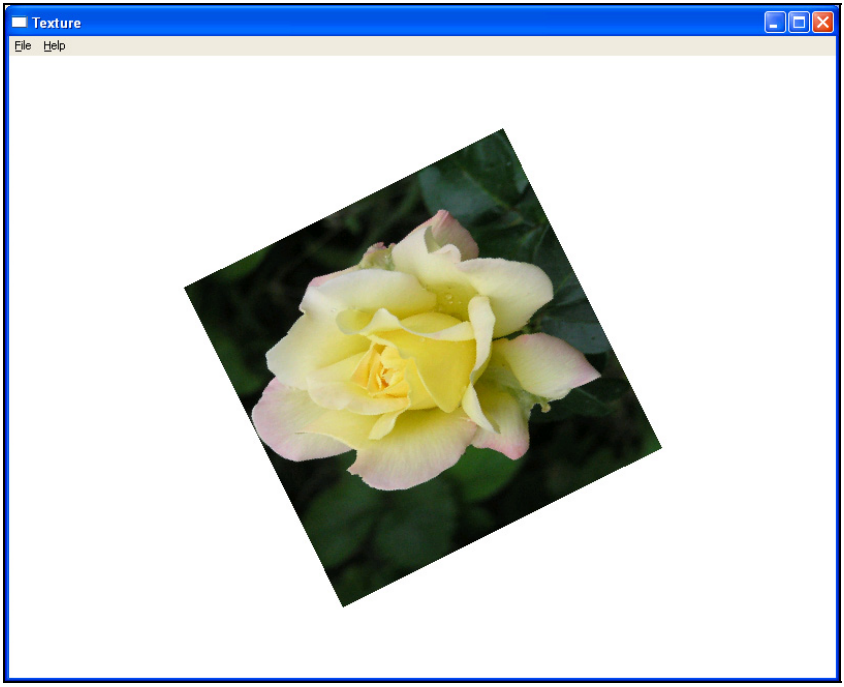

Рис. 5.4. Результат отображения прямоугольника с текстурой

Чем выше качество текстуры, тем лучше будет выглядеть результирующая картинка. Я, конечно, не художник и не могу рисовать такие текстуры, как в играх или профессиональных демо-роликах, поэтому обхожусь цифровым фотоаппаратом и простейшим графическим редактором, с помощью которого и создаю нужные графические образы, используемые в качестве текстур.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Texture.

5.6.2. Взрыв на текстильной фабрике

Нарисовать квадрат с текстурой слишком просто. Давайте попробуем разорвать эту текстуру в клочья, как мы взрывали прямоугольник в *разд. 5.5*.

Для описания координаты вершины с текстурой необходимо указать координату текстуры, которая связана с вершиной. Посмотрите на рис. 5.5, где показаны координаты текстур вершин. По горизонтали показаны координаты U, а по вертикали координаты V.

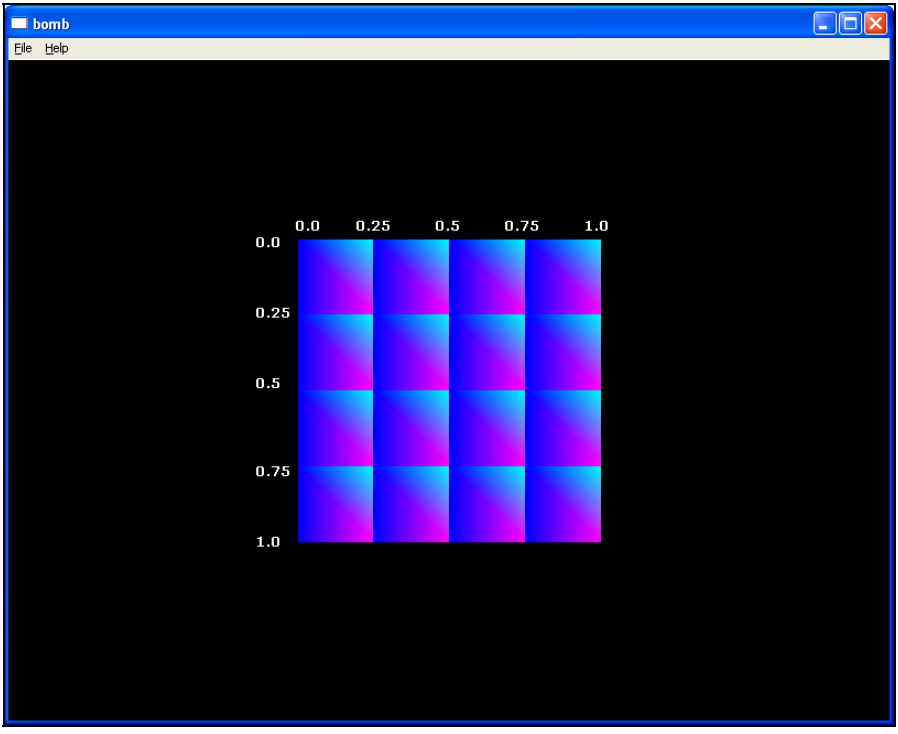

Рис. 5.5. Координаты текстур для прямоугольников

Вот тут возникает одна сложность. В *разд. 5.5* для вывода 16-ти прямоугольников использовался только один маленький, который просто смещался с помощью мировой матрицы. Тот же метод здесь не годится, потому что помимо координат вершин нужно изменять и координаты текстуры, а этого с помощью мировой матрицы сделать нельзя.

Можно создать 64 вершины и описать все треугольники, но это тоже не очень хороший выход. Давайте посмотрим, как можно обойтись 4-мя вершинами, и при этом корректно отобразить текстуру. Итак, код инициализации будет выглядеть, как это показано в листинге 5.10.

Листинг 5.10. Инициализация переменных

```
// Инициализация Direct3D
if (DX3DInit(&pD3D, &pD3DDevice, hWnd, iWidth, iHeight, FALSE)!=S_OK)
{
 MessageBox(hWnd, "Ошибка инициализации DirectX!", "Error", MB_OK);
  return FALSE;
}

// Загрузка текстуры
D3DXCreateTextureFromFile(pD3DDevice, "tex.bmp", &tQuadTexture);

// Создание вершинного буфера
if (FAILED(pD3DDevice->CreateVertexBuffer(sizeof(sVertex) * 4,
   D3DUSAGE_WRITEONLY, D3DFVF_XYZ | D3DFVF_DIFFUSE,
   D3DPOOL_DEFAULT, &vTriangleBuffer, 0)))
 return FALSE;

// Задание матрицы просмотра
D3DMATRIX View = {
  1, 0, 0, 0,
  0, 1, 0, 0,
  0, 0, 1, 0,
  0, 0, 2, 1,
 };
pD3DDevice->SetTransform(D3DTS_VIEW, &View);

// Инициализация параметров массива Quads
for (int i=0; i<4; i++)
 for (int j=0; j<4; j++)
 {
  Quads[i*4+j].x=j*0.2f;
  Quads[i*4+j].y=i*0.2f;
```

```
  Quads[i*4+j].xInc=(float)((rand() % 8) / 100.0f-0.05f);
  Quads[i*4+j].yInc=(float)((rand() % 8) / 100.0f-0.05f);
  Quads[i*4+j].angleInc=(float)((rand() % 8) / 50.0f);
  Quads[i*4+j].angle=0;
 }
```

Здесь мы инициализируем Direct3D, создаем вершинный буфер, устанавливаем матрицу отображения и заполняем массив `Quads` значениями о параметрах движения каждого отдельного прямоугольника. Обратите внимание, что вершинный буфер создается, но не заполняется. Его мы будем заполнять во время отображения сцены.

Код отображения показан в листинге 5.11.

Листинг 5.11. Функция отображения квадрата с текстурой

```
void DrawScene()
{
 // Устанавливаем вершинный формат и включаем текстуру
 pD3DDevice->SetStreamSource(0, vTriangleBuffer, 0, sizeof(sVertex));
 pD3DDevice->SetVertexShader(NULL);
 pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_TEX1);
 pD3DDevice->SetTexture(0, tQuadTexture);

 // Запуск цикла рисования частиц
 for (int i=0; i<4; i++)
  for (int j=0; j<4; j++)
  {
   // Формирование массива вершин
   Triangle[0].x=-0.4f;
   Triangle[0].y=-0.4f;
   Triangle[0].z=0.1f;
   Triangle[0].u = (j)*0.25f;
   Triangle[0].v = (i)*0.25f;

   Triangle[1].x=-0.4f;
   Triangle[1].y=-0.2f;
   Triangle[1].z=0.1f;
   Triangle[1].u = (j)*0.25f;
   Triangle[1].v = (i+1)*0.25f;

   Triangle[2].x=-0.2f;
   Triangle[2].y=-0.4f;
   Triangle[2].z=0.1f;
```

```
   Triangle[2].u = (j+1)*0.25f;
   Triangle[2].v = (i)*0.25f;

   Triangle[3].x=-0.2f;
   Triangle[3].y=-0.2f;
   Triangle[3].z=0.1f;
   Triangle[3].u = (j+1)*0.25f;
   Triangle[3].v = (i+1)*0.25f;

   // Копирование массива в вершинный буфер
   void * pBuf;
   if (FAILED(vTriangleBuffer->Lock(0, sizeof(sVertex) * 4, &pBuf, 0)))
    return;
   memcpy(pBuf, Triangle, sizeof(sVertex) * 4);
   vTriangleBuffer->Unlock();

   // Установка мировой матрицы
   D3DMATRIX World = {
    cos(Quads[i*4+j].angle), -sin(Quads[i*4+j].angle), 0, 0,
    sin(Quads[i*4+j].angle), cos(Quads[i*4+j].angle), 0, 0,
    0, 0, 1, 0,
    Quads[i*4+j].x, Quads[i*4+j].y, 0, 1,
   };
  pD3DDevice->SetTransform(D3DTS_WORLD, &World);

  // Отображение текущего прямоугольника
  pD3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);

  // Если настало время, то изменяем положение и угол прямоугольника
  double tCurrentTime = GetTickCount();
  if ((tCurrentTime-tStartAnimTime)>AnimTime)
  {
   Quads[i*4+j].x+=Quads[i*4+j].xInc;
   Quads[i*4+j].y+=Quads[i*4+j].yInc;
   Quads[i*4+j].angle+=Quads[i*4+j].angleInc;
  }
 }
}
```

В данном примере массив параметров вершин заполняется динамически, благодаря чему мы легко можем рассчитать координаты текущей точки текстуры и установить ее текущей вершине. Затем заполненный массив копируется в вершинный буфер, который уже был создан на этапе инициализации. Остальной код отображения прямоугольников нам уже знаком.

Это только пример, который показывает, что параметры вершин можно формировать динамически во время генерации сцены. В реальных условиях я бы не стал писать подобный код, потому что он слишком расточителен. Самое слабое звено программы — цикл, а мы реально усложнили его. Формирование сцены будет намного быстрее, если заранее подготовить все 64 вершины и потом только выводить их, а не рассчитывать. Разумеется, мы теряем лишнюю память, но это копейки по сравнению с потерями в производительности. Современные компьютеры обладают достаточным количеством памяти, чтобы не экономить какие-то 640 байт. Это капля в море по сравнению с 512 Мбайт, которые уже стали стандартом для домашней персоналки.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\BombAndTexture.

5.6.3. Прозрачность текстур

Как сделать прямоугольник прозрачным, если на него натянута текстура? Устанавливать прозрачность цвета вершины бесполезно, ведь под текстурой цвета не видно и не видно будет и прозрачности. Поскольку прозрачность устанавливается для цвета, а не самой вершине, необходим другой метод решения проблемы, и этот метод есть — настроить прозрачность текстуры. Вы можете настроить прозрачность каждого пиксела текстуры.

Итак, давайте напишем пример, в котором на заднем плане будет отображаться прямоугольник с натянутой текстурой. Это будет фоновое изображение. Для реализации этого прямоугольника необходимо создать прямоугольник в экранных координатах, натянуть на него текстуру фона и затем отобразить полученное. Я думаю, что с этим проблем не возникнет. Достаточно взять пример из *разд. 5.3*, увеличить размеры прямоугольника, который мы построили в этом разделе, чтобы прямоугольник занимал весь экран, и натянуть на него текстуру.

Код создания второго прямоугольника, который будет вращаться на переднем плане, также не сложен, и его можно взять из *разд. 5.6.1*. Единственное — изменим загрузку текстуры. Раньше мы использовали функцию `D3DXCreateTextureFromFile`, которая создает интерфейс `IDirect3DTexture9` и загружает в него текстуру. Функция работает хорошо, но нас не устраивает то, что она устанавливает все параметры по умолчанию. Нам нужен способ, с помощью которого можно было бы контролировать параметры создаваемого интерфейса, и этот способ есть в библиотеке Direct3D — функция `D3DXCreateTextureFromFileEx`. Это более расширенная функция, которая в общем виде выглядит следующим образом:

```
HRESULT D3DXCreateTextureFromFileEx(
  LPDIRECT3DDEVICE8 pDevice,
  LPCTSTR pSrcFile,
  UINT Width,
  UINT Height,
  UINT MipLevels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  DWORD Filter,
  DWORD MipFilter,
  D3DCOLOR ColorKey,
  D3DXIMAGE_INFO* pSrcInfo,
  PALETTEENTRY* pPalette,
  LPDIRECT3DTEXTURE8* ppTexture
);
```

Тут у нас аж 14 параметров. Давайте коротко рассмотрим их.

За более подробной информацией придется обратиться к файлам помощи Direct3D, хотя, эта справка не блещет информацией. Итак, функция `D3DXCreateTextureFromFileEx` получает следующие параметры:

- `pDevice` — устройство Direct3D, которое используется функцией при загрузке текстуры;
- `pSrcFile` — имя или полный путь к графическому файлу. Чем хороша эта функция (и `D3DXCreateTextureFromFile` тоже), так это тем, что она поддерживает основные графические форматы: BMP, DDS, DIB, JPG, PNG и TGA;
- `Width` и `Height` — ширина и высота текстуры соответственно. Если в этих параметрах указаны нулевые значения, то значения ширины и высоты текстуры будут взяты из графического файла в соответствии с размерами растрового рисунка;
- `MipLevels` — количество уровней текстур. Это значение определяет количество последовательности текстур;
- `Usage` — параметры использования. Мы не будем применять особых параметров, поэтому будем указывать здесь нулевое значение;
- `Format` — формат пиксела. Здесь можно указать одно из значений перечисления `D3DFORMAT`. Этот параметр необходимо задать явно. Поскольку мы будем управлять прозрачностью, то следует выбрать тип `D3DFMT_A8R8G8B8`;

- `Pool` — память, в которую будет помещена текстура. Значение по умолчанию для нас недопустимо, потому что в этом случае нельзя будет блокировать интерфейс поверхности и изменять его напрямую. Чтобы блокировка была доступной, установим в этом параметре значение `D3DPOOL_MANAGED`;
- `Filter` — фильтр для изображения. Он пока что нам не нужен, поэтому оставим это значение по умолчанию;
- `MipFilter` — уровень фильтрации, который нами также использоваться не будет;
- `ColorKey` — цвет пиксела, который при загрузке будет автоматически становиться прозрачным. Если указать 0, то прозрачного пиксела не будет;
- `pSrcInfo` — структура типа `D3DXIMAGE_INFO`, в которую будет записана информация о загруженном файле. Мы эту информацию можем посмотреть в любом графическом редакторе, потому что загружаем заранее подготовленный файл. Если ваша программа дает пользователю возможность загружать любую текстуру, то через этот параметр вы сможете получить информацию о загруженном растре;
- `pPalette` — указатель на палитру;
- `ppTexture` — указатель на интерфейс типа `IDirect3DTexture9`, который будет создан, и в него будут загружены растровые данные текстуры.

Для реализации примера нам понадобятся две структуры, которые будут описывать формат пиксела для фонового прямоугольника в экранных координатах и маленького треугольника, который мы станем вращать:

```
struct sVertex
{  float x, y, z, u, v; };

struct sVertexRHW
{  float x, y, z, RHW, u, v;};
```

Помимо этого, нам понадобятся переменные для хранения вершинного буфера вращающегося прямоугольника и прямоугольника заднего буфера, а также интерфейсы для хранения текстур:

```
IDirect3DVertexBuffer9 * vQuadBuffer;
IDirect3DTexture9 *tQuadTexture;

IDirect3DTexture9 *tBackTexture;
IDirect3DVertexBuffer9 * vBackBuffer;
```

Обратите внимание, что мы обходимся без массивов вершин. Это сделано намеренно, чтобы вы увидели еще один вариант заполнения вершинного буфера.

Код инициализации программы представлен в листинге 5.12. Код инициализации Direct3D и установку матрицы просмотра я опустил, дабы сэкономить место в книге. Помимо этого, показан только код задания первой вершины каждого прямоугольника, а все остальное опять же опущено. Полный код вы всегда можете увидеть на компакт-диске, а здесь мы рассмотрим только самое интересное.

Листинг 5.12. Инициализация прямоугольников и задание прозрачности текстуры

```
// Переменные для вершинного буфера вращающегося прямоугольника
sVertex Quad[4];

Quad[0].x=-0.5f;
Quad[0].y=-0.5f;
Quad[0].z=0.1f;
Quad[0].u=0.0f;
Quad[0].v=1.0f;

// Здесь идет заполнение остальных вершин прямоугольника

// Копирование массива в вершинный буфер
void * pBuf;
if (FAILED(pD3DDevice->CreateVertexBuffer(sizeof(sVertex) * 4,
   D3DUSAGE_WRITEONLY, D3DFVF_XYZ | D3DFVF_TEX1,
   D3DPOOL_DEFAULT, &vQuadBuffer, 0)))
 return FALSE;
if (FAILED(vQuadBuffer->Lock(0, sizeof(sVertex) * 4, &pBuf, 0)))
  return FALSE;
memcpy(pBuf, Quad, sizeof(sVertex) * 4);
vQuadBuffer->Unlock();

// Определяем вершины прямоугольника заднего плана
sVertexRHW BackQuad[4];

BackQuad[0].x=0.0f;
BackQuad[0].y=0.0f;
BackQuad[0].z=0.2f;
BackQuad[0].u=0.0f;
BackQuad[0].v=0.0f;
```

```
// Здесь идет заполнение остальных вершин прямоугольника

// Создание вершинного буфера
if (FAILED(pD3DDevice->CreateVertexBuffer(sizeof(sVertexRHW) * 4,
   D3DUSAGE_WRITEONLY, D3DFVF_XYZRHW | D3DFVF_TEX1,
   D3DPOOL_DEFAULT, &vBackBuffer, 0)))
 return FALSE;
if (FAILED(vBackBuffer->Lock(0, sizeof(sVertexRHW) * 4, &pBuf, 0)))
 return FALSE;
memcpy(pBuf, BackQuad, sizeof(sVertexRHW) * 4);
vBackBuffer->Unlock();

// Загрузка текстур
D3DXCreateTextureFromFileEx(pD3DDevice, "tex.bmp", 256, 256, 0,
    0, D3DFMT_A8R8G8B8, D3DPOOL_MANAGED, D3DX_DEFAULT, D3DX_DEFAULT,
    0, 0, 0, &tQuadTexture);
D3DXCreateTextureFromFile(pD3DDevice, "back.bmp", &tBackTexture);

// Заполнение альфа-составляющей
D3DLOCKED_RECT text_rect;
if (FAILED(tQuadTexture->LockRect(0, &text_rect, 0, 0)))
 return FALSE;

BYTE* dst=(BYTE*)text_rect.pBits;
for (int i=0; i<256; i++)
 for (int j=0; j<256; j++)
 {
  *(DWORD *)(dst+i*text_rect.Pitch+j*4)&=0x00FFFFFF;
  *(DWORD *)(dst+i*text_rect.Pitch+j*4)+=0x11000000;
 }

tQuadTexture->UnlockRect(0);
```

Думаю, что с заполнением вершин уже не должно возникнуть проблем, поэтому этот код я привел вкратце, чтобы сэкономить место. Единственное отличие в данном примере — массив для хранения вершин объявлен локально. Это будет более корректным, ведь мы используем его только при заполнении, и нет смысла держать массив, как глобальную переменную.

После заполнения вершин загружаем текстуры из растрового файла. Текстура, которая предназначена для фонового прямоугольника, будет загружаться с помощью функции `D3DXCreateTextureFromFile`, а значит, результирующий интерфейс будет с параметрами по умолчанию. Это нас устраивает, потому что этой текстурой мы управлять не станем.

Загрузка текстуры для вращающегося прямоугольника загружается с помощью функции D3DXCreateTextureFromFileEx. Это необходимо, чтобы мы могли явно задать формат точки D3DFMT_A8R8G8B8, который позволит нам управлять прозрачностью и установить флаг D3DPOOL_MANAGED, чтобы память можно было блокировать.

Теперь давайте настроим прозрачность. Растровый файл формата BMP, который мы загрузили для текстуры, не содержит информации об альфа-канале, поэтому мы должны самостоятельно установить каждой точке значение прозрачности. Для этого нужно заблокировать поверхность текстуры и через прямой доступ к памяти данных изменить значение альфа-канала.

Для блокировки поверхности текстуры у интерфейса IDirect3DTexture9 есть метод LockRect. В общем виде метод выглядит следующим образом:

```
HRESULT LockRect(
  UINT Level,
  D3DLOCKED_RECT *pLockedRect,
  CONST RECT *pRect,
  DWORD Flags
);
```

Здесь имеется четыре параметра:

- Level — уровень текстуры, который нужно заблокировать. Вспоминаем, что уровней может быть несколько, и каждый из них хранит свою текстуру с определенной детализацией. В нашем случае уровни не используются, поэтому указываем значение 0;
- pLockedRect — указатель на структуру D3DLOCKED_RECT, описывающую заблокированную область текстуры. Эта структура состоит из двух полей:
  - Pitch — ширина строки текстуры;
  - pBits — указатель на память, где расположены данные;
- pRect — указатель на структуру RECT, описывающую прямоугольную область текстуры, которую надо заблокировать. Мы указываем нулевое значение, чтобы заблокировать всю поверхность растра;
- Flags — флаги блокировки текстуры, которые мы не станем использовать.

Для удобства после блокировки в переменной dst сохраняем указатель на память текстуры:

```
BYTE* dst=(BYTE*)text_rect.pBits;
```

Теперь запускаем цикл, который будет перебирать все пикселы и изменять значение байта прозрачности:

```
for (int i=0; i<256; i++)
  for (int j=0; j<256; j++)
  {
    *(DWORD *)(dst+i*text_rect.Pitch+j*4)&=0x00FFFFFF;
    *(DWORD *)(dst+i*text_rect.Pitch+j*4)+=0x11000000;
  }
```

Таким образом, мы создали интерфейс для хранения текстуры формата D3DFMT_A8R8G8B8. Это очень удобно, потому что каждый пиксел описывается 4 байтами, т. е. значением DWORD. Размер картинки выбран 256×256, который будет кратен степени двойки. Итак, внутри цикла мы определяем положение очередной точки. Для этого используется следующая формула:

```
dst+i*text_rect.Pitch+j*4
```

К указателю интерфейса прибавляем значение счетчика i (координата Y изменяемой точки), умноженное на ширину строки, и прибавляем значение счетчика j (координата X), умноженное на ширину пиксела, т. е. на число 4.

У каждого пиксела сначала убираем альфа-канал. Для этого следует выполнить логическую операцию "И" с числом 0x00FFFFFF. В этом числе первый байт равен нулю, а значит, этот байт будет обнулен. Это как раз альфа-канал, который нам и нужно убрать.

После этого, прибавляем к значению цвета пиксела число 0x11000000. Теперь альфа-канал будет увеличен на значение 0x11. Значение остальных составляющих цвета пиксела остаются не тронутыми.

На этом подготовительные действия можно считать законченными, и следует переходить к отображению сцены. Как всегда это будет происходить по событию WM_PAINT, где должна очищаться поверхность, и вызываться функция DrawScene, где и будет происходить создание сцены. Код функции показан в листинге 5.13.

Листинг 5.13. Функция рисования прямоугольника с прозрачной текстурой

```
void DrawScene()
{
  // Установить буфер и формат прямоугольника заднего плана
  pD3DDevice->SetStreamSource(0, vBackBuffer, 0, sizeof(sVertexRHW));
  pD3DDevice->SetFVF(D3DFVF_XYZRHW | D3DFVF_TEX1);
  pD3DDevice->SetTexture(0, tBackTexture);

  // Отобразить прямоугольник
  pD3DDevice->DrawPrimitive(D3DPT_TRIANGLEFAN, 0, 2);
```

```
  // Расчет угла поворота и установка мировой матрицы
  float b = float((ViewAngle++)%0x1111)/24;

  D3DMATRIX World = {
    cos(b), -sin(b), 0, 0,
    sin(b), cos(b), 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1,
  };
  pD3DDevice->SetTransform(D3DTS_WORLD, &World);

  // Установка вершин и их формата для второго прямоугольника
  pD3DDevice->SetStreamSource(0, vQuadBuffer, 0, sizeof(sVertex));
  pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_TEX1);
  pD3DDevice->SetTexture(0, tQuadTexture);

  // Разрешить прозрачность
  pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, 1);
  pD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
  pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

  // Отобразить прямоугольник
  pD3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);

  pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, 0);
}
```

У нас два прямоугольника с разным форматом вершин, поэтому приходится сначала устанавливать один вершинный буфер, указывать его формат и текстуру, а затем все то же самое производить для второго прямоугольника.

Параметры прозрачности включаются точно так же, как мы это делали для цветного прямоугольника, поэтому все команды вам уже должны быть знакомы. После отображения прозрачного прямоугольника отключаем использование прозрачности, иначе при следующей попытке перерисовать сцену прозрачным будет и прямоугольник заднего плана, который просто исчезнет, потому что там альфа-канал не заполнен и равен 0.

Конечно, можно установить альфа-канал и у текстуры, которую мы натягиваем на задний прямоугольник, и тогда достаточно будет один раз включить прозрачность, но это будут лишние расходы процессора на анализ текстуры, у которой все пикселы не прозрачны. Зачем анализировать то, что не имеет прозрачности? Поэтому лучше не напрягать процессор лишними расчетами.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\TextureAlpha.

5.6.4. Анимация прозрачности

Давайте немного улучшим пример, который мы написали в *разд. 5.6.3*. Во-первых, в качестве текстуры для вращающегося прямоугольника я выбрал розу, у которой фон равен цвету 0x00FF00FF. Это будет абсолютно прозрачная область, которую мы вообще выводить не станем. Для остальной части цветка будем плавно изменять прозрачность.

Рис. 5.6. Текстура для вращающегося прямоугольника

Писать проект с нуля не станем, а возьмем код из предыдущего раздела. Для начала изменим начальное значение прозрачности, чтобы все точки были прозрачными. Потом переходим к функции `DrawScene` и корректируем ее, как показано в листинге 5.14.

Листинг 5.14. Вывод прозрачного цветка

```
void DrawScene()
{
  // Установка вершин заднего прямоугольника
  pD3DDevice->SetStreamSource(0, vBackBuffer, 0, sizeof(sVertexRHW));
```

```
  pD3DDevice->SetFVF(D3DFVF_XYZRHW | D3DFVF_TEX1);
  pD3DDevice->SetTexture(0, tBackTexture);

  // Вывод заднего прямоугольника
  pD3DDevice->DrawPrimitive(D3DPT_TRIANGLEFAN, 0, 2);

  // Изменение угла мировой матрицы
  float b = float((ViewAngle++)%0x1111)/24;

  D3DMATRIX World = {
    cos(b), -sin(b), 0, 0,
    sin(b), cos(b), 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1,
  };
  pD3DDevice->SetTransform(D3DTS_WORLD, &World);

  // Установка вершинного буфера переднего прямоугольника
  pD3DDevice->SetStreamSource(0, vQuadBuffer, 0, sizeof(sVertex));
  pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_TEX1);

  // Изменение прозрачности
  D3DLOCKED_RECT text_rect;
  if (FAILED(tQuadTexture->LockRect(0, &text_rect, 0, 0)))
          return;
   BYTE* dst=(BYTE*)text_rect.pBits;
   for (int i=0; i<256; i++)
    for (int j=0; j<256; j++)
    {
      if (*(DWORD *)(dst+i*text_rect.Pitch+j*4)!=0x00FF00FF)
        *(DWORD *)(dst+i*text_rect.Pitch+j*4)+=0x01000000;
    }
  tQuadTexture->UnlockRect(0);

  // Установка прозрачности и текстуры
  pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, 1);
  pD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
  pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);
  pD3DDevice->SetTexture(0, tQuadTexture);

  // Отображение второго прямоугольника
  pD3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);

  pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, 0);
}
```

В этом примере перед отображением вращающегося прямоугольника запускается цикл, который перебирает все пикселы текстуры с розой. Если цвет пиксела не равен 0x00FF00FF, то увеличиваем альфа-составляющую пиксела. Таким образом, будет постепенно проявляться форма цветка, а фоновая часть всегда будет прозрачной и невидимой.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\TextureAlpha2.

5.6.5. Анимация текстуры

Благодаря возможности прямого доступа к поверхности текстуры, вы можете управлять не только прозрачностью, но и растром текстуры. Например, можно даже динамически рисовать картинку, которая будет натягиваться на объект. Таким образом, у нас получится анимация текстуры.

Давайте посмотрим еще один интересный пример анимации текстуры и заодно увидим, как можно вручную загрузить текстуру из файла. До этого мы пользовались удобными, но универсальными функциями `D3DXCreateTextureFromFile` и `D3DXCreateTextureFromFileEx`.

Начнем с рассмотрения процесса загрузки графического файла в интерфейс текстуры `IDirect3DTexture9`. Хотя нет, для анимации лучше загрузить графические данные не в интерфейс, а в отдельный массив типа `DWORD`. Для удобства, напишем функцию `LoadTexture`, которая показана в листинге 5.15.

Листинг 5.15. Функция загрузки графического файла в массив

```
bool LoadTexture(DWORD* text_buf, int iWidth, int iHeight, char*
filename)
{
 HANDLE hFile = CreateFile(filename, GENERIC_READ, 0, NULL,
         OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, 0);
 if(hFile == INVALID_HANDLE_VALUE)
  return FALSE;

 BITMAPFILEHEADER bmpfilehdr;
 DWORD dwRead;
 if(!ReadFile(hFile, &bmpfilehdr, sizeof(bmpfilehdr), &dwRead, NULL))
  return FALSE;

 char* ptr=(char*)&bmpfilehdr.bfType;
 if (*ptr!='B' || *++ptr!='M')
  return FALSE;
```

```
 BITMAPINFOHEADER bmpinfohdr;
 if(!ReadFile(hFile, &bmpinfohdr, sizeof(bmpinfohdr), &dwRead, NULL))
  return FALSE;

 if (bmpinfohdr.biCompression!=BI_RGB)
  return 0;

 int iImageSize=bmpinfohdr.biSizeImage;
 if (iImageSize==0)
   iImageSize=((iWidth*4+3) & ~3)*iHeight;

 BYTE* buf=new BYTE[iImageSize];
 if(!ReadFile(hFile, buf, iImageSize, &dwRead, NULL))
   return FALSE;

 if (bmpinfohdr.biBitCount!=24)
   return FALSE;

 int bytesgiven=(iWidth*3+3) & ~3;
 DWORD* btSurf = text_buf;
 BYTE* imagebits = (BYTE*)(&buf[(iHeight-1)*bytesgiven]);

 for (int i=0; i<iHeight; i++)
 {
   RGBTRIPLE* tlImage=(RGBTRIPLE*)imagebits;
   for (int p=0; p<iWidth; p++)
   {
     *btSurf = (DWORD)((tlImage->rgbtBlue) | (tlImage->rgbtGreen << 8) |
              (tlImage->rgbtRed << 16) | (0xFF << 24));
     tlImage++;
     btSurf++;
   }
   imagebits -= bytesgiven;
 }

 delete [] buf;
 return TRUE;
}
```

Функция принимает в качестве параметров четыре значения: указатель на массив, в который необходимо сохранить растровые данные, ширину и высоту изображения и, наконец, имя графического файла.

В *разд. 3.4* мы рассматривали функцию быстрой загрузки изображения `LoadBMPToSurfaceFast`. Функция `LoadTexture` имеет схожую структуру и при ее

написании я использовал код LoadBMPToSurfaceFast, дабы не изобретать очередной велосипед при написании схожей функции, которая также читает графический файл. Разница только в том, что здесь данные загружаются не в поверхность, а в массив.

Теперь посмотрим, как воспользоваться функцией:

```
LoadTexture((DWORD*)&text, 255, 255, "tex.bmp");
```

Переменная text — это указатель на массив типа DWORD, которого должно быть достаточно для хранения растровых данных. Например, для картинки 256×256 необходимо 65 536 байт, а значит, массив должен быть следующего размера:

```
DWORD text[65536];
```

Теперь у нас есть массив, но он бесполезен без интерфейса IDirect3DTexture9. Для создания массива можно воспользоваться методом CreateTexture. Этот метод выглядит следующим образом:

```
HRESULT CreateTexture(
  UINT Width,
  UINT Height,
  UINT Levels,
  DWORD Usage,
  D3DFORMAT Format,
  D3DPOOL Pool,
  IDirect3DTexture9** ppTexture,
  HANDLE* pHandle
);
```

Рассмотрим параметры этого метода:

- Width и Height — ширина и высота текстуры соответственно. Если в этих параметрах указаны нулевые значения, то значения ширины и высоты текстуры будут взяты из графического файла в соответствии с размерами растрового рисунка;
- Levels — количество уровней текстур. Это значение определяет количество последовательности текстур;
- Usage — параметры использования. Мы не будем использовать особых параметров, поэтому укажем здесь нулевое значение;
- Format — формат пиксела. Здесь можно указать одно из значений перечисления D3DFORMAT. Этот параметр необходимо задать явно. Поскольку мы будем управлять прозрачностью, то следует выбрать тип D3DFMT_A8R8G8B8;
- Pool — память, в которую помещается текстура. Значение по умолчанию для нас недопустимо, потому что в этом случае нельзя будет блокировать

интерфейс поверхности и изменять его напрямую. Для доступности блокировки установим в этом параметре значение D3DPOOL_MANAGED;

- ppTexture — указатель на интерфейс типа IDirect3DTexture9, который будет создан, и в него загружены растровые данные текстуры.
- pHandle — пока зарезервировано и параметр должен быть равен нулю.

Для нашего примера необходимо создать интерфейс следующим образом:

```
pD3DDevice->CreateTexture(256, 256, 0, 0, D3DFMT_A8R8G8B8,
              D3DPOOL_MANAGED, &tQuadTexture, NULL);
```

Теперь посмотрим на простейший пример использования буфера растра для анимации текстуры (листинг 5.16).

Листинг 5.16. Пример анимации текстуры

```
void DrawScene()
{
  // Здесь может выводиться фоновый прямоугольник с текстурой
  // и устанавливаться мировая матрица

  // Блокировка интерфейса текстуры
  D3DLOCKED_RECT text_rect;
  if (FAILED(tQuadTexture->LockRect(0, &text_rect, 0, 0)))
   return;

  // Заполнение интерфейса текстуры картинкой
  BYTE* dst=(BYTE*)text_rect.pBits;
  for (int i=0; i<TextureHeight; i++)
    for (int j=0; j<256; j++)
     if (text[i*255+j]!=0xFFFF00FF)
       *(DWORD *)(dst+i*text_rect.Pitch+j*4)=text[i*255+j];

  // Заполнение интерфейса текстуры прозрачными пикселами
  for (int i=TextureHeight; i<256; i++)
   for (int j=0; j<256; j++)
     *(DWORD *)(dst+i*text_rect.Pitch+j*4)=0x00000000;

  // Снятие блокировки
  tQuadTexture->UnlockRect(0);

  // Увеличение переменной TextureHeight
  TextureHeight++;
  if (TextureHeight>255)
         TextureHeight=0;
```

```
  // Установка буфера вершин и его формата
  pD3DDevice->SetStreamSource(0, vQuadBuffer, 0, sizeof(sVertex));
  pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_TEX1);

  // Установка текстуры
  pD3DDevice->SetTexture(0, tQuadTexture);

  // Настройка параметров отображения (включение прозрачности)
  pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, 1);
  pD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
  pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

  // Отображение картинки
  pD3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);

  // Запрет прозрачности
  pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, 0);
}
```

В данном примере для анимации заведена переменная `TextureHeight`, которая будет изменяться от 0 до значения высоты текстуры. Во время каждого формирования сцены значение переменной увеличивается и динамически создается текстура. В качестве текстуры будет создан растр, в котором строки от 0 до `TextureHeight` заполнены строками из картинки, а остальные делаются прозрачными. Таким образом, текстура плавно появляется на объекте прямоугольника.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\TextureLoad.

5.7. Добро пожаловать в истинное 3D-измерение

Уже рассмотрено пять разделов этой главы, которая посвящена третьему измерению, а мы только сейчас погружаемся в реальное третье измерение. А что же было до этого? Действительно, для формирования сцены мы применяли Direct3D, но при этом координата по оси Z реально не использовалась. Доказательства? Давайте откроем проект из *разд. 5.2*, где вращались треугольник и прямоугольник вокруг оси Z в разных направлениях. У обеих фигур Z-координата одинакова.

Давайте попробуем установить для всех вершин разные значения координат, чтобы фигуры пересекались, например, как это показано в листинге 5.17.

Листинг 5.17. Координаты пересекающихся объектов

```
// Координаты треугольника
Triangle[0].x=0.7f;
Triangle[0].y=-0.3f;
Triangle[0].z=-0.1f;

Triangle[1].x=-0.7f;
Triangle[1].y=-0.3f;
Triangle[1].z=0.2f;

Triangle[2].x=0.0f;
Triangle[2].y=0.7f;
Triangle[2].z=0.1f;

// Координаты прямоугольника
Triangle[3].x=-0.5f;
Triangle[3].y=-0.5f;
Triangle[3].z=-0.3f;

Triangle[4].x=-0.5f;
Triangle[4].y=0.5f;
Triangle[4].z=0.4f;

Triangle[5].x=0.5f;
Triangle[5].y=0.5f;
Triangle[5].z=-0.1f;

Triangle[6].x=0.5f;
Triangle[6].y=-0.5f;
Triangle[6].z=0.2f;
```

Запустите пример, и в результате вы увидите, что фигуры не пересекаются. Как бы вы не устанавливали Z-координату, фигура, которая отображается первой, будет дальше, чем фигура, созданная последней. Это значит, что Direct3D пока не использует координату по оси Z, поэтому нужно его заставить сделать это.

Для того чтобы Z-координата реально заработала, необходимо включить тест глубины, это делается во время инициализации Direct3D. У нас есть универсальная функция `DX3DInit`, но не будем ее трогать, чтобы не утратить работо-

способность у примеров из предыдущих глав. Создайте новую функцию DX3DInitZ и скопируйте в нее код функции DX3DInit.

Теперь внесем два очень важных изменения. В самом начале заполняется структура d3dpp типа D3DPRESENT_PARAMETERS. Давайте добавим следующие две строки кода, чтобы они заполнялись вне зависимости от выбранного режима, т. е. до того, как код разделяется на две части:

```
d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
d3dpp.EnableAutoDepthStencil = TRUE;
```

В первой строке мы заполняем свойство AutoDepthStencilFormat, которое определяет формат создаваемой поверхности для буфера глубины. Во второй строке мы разрешаем EnableAutoDepthStencil автоматическое использование этого буфера.

Теперь необходимо найти строку, где вызывается функция D3DXMatrixPerspectiveFovLH, которая задает матрицу проекции. Предпоследний параметр — это расстояние до ближней плоскости отсечения. В функции DX3DInit этот параметр равен нулю, но при этом проекция получится плоской, даже при включенном тесте глубины. Измените это значение на любое положительное число, но помните, что все точки, координаты которых будут меньше этого значения, не отобразятся. Для примеров из данной книги я буду использовать значение 2.

Теперь измените в проекте вызов функции DX3DInitZ на DX3DInit. Только не торопитесь запускать проект, иначе вы увидите мусор. Мы включили тест глубины, но пока не сделали еще одной интересной вещи — не очистили этот буфер. Он содержит мусор, который постоянно будет расти. Для очистки Z-буфера используется уже знакомый нам метод Clear. Он уже вызывается у нас, но очищает только сцену, а не буфер. Чтобы очистился и Z-буфер, необходимо добавить в третий параметр флаг D3DCLEAR_ZBUFFER:

```
pD3DDevice->Clear(1, (D3DRECT*)&r,
      D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
      D3DCOLOR_XRGB(ViewAngle,0, 0), 1.0, 0);
```

Вот теперь пример готов к использованию, и вы можете запустить результат. Чтобы лучше было видно, как фигуры перекрывают друг друга, можно отключить альфа-смешивание.

Сделаем небольшое отступление от темы. Играясь с примером, я случайно отключил очистку сцены (не включил флаг D3DCLEAR_TARGET в третий параметр метода очистки Clear). Результат получился достаточно интересный (рис. 5.7). Чтобы увидеть все это воочию, попробуйте убрать флаг очистки. Иногда он действительно позволяет сделать сцену интересной, если используется анимация.

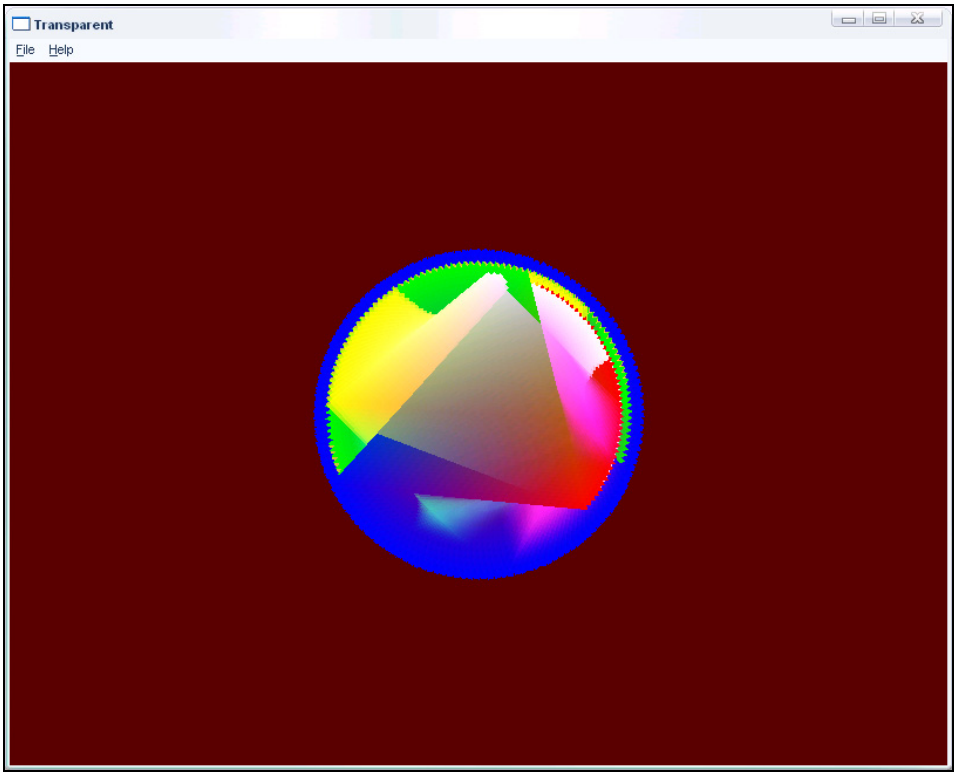

Рис. 5.7. Результат без очистки буфера сцены

Данный случай лишний раз показывает, что красивые сцены могут получаться спонтанно. Хотя, почему могут? Они почти всегда появляются спонтанно. Создавая какую-то сцену, мы играем с числами до тех пор, пока не получим нужный или красивый результат. Если же мы не добились того, что ожидали, но остановились на красивом результате, то можно сказать, что такой эффект получился спонтанно, что достаточно часто происходит в демо-сцене.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\ZAxis.

5.8. Материалы и освещение

Игра с материалами и освещением позволяют добиться достаточно интересных результатов. В *разд. 2.11* мы уже немного познакомились и с освещением и с материалами, но затронули эту тему слишком поверхностно. Сейчас

же мы немного углубимся в эту тему и рассмотрим некоторые варианты игры со светом.

5.8.1. Big sound

Давайте рассмотрим пример, в котором будет как бы двигаться прожектор, освещая сцену. В моем городе есть дискотека, которая называется "Биг Саунд", и на крыше здания стоит прожектор, который хаотично вращается, освещая небо. Именно этот прожектор натолкнул меня на этот пример.

Чтобы объекты, которым назначен материал, корректно отображались на фоне света, необходимо определить нормали для вершин. По нормали Direct3D определяет, как свет будет падать на объект и как будет отражаться. Получается, что при описании вершины, помимо координаты вершины объекта, необходимо указать координату точки, в которую должен быть направлен вектор нормали.

Поскольку объект не будет окрашиваться в определенный цвет, то вершине не нужно задавать цвет. Таким образом, структура, описывающая вершину, должна выглядеть следующим образом:

```
struct sVertex
{
  float x, y, z;
  float nx, ny, nz;
};
```

Эта структура описывает вершину координатами точки и координатами нормали, а значит, при создании и использовании буфера вершин необходимо применять флаги `D3DFVF_XYZ` и `D3DFVF_NORMAL`.

Теперь объявим массив из `Triangle` из семи элементов типа `sVertex` для описания семи вершин (3 на треугольник и 4 на прямоугольник). Сейчас описание вершины будет выглядеть следующим образом:

```
Triangle[0].x=0.7f;
Triangle[0].y=-0.3f;
Triangle[0].z=0.5f;
Triangle[0].nx=0.0f;
Triangle[0].ny=0.0f;
Triangle[0].nz=1.0f;
```

Это пример задания координат первой вершины.

Теперь создадим два материала. Для этого введем две переменные `mat` и `mat2` типа `D3DMATERIAL9`:

```
D3DMATERIAL9 mat, mat2;
```

Структуру D3DMATERIAL9, которая используется для задания материала, мы рассматривали в *разд. 2.11.1*.

Следующий пример заполняет значениями структуры:

```
ZeroMemory(&mat, sizeof(D3DMATERIAL9));
mat.Diffuse.r=0;
mat.Diffuse.g=255;
mat.Diffuse.b=0;
mat.Diffuse.a=255;
mat.Ambient=mat.Diffuse;

ZeroMemory(&mat2, sizeof(D3DMATERIAL9));
mat2.Diffuse.r=0;
mat2.Diffuse.g=255;
mat2.Diffuse.b=255;
mat2.Diffuse.a=255;
mat2.Ambient=mat2.Diffuse;
```

Теперь перейдем к отображению, т. е. к структуре DrawScene, которая приведена в листинге 5.18. В этом листинге показано только самое интересное, а именно работа со светом. Вывод треугольника и прямоугольника я убрал, дабы сэкономить место, потому что мы этот код видели не раз, а полный вариант функции можно найти на прилагаемом компакт-диске.

Листинг 5.18. Функция анимации с использованием освещения

```
void DrawScene()
{
 float b = float((ViewAngle++)%0x1111)/24;

 // Создание освещения
 D3DLIGHT9 Light;
 ZeroMemory(&Light, sizeof(D3DLIGHT9));
 Light.Type = D3DLIGHT_DIRECTIONAL;
 Light.Diffuse.r = Light.Diffuse.g = Light.Diffuse.b =
      Light.Diffuse.a = 1.0f;
 Light.Position = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
 Light.Direction = D3DXVECTOR3(0.0f, cos(b)*1000, 0.5f);
 pD3DDevice->SetLight(0, &Light);
 pD3DDevice->LightEnable(0, TRUE);

 // Включить освещение
 pD3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
```

```
 // Установить материал для треугольника
 pD3DDevice->SetMaterial(&mat);
 // Здесь должен быть код отображения треугольника

 // Установить материал для треугольника
 pD3DDevice->SetMaterial(&mat2);
 // Здесь должен быть код отображения прямоугольника

 pD3DDevice->SetRenderState(D3DRS_LIGHTING, FALSE);
}
```

В самом начале сцены мы создаем источник освещения. Самое интересное — это расположение этого источника. В свойстве `Position` задается точка, в которой находится источник освещения. В параметре `Direction` указываем точку, в которую должен светить источник:

```
Light.Position = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
Light.Direction = D3DXVECTOR3(0.0f, cos(b)*1000, 0.5f);
```

Обратите внимание, что в качестве Y-координаты точки, в которую светит источник освещения, устанавливаем значение `cos(b)*1000`. Переменная `b` увеличивается на 1 при каждом отображении сцены, а благодаря функции `cos`, движение источника освещения вдоль оси Y напоминает синусоиду, т. е. луч света будет то подниматься вверх, то опускаться.

После задания цвета и положения источника освещения делаем его активным и разрешаем отображение света.

Теперь необходимо воспроизвести объекты. Чтобы при отображении фигура была окрашена цветом материала, перед каждым вызовом метода `DrawPrimitive` устанавливаем соответствующий материал. Например, следующая строка с помощью метода `SetMaterial` устанавливает в качестве текущего материал `mat` для объекта треугольника:

```
pD3DDevice->SetMaterial(&mat);
```

После формирования сцены отключаем освещение, потому что оно нам больше не нужно, ведь при следующем формировании сцены будет создаваться новый источник света:

```
pD3DDevice->SetRenderState(D3DRS_LIGHTING, FALSE);
```

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Material.

5.8.2. Свечка

Давайте теперь поиграем с точечным источником света, который будет освещать объект, созданный из сетки Mesh. Сетку загрузим из X-файла, причем при этом объект обтянется текстурой.

Для примера нам понадобятся следующие переменные для загрузки сетки:

```
DWORD dwNumMaterials;

ID3DXMesh *pMesh;
LPDIRECT3DTEXTURE9 *pMeshTextures;
D3DMATERIAL9 *pMeshMaterials;
```

Помимо этого, объекту будет назначен собственный материал, а не материал объекта из X-файла. Для этого нам понадобится структура типа `D3DMATERIAL9`:

```
D3DMATERIAL9 mat;
```

Помимо сетки на нашей сцене будет отображаться объект, символизирующий источник света. Вполне логично было бы сделать этот объект в виде шара, но это будет сложный объект, который проще загрузить из X-файла, а лишнюю сетку загружать не хочется. Поэтому упростим задачу, и пусть источник света символизирует треугольник. Для создания треугольника нам понадобятся следующая структура и переменные:

```
struct sVertex
{
  float x, y, z;
  DWORD color;
};
sVertex Triangle[3];
IDirect3DVertexBuffer9 * vTriangleBuffer;
```

Код инициализации проекта показан в листинге 5.19.

Листинг 5.19. Инициализация проекта

```
// Инициализация Direct3D
if (DX3DInitZ(&pD3D, &pD3DDevice, hWnd, iWidth, iHeight, FALSE)!=S_OK)
{
  MessageBox(hWnd, "Ошибка инициализации DirectX!", "Error", MB_OK);
  return FALSE;
}

// Загрузка сетки из файла
dwNumMaterials = LoadMesh("nlo.x", pD3DDevice, &pMesh,
     &pMeshTextures, "texture.bmp", &pMeshMaterials);
```

```
// Определение материала
ZeroMemory(&mat, sizeof(D3DMATERIAL9));
mat.Diffuse.r=255;
mat.Diffuse.g=255;
mat.Diffuse.b=255;
mat.Diffuse.a=255;
mat.Ambient=mat.Diffuse;

// Определение вершин треугольника
Triangle[0].x=0.5f;
Triangle[0].y=-0.5f;
Triangle[0].z=-0.1f;

Triangle[1].x=-0.5f;
Triangle[1].y=-0.5f;
Triangle[1].z=0.2f;

Triangle[2].x=0.0f;
Triangle[2].y=0.5f;
Triangle[2].z=0.1f;

// Определение цветов вершин треугольника
Triangle[0].color = D3DCOLOR_XRGB(255, 255, 0);
Triangle[1].color = D3DCOLOR_XRGB(255, 255, 0);
Triangle[2].color = D3DCOLOR_XRGB(255, 255, 0);

// Создание вершинного буфера
void * pBuf;
if (FAILED(pD3DDevice->CreateVertexBuffer(sizeof(sVertex) * 3,
   D3DUSAGE_WRITEONLY, D3DFVF_XYZ | D3DFVF_DIFFUSE,
   D3DPOOL_DEFAULT, &vTriangleBuffer, 0)))
 return FALSE;

// заполнение вершинного буфера
if (FAILED(vTriangleBuffer->Lock(0, sizeof(sVertex) * 3, &pBuf, 0)))
 return FALSE;
memcpy(pBuf, Triangle, sizeof(sVertex) * 3);
vTriangleBuffer->Unlock();
```

Все, что показано в этом листинге, нам уже знакомо. Сначала мы инициализируем Direct3D с помощью функции `DX3DInitZ`. Эта функция инициализирует Direct3D с поддержкой Z-буфера.

Затем загружаем сетку с помощью функции `LoadMesh`. В качестве сетки будет файл nlo.x, в котором находится объект в виде звезды (рис. 5.8).

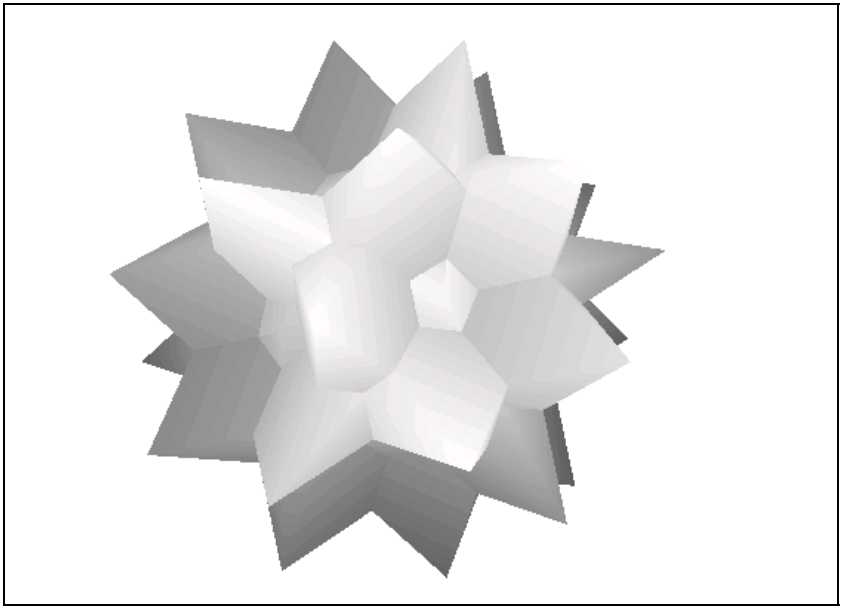

Рис. 5.8. 3D-звезда

Теперь определяем материал, у которого все составляющие цвета `Diffuse` и `Ambient` будут равны 255.

Остальной код задает координаты и цвет вершин треугольника, символизирующего объект освещения, а также создает буфер вершин, в который все это дело копируется.

Теперь посмотрим на код отображения сцены, т. е. функцию `DrawScene` (листинг 5.20).

Листинг 5.20. Отображение звезды и точечного источника света

```
void DrawScene()
{
 float b = float((ViewAngle++)%0x1111)/24;
 float b1 = float((ViewAngle++)%0x1111)/260;

 // Создание источника света
 D3DLIGHT9 light;
 ZeroMemory(&light, sizeof(D3DLIGHT9));
```

```
light.Type = D3DLIGHT_POINT;
light.Position = D3DXVECTOR3(cos(b1), cos(b1)*50, 20);
light.Diffuse.r = light.Diffuse.g = light.Diffuse.b = 1.0f;
light.Ambient=light.Diffuse;
light.Specular=light.Diffuse;
light.Range=40.0f;
light.Attenuation0=0.0f;
light.Attenuation1=1.0f;
light.Attenuation2=0.0f;
pD3DDevice->SetLight(0, &light);
pD3DDevice->LightEnable(0, TRUE);

// Параметры отображения сцены
pD3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
pD3DDevice->SetMaterial(&mat);

// Установка матрицы положения звезды
D3DMATRIX World = {
   cos(b)*cos(b), cos(b)*sin(b), sin(b), 0,
   -sin(b), cos(b), 0, 0,
   -sin(b)*cos(b), -sin(b)*sin(b), cos(b), 0,
   0, 0, 40, 1,
};
pD3DDevice->SetTransform(D3DTS_WORLD, &World);

// Отображение звезды
for (DWORD i=0; i<dwNumMaterials; i++)
{
 if (pMeshTextures[i])
  pD3DDevice->SetTexture(0, pMeshTextures[i]);
 pMesh->DrawSubset(i);
}
pD3DDevice->SetRenderState(D3DRS_LIGHTING, FALSE);

// Установка матрицы положения объекта солнца
D3DMATRIX World1 = {
   1, 0, 0, 0,
   0, 1, 0, 0,
   0, 0, 1, 0,
   cos(b1), cos(b1)*50, 20, 1,
};
pD3DDevice->SetTransform(D3DTS_WORLD, &World1);
```

```
 // Отображение солнца
 pD3DDevice->SetStreamSource(0, vTriangleBuffer, 0, sizeof(sVertex));
 pD3DDevice->SetVertexShader(NULL);
 pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE);
 pD3DDevice->SetTexture(0, NULL);
 pD3DDevice->DrawPrimitive(D3DPT_TRIANGLELIST, 0, 1);
}
```

Сначала определяем два индекса, на основе которых будет определяться положение источника света и угол поворота звезды. Звезда будет вращаться быстрее, чем двигаться источник света, поэтому нам и нужно два коэффициента.

Теперь определяем источник света. Для этого используется структура `D3DLIGHT9`. Для точечного источника света необходимо заполнить следующие поля:

- `Type` — здесь указывается значение `D3DLIGHT_POINT`, что соответствует точечному источнику света;
- `Position` — задается позиция источника света. Направление освещения (свойство `Direction`) указывать не нужно, потому что точечный источник рассеивает свет во все стороны;
- `Range` — расстояние, на которое должен рассеиваться свет;
- `Attenuation0` — определяет интенсивность света в пределах радиуса освещения. При этом, все точки в пределах радиуса освещаются одинаково, что не совсем естественно. Это значение должно быть от 0 до 1;
- `Attenuation1` — задает интенсивность света в пределах радиуса освещения. Если заполнен этот параметр, то, чем дальше объект от источника света, тем менее освещенным он будет. Затухание света в зависимости от расстояния будет вычисляться линейно по формуле 1/D, где D — это расстояние от источника света. Это значение должно быть от 0 до 1;
- `Attenuation2` — определяет интенсивность света в пределах радиуса освещения. Если заполнен этот параметр, то, чем дальше объект от источника света, тем менее освещенным он будет, но, в отличие от предыдущего параметра, затухание света в зависимости от расстояния будет вычисляться квадратично.

В данном примере мы используем линейное затухание, но, на мой взгляд, квадратичное выглядит более естественным. Для квадратичного затухания параметр `Attenuation2` должен быть больше нуля, а остальные параметры лучше оставить нулевыми.

После задания параметров освещения, создаем источник освещения с индексом 0 и разрешаем Direct3D использовать освещение.

Далее следует формирование матрицы положения объекта звезды и отображение самого объекта. Обратите внимание, что в цикле отображения сетки нет установки материала. Материал устанавливается заранее:

```
pD3DDevice->SetMaterial(&mat);
```

Дело в том, что материал, который находится в X-файле, не сможет отражать свет, и если вы попытаетесь добавить в цикл следующую строку:

```
pD3DDevice->SetMaterial(&pMeshMaterials[i]);
```

то объект выведется, но звезда будет черной. Это сделано специально, чтобы показать вам такой нежелательный эффект. Помните, если объект отображается черным цветом, значит, может быть проблема с материалом. Попробуйте просто убрать строку назначения материала и посмотреть на результат. Если объект отобразится корректно, и вы увидите на нем текстуру, то проблема с материалом.

Указывать материал для объекта, который обтянут текстурой, не обязательно. Материал влияет на цвет объекта, но этот цвет под текстурой не будет виден.

После отображения звезды текущую текстуру обязательно обнуляем:

```
pD3DDevice->SetTexture(0, NULL);
```

Если же не обнулить текстуру, то все последующие создаваемые объекты будут обтянуты текстурой, которая использовалась в сетке, а нам это не нужно.

После этого отображаем треугольник, который будет символизировать источник освещения. При этом положение объекта соответствует положению источника света.

Если хотите, чтобы сцена была более реалистичной, то необходимо отобразить объект солнца в виде сферы желтого цвета, свечи или лампочки накаливания, которая будет рассеивать свет.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Light_Point.

5.9. Свечение

Свечение является очень интересным эффектом. Если посмотреть последние демо-ролики, которые появляются в Интернете, то в каждом втором уж точно есть что-то светящееся.

Пример, который мы сейчас рассмотрим, будет создавать светящийся прямоугольник (рис. 5.9). Конечно, черно-белая полиграфия не позволит вам уви-

деть всю красоту, поэтому придется подождать, пока мы не напишем пример. Чем интересен этот пример? Мы рассмотрим, как создавать и использовать объемные текстуры и создадим очень красивый пример.

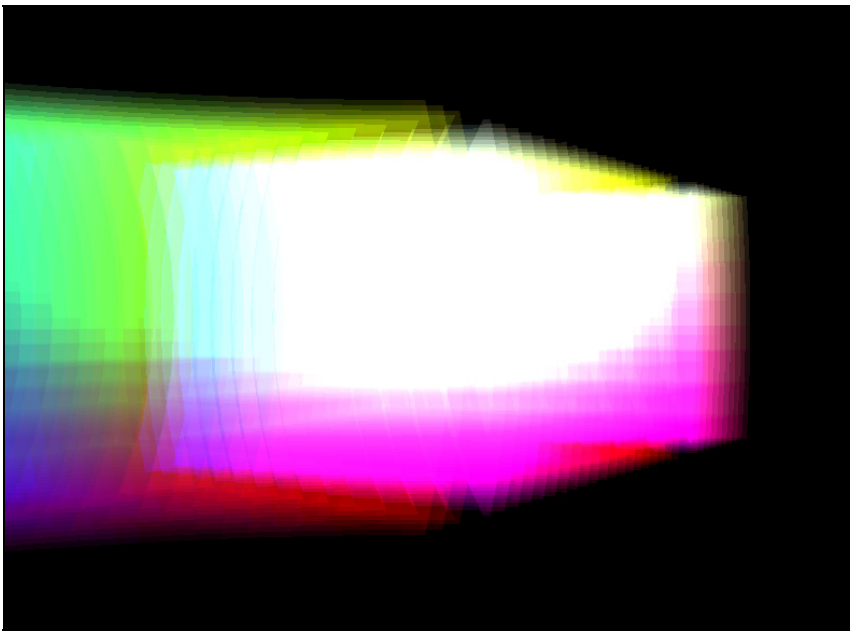

Рис. 5.9. Светящийся прямоугольник

5.9.1. Инициализация

Инициализация Direct3D будет производиться с помощью функции `DX3DInitZ`. После этого вызывается функция `CreateQuad`, в которой будет выполняться инициализация отображаемых объектов. Код этой функции представлен в листинге 5.21.

Листинг 5.21. Инициализация объектов

```
void CreateQuad()
{
 DWORD r, g, b, a;

 // Создание интерфейса объемной текстуры
 pD3DDevice->CreateVolumeTexture(32, 32, 32, 1, 0,
         D3DFMT_A8R8G8B8, D3DPOOL_MANAGED, &iTexture, 0);
```

```
D3DLOCKED_BOX lBox;
if (FAILED(iTexture->LockBox(0, &lBox, 0, 0)))
 return;

float du, dv, dw;
// Заполнение текстуры
for (UINT i=0; i < 32; i++)
{
 BYTE* pSliceStart = (BYTE*)lBox.pBits;
 for (UINT j=0; j< 32; j++)
 {
  for (UINT z=0; z<32; z++)
  {
   if ((i==0) || (i==31) || (j==0) || (j==31) || (z==0) || (z==31))
   {
    r = 0;
    g = 0;
    b = 0;
    a = 0;
   }
   else
   {
    r = (BYTE) ((i/31.0f)*255.0f);
    g = (BYTE) ((j/31.0f)*255.0f);
    b = (BYTE) ((z/31.0f)*255.0f);
    a = 0x00;
   }
   ((DWORD*)lBox.pBits)[z] = D3DCOLOR_ARGB(a, r, g, b);
  }
  lBox.pBits = (BYTE*)lBox.pBits + lBox.RowPitch;
 }

 lBox.pBits = pSliceStart + lBox.SlicePitch;
 iTexture->UnlockBox( 0 );
}

iTotalVerts = NUM_SHELLS * (NUM_CELLS + 1) * (NUM_CELLS + 1);
iTotalIndices = NUM_SHELLS * NUM_CELLS2 * 6;

if (FAILED(pD3DDevice->CreateVertexBuffer(
       iTotalVerts*sizeof(sVertex),
       D3DUSAGE_WRITEONLY, D3DFVF_XYZ | D3DFVF_DIFFUSE,
       D3DPOOL_MANAGED, &iVortexBuffer, 0)))
 return;
```

```
if (FAILED(pD3DDevice->CreateIndexBuffer(iTotalIndices*sizeof(WORD),
    D3DUSAGE_WRITEONLY, D3DFMT_INDEX16,
    D3DPOOL_MANAGED, &iIndexTexture,0)))
 return;

sVertex* pVertices;
float len, delta;
int x, y;
int Vert = 0;

// Создаем буфер вершин
iVortexBuffer->Lock( 0, iTotalVerts*sizeof(sVertex),
  (void**)&pVertices, 0 );
dw = NEAR_SHELL;

for(int i=0; i<NUM_SHELLS; i++)
{
 du = -tanf(iFOV/2.0f) * dw;
 delta = ((2.0f * tanf(iFOV/2.0f) * dw) / ((float)NUM_CELLS));

 for (x=0; x<=NUM_CELLS; x++)
 {
  dv = -tanf(iFOV/2.0f) * dw;
  for (y=0; y<=NUM_CELLS; y++)
  {
   len = sqrtf(du*du + dv*dv + dw*dw);

   pVertices[Vert].x = du / len;
   pVertices[Vert].y = dv / len;
   pVertices[Vert].z = dw / len;

   pVertices[Vert].x *= dw;
   pVertices[Vert].y *= dw;
   pVertices[Vert].z *= dw;
   pVertices[Vert].z *= -1.0f;

   pVertices[Vert].color = D3DCOLOR_RGBA((int)(255*0.5f),
        (int)(0.5f), (int)(fabs(pVertices[Vert].z)), (int)(1.0f));

   Vert++;
   dv += delta;
  }
  du += delta;
 }
```

```
  dw += (2.0f * ROOT_THREE) / ((float)NUM_SHELLS);
 }
 iVortexBuffer->Unlock();

 // Индексный буфер
 WORD* pIndex;
 WORD  Index;

 iIndexTexture->Lock(0, iTotalIndices*sizeof(WORD), (void**)&pIndex, 0);

 for (int i=NUM_SHELLS-1; i>=0; i--)
 {
  Index = (NUM_CELLS+1) * (NUM_CELLS+1) * i;

  for(x=0; x<NUM_CELLS; x++)
  {
   for(y=0; y<NUM_CELLS; y++)
   {
    *pIndex++ = (x + 0) * (NUM_CELLS+1) + (y + 0) + Index;
    *pIndex++ = (x + 0) * (NUM_CELLS+1) + (y + 1) + Index;
    *pIndex++ = (x + 1) * (NUM_CELLS+1) + (y + 1) + Index;

    *pIndex++ = (x + 0) * (NUM_CELLS+1) + (y + 0) + Index;
    *pIndex++ = (x + 1) * (NUM_CELLS+1) + (y + 1) + Index;
    *pIndex++ = (x + 1) * (NUM_CELLS+1) + (y + 0) + Index;
   }
  }
 }
 iIndexTexture->Unlock();

 // Изменяем матрицу проекции
 D3DXMATRIX matProjection;
 D3DXMatrixPerspectiveFovRH(&matProjection, iFOV, 1.0f, 2.0f, 1000.0f );
 pD3DDevice->SetTransform(D3DTS_PROJECTION, &matProjection);
}
```

Функция инициализации получилась не маленькой, но очень интересной и у нас будет что посмотреть.

В самом начале мы инициализируем переменную `iTexture`, которая является указателем на экземпляр интерфейса `IDirect3DVolumeTexture9`. Это интерфейс текстуры, но он отличается от того, что мы рассматривали ранее (`IDirect3DTexture9`), потому что интерфейс `IDirect3DVolumeTexture9` — это объемная текстура.

Для того чтобы создать интерфейс IDirect3DVolumeTexture9, необходимо воспользоваться методом CreateVolumeTexture. Этот метод имеет те же параметры, что и CreateTexture, который мы использовали в *разд. 5.6.5*, только здесь нужно указать не только ширину и высоту текстуры, но и глубину. В данном примере все эти значения (первые три параметра) равны 32.

После создания интерфейса необходимо заблокировать его поверхность для заполнения растровыми данными текстуры. Вот тут нужно быть внимательным, потому что для блокировки используется метод LockBox, а не LockRect. Количество параметров у обоих методов одинаковы, а разница заключается в том, что метод LockBox блокирует трехмерный куб, а не плоский прямоугольник, ведь создаваемая нами текстура объемна и имеет три измерения.

Параметры у метода LockBox схожи с LockRect, только нужно указать не плоский прямоугольник блокируемой поверхности, а трехмерный куб с помощью структуры типа D3DBOX, которая выглядит следующим образом:

```
typedef struct _D3DBOX {
    UINT Left;
    UINT Top;
    UINT Right;
    UINT Bottom;
    UINT Front;
    UINT Back;
} D3DBOX;
```

Как видите, помимо привычных параметров, определяющих область прямоугольника, есть два параметра Front и Back, которые определяют ближнюю и дальнюю позицию прямоугольника. Нам необходимо заполнять всю поверхность, поэтому все параметры, кроме второго, равны нулю.

В качестве второго параметра необходимо передать указатель на структуру типа D3DLOCKED_BOX. Через нее мы получим параметры заблокированной поверхности. Структура имеет следующий вид:

```
typedef struct _D3DLOCKED_BOX {
    int RowPitch;
    int SlicePitch;
    void *pBits;
} D3DLOCKED_BOX;
```

В структуре используются три параметра:

- RowPitch — смещение в байтах от левого края одной строки до левого края другой строки;

- SlicePitch — смещение в байтах от верхнего левого края одного слоя до верхнего левого другого слоя;
- pBits — указатель на память, где находятся растровые данные структуры.

После блокировки поверхности запускаются циклы, которые перебирают все точки текстуры. Для этого нам необходимо три цикла, один перебирает все значения координаты X, второй координату Y и последний — координату Z текстуры. В нашем случае используются три цикла, а соответствующие координаты определяются переменными i, j и z. Наверно лучше было бы заменить i на x, а j на y, но я привык для циклов использовать именно переменные i и j.

На каждом этапе цикла определяем цвет точки текстуры. Если текущая точка крайняя по отношению к любому краю куба, который описывает поверхность, то цвет точки будет равен нулю. Если нет, то для определения цвета используется формула:

```
Красный цвет = i/31*255;
```

Тем же макаром определяются и остальные составляющие цвета, только для зеленого переменная i меняется на j, а для голубого цвета на z.

Смысл формулы прост — каждая координата определяется значениями от 0 до 31, а мы как бы переводим ее в формат от 0 до 255, точно так же, как мы переводили цветовые составляющие из 16-битного вида к 24-битному.

В результате, полученная текстура будет состоять из точек, цвет которых будет плавно переходить от черного в левом верхнем ближнем углу до белого в правом нижнем дальнем углу. Это будет что-то типа трехмерного градиента.

Теперь рассчитываем количество вершин и необходимых индексов для построения фигуры и создаем интерфейсы для хранения буфера вершин и буфера индексов. В создании буферов вершин и индексов нет ничего нового, и все это мы уже видели.

Теперь блокируем буфер вершин, и буфер индексов заполняем их.

Самое последнее, что необходимо сделать, — изменить матрицу проекции. По умолчанию функция DX3DInitZ инициализирует матрицу так, чтобы соотношение сторон было равно 1 (третий параметр функции D3DXMatrixPerspectiveFovRH). К тому же, мы используем функцию D3DXMatrixPerspectiveFovRH, а не D3DXMatrixPerspectiveFovLH:

```
D3DXMATRIX matProjection;
D3DXMatrixPerspectiveFovRH(&matProjection, iFOV,
       1.0f, 2.0f, 1000.0f );
pD3DDevice->SetTransform(D3DTS_PROJECTION, &matProjection);
```

В чем отличие между функциями D3DXMatrixPerspectiveFovRH и D3DXMatrix-PerspectiveFovLH. Помимо предпоследней буквы в названии функции отличие заключается в создаваемой перспективе. Первая создает правую перспективу (right-handed), а вторая левую (left-handed). При левосторонней системе координата Z направлена от наблюдателя, а при правосторонней на наблюдателя. В данном случае нам будет удобнее работать с правосторонней системой.

5.9.2. Отображение

Теперь посмотрим на функцию отображения (листинг 5.22).

Листинг 5.22. Функция отображения фигуры с использованием 3D-текстуры

```
void DrawScene()
{
 fViewAngle += 0.01f;

 // Рассчитываем матрицу преобразований
 D3DXMATRIX MatTex, MatTexRot;
 D3DXMatrixRotationY(&MatTexRot, fViewAngle);

 D3DMATRIX View = {
  0.5, 0, 0, 0,
  0, 0.5, 0, 0,
  0, 0, 0.5, 0,
  0.5, 0.5, 0.5*CENTER, 1,
 };

 D3DXMatrixIdentity(&MatTex);
 D3DXMatrixMultiply(&MatTex, (D3DXMATRIX*)&View, &MatTexRot);

 // Устанавливаем матрицу перемещений текстуре
 pD3DDevice->SetTransform(D3DTS_TEXTURE0,  &MatTex);

 // Выбираем текстуру
 pD3DDevice->SetTexture(0, iTexture);

 // Устанавливаем буфер вершин и индексный буфер
 pD3DDevice->SetStreamSource( 0, iVortexBuffer, 0, sizeof(sVertex) );
 pD3DDevice->SetIndices(iIndexTexture);

 // Настраиваем параметры текстуры
 pD3DDevice->SetTextureStageState(0, D3DTSS_COLOROP,   D3DTOP_SELECTARG1);
 pD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAARG1, D3DTA_TEXTURE);
```

```
 pD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAARG2, D3DTA_TFACTOR);
 pD3DDevice->SetTextureStageState(0, D3DTSS_ALPHAOP, D3DTOP_SELECTARG2);
 pD3DDevice->SetRenderState(D3DRS_TEXTUREFACTOR,
               D3DCOLOR_ARGB(0x22, 0x66, 0x33, 0x99));

 // Включаем прозрачность
 pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
 pD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCALPHA);
 pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);

 // Снова настройки текстуры
 pD3DDevice->SetTextureStageState(0, D3DTEXTURESTAGESTATETYPE(13),
               D3DTADDRESS_CLAMP);
 pD3DDevice->SetTextureStageState(0, D3DTEXTURESTAGESTATETYPE(14),
               D3DTADDRESS_CLAMP);

pD3DDevice->SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
               D3DTTFF_COUNT3);

 // Отображение объекта
 pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE);
 pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0,
               iTotalVerts, 0, NUM_CELLS2*2*NUM_SHELLS);

 // Отключаем освещение и текстуру
 pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
 pD3DDevice->SetTexture(0, NULL);
}
```

Рассмотрим, что происходит в этой функции. Тем более, есть на что посмотреть. Первая строка увеличивает переменную `fViewAngle` на 0.01. Это угол, на который нужно повернуть сцену.

При каждом отображении сцены нам придется выполнять аж три операции — масштабирование, перемещение сцены и поворот. Для выполнения всего этого нужно перемножать матрицы, но как это сделать? Можно заранее выполнить эти операции, ведь если знать как, то перемножить можно "за пять секунд". Но мы лучше посмотрим процесс перемножения.

Для начала объявляем две переменные для хранения матрицы: `MatTex` и `MatTexRot`. В первую переменную будет записан результат, который нужно установить, а вторая переменная будет содержать матрицу поворота сцены. Обратите внимание, что обе переменные имеют тип `D3DXMATRIX`, а не привычный нам `D3DMATRIX`. Разница в этих структурах небольшая, но есть.

Чтобы установить матрицу поворота, используем функцию D3DXMatrixRotationY. Этой функции необходимо передать два параметра: матрицу, которую нужно установить, и угол, на который требуется повернуть сцену вокруг оси Y. Если нужно повернуть сцену вокруг оси X, то необходимо использовать функцию D3DXMatrixRotationX, а для поворота вокруг оси Z применяется функция D3DXMatrixRotationZ. Все они имеют одни и те же параметры и выполняются одинаковым образом.

Далее объявляем матрицу, которая будет позиционировать объект в пространстве:

```
D3DMATRIX View = {
 0.5, 0, 0, 0,
 0, 0.5, 0, 0,
 0, 0, 0.5, 0,
 0.5, 0.5, 0.5*CENTER, 1,
};
```

Теперь нужно объединить две матрицы. Для этого используется функция D3DXMatrixMultiply. Ей передается три параметра, которые имеют тип D3DXMATRIX:

- результирующая матрица;
- первая матрица преобразований;
- вторая матрица преобразований.

В матрицу, указанную в первом параметре, будет записан результат объединения матриц из второго и третьего параметра. В нашем случае объединяются матрицы View и MatTexRot, а результат записывается в MatTex. При этом матрица View имеет тип D3DMATRIX и ее необходимо привести к типу D3DXMATRIX:

```
D3DXMatrixMultiply(&MatTex, (D3DXMATRIX*)&View, &MatTexRot);
```

Теперь можно использовать матрицу MatTex. Раньше с помощью матриц мы модифицировали положения объектов, а в данном случае объект у нас будет неподвижным, изменяться же будет именно текстура:

```
pD3DDevice->SetTransform(D3DTS_TEXTURE0,  &MatTex);
```

Действительно, с помощью метода SetTransform можно изменять положение не только объектов, но и текстуры. После этого текстуру можно выбирать и использовать.

5.9.3. Настройки текстуры

Далее следует множество настроек отображения текстуры с помощью метода SetTextureStageState. При помощи этого метода можно указать великое

множество различных настроек и разобраться в них будет непросто. Метод SetTextureStageState в общем виде выглядит следующим образом:

```
HRESULT SetTextureStageState(
  DWORD Stage,
  D3DTEXTURESTAGESTATETYPE Type,
  DWORD Value
);
```

Здесь имеется три параметра:

- Stage — ступень текстуры. Всего возможно 8 ступеней, поэтому значение этого параметра может изменяться от 0 до 7;
- Type — тип состояния текстуры, значение которого нужно установить;
- Value — значение, которое нужно установить состоянию. Возможные значения этого параметра зависят от того, какой тип состояния устанавливается, т. е. от значения второго параметра Type.

Давайте немного подробнее остановимся на втором параметре. Он имеет тип D3DTEXTURESTAGESTATETYPE, а это не что иное, как перечисление (enum):

```
typedef enum _D3DTEXTURESTAGESTATETYPE {
  D3DTSS_COLOROP = 1,
  D3DTSS_COLORARG1 = 2,
  D3DTSS_COLORARG2 = 3,
  D3DTSS_ALPHAOP = 4,
  D3DTSS_ALPHAARG1 = 5,
  D3DTSS_ALPHAARG2 = 6,
  D3DTSS_BUMPENVMAT00 = 7,
  D3DTSS_BUMPENVMAT01 = 8,
  D3DTSS_BUMPENVMAT10 = 9,
  D3DTSS_BUMPENVMAT11 = 10,
  D3DTSS_TEXCOORDINDEX = 11,
  D3DTSS_BUMPENVLSCALE = 22,
  D3DTSS_BUMPENVLOFFSET = 23,
  D3DTSS_TEXTURETRANSFORMFLAGS = 24,
  D3DTSS_COLORARG0 = 26,
  D3DTSS_ALPHAARG0 = 27,
  D3DTSS_RESULTARG = 28,
  D3DTSS_CONSTANT = 32,
  D3DTSS_FORCE_DWORD = 0x7fffffff
} D3DTEXTURESTAGESTATETYPE;
```

Вот тут перечислено все, что можно установить. Хотя нет, не все. Если вы посмотрите на числа элементов перечислений, то они идут не последователь-

но. Дело в том, что некоторые параметры уже не используются. И все же, их можно применить. Так, например, в DirectX8 есть два хороших параметра D3DTSS_ADDRESSU и D3DTSS_ADDRESSV, которым соответствуют значения 13 и 14. Чтобы использовать их в нашем примере, мы просто устанавливаем параметры по значению:

```
pD3DDevice->SetTextureStageState(0, D3DTEXTURESTAGESTATETYPE(13),
                D3DTADDRESS_CLAMP);
pD3DDevice->SetTextureStageState(0, D3DTEXTURESTAGESTATETYPE(14),
                D3DTADDRESS_CLAMP);
```

Все возможные параметры мы рассмотреть не сможем, а некоторые все же посмотрим:

- D3DTSS_COLOROP — операция смешивания цвета текстуры определяется по одному из членов перечисления D3DTEXTUREOP. Нужное значение из этого перечисления необходимо указать в третьем параметре функции SetTextureStageState;
- D3DTSS_ALPHAOP — операция альфа-смешивания текстуры. В третьем параметре может быть одно из значений перечисления D3DTEXTUREOP;
- D3DTSS_ALPHAARG1 — первый альфа-аргумент. В третьем параметре указывается один из флагов D3DTA. По умолчанию используется значение D3DTA_TEXTURE;
- D3DTSS_ALPHAARG2 — второй альфа-аргумент. В третьем параметре указывается один из флагов D3DTA. По умолчанию используется флаг D3DTA_CURRENT;
- D3DTSS_BUMPENVMAT00 — третий параметр — это вещественное число, определяющее коэффициент шероховатости в точке 0:0;
- D3DTSS_BUMPENVMAT01 — третий параметр — это вещественное число, определяющее коэффициент шероховатости (bump-mapping) в точке 0:1;
- D3DTSS_BUMPENVMAT10 — третий параметр — это вещественное число, определяющее коэффициент шероховатости (bump-mapping) в точке 1:0;
- D3DTSS_BUMPENVMAT11 — третий параметр — это вещественное число, определяющее коэффициент шероховатости (bump-mapping) в точке 1:1.

5.9.4. Завершающая стадия

Далее включаем прозрачность. Это мы делали уже не раз, поэтому дополнительные комментарии к следующему коду не нужны:

```
pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
pD3DDevice->SetRenderState(D3DRS_SRCBLEND,  D3DBLEND_SRCALPHA);
pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);
```

И самое последнее — отображаем объект. После отображения отключаем прозрачность и обнуляем текущую прозрачность. Это необходимо, особенно в том случае, если вы захотите выводить другие объекты, которые не должны обладать возможностями прозрачности и использовать указанную текстуру.

В качестве дополнения, могу посоветовать убрать из примера следующую строку:

```
pD3DDevice->SetTextureStageState(0, D3DTEXTURESTAGESTATETYPE(14),
   D3DTADDRESS_CLAMP);
```

Потом запустите пример и посмотрите на результат. Он будет достаточно интересным.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\LightObject.

5.9.5. Точечные текстуры

В качестве "контрольного выстрела" рассмотрим еще один интересный пример реализации эффекта свечения. Скриншот тут приводить бесполезно, потому что черно-белая полиграфия просто не сможет передать этой красоты. Это необходимо видеть в реалее.

Этот эффект свечения мы достигнем благодаря тому, что натянем текстуру на точку. Да, именно на точку, и в Direct3D это вполне возможно. Для реализации примера я взял код из *разд. 2.10*, где мы создавали вращающийся куб. Заменим вывод куба из треугольников на простое отображение вершин:

```
pD3DDevice->DrawIndexedPrimitive(D3DPT_POINTLIST, 0, 0,
       iVertsNum, 0, iPointsNum);
```

Для отображения куба использовалась та же самая строка, только первый параметр был `D3DPT_TRIANGLELIST`, что заставляло Direct3D воспринимать вершины в буфере по три штуки, формируя треугольники. Мы заменили это значение на `D3DPT_POINTLIST`, и теперь каждая вершина будет отдельной точкой.

Точка сама по себе будет слишком маленькой, поэтому нужно увеличить ее размер. Для этого выполняем следующий код на этапе инициализации переменных (но после инициализации Direct3D):

```
DWORD ps;
pD3DDevice->GetRenderState(D3DRS_POINTSIZE, &ps);
pD3DDevice->SetRenderState(D3DRS_POINTSIZE, ps*1.02);
```

В первой строке объявляем переменную типа DWORD. В такую переменную будет записан текущий размер точки. Во второй строке определяем текущий размер. Для этого вызываем метод GetRenderState, в первом параметре которого необходимо указать флаг D3DRS_POINTSIZE, а во втором — переменную типа DWORD, в которую будет записан результат.

Последняя строка устанавливает новое значение с помощью метода SetRenderState. Первый параметр снова D3DRS_POINTSIZE, а второй — равен текущему значению размера точки, умноженному на 1.02.

Как загрузить текстуру, я думаю, уже объяснять не надо. Просто смотрим и наслаждаемся:

```
D3DXCreateTextureFromFileEx(pD3DDevice, "Particle.bmp",
       D3DX_DEFAULT, D3DX_DEFAULT, D3DX_DEFAULT, 0,
       D3DFMT_UNKNOWN, D3DPOOL_MANAGED,
       D3DX_FILTER_TRIANGLE|D3DX_FILTER_MIRROR,
       D3DX_FILTER_TRIANGLE|D3DX_FILTER_MIRROR,
       0, NULL, NULL, &pTexture);
```

Загрузка происходит из растрового файла Particle.bmp в переменную pTexture, которая имеет тип IDirect3DTexture9.

Теперь переходим к отображению. Следующий код должен находиться в функции DrawScene:

```
// Установка вершинного буфера и формата вершин
pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE);
pD3DDevice->SetStreamSource(0, vBuffer, 0, sizeof(sVertex));
pD3DDevice->SetIndices(iBuffer);

// Включаем альфа-смешивание
pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
pD3DDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_ONE);
pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_ONE);

// Включаем возможность текстурирования и масштабирования вершин
pD3DDevice->SetRenderState( D3DRS_POINTSPRITEENABLE, TRUE);
pD3DDevice->SetRenderState( D3DRS_POINTSCALEENABLE,  TRUE);

// Выбираем текущую текстуру
pD3DDevice->SetTexture(0, pTexture);

// Отображаем вершинный буфер
pD3DDevice->DrawIndexedPrimitive(D3DPT_POINTLIST, 0, 0,
          iVertsNum, 0, iPointsNum);
```

```
// Отключаем сделанные настройки
pD3DDevice->SetRenderState( D3DRS_POINTSPRITEENABLE, FALSE );
pD3DDevice->SetRenderState( D3DRS_POINTSCALEENABLE,  FALSE );
pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, AFALSE);
```

Перед этим кодом можно установить мировую матрицу, поворачивая ее, чтобы точки вращались как светомузыка на дискотеке.

Теперь посмотрим, что происходит во время отображения. В самом начале мы устанавливаем буфер вершин, настраиваем формат и устанавливаем индексы. На самом деле, индексы не нужны, ведь они показывают, в каком порядке связывать вершины, а мы выводим их независимо, и поэтому связей нет. В данном случае индексный буфер остался как наследие примера, который мы модифицируем.

Следующим этапом устанавливаем альфа-смешивание. Именно оно придает нашей сцене дополнительную эффектность, а на черном фоне за счет смешения получится эффект свечения разноцветных лампочек. Кстати о птичках. Для этого примера не должно быть теста глубины. Проблема решается инициализацией Direct3D с помощью функции `DX3DInit`, а не `DX3DInitZ`, которую мы использовали в примерах в последнее время.

После этого устанавливается возможность текстурирования точек. Для этого необходимо вызвать метод `SetRenderState`, в качестве первого параметра указать флаг `D3DRS_POINTSPRITEENABLE`, а второй параметр должен быть равен `TRUE`. Именно флаг `D3DRS_POINTSPRITEENABLE` разрешает текстурировать даже отдельно стоящие вершины.

После этого, таким же методом устанавливаем свойство `D3DRS_POINTSCALEENABLE` в `TRUE`. Этот флаг разрешает масштабировать вершины.

Выполнив все настройки, устанавливаем в качестве текущей текстуру, загруженную в переменную `pTexture`.

Далее идет самое простое — отображение буфера вершин и отключения всех настроек, которые мы включили. В данном случае настройки можно не отключать, потому что мы выводим только эти вершины, но если на вашей сцене будут и другие объекты, которые не должны обладать указанными свойствами, то отключение необходимо.

Запустите пример и насладитесь его красотой.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Points.

5.10. Отображение на текстуре

Динамическое создание текстур является очень мощным эффектом, но создавать текстуру вручную достаточно сложно и накладно. В этом случае приходится на каждом этапе цикла перезаполнять поверхность текстуры по определенному алгоритму, который рассчитывается центральным процессором, нагрузка на который в играх и демо-роликах итак слишком высокая. Необходим более эффективный метод динамического создания текстуры, и возложить труд по расчетам сцены на процессор видеокарты, который разрабатывался специально для работы с графикой, и выполнить работу лучше и быстрее. Центральный процессор в это время может заниматься другими полезными расчетами.

Итак, давайте создадим пример, в котором текстура будет создаваться динамически. Как это будет происходить? Мы загрузим сетку, которая будет формировать чайник из файла teapot.x, этот файл можно найти в составе DirectX SDK. Но чайник будет выводиться не на экран, а на текстуру. Таким образом, мы будем рисовать сетку не на экране, а на поверхности текстуры, а затем эта текстура будет натягиваться на поверхность сферы. Код должен быть очень интересным, а результат красивым (рис. 5.10). А главное, что рисование поддерживается современными видеокартами аппаратно.

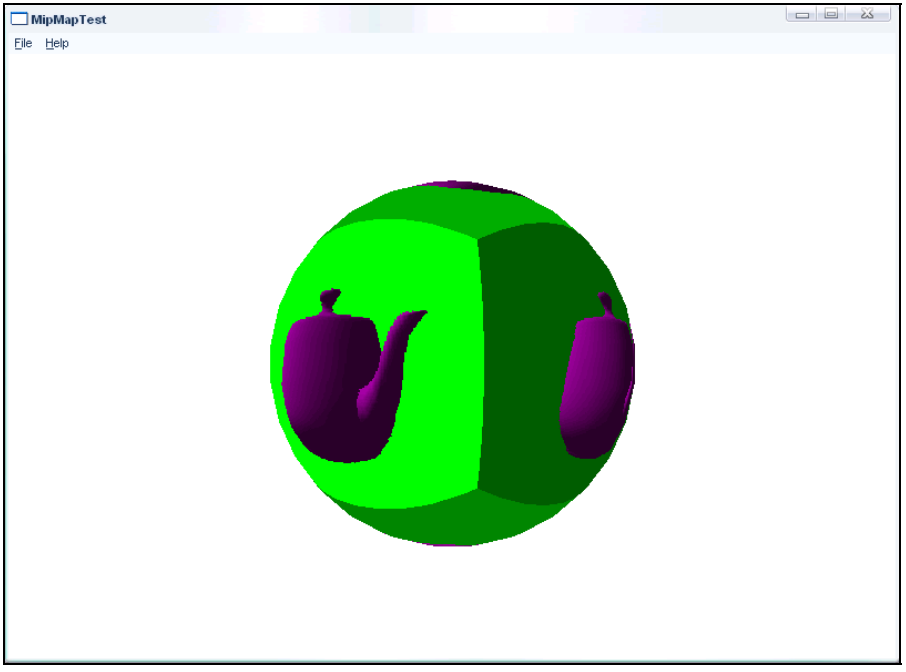

Рис. 5.10. Результат работы отображения чайника на текстуре

5.10.1. Подготовка

Для реализации примера создаем новый проект и добавим инициализацию Direct3D с помощью функции `DX3DInitZ`. Помимо этого нам понадобятся несколько вспомогательных переменных.

Для начала объявим структуру, описывающую тип вершины, из которых будет строиться сфера:

```
struct sVertex
{
    D3DXVECTOR3 p;
    D3DXVECTOR3 n;
    static const DWORD FVF;
};
```

Теперь нам понадобится структура, которая будет описывать сетку чайника:

```
struct TEAPOT_VERTEX
{
    D3DXVECTOR3 p;
    D3DXVECTOR3 n;
    static const DWORD FVF;
};
```

В принципе, описывать всю структуру не обязательно, потому что нам необходим только ее размер, который можно было бы просто подсчитать заранее. Но я решил показать вам именно структуру, чтобы было видно, какой формат вершин используется для вывода объекта на экран.

Теперь нам нужны переменные, в которые будут загружаться сетка, материалы и текстура этой сетки:

```
ID3DXMesh* mTeapot;
LPDIRECT3DTEXTURE9 *pMeshTextures;
D3DMATERIAL9 *pMeshMaterials;
```

Материал чайника из X-файла использовать не будем. Лучше назначим собственный материал, сохраним его в переменной `TeapotMat`, и сделаем его сиреневого цвета.

Теперь нам понадобится буфер вершин и текстура для сферы. Для этого заведем следующие переменные:

```
IDirect3DVertexBuffer9 * vbSphere;
IDirect3DCubeTexture9 * cubeTexture;
```

И наконец, мы будем изменять поверхность, на которой необходимо рисовать. По умолчанию вывод объектов происходит на поверхность заднего бу-

фера, а мы будем переключать вывод на текстуру. После этого нужно будет вернуть вывод на задний буфер, поэтому понадобится переменная, где мы будем сохранять указатель на указатель заднего буфера:

```
IDirect3DSurface9 *  pBackBuffer;
```

5.10.2. Инициализация

Теперь переходим к рассмотрению инициализации переменных и загрузки чайника. Код инициализации можно увидеть в листинге 5.23.

Листинг 5.23. Инициализация переменных

```
DWORD dwNumMaterials = LoadMesh("teapot.x", pD3DDevice, &mTeapot,
    &pMeshTextures, "texture.bmp", &pMeshMaterials);

// Инициализация сферы
DWORD dwSphereRings    = 20;
DWORD dwSphereSegments = 20;
dwSphereVerticesNum = 2 * dwSphereRings * (dwSphereSegments+1);

// Создаем вершинный буфер
pD3DDevice->CreateVertexBuffer(dwSphereVerticesNum*sizeof(sVertex),
    D3DUSAGE_WRITEONLY, 0, D3DPOOL_MANAGED, &vbSphere, NULL);

// Заполнение вершинного буфера сферы
sVertex* vert;
vbSphere->Lock( 0, 0, (void**)&vert, 0 );

float fDeltaRingAngle = ( D3DX_PI / dwSphereRings );
float fDeltaSegAngle  = ( 2.0f * D3DX_PI / dwSphereSegments );

D3DXVECTOR4 vT;
for (int rings=0; rings<dwSphereRings; rings++)
{
 float r0 = sinf((rings+0) * fDeltaRingAngle);
 float r1 = sinf((rings+1) * fDeltaRingAngle);
 float y0 = cosf((rings+0) * fDeltaRingAngle);
 float y1 = cosf((rings+1) * fDeltaRingAngle);

 for (int segs=0; segs<dwSphereSegments+1; segs++)
 {
  float x0 = r0 * sinf(segs * fDeltaSegAngle);
  float z0 = r0 * cosf(segs * fDeltaSegAngle);
```

```
  float x1 = r1 * sinf(segs * fDeltaSegAngle);
  float z1 = r1 * cosf(segs * fDeltaSegAngle);

  (*vert).p = (*vert).n = D3DXVECTOR3(x0,y0,z0);
  vert++;

  (*vert).p = (*vert).n = D3DXVECTOR3(x1,y1,z1);
  vert++;
 }
}

vbSphere->Unlock();

// Инициализация материалов
ZeroMemory( &TeapotMat, sizeof(D3DMATERIAL9) );
TeapotMat.Diffuse.r = TeapotMat.Ambient.r = 0.5f;
TeapotMat.Diffuse.g = TeapotMat.Ambient.g = 0.0f;
TeapotMat.Diffuse.b = TeapotMat.Ambient.b = 0.5f;
TeapotMat.Diffuse.a = TeapotMat.Ambient.a = 0.5f;

// Инициализация освещения
D3DLIGHT9 light;
ZeroMemory( &light, sizeof(D3DLIGHT9) );
light.Type = D3DLIGHT_DIRECTIONAL;
light.Diffuse.r = 1.0f;
light.Diffuse.g = 1.0f;
light.Diffuse.b = 1.0f;
light.Position.x = -0.4f;
light.Position.y = -0.4f;
light.Position.z = 0.0f;
light.Range = 1000.0f;
D3DXVECTOR3 vecLightDirUnnormalized(-0.4, -0.4, 0.0);
D3DXVec3Normalize((D3DXVECTOR3*)&light.Direction,
   &vecLightDirUnnormalized);
pD3DDevice->SetLight(0, &light);
pD3DDevice->LightEnable(0, TRUE);
pD3DDevice->SetRenderState(D3DRS_AMBIENT, 0xff555555);

// Установка матрицы просмотра
D3DMATRIX View = {
   1, 0, 0, 0,
   0, 1, 0, 0,
   0, 0, 1, 0,
   0, 0, 4.5, 1,
};
```

```
pD3DDevice->SetTransform(D3DTS_VIEW, &View);

pD3DDevice->GetBackBuffer (0, 0, D3DBACKBUFFER_TYPE_MONO, &pBackBuffer);

// Инициализация текстуры
pD3DDevice->CreateCubeTexture (256, 0,
     D3DUSAGE_RENDERTARGET|D3DUSAGE_AUTOGENMIPMAP,
     D3DFMT_A8R8G8B8, D3DPOOL_DEFAULT , &cubeTexture, NULL);
```

В последнее время размер кода инициализации в наших примерах превышает размер кода отображения. Давайте посмотрим, что здесь происходит.

В самом начале загружаем сетку чайника с помощью функции `LoadMesh`. Вот тут проявляется недостаток этой функции, ведь она полностью разбирает загруженный объект по материалам и текстурам, которые мы не будем использовать. Это лишние затраты и лишние переменные. Можно изменить функцию `LoadMesh` и перед разбором сетки проверять, если переменные для хранения материала и текстуры нулевые, то не производить разбора. Но лучше написать еще один вариант функции загрузки сетки, который будет принимать в качестве параметров только имя файла сетки, указатель на устройство `IDirect3DDevice9` и указатель на интерфейс `ID3DXMesh` для хранения сетки. Реализацию вышесказанного оставлю на вашей совести, пускай она вас грызет, а не меня ☺.

Далее, создается вершинный буфер с помощью уже знакомого нам метода `CreateVertexBuffer`. Тут ничего нового нет, поэтому быстро движемся дальше. А дальше идет блокирование созданного буфера и заполнение его вершинами сферы. Я не стал выдумывать тут велосипед, а воспользовался одним из популярных алгоритмов формирования сферы. Для удобства, перед заполнением буфера введены три переменные:

```
DWORD dwSphereRings = 20;
DWORD dwSphereSegments = 20;
dwSphereVerticesNum = 2 * dwSphereRings * (dwSphereSegments+1);
```

Первая переменная (`dwSphereRings`) определяет количество колец, из которых будет формироваться сфера. Это как бы количество экваторов по горизонтали. Переменная `dwSphereSegments` определяет количество экваторов по вертикали. Чем больше их будет, тем более гладкой получится сфера, но и больше понадобится ресурсов для ее вывода. Поэтому необходимо выбрать оптимальное значение качества и скорости. Для нашего примера обе переменные будут равны 20.

Двигаемся дальше. У нас есть сетка чайника и вершинный буфер сферы. Теперь инициализируем материал сиреневого цвета (0.5, 0.0, 0.5), которым

окрашивается чайник, и направленный источник света, который должен освещать чайник и придавать ему больше реалистичности. Источник света будет освещать сцену слева, поэтому делаем этот источник нулевым и включаем его.

После этого включаем освещение всей сцены серым источником света (55, 55, 55):

```
pD3DDevice->SetRenderState(D3DRS_AMBIENT, 0xff555555);
```

Далее, устанавливаем матрицу отображения, чтобы увидеть нашу сферу.

Нам еще необходимо узнать, где находится задний буфер, и получить на него указатель. Для этого используется метод `GetBackBuffer`:

```
pD3DDevice->GetBackBuffer(0, 0, D3DBACKBUFFER_TYPE_MONO,
    &pBackBuffer);
```

Этот метод мы еще не использовали, поэтому посмотрим, как он выглядит:

```
HRESULT GetBackBuffer(
   UINT iSwapChain,
   UINT BackBuffer,
   D3DBACKBUFFER_TYPE Type,
   IDirect3DSurface9 **ppBackBuffer
);
```

Здесь имеются четыре параметра:

- `iSwapChain` — число, определяющее цепочку смены буферов;
- `BackBuffer` — индекс заднего буфера, который нужно получить;
- `Type` — тип буфера. В DirectX9 поддерживается только тип `D3DBACKBUFFER_TYPE_MONO`;
- `ppBackBuffer` — указатель на указатель интерфейса `IDirect3DSurface9`, где необходимо сохранить задний буфер.

У нас только один задний буфер, поэтому первые два параметра равны нулю, поэтому мы получим единственно существующий интерфейс поверхности заднего буфера.

Самое последнее, что мы делаем — создаем кубическую текстуру. Для этого используется метод `CreateCubeTexture`, который в общем виде выглядит следующим образом:

```
HRESULT CreateCubeTexture(
  UINT EdgeLength,
  UINT Levels,
  DWORD Usage,
  D3DFORMAT Format,
```

```
  D3DPOOL Pool,
  IDirect3DCubeTexture9 **ppCubeTexture,
  HANDLE* pHandle
);
```

Рассмотрим параметры этого метода, потому что мы его еще не видели. В принципе все параметры, кроме первого, нам знакомы и они идентичны методу `CreateTexture`, который создает плоскую текстуру. Разница только в первом параметре. Если же для определения размера текстуры в методе `CreateTexture` требуется два параметра (ширина и высота), то для кубической необходимо только одно значение — параметр `EdgeLength`. Этот параметр определяет размер грани куба.

В нашем случае размер грани равен 256. Это значит, что куб текстуры будет иметь ширину, высоту и глубину 256 пикселов.

5.10.3. Отображение текстуры с анимацией

Теперь посмотрим на функцию `DrawScene`, которая будет отображать сцену. Я думаю, не надо объяснять, как должна вызываться эта функция? Да, точно так же, как и во всех предыдущих примерах, т. е. по событию `WM_PAINT` между вызовами методов `BeginScene` и `EndScene`. Итак, функция отображения сцены `DrawScene` показана в листинге 5.24.

Листинг 5.24. Функция отображения сцены DrawScene

```
void DrawScene()
{
 D3DXMATRIXA16 matTeapotWorld;
 float fNear = 1.1f;
 float fFar  = 100.0f;

 fTime +=0.01;
 // Поворачиваем матрицу matTeapotWorld
 D3DXMatrixRotationY(&matTeapotWorld, fmodf(2.0f*fTime, D3DX_PI*2.0f));
 pD3DDevice->SetTransform(D3DTS_WORLD, &matTeapotWorld);

 IDirect3DSurface9 *pTextureSurfaceLevel;

 // Цикл рисования чайника на каждой грани куба
 for (int i=0; i<6; i++)
 {
  // Получаем указатель на поверхность текущей грани
  cubeTexture->GetCubeMapSurface( (D3DCUBEMAP_FACES)i,0,
         &pTextureSurfaceLevel);
  cubeTexture->Release();
```

```
 // Устанавливаем вывод графики на текущую текстуру
 pD3DDevice->SetRenderTarget (0, pTextureSurfaceLevel);

 // Очищаем поверхность текстуры
 pD3DDevice->Clear(0, 0, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        D3DCOLOR_XRGB(0, 255-40*i, 0), 1.0f, 0L);

 // Включаем освещение и устанавливаем материал
 pD3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE);
 pD3DDevice->SetMaterial(&TeapotMat);

 LPDIRECT3DVERTEXBUFFER9 pVB;
 LPDIRECT3DINDEXBUFFER9 pIB;

 // Получаем буфер вершин и индексный буфер чайника
 mTeapot->GetVertexBuffer(&pVB);
 mTeapot->GetIndexBuffer(&pIB);

 // Определяем размер буфера вершин и индексного буфера
 DWORD numVertices = mTeapot->GetNumVertices();
 DWORD numFaces = mTeapot->GetNumFaces();

 // Устанавливаем параметры отображения объекта сетки
 pD3DDevice->SetFVF(mTeapot->GetFVF());
 pD3DDevice->SetStreamSource(0, pVB, 0, sizeof(sVertexTeapot));
 pD3DDevice->SetIndices(pIB);

 // Отображаем чайник на грани текстуры
 pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0,
         numVertices, 0, numFaces);

 // Освобождаем буфер вершин и индексный буфер
 if (pVB){pVB->Release; pVB=NULL;}
 if (pIB){pIB->Release; pIB=NULL;}
}

// Теперь будем выводить на задний буфер сцены
pD3DDevice->SetRenderTarget (0, pBackBuffer);

// Очищаем поверхность
pD3DDevice->Clear (0L, NULL, D3DCLEAR_TARGET, 0xFFFFFFFF, 1.0f, 0L);
// Отключаем свет
pD3DDevice->SetRenderState(D3DRS_LIGHTING, FALSE);
```

```
 // Настраиваем и устанавливаем кубическую текстуру
 pD3DDevice->SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
          D3DTTFF_COUNT3);
 pD3DDevice->SetTexture (0, cubeTexture);

 // Устанавливаем буфер вершин сферы и параметры вершины
 pD3DDevice->SetStreamSource(0, vbSphere, 0, sizeof(sVertex));
 pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_TEXCOORDSIZE3(1));

 // Отображаем сферу
 pD3DDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0,
              dwSphereVerticesNum-2 );

 pD3DDevice->SetTexture (0, NULL);
}
```

Давайте рассмотрим этот код и увидим, как происходит рисование на текстуре. В самом начале функции `DrawScene` поворачиваем мировую матрицу вокруг оси Y с помощью функции `D3DXMatrixRotationY`. Вот тут нужно заметить, что эта матрица будет поворачивать не сферу. А что же тогда, если у нас больше нет объектов на сцене? А как же чайник? Да, именно чайник будет вращаться. Таким образом, будет создаваться эффект, в котором текстура состоит из вращающегося вокруг оси Y изображения чайника.

Теперь запускаем цикл от 0 до 5. Внутри цикла будем отображать одну "сторону" сферы. Что-то я какую-то глупость сказал, потому что у сферы не может быть стороны. Но мы все же разделим сферу на шесть равных частей, как у куба, и каждую сторону будем выводить в отдельности. Недаром мы создали кубическую текстуру. Куб имеет 6 граней, и с помощью цикла мы заполним каждую из них своим изображением, а затем натянем эту текстуру на сферу. Действительно, кубическую текстуру без проблем можно натянуть на объект любой формы. Именно благодаря этому у нас получается такой интересный эффект.

Итак, внутри цикла с помощью метода `GetCubeMapSurface` получаем указатель на память, где находятся данные поверхности очередных граней. Этот метод выглядит следующим образом:

```
HRESULT GetCubeMapSurface(
  D3DCUBEMAP_FACES FaceType,
  UINT Level,
  IDirect3DSurface9 **ppCubeMapSurface
);
```

Здесь имеется три параметра:

- `FaceType` — грань, которую необходимо определить. Этот параметр имеет тип перечисления `D3DCUBEMAP_FACES`, значения которого изменяются от 0 до 6, каждое из них определяет одну из 6 граней кубической текстуры;
- `Level` — уровень многоуровневой текстуры. У нас только один уровень, поэтому указываем значение 0;
- `ppCubeMapSurface` — указатель на указатель типа интерфейса `IDirect3DSurface9`, куда будет сохранен указатель на поверхность текстуры определенной ранее грани.

Итак, с помощью функции `GetCubeMapSurface` мы получаем указатель очередной грани. После этого мы должны указать, что вывод Direct3D-графики должен происходить именно на эту поверхность. Для этого используется метод `SetRenderTarget`. Первый параметр этого метода — это цель, где должен происходить рендеринг, а второй параметр — поверхность. Не все устройства поддерживают множественный рендеринг, поэтому не будем его использовать (да и не нужен он нам), а оставим равным нулю. В качестве второго параметра указываем указатель на текстуру грани, который мы только что получили. Теперь весь вывод графики будет отображаться не на сцене, а на текстуре.

Первым делом вызываем метод `Clear`, чтобы очистить буфер. На каждом этапе цикла выбираем определенный оттенок зеленого цвета, чтобы каждая грань была окрашена своим цветом, и вы могли сразу отличить границы грани на сфере. Теперь включаем освещение, выбираем материал, которым будем рисовать, и рисуем чайник.

Для того чтобы вывести только чайник, не учитывая материал и его текстуру из X-файла, используем следующий метод отображения сетки:

```
mTeapot->GetVertexBuffer(&pVB);
mTeapot->GetIndexBuffer(&pIB);
numVertices = mTeapot->GetNumVertices();
numFaces = mTeapot->GetNumFaces();

pD3DDevice->SetFVF(mTeapot->GetFVF());
pD3DDevice->SetStreamSource(0, pVB, 0, sizeof(sVertexTeapot));
pD3DDevice->SetIndices(pIB);
pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0,
                numVertices, 0, numFaces);
```

Сначала получаем буфер вершин сетки и индексный буфер с помощью методов `GetVertexBuffer` и `GetIndexBuffer` соответственно. Эти методы интерфейса `ID3DXMesh` позволяют получить необходимые для отображения объекта

чайника буферы из загруженной сетки. После этого определяем количество элементов в каждом буфере. Для этого используются методы `GetNumVertices` и `GetNumFaces`.

Необходимые для отображения объекта буферы есть, теперь нужно настроить вывод. Сначала устанавливаем формат вершины. Мы можем и не знать, какой формат используется в X-файле, но это легко определить с помощью метода `GetFVF`. Результат этого метода и передаем методу `SetFVF`.

Теперь, для вывода чайника достаточно только установить в качестве текущего полученный буфер вершин и индексный буфер и вывести объект с помощью метода `DrawIndexedPrimitive`. Напоминаю, что вывод объекта происходит не на сцену, а на текстуру.

После отображения чайника можно уничтожать индексный буфер и буфер вершин. На следующем шаге оба буфера будут заново созданы.

Когда все грани собраны, можно выводить объект сферы и натянуть на него кубическую грань. Для этого сначала нужно сделать текущий вывод не на текстуру, а на задний буфер Direct3D, чтобы сфера оказалась на экране. У нас есть указатель на задний буфер, который мы получили во время инициализации. Необходимо только перенаправить вывод на него с помощью метода `SetRenderTarget`.

Теперь, используя метод `Clear`, очищаем задний буфер, отключаем освещение и настраиваем текстуру. Чтобы наша текстура отобразилась корректно, обязательно нужно выполнить следующую строку кода, перед отображением объекта:

```
pD3DDevice->SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
   D3DTTFF_COUNT3);
```

Здесь мы устанавливаем параметр текстуры `D3DTSS_TEXTURETRANSFORMFLAGS`. В качестве значения будет флаг `D3DTTFF_COUNT3`. Это означает, что во время натягивания текстуры на объект должны использоваться трехмерные координаты текстуры.

Остальной код уже должен быть понятен — устанавливаем формат вершин и вершинный буфер сферы и отображаем объект.

Запустите пример и посмотрите на результат работы. Он поразителен. На экране вращается сфера, а на текстуре отображается чайник, который также вращается. Но это еще не все. Чайник освещается направленным светом слева, поэтому левая часть находится на свету и выглядит ярче, а правая располагается в тени, поэтому светлее.

Если вы увидели этот пример, то ваша видеокарта поддерживает аппаратную работу с текстурой. Сцена будет отображаться достаточно быстро и без за-

держек, а нагрузка на процессор не высока. Я нагрузку не замерял, но на моем ноутбуке во время выполнения этого примера даже вентилятор не увеличивал обороты и работал в экономном режиме, а ведь если процессор в портативном компьютере начинает сильно напрягаться и греться, то вентилятор работает быстрее и громче, и это сразу заметно.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\MipMapTest.

5.11. Не все золото, что блестит

Давайте рассмотрим еще один интересный пример рисования на поверхности текстуры. Мы опять же будем использовать кубическую текстуру и выводить на ее поверхность сцену, но эффект получится намного круче.

Фигура, на которую мы натянем текстуру, — кегля. Мысленно представьте себе эту фигуру. Она имеет гладкую и плавно изгибающуюся поверхность. Необходимо рисовать на текстуре как бы отражение мира. В качестве мира у нас будет куб, обтянутый текстурой, а внутри него будет находиться вращающаяся звезда. Поскольку мы создаем эффект отражения, то звезда должна отражаться от тех поверхностей кегли, на которые может упасть проекция. На рис. 5.11 показан результат работы примера. Этот пример и предстоит нам сейчас написать.

Пример будем строить не просто изменением поверхности, на которую нужно выводить сцену, а с помощью эффекта. Что это такое и как выглядит? Давайте познакомимся с примером, и вы все увидите на практике.

Итак, создадим новый проект и добавим в него инициализацию Direct3D. Теперь посмотрим, какие переменные нам понадобятся.

Для начала нам потребуются три набора переменных для хранения трех сеток, которые будут загружаться из X-файла:

```
// Переменные для сетки звезды
ID3DXMesh *pObject;
LPDIRECT3DTEXTURE9 *pMeshTextures;
D3DMATERIAL9 *pMeshMaterials;
DWORD dwNumMaterials;

// Переменные для сетки кегли
ID3DXMesh *pTube;
LPDIRECT3DTEXTURE9 *pTubeTextures;
D3DMATERIAL9 *pTubeMaterials;
DWORD dwTubeNumMaterials;
```

```
// Переменные для сетки комнаты
ID3DXMesh *pRoom;
LPDIRECT3DTEXTURE9 *pRoomTextures;
D3DMATERIAL9 *pRoomMaterials;
DWORD dwRoomMaterials;
```

Первая сетка описывает что-то похожее на звезду, которая будет двигаться по текстуре, вокруг кегли. Вторая сетка описывает кеглю. На ее текстуре необходимо рисовать. Последняя сетка — это куб, который символизирует комнату. Боковые стены внутренней стороны куба обтянуты текстурой с изображением красивой розы, а верхняя и нижняя стороны (пол и потолок комнаты) необходимо окрасить в синий и белый цвет соответственно. Эта комната также проецируется на кеглю, и на фоне ее стен будет рисоваться звезда. Это придаст сцене еще большей красоты.

Теперь нам понадобятся два интерфейса:

```
ID3DXRenderToEnvMap* pRenderToEnvMap;
IDirect3DCubeTexture9* pCubeTexture;
```

Рис. 5.11. Результат работы программы

Интерфейс `ID3DXRenderToEnvMap` (карта окружения) используется для создания изощренных сцен на текстуре, без применения комплексной геометрии. Этот интерфейс поддерживает создание поверхностей следующих разновидностей: куб, сфера, полусфера или параболическая поверхность. Нам этот интерфейс понадобится для создания кубической текстуры.

Интерфейс `IDirect3DCubeTexture9` нам уже знаком и мы его снова будем использовать для создания кубической текстуры, а поможет нам в этом интерфейс `ID3DXRenderToEnvMap`.

Теперь посмотрим на код инициализации переменных, который показан в листинге 5.25.

Листинг 5.25. Инициализация переменных

```
// Рассчитываем матрицу положения кегли
D3DXMATRIXA16 m;
D3DXMatrixIdentity(&mWorld);
D3DXMatrixRotationX(&mWorld, -2.0f);
D3DXMatrixScaling(&m, 0.6f, 0.2f, 0.3f);
D3DXMatrixMultiply(&mWorld, &mWorld, &m);

D3DXMatrixTranslation(&mView, 0.0f, -1.0f, 7.0f);

// Загружаем сетки
dwTubeNumMaterials = LoadMesh("mirror.x", pD3DDevice,
    &pTube, &pTubeTextures, "texture.bmp", &pTubeMaterials);

dwNumMaterials = LoadMesh("nlo.x", pD3DDevice,
   &pObject, &pMeshTextures, "texture.bmp", &pMeshMaterials);

dwRoomMaterials = LoadMesh("room.x", pD3DDevice,
   &pRoom, &pRoomTextures, "Room.bmp", &pRoomMaterials);

// Создаем интерфейс ID3DXRenderToEnvMap
if (FAILED(D3DXCreateRenderToEnvMap(pD3DDevice, 256, 1,
  D3DFMT_A8R8G8B8, TRUE, D3DFMT_D16, &pRenderToEnvMap)))
 return E_FAIL;

// Создаем интерфейс кубической текстуры
if (FAILED(D3DXCreateCubeTexture(pD3DDevice, 256, 1,
   D3DUSAGE_RENDERTARGET, D3DFMT_A8R8G8B8, D3DPOOL_DEFAULT,
   &pCubeTexture)))
 return E_FAIL;
```

```
// Определяем две матрицы проекции
pD3DDevice->GetTransform(D3DTS_PROJECTION, &mProject);
D3DXMatrixPerspectiveFovLH(&matProj, D3DX_PI * 0.4f, 1.0f, 0.5f,
1000.0f);
```

В самом начале рассчитываем матрицу положения кегли. Сетка кегли будет неподвижной, поэтому можно заранее рассчитать положение и перед прорисовкой только устанавливать эту матрицу. Когда мы начнем прорисовывать движущуюся звезду, текущая матрица станет изменяться. После этого, достаточно только восстановить рассчитанную во время инициализации матрицу.

Далее, загружаем три сетки уже знакомой нам функцией `LoadMesh`. Тут ничего нового нет, поэтому движемся дальше. А дальше мы встречаемся с функцией `D3DXCreateRenderToEnvMap`, которая создает интерфейс `ID3DXRenderToEnvMap`. Посмотрим, как выглядит эта функция и что она конкретно делает.

```
HRESULT D3DXCreateRenderToEnvMap(
  LPDIRECT3DDEVICE9 pDevice,
  UINT Size,
  D3DFORMAT Format,
  BOOL DepthStencil,
  D3DFORMAT DepthStencilFormat,
  LPD3DXRenderToEnvMap* ppRenderToEnvMap
);
```

Здесь имеется 6 параметров. Давайте кратко рассмотрим их:

- `pDevice` — указатель на Direct3D-устройство;
- `Size` — размер карты. У нас кубическая текстура будет размером в 256 единиц, и карта тоже должна быть такого же размера;
- `Format` — формат точки;
- `DepthStencil` — если параметр равен `TRUE`, то поверхность будет поддерживать буфер глубины. Это нам потребуется, чтобы проекция комнаты и звезды отражалась корректно с учетом положения в 3D-пространстве;
- `DepthStencilFormat` — формат буфера глубины;
- `ppRenderToEnvMap` — указатель на переменную типа `ID3DXRenderToEnvMap`, в которую будет записан созданный интерфейс.

После создания интерфейса карты окружения создаем непосредственно кубическую текстуру.

Напоследок определяем две матрицы проекции. Для начала запоминаем матрицу, которая была создана по умолчанию во время инициализации Direct3D

в функции DX3DInitZ. Для этого используем метод GetTransform, которому в качестве первого параметра указываем флаг D3DTS_PROJECTION (указывает на необходимость получить матрицу проекции), а второй параметр должен содержать переменную, где необходимо сохранить текущую матрицу.

После этого, с помощью метода D3DXMatrixPerspectiveFovLH, создаем еще одну матрицу, которая будет квадратной (соотношение сторон ее равно единице). Эта матрица используется при отображении графики на текстуре.

Функция рисования приведена в листинге 5.26.

Листинг 5.26. Отображение кегли с анимированной текстурой

```
void DrawScene()
{
 ViewAngle+=0.05f;
 // Рассчитываем матрицу положения звезды
 D3DXMATRIXA16  mat;
 D3DXMatrixScaling(&mFlyer, 0.1f, 0.1f, 0.1f);
 D3DXMatrixTranslation(&mat, 0.0f, 0.0f, 3.0f);
 D3DXMatrixMultiply(&mFlyer, &mFlyer, &mat);
 D3DXMatrixRotationY(&mat, ViewAngle);
 D3DXMatrixMultiply(&mFlyer, &mFlyer, &mat);

 HRESULT hr;

 // Генерируем карту
 D3DXMATRIXA16 mViewDir(mView);
 mViewDir._41 = 0.0f; mViewDir._42 = 0.0f; mViewDir._43 = 0.0f;
 // Цикл отображения сторон куба карты
 hr = pRenderToEnvMap->BeginCube( pCubeTexture );
 for (int i=0; i<6; i++)
 {
  pRenderToEnvMap->Face((D3DCUBEMAP_FACES) i, 0);

  D3DXVECTOR3 vLookDir;
  D3DXVECTOR3 vUpDir;

  switch(i)
  {
   case D3DCUBEMAP_FACE_POSITIVE_X:
    vLookDir = D3DXVECTOR3( 1.0f, 0.0f, 0.0f );
    vUpDir   = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
    break;
```

```
 case D3DCUBEMAP_FACE_NEGATIVE_X:
  vLookDir = D3DXVECTOR3(-1.0f, 0.0f, 0.0f );
  vUpDir   = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
  break;
 case D3DCUBEMAP_FACE_POSITIVE_Y:
  vLookDir = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
  vUpDir   = D3DXVECTOR3( 0.0f, 0.0f,-1.0f );
  break;
 case D3DCUBEMAP_FACE_NEGATIVE_Y:
  vLookDir = D3DXVECTOR3( 0.0f,-1.0f, 0.0f );
  vUpDir   = D3DXVECTOR3( 0.0f, 0.0f, 1.0f );
  break;
 case D3DCUBEMAP_FACE_POSITIVE_Z:
  vLookDir = D3DXVECTOR3( 0.0f, 0.0f, 1.0f );
  vUpDir   = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
  break;
 case D3DCUBEMAP_FACE_NEGATIVE_Z:
  vLookDir = D3DXVECTOR3( 0.0f, 0.0f,-1.0f );
  vUpDir   = D3DXVECTOR3( 0.0f, 1.0f, 0.0f );
  break;
}

// Расчет положения комнаты
D3DXMATRIXA16 mView;
D3DXVECTOR3 vEyePt   = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
D3DXMatrixLookAtLH(&mView, &vEyePt, &vLookDir, &vUpDir);
D3DXMatrixMultiply(&mView, &mViewDir, &mView);

D3DXMATRIXA16 matWorld;
D3DXMatrixScaling(&matWorld, 10.0f, 10.0f, 10.0f);

pD3DDevice->SetTransform(D3DTS_WORLD, &matWorld);
pD3DDevice->SetTransform(D3DTS_VIEW, &mView);
pD3DDevice->SetTransform(D3DTS_PROJECTION, &matProj);

// Отображение комнаты на текстуре
pD3DDevice->SetRenderState( D3DRS_ZFUNC, D3DCMP_ALWAYS );
for (DWORD i=0; i<dwRoomMaterials; i++)
{
 if (pRoomTextures[i])
  pD3DDevice->SetTexture(0, pRoomTextures[i]);
 pRoom->DrawSubset(i);
}
```

```
  // Расчет положения звезды
  pD3DDevice->SetRenderState(D3DRS_ZFUNC, D3DCMP_LESSEQUAL);
  pD3DDevice->SetTransform(D3DTS_WORLD, &mFlyer);

  for (DWORD i=0; i<dwNumMaterials; i++)
  {
   if (pMeshTextures[i])
    pD3DDevice->SetTexture(0, pMeshTextures[i]);
   pObject->DrawSubset(i);
  }
 }
 pRenderToEnvMap->End(0);

 // Установка матрицы положения кегли
 pD3DDevice->SetTransform(D3DTS_PROJECTION, &mProject);
 pD3DDevice->SetTransform(D3DTS_VIEW, &mView);
 pD3DDevice->SetTransform(D3DTS_WORLD, &mWorld);

 // Установка текстуры и ее параметров
 pD3DDevice->SetTexture(0, pCubeTexture);
 pD3DDevice->SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
        D3DTTFF_COUNT3);
 pD3DDevice->SetTextureStageState(0, D3DTSS_TEXCOORDINDEX,
        D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR);

 // Отображения кегли
 for (DWORD i=0; i<dwTubeNumMaterials; i++)
  pTube->DrawSubset(i);
}
```

Нескромная получилась функция? Я тоже так думаю. Половина кода вам уже знакома и мы подобные приемы использовали, так что с их пониманием проблем не возникнет. Самое интересное будет в создании и заполнении карты окружения, поэтому все знакомые вещи будем просто проскакивать.

В самом начале рассчитываем матрицу положения звезды. Она будет вращаться вокруг оси Y и как бы по поверхности кегли.

После этого вызывается метод `BeginCube` интерфейса карты окружения. Этот метод запускает создание кубической карты окружения. В качестве параметра передается указатель на кубическую текстуру, на которой и будет происходить рисование. Интерфейс `ID3DXRenderToEnvMap` всего лишь помогает нам нарисовать кубическую текстуру.

Далее, запускается цикл от 0 до 5, в котором мы будем рисовать каждую поверхность куба. Первым делом вызываем метод `Face` интерфейса

ID3DXRenderToEnvMap. В качестве параметра необходимо указать, какую именно поверхность куба нужно рисовать. Поверхности нумеруются от 0 и до 5, как раз именно столько выполняется цикл. Если уже было начато рисование какой-то поверхности, то метод Face завершает этот процесс и начинает сцену для указанной поверхности.

Далее идет большой метод switch, который, в зависимости от текущей поверхности, устанавливает матрицу взгляда на сцену. Здесь заполняется два вектора: vLookDir и vUpDir, которые определяют направление просмотра. Зная направление просмотра поверхности, мы можем корректно ее отобразить.

Определив матрицу, устанавливаем ее и отображаем куб, символизирующий комнату и звезду.

Когда цикл отображения поверхностей окружения закончит свою работу, вызываем метод End интерфейса ID3DXRenderToEnvMap, чтобы завершить формирование текстуры.

Теперь у нас есть кубическая текстура и можно устанавливать ее в качестве текущей и выводить саму кеглю. Текстура устанавливается следующими тремя строчками кода:

```
pD3DDevice->SetTexture(0, pCubeTexture);
pD3DDevice->SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
        D3DTTFF_COUNT3);
pD3DDevice->SetTextureStageState(0, D3DTSS_TEXCOORDINDEX,
        D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR);
```

Первые две строки нам уже знакомы. Первая из них выбирает кубическую текстуру в качестве текущей, а вторая указывает, что необходимо использовать трехмерные координаты. Последняя строка настраивает параметр D3DTSS_TEXCOORDINDEX. Что это такое? Этот параметр определяет индекс координат. В данном случае мы задаем параметр D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR, который говорит, что необходимо использовать координаты вектора отражения, трансформированного в координаты камеры. Не понятно? На практике это будет выглядеть, как будто сцена текстуры отражается от поверхности кегли.

Настроив текстуру, можно выводить изображение кегли. Вы имеете возможность запустить пример и наслаждаться его красотой. На мой взгляд, эффект получился хорошим.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\MipMapTest2.

5.12. Эффекты

Давайте поговорим об эффектах и интерфейсе `ID3DXEffect`. Этот интерфейс позволяет серьезно упростить создание сцены и сделать код более гибким. В данном случае не будем создавать что-то новое, а поиграем с примером из *разд. 5.11*, сделав вывод кегли с помощью эффекта.

Для начала посмотрим, что такое эффект. На самом деле это строка, которая описывает параметры вывода сцены. Здесь может быть указание параметров текстуры, имени, положение и т. д. Посмотрим на вариант кода, который мы будем использовать в своем примере:

```
textureCUBE CubeMapTex;
technique Cube
{
 pass P0
 {
  Texture[0] = <CubeMapTex>;
  TexCoordIndex[0] = CameraSpaceReflectionVector;
  TextureTransformFlags[0] = Count3;
 }
}
```

Этот код написан на языке HLSL (High Level Shader Language, высокоуровневый язык шейдеров). Давайте разберемся, что означает этот текст. В первой строке происходит что-то похожее на объявление переменной `CubeMapTex`, которая будет иметь тип кубической текстуры. После этого начинается новый блок `technique` с именем `Cube`. В такой блок заключаются параметры отображения. Откуда название `technique`? До DirectX 8 интерфейс эффектов назывался `ID3DXTechnique`, и он определяет технику отображения.

Внутри этого блока может быть объявлено несколько блоков `pass`. Каждый блок содержит свои параметры. В данном случае, есть один блок с именем `P0`, в котором задается текстура, задается индекс координат и флаги отображения. Посмотрите на эти три строки и сравните со строками настройки текстуры, которые мы использовали при выводе кегли в *разд. 5.11*:

```
pD3DDevice->SetTexture(0, pCubeTexture);
pD3DDevice->SetTextureStageState(0, D3DTSS_TEXCOORDINDEX,
         D3DTSS_TCI_CAMERASPACEREFLECTIONVECTOR);
pD3DDevice->SetTextureStageState(0, D3DTSS_TEXTURETRANSFORMFLAGS,
         D3DTTFF_COUNT3);
```

Как видите, они схожи, потому что задают одни и те же параметры. Эффект как бы заменяет необходимость настраивать текстуру. Достаточно только

указать, какой эффект использовать при отображении объекта, и все настройки произойдут автоматически в соответствии со строкой инициализации эффекта.

Очень интересно задается текстура. Если флаги имеют заранее определенные в Direct3D имена, то имя интерфейса текстуры мы явно прописать не можем. Вместо этого, текстуре назначается значение переменной `CubeMapTex`, которую мы ввели в первой строке. Когда мы будем создавать эффект, то достаточно будет этой переменной указать имя созданной кубической текстуры.

Более подробно о вершинных шейдерах мы поговорим в *разд. 5.16*.

Давайте рассмотрим использование интерфейса `ID3DXEffect` на практике. Для начала нам понадобится строковая переменная, которая будет хранить текст эффекта. В заголовочном файле пропишем следующую переменную:

```
const char cEffect[] =
    "textureCUBE CubeMapTex;\n"

       "technique Cube\n"
    "{\n"
        "pass P0\n"
        "{\n"
            "Texture[0] = <CubeMapTex>;\n"

            "TexCoordIndex[0] = CameraSpaceReflectionVector;\n"
            "TextureTransformFlags[0] = Count3;\n"
        "}\n"
    "}\n";
```

Теперь нам понадобится переменная типа `ID3DXEffect`:

```
ID3DXEffect* pEffect;
```

В код инициализации переменных нужно добавить (можно до, а можно и после создания текстуры) еще и код создания эффекта:

```
D3DXCreateEffect(pD3DDevice, cEffect, sizeof(cEffect)-1, NULL,
          NULL, 0, NULL, &pEffect, NULL);
```

Функция `D3DXCreateEffect` в общем виде выглядит следующим образом:

```
HRESULT D3DXCreateEffect(
  LPDIRECT3DDEVICE9 pDevice,
  LPCVOID pSrcData,
  UINT SrcDataLen,
  CONST D3DXMACRO* pDefines,
  LPD3DXINCLUDE pInclude,
```

```
  DWORD Flags,
  LPD3DXEFFECTPOOL pPool,
  LPD3DXBUFFER* ppEffect,
  LPD3DXBUFFER *ppCompilationErrors
);
```

Параметров много, но они простые. Основное, что нас интересует, — это:

- `pDevice` — указатель на Direct3D-устройство;
- `pSrcData` — строка, которая содержит описание эффекта;
- `SrcDataLen` — длина строки;
- `ppEffect` — указатель на указатель интерфейса эффекта, куда будет записан результат.

После создания интерфейсов эффекта и текстуры необходимо указать нашу кубическую текстуру эффекту. Это делается следующим образом:

```
pEffect->SetTexture("CubeMapTex", pCubeTexture);
```

Метод `SetTexture` устанавливает текстуру и принимает два параметра — имя переменной, которую мы ввели в строке, описывающей эффект, и указатель на устанавливаемую текстуру.

Теперь отображение объекта кегли будет выглядеть следующим образом:

```
UINT uPass;
pEffect->Begin(&uPass, 0);
for (int i=0; i<uPass; i++)
{
 pEffect->Pass(i);
 for (DWORD i=0; i<dwTubeNumMaterials; i++)
   pTube->DrawSubset(i);
}
pEffect->End();
```

Первым делом вызывается метод `Begin` интерфейса `ID3DXEffect`. Этому методу передается два параметра — указатель на переменную типа `UINT` и флаги отображения. В переменной будет сохранено количество проходов, которые есть в эффекте, а с помощью флагов можно определить, нужно ли сохранять текущее состояние перед рисованием. Если в качестве флага указано 0, то все изменения, сделанные эффектом, сохраняться не будут. В нашем случае все настройки текстуры после рисования кегли сбросятся.

Теперь запускаем цикл от 0 до количества проходов, которые мы получили во время вызова метода `Begin`. На каждом этапе просто вызываем метод `Pass`, указывая в качестве параметра номер прохода (текущее значение цикла) и отображаем объект. Во время вызова метода `Pass` интерфейс `ID3DXEffect` ана-

лизирует содержимое строки и устанавливает параметры, которые были установлены в строке эффекта.

В нашем случае в строке эффекта только один, нулевой проход — `pass P0`, поэтому можно даже не использовать цикл, а просто применить следующий код для вывода кегли:

```
UINT uPass;
pEffect->Begin(&uPass, 0);
pEffect->Pass(0);
for (DWORD i=0; i<dwTubeNumMaterials; i++)
 pTube->DrawSubset(i);
pEffect->End();
```

Как видите, с помощью эффектов намного проще настраивать вывод. В данном случае мы избавляемся от необходимости настройки текстуры.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Effect.

5.13. Обман зрения

В 3D-мире также очень часто используется обман зрения. Давайте попробуем воспользоваться обманом для повышения производительности. Недавно я нашел у Microsoft очень интересный пример, который показывает, как создать ландшафт и поместить на него деревья.

Деревьев будет много. Хотя не все они одновременно отображаются на экране, для процессора будет накладно рассчитывать каждую веточку и листик. Даже если каждое дерево карликовое и состоит из 10 веток, по 10 листиков на каждой, то центральный процессор и видеокарта будут в шоке. Как же тогда сэкономить процессорные такты и отобразить необходимую сцену? Конечно же, обмануть зрителя. Достаточно только создать прямоугольный объект и натянуть на него текстуру дерева. В этом случае зритель будет видеть дерево, но единственный недостаток — оно будет плоским, а не трехмерным. С какой стороны вы не посмотрели бы на объект, он будет иметь одну и ту же форму, потому что это всего лишь текстура, а не динамически рассчитываемый объект.

В демо-роликах, когда сцена изменяется быстро, мы легко можем обманывать пользователя, потому что из-за большой скорости движения зритель может не заметить подвох. В играх за такой трюк уже могут дать по шапке. Дело в том, что игрок может остановиться и в спокойной обстановке осмотреть

округу, а если есть возможность обойти дерево вокруг, то обман сразу же откроется.

Я немного модифицировал и упростил этот пример, и именно модифицированную версию мы сейчас и рассмотрим. Очень познавательный пример. Для понимания этого достаточно только взглянуть на рис. 5.12, где показан результат работы примера.

Рис. 5.12. Ландшафт с множеством деревьев

Для начала посмотрим на инициализацию (листинг 5.27).

Листинг 5.27. Инициализация поляны и деревьев

```
for (WORD i=0; i<NUM_TREES; i++)
{
 do
 {
  // Генерируем позицию дерева
  float fTheta=2.0f*D3DX_PI*(FLOAT)rand()/RAND_MAX;
```

```
  float fRadius=25.0f + 55.0f * (FLOAT)rand()/RAND_MAX;
  m_Trees[i].vPos.x=fRadius * sinf(fTheta);
  m_Trees[i].vPos.z=fRadius * cosf(fTheta);
  m_Trees[i].vPos.y=HeightField(m_Trees[i].vPos.x, m_Trees[i].vPos.z );
 } while( !IsTreePositionValid( i ) );

 // Случайным образом задаем размер дерева
 float fWidth  = 1.0f + 0.2f * (FLOAT)(rand()-rand())/RAND_MAX;
 float fHeight = 1.4f + 0.4f * (FLOAT)(rand()-rand())/RAND_MAX;

 // Генерируем цвет прямоугольника
 DWORD r = (255-190) + (DWORD)(190*(FLOAT)(rand())/RAND_MAX);
 DWORD g = (255-190) + (DWORD)(190*(FLOAT)(rand())/RAND_MAX);
 DWORD b = 0;
 DWORD dwColor = 0xff000000 + (r<<16) + (g<<8) + (b<<0);

 // Определяем вершины прямоугольника на основе сгенерированных размеров
 m_Trees[i].v[0].p     = D3DXVECTOR3(-fWidth, 0*fHeight, 0.0f );
 m_Trees[i].v[0].color = dwColor;
 m_Trees[i].v[0].tu    = 0.0f;   m_Trees[i].v[0].tv = 1.0f;
 m_Trees[i].v[1].p     = D3DXVECTOR3(-fWidth, 2*fHeight, 0.0f  );
 m_Trees[i].v[1].color = dwColor;
 m_Trees[i].v[1].tu    = 0.0f;   m_Trees[i].v[1].tv = 0.0f;
 m_Trees[i].v[2].p     = D3DXVECTOR3( fWidth, 0*fHeight, 0.0f  );
 m_Trees[i].v[2].color = dwColor;
 m_Trees[i].v[2].tu    = 1.0f;   m_Trees[i].v[2].tv = 1.0f;
 m_Trees[i].v[3].p     = D3DXVECTOR3( fWidth, 2*fHeight, 0.0f  );
 m_Trees[i].v[3].color = dwColor;
 m_Trees[i].v[3].tu    = 1.0f;   m_Trees[i].v[3].tv = 0.0f;

 // Случайным образом задаем текстуру
 m_Trees[i].dwTreeTexture = (DWORD)((NUMTREETEXTURES * rand()) /
          (FLOAT)RAND_MAX );
}

// Создать текстуру буфера
for( DWORD i=0; i<NUMTREETEXTURES; i++ )
  D3DXCreateTextureFromFileEx(pD3DDevice, g_strTreeTextures[i],
        D3DX_DEFAULT, D3DX_DEFAULT, D3DX_DEFAULT, 0, D3DFMT_UNKNOWN,
        D3DPOOL_MANAGED, D3DX_FILTER_TRIANGLE|D3DX_FILTER_MIRROR,
        D3DX_FILTER_TRIANGLE|D3DX_FILTER_MIRROR, 0, NULL, NULL,
        &TreeTextures[i]);
```

```
// Генерируем прямоугольник, на который будет натянута текстура дерева
if( FAILED( hr = pD3DDevice->CreateVertexBuffer(
     NUM_TREES*4*sizeof(TREEVERTEX), D3DUSAGE_WRITEONLY,
     TREEVERTEX::FVF, D3DPOOL_MANAGED, &VBTree, NULL)))
 return hr;

// Копируем массив деревьев в буфер вершин
TREEVERTEX* v;
VBTree->Lock(0, 0, (void**)&v, 0);
DWORD dwOffset = 0;
for (int iTree = 0; iTree<NUM_TREES; iTree++)
{
 memcpy( &v[dwOffset], m_Trees[iTree].v, 4*sizeof(TREEVERTEX) );
 m_Trees[iTree].dwOffset = dwOffset;
 dwOffset += 4;
}
VBTree->Unlock();

// Загрузка сетки фона и ландшафта
skynum=LoadMesh("SkyBox2.x", pD3DDevice, &m_pSkyBox, &pMeshTextures1,
   "sky.bmp", &pMeshMaterials1);

seanum=LoadMesh("SeaFloor.x", pD3DDevice, &m_pTerrain, &pMeshTextures2,
   "texture.bmp", &pMeshMaterials2);
```

В листинге показан только самый интересный код, который уже немного приоткрывает занавес над секретом создания сцены.

В самом начале запускается цикл, который перебирает все элементы массива `m_Trees` и инициализирует их элементы. Массив `m_Trees` — это структура следующего вида:

```
struct Tree
{
 TREEVERTEX v[4];
 D3DXVECTOR3 vPos;
 DWORD dwTreeTexture;
 DWORD dwOffset;
};
```

Первый параметр `v` — массив из четырех вершин. С помощью этих вершин мы описываем прямоугольный объект, на который натягивается текстура дерева. Обратите внимание, что количество вершин только 4 и с их помощью можно описать только плоский прямоугольник, поэтому деревья просто не

смогут быть объемными. Область прямоугольника рассчитывается на основе случайно сгенерированных значений ширины и высоты.

Следующий параметр `vPos` — это позиция дерева. Координаты X и Z определяются случайным образом, а координата Y определяется высотой ландшафта в точке с координатами X и Z. Это значит, что координата Y ландшафта и дерева в точке X, Z одинаковы.

Следующий параметр `dwTreeTexture` — определяет номер текстуры, которая натягивается на прямоугольник. Всего у нас имеется три текстуры, и благодаря их случайному чередованию и окрашиванию в разный цвет, будет создаваться впечатление леса из множества разных деревьев.

Последний параметр `dwOffset` — это смещение в байтах от начала массива, где расположены вершины прямоугольника, описывающего дерево. Дело в том, что мы создадим 1 буфер вершин, в котором будут храниться вершины всех прямоугольников. Чтобы быстро узнать, где именно в буфере вершин находится информация о прямоугольнике и используется параметр `dwOffset`. Этот параметр мы пока заполнить не можем, поэтому оставим его на потом, когда будет готов буфер вершин.

Следующим этапом создаются поверхности для хранения текстур и буферы вершин. Для создания текстуры используется уже знакомая нам функция `D3DXCreateTextureFromFileEx`.

После этого создаем буфер вершин, блокируем его и заполняем значениями. Одновременно заполняем значением поле `dwOffset` в массиве структур `m_Trees`.

Самое последнее, что делает наш код инициализации — загружает объекты сетки неба и ландшафта. Да, это будут сетки. Объект неба — это всего лишь куб, который находится в файле SkyBox2.x. Внутренняя поверхность куба будет обтянута текстурой из файла sky.bmp. Наш ландшафт, а также камера будут находиться внутри куба, поэтому мы увидем его внутреннюю поверхность.

Таким же образом может создаваться и задний фон в компьютерных играх. Никто не будет создавать небо из вершин, потому что это излишние расходы процессорного времени. Намного проще натянуть текстуру. Да, она будет статичной, но зритель может закрыть на это глаза, потому что все действие происходит на переднем плане и все внимание приковывается именно на действие.

Следующим этапом загружается содержимое файла SeaFloor.x, где находится сетка моря, имитирующая море, с небольшой волнистостью. После загрузки сетки волнистость программно увеличивается, чтобы получилась холмистость, но этот код мы опустили.

Текстура, которая будет отображаться на ландшафте, имеет размер всего 256×256 пикселов и при натягивании на сетку произойдет серьезное масштабирование. В результате, вы увидите просто огромные зерна, а я бы сказал даже квадраты из-за слишком большого растягивания растра. Чтобы сгладить картинку, необходимо применить параметр `D3DSAMP_MAGFILTER`. Давайте присвоим ему на этапе инициализации значение `D3DTEXF_LINEAR`:

```
pD3DDevice->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
```

Текстура ландшафта — это несколько кирпичей, но после растягивания и сглаживания получится достаточно приятная поверхность.

Теперь остановимся на текстурах, которые в нашем примере загружаются из DDS-файлов. Что это за файлы? Графические форматы, которые мы использовали ранее (например, BMP), не сжимают графику, и файл текстуры получается слишком большим. Это нелепое расходование памяти. Но главная проблема растровых изображений в файлах BMP — они не содержат информации об альфа-канале.

Файлы с расширением dds решают обе проблемы и делают это великолепно. Во-первых, растровое изображение в DDS-файле сжимается. При этом сама видеокарта будет работать с уже сжатыми данными. Это позволит экономить память не только на диске, но и в видеопамяти. С другой стороны, если видеокарта не поддерживает формат сжатия, который использован в DDS-файле, то могут возникнуть серьезные проблемы.

Теперь поговорим о прозрачности. Посмотрим на рис. 5.13, где показано изображение дерева. Белый цвет фона автоматически будет воспринят как прозрачный, потому что в файле есть альфа-канал, который указывает на прозрачность.

Для создания DDS-текстуры можно воспользоваться утилитой DxTex.exe, которую вы можете найти в каталоге \DXSDK\Bin\DXUtils. Эта утилита может запускаться как в визуальном режиме, так и из командной строки. Команда выглядит следующим образом:

```
dxtex [infilename] [-a alphaname] [-m]
   [DXT1|DXT2|DXT3|DXT4|DXT5] [outfilename]
```

- `infilename` — картинка в формате BMP, которую необходимо преобразовать;
- `-a alphaname` — ключ `-a` задает BMP-файл с картинкой, которая будет использоваться в качестве альфа-канала;
- `-m` — генерировать mipmap-текстуры;
- `DXT1|DXT2|DXT3|DXT4|DXT5` — формат сжатия изображения. Данный формат должен поддерживаться видеокартой;
- `outfilename` — имя выходного файла.

Рис. 5.13. Текстура дерева

Для создания DDS-файла можно воспользоваться и программой Photoshop. Но для этого необходимо установить специальный фильтр. Но если представить, сколько стоит Photoshop, то лучше воспользоваться бесплатной утилитой DxTex, хотя она и простая.

Функцию `DrawScene`, которая у нас всегда отображает сцену, мы опустим. Ее вы сможете найти на прилагаемом компакт-диске. В ней мы сначала изменяем матрицу просмотра, двигая камеру по кругу, чтобы была возможность осматривать ландшафт. Затем отображается сетка куба, символизирующего небо, и сетка ландшафта. Все это мы уже видели ни один раз, поэтому в этом коде ничего нового вы не найдете, и, следовательно, не будем тратить наше драгоценное время и место в книге.

Теперь переходим к рисованию деревьев. Они отображаются в отдельной функции `DrawTrees`, которая приведена в листинге 5.28. Вот на ней мы и остановим свое внимание.

Листинг 5.28. Отображение деревьев

```
void DrawTrees()
{
 pD3DDevice->SetRenderState( D3DRS_ALPHABLENDENABLE,   TRUE );
 pD3DDevice->SetRenderState( D3DRS_SRCBLEND,  D3DBLEND_SRCALPHA );
 pD3DDevice->SetRenderState( D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA );
```

```
 // Включить прозрачность
 pD3DDevice->SetRenderState( D3DRS_ALPHATESTENABLE, TRUE );
 pD3DDevice->SetRenderState( D3DRS_ALPHAREF,        0x08 );
 pD3DDevice->SetRenderState( D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL );

 // Устанавливаем вершинный буфер
 pD3DDevice->SetStreamSource( 0, VBTree, 0, sizeof(TREEVERTEX) );
 pD3DDevice->SetFVF( TREEVERTEX::FVF );

 // Цикл отображения деревьев
 for (DWORD i=0; i<NUM_TREES; i++)
 {
  // Проверяем, является ли текущее дерево невидимым
  if (0 > ( m_Trees[i].vPos.x - mEyePt.x ) * g_vDir.x +
          ( m_Trees[i].vPos.z - mEyePt.z ) * g_vDir.z)
      break;

  // Устанавливаем текстуру
  pD3DDevice->SetTexture( 0, TreeTextures[m_Trees[i].dwTreeTexture] );

  // Перемещаем матрицу в положение дерева
  m_matBillboardMatrix._41 = m_Trees[i].vPos.x;
  m_matBillboardMatrix._42 = m_Trees[i].vPos.y;
  m_matBillboardMatrix._43 = m_Trees[i].vPos.z;
  pD3DDevice->SetTransform( D3DTS_WORLD, &m_matBillboardMatrix );

  // Отобразить дерево
  pD3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP,m_Trees[i].dwOffset,2);
 }

  // Восстанавливаем состояние матрицы и параметры отображения
  D3DXMATRIXA16  matWorld;
  D3DXMatrixIdentity( &matWorld );
  pD3DDevice->SetTransform(D3DTS_WORLD, &matWorld);
  pD3DDevice->SetRenderState(D3DRS_ALPHATESTENABLE, FALSE);
  pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
}
```

Я надеюсь, что все понятно и по комментариям. Единственное пояснение, которое хотелось бы дать, так это о методе повышения производительности, который использовался в этом методе. Дело в том, что у нас аж 500 случайно разбросанных по ландшафту деревьев. Проверять все на видимость будет достаточно сложной и утомительной задачей. Необходим способ, позволяющий сэкономить время, и этот способ есть. Перед отображением деревьев

можно отсортировать их по текущему расположению. Для этого воспользуемся функцией `qsort`. После этого, во время отображения достаточно выводить деревья до тех пор, пока мы не встретим невидимое дерево. Остальные деревья в массиве уж точно будут невидимы.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\LandScape1.

5.14. Зеркало

Давайте посмотрим, как можно создать эффект зеркала. Этот эффект, так же, как и рассмотренный в *разд. 5.13*, относится к обману зрения. Давайте попробуем нарисовать объект тубы и зеркало, в котором будет отражаться этот объект. На рис. 5.14 вы можете увидеть результат работы примера, который мы сейчас будем рассматривать.

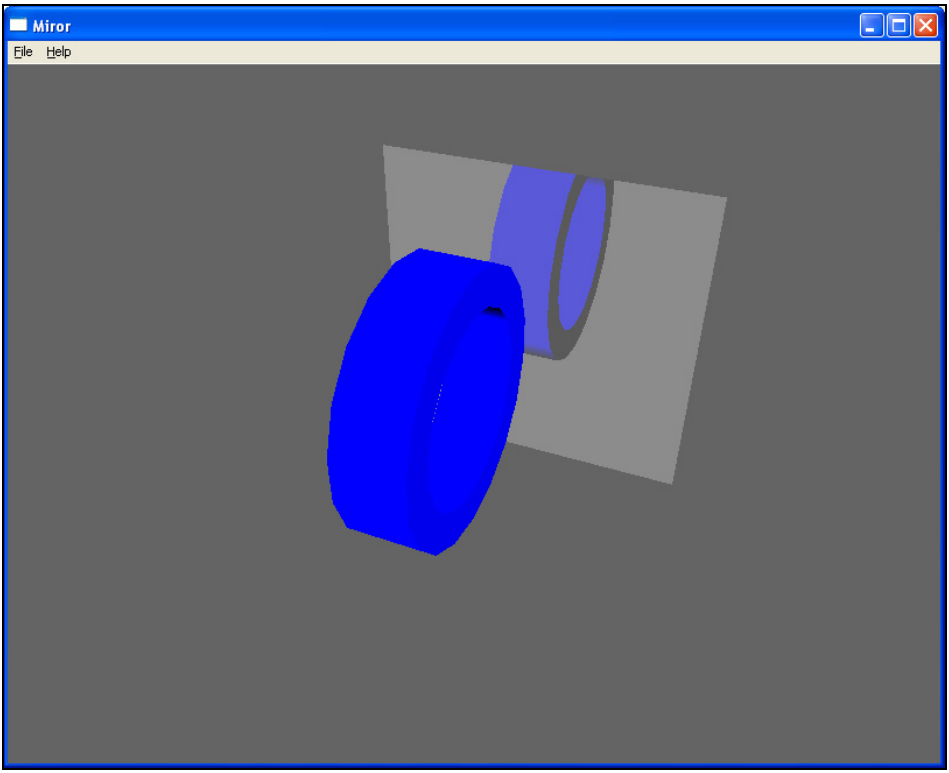

Рис. 5.14. Пример эффекта зеркала

Меньше слов, а сразу переходим к делу. Давайте посмотрим, что нам понадобится для реализации эффекта. Для начала нам нужно само зеркало. Не будем усложнять его оборочками, а возьмем простой прямоугольник. Для описания его вершин нам необходимы координаты и цвет. Таким образом, нам понадобится следующая структура:

```
struct sVERTEX
{
 D3DXVECTOR3 p;
 DWORD color;
};
```

Раньше координаты вершины мы описывали с помощью трех параметров типа `float`, но в данном примере в целях оптимизации вершина описывается структурой `D3DXVECTOR3`. Результат для Direct3D одинаков, а для нашего кода так удобнее.

Теперь введем четыре переменные типа `D3DXVECTOR3`:

```
D3DXVECTOR3 pt1(-3.0f, 3.0f, 3.0f);
D3DXVECTOR3 pt2( 3.0f, 3.0f, 3.0f);
D3DXVECTOR3 pt3(-3.0f,-3.0f, 3.0f);
D3DXVECTOR3 pt4( 3.0f,-3.0f, 3.0f);
```

Эти переменные описывают вершины прямоугольника зеркала. Но они будут использоваться не только для этого, а зачем еще, вы узнаете немного позже.

Вершины нужно где-то хранить, поэтому введем переменную для хранения буфера вершин типа `IDirect3DVertexBuffer9`:

```
IDirect3DVertexBuffer9 *pMirror;
```

Теперь об объекте, который будет отображаться. Не будем мудрить, а загрузим его из X-файла. Для этого нам понадобятся следующие переменные:

```
ID3DXMesh *pObject;
LPDIRECT3DTEXTURE9 *pMeshTextures;
D3DMATERIAL9 *pMeshMaterials;
DWORD dwNumMaterials;
```

Теперь посмотрим на инициализацию всех этих переменных, которая показана в листинге 5.29. Я оставил в листинге только самое интересное, а полный код можно увидеть в исходнике на прилагаемом компакт-диске.

Листинг 5.29. Инициализация программы зеркала

```
// Загружаем сетку объекта, который будем отображать в зеркале
dwNumMaterials = LoadMesh("mirror.x", pD3DDevice, &pObject,
    &pMeshTextures, "texture.bmp", &pMeshMaterials);
```

```
// Создаем буфер вершин
if (FAILED( pD3DDevice->CreateVertexBuffer(4*sizeof(sVERTEX),
         D3DUSAGE_WRITEONLY, D3DFVF_XYZ | D3DFVF_DIFFUSE,
         D3DPOOL_MANAGED, &pMirror, NULL ) ) )
   return E_FAIL;

// Заполняем буфер вершин
sVERTEX* v;
pMirror->Lock( 0, 0, (void**)&v, 0 );
v[0].p = pt1;
v[1].p = pt2;
v[2].p = pt3;
v[3].p = pt4;
v[0].color=v[1].color=v[2].color=v[3].color=0x80ffffff;
pMirror->Unlock();

D3DLIGHT9 light;
// Здесь опущен код инициализации лампочки
pD3DDevice->SetLight(0, &light);
pD3DDevice->LightEnable(0, TRUE);
```

Во время инициализации создается еще два материала синего и серого цвета. Синим цветом будем окрашивать объект, который должен отображаться в зеркале, а серым само зеркало.

Благодаря тому, что описание позиции вершины происходит в виде переменной типа `D3DXVECTOR3`, то заполнение буфера очень сильно упрощается. Цвет всех вершин будет белым (в принципе, не имеет значение, потому что используется материал), но при этом полупрозрачным. Альфа-канал равен 80. Зачем это делать? Я думаю, пора открыть занавес над обманом. Дело в том, что объект тубы мы станем рисовать дважды. Один раз перед зеркалом, а второй раз за ним. Смотря в зеркало, мы не будем видеть реального отражения, а увидим находящуюся сзади зеркала копию тубы.

Но не все так просто. Если бы зеркало занимало весь экран, то простой копии было бы достаточно, но наше зеркало небольшое, а объект еще должен вращаться. Во время вращения объект может выйти за пределы зеркала, а необходимо сделать так, чтобы выступающие части не были видны, а обрезались. Вот над этим мы и поколдуем во время формирования сцены.

Теперь посмотрим, как происходит отображение сцены, которое показано в листинге 5.30.

Листинг 5.30. Отображение сцены

```
void DrawScene()
{
 // Переменные, которые понадобятся нам в примере
 D3DXMATRIXA16 mView;
 D3DXMATRIXA16 mWorld;
 D3DXMATRIXA16 mReflectInMirror;
 D3DXPLANE  plane;

 pD3DDevice->SetRenderState(D3DRS_LIGHTING, TRUE);

 // Рассчитываем индексы положения
 ViewAngle1+=ViewAngle1Offset;
 if (ViewAngle1>1)
  ViewAngle1Offset=-0.01;
 if (ViewAngle1<-1)
  ViewAngle1Offset=0.01;

 // Рассчитываем и устанавливаем положение камеры
 vEyePt.x = 10*ViewAngle1;
 vEyePt.y = 10*ViewAngle1;
 vEyePt.z = -10;
 D3DXMatrixLookAtLH(&mView, &vEyePt, &vLookatPt, &vUpVec);
 pD3DDevice->SetTransform(D3DTS_VIEW, &mView);

 // Отображение объекта перед зеркалом
 ViewAngle += 0.1;
 RenderObject();

 pD3DDevice->GetTransform(D3DTS_WORLD, &mWorld);

 // Смещаем матрицу, чтобы рисовать объект сзади зеркала
 D3DXPlaneFromPoints(&plane, &pt1, &pt2, &pt3);
 D3DXMatrixReflect(&mReflectInMirror, &plane);
 pD3DDevice->SetTransform(D3DTS_WORLD, &mReflectInMirror);

 // Включаем обрезание краев объекта по плоскостям
 pD3DDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CW);

 pD3DDevice->SetClipPlane(0,
        *D3DXPlaneFromPoints(&plane, &pt2, &pt1, &vEyePt));
 pD3DDevice->SetClipPlane(1,
        *D3DXPlaneFromPoints(&plane, &pt4, &pt2, &vEyePt));
```

```
 pD3DDevice->SetClipPlane(2,
         *D3DXPlaneFromPoints(&plane, &pt3, &pt4, &vEyePt));
 pD3DDevice->SetClipPlane(3,
         *D3DXPlaneFromPoints(&plane, &pt1, &pt3, &vEyePt));
 pD3DDevice->SetRenderState(D3DRS_CLIPPLANEENABLE,
   D3DCLIPPLANE0 | D3DCLIPPLANE1 | D3DCLIPPLANE2 | D3DCLIPPLANE3);

 // Отображаем объект сзади зеркала
 RenderObject();

 // Устанавливаем параметры отображения зеркала
 pD3DDevice->SetTransform(D3DTS_WORLD, &mWorld);
 pD3DDevice->SetRenderState(D3DRS_CLIPPLANEENABLE, 0x00);
 pD3DDevice->SetRenderState(D3DRS_CULLMODE, D3DCULL_CCW);

 // Включаем прозрачность
 pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, TRUE);
 pD3DDevice->SetRenderState(D3DRS_SRCBLEND,  D3DBLEND_SRCALPHA);
 pD3DDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_INVSRCALPHA);

 // Отображаем прямоугольник зеркала
 pD3DDevice->SetStreamSource(0, pMirror, 0, sizeof(sVERTEX));
 pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE);
 pD3DDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);

 // Отключаем прозрачность
 pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE, FALSE);
}
```

Посмотрим по частям, что здесь происходит. В самом начале включаем освещение и рассчитываем коэффициенты `ViewAngle` и `ViewAngle1`. Первый коэффициент вращает тубу, а второй влияет на положение сцены, двигая ее по экрану, чтобы объект иногда выходил за пределы экрана.

Далее идет установка матрицы положения объекта и вызывается функция `RenderObject`. Эта функция рисует объект, а ее код показан в листинге 5.31.

Листинг 5.31. Отображение объекта

```
void RenderObject()
{
 D3DXMATRIXA16 matLocal, matWorldSaved;

 // Поворачиваем сцену
 D3DXMatrixRotationY(&ObjectMatrix, ViewAngle);
```

```
 // Запоминаем текущую матрицу
 pD3DDevice->GetTransform(D3DTS_WORLD, &matWorldSaved);

 // Устанавливаем матрицу положения объекта
 D3DXMatrixMultiply(&matLocal, &ObjectMatrix, &matWorldSaved);
 pD3DDevice->SetTransform(D3DTS_WORLD, &matLocal);

 // Устанавливаем материал
 pD3DDevice->SetMaterial(&mat1);

 // Отображаем сетку
 for (DWORD i=0; i<dwNumMaterials; i++)
  pObject->DrawSubset(i);

 // Обнуляем материал
 pD3DDevice->SetMaterial(&mat2);

 // Восстанавливаем материал
 pD3DDevice->SetTransform(D3DTS_WORLD, &matWorldSaved);
}
```

Как говорится, без дополнительных комментариев, потому они тут просто не нужны. Все, что здесь описано, нам уже знакомо, а по комментариям в листинге можно понять зачем. Функция `RenderObject` необходима для отображения объекта тубы, ну а вынесли мы ее в отдельную функцию потому, что отображение будет происходить дважды — перед зеркалом и за ним.

Но вернемся к листингу 5.30 и закончим его рассмотрение. Самое интересное еще впереди, и именно в этом листинге.

После отображения первой тубы сдвигаем матрицу:

```
D3DXPlaneFromPoints(&plane, &pt1, &pt2, &pt3);
D3DXMatrixReflect(&matReflectInMirror, &plane);
```

Вот тут у нас есть две интересные функции. Первая `D3DXPlaneFromPoints` строит поверхность по трем вершинам. В первом параметре необходимо передать переменную типа `D3DXPLANE`, куда будет помещена поверхность, а в остальных трех указываются вершины, по которым и строится поверхность.

Следующая функция создает зеркальную матрицу и помещает ее в первый параметр для переданной в качестве второго параметра поверхности.

Положение второй тубы вычислили и теперь необходимо сделать так, чтобы при отображении все, что выходит за пределы зеркала, обрезалось. Для этого можно создать поверхности отсечения с помощью метода `SetClipPlane`. В нашем случае создается 4 поверхности отсечения:

```
pD3DDevice->SetClipPlane(0, *D3DXPlaneFromPoints(
                &plane, &pt2, &pt1, &vEyePt));
pD3DDevice->SetClipPlane(1, *D3DXPlaneFromPoints(
                &plane, &pt4, &pt2, &vEyePt));
pD3DDevice->SetClipPlane(2, *D3DXPlaneFromPoints(
                &plane, &pt3, &pt4, &vEyePt));
pD3DDevice->SetClipPlane(3, *D3DXPlaneFromPoints(
                &plane, &pt1, &pt3, &vEyePt));
```

У метода всего два параметра: индекс устанавливаемой поверхности и сама поверхность.

Для создания поверхности используем функцию `D3DXPlaneFromPoints`, которая строит то, что нам нужно по трем точкам и помещает это в первый параметр. В принципе, пару абзацев назад мы уже говорили об этой функции.

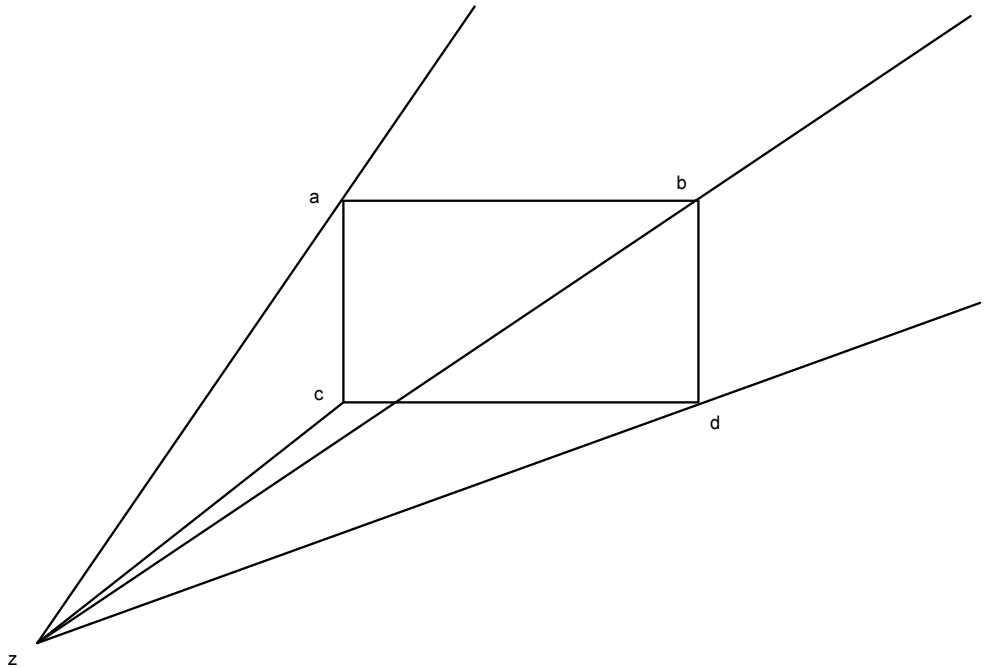

Рис. 5.15. Плоскости отсечения

Теперь посмотрим, что именно создается. Я особо не художник, но постарался графически показать, что создается на рис. 5.15. На рисунке показан прямоугольник, который символизирует зеркало. Точка z, откуда идут лучи, — это точка, из которой мы смотрим на мир. Поверхности отсечения строятся по двум точкам прямоугольника зеркала и по точке, из которой мы смотрим.

Например, первая поверхность отсечения выглядит как объединение точек zab. Лучи на рисунке не зря движутся дальше, потому что поверхность не ограничивается этими точками, она бесконечна, просто проходит через указанные точки.

Таким образом, поверхности отсечения обрезают все, что выйдет за пределы зеркала. Если посмотреть на сцену из точки z, то вы просто не увидите ничего, что за пределами прямоугольника.

Чтобы обрезание заработало, его необходимо еще и включить. Для этого используется метод `SetRenderState`:

```
pD3DDevice->SetRenderState(D3DRS_CLIPPLANEENABLE,
    D3DCLIPPLANE0 | D3DCLIPPLANE1 | D3DCLIPPLANE2 | D3DCLIPPLANE3);
```

С помощью этого метода устанавливается свойство отображения `D3DRS_CLIPPLANEENABLE`, т. е. обрезание. В качестве второго параметра необходимо указать, какие у нас существуют плоскости отсечения. Мы создали поверхности с индексами от 0 до 3, и чтобы Direct3D использовал их для отсечения, нужно указать четыре флага `D3DCLIPPLANEn`, где `n` от 0 до 3.

Вот теперь вызываем функцию `RenderObject`, чтобы отобразить вторую тубу в новом положении и с учетом поверхностей отсечения.

Самое сложное сделано. Теперь необходимо только включить прозрачность (чтобы сквозь прямоугольную поверхность увидеть тубу, как в зеркале) и, установив буфер вершин, показать объект, символизирующий зеркало.

Запустите пример, и убедитесь, что объект отражения тубы не выходит за пределы прямоугольника, символизирующего зеркало. Таким образом, можно создавать кривые зеркала, если при этом искривлять матрицу объекта отражения.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Mirror.

5.15. Продвинутое зеркало

Зеркало, которое мы рассмотрели в *разд. 5.14*, создать достаточно легко, но если у вас будет сложная сцена, то реализовать его в зеркале будет намного сложнее и накладно для видеопроцессора. Есть способ лучше — можно воспользоваться анимацией текстуры. Например, в *разд. 5.10* мы создали эффект анимации чайника на поверхности сферы. Достаточно только нарисовать перед сферой такой же чайник и получится эффект зеркала.

В *разд. 5.11* мы также рассматривали эффект рисования на текстуре. Этот пример еще больше подходит для адаптации к зеркалу, тем более что даже во время отображения мы указали, чтобы использовать зеркальные координаты. Попробуйте нарисовать вокруг кегли комнату и звезду так, чтобы содержимое текстуры оказалось зеркальным для вашей сцены (рис. 5.16).

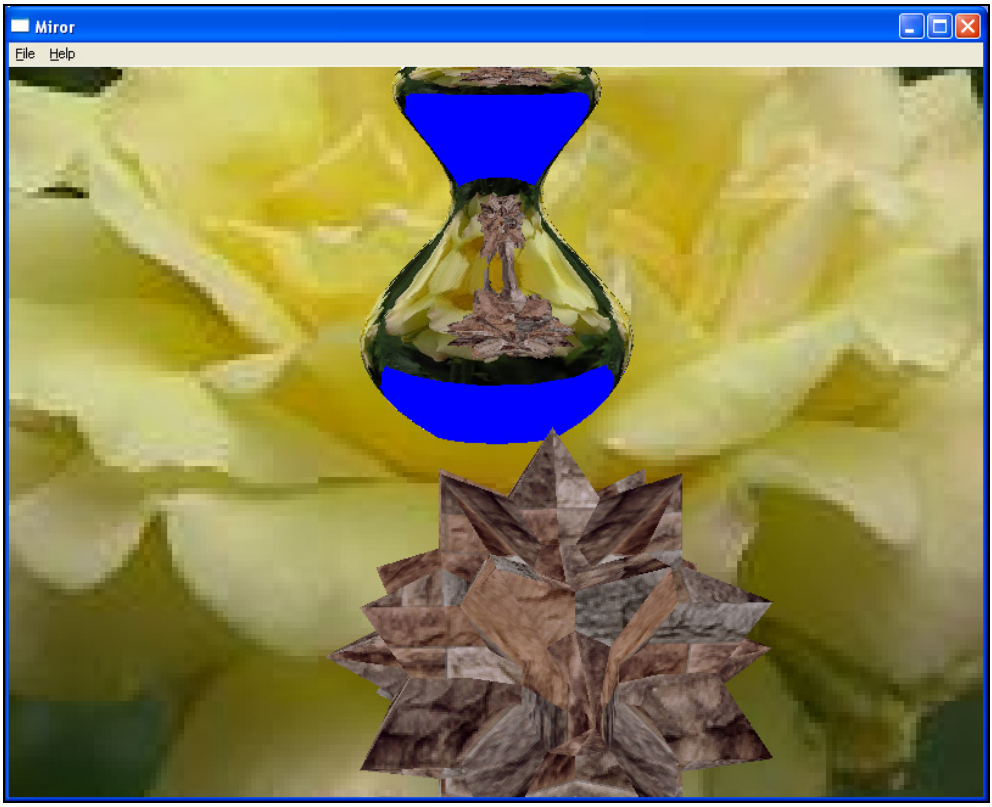

Рис. 5.16. Результат зеркального отображения сцены на кегле

Мы не будем рассматривать код этого примера, потому что все необходимые для этого задания у нас есть и есть почти готовый пример. Достаточно только добавить несколько строк кода, к примеру из *разд. 5.11*, и все будет готово. Если же у вас не получится, то обратитесь к компакт-диску, где есть готовое решение.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Mirror2.

5.16. Введение в вершинный шейдер

Все эффекты, которые мы рассматривали до этого, были простыми и чаще всего основывались на обмане зрения или простых преобразованиях. Но мир не стоит на месте и такой простотой сложно удивить профессионала. Действительно, с их помощью можно создать 3D-мир и сделать что-то интересное, но не более того. Шедевра, претендующего на 1-е место на демо-вечеринке у вас наверняка не получится.

Конечно, можно вернуться к 2D-миру и формировать 3D-сцену ручками, самостоятельно просчитывая освещение и тени. Вот такую работу, если она, конечно, получится красивой, оценит любой профессионал.

Но и в 3D-мире есть способ создания более сложных сцен, которые поразят даже ваше воображение. Для этого необходимо всего лишь воспользоваться *вершинным шейдером*. Что это такое? Это небольшая программа, которая позволяет управлять регистрами видеокарты, благодаря чему можно создать невероятные по красоте эффекты. Для написания таких программ используется специализированный *язык HLSL* (High Level Shader Language, высокоуровневый язык шейдеров).

Существует два типа шейдеров — вершинный и пиксельный. Давайте рассмотрим, что они представляют и как используются.

Вершинный шейдер, как следует из названия, предназначен для работы вершины. Программа шейдера выполняется для каждой вершины во время ее обработки графической картой. Здесь можно изменять цвета вершин, координаты текстуры и т. д. Все это мы можем делать в программе с помощью интерфейсов Direct3D, а можем возложить на вершинный шейдер.

В *разд. 5.12* мы уже использовали язык HLSL, но там мы не управляли регистрами, а только упростили код программы за счет переноса некоторых установок текстуры в специальную строку эффекта, и интерфейс `ID3DXEffect` автоматически производил все необходимые настройки, в соответствии с этой строкой.

Пиксельный шейдер используется при формировании фрагментов (пикселов) сцены. С помощью пиксельного шейдера вы можете попиксельно воздействовать на сцену, создавая эффекты сумасшедшей сложности.

Если раньше в компьютерных играх для создания теней и освещения использовались ручные расчеты, а сцена формировалась, как и десятки лет назад с помощью рисования трехмерного мира на двумерной плоскости, то теперь подобные эффекты легко реализовать с помощью пиксельных шейдеров.

Изначально шейдеры приходилось писать на языке ассемблера, но это достаточно сложно. Если вы знакомы с этим языком, то, наверно, можете предста-

вить себе, как будет выглядеть функция преобразования матриц. А ведь без матриц в трехмерной графике никуда.

В настоящее время для работы с шейдерами используется язык HLSL, о котором мы уже говорили и даже слегка использовали в *разд. 5.12*. Давайте немного познакомимся с языком HLSL, а затем попробуем написать более качественные (по сравнению с уже рассмотренными примерами) эффекты.

Все примеры, которые мы рассматривали ранее, будут великолепно работать даже на старых видеоускорителях, разработанных в конце 90-х годов. Эти ускорители до сих пор могут стоять в пользовательских компьютерах (например, Riva TNT), и вы должны учитывать, что в них нет поддержки вершинного или пиксельного шейдера. Даже в современных видеокартах шейдеры могут не работать. Например, на моем ноутбуке установлен видеочип Extrime Graphics от Intel. Чип хороший, но шейдеры не воспринимает, поэтому последующие примеры на подобных компьютерах работать просто не будут.

Тем не менее познакомиться с этим языком никому не помешает. Тем более что к моменту выхода книги современные видеоускорители от ATI и NVIDIA получат еще большее распространение, а к тому моменту, как она попадет к вам в руки, вы прочитаете ее и напишете свою игру или демо-ролик, видеокарты с поддержкой шейдеров станут еще более распространенными.

Если немного перефразировать слова самого богатого человека планеты, то получится, что современный бизнес движется со скоростью мысли. Пока мы раздумываем, что будет завтра, это завтра уже наступает и охватывает всю планету. Не удивлюсь, если уже к выходу книги технология HLSL устареет. Хотя нет, она достаточно перспективна, и скорей всего просто появится новая, более совершенная версия шейдеров. Даже уже сейчас существует несколько версий.

Как и у любого другого языка, в HLSL есть свои типы данных. Основным является вещественное число типа `float`, размером в 32 бита, и этот тип должен быть во всех видеокартах. Остальные типы могут и отсутствовать, но будут эмулироваться через тип `float`. Следующий пример показывает, как можно объявить вещественную переменную `Variable`:

```
float Variable;
```

Пока все идентично описанию переменных в C++. Переменная может быть статичной (`static`), внешней (`extern`), форменной (`unoform`) или константой (`const`):

```
extern float a;
static float b;
const float c;
unoform float d;
```

Ключевое слово `extern` говорит о том, что переменная `a` может изменяться или читаться приложением с помощью методов `GetVertexShaderConstantx` и `SetVertexShaderConstantx`.

Переменную `b` изменить или прочитать из приложения нельзя, потому что она является статичной. Зато программа шейдера может использовать ее. Если необходимо, чтобы приложение не могло изменять и повлиять на какую-то переменную и соответственно нарушить тем самым работу программы шейдера.

Переменная `c` является константой и ее изменить не может никто. Последняя переменная `d` объявлена как `unoform`. Такая переменная может быть изменена только между вызовами функций рисования.

В языке HLSL есть еще следующие простые типы данных:

- `double` — вещественное число в 64 бита;
- `half` — вещественное число в 16 бит;
- `int` — целое число;
- `bool` — булево значение.

Как видите, имена типов очень похожи на те, что есть у языка C++. Это простые типы данных, но ни для одного языка программирования этого не будет достаточно. Например, для нас, как для программистов графических приложений, очень важны матрицы и векторы. Для объявления каждого из этих типов есть несколько вариантов. Например, самый распространенный способ объявить матрицу:

```
типMxN переменные
```

Сначала пишется тип данных, а потом размерность. Значения `M` и `N` определяют размеры матрицы. Чаще всего мы работаем с матрицами 4×4, где каждый элемент является вещественным значением. Такая матрица может быть объявлена следующим образом:

```
float4x4 Matrix;
```

В этом примере объявлена матрица с именем `Matrix`, размером 4×4, позволяющая хранить вещественные числа.

Теперь посмотрим на вектор. Чаще всего его объявляют как:

```
float4 Vector;
```

Здесь объявлена векторная переменная `Vector`, состоящая из 4-х значений. Тоже самое на языке HLSL можно написать еще и следующим образом:

```
float Vector[4];
```

Инициализация переменных происходит так же, как и на C++, поэтому тут лишние пояснения не нужны. Увидим все на примере. Математические операторы тоже ничем не отличаются, и вы можете использовать операторы сложения, вычитания, деления и т. д. точно так же, как и в C++.

Теперь переходим к структурам. Они тоже похожи на C++. В общем виде это выглядит так:

```
struct имя
{
 Описание полей структуры
}
```

Следующий пример показывает структуру `VS_OUTPUT`:

```
struct VS_OUTPUT
{
 float4 pos :POSITION;
 float4 col :COLOR;
};
```

При работе с шейдерами нужно быть внимательным, дело в том, что существует несколько версий вершинных шейдеров и каждая из них отличается возможностями и количеством поддерживаемых команд. При этом не каждая видеокарта может поддерживать запрашиваемую вами версию. До версии 1.4 шейдеры поддерживают только до 12-ти команд на всю программу. Для начала нам будет достаточно, но если нужно больше команд, то придется выбирать версию шейдера поновее, например, версию 1.4 (28 команд и 8 констант) или переходить на ветку 2.0.

На этом пока закончим рассмотрение теории шейдеров и перейдем к практической части. О языке HLSL можно писать отдельную книгу, и полное описание функций языка отнимет не менее 200 страниц, а на описание примеров для закрепления теории уйдет в два раза больше. Вспомните, как в *разд. 5.12* мы с помощью шейдера использовали свойства текстур Direct3D, а этих свойств много и все их можно использовать. Вместо теории мы рассмотрим несколько интересных практических примеров, и на них изучим основные возможности шейдеров.

5.17. Простейший пример шейдеров

Давайте сначала напишем код шейдера, а потом уже посмотрим, как его использовать в программе. В качестве рассмотрения возьмем пример из *разд. 2.10*, где мы создавали вращающийся куб, только теперь за вращение станет отвечать шейдер. В дополнение к этому, шейдер будет изменять цвет куба.

5.17.1. Пишем шейдер

Итак, код этого шейдера показан в листинге 5.32.

Листинг 5.32. Шейдер, изменяющий положение и цвет вершин

```
float4x4 mat : WORLDVIEWPROJECTION;
float TimeFactor = 1.0f;

struct VS_OUTPUT
{
    float4 pos  : POSITION;
    float4 col : COLOR0;
};


VS_OUTPUT ShaderFunc(float4 Pos : POSITION)
{
    VS_OUTPUT Out = (VS_OUTPUT)0;

    // Set position
    Out.pos = mul(Pos, mat);

    // Set color
    Out.col.r = TimeFactor;
    Out.col.g = Out.col.r/10;
    Out.col.b = 1- Out.col.g;

    return Out;
}


technique Transform
{
    pass P0
    {
        VertexShader = compile vs_1_1 ShaderFunc();
    }
}
```

Весь этот код необходимо сохранить в файле simple.vsh. Впоследствии мы загрузим его из программы на C++ и используем для вывода куба.

Теперь посмотрим, что у нас тут происходит. В самом начале объявляются две глобальные переменные:

```
float4x4 mat : WORLDVIEWPROJECTION;
float TimeFactor = 1.0f;
```

Первая переменная имеет тип матрицы размером 4×4 и состоит из вещественных чисел. Обратите внимание, что после объявления переменной стоит двоеточие и ключевое слово WORLDVIEWPROJECTION. Это даже выражение, которое состоит из WORLD, VIEW и PROJECTION, т. е. из трех матриц: мировой, просмотра и проекции.

Следующая переменная TimeFactor, которая имеет тип float и ей сразу же присваивается значение 1.0.

Вторая переменная имеет значение по умолчанию, поэтому программа не обязана изменять его. Матрица положения не имеет значений, поэтому тут необходимо указать значение, прежде чем вызывать код шейдера, иначе могут возникнуть проблемы.

Следующим этапом объявляется структура VS_OUTPUT:

```
struct VS_OUTPUT
{
    float4 pos  : POSITION;
    float4 col : COLOR0;
};
```

Эта структура имеет определенное значение, я привык давать ей имя VS_OUTPUT, но это не есть обязательно. Такая структура используется для связи входящих и выходящих вершин, т. е., устанавливая значения полей структуры, мы влияем на результирующее состояние вершины.

Структура состоит из двух полей типа float4, т. е. из двух векторов, по 4 значения в каждом. Первая переменная с именем pos — это вектор позиции вершины. Об этом говорит ключевое слово (модификатор) POSITION после объявления переменной. Вторая переменная col — это цвет вершины, об этом говорит ключевое слово (модификатор) COLOR0. Получается, что через pos и col мы сможем устанавливать цвет и позицию вершины.

Имена переменных могут быть любыми, а вот тип и модификатор обязательно должны быть такими. Если попытаться определить позицию через переменную float, а не вектор float4, то произойдет ошибка.

Теперь объявляем функцию с именем ShaderFunc:

```
VS_OUTPUT ShaderFunc(float4 Pos : POSITION)
```

Имя может быть любым, а вот возвращаемое значение и передаваемые параметры должны быть именно такими. Что возвращает функция? А возвращает она структуру типа VS_OUTPUT, а на входе она получает векторную переменную pos типа float4. Функция получает в качестве параметра позицию вер-

шины, а в результате должна вернуть структуру VS_OUTPUT, которую мы только что рассматривали.

Получив текущую позицию вершины, функция может изменить ее и, назначив свет, вернуть результат через структуру VS_OUTPUT. Установленные нами в функции значения будут использоваться видеокартой при формировании сцены. Данная функция должна вызываться для каждой вершины, из которых будет состоять отображаемый через шейдер объект.

Теперь посмотрим на код функции. В самом начале объявляется переменная Out, которая имеет тип VS_OUTPUT:

```
VS_OUTPUT Out = (VS_OUTPUT)0;
```

Через эту переменную мы сформируем значение, которое нужно вернуть. Давайте же посмотрим, как формируются значения возвращаемой структуры. Сначала мы устанавливаем позицию с помощью функции mul:

```
Out.pos = mul(Pos, mat);
```

Функция mul перемножает два значения, указанных в качестве параметра, и возвращает результат. В качестве параметров могут быть скаляр (число), вектор или матрица. В данном случае мы перемножаем текущую позицию вершины с матрицей положения, получая, таким образом, положение вершины на сцене с учетом преобразований, которые накладываются матрицами WORLD, VIEW и PROJECTION. Именно эти матрицы преобразования передаются нам через переменную mat.

При работе с функцией mul нужно быть очень осторожным. Дело в том, что, несмотря на операцию перемножения, от перемены мест параметров результат меняется. Это матрицы, и здесь лучше ничего не менять местами, иначе результат будет неожиданным. Например, результат вызова функции mul(Pos, mat) даст один результат, а mul(mat, Pos) другой.

Теперь задаем цвет вершин:

```
Out.col.r = TimeFactor;
Out.col.g = Out.col.r/10;
Out.col.b = 1- Out.col.g;
```

За цвет у нас отвечает поле col структуры Out. Это поле также является структурой, которое состоит из трех значений r, g и b (красный, зеленый и голубой цвет вершины). Значение красного цвета будет равно переменной TimeFactor, зеленая составляющая будет равна красной, деленной на 10, а голубая равна единице минус зеленая составляющая.

Можно было бы явно задать цвет вершины, но мы рассчитываем его на основе значения переменной TimeFactor, чтобы во время анимации происходило не только вращение куба, но и плавное изменение цвета.

Если вы хотите явно задать составляющие цвета, то можно указать их значения напрямую. Например, следующий код задает красный цвет:

```
Out.col.r = 1;
Out.col.g = 0;
Out.col.b = 0;
```

Каждая составляющая цвета изменяется от 0 до 1. Единица соответствует полной яркости. В нашем случае ярким является красный цвет, а остальные просто отсутствуют.

Мы привыкли работать с цветом, когда он изменяется от 0 до 255 (или FF в шестнадцатеричном исчислении). Преобразовать в диапазон от 0 до 1 не так уж сложно:

```
Out.col.r = 1/255*r;
Out.col.g = 1/255*g;
Out.col.b = 1/255*b;
```

В данном случае `r`, `g` и `b` — это значения каждой из составляющих в диапазоне от 0 до 255. Чтобы их установить, мы сначала делим 1 на 255, а потом умножаем на значение переменной `r`, `g` или `b` соответственно.

Самое последнее, что делает наша функция, — возвращает сформированную структуру `Out`. Как и в языке C++, это делается с помощью оператора `return`:

```
return Out;
```

Теперь у нас есть еще одна функция с ключевым словом `technique`:

```
technique Transform
{
    pass P0
    {
        VertexShader = compile vs_1_1 ShaderFunc();
    }
}
```

В *разд. 5.12* мы уже немного познакомились с такой функцией, а сейчас мы посмотрим на нее с другой стороны. Функция `technique` позволяет задать технику отображения. Именно ее мы будем вызывать из программы на C++. Внутри такой функции определяются шаги отображения `pass`. В данном случае только один шаг, который выполняет строку:

```
VertexShader = compile vs_1_1 ShaderFunc();
```

Чтобы лучше понять смысл выполняемых этой строкой действий, попробую проговорить русским языком, что здесь происходит: вершинному буферу

присвоить результат компиляции вершинного шейдера версии 1.1 из функции `ShaderFunc`.

Теперь то же самое, только смотрим на код. Переменная `VertexShader` указывает на установленный сейчас вершинный шейдер. Ему мы присваиваем результат выполнения оператора `compile` (компиляция). Далее указывается версия шейдера. Это делается в формате `vs_n_m`. Значения `n` и `m` определяют версию. Чем выше версия, тем больше у него возможностей и больше инструкций он может выполнить, но и тем меньше вероятность, что данная версия будет поддерживаться видеокартой. Поэтому не стоит выбирать последнюю версию, иначе программа может не запуститься на вашем компьютере. Нам достаточно будет версии 1.1. Последнее, что нужно указать, — функцию с шейдером, которую необходимо откомпилировать и установить в качестве вершинного шейдера в переменную `VertexShader`.

Сразу же заметим, что пиксельный шейдер устанавливается примерно таким же образом, только его нужно записать в переменную `PixelShader`. Например:

```
PixelShader = compile ps_1_1 ShaderFunc();
```

Обратите внимание, что версия пиксельного шейдера указывается как в формате `ps_n_m`.

5.17.2. Использование вершинного шейдера

Создайте новое приложение и добавьте инициализацию Direct3D. Код отображения куба взят из *разд. 2.10*. Обратитесь к *разд. 5.9.1* и вы увидите, что там у нас была функция `CreateQuad`. Функция была создана не зря, а чтобы вы легко смогли скопировать весь ее код в новое приложение. Сделайте это, и не забудьте скопировать все необходимые переменные, а именно переменные для буфера вершин, для буфера индексов и константы, которые описывают количество элементов в буферах. Единственное, что необходимо сделать, — убрать из описания вершины цветовую составляющую. Вершина будет описываться только позицией:

```
struct sVertex
{
  float x, y, z;
};
```

Шейдер, который мы написали в *разд. 5.17.1*, генерирует цвет вершин, поэтому его не нужно записывать в буфер и усложнять себе жизнь.

Помимо этого объявим две матрицы:

```
D3DMATRIX View = {
  1, 0, 0, 0,
  0, 1, 0, 0,
```

```
  0, 0, 1, 0,
  0, 0, 5, 1,
};

D3DXMATRIX matProjection;
```

Первая матрица будет хранить видовую матрицу. Она используется во время отображения, и устанавливать ее на этапе инициализации не нужно. Вторая переменная `matProjection` — это матрица проекции. В нее мы запишем текущую установленную во время инициализации матрицу.

Из *разд. 5.12* мы знаем, что для использования эффекта нам необходим интерфейс `ID3DXEffect`, поэтому добавим переменную `pEffect` для хранения указателя на этот интерфейс:

```
ID3DXEffect* pEffect;
```

Но это еще не все, нам нужно описать вершину, а для этого применяется интерфейс `IDirect3DVertexDeclaration9`, поэтому введем еще следующую глобальную переменную:

```
IDirect3DVertexDeclaration9 *VertDecl;
```

Теперь нужно проинициализировать новые переменные. Мы же не зря их добавили в программу. После инициализации и заполнения вершинного и индексного буферов, которые вы взяли из *разд. 2.10*, пишем следующий код:

```
// Создаем эффект (шейдер) из файла
D3DXCreateEffectFromFile(pD3DDevice, "simple.vsh", 0, 0, 0, 0,
     &pEffect, 0);
pEffect->SetTechnique("Transform");

// Структура объявления вершин
D3DVERTEXELEMENT9 decl[]=
{
  {0, 0, D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_POSITION, 0},
   D3DDECL_END()
};

// Создаем объявление вершин
pD3DDevice->CreateVertexDeclaration(decl, &VertDecl);

// Создаем переменную матрицы проекции
float Aspect = (float)iWidth / (float)iHeight;
D3DXMatrixPerspectiveFovLH(&matProjection, D3DX_PI/4.0f,
          Aspect, 0.1f, 1000.0f);
```

В самом начале мы создаем шейдер с помощью функции D3DXCreateEffectFromFile, которая очень похожа на D3DXCreateEffect, рассмотренную нами ранее, в *разд. 5.12*. Функция загружает шейдер из указанного файла и в общем виде выглядит следующим образом:

```
HRESULT D3DXCreateEffectFromFile(
    LPDIRECT3DDEVICE9 pDevice,
    LPCSTR pSrcFile,
    CONST D3DXMACRO* pDefines,
    LPD3DXINCLUDE pInclude,
    DWORD Flags,
    LPD3DXEFFECTPOOL pPool,
    LPD3DXEFFECT* ppEffect,
    LPD3DXBUFFER *ppCompilationErrors
);
```

Функция претерпела серьезные изменения по сравнению с 8-й версией DirectX, и для нее значительно увеличилось количество передаваемых параметров. Минимально необходимо (и именно так мы поступим) указать три параметра:

- pDevice — указатель на Direct3D-устройство;
- pSrcFile — строку, которая содержит имя загружаемого файла;
- ppEffect — указатель на указатель интерфейса эффекта, куда будет записан результат.

Эффект загружен, но в нем может быть несколько функций типа technique. Необходимо выбрать ту, которую нужно использовать в данный момент. Поскольку у нас только одна функция technique, то можно сделать это на этапе инициализации с помощью метода SetTechnique, интерфейса ID3DXEffect:

```
pEffect->SetTechnique("Transform");
```

Методу передается имя функции. У нас она называется Transform и именно это имя мы передаем.

После этого объявляется переменная массива decl типа D3DVERTEXELEMENT9. Эта структура необходима для того, чтобы сообщить Direct3D, какие данные будут передаваться в шейдер. В массиве может быть несколько элементов, и каждый будет описывать один компонент данных вершин. В нашем случае вершина описывается следующим образом:

```
D3DVERTEXELEMENT9 decl[]=
{
  {0, 0, D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_POSITION, 0},
   D3DDECL_END()
};
```

Здесь объявлен массив значений типа D3DVERTEXELEMENT9. Внутри описания в фигурных скобках через запятую мы должны описать параметры структуры. В нашем случае только один параметр:

```
{0, 0, D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_POSITION, 0}
```

Здесь в фигурных скобках перечислены значения элементов структуры D3DVERTEXELEMENT9. Описание должно заканчиваться оператором D3DDECL_END().

Структура D3DVERTEXELEMENT9 в общем виде выглядит следующим образом:

```
typedef struct _D3DVERTEXELEMENT9 {
    BYTE Stream;
    BYTE Offset;
    BYTE Type;
    BYTE Method;
    BYTE Usage;
    BYTE UsageIndex;
} D3DVERTEXELEMENT9;
```

Давайте рассмотрим параметры этой структуры, потому что их понимание необходимо для дальнейшего изучения эффектов.

Stream — номер потока данных. Шейдер может получать данные из нескольких потоков, но мы пока что воспользуемся только одним, поэтому данный параметр сейчас равен нулю;

Offset — смещение в буфере, где хранятся данные. В нашем случае только одна запись. Если бы их было несколько, то в этом параметре необходимо было бы указать смещение от начала. Например, если бы вершина описывалась координатами и нормалью, то описание вершины выглядело бы следующим образом:

```
D3DVERTEXELEMENT9 decl[] =
{
  {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_POSITION, 0},
  {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
            D3DDECLUSAGE_NORMAL, 0},
  D3DDECL_END()
};
```

У первой записи второй параметр равен нулю. Это значит, что в вершинном буфере сначала идет позиция вершины, которая описывается тремя значениями типа float (4 байта). Получается, что описание позиции занимает

12 байт (3 значения координаты, умноженные на размер типа `float`, который описывает каждую координату). Вторая строка описывает нормаль, которая смещена в буфере вершин на значение позиции вершины, поэтому второй параметр равен 12.

`Type` — тип данных, используемый в описании. У нас координаты вершины описываются типом `float`, а значит, здесь необходимо указать флаг `D3DDECLTYPE_FLOAT3`.

`Method` — метод обработки данных. Мы будем применять значение по умолчанию и флаг `D3DDECLMETHOD_DEFAULT`.

`Usage` — для чего будут использоваться данные. В нашем случае данные описывают позицию. Вспомните, как мы описали структуру `sVertex`. В ней только три координаты вершины, описывающие ее позицию, и ничего больше. Именно это нужно указать в нашей структуре с помощью флага `D3DDECLUSAGE_POSITION`. Помимо этого вы можете указывать следующие флаги:

- `D3DDECLUSAGE_NORMAL` — нормаль;
- `D3DDECLUSAGE_TEXCOORD` — координата текстуры;
- `D3DDECLUSAGE_COLOR` — цвет;
- `D3DDECLUSAGE_FOG` — туман.

Это основные флаги, которые вы будете использовать.

Последний параметр структуры `D3DVERTEXELEMENT9` (`UsageIndex`) позволяет задать индекс использования. Задавая разные индексы, можно указать множественные типы использования. Мы индексы применять не будем, поэтому у нас этот параметр равен нулю.

Теперь у нас есть структура, описывающая вершину, но это только структура. Чтобы передать ее Direct3D, необходимо еще создать интерфейс `IDirect3DVertexDeclaration9`, переменную такого типа мы уже объявили. Создание интерфейса происходит с помощью метода `CreateVertexDeclaration`:

```
pD3DDevice->CreateVertexDeclaration(decl, &VertDecl);
```

Первый параметр — это массив структур типа `D3DVERTEXELEMENT9`, а второй параметр — это переменная, указывающая на интерфейс `IDirect3Dvertex-Declaration9`, который необходимо создать.

И напоследок, в глобальной переменной `matProjection` создаем матрицу проекции. Она будет точно такая же, как и на этапе инициализации Direct3D, потому что код просто скопирован из функции `DX3DInitZ`.

Теперь переходим к отображению, которое будем производить в функции `DrawScene` (листинг 5.33).

Листинг 5.33. Функция отображения куба с помощью шейдера

```
void DrawScene()
{
 // Изменяем угол поворота
 ViewAngle+=ViewAngleOffset;
 if (ViewAngle>10)
  ViewAngleOffset=-0.05f;
 if (ViewAngle<0)
  ViewAngleOffset=0.05f;

 // Рассчитываем мировую матрицу
 float b=ViewAngle;
 D3DMATRIX World = {
  cos(b)*cos(b), cos(b)*sin(b), sin(b), 0,
  -sin(b), cos(b), 0, 0,
  -sin(b)*cos(b), -sin(b)*sin(b), cos(b), 0,
  0, 0,  0, 1,
 };

 // Объединяем матрицы проекции, просмотра и мировую
 D3DXMATRIX Full;
 D3DXMatrixMultiply(&Full, (D3DXMATRIX*)&World, (D3DXMATRIX*)&View);
 D3DXMatrixMultiply(&Full, &Full, &matProjection);

 // Устанавливаем переменные шейдера
 pEffect->SetValue("TimeFactor", &ViewAngle, D3DX_DEFAULT);
 pEffect->SetValue("mat", &Full, D3DX_DEFAULT);

 // Указываем шейдеру формат вершин
 pD3DDevice->SetVertexDeclaration(VertDecl);

 // Устанавливаем буфер вершин и индексы
 pD3DDevice->SetStreamSource(0, vBuffer, 0, sizeof(sVertex));
 pD3DDevice->SetIndices(iBuffer);

 // Отображаем объект с использованием шейдера
 UINT uPass;
 pEffect->Begin(&uPass, 0);
 for (int i=0; i<uPass; i++)
 {
  pEffect->Pass(i);
```

```
  pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,
                0, 0, iVertsNum, 0, iPointsNum);
 }
 pEffect->End();
}
```

Давайте посмотрим, что здесь происходит. В самом начале мы изменяем переменную `ViewAngle`. На ее основе будет определяться угол, на который нужно повернуть куб, и на основе этой переменной будет определяться цвет вершин.

После этого мы рассчитываем матрицу поворота, на основе переменной `ViewAngle`. Результат объединяется с матрицей вида и проекции, которые мы создали на этапе инициализации и сохранили в глобальных переменных `matProjection` и `View`.

Матрицу мы рассчитали, но устанавливать с помощью метода `SetTransform` не будем. Почему? Вспомните, что у нас происходит в шейдере. Каждая вершина будет устанавливаться на основе переданной в шейдер матрицы, поэтому текущие настройки проекции не повлияют на положение вершин.

Теперь нам необходимо передать шейдеру значения переменных. Это выполняется с помощью метода `SetValue`, которому передается три значения:

- строка, которая содержит имя переменной в шейдере. У нас две переменных `TimeFactor` и `mat`;
- значение переменной, которую необходимо установить;
- размер устанавливаемых данных. Можно указать значение `D3DX_DEFAULT`, и тогда размер данных будет определен по размеру переменной, указанной во втором параметре.

Следующий код показывает, как мы должны задать значения переменных нашего простого шейдера:

```
pEffect->SetValue("TimeFactor", &ViewAngle, D3DX_DEFAULT);
pEffect->SetValue("mat", &Full, D3DX_DEFAULT);
```

Теперь необходимо указать Direct3D, как описаны вершины, передаваемые в шейдер. Когда мы работали без шейдера, то мы использовали метод `SetFVF`, которому передаются флаги, например:

```
pD3DDevice->SetFVF(D3DFVF_XYZ | D3DFVF_DIFFUSE);
```

Но в случае с шейдерами такая песня не проходит. Тут необходимо указать интерфейс типа `IDirect3DVertexDeclaration9`. Интерфейс у нас есть, а указать его можно с помощью метода `SetVertexDeclaration`:

```
pD3DDevice->SetVertexDeclaration(VertDecl);
```

У метода всего один параметр — устанавливаемый интерфейс, в котором создано описание вершины. Все это у нас уже проделано.

Теперь можно устанавливать буфер вершин и индексный буфер. Тут ничего нового и все мы это уже делали:

```
pD3DDevice->SetStreamSource(0, vBuffer, 0, sizeof(sVertex));
pD3DDevice->SetIndices(iBuffer);
```

Самое последнее, что мы выполним:

```
UINT uPass;
pEffect->Begin(&uPass, 0);
for (int i=0; i<uPass; i++)
{
 pEffect->Pass(i);
 pD3DDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, 0,
        iVertsNum, 0, iPointsNum);
}
pEffect->End();
```

Вспомните, что в *разд. 5.12* мы уже рассматривали этот алгоритм вывода данных через шейдер.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Shader.

5.18. Управление освещением в шейдере

В *разд. 5.17* мы создали куб, который управлялся шейдером, но он выглядел неестественно. Это потому, что не было освещения, а без него все точки будут одного цвета и переходы между гранями сольются. Давайте рассмотрим, как добавить к нашему вращающемуся кубу освещение и сделать сцену более реалистичной.

Итак, не будем писать программу с нуля, а модифицируем то, что мы уже создали в *разд. 5.17*.

5.18.1. Нормали

Однако прежде чем добавлять освещение, необходимо определить нормали вершин, без которых объект не будет освещенным. *Нормаль* — вектор, определяющий направление в пространстве, длина которого равна единице. Получается, что нам необходимо добавить к каждой вершине прямоугольника нормаль.

В *разд. 5.8* мы уже использовали нормали, но тогда мы не заостряли внимание на процессе их создания. Если честно, то тогда я выставил направление "от фонаря", лишь бы сцена выглядела приемлемо. Сейчас поступим по всем законам и рассчитаем вектор.

Для начала нам необходимо переопределить описание вершины, теперь она будет хранить не только положение, но и нормаль:

```
struct sVertex
{
    D3DXVECTOR3 p; // Вектор положения
    D3DXVECTOR3 n; // Вектор нормали
};
```

Для удобства, позиция и нормаль имеют векторный тип `D3DXVECTOR3`. Давайте посмотрим, как заполнить поля этой структуры. Например, следующая строка заполняет позицию нулевой вершины:

```
svVortexList[0].p = D3DXVECTOR3(-0.5f, -0.5f, -0.5f);
```

Для расчета нормали к этой вершине необходимо выполнить строку кода:

```
D3DXVec3Normalize(&svVortexList[0].n, &svVortexList[0].p);
```

Здесь вызывается функция `D3DXVec3Normalize`. Этой функции передается два параметра:

- указатель на переменную типа `D3DXVECTOR3`, в которую будет записан результат расчета, т. е. вектор нормали;
- указатель на вектор `D3DXVECTOR3`, содержащий координаты вершины.

Как видите, с помощью функции `D3DXVec3Normalize` не нужно выдумывать координаты нормали, их легко рассчитать и установить в структуру описания вершины.

Теперь необходимо модифицировать массив описания вершины `D3DVERTEXELEMENT9` следующим образом:

```
D3DVERTEXELEMENT9 decl[]=
{
 {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                D3DDECLUSAGE_POSITION, 0},
 {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                D3DDECLUSAGE_NORMAL, 0},
 D3DDECL_END()
};
```

В *разд. 5.17.2* мы уже анализировали точно такое же объявление, когда рассматривали второй параметр структуры (смещение).

Остальной код инициализации не изменяется, поэтому его рассматривать не будем.

5.18.2. Шейдер

Теперь посмотрим на шейдер, который будет выводить вершины с учетом освещения. Он показан в листинге 5.34.

Листинг 5.34. Шейдер с учетом освещения

```
// Структура связи с выходными данными
struct VS_OUTPUT
{
    float4 pos  : POSITION;
    float4 col : COLOR0;
};

// Освещение
float3 Light = {0.0f, 0.0f, -1.0f};

// Глобальные переменные
float4x4 mat : WORLDVIEWPROJECTION;
float TimeFactor = 1.0f;

// Функция шейдера
VS_OUTPUT ShaderFunc(
 float4 Pos : POSITION,
 float3 Normal : NORMAL)
{
  VS_OUTPUT Out = (VS_OUTPUT)0;

  // Определение позиции
  Out.pos = mul(Pos, mat);

  // Определение цвета
  Out.col.r = TimeFactor/10;
  Out.col.g = 1-Out.col.r;
  Out.col.b = 1-Out.col.r;

  // Коррекция с учетом нормали
  float4 normalmap = mul(Normal, mat);
  Out.col *= dot(normalmap, Light);

  return Out;
}
```

```
// Функция technique
technique Transform
{
    pass P0
    {
        VertexShader = compile vs_1_1 ShaderFunc();
    }
}
```

Помимо уже знакомых нам глобальных переменных и описания структуры, появилось описание источника света:

```
float3 Light = {0.0f, 0.0f, -1.0f};
```

Мы его располагаем в нулевой точке и отодвигаем всего на единицу назад по оси Z. Функция `technique` ничем не отличается от уже рассмотренной, здесь только устанавливается в качестве вершинного шейдера функция `ShaderFunc`. Эта функция получает на входе позицию и нормаль.

```
VS_OUTPUT ShaderFunc(
 float4 Pos : POSITION,
 float3 Normal : NORMAL)
```

Именно эти данные мы описали как входные для шейдера, когда создавали массив структур `D3DVERTEXELEMENT9`.

Начало функции нам уже знакомо, потому что здесь мы задаем позицию и цвет вершины. Тут ничего не изменилось. Но затем необходимо откорректировать цвет с учетом нормалей, чтобы поверхность куба выглядела более реалистичной. Для этого выполняется следующий код:

```
float4 normalmap = mul(Normal, mat);
Out.col *= dot(normalmap, Light);
```

Сначала перемножаем нормаль с текущей матрицей положения, чтобы позиционировать нормаль в текущем положении вершины. После этого умножаем цвет на результирующую точку из двух векторов, которую возвращает функция `dot`. Этой функции необходимо передать два вектора (мы передаем вектор источника освещения и вектор преобразованной к матрице положения нормали), а в результате мы получаем создаваемую векторами точку.

Функция отображения не изменяется. Нам все так же нужно передать шейдеру значение переменной `TimeFactor`, на основе которой будет рассчитан цвет вершин, и матрицу, объединяющую в себе матрицы проекции, просмотра и мировую.

Запустите пример и убедитесь, что куб стал более реалистичным и теперь видно, что он трехмерный. Даже на рисунке это можно заметить (рис. 5.17).

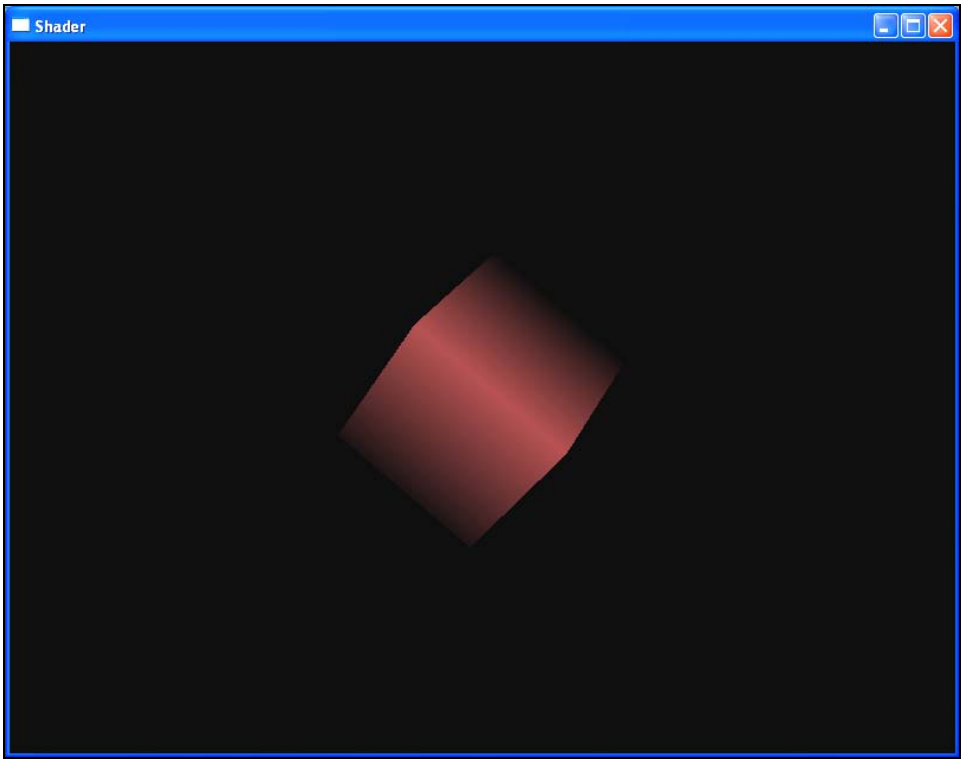

Рис. 5.17. Куб, в котором освещение, поворот и цвет задаются шейдером

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Shader1.

5.19. Невесомая капля

Может показаться, что вершинный буфер помогает только упростить разработку. Действительно, мы можем изменять освещение и поведение объекта в файле шейдера, не затрагивая исходного кода самой программы. Это хорошо, но пока не видно мощь, из-за которой можно было бы перейти на шейдер и увидеть всю его красоту.

Сейчас мы рассмотрим пример создания невесомой капли. Если поместить каплю жидкости в невесомость, то она будет парить в безвоздушном пространстве и деформироваться, создавая интересные, плавные формы. Над алгоритмом этого примера без использования Direct3D бились многие программисты, оптимизируя его и создавая различные варианты.

Если попытаться создать эффект невесомой капли под MS-DOS или Windows с использованием только DirectDraw, то вы ощутите все прелести программирования демо-роликов, потому что это тяжелый, но интересный алгоритм (если рисовать по законам физики с учетом освещения). Сейчас же мы увидим, как просто эта задача решается с использованием Direct3D и шейдеров.

Для начала рассмотрим теорию создания капли, чтобы вы понимали, что и для чего мы делаем. Деформировать мы будем сферу, которую загрузим из X-файла. Как будет происходить деформация? Посмотрите на рис. 5.18, где показана сфера, созданная из сетки треугольников. Точнее сказать, это была сфера, пока ее не вытянули в правую сторону. Для этого все вершины, расположенные справа от центра, были сдвинуты на несколько пикселов. Это всего лишь скриншот из 3D-редактора, а наша задача сделать что-то подобное. Сдвиг будет происходить по формуле: новое значение Y-координаты будет равно синусу от текущей позиции Y плюс коэффициент. В качестве коэффициента может быть значение времени или просто на каждом этапе увеличиваемое число.

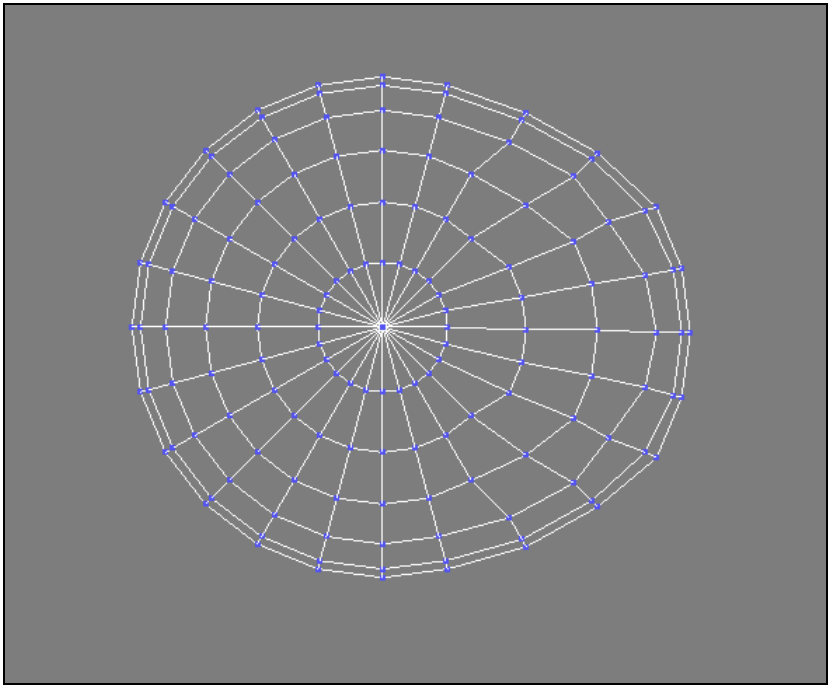

Рис. 5.18. Деформация сферы

Итак, в листинге 5.35 показан код шейдера, который деформирует каплю по синусоиде.

Листинг 5.35. Деформирование капли

```
// Структура для связи входных и выходных данных
struct VS_OUTPUT
{
    float4 pos  : POSITION;
    float4 col : COLOR0;
};

// Определяем освещение
float3 Light = {0.0f, 0.0f, -3.0f};

// Глобальные переменные
float4x4 mat : WORLDVIEWPROJECTION;
float TimeFactor = 1.0f;

// Функция вершинного шейдера
VS_OUTPUT ShaderFunc(
 float4 Pos : POSITION,
 float3 Normal : NORMAL)
{
    VS_OUTPUT Out = (VS_OUTPUT)0;

    // Задаем цвет вершины
    float4 normalmap = mul(Normal, mat);
    Out.col.r = sin(TimeFactor/10);
    Out.col.g = 1-Out.col.r;
    Out.col.b = 1-Out.col.r;
    Out.col *= dot(normalmap, Light);

    // Задаем позицию вершины
    Pos.y += sin(Pos+TimeFactor);
    Out.pos = mul(Pos, mat);

    return Out;
}

// Функция трансформации
technique Transform
{
    pass P0
    {
        VertexShader = compile vs_1_1 ShaderFunc();
    }
}
```

В принципе, большая часть кода вам уже должна быть знакома по *разд. 5.18*. Разница только в том, как мы определяем позицию вершины:

```
Pos.y += sin(Pos+TimeFactor);
Out.pos = mul(Pos, mat);
```

Здесь мы присваиваем позиции Y результат вычисления синуса от суммы текущей позиции и переменной `TimeFactor`. Поскольку позиции двух рядом стоящих вершин отличаются не сильно, то и результат вычисления синуса будет близким по значению. Таким образом, мы получим плавное вытягивание.

Но не станем спешить использовать этот шейдер. Давайте его усложним и добавим еще одну шейдерную функцию с именем `ShaderFuncColored`:

```
VS_OUTPUT ShaderFuncColored(
 float4 Pos : POSITION, float3 Normal : NORMAL)
{
 VS_OUTPUT Out = (VS_OUTPUT)0;

 // Расчет цвета
 float4 normalmap = mul(Normal, mat);
 Out.col = sin(Pos);
 Out.col *= dot(normalmap, Light);

 // Расчет позиции
 Pos.y += sin(Pos+TimeFactor);
 Out.pos = mul(Pos, mat);

 return Out;
}
```

В этой функции мы присваиваем цвету точки не просто одно и то же значение, а синус от текущей позиции. Таким образом, на поверхности капли цвет должен плавно изменяться (по синусоиде), потому что позиция каждой вершины отличается, и будет отличаться результат функции `sin`.

Для того чтобы установить эту функцию, напишем отдельную функцию `technique` с именем `TransformColored`:

```
technique TransformColored
{
    pass P0
    {
        VertexShader = compile vs_1_1 ShaderFuncColored();
    }
}
```

Теперь у нас в шейдере есть две функции technique, каждая устанавливает свой вариант шейдерной функции. Посмотрим, как их использовать. Создайте новый проект и добавьте инициализацию Direct3D. Код инициализации будет выглядеть следующим образом:

```
// Загрузка сетки
dwNumMaterials = LoadMesh("nlo.x", pD3DDevice,  &pMesh,
     &pMeshTextures, "texture.bmp", &pMeshMaterials);

// Загружаем шейдер
D3DXCreateEffectFromFile(pD3DDevice, "simple.vsh", 0, 0, 0, 0,
        &pEffect, 0);
pEffect->SetTechnique("Transform");

// Описываем вершину
D3DVERTEXELEMENT9 decl[]=
{
   {0,0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                       D3DDECLUSAGE_POSITION,0},
   {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
                       D3DDECLUSAGE_NORMAL, 0},
   D3DDECL_END()
};
pD3DDevice->CreateVertexDeclaration(decl, &VertDecl);

// Запоминаем матрицу проекции
float Aspect = (float)iWidth / (float)iHeight;
D3DXMatrixPerspectiveFovLH(&matProjection, D3DX_PI/4.0f, Aspect,
      0.1f, 1000.0f);
```

В самом начале происходит загрузка сетки из файла nlo.x, в котором должна быть сфера. Могут быть и другие фигуры, но желательно, чтобы они были гладкими и хоть немного по форме напоминали сферу, иначе эффект может не получиться.

Затем загружаем шейдер. В качестве функции technique на начальном этапе устанавливаем функцию Transform. Затем идет описание вершины, которая описывается положением и нормалью. И напоследок — запоминаем матрицу проекции.

Теперь переходим к отображению. Код будет идентичен тому, что мы написали в *разд. 5.18*, за исключением двух нюансов. В самом начале функции DrawScene у нас будет увеличиваться переменная ViewAngle на 0.5, и если значение превысит 10, то в качестве функции шейдера устанавливаем функцию TransformColored:

```
ViewAngle+=0.5f;
if (ViewAngle>10.0f)
  pEffect->SetTechnique("TransformColored");
```

Как видите, во время выполнения отображения мы легко можем поменять функцию шейдера, если у нас в файле их несколько.

Второй нюанс — цикл отображения сетки с помощью шейдера. Раньше мы использовали функцию `DrawIndexedPrimitive`, а теперь для отображения сетки необходимо применить метод `DrawSubset`:

```
UINT uPass;
pEffect->Begin(&uPass, 0);
for (int i=0; i<uPass; i++)
{
 pEffect->Pass(i);
 for (DWORD i=0; i<dwNumMaterials; i++)
  pMesh->DrawSubset(i);
}
pEffect->End();
```

Попробуйте запустить пример. Первые несколько десятков секунд капля будет деформироваться, но цвет всей капли останется однотонным, если не считать эффекта затенения некоторых частей капли за счет освещения. После этого произойдет переключение на функцию шейдера `TransformColored`, и цвет каждой вершины будет определяться по синусоиде, что сделает пример более интересным.

Для самостоятельной работы попробуйте изменять в шейдере координату X, а не Y. Результат станет не очень похож на каплю в невесомости, но будет достаточно интересным. Можете попробовать загружать не сферу, а тор или что-нибудь еще. С помощью объектов разной формы (главное, чтобы поверхность была схожа со сферической, и может быть хоть чайником) возможно создание очень интересных эффектов.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Blob.

5.20. Пиксельный шейдер

В предыдущих примерах мы стремились направлять свет прямо на объект. Я старался в примерах как можно реже направлять свет сбоку или под углом, особенно на сглаженные поверхности, например, сферы. Почему? Посмотри-

те на рис. 5.19, где сфера освещается справа. Поверхность стала неравномерной. Чтобы лучше ощутить проблему, запустите пример из каталога Sample/BadLight на прилагаемом компакт-диске.

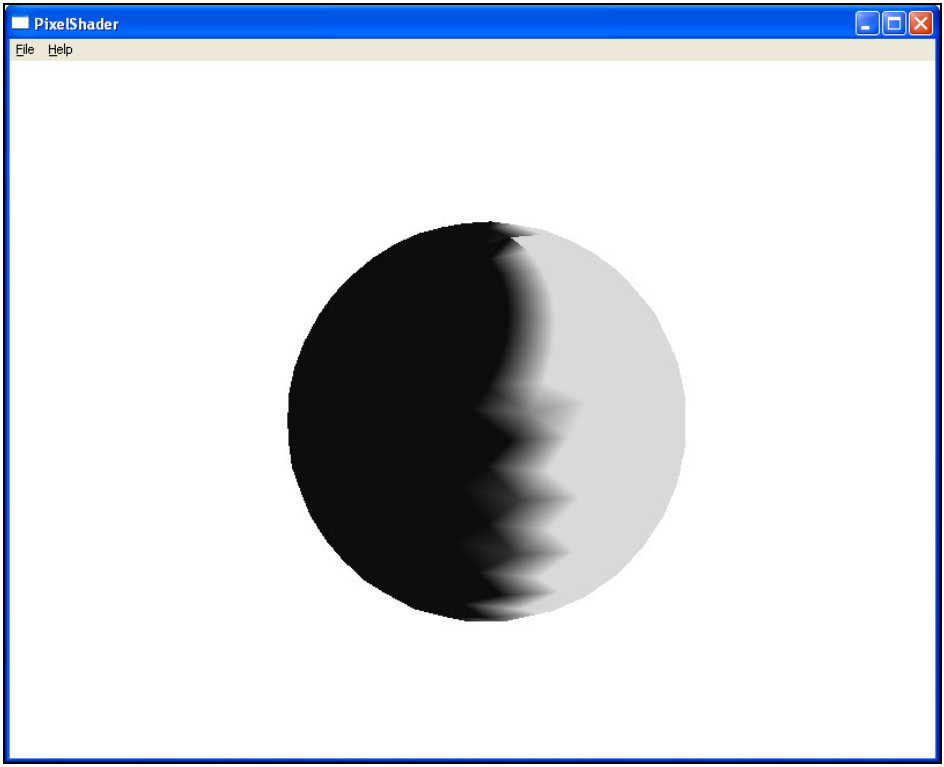

Рис. 5.19. Освещение сферы сбоку

Почему цвет сферы получился таким резанным на границе освещенной и неосвещенной поверхности сферы? Чтобы увидеть это, посмотрим на рис. 5.20. Здесь я создал сферу в виде сетки в 3D-редакторе, а потом с помощью простого растрового редактора попытался закрасить правую часть. При этом жирная вертикальная линия показывает границу освещения. При вершинном отображении сцены движок Direct3D оперирует вершинами и гранями, поэтому невозможно сделать плавный переход.

Проблему можно решить двумя способами:

1. Увеличить количество вершин, чтобы сфера создавалась из большего количества треугольников. В этом случае треугольники будут достаточно маленькими, и прерывистость края будет не такой заметной. Но проблема не решается полностью, к тому же увеличение вершин приводит к тому,

что заметно падает производительность. Каждая вершина — это проблема для видеопроцессора.

2. Использовать пиксельный шейдер. Необходимо установить функцию пиксельного шейдера, и она будет вызываться видеокартой перед отображением каждой очередной точки. Наша задача в этой функции определить цвет каждой вершины.

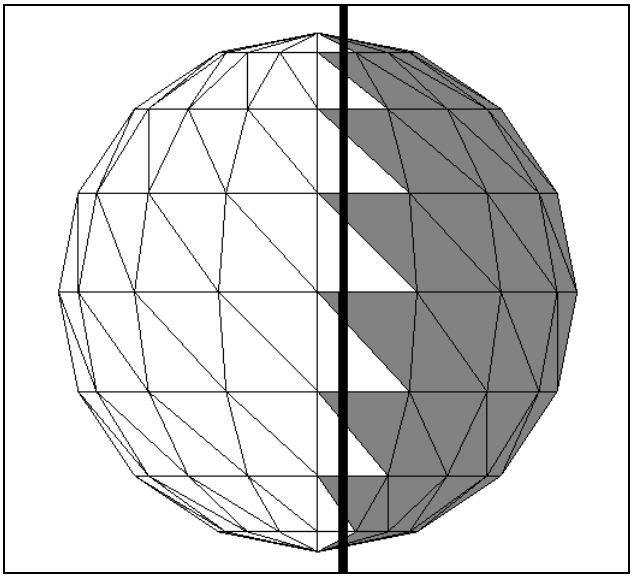

Рис. 5.20. Закрашивание поверхности сферы по определенной границе

Первое решение проблемы вы можете реализовать сами, но я бы не стал этого делать, если есть пиксельный шейдер, который все сделает намного лучше. Это только на первый взгляд кажется, что определение цвета слишком сложная задача, вот сейчас вы увидите, что проблема решается достаточно легко, быстро и непринужденно.

Сначала рассмотрим код шейдера, а потом поговорим о том, как создать пример, использующий его. Код шейдера показан в листинге 5.36.

Листинг 5.36. Код шейдера создания качественного освещения

```
// Глобальные переменные, которые будут задаваться из программы
float4x4 mat : WORLDVIEWPROJECTION;
float4x4 worldmat : WORLD;
float4 light;
float4 facecolor;
```

```
// Описание структуры
struct VS_OUTPUT
{
    float4 Position : POSITION;
    float3 Normal : TEXCOORD1;
    float3 LightDir : TEXCOORD0;
};

// Функция вершинного шейдера
VS_OUTPUT Vexel_Sh(
    float4 pos : POSITION,
    float3 nor : NORMAL
    )
{
 VS_OUTPUT Out = (VS_OUTPUT)0;

 // Определяем позицию
 Out.Position = mul(pos, mat);
 // Рассчитываем нормаль
 Out.Normal = normalize(mul(nor, worldmat));
 // Задаем освещение
 Out.LightDir = light;
// Возвращаем результат расчетов
 return Out;
}

float4 Pixel_Sh(
    float3 normal : TEXCOORD1,
    float3 lightdir : TEXCOORD0
  ) : COLOR0
{
 // Определяем нормаль для освещения
 float3 lightn = normalize(lightdir);
 // Яркость точки равна точки пересечения источника
 // света и нормали света
 float4 diffuse = saturate(dot(lightn, normalized));
 // Результирующий цвет пиксела равен произведению
 // цвета точки на коэффициент
 return facecolor * diffuse;
}

technique PixelLight
{
    pass P0
```

```
    {
        VertexShader = compile vs_1_1 Vexel_Sh();
        PixelShader  = compile ps_2_0 Pixel_Sh();
    }
}
```

Это самый простейший алгоритм расчета освещения через пиксельный шейдер. Можно было бы реализовать что-то более сложное, но в учебных целях я не стал усложнять жизнь.

Начнем рассмотрение кода с самого конца, а именно с функции `PixelLight`. Здесь задается не только вершинный шейдер (его роль будет выполнять функция `Vexel_Sh`), но и пиксельный (это будет функция `PixelLight`). Обратите внимание, что для пиксельного шейдера используется версия 2.0. Это важно, потому что версия 1.1 не сможет откомпилировать этот код и наверняка вернет ошибку. Для вершинного шейдера мы ничего сложного не используем, поэтому будет достаточно версии 1.1.

Теперь посмотрим на структуру, которая должна описывать выходные данные для вершинного шейдера:

```
struct VS_OUTPUT
{
    float4 Position : POSITION;
    float3 Normal : TEXCOORD1;
    float3 LightDir : TEXCOORD0;
};
```

Здесь имеется три параметра — позиция вершины, нормаль и позиция освещения. При этом только первые два параметра будут передаваться в вершинный буфер, а третий параметр — будет задаваться. В него мы будем записывать положение источника света. Зачем? Это на практике покажет нам, что при использовании вершинного и пиксельного шейдера одновременно эта структура выполняет роль средства передачи параметров из вершинного в пиксельный шейдер. Положение источника света не изменяется для всей сцены, ведь он находится в одном и том же месте. Но для каждой вершины мы будем указывать его, чтобы передать это значение в пиксельный шейдер. В реальной программе можно было бы обойтись и без этой переменной.

На уровне вершин мы всего лишь подготавливаем переменные. В качестве входящих данных получаем позицию вершины и ее нормаль (в нашем случае это параметры функции `Vexel_Sh`). В самом начале функции мы объявляем переменную `Out` типа `VS_OUTPUT` и заполняем ее значениями:

```
VS_OUTPUT Out = (VS_OUTPUT)0;
```

Теперь заполняем параметры структуры:

```
Out.Position = mul(pos, mat);
Out.Normal = normalize(mul(nor, worldmat));
Out.LightDir = light;
```

В первой строке определяем позицию вершины, с учетом матрицы положения объекта в пространстве. Для этого перемножаем позицию вершины и переменную `mat`, где у нас хранится объединенная матрица, объединяющая матрицу проекции, мировую и просмотра.

Затем нормализуем нормаль вершины с мировой матрицей. На самом деле, нам придется передавать в шейдер не только объединенную матрицу `mat`, но и отдельно мировую матрицу.

В последней строке заполняется поле `LightDir`, куда записываем положение источника освещения, который передается в шейдер через переменную `light`.

Теперь посмотрим, как будет выполняться формирование сцены. Сначала программа запрашивает вывод определенного объекта. Для каждой вершины объекта вызывается функция вершинного шейдера, где у нас формируется позиция вершины с учетом матриц, и заполняются параметры `Normal` и `LightDir`, которые пока никак не влияют на результирующую сцену.

Когда все вершины просчитаны и положение объекта определено, прежде чем вывести каждый пиксел результирующего объекта, вызывается функция пиксельного шейдера. Ей передаются параметры — нормаль и положение источника света, т. е. второй и третий параметр структуры `VS_OUTPUT`.

В качестве результата функция пиксельного шейдера возвращает цвет пиксела. Об этом говорит ключевое слово `COLOR0`, после объявления функции:

```
float4 Pixel_Sh(
    float3 normal : TEXCOORD1,
    float3 lightdir : TEXCOORD0
  ) : COLOR0
```

Теперь смотрим код функции:

```
float3 lightn = normalize(lightdir);
float4 diffuse = saturate(dot(lightn, normal));

return facecolor * diffuse;
```

Можно в этом коде вместо переменной `lightdir` использовать напрямую глобальную переменную `light`, и в этом случае можно убирать соответствующий параметр у функции и поле `LightDir` у структуры `VS_OUTPUT`, потому что значение этого поля и переменной `light` одинаковы.

Код расчета цвета пиксела прост. Необходимо определить нормаль для источника освещения. Затем выясняется точка нормали света и нормали, полученной в качестве параметра функции (вспомните, что там находится). Теперь насыщенность пиксела можно определить с помощью функции `saturate`. Результат выполнения этой функции записывается в переменную `diffuse`.

Теперь для определения цвета пиксела достаточно перемножить цвет объекта на насыщенность из переменной `diffuse`. Цвет объекта неизменен и находится в глобальной переменной `facecolor`.

Затем посмотрим на код использования этого шейдера. Для реализации примера нам понадобится загрузить в Mesh-сферу из X-файла и загрузить шейдер в интерфейс `ID3DXEffect`. При описании вершины необходимо создать две структуры, которые будут содержать информацию о позиции и нормали, потому что именно эти данные загружаются вместе с сеткой из X-файла:

```
D3DVERTEXELEMENT9 decl[]=
{
 {0,0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
          D3DDECLUSAGE_POSITION, 0},
 {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
          D3DDECLUSAGE_NORMAL, 0},
 D3DDECL_END()
};
```

Все это вы сможете сделать сами. В принципе, код отображения тоже уже можете написать, но я его все же приведу (листинг 5.37).

Листинг 5.37. Код отображения сферы с пиксельным освещением

```
void DrawScene()
{
 // Увеличиваем значение угла поворота
 ViewAngle+=ViewAngleOffset;
 if (ViewAngle>10)
  ViewAngleOffset=-0.05f;
 if (ViewAngle<0)
  ViewAngleOffset=0.05f;

 // Рассчитываем матрицу поворота
 float b=ViewAngle;
 D3DMATRIX World = {
  cos(b)*cos(b), cos(b)*sin(b), sin(b), 0,
```

```
  -sin(b), cos(b), 0, 0,
  -sin(b)*cos(b), -sin(b)*sin(b), cos(b), 0,
  0, 0, 0, 1,
 };

 // Объединяем матрицы: просмотра, мировую и проекции
 D3DXMATRIX Full;
 D3DXMatrixMultiply(&Full, (D3DXMATRIX*)&World, (D3DXMATRIX*)&View);
 D3DXMatrixMultiply(&Full, &Full, &matProjection);

 // Задаем значения переменных шейдера
 pEffect->SetValue("mat", &Full, D3DX_DEFAULT);
 pEffect->SetValue("worldmat", &World, D3DX_DEFAULT);
 pEffect->SetValue("light", new D3DXVECTOR4(ViewAngle*2, 0, -5, 1),
      D3DX_DEFAULT);
 pEffect->SetValue("facecolor", new D3DXVECTOR4(0.5f, 0.8f, 0.5f, 1.0f),
      D3DX_DEFAULT);

 // Задаем тип вершины
 pD3DDevice->SetVertexDeclaration(VertDecl);

 // Отображаем объект
 UINT uPass;
 pEffect->Begin(&uPass, 0);
 for (int i=0; i<uPass; i++)
 {
  pEffect->Pass(i);
  for (DWORD i=0; i<dwNumMaterials; i++)
   pMesh->DrawSubset(i);
 }
 pEffect->End();
}
```

Во время задания переменных шейдера нам необходимо передать объединенную матрицу, мировую матрицу, положение источника света и цвет поверхности. Цвет поверхности и положение источника освещения имеют тип `D3DXVECTOR4`. Посмотрите, как оригинально можно передать эти значения без введения отдельных переменных. Мы сразу же инициализируем и передаем значение вектора.

Обратите также внимание, что координата X у вектора положения источника освещения равна значению переменной `ViewAngle`, умноженной на 2. Это значит, что источник будет двигаться по горизонтали одновременно с вращением сферы, освещая объект с разных сторон.

Запустите пример, и убедитесь, что освещение стало плавным и более естественным (рис. 5.21).

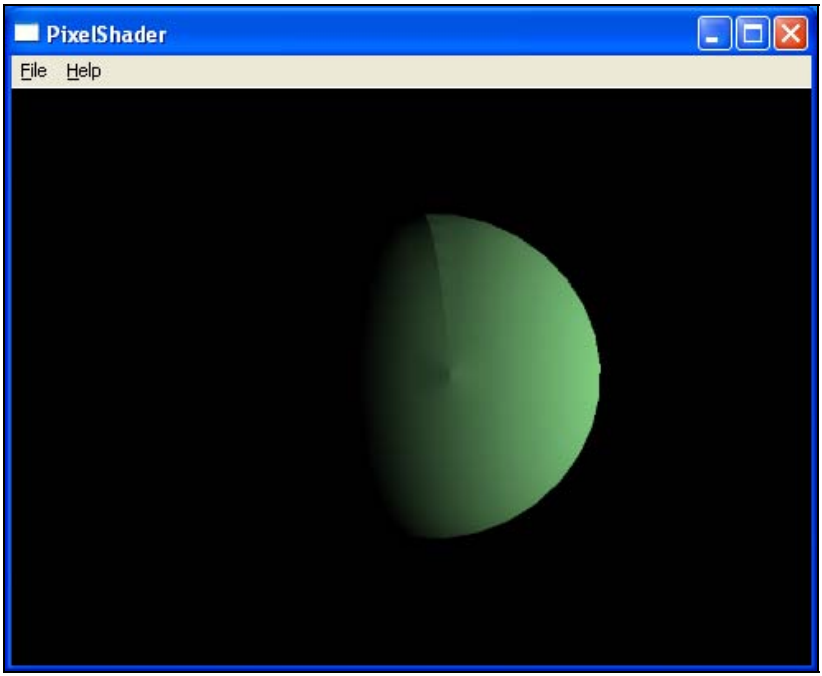

Рис. 5.21. Освещение сферы с помощью пиксельного шейдера

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\PixelShader.

5.21. Блики

Освещение, которое мы получили в предыдущем разделе, вышло плавным и более естественным, но пока не идеальным. Дело в том, что далеко не все материалы выглядят так тускло, некоторые могут давать блики. Например, полированный металл. У вас есть люстра с металлическими частями? Если да, то на ней металл скорей всего будет с бликами на ярком солнце, т. е. цвет будет не однотонным, а со стороны источника света имеются более светлые участки.

Для программирования бликов нужно немного напрячься и добавить несколько строчек кода в шейдер, который мы уже написали в *разд. 5.20*. Код

самой программы не изменится, только VSH-файл. Этот код показан в листинге 5.38.

Листинг 5.38. Шейдер с бликами

```
float4x4 mat : WORLDVIEWPROJECTION;
float4x4 worldmat : WORLD;
float4 light;
float4 facecolor;
const float specfactor = 16;

struct VS_OUTPUT
{
    float4 Position : POSITION;
    float3 Normal : TEXCOORD1;
    float3 LightDir : TEXCOORD0;
    float3 eyepos : TEXCOORD2;
};

VS_OUTPUT Vexel_Sh(
    float4 pos : POSITION,
    float3 nor : NORMAL
    )
{
    VS_OUTPUT Out = (VS_OUTPUT)0;

    Out.Position = mul(pos, mat);

    Out.LightDir = light;

    Out.Normal = normalize(mul(nor, worldmat));

    float3 worldpos = normalize(mul(pos, worldmat));
    Out.eyepos = light - worldpos;

    return Out;
}

float4 Pixel_Sh(
    float3 normal : TEXCOORD1,
    float3 lightdir : TEXCOORD0,
    float3 eye : TEXCOORD2
  ) : COLOR0
```

```
{
    float3 normalized = normalize(normal);
    float3 lightn = normalize(lightdir);
    float4 diffuse = saturate(dot(light, normalized));

    float3 reflection = normalize(2 * diffuse * normalized - lightn);
    float3 eyen = normalize(eye);
    float4 specular = pow(saturate(dot(reflection, eyen)), specfactor);

    return facecolor * diffuse + specular;
}

technique PixelLight
{
    pass P0
    {
        VertexShader = compile vs_1_1 Vexel_Sh();
        PixelShader  = compile ps_2_0 Pixel_Sh();
    }
}
```

Результат работы программы представлен на рис. 5.22. Обратите внимание, что с правой стороны на сфере видно белое пятно блика, чего не было в примере из *разд. 5.20*.

Код примера немного нарушает законы физики и это происходит в функции вершинного шейдера в следующей строке:

```
Out.eyepos = light - worldpos;
```

Переменная `eyepos` не зря названа именно так. В переводе это означает *позиция глаза* (eye position). А мы вычисляем позицию источника света. Таким образом, блик будет всегда с той стороны, где светит источник освещения, а он должен быть в точке пересечения вектора направления вашего взгляда и вектора направления освещения.

С остальным кодом попробуйте разобраться самостоятельно. Он отличается не сильно, добавлено всего несколько строчек и одна переменная `eyepos`, которую мы уже немного затронули.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\PixelShader.

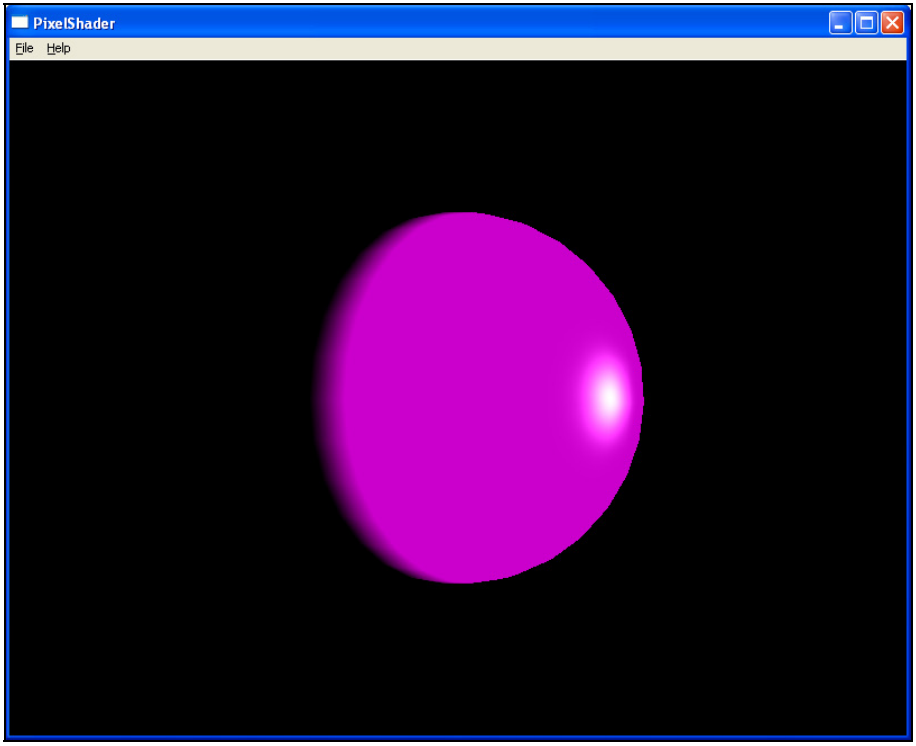

Рис. 5.22. Отображение освещения с бликами

5.22. Сердечный приступ

Теперь мы начнем создавать уже более интересные эффекты, и первым на очереди будет график кардиограммы. Результат, который мы постараемся достигнуть, показан на рис. 5.23.

Если сгладить углы и уровнять высоту каждого всплеска, то получится синусоида. На самом деле, алгоритм будет построен именно на основе тригонометрии, нам же необходимо будет только заострить углы и сделать высоту всплесков разными. К тому же, не помешает немного размыть линии, чтобы они выглядели более естественными.

Листинг 5.39. Код создания изображения кардиограммы

```
float4x4 view_proj_matrix;
float4 color;
float ViewAngle;
```

```
const float baseamp=0.10;
const float speed=3.0;
const float rate=15.0;

struct VS_OUTPUT {
  float4 Pos: POSITION;
  float2 xyvec: TEXCOORD;
};

// Vertex Shader
VS_OUTPUT vertsh(float4 Pos: POSITION)
{
   VS_OUTPUT Out;

   Pos.xy = sign(Pos.xy);

   Out.Pos = float4(Pos.xy, 0, 1);
   Out.xyvec = Pos.xy;

   return Out;
}

// Pixel Shader
float4 pixsh(float2 pos: TEXCOORD0) : COLOR
{
   float x = speed * ViewAngle + rate * pos.x;

   float base = (1 + cos(x*2.5)) * (1 + sin(x*3.5));

   float z = frac(0.05 * x);
   z = max(z, 1 - z);
   z = pow(z, 20);
   float pulse = exp(-10000 * z);

   return pow(saturate(1 - abs(pos.y - (baseamp * base + pulse - 0.5))),
       16) * color;
}


technique PixelLight
{
    pass P0
    {
        VertexShader = compile vs_2_0 vertsh();
        PixelShader  = compile ps_2_0 pixsh();
    }
}
```

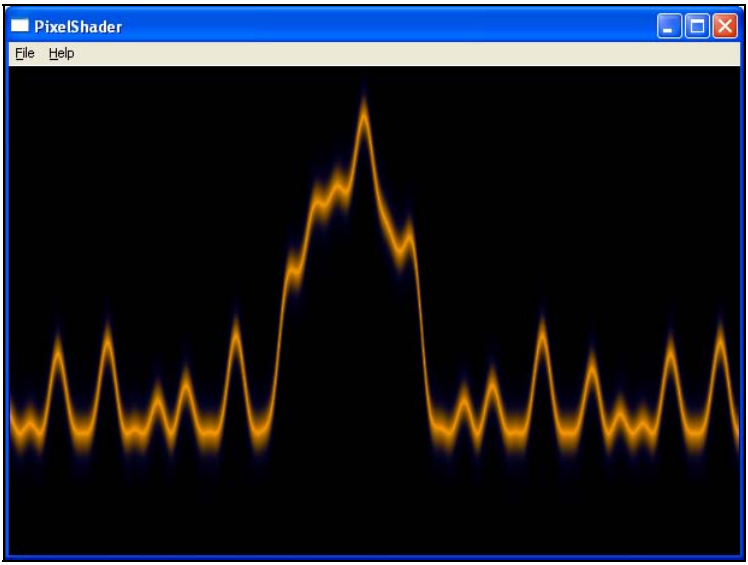

Рис. 5.23. Отображение графика кардиограммы

Для того чтобы выполнить этот шейдер, нам необходимо задать три переменные:

- `view_proj_matrix` — объединенная матрица;
- `color` — цвет линии;
- `ViewAngle` — от этой переменной зависит смещение графика, т. е., благодаря этой переменной, будет происходить движение справа налево.

После этого достаточно вывести на экран любую фигуру. Какую и как не имеет значения, потому что шейдер кардиограммы заполнит всю поверхность экрана.

Теперь давайте немного рассмотрим, что делают наши шейдеры. Тут есть несколько новых функций, которые мы еще не рассматривали. Начнем с вершинного шейдера, который получает на входе позицию вершины.

```
VS_OUTPUT vertsh(float4 Pos: POSITION)
```

Как всегда, в самом начале объявляем переменную типа `VS_OUTPUT`, для связи входящих и исходящих данных. После этого в компоненты `xy` позиции, которую мы получили в качестве параметра, записываем результат выполнения функции `sign`. Эта функция возвращает одно из следующих значений:

- 1 — если переданное значение больше нуля;
- 0 — если переданное значение равно нулю;
- –1 — если переданное значение меньше нуля.

На основе результата задается новая позиция (поле `Pos` выходной структуры) и вектор направления вершины (поле `xyvec`).

Остальное — сплошная математика в функции `pixsh`. Чтобы вы смогли разобраться с этой математикой, рассмотрим новые функции языка HLSL, которые мы использовали:

- первая функция — `pow`. Ей передается два параметра `X` и `Y`, а результатом будет число `X` в степени `Y`;
- функция `exp` возвращает экспоненту переданного в качестве параметра числа;
- следующая функция `saturate` — помещает указанное в качестве параметра число в пределах от 0 до 1;
- cos — возвращает косинус указанного числа;
- sin — возвращает синус указанного числа.

Для удобства, в самом начале шейдера заведено три константы:

```
const float baseamp=0.10; // Базовая амплитуда
const float speed=3.0;    // Скорость движения
const float rate=15.0;    // Частота
```

Изменяя эти константы, вы можете изменить внешний вид кардиограммы. Можно сделать возможность изменения их из программы (например, плавно увеличивать значение `rate`). Попробуйте сами поиграть с константами и посмотреть на результат. Это поможет лучше понять алгоритм.

Чтобы сердечный приступ рисовался на текстуре сферы, которую мы отображаем на экране, а не на весь экран, необходимо модифицировать вершинный шейдер следующим образом:

```
VS_OUTPUT Vexel_Sh(float4 Pos: POSITION)
{
   VS_OUTPUT Out;

   // Связываем координаты вершины с матрицей проекции
   Out.Pos = mul(Pos, mat);

   // Вектор для расчета сердечного приступа
   Out.xyvec = Pos.xy;

   return Out;
}
```

Отличие этого шейдера в том, что координаты вершины объединяются с суммарной матрицей положения. Таким образом, мы позиционируем нашу сцену в отображаемом шейдером объекте.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\CoolShader1.

5.23. Огненный дракон

Я заметил, что многие программисты, работающие с графикой, стремятся научиться создавать огонь. Не знаю точно, откуда это стремление, но мне кажется, что это связано со стремлением создавать игры и приближенные к реальности вспышки огня или взрывы. В демо-роликах эффект огня тоже используется очень часто. Каждый стремится создать наиболее реалистичный факел. Давайте рассмотрим несколько алгоритмов, которые помогут вам.

5.23.1. Костер

Начнем рассмотрение огня с костра, как наиболее часто используемого в роликах. В листинге 5.40 показан пример алгоритма огня.

Листинг 5.40. Пример шейдера создания костра

```
float4x4 mat : WORLDVIEWPROJECTION;
float4x4 worldmat : WORLD;
float ViewAngle;
const float sideFade = 30.0f;
const float WidthFactor = 0.05f;
const float xWind = 0.05f;

struct VS_OUTPUT
{
   float4 Pos: POSITION;
   float2 tex_coord: TEXCOORD0;
};

VS_OUTPUT Vexel_Sh(float4 Pos: POSITION)
{
  VS_OUTPUT Out;

  Out.Pos = mul(Pos, mat);
  Out.tex_coord = Pos.xy;

  return Out;
}
```

```
Texture meshTexture;
Texture noiceTexture;

sampler Flame = sampler_state {texture = <meshTexture>;
       mipfilter = LINEAR; minfilter = LINEAR; magfilter = LINEAR;
       addressu=clamp; addressv=wrap;};
sampler Noise = sampler_state {texture = <noiceTexture>;
       mipfilter = LINEAR; minfilter = LINEAR; magfilter = LINEAR;
       addressu=wrap; addressv=wrap;};

float4 Pixel_Sh(float2 tex_coord: TEXCOORD0) : COLOR
{
  float3 coord;

  coord.x = tex_coord.x;
  coord.y = tex_coord.y - ViewAngle;
  coord.z = 1;

  float noisy = tex2D(Noise, coord);

  float shape = WidthFactor * (0.5 -
              sideFade * tex_coord.x * tex_coord.x);

  float heat = saturate(shape + noisy - tex_coord.y);

  float4 flame = tex2D(Flame, heat);

  return flame;
}

technique PixelLight
{
  pass P0
  {
    VertexShader = compile vs_2_0 Vexel_Sh();
    PixelShader  = compile ps_2_0 Pixel_Sh();
  }
}
```

В функции `PixelLight` нет никаких изменений, по сравнению с предыдущим примером. Все также устанавливаем пиксельный и вершинный шейдер и для компиляции обоих выбираем вторую версию.

В самом начале заведено три переменные, две для хранения матрицы (суммарной и вида) и одна переменная хранит счетчик `ViewAngle`, благодаря которому будет происходить анимация сцены, т. е. огонь будет гореть.

Далее идут три константы:

```
float sideFade = 30.0f;
float WidthFactor = 0.05f;
float xWind = 0.05f;
```

Первая определяет затухание огня (высоту), вторая ширину пламени, а последняя (`xWind`) — ветер. Если он положительный, то огонь наклонен влево, иначе вправо.

Структура, которая связывает входящие и исходящие данные, содержит позицию вершины и текстурные координаты:

```
float4 Pos: POSITION;
float2 tex_coord: TEXCOORD0;
```

Огонь будет окрашиваться на основе двух текстур. Первая текстура содержит градиент перехода от черного к белому через красный и желтый. Если посмотреть на огонь, то возле источника горения, где самая высокая температура, цвет огня белый. С отдалением от источника пламени цвет желтеет, а если отодвинуться еще дальше, то краснеет. Именно поэтому наша текстура имеет такой градиент, и она будет храниться в файле Flame.bmp и загружаться в основной программе. В шейдере необходимо только объявить переменную для данной текстуры и описать параметры ее отображения, что и происходит в следующих строках:

```
Texture meshTexture;

sampler Flame = sampler_state {
   texture = <meshTexture>;
   mipfilter = LINEAR;
   minfilter = LINEAR;
   magfilter = LINEAR;
   addressu=clamp;
   addressv=wrap;
};
```

Вторая текстура может содержать какой-либо узор или просто изображение шума[1]. На ее основе градиент будет превращаться в огонь. В качестве этой

[1] Хаотичные точки, разбросанные по картинке. Их легко создать в Photoshop с помощью фильтра Noise (Шум).

текстуры я выбрал файл LobbyCube.dds из состава DirectX SDK. Если честно, то я даже не знаю, что находится в этом файле ☺. Сначала я выбрал текстуру дерева, которую мы использовали в *разд. 5.13*, но огонь получился слегка неестественным, потому что языки пламени стали в виде веток дерева, а с текстурой LobbyCube.dds языки выглядят более натурально. Идеальный вариант, если текстура будет содержать изображения шума, но у меня не оказалось под рукой ничего подходящего, а рисую я плохо.

Эта текстура в шейдере описана следующим образом:

```
Texture noiceTexture;
sampler Noise = sampler_state {
   texture = <noiceTexture>;
   mipfilter = LINEAR;
   minfilter = LINEAR;
   magfilter = LINEAR;
   addressu=wrap;
   addressv=wrap;
};
```

Обратите внимание на параметры `addressu` и `addressv`. Эти значения должны быть именно такими, иначе текстуры будут наложены неверно.

Теперь посмотрим на пиксельный шейдер. В самом начале мы заводим вектор из трех полей с именем `coord`:

```
float3 coord;
```

Теперь заполняем параметры этого вектора:

```
coord.x = tex_coord.x;
coord.y = tex_coord.y - ViewAngle;
coord.z = 1;
```

Координата X вектора `coord` будет соответствовать координате X пиксела. А вот значение Y нужно немного скорректировать на значение переменной `ViewAngle`, чтобы происходил эффект движения текстуры снизу вверх, т. е. эффект горения. Координата Z нами не будет использоваться, потому что костер двумерный.

Теперь определим точку на текстуре `Noise`, которая находится в координате `coord`:

```
float noisy = tex2D(Noise, coord);
```

Затем рассчитываем форму костра с помощью следующей строки кода:

```
float shape = WidthFactor *
     (0.5 - sideFade * tex_coord.x * tex_coord.x);
```

Таким образом, мы получили общую форму, но это еще не все. Из этой формы нужно определить область, которая будет более "горячей" (находится ближе к источнику горения):

```
float heat = saturate(shape + noisy - tex_coord.y);
```

Теперь достаточно только определить цвет вершины по градиенту из текстуры `Flame`:

```
float4 flame = tex2D(Flame, heat);
```

Сейчас посмотрим, как можно использовать этот код. Для начала, необходимо загрузить объект, внутри которого будет гореть огонь, точнее сказать, огонь будет рисоваться на поверхности текстуры этого объекта. Не будем долго выдумывать, а возьмем сферу, которую мы использовали в предыдущем проекте.

Помимо этого, нам понадобятся две переменные для хранения текстуры:

```
IDirect3DTexture9 *Texture;
IDirect3DTexture9 *TextureNoice;
```

Первая текстура будет содержать градиент огня, а вторая текстуру шума. Загрузим соответствующие текстуры на этапе инициализации следующим образом:

```
D3DXCreateTextureFromFile(pD3DDevice, "flame.bmp", &Texture);
D3DXCreateTextureFromFile(pD3DDevice, "LobbyCube.dds",
   &TextureNoice);
```

Загрузка шейдера должна выглядеть так:

```
D3DXCreateEffectFromFile(pD3DDevice, "simple.vsh",
    0, 0, 0, 0, &pEffect, 0);
pEffect->SetTechnique("PixelLight");
pEffect->SetValue("meshTexture", &Texture, D3DX_DEFAULT);
pEffect->SetValue("noiceTexture", &TextureNoice, D3DX_DEFAULT);
```

В первой строке мы загружаем шейдер из файла, а во второй выбираем функцию `Technique`, которая будет использоваться для вывода графики. После этого заполняем переменные `meshTexture` и `noiceTexture` в шейдере. Текстуры не изменяются, поэтому их можно заполнить уже на этапе загрузки и инициализации.

Далее, у вас должен идти код описания вершин. Он ничем не отличается от предыдущих примеров и включает в себя две структуры — описание позиции и нормали, хотя второе мы не используем, ведь у нас нет даже освещения:

```
D3DVERTEXELEMENT9 decl[]=
{
 {0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_POSITION, 0},
 {0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_NORMAL, 0},
 D3DDECL_END()
};

pD3DDevice->CreateVertexDeclaration(decl, &VertDecl);
```

Код отображения сцены идентичен коду из *разд. 5.21*. Вам не нужно выбирать текстуры перед отображением поверхности сферы, все сделает шейдер, которому мы уже указали, какие текстуры нужно использовать, а он "знает", как это делать.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\CoolShader2.

5.23.2. Ядро

Давайте немного модифицируем пример из *разд. 5.23.1* и сделаем не костер, а эффект ядра. Может быть, вы помните, что были такие пушки в средние века, которые стреляли ядрами. Только наше ядро будет нагреваться до высокой температуры, создавая эффект раскаленного металла.

Основа примера будет такая же, как и в *разд. 5.23.1*, с двумя только отличиями:

1. В качестве текстуры шума будет не LobbyCube.dds, как в прошлом разделе, а noise.dds, которая также входит в состав DirectX SDK. Чтобы не перекомпилировать пример, я просто переименовал noise.dds в LobbyCube.dds;
2. Изменится пиксельный шейдер, который показан в листинге 5.41.

Листинг 5.41. Пиксельный шейдер для создания эффекта раскаленного ядра

```
float4 Pixel_Sh(float2 tex_coord: TEXCOORD0) : COLOR
{
   float z = frac(ViewAngle);
   float i = pow(z, 0.5);

   float size = (i * (i-2) + 1) * i * 7;
   float dist = length(tex_coord) / size;
```

```
   float n = tex3D(Noise, float3(0.5 * tex_coord, 0.5 * dist));

   float4 flame = tex1D(Flame, size + size * (n - dist));

   return flame;
}
```

Остальной код программы изменять не нужно. Теперь запускайте пример и наслаждайтесь, но для еще большего эффекта можно добавить поворот сцены, чтобы ядро вращалось.

Для лучшего понимания, как реализуется алгоритм, запустите пример. Сфера обтянута текстурой LobbyCube.dds, а под этой текстурой рисуется круглый взрыв из текстуры Fire.bmp. Взрыв, просвечиваясь сквозь шум из LobbyCube.dds, создает эффект раскаленного ядра.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\CoolShader3.

5.23.3. Огненная лава

Огненная лава очень часто используется в играх и этот эффект является одним из вариантов огня. Нам также понадобится текстура огня и текстура шума. Код примера будет отличаться от примера из *разд. 5.23.2* только пиксельным шейдером. Код программы и вершинного шейдера будет таким же.

Пиксельный шейдер, создающий эффект огненной лавы, показан в листинге 5.42.

Листинг 5.42. Пиксельный шейдер создания эффекта лавы

```
float4 Pixel_Sh(float2 tex_coord: TEXCOORD0) : COLOR
{
 float3 coord;
 coord.xy = tex_coord;
 coord.z = 1;

 // Определяем координаты текстуры шума
 float noiseX = tex3D(Noise, coord);
 float noiseY = tex3D(Noise, coord);

 // Координаты текстуры огня
 coord.z = ViewAngle / 2;
 coord.x = tex_coord.x + noiseX * 0.1+coord.z;
 coord.y = tex_coord.y + noiseY * 0.1+coord.z;
```

```
 // Определяем цвет шума
 float n = tex3D(Noise, coord);

 // Объединяем цвет шума и цвет точки на текстуре огня
 float4 flame = tex1D(Flame, n-0.15);

 // Возвращаем цвет точки
 return flame;
}
```

То, что лава отображается на сфере, позволяет создать эффект раскаленной планеты или даже солнца. Цвет лавы можно регулировать в последней строке кода:

```
float4 flame = tex1D(Flame, n-0.15);
```

коэффициент 0.15 делает лаву более красной. Если уменьшить это значение или вообще убрать, то лава пожелтеет.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\CoolShader4.

5.24. Морфинг

Еще одна задача, которую очень часто решают демо-программисты — *морфинг* (анимация, в которой один объект плавно превращается в другой). Давайте попробуем создать пример, в котором куб будет превращаться в сферу и обратно. Код шейдера, реализующий эту задачу, показан в листинге 5.43.

Листинг 5.43. Реализация морфинга

```
float4x4 mat;
float4x4 worldmat;
float ViewAngle;
float Min = -3.0f;
float Max = 3.0f;
float4 lightDir = {0.0, 0.0, -0.8, 1};

struct VS_OUTPUT
{
   float4 Pos:     POSITION;
   float3 normal:  TEXCOORD0;
```

```
   float3 viewvec: TEXCOORD1;
   float4 basecol: TEXCOORD2;
};

VS_OUTPUT Vexel_Sh(float4 Pos: POSITION,
  float3 normal: NORMAL)
{
   VS_OUTPUT Out;

   float3 spherePos = normalize(Pos.xyz);

   float t = frac(ViewAngle);
   t = smoothstep(0, 0.5, t) - smoothstep(0.5, 1, t);

   float lrp = Min + (Max - Min) * t;

   Pos.xyz = lerp(spherePos, Pos, lrp);
   normal = lerp(spherePos, normal, lrp);

   Out.basecol = 0.5 + 0.5 * Pos;

   Out.Pos = mul(Pos, mat);
   Out.normal = mul(normal, worldmat);
   Out.viewvec = mul(Pos, worldmat);

   return Out;
}

float4 Pixel_Sh(float3 normal: TEXCOORD0, float3 viewvec: TEXCOORD1,
float4 basecol: TEXCOORD2) : COLOR
{
    float3 normalized = normalize(normal);
    float3 lightn = normalize(lightDir);
    float4 diffuse = saturate(dot(lightDir, normalized));

    float3 reflection = normalize(2 * diffuse * normalized - lightn);
    float3 eyen = normalize(viewvec);
    float4 specular = pow(saturate(dot(reflection, eyen)), 16);

    return basecol * diffuse + specular;
}

technique PixelLight
{
    pass P0
```

```
    {
        VertexShader = compile vs_2_0 Vexel_Sh();
        PixelShader  = compile ps_2_0 Pixel_Sh();
    }
}
```

Чтобы пример был более красивым, в нем реализовано освещение. Именно для этого необходим пиксельный шейдер. Если бы не освещение, то можно было бы обойтись и без него.

Функция `technique` все также имеет имя `PixelLight`. Нет, разумеется, я могу придумать что-то более оригинальнее, но просто уже привык к этой функции и бесхитростно копирую ее из примера в пример.

Самое интересное кроется в вершинном шейдере. Но как сделать из куба сферу? Действительно, если для куба достаточно всего 8 вершин, то для сферы нужно намного больше, потому что через 8 точек сферу нарисовать просто нереально. Проблема решается легко, если куб сделать из большего количества вершин, тем более что это не так уж и сложно. Давайте сделаем каждую грань из 16-ти вершин, как показано на рис. 5.24.

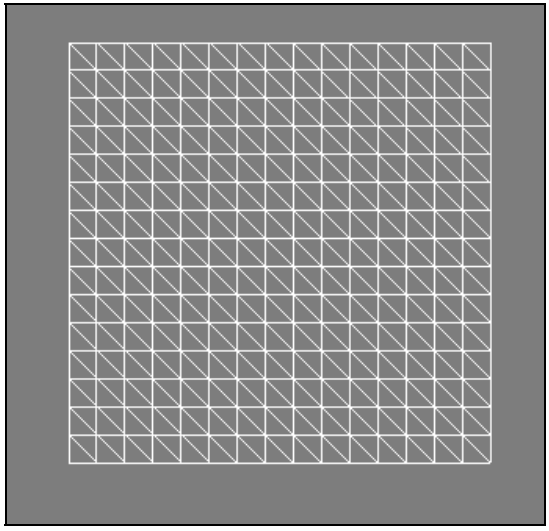

Рис. 5.24. Куб из 16-ти вершин на грань

Вот теперь можно превратить куб в сферу. Достаточно только прописать алгоритм плавного движения вершин от куба к сфере и обратно.

В самом начале шейдера задаются две переменные, которые определяют начальную и конечную позицию вершин:

```
float Min = -3.0f;
float Max = 3.0f;
```

В данном случае движение будет идти от –3 до +3, а это больше, чем нужно для сферы и куба, но результат красивый, поэтому я и установил эти значения. Если хотите получить именно куб и сферу, то измените минимум на –2, а максимум на +2.

Обратите внимание, как определяется позиция сферы:

```
float3 spherePos = normalize(Pos.xyz);
```

Мы просто вычисляем нормаль к текущей позиции вершины и движемся по ней.

Далее идет расчет положения вершины и имеется пара новых функций. Первая из них это `smoothstep`. Этой функции передается три параметра — минимум, максимум и значение X. Функция возвращает плавную Эрмитову интерполяцию от 0 до 1. Что это за интерполяция? Если третий параметр меньше минимума, то результатом будет 0, а если больше максимума, то результат будет 1.

Следующая функция `lerp`, которая так же, как и `smoothstep`, получает три параметра и тоже выполняет интерполяцию, только линейную.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Morth.

5.25. Молния

Двигаемся дальше. Наши эффекты становятся все сложнее и сложнее, и мы подошли к созданию эффекта молнии. Все последние эффекты основаны на шейдере и молния, которую мы сейчас будем рассматривать, тоже основана на шейдере и нет смысла писать пример с нуля. Программу можно взять из *разд. 5.22* (любой пример огня), убрать лишь текстуру огня, но оставить шум, а в коде шейдера у нас изменится только пиксельная функция, которую нам еще предстоит рассмотреть. На рис. 5.25 показан результат выполнения программы.

Итак, пиксельный шейдер создания молнии будет выглядеть следующим образом:

```
float4 Pixel_Sh(float2 texCoord: TEXCOORD0) : COLOR
{
   float2 t = float2(ViewAngle - abs(texCoord.y), ViewAngle);
```

```
    float noise = tex3D(Noise, float3(texCoord.x, t));

    float dist = abs(texCoord.y - height *
           (noise * 2 - 1) * texCoord.x);

    float factor = 1.0 - pow(dist*3, glowFallOff);
    factor *= factor;

    return (factor * 300) * color;
}
```

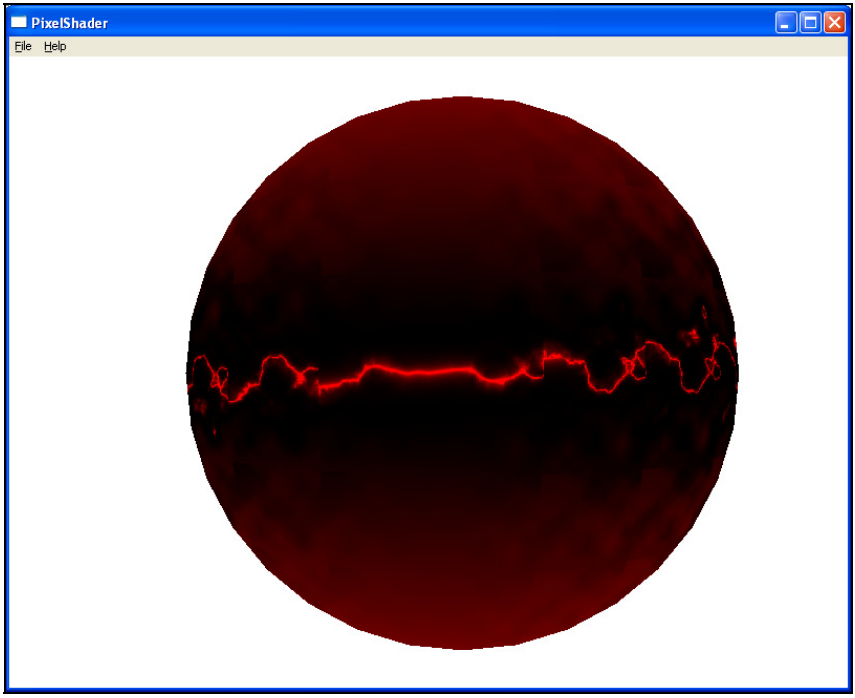

Рис. 5.25. Эффект молнии в кубе

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\Lightning.

5.26. Кубические текстуры в шейдере

Давайте посмотрим, как можно использовать кубические текстуры совместно с шейдером. Сначала посмотрим на код шейдера (листинг 5.44).

Листинг 5.44. Шейдер назначения объекту кубической текстуры

```
float4x4 mat : WORLDVIEWPROJECTION;
float ViewAngle;

// Описание текстуры
Texture sphereTexture;
sampler lightsphere  = sampler_state
  {texture = <sphereTexture>;
   mipfilter = LINEAR; minfilter = LINEAR; magfilter = LINEAR;
   addressu=clamp; addressv=clamp;};

// Структура связи входящих и исходящих данных
struct VS_OUTPUT
{
    float4 Position : POSITION;
    float3 Normal : TEXCOORD0;
};

// Вершинный шейдер
VS_OUTPUT Vexel_Sh(
    float4 Pos : POSITION,
    float3 nor : NORMAL
    )
{
   VS_OUTPUT Out = (VS_OUTPUT)0;

   Out.Position = mul(Pos, mat);

   Out.Normal = nor;

   return Out;
}

// Пиксельный шейдер
float4 Pixel_Sh(
    float3 normal : TEXCOORD0
  ) : COLOR0
{
 return texCUBE(lightsphere, normal);
}

// Функция technique
technique PixelLight
```

```
{
    pass P0
    {
        VertexShader = compile vs_1_1 Vexel_Sh();
        PixelShader  = compile ps_2_0 Pixel_Sh();
    }
}
```

Как видите, тут ничего сложного нет. Вершинный шейдер всего лишь устанавливает позицию и нормаль, а пиксельный шейдер использует эту нормаль для определения цвета вершины. Для этого вызывается функция `texCUBE`, которой нужно передать в качестве первого параметра кубическую текстуру, а во втором параметре нормаль.

В описании текстуры в начале шейдера ничего не говорится о том, что у нас что-то кубическое:

```
Texture sphereTexture;
sampler lightsphere  = sampler_state
  {texture = <sphereTexture>;
   mipfilter = LINEAR; minfilter = LINEAR; magfilter = LINEAR;
   addressu=clamp; addressv=clamp;};
```

Получается, что с кубическими текстурами можно работать так же, как и с простыми. Разница только в функции `texCUBE`, определяющей нужный цвет вершины с учетом кубической текстуры.

Главное отличие находится в основной программе, в которой необходимо загрузить и передать шейдеру именно кубическую текстуру, а не простую. Чтобы сделать это, нужно сначала объявить переменную типа `IDirect3DCubeTexture9`:

```
IDirect3DCubeTexture9 * TextureSphere;
```

Теперь в эту переменную загружаем из файла текстуру. Удачным вариантом здесь будет воспользоваться функцией `D3DXCreateCubeTextureFromFile`:

```
D3DXCreateCubeTextureFromFile(pD3DDevice, "LightCube.dds",
   &TextureSphere);
```

У функции всего три параметра:

- интерфейс устройства Direct3D;
- файл, содержащий текстуру;
- указатель на интерфейс `IDirect3DCubeTexture9`, куда необходимо загрузить текстуру.

Теперь у нас есть текстура и ее нужно назначить. Для этого используем метод `SetValue`:

```
pEffectSphere->SetValue("sphereTexture", &TextureSphere,
   D3DX_DEFAULT);
```

Точно так же мы назначали и некубические текстуры.

Вот и все. Как видите, код программы вывода кубической текстуры с использованием шейдера не такой уж и сложный.

Если вы запустите пример, то в окне должна появиться сфера, каждая "сторона" которой окрашена своим цветом (рис. 5.26).

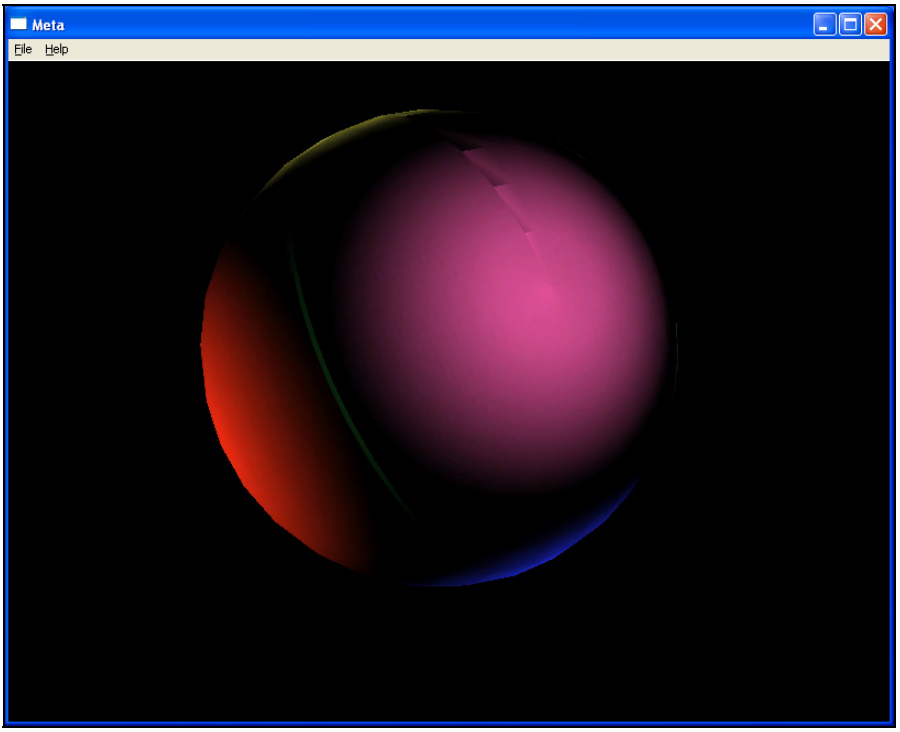

Рис. 5.26. Результат работы программы

Если посмотреть на текстуру, натягиваемую на сферу, то она состоит из 6-ти изображений, каждое своего цвета (рис. 5.27). Все шесть изображений выстроены в ряд, а наша программа любое из них натягивает на свою сторону сферы. На рис. 5.27 каждая картинка состоит из 6-ти копий разного масштаба, дабы экономить время и увеличить скорость текстурирования, хотя можно было бы обойтись и одной копией. Просто изображение выполнено специ-

альным плагином (plug-in) для Photoshop и копии были сделаны автоматически.

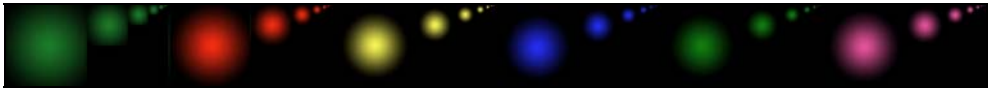

Рис. 5.27. Кубическая текстура

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\CubeTexture.

5.27. Каждому объекту свой шейдер

По мере прочтения книги, наши примеры от раздела к разделу все более усложняются, и теперь уже одним шейдером не отделаешься. В поисках хорошей идеи я наткнулся на демку, в которой внутри комнаты вращался шар с кубической текстурой (примерно, как в *разд. 5.26*) и излучал свет на стены комнаты. Получился достаточно красивый эффект цветомузыки. Да, мы не зря создали сферу, а в этом разделе научим ее еще и излучать свет.

Итак, давайте модифицируем пример из *разд. 5.26* и сделаем его более интересным. Шейдер для сферы не изменится, но в нашей сцене появится куб, внутри которого и будет происходить действие, как в комнате. Этому кубу понадобится шейдер, отображающий отражение света, излучаемого сферой.

Листинг 5.45. Шейдер для комнаты

```
float4x4 mat : WORLDVIEWPROJECTION;
float view_position;
float3x3 mat1;
float4 light_pos = {0, 0, -20, 1};
float ViewAngle;

// Описание текстур сферы
Texture sphereTexture;
sampler lightsphere = sampler_state {
   texture = <sphereTexture>;
   mipfilter = LINEAR; minfilter = LINEAR; magfilter = LINEAR;
   addressu=clamp; addressv=clamp;
};
```

```
// Описание текстуры для стены
Texture wallTexture;
sampler wall = sampler_state {
  texture = <wallTexture>;
  mipfilter = LINEAR; minfilter = LINEAR; magfilter = LINEAR;
  addressu=wrap; addressv=wrap;
};

// Структура связи входных и выходных данных
struct VS_OUTPUT
{
  float4 Position : POSITION;
  float3 tex_coord: TEXCOORD0;
  float3 light_vec: TEXCOORD1;
  float3 light_pos:      TEXCOORD3;
};

// Вершинный шейдер
VS_OUTPUT Vexel_Sh(
    float4 Pos : POSITION,
    float3 normal : NORMAL
    )
{
   VS_OUTPUT Out = (VS_OUTPUT)0;

   Out.Position = mul(Pos, mat);

   Out.tex_coord.x = lerp(Pos.x, Pos.y, abs(normal.x));
   Out.tex_coord.y = lerp(Pos.z, Pos.y, abs(normal.z));

   Pos.xyz *= -100;

   float3 lightvec = light_pos - Pos;
   Out.light_vec.x = dot(lightvec, normal);
   Out.light_vec.y = dot(lightvec, normal);
   Out.light_vec.z = dot(lightvec, normal);

   Out.light_pos = mul(mat1, lightvec);

   return Out;
}

// Пиксельный шейдер
float4 Pixel_Sh(
   float2 tex_coord : TEXCOORD0,
```

```
   float3 light_vec : TEXCOORD1,
   float3 light_pos : TEXCOORD3
  ) : COLOR0
{
   float4 wallpix = tex2D(wall, tex_coord);

   float4 light = texCUBE(lightsphere, light_pos);

   return (wallpix) * light + 0.3 * wallpix;
}

// Функция отображения technique
technique PixelLight
{
    pass P0
    {
        VertexShader = compile vs_1_1 Vexel_Sh();
        PixelShader  = compile ps_2_0 Pixel_Sh();
    }
}
```

В вершинном шейдере просто рассчитываются координаты вершины на основе текущей матрицы и положение источника света. Пиксельный шейдер рассчитывает цвет пиксела стены, при этом учитывается не только цвет текстуры стены, но и цвет отражения текстуры сферы.

Теперь посмотрим код программы, которая будет выводить сцену с учетом обоих шейдеров. Поскольку шейдера два, то нам понадобятся две переменные `ID3DXEffect`:

```
ID3DXEffect* pEffectSphere;
ID3DXEffect* pEffectWall;
```

При этом нам достаточно только одного интерфейса описания вершин `IDirect3DVertexDeclaration9`. Дело в том, что оба эффекта должны выводить данные, загруженные из сетки, и в обоих случаях будет один и тот же формат.

Итак, код инициализации теперь выглядит следующим образом (листинг 5.46).

Листинг 5.46. Код инициализации объектов сцены и шейдеров

```
// Загрузка сетки сферы
dwNumMatSphere = LoadMesh("sphere.x", pD3DDevice,
   &pMeshSphere, &pMeshTexturesSphere,
   "texture.bmp", &pMeshMaterialsSphere);
```

```
// Загрузка сетки комнаты
dwRoomMaterials = LoadMesh("room.x", pD3DDevice,
   &pRoom, &pRoomTextures,
   "Room.bmp", &pRoomMaterials);

// Загрузка текстур куба и сферы
D3DXCreateCubeTextureFromFile(pD3DDevice, "LightCube.dds",
    &TextureSphere);
D3DXCreateTextureFromFile(pD3DDevice, "wall.jpg", &Texture1);

// Загрузка шейдера сферы
D3DXCreateEffectFromFile(pD3DDevice, "sphere.vsh", 0, 0, 0, 0,
        &pEffectSphere, 0);

// Устанавливаем параметры шейдера сферы
pEffectSphere->SetTechnique("PixelLight");
pEffectSphere->SetValue("sphereTexture", &TextureSphere,
   D3DX_DEFAULT);

// Загрузка шейдера комнаты
D3DXCreateEffectFromFile(pD3DDevice, "wall.vsh", 0, 0, 0, 0,
   &pEffectWall, 0);
// Установка параметров комнаты
pEffectWall->SetTechnique("PixelLight");
pEffectWall->SetValue("sphereTexture", &TextureSphere,
     D3DX_DEFAULT);
pEffectWall->SetValue("wallTexture", &Texture1, D3DX_DEFAULT);
```

Обратите внимание, что текстура, которая назначается сфере, передается также и шейдеру комнаты, которому необходимо знать форму источника света. Код листинга очень подробно закомментирован, поэтому не составит труда разобраться с ним.

Теперь нужно вывести объекты сцены. Соответствующий код показан в листинге 5.47.

Листинг 5.47. Код отображения светомузыки

```
void DrawScene()
{
 // Поворачиваем угол обзора
 ViewAngle+=ViewAngleOffset;
 float b=ViewAngle;
```

```
// Матрица поворота для сферы
D3DMATRIX World = {
 cos(b)*cos(b), cos(b)*sin(b), sin(b), 0,
 -sin(b), cos(b), 0, 0,
 -sin(b)*cos(b), -sin(b)*sin(b), cos(b), 0,
 0, 0, 0, 1,
};

// Матрица 3×3, которая включает только поворот
D3DMATRIX matWorld1 = {
 cos(b)*cos(b), cos(b)*sin(b), sin(b),
 -sin(b), cos(b), 0,
 -sin(b)*cos(b), -sin(b)*sin(b), cos(b),
};

// Устанавливаем формат вершин. Так как он одинаков для всех
// объектов, это действие выполняется только один раз
pD3DDevice->SetVertexDeclaration(VertDeclSphere);

D3DXMATRIXA16 matWorld
D3DXMatrixScaling(&matWorld, 10.0f, 10.0f, 10.0f);

// Объединяем матрицы для комнаты
D3DXMATRIX Full1;
D3DXMatrixMultiply(&Full1, (D3DXMATRIX*)&World, (D3DXMATRIX*)&View1);
D3DXMatrixMultiply(&Full1, (D3DXMATRIX*)&View1, &matProjection);

// Задаем переменные для шейдера комнаты
pEffectWall->SetValue("mat", &Full1, D3DX_DEFAULT);
pEffectWall->SetValue("mat1", &matWorld1, D3DX_DEFAULT);
pEffectWall->SetValue("view_position", &matWorld, D3DX_DEFAULT);
pEffectWall->SetValue("ViewAngle", &ViewAngle, D3DX_DEFAULT);

// Выводим сетку комнаты через шейдер pEffectWall
pD3DDevice->SetRenderState( D3DRS_ZFUNC, D3DCMP_ALWAYS );
UINT uPass;
pEffectWall->Begin(&uPass, 0);
for (int i=0; i<uPass; i++)
{
 pEffectWall->Pass(i);
 for (DWORD i=0; i<dwRoomMaterials; i++)
      pRoom->DrawSubset(i);
}
pEffectWall->End();
```

```
 // Формируем суммарную матрицу для сферы
 D3DXMATRIX Full;
 D3DXMatrixMultiply(&Full, (D3DXMATRIX*)&World, (D3DXMATRIX*)&View);
 D3DXMatrixMultiply(&Full, &Full, &matProjection);

 // Задаем параметры для сферы
 pEffectSphere->SetValue("mat", &Full, D3DX_DEFAULT);
 pEffectSphere->SetValue("view_position", &View, D3DX_DEFAULT);
 pEffectSphere->SetValue("ViewAngle", &ViewAngle, D3DX_DEFAULT);

 // Выводим сетку сферы через шейдер pEffectSphere
 pEffectSphere->Begin(&uPass, 0);
 for (int i=0; i<uPass; i++)
 {
  pEffectSphere->Pass(i);
  for (DWORD i=0; i<dwNumMatSphere; i++)
     pMeshSphere->DrawSubset(i);
 }
 pEffectSphere->End();
}
```

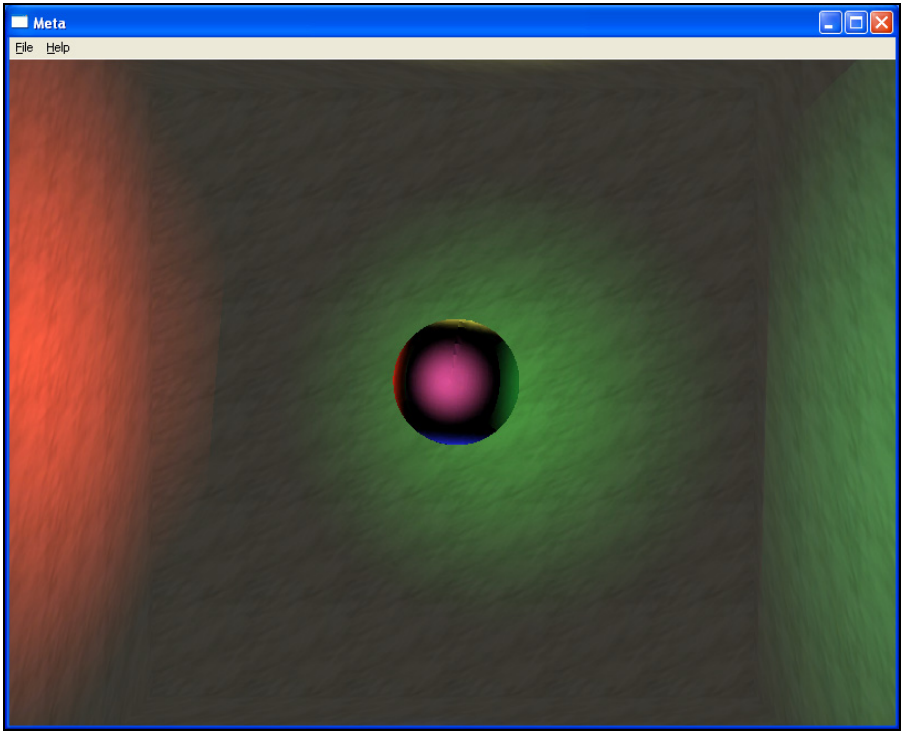

Рис. 5.28. Результат работы программы светомузыки

Все функции и методы, которые использованы в этом примере, вам уже знакомы, а как они применяются, помогут разобраться комментарии. Единственное, на чем хочется остановиться — матрица `matWorld1`. Это матрица размером 3×3, которая включает в себя только поворот и не отражает положение объекта. Это необходимо, чтобы упростить расчеты положения света в вершинном шейдере. Наш шар будет только вращаться, и для него нужны данные только о повороте.

Запустите пример, и вы увидите что-то похожее на рис. 5.28.

Примечание

Исходный код примера находится на прилагаемом компакт-диске в каталоге \Chapter5\CubeTexture1.

Заключение

За что я люблю графику, так это за то, что она дает бескрайние просторы пищи для размышлений. Я надеюсь, что данная книга прибавила и вам пищи для ума хотя бы на некоторое время, и теперь будет чем заняться в свободный вечерок, а может и бессонную ночь. В этом случае можно считать, что программа минимум этой книги выполнена. Лично я люблю играть с графикой в свободное время, жаль только, что этого времени маловато.

Ну а если материал книги оказался полезным для решения реальной задачи в какой-либо из ваших графических программ (а может, и программ другого типа), то можно считать, что программа выполнена на половину.

Конечно, написать все о графике в одной книге просто невозможно, потому что это слишком обширная тема, особенно если касаться демо-сцены. Если же мне удалось вас заинтересовать и погрузить в сцену, то можно считать, что выполнена программа максимум. Надеюсь, что на одной из ближайших демо-вечеринок будет выставлена и ваша работа.

Если у вас получится улучшить какой-либо из примеров книги, и не жалко будет поделиться своим творением, то я с удовольствием жду ваши письма на адрес **horrific@vr-online.ru**. Только очень прошу — не высылать мне исполняемые файлы, я смотрю только исходники. Я знаю вас хакеров и любителей вирусов и троянов, но у меня нет антивируса (не накопил еще денег ☺, а нелегальные копии я стараюсь не использовать). А поскольку мой компьютер содержит слишком важные данные и рисковать ими нельзя, то все исполняемые файлы будут "резаться на корню".

ПРИЛОЖЕНИЕ

Описание компакт-диска

Папки	Описание
\Chapter1	Программы, описанные в *главе 1*. Шаблонные приложения 2D- и 3D-программы
\Chapter2	Программы, описанные в *главе 2*. Примеры, иллюстрирующие основы работы с DirectDraw и Direct3D
\Chapter3	Программы, описанные в *главе 3*. Примеры, иллюстрирующие основы оптимизации
\Chapter4	Программы, описанные в *главе 4*. 2D-эффекты
\Chapter5	Программы, описанные в *главе 5*. 3D-эффекты
\Common	Общие модули, которые используются в проектах книги
\Additional	Дополнительная информация к книге
\Demo	Демонстрационные программы
\Sample	Пример, иллюстрирующий работу освещения Direct3D

Список литературы

1. Фленов М. Программирование на C++ глазами хакера. — СПб.: БХВ-Петербург, 2004. — 350 с.
2. Фленов М. Программирование в Delphi глазами хакера. — СПб.: БХВ-Петербург, 2003. — 370 с.
3. Фленов М. Компьютер глазами хакера. — СПб.: БХВ-Петербург, 2005. — 336 с.
4. Фленов М. Библия Delphi. — СПб.: БХВ-Петербург, 2004. — 880 с.

Предметный указатель

X

Б

В

Д

И

К

М

Н

П

С

У

Ф

Ш

Э

Я