в помощь разработчику игр

CD-ROM
в комплекте

DIRECTX®

ПРОДВИНУТАЯ АНИМАЦИЯ

КУДИЦ-ОБРАЗ

Джим Адамс

Редактор серии Андре ЛаМот
(CEO Xtreme Games LLC)

Advanced Animation with DirectX

Jim Adams

PREMIER PRESS

Джим Адамс

DirectX: продвинутая анимация

КУДИЦ-ОБРАЗ
Москва • 2004

ББК 32.973.26

Адамс Д.

DirectX: продвинутая анимация / Пер. с англ. - М.: КУДИЦ-ОБРАЗ, 2004.-480 с.

Данная книга посвящена современным методам анимации с использованием DirectX. В книге подробно освещаются последние достижения в области персонажной, лицевой и скелетной анимации. Вы узнаете о методах морфирования, как учитывать законы физики для правдоподобной анимации твердых и мягких тел, как использовать частицы в анимации. Подробно рассматривается работа с форматом .X файлов, анимированными текстурами, использование шейдеров, синхронизация лицевовой анимации и звука. Если вы хотите овладеть приемами трехмерной анимации, то эта книга для вас — на текущей момент это лучшая книга в своей области на русском языке.

ISBN 1-59200-037-1
ISBN 5-9579-0025-7

Джим Адамс
DirectX: продвинутая анимация

Учебно-справочное издание

Перевод с англ. В. Ю. Щербаков
Науч. ред. И. В. Кошечкин

«ИД КУДИЦ-ОБРАЗ».
119049, Москва, Ленинский пр-т, д. 4, стр. 1А. Тел.: 333-82-11, ok@kudits.ru

Подписано в печать 30.06.2004
Формат 70x90/16. Бум. газетная. Печать офсетная
Усл. печ. л. 35,1. Тираж 3000. Заказ 1375

Отпечатано в ОАО «Щербинская типография»
117623, г. Москва, ул. Типографская, д. 10

ISBN 1-59200-037-1
ISBN 5-9579-0025-7 «ИД КУДИЦ-ОБРАЗ», 2004

Краткое содержание

Благодарности

Написание книги явилось нелегкой задачей. Множество часов было потрачено на написание, редактирование, форматирование и распечатку этой книги. Я очень ценю терпение и потраченные силы всех тех, кто участвовал в разработке этой книги. Этим людям я хочу сказать спасибо!

Прежде всего, я хотел бы поблагодарить свою семью, за поддержу. Мою любящую жену, за ее терпение, пока я писал эту книгу. Моих детей, Michael и John, за то что были со мной. Хочу также сказать спасибо за поддержку моей матери Pam, братьям John, Jeff и Jason, сестре Jennifer, и племяннику Jordan. Bill and Doris Fong - за все, что они сделали для меня. Моему приятелю Jeff Young - за предоставление помощи в завершении книги и Richard Young - за предоставление места на сайте для этой книги.

Всей издательской команде - Emi Smith, Cathleen Snyder, Andre LaMothe и Brandon Penticuff - я бы хотел сказать спасибо, в т. ч. и за возможность поработать с вами снова. Emi, твое бесконечное терпение просто поражает. Спасибо тебе, Cathleen за то, что процесс редактирования прошел так легко. И, наконец, Andre за помощь в создании такой великолепной серии книг. И, конечно же, Brandon, - спасибо, что смог поместить все так быстро на компакт-диск!

Также я бы хотел поблагодарить фирму Caligari, которая помогает игровому сообществу своими замечательными продуктами , за создание trueSpace и такой замечательной вещи как Facial Animator; Discreet за их программу моделирования 3D Studio Max; Curious Labs за их необычайную программу Poser для создания фигур человека; Microsoft, за создание такой полезной вещи как DirectX.

И, наконец, я бы хотел поблагодарить всех читателей за покупку этой книги!

06 авторе

Джим Адамс начал свою карьеру программиста в зрелом возрасте 9 лет, когда проявились его любопытство и воображение. Двадцать один год спустя в возрасте 30 лет (очень зрелый!) Джим все еще интересуетсяъ современными технологиями создания игр, как например описанными в этой книге. Несмотря на все это, Джим находит время для семьи, успевает написать книгу-другую и изредка помочь своей жене.

Между моментами, когда Джим занимается написанием книг и программированием, вы можете его найти модерирующим DirectX форум на одном из лучших веб-сайтов посвященных программированию игр http://www.GameDev.net. Обязательно зайдите туда!

Письмо редактора серии

Добро пожаловать в DirectX: продвинутая анимация. Эта книга полностью посвящена технологиям трехмерного анимирования, используемым при разработке высококачественных AAA тайтлов, таких как Unreal, Doom III и др., позволяющих использовать скелетную и лицевую анимации трехмерных моделей. Большей частью, материал этой книги никогда раньше не издавался. Вы можете найти некоторые темы этой книги в Интернете, в статьях или других книгах, но это единственная книга, посвященная продвинутым методам анимирования в играх. Целью автора этой книги и моей было показать продвинутые технологии анимирования и базовую физику, на которую он опирается. Конечно при слове "физика" или "математика" у вас начинают бегать мурашки, но Джим постарался использовать объяснения физических моделей с точки зрения программиста.

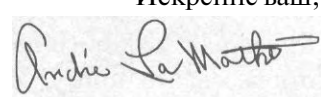
В этой книге рассмотрены замечательные вещи, как например физика "кукол". Великолепным примером этой технологии является Unreal Tournament; когда вы убиваете оппонента, его тело подкашивается, и сила выстрела отбрасывает его, совсем как куклу!

В книге также рассмотрены анимация одежды и имитация пружин. Используя эти технологии, вы можете создать все что угодно, начиная от одежды и заканчивая мягкими телами, которые могут изменять, а потом восстанавливать свою форму под действием внешних сил.

Наконец, самыми замечательными технологиями являются лицевая анимация и синхронизация губ. Эти технологии используются в очень малом количестве игр, и наверное, пока не будут использоваться. Опять же, Джим объясняет все тонкости этой технологии.

В завершение хотелось бы сказать, что эта книга действительно является продвинутой и содержит очень серьёзный материал. Если вы начинающий программист, то некоторые моменты останутся непонятными для вас, но прежде чем погружаться в это безумие, я рекомендую вам изучить основы трехмерной графики, математику и физику на уровне средней школы. Тем не менее, после того как вы освоите эту книгу, 90% разработчиков игр даже не будут иметь понятия о известных вам технологиях.

Искренне ваш,



Андрё ЛаМот (Andre LaMothe),
редактор серии Premier Press Game Development

Введение

Итак, вы уже знаете основы. Вы уже научились рисовать полигоны и мешы, смешивать текстуры, управлять буферами вершин и работать с вершинными шейдерами, так чему же еще необходимо научиться честолюбивому программисту игр? Замечательно, вы обратились по правильному адресу, если готовы двинуться далее основ.

Добро пожаловать в DirectX: продвинутая анимация! Эта книга является гидом от основ программирования графики с помощью DirectX к большому миру продвинутой анимации! Посмотрите, как используя информацию, которую вы уже знаете, можно очень быстро создать множество привлекательных графических эффектов.

Однако вам не понадобятся учебники средней школы, потому что эта книга не будет надоедать вам теорией и алгоритмами. Вместо этого вы увидите настоящие примеры самых современных технологий анимации. Содержа в себе очень простые теории, полностью откомментированный код и замечательные демонстрационные программы, эта книга сделает изучение продвинутых технологий легким и радостным!

О чем эта книга

Как вы можете видеть из содержания (вы же его смотрели?) эта книга была написана для средних и продвинутых программистов. Поэтому в ней нет разделов для начинающих (ну, почти нет) - только сложные теории и код!

Это означает, что не тратилось место на такие простые вещи, как инициализация Direct3D или использование механизма сообщений Windows, так что для изучения этой книги необходимы базовые знания Direct3D. Подождите! Пока не откладывайте книгу. Я говорю о самых началах, таких как инициализация Direct3D, использование материалов и текстур, управление буферами вершин. Если вы все это знаете, определенно вы переходите к более продвинутым методам, а эта книга как раз для этого!

Почему вам следует прочитать эту книгу

Это вопрос на миллион долларов - зачем читать эту книгу? Давайте смотреть правде в глаза: игровые технологии развивают очень быстро, и многие из нас не успевают уследить за ними. Сегодня одно, завтра другое. В целом, книга содержит информацию о продвинутых технологиях, которые вы можете использовать в своих проектах и которые удивят остальных!

Так какие же продвинутые технологии анимации я рассматриваю? Ну, мой друг, читайте далее, чтобы увидеть, что может предложить вам эта книга.

Что содержится в этой книге?

В этой книге содержится 14 глав, полностью посвященных продвинутой анимации. Каждая глава освещает одну технологию анимации; если не брать несколько первых глав, книга является полностью модульной, т. е. вы можете пропустить не интересные вам главы и перейти к интересующим вас темам.

Конечно же, вас очень интересует каждая глава книги, так почему бы не воспользоваться моментом и не посмотреть, что вас ожидает! Следующий список содержит резюме каждой главы.

- **Глава 1: Подготовка к изучению книги.** Подготовьте себя, потому что эта книга сразу же переходит к использованию DirectX для создания привлекательных анимаций в ваших программах! Эта глава поможет проинсталлировать DirectX и настроить компилятор, а потом сразу же переходит к рассмотрению библиотеки объектов и функций, созданных для ускорения разработки приложений. Прежде чем читать остальные главы, вы должны обязательно прочитать эту главу.
- **Глава 2: Синхронизация анимации и движения.** Синхронизация является важной частью анимации, и эта глава является вводной по отношению к остальным. В ней содержится информация о перемещении и анимировании мешей во времени.

  Глава 3: Использование формата файла .X. Получение данных меша является достаточно сложным. В этой главе содержится информация об использовании собственного формата хранения трехмерной графики корпорации Microsoft .X. В этой главе вы научитесь сохранять и загружать информацию о трехмерном меше и использовать файлы .X для хранения специализированных данных проекта.
- **Глава 4: Работа со скелетной анимацией.** Наверное, наиболее продвинутой техникой анимации в реальном времени является скелетная анимация. Эта

глава освещает тему скелетной анимации, показывая, как использовать эту технологию для управления мешами, основанными на соответствующем наборе костей.

- **Глава 5: Использование скелетной анимации, основанной на ключевых кадрах.** Одна из самых распространенных технологий анимации использует наборы анимаций, основанных на ключевых кадрах (создаваемых в пакетах трехмерного моделирования, таких как 3D Studio Max фирмы Discreet или trueSpace фирмы Caligari). Эта глава показывает, как использовать информацию скелетной анимации, основанной на ключевых кадрах (хранимой в файле .X) для анимирования меша.
- **Глава 6: Комбинирование скелетных анимаций.** Вам надоела статичность анимаций, основанных на ключевых кадрах? Я говорю об анимациях, которые никогда не меняются! А как насчет комбинирования нескольких наборов анимаций для создания новых?
- **Глава 7: Создание кукольной анимации.** Вот оно; я знаю вы этого ждали. В этой главе содержится информация о создании собственных кукольных анимаций. Вам не потребуются учебники по физике; в этой книге содержится вся необходимая информация, изложенная в простом виде.
- **Глава 8: Работа с морфирующей анимацией.** Несмотря на скелетные анимации, морфирующие анимации имеют право на существование в мире продвинутых анимаций! Посмотрите, как можно использовать простые технологии анимации, чтобы морфировать меши.
- **Глава 9: Использование морфирующей анимации, основанной на ключевых кадрах.** Даже морфирующие анимации должны быть последовательными, не так ли? Посмотрите, как определять и создавать собственные последовательности объектов анимации, основанной на ключевых кадрах и применять их в технологии морфирующей анимации! Также в этой главе расширяется полезность .X созданием собственных шаблонов морфируемых ключевых кадров!
- **Глава 10: Комбинирование морфированных анимаций.** Еще раз мы используем комбинирование, на этот раз для соединения различных комбинированных анимаций в новые в реальном времени!
- **Глава 11: Морфируемая лицевая анимация.** Они ходят, прыгают и даже разговаривают! При использовании технологий лицевой анимации, показанной в этой главе, игровые персонажи оживают. Если вы не знаете возможностей лицевой анимации, посмотрите на игры Medal of Honor: Frontline фирмы Electronic Art или Gate: Dark Alliance фирмы Interplay.

Глава 12: Использование частиц в анимации. Какая же современная игра не использует сверкающие, клубящиеся, хлещущие капли графических пикселей, которые называются частицами? Ваша, скажите вы? Хорошо, не волнуйтесь, - эта глава покажет, как можно эффективно использовать анимацию частиц.

Глава 13: Имитирование одежды и анимация мешей мягких тел. Закройте глаза и представьте, что игровой персонаж носит шелковый плащ, простирающийся до земли, который развевается при малейшем ветре. При использовании имитации одежды вы можете дать вашему персонажу плащ, одежду и еще множество всего. Дополните это использованием мешей мягких тел, которые могут деформироваться от малейшего прикосновения, и вы получите продвинутую технологию анимации!

- Глава 14: Использование анимированных текстур. Лучшее я оставил на потом. Эта глава иллюстрирует применение анимированных текстур, используемых для рисования трехмерного мира. Анимированный огонь, текущая вода и еще множество всяких возможностей, и все это при использовании анимированных текстур, рассмотренных в этой главе!

Фу! Вашему вниманию были представлены основные темы! После того как я написал книгу, я не могу вам ничем помочь, но мне интересно, в каких приложениях будут использоваться описанные технологии. Я предвижу множество игр, использующих эти технологии, и уверен, что как раз вы их и сделаете. Я уверен, что после прочтения книги (если вы как я, то сделаете это не менее шести раз), вы найдете способы применить полученные знания в игровых проектах.

Я знаю, что вам не терпится начать, поэтому позвольте мне еще раз сказать вам добро пожаловать и поблагодарить за покупку этой книги. Я надеюсь, что она поможет вам изучить теорию и технологии, используемые в трехмерных анимациях. Веселитесь и получайте удовольствие!

Часть I

Подготовка

1. Подготовка к изучению книги

Глава 1

Подготовка к изучению книги

Альфа и омега - начало и конец. Все имеет начало и в этой книге данная глава представляет собой начало грандиозного путешествия в мир современной анимации. Позади вас - мощь Microsoft DirectX SDK (средство разработки программного обеспечения DirectX); перед вами - длинный извилистый путь, заполненный неуверенностью. Однако прежде чем вы начнете, необходимо сделать множество вещей, чтобы получаемый опыт был радостным.

Эта глава содержит информацию, которая поможет вам подготовиться к изучению разделов и кодов книги, от инсталляции DirectX SDK до понимания, как использовать ряд полезных функций для уменьшения времени разработки ваших проектов. Прежде чем вы углубитесь в книгу, я настоятельно рекомендую вам полностью прочитать эту главу и не торопиться устанавливать DirectX и компилятор. После этой главы вы будете готовы начать грандиозное путешествие.

К концу этой книги вы обнаружите, что мир современной анимации настолько труден или настолько прост, насколько вы его сделаете. С теми знаниями, которыми вы уже обладаете и с помощью этой книги, я уверен, что для вас это будет просто!

В этой главе вы научитесь:

- Устанавливать DirectX SDK;
- Выбирать подходящие динамические библиотеки DirectX;
- Конфигурировать ваш компилятор для использования DirectX;
- Использовать полезные функции, описанные в книге, для облегчения процесса разработки.

Установка DirectX SDK

Добро пожаловать в грандиозный мир программирования DirectX! Прежде чем углубляться в текст и коды этой книги, установите Microsoft DirectX Software Development Kit (DX SDK), если вы еще этого не сделали. Если вы незнакомы с процессом установки DirectX, не волнуйтесь - он был настолько упрощен, что вам не придется сделать более нескольких щелчков мыши. Что же касается опытных программистов DirectX, вам тоже будет полезно взглянуть на инструкции по установке, на случай если вы чего-то не заметили.

Во время написания этой книги, вышло Microsoft DirectX SDK, версия 9. Всегда лучше удалять предыдущие установки DirectX SDK, прежде чем устанавливать новые. Не волнуйтесь, вы не потеряете ценного кода, потому что каждая новая версия DirectX SDK включает в себя неизмененный код предыдущих выпусков! Именно так, с DirectX 9 (и предыдущими версиями DirectX) у вас имеется полный доступ к каждому компоненту, когда-либо сделанному в DirectX, от самого старого объекта поверхности DirectDraw до объектов для работы с новейшими устройствами трехмерного ускорения! Так что код пользователей DirectX 8 будет работать и с DirectX 9, т. к. версия 9 содержит объекты всех предыдущих версий.

Для пользователей DirectX 8 обновление от DirectX 8 SDK до DirectX 9 SDK может выглядеть немного шокирующим, но как только вы присмотритесь, вы поймете, что не так уж много изменилось. Просто в DirectX 9 SDK добавлено несколько новых особенностей по сравнению с версией 8.

Microsoft упаковала DirectX SDK в единый, легко запускаемый инсталляционный пакет, который возможно вы найдете на компакт-диске, поставляемом с книгой. Процесс инсталляции DirectX остается неизменным для последних версий, так что если вы инсталлировали его ранее, вы уже знаете, что нужно делать. В DirectX 9, инсталляционный экран (показанный на рис. 1.1 и рис. 1.2) немного изменился внешне, в то время как основные шаги установки остаются неизменными во всех версиях.

Для простоты, люди из Primier Press создали меню в интерфейсе компакт-диска, чтобы вы могли легко находить нужные программы. Так получилось, что одна из этих программ - DirectX SDK. Чтобы получить доступ к меню (если оно автоматически не появилось при вставке компакт-диска), нажмите Start (Пуск) и выберете Run (Выполнить). Напечатайте "D:\start_here.html" (где D - буква вашего CD_ROM) в текстовом поле и нажмите кнопку "Ок".

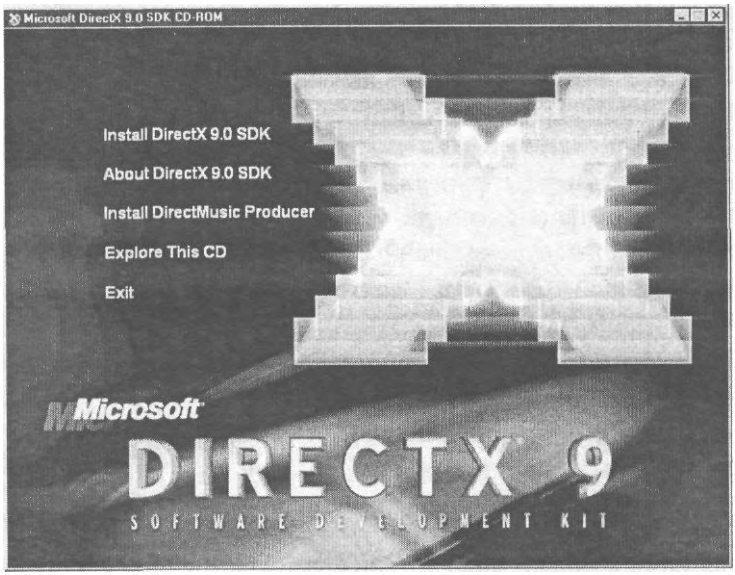


Рис. 1.1. Вводный экран DirectX 9 SDK предоставляет вам несколько вариантов - наиболее важные инсталляционные опции, связанные с DirectX 9 SDK

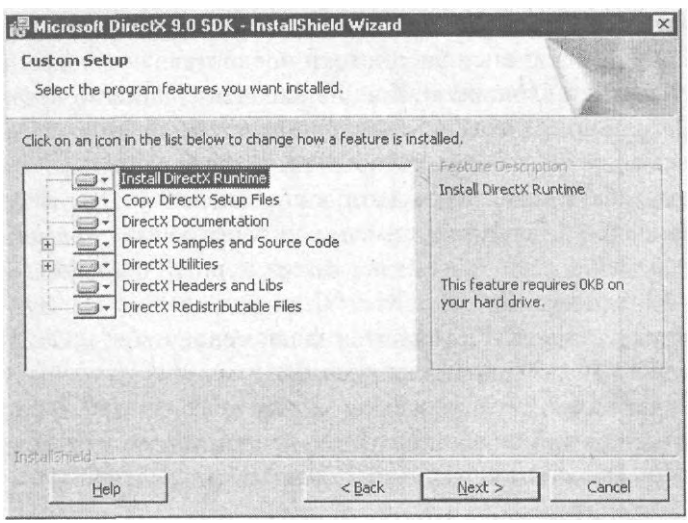


Рис. 1.2. Щелкнув по "Install"("Установка") DirectX 9.0 SDK вы попадете в мастер InstallShield, где вы можете решить, какие компоненты SDK установить

Вы увидите лицензионное соглашение Primier Press. Пожалуйста, прочитайте соглашение и, если вы согласны с условиями, щелкните "I agree" ("Я согласен"), чтобы продолжить. Появится основной интерфейс диска, где вы можете увидеть и выбрать опцию "Install DirectX SDK" ("Установка DirectX SDK").

После того как появится диалоговое окно установки Microsoft DirectX SDK мастера InstallShield, вам придется решить, какие компоненты включить в инсталляцию. Если вы неопытный пользователь, вам необходимо щелкнуть "Next" ("Далее") чтобы установить компоненты, выбранные по умолчанию. Опытные пользователи могут расширить настройки, чтобы выбрать то, что им необходимо. В любом случае, для завершения установки DirectX 9.0 SDK, убедитесь, что вы нажали "Next", и следуете указаниям, появляющимся на экране. Прежде чем вы узнаете об этом, вы будете готовы работать с SDK!

Выбор отладочных или рабочих версий библиотек

Другая важная часть использования DirectX SDK - это выбор динамических (run-time) библиотек разработчика, которые вы будете использовать. Эти библиотеки отличаются от динамических библиотек, которые будут использовать конечные пользователи; они позволяют вам выбирать, будут ли использованы отладочные (debug) или рабочие (retail) библиотеки.

Разработчики используют отладочные run-time библиотеки для отслеживания ошибок в проекте. Отладочные библиотеки предоставляют полезные сообщения во время компиляции и выполняют специальные операции во время выполнения программы. Например, DirectSound во время тестирования звуковых буферов заполняет их статическим звуком, который вы можете услышать.

Когда вы разрабатываете игровой проект, рекомендуется использовать отладочные динамические библиотеки, заменяя их на рабочие динамические библиотеки во время финального тестирования ваших игр. Чтобы переключаться между этими двумя библиотеками, откройте "Control Panel" ("Панель управления") и выберите иконку "DirectX". Появится диалоговое окно "DirectX Properties" ("Свойства DirectX"), как показано на рис. 1.3.

На рис. 1.3 вы видите, что я выбрал вкладку "Direct3D". В этой вкладке вы можете выбирать динамическую библиотеку, используемую в секции "Debug/Retail D3D Runtime". Вы можете выбрать "Use Debug Version of Direct3D" ("Использовать

1. Retail - рабочая версия; версия, выпущенная в продажу; окончательная версия. — *Примеч. перев.*

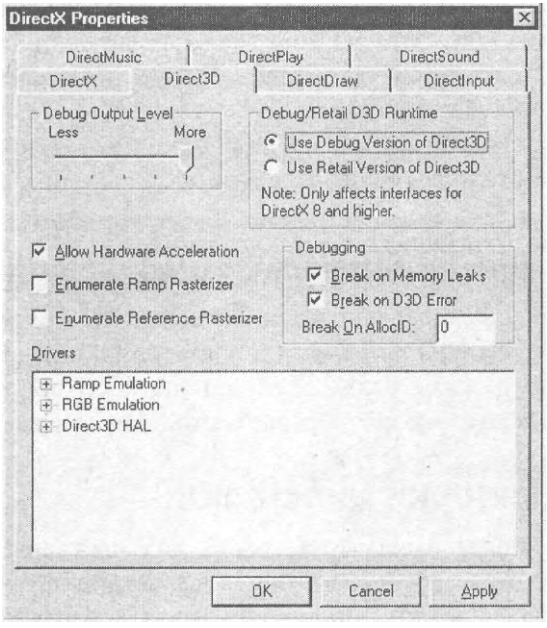


Рис. 1.3. Системные свойства DirectX предоставляют вам множество вариантов. Если выбрана вкладка Direct3D, вы можете видеть, что используются отладочные библиотеки на самом высоком уровне отладки

отладочную версию Direct3D") или "Use Retail Version of Direct3D"("Использовать рабочую версию Direct3D"). Я рекомендую использовать вариант "Use Debug Version of Direct3D" везде, где только возможно.

Однако, если вы решите использовать отладочные библиотеки, имейте ввиду, что программы будут работать не максимально эффективно. Отладочные библиотеки созданы для отладки ваших DirectX приложений; некоторые особенности реализованы так, что изменяют принцип работы DirectX. Например, отладочные библиотеки будут выдавать отладочную информацию о каждом интерфейсе DirectX, о его использовании, освобождении. Каждый нюанс записывается, что замедляет выполнение вашего приложения.

Чтобы корректировать количество отладочных сообщений Direct3D, которые Direct3D будет предоставлять отладчику, вы можете перемещать ползунок в части "Debug Output Level"("Уровень вывода отладки") свойств Direct3D. Чем больше уровень отладки, тем больше сообщений вы будете получать. Я рекомендую переместить этот ползунок в крайнее правое положение, чтобы получать все отла-

дочные сообщения во время разработки проекта. Опять же, это замедляет выполнение вашей программы, но зато вы можете контролировать все, что происходит. Когда ваш проект близится к завершению, быстро измените используемые библиотеки в панели управления DirectX.

Когда вы установили динамические библиотеки и уровень отладки, щелкните "Ок" и закройте диалоговое окно свойств DirectX.

Настройка вашего компилятора

После того как вы проинсталлировали DirectX SDK, вам необходимо подготовить ваш компилятор для работы с кодом и примерами книги. Несмотря на то, что инсталляционная программа DirectX SDK хорошо настраивает некоторые опции компилятора для вас, я хочу рассмотреть все настройки, которые вам надо изменить.

Установка директорий DirectX SDK

Основная (и наиболее важная) настройка для инсталляции DirectX SDK-директории. Ваш компилятор должен знать, где искать заголовочные и библиотечные файлы DirectX. Обычно инсталляционная программа вставляет директории SDK в компилятор Microsoft Visual C/C++ сама, но возможно когда-нибудь вам потребуется самостоятельно добавлять эти директории.

Чтобы добавить директории в MSVC версии 6, откройте ваш компилятор и выберите "Tools" ("Инструменты"). В появившемся списке выберите "Options" ("Опции") и щелкните вкладку "Directories" ("Директории"). Появиться диалоговое окно "Options" (как показано на рис. 1.4).

Пользователи Visual Studio .NET должны выбрать "Tools", после чего выбрать "Projects folder" ("Папки проекта") в диалоговом окне "Options". Далее выберите "VC++ Directories" ("Директории VC++"), чтобы открыть список директорий справа от диалогового окна "Options" (как показано на рис. 1.5).

В обоих случаях вы увидите список директорий, в которых компилятор ищет библиотеки и заголовочные файлы. Обратите внимание на раскрывающееся меню "Show Directories For"("Показывать директории для"). Для начала вам необходимо установить директорию для заголовочных файлов DirectX SDK, так что выберите из раскрывающегося меню "Include Files" ("Подключаемые файлы").

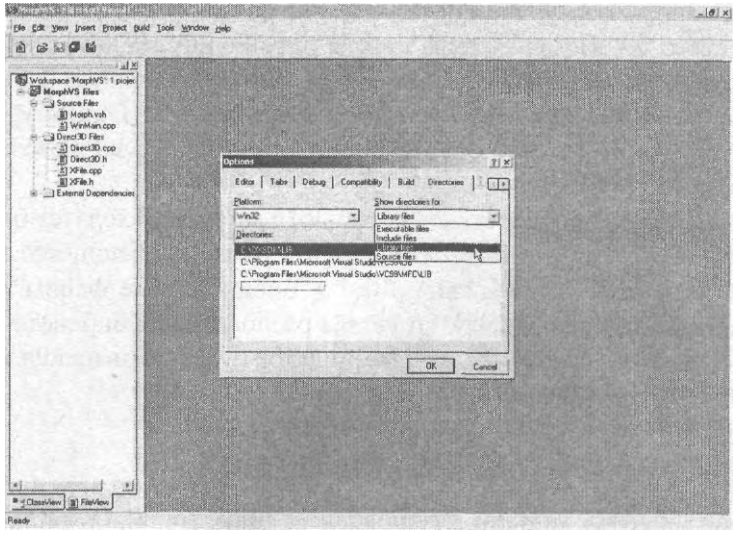


Рис. 1.4. Диалоговое окно "Options" отображает список директорий, в которых Visual C/C++ ищет заголовочные и библиотечные файлы

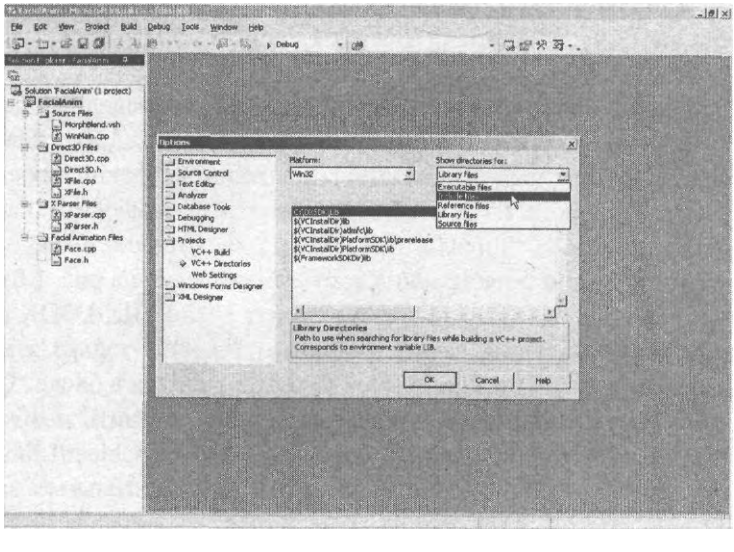


Рис. 1.5. Выбор " VC++ Directories " приводит к появлению списка директорий справа от диалогового окна в Visual Studio .NET

Далее щелкните кнопку "New"("Новый" - небольшой заштрихованный квадрат слева от красного X). Новая строчка списка директорий станет активна в разделе диалогового окна "Directories" . Щелкните по кнопке с многоточием справа от текстового курсора, чтобы открылось диалоговое окно "Choose Directory" ("Выберите директорию"). Укажите проинсталлированную директорию с заголовочными файлами DirectX и нажмите "OK".

Далее вам необходимо указать директорию, в которой находятся библиотечные файлы DirectX SDK, так что ещё раз щелкните на раскрывающееся меню "Show Directories For" и выберите "Library files" ("Библиотечные файлы"). Повторите процесс, щелкнув на кнопку "New" и указав расположение библиотечных файлов DirectX SDK. Когда вы закончите, ваш компилятор будет готов использовать директории DirectX для компиляции.

Привязывание к библиотекам DirectX

После того как вы указали установочные директории DirectX, следующим важным шагом является привязывание библиотек, которые вы будете использовать в проекте. Заметьте, привязывание файлов действует только в пределах проекта, так что убедитесь, что ваш проект игры открыт, перед тем как продолжить.

Для MSVC 6 щелкните "Project" ("Проект"), выберите "Settings" ("Настройки"). Выберите вкладку "Link"("Связь"). Появятся свойства "The Project Settings Link" (как показано на рис. 1.6).

Для Visual Studio .NET откройте ваш проект, потом выделите его в "Solution Explorer" ("Проводник решений") (как показано на рис. 1.7). Далее выберите "Project", а потом "Properties", чтобы открыть диалоговое окно проекта "Property Pages" ("Страницы свойств"). В отображенной папке, выберите папку "Linker" ("Связыватель") и щелкните "Input" ("Ввод"). Справа от диалогового окна "Property Pages" вы увидите проектные настройки связи (как показано на рис. 1.8).

Для этой книги я использовал только библиотеки Direct3D, D3DX и DirectShow, так что в процессе связывания вам необходимо добавить только эти библиотеки (кроме стандартных прикладных библиотек уже включенных в блоке "Object/Library Modules" (Объектные/Библиотечные модули)). Чтобы добавить необходимые библиотеки, вставьте следующий текст в текстовый блок "Object/Library Modules" (для MSVC 6) или в "Additioanl Dependecies" ("Дополнительные зависимости") (для VC.NET):

```
d3d9.lib d3dx9.lib d3dxof.lib dxguid.lib winmm.lib
```

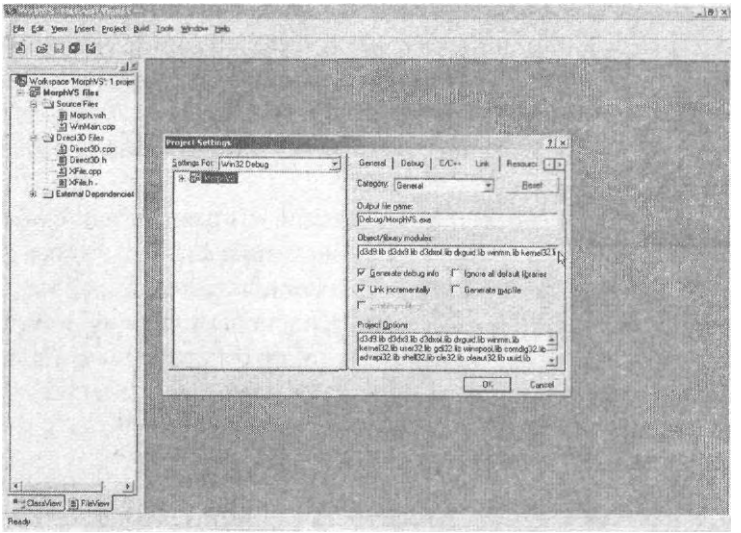


РИС. 1.6. Свойства " Project Settings Link " позволяют непосредственно задать, какие библиотечные файлы привязываются к вашему приложению

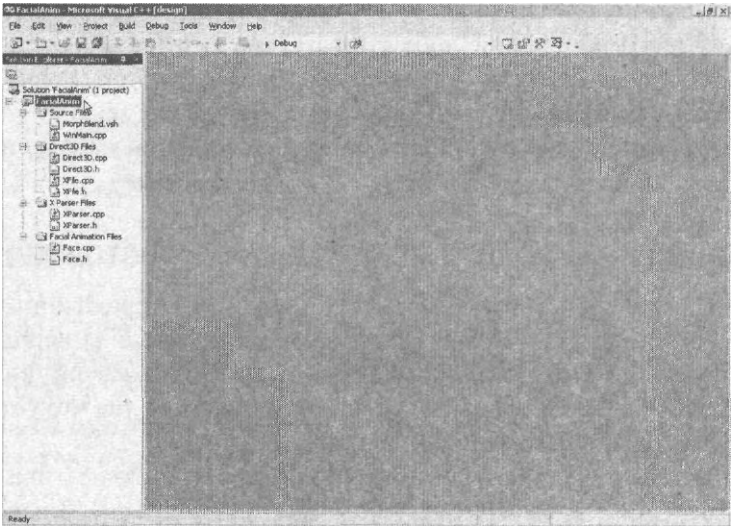


Рис. 1.7. Visual Studio .NET показывает все файлы и проекты, используя Solution Explorer

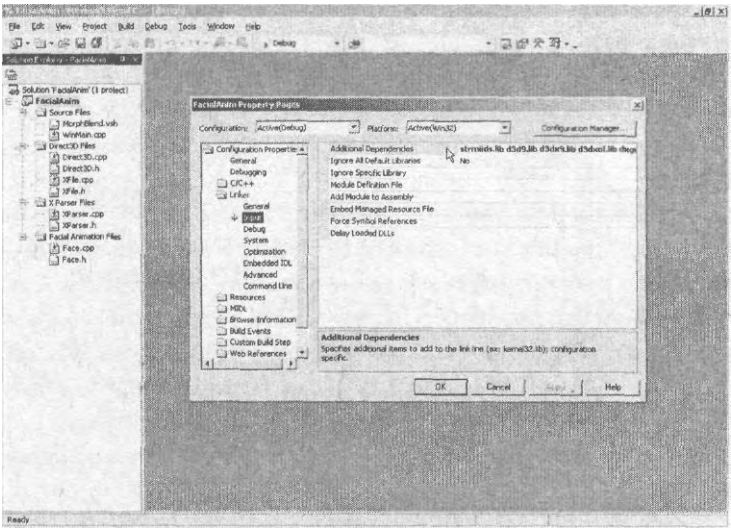


Рис. 1.8. Вы можете изменять настройки связывания в диалоговом окне проекта " Property Pages " наряду с другими настройками

Если вы используете DirectShow для создания анимированных текстур, вам необходимо добавить strmbasd.lib либо strmbase.lib в текстовый блок "Object/Library Modules". Чтобы узнать особенности использования библиотек DirectShow в ваших проектах, читайте главу 14 "Использование анимированных текстур".

Сейчас вы готовы к завершению, установив заданное по умолчанию состояние символа компилятора.

Установка используемого по умолчанию состояния символа

Т. к. я являюсь программистом старой школы, я предпочитаю следовать старыми путями, особенно когда приходится иметь дело с используемым по умолчанию состоянием символьного типа данных. Не знаю как вы, но я использую беззнаковые символьные типы данных чаще, чем знаковые, так что установка этого состояния является приоритетом.

Чтобы установить используемое по умолчанию состояние типа символьных данных в Visual C/C++ 6, выберите "Project" и потом "Settings". Щелкните вкладку C/C++ (как показано на рис. 1.9), выберите "General" ("Основное") из раскрывающегося меню "Category"("Категория") и допишите /J к концу текста

"Project Options". Нажав "OK" вы установите, чтобы по умолчанию использовались беззнаковые символы, когда вы задаете символьную переменную.

Пользователям Visual Studio .NET необходимо выбрать проект в " Solution Explorer " и нажать "Project","Properties". В списке папок выберите C/C++ и щелкните на "Command Line" ("Командная строка"). В текстовом блоке "Additional Options" напишите /J.

И это все, что необходимо сделать, чтобы установить DirectX и ваш компилятор! Я уверен, что теперь для вас инсталляция DirectX - это пустяк, и однажды пройдя этот процесс, вы будете готовы моментально начать кодировать.

Стоп! Прежде чем вы двинетесь дальше, давайте обсудим одну чрезвычайно важную вещь для разработки серьезных программ - создание многократно используемой библиотеки полезных функций для убыстрения процесса создания приложений. Я создал целый набор функций, которые помогут вам пропустить тривиальную работу с DirectX, такую как инициализация режима отображения и загрузка/отрисовка меша[2].

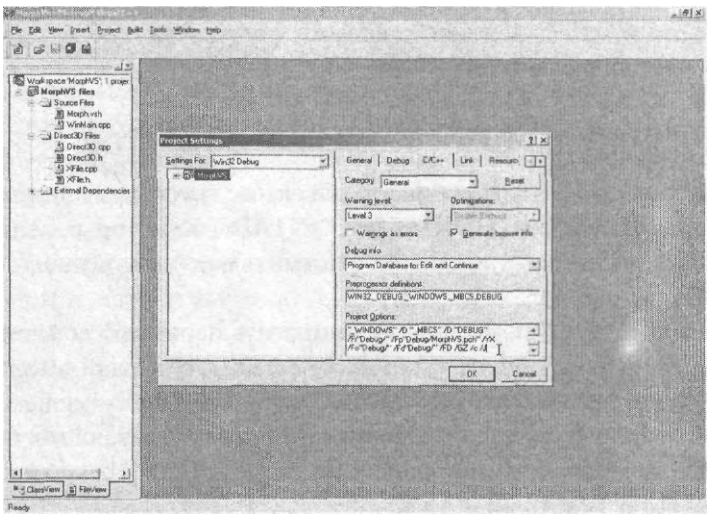


Рис. 1.9. Диалоговое окно " Project Settings " позволяет вам добавлять команды компилятора в текстовый блок "Project Options" наряду с возможностью менять множество других настроек

Давайте поближе познакомимся с созданными мной вспомогательными функциями, их использованием в книге и в ваших проектах.

2. Mesh - меш, трехмерный объект состоящий из полигонов. - *Примеч. науч. ред.*

Использование вспомогательного кода книги

Т. к. это продвинутая книга, я предполагаю, что вы по крайней мере имеете опыт работы с DirectX. Я имею в виду, что вы достаточно опытны, чтобы понять, как Direct3D использует конвейер рендеринга, буферы вершин, текстуры и меши - основы необходимые для создания приложений Direct3D.

Чтобы ускорить разработку и уменьшить время написания тривиального, повторяющегося кода, я создал небольшой набор функций и объектов, которые используются на протяжении всей книги. Эти функции помогают вам иметь дело с теми особенностями Direct3D, которые вы используете чаще всего, такие как инициализация Direct3D, загрузка и визуализация (рендеринг) мешей.

Наряду с этими функциями, я также создал набор объектов, которые расширяют функциональность некоторых объектов D3DX. Эти объекты - расширенные версии объектов D3DXFRAME и D3DXMESHCONTAINER, а также объекты, которые помогут использовать .X файлы в ваших проектах.

Итак, чтобы не делать короткую историю длиннее, чем надо, давайте вернемся к делу и посмотрим, что за вспомогательными функциями и объектами вы располагаете, начав с вспомогательных объектов.

Использование вспомогательных объектов

Как я ранее замечал, я создал набор объектов, которые расширяют функциональность D3DXFRAME и D3DXMESHCONTAINER, которые являются частью библиотеки D3DX. Если вы незнакомы с этими объектами, позвольте мне дать их небольшой обзор.

Объект D3DXFRAME помогает сформировать иерархию ссылок на фреймы . Эти ссылки используются для соединения нескольких мешей вместе, в то время как каждый фрейм производит над присоединенным к нему мешем собственные преобразования. Этот путь использования фреймов, указывающих на меш, позволяет минимизировать количество мешей, т. к. можно делать ссылки на меш вместо того, чтобы перезагружать их.

Например, представьте машину, состоящую их корпуса и четырех колес. Корпус и колеса образуют 2 меша. Эти два меша используются совместно с пятью фреймами (один для корпуса и четыре для колес). При визуализации преобразования каждого фрейма используются для расположения и отрисовки меша исполь-

3. Frame — фрейм, невидимый куб, который предоставляет связи для объектов в сцене. Объекты (меши) могут быть помещены в сцену с указанием их пространственных отношений к существующему фрейму. Видимые объекты получают свои положения и ориентацию из фреймов. - *Примеч. науч. ред.*

зуемого фреймом. Это значит что один фрейм преобразовывает и отрисовывает корпус один раз, в то время как остальные фреймы преобразовывают и отрисовывают меш колеса четыре раза.

Что касается объекта D3DXMESHCONTAINER, он используется чтобы хранить меш и ссылки на серию других мешей (используя связанный список). Вы спросите: почему вместо этого объекта не использовать ID3DXBaseMesh? Функциональность D3DXMESHCONTAINER больше, чем вы можете ожидать. Во-первых, он может хранить любые типы мешей, будь то обычные (regular), скелетные[4] (skinned) или прогрессивные[5] (progressive). Во-вторых, объект D3DXMESHCONTAINER содержит информацию о материалах и данные эффектов.

Для получения полной информации об объектах D3DXMESHCONTAINER и D3DXFRAME обратитесь к документации DirectX SDK. А сейчас давайте посмотрим, как я реализовал расширение функциональности этих объектов (определенных в заголовочном файле \common\Direct3D.h компакт диска этой книги), начав с D3DXFRAME.

Расширение D3DXFRAME

Сам по себе объект D3DXFRAME очень полезен, но, к сожалению, он имеет некоторые недостатки, такие как отсутствие данных для преобразований при анимации меша, функций обрабатывающих данные анимации, конструктора и деструктора.

Чтобы исправить эти недостатки, я создал расширенную версию D3DXFRAME, которую назвал D3DXFRAME_EX. Этот новый объект содержит дополнение в виде двух объектов D3DMATRIX и шести функций. Два объекта матрицы содержат первоначальное преобразование фрейма (прежде чем любое анимационное преобразование наложено) и комбинированное преобразование всех родительских фреймов, к которым данный фрейм присоединен (в иерархии).

Вот как я определил структуру D3DXFRAME_EX, использующую два матричных объекта.

```
struct D3DXFRAME_EX : D3DXFRAME
{
 D3DXMATRIX matCombined; // комбинированная матрица
 D3DXMATRIX matOriginal; // первоначальная матрица
```

4. Skinned mesh - скелетный меш, меш - который обтягивает иерархический скелет из костей. К каждой вершине меша может быть прикреплена одна или несколько костей. Используется для анимирования моделей. - *Примеч. науч. ред.*

5. Progressive mesh - прогрессивный меш, это меш с возможностью динамически определять какое количество вершин и граней визуализировать. То есть возможно гибкое управление детализацией. - *Примеч. науч. ред.*

Вы узнаете об этих двух матрицах и как с ними работать далее в этой книге. На данный момент давайте просто перейдем к рассмотрению функций, начнем с конструктора. Конструктор предназначен для очистки данных структуры (включая изначальные данные базового объекта D3DXFRAME).

```
D3DXFRAME_EX()
{
  Name - NULL;
  pMeshContainer = NULL;
  pFrameSibling = pFrameFirstChild = NULL;
  D3DXMatrixIdentity(&matCombined);
  D3DXMatrixIdentity(&matOriginal);
  D3DXMatrixIdentity(&TransformationMatrix);
}
```

С другой стороны, деструктор предназначен для освобождения данных используемых объектом D3DXFRAME_EX.

```
~D3DXFRAME_EX()
{
  delete [] Name; Name = NULL;
  delete pFrameSibling; pFrameSibling = NULL;
  delete pFrameFirstChild; pFrameFirstChild = NULL;
}
```

Как вы можете видеть, конструктор и деструктор являются типичными - инициализация данных объекта и освобождение ресурсов, когда объект уничтожается. Далее идет набор функций, которые помогут вам найти заданный фрейм в иерархии, вернуть матрицы анимации к их первоначальному состоянию, обновить иерархию, после изменения преобразования и посчитать количество фреймов в иерархии.

Первая функция Find используется для нахождения заданного фрейма в иерархии, возвращая указатель на него. Если вы не знаете, каждый объект D3DXFRAME (и унаследованный от него D3DXFRAMEEX) имеет буфер данных Name, который можно заполнять любым текстом. Обычно фреймы называются так же как и кости, которые определяют иерархию (о чем я расскажу в главе 4 "Работа со скелетной анимацией").

Чтобы найти заданный фрейм (и получить указатель на него), просто вызовите функцию Find, указав имя фрейма, который необходимо найти, в качестве единственного параметра.

```
//Функция просматривает иерархию на совпадения имени фрейма
D3DXFRAME_EX *Find(const char *FrameName)
{
  D3DXFRAME_EX *pFrame, *pFramePtr;
```

```
    //возвращаем этот экземпляр фрейма, если имя совпадает
    if(Name && FrameName && !strcmp(FrameName, Name))
     return this;

    //просмотр родственников
    if((pFramePtr = (D3DXFRAME_EX*)pFrameSibling)) {
     if((pFrame = pFramePtr->Find(FrameName)))

      return pFrame;
    }

    //просмотр потомков
    if((pFramePtr = (D3DXFRAME_EX*)pFrameFirstChild)) {
     if((pFrame = pFramePtr->Find(FrameName)))
      return pFrame;
    }

    //ничего не найдено
    return NULL;
}
```

Функция Find сравнивает переданное в качестве параметра имя с именем текущего фрейма; если они совпадают, возвращается указатель на фрейм. Если совпадений не найдено, тогда просматривается связанный список, используя для этого рекурсивные вызовы Find.

Далее в списке добавленных функций стоит Reset, которая просматривает всю иерархию фреймов (которая, кстати, является связанным списком потомков и родственников). Для каждого найденного фрейма она копирует первоначальное преобразование в текущее. Вот код:

```
//Устанавливает матрицы преобразования в изначальные
void Reset()
{
  //копируем оригинальную матрицу
  TransformationMatrix = matOriginal;

  //сбрасываем родственные фреймы
  D3DXFRAME_EX *pFramePtr;
  if((pFramePtr = (D3DXFRAME_EX*)pFrameSibling))
    pFramePtr->Reset();

  //сбрасываем дочерние фреймы
  if((pFramePtr = (D3DXFRAME_EX*)pFrameFirstChild))
    pFramePtr->Reset();
}
```

Обычно Reset используется для восстановления преобразований иерархии фреймов к тому состоянию, которое было при создании или загрузке фреймов. Повторюсь, полная иерархия фреймов описана в главе 4. Чтобы вас не запутывать,

следующая функция в списке - UpdateHierarchy, основное назначение которой - восстановление полного списка преобразований иерархии фреймов после того, как любая из трансформаций изменилась.

Восстановление иерархии является необходимым, чтобы удостовериться, что меш восстановлен или отрисован корректно, после того как вы обновили анимацию. Здесь, опять же, используется скелетная анимация, просто проконсультируйтесь с главой 4 для дополнительной информации. А сейчас, давайте проверим код, который накладывает дополнительную матрицу преобразования на корневой фрейм иерархии.

```
//функция комбинирования матриц в иерархии фреймов
void UpdateHierarchy(D3DXMATRIX *matTransformation = NULL)
{
  D3DXFRAME_EX *pFramePtr;
  D3DXMATRIX matIdentity;

  //  использовать единичную матрицу, если ничего не задано
  if(!matTransformation) {
  D3DXMatrixIdentity(&matIdentity);
  matTransformation = &matIdentity;
  }

  //комбинировать матрицы с заданной
  matCombined = TransformationMatrix *  (*matTransformation);

  //комбинировать с родственными фреймами
  if((pFramePtr =  (D3DXFRAME_EX*)pFrameSibling))
  pFramePtr->UpdateHierarchy(matTransformation);

  //комбинировать с дочерними фреймами
  if((pFramePtr =  (D3DXFRAME_EX*)pFrameFirstChild))
  pFramePtr->UpdateHierarchy(&matCombined);
}
```

При изучении становится очевидна простота кода функции UpdateHierarchy. Более подробно рассмотренная в последующих главах, UpdateHierarchy преобразует собственную матрицу преобразования фрейма (хранимую в matTransformation), используя матрицу, переданную как необязательный параметр функции. Таким образом, фрейм наследует преобразование родительского фрейма в иерархии, т. е. каждая трансформация накладывается на всю иерархию.

Наконец объект D3DXFRAME_EX имеет функцию Count, которая помогает посчитать число фреймов в иерархии. Это становится возможным используя рекурсивные вызовы функции Count для каждого фрейма в связанном списке. Для каждого фрейма, найденного в списке, переменная счетчика (которую вы передаете в качестве параметра) увеличивается. Посмотрите на код функции Count чтобы увидеть что я имею ввиду.

```
void Count(DWORD *Num)
{
  //Проверка на ошибку,
  if(!Num)
    return;

  //увеличить количество фреймов
   (*Num)+=1;

  //Обработать родственные фреймы
  D3DXFRAME_EX *pFrame;
  if((pFrame=(D3DXFRAME_EX*)pFrameSibling))
    pFrame->Count(Num);

  //Обработать дочерние фреймы
  if((pFrame=(D3DXFRAME_EX*)pFrameFirstChild))
  pFrame->Count(Num);

  }
};
```

Это все в значительной степени описывает объект D3DXFRAME_EX. Если вы использовали объект D3DXFRAME (а вы должны были, если вы пользователь DX9), тогда все, что я только что показал вам, должно быть достаточно понятно.

Двигаясь дальше, позвольте мне представить вам следующий вспомогательный объект, который расширяет функциональность D3DXMESHCONTAINER.

Расширение D3DXMESHCONTAINER

Принимая во внимание то, что вы могли использовать объект ID3DXMesh, чтобы хранить данные ваших мешей, вы наверное обнаружили, как сложно хранить материал меша и данные эффектов раздельно. Но это еще не все. Как насчет использования других меш-объектов D3DX таких как ID3DXPMesh и ID3DXSkinMesh? Почему бы не сделать единый меш-объект, который содержал бы все эти типы мешей и данные о материалах?

На самом деле такой объект есть, он называется D3DXMESHCONTAINER! Объект D3DXMESHCONTAINER хранит указатели на данные меша (независимо от используемого типа меша), все материалы и данные эффектов. Он также содержит указатели на буфер смежности (adjacency buffer) и объект-скелетный меш. И как будто этого было недостаточно, D3DXMESHCONTAINER содержит указатели на связанный список меш-объектов.

Что бы я мог сделать, чтобы расширить функциональность уже существующую у D3DXMESHCONTAINER спросите вы? Ну, с одной стороны D3DXMESHCONTAINER не имеет конструктора и деструктора. Также отсутствуют данные текстур - есть только буфер, содержащий их имена, используемые мешем. Наконец, нет поддержки хранения анимационных данных скелетного меша.

Нет проблем, потому что расширение D3DXMESHCONTAINER очень просто! Новая версия, которую я назвал D3DXMESHCONTAINER_EX, добавляет всего четыре новых объекта данных и три функции. Объекты данных включают в себя массив объектов-текстур, объект-скелетный меш (для хранения анимированного скелетного меша) и два массива матриц.

Вот как я определил обьект D3DXMESHCONTAINER_EX и четыре ранее упомянутые переменные.

```
struct D3DXMESHCONTAINER_EX : D3DXMESHCONTAINER
{
  IDirect3DTexture9 **pTextures;
  ID3DXMesh *pSkinMesh;
  D3DXMATRIX **ppFrameMatrices;
  D3DXMATRIX *pBoneMatrices;
```

Массив указателей pTexture содержит объекты-текстуры, используемые для визуализации меша. Я создаю массив pTexture, сначала загружая меш, а потом получая текстурные буфера (D3DXMESHCONTAINER::pMaterials) для имен файлов используемых текстур.

Что касается pSkinMesh, он требуется, только если вы используете скелетный меш (который я рассмотрю в главах 4-7). Видите ли, когда загружается скелетный меш, сами данные меша хранятся в D3DXMESHCONTAINER::MeshData::pMesh. Единственная проблема в том, что необходим еще один меш контейнер для хранения анимированного скелетного меша. Для этой цели и служит pSkinMesh.

Наконец, вы обнаружите массивы матриц ppFrameMatrices and pBoneMatrices. Не углубляясь, они тоже используются для скелетного меша, и эти массивы рассматриваются в главе 4. В данный момент важно понять, что скелетный меш анимируется прикреплением вершин меша к основной иерархии костей. Таким образом, вместе с костями двигаются и вершины, ppFrameMatrices и pBoneMatrices используются для связки вершин с костями.

Кроме переменных в D3DXMESHCONTAINER_EX есть еще несколько функций. Первые две - конструктор и деструктор:

```
D3DXMESHC0NTAINER_EX()
{
  Name = NULL;
  MeshData.pMesh = NULL;
  pMaterials = NULL;
  pEffects = NULL;
  NumMaterials = 0;
  pAdjacency = NULL;
  pSkinInfo = NULL;
  pNextMeshContainer = NULL;
  pTextures = NULL;
  pSkinMesh = NULL;
  ppFrameMatrices = NULL;
  pBoneMatrices = NULL;
}

~D3DXMESHCONTAINER_EX()
{
  if(pTextures && NumMaterials) {
    for(DWORD i=0;i<NumMaterials;i++)
     ReleaseCOM(pTextures[i]);
  }
  delete [] pTextures; pTextures = NULL;
  NumMaterials = 0;

  delete [] Name; Name = NULL;
  delete [] pMaterials; pMaterials = NULL;
  delete pEffects; pEffects = NULL;

  delete [] pAdjacency; pAdjacency = NULL;
  delete [] ppFrameMatrices; ppFrameMatrices = NULL;
  delete [] pBoneMatrices; pBoneMatrices = NULL;

  ReleaseCOM(MeshData.pMesh);
  ReleaseCOM(pSkinInfo);
  ReleaseCOM(pSkinMesh);

  delete pNextMeshContainer; pNextMeshContainer = NULL;
}
```

Конструктор и деструктор предназначены для инициализации и освобождения данных используемых объектом, соответственно. Обратите внимание на использование макроса ReleaseCOM, который я опишу в следующем разделе "Проверка вспомогательных функций". Вкратце, ReleaseCOM - макрос, который безопасно освобождает COM-интерфейс и устанавливает указатель интерфейса в NULL.

Третья функция в D3DXMESHCONTAINER_EX - Find, которая позволяет просматривать связанные списки мешей для поиска указанного меша, совсем как D3DXFRAME_EX::Find. Используются быстрее сравнение строк для проверки имен и рекурсивные вызовы Find для поиска по всему списку.

```
D3DXMESHC0NTAINER_EX *Find(char *MeshName)
{
  D3DXMESHC0NTAINER_EX *pMesh, *pMeshPtr;

  //возвращаем этот меш, если имя совпало
  if(Name && MeshName && !strcmp(MeshName, Name))
    return this;

  //просматриваем список далее
  if((pMeshPtr = (D3DXMESHCONTAINER_EX*)pNextMeshContainer)) {
    if((pMesh = pMeshPtr->Find(MeshName)))
      return pMesh;

  }

  //результат не найден
  return NULL;
};
```

Вот и все с вспомогательными объектами! Объекты D3DXFRAME_EX и D3DXMESHCONTAINER_EX чрезвычайно полезны, когда имеешь дело с Direct3D; поэтому вам необходимо уделить им как можно больше времени, чтобы привыкнуть к ним. Я думаю, они вам пригодятся и в собственных проектах.

Кроме вспомогательных объектов, есть ещё множество вспомогательных функций, которые мне хотелось бы представить вашему вниманию и которые должны помочь вам в решении тривиальных задач, часто встречающихся в проектах, использующих Direct3D.

Проверка вспомогательных функций

Вспомогательные функции, которые я решил сделать для этой книги, немногочисленны, но они представляют собой основной код, который вы будете использовать в своих проектах. Эти функции предназначены для освобождения интерфейсов COM, инициализации Direct3D, загрузки мешей и вершинных шейдеров, обновления скелетных мешей, визуализации мешей, используя стандартные методы и вершинные шейдеры.

Позвольте мне начать с функции (или даже макроса), которую вы можете использовать для освобождения ваших COM интерфейсов.

Освобождение COM интерфейсов

Первым в наборе вспомогательных функций (которые хранятся в файле \common\Direct3D.cpp компакт диска) является макрос ReleaseCOM, который может быть использован для безопасного освобождения COM интерфейсов в вашем проекте, даже если эти объекты не допустимы (указатели NULL).

```
#define ReleaseCOM(x) { if(x!=NULL) x->Release(); x=NULL; }
```

Функция ReleaseCOM использует один параметр - указатель на COM интерфейс, который вы хотите безопасно освободить. Например, следующий кусочек кода демонстрирует, как загрузить и освободить текстуру, используя макрос ReleaseCOM:

```
IDirect3DTexture9 *pTexture = NULL;
D3DXCreateTextureFromFile(pDevice, "texture.bmp", &pTexture);
ReleaseCOM(pTexture);
```

В этом примере макрос ReleaseCOM освобождает интерфейс IDirect3DTexture9 и устанавливает указатель pTexture обратно в NULL. Даже если текстура не смогла загрузиться и pTexture является NULL, вызов ReleaseCOM не повлечет за собой никаких неблагоприятных эффектов.

Следующая вспомогательная функция помогает инициализировать Direct3D.

Инициализация Direct3D

Следующая вспомогательная функция InitD3D используется для инициализации Direct3D, создания 3D устройства и окна отображения. Я постарался сделать код как можно проще, применив стандартный код инициализации, который используется во всех приложениях Direct3D, но чтобы функция работала со всеми остальными примерами в этой книге, я добавил немного дополнений.

Для выполнения функция InitD3D использует пять параметров (и стандартный возвращаемый тип COM HRESULT), как показано в прототипе функции:

```
HRESULT InitD3D(IDirect3D9 **ppD3D,
        IDirect3DDevice9 **ppD3DDevice,
        HWND hWnd,
        BOOL ForceWindowed = FALSE,
        BOOL MultiThreaded = FALSE);
```

После просмотра кода функции вам станет понятно назначение каждого параметра. Вначале идет несколько переменных, используемых во всей функции:

```
HRESULT InitD3D(IDirect3D9 **ppD3D,
        IDirect3DDevice9 **ppD3DDevice,
        HWND hWnd,
        BOOL ForceWindowed,
        BOOL MultiThreaded)
{
  IDirect3D9 *pD3D = NULL;
  IDirect3DDevice9 *pD3DDevice = NULL;
  HRESULT hr;
```

В этом кусочке кода вы видите локальные объекты IDirect3D9 IDirect3DDevice9, используемые для инициализации Direct3D и 3D устройства. Эти переменные позже сохраняются в ppD3D и ppD3Ddevice, которые указаны как параметры функции. Наконец переменная HRESULT содержит коды, возвращаемые функциями Direct3D. Если любая из этих функций возвратит ошибку (которая определяется с помощью макроса FAILED), результирующий код возникшей ошибки возвращается вызывателю InitD3D.

Первое, что делает функция InitD3D, - проверяет, были ли переданы корректные указатели на объекты Direct3D, 3D устройства и дескриптор окна. Если проверка оказывается неудачной, то InitD3D возвращает ошибку. После чего для создания интерфейса Direct3D вызывается функция Direct3DCreate9, результат работы которой сохраняется в ppD3D (параметр InitD3D).

```
//Проверка ошибки
if(!ppD3D || !ppD3DDevice || !hWnd)
  return E_FAIL;

//инициализация Direct3D
if((pD3D = Direct3DCreate9(D3D_SDK_VERSION)) == NULL)
  return E_FAIL;
*ppD3D = pD3D;
```

Пока ничего экстраординарного. Однако то, что будет дальше, может заставить вас задуматься. Демонстрационные программы, содержащиеся на компакт-диске этой книги, дают вам возможность запускать их в оконном либо полноэкранном режиме. Иногда программы должны выполняться в оконном режиме, поэтому, чтобы удостовериться, что демонстрационная программа запущена в оконном режиме, в прототипе функции InitD3D есть флаг ForceWindowed. Когда флаг установлен в FALSE (ложь), пользователя спросят, запускать ли программу в полноэкранном режиме. Если ForceWindowed установлен в TRUE (истина), пользователю не будет задано никаких вопросов, и функция InitD3D будет предполагать, что был выбран оконный режим.

Следующий кусочек кода тестирует флаг ForceWindowed; если он установлен в FALSE, функция отобразит сообщение и подождет пока пользователь выберет либо оконный либо полноэкранный режим. Если флаг ForceWindowed установлен в TRUE или если пользователь выберет оконный режим, Mode принимает значение IDNO; иначе Mode принимает значение IDYES, обозначая, что приложение запустится в полноэкранном режиме.

```
//Спросить хочет ли пользователь запустить программу в оконном, или
//полноэкранном режиме, или перейти в оконный, если флаг установлен
int Mode;
if(ForceWindowed == TRUE)
  Mode = IDNO;
else
  Mode = MessageBox(hWnd, \
         "Use fullscreen mode? (640x480x16)", \
         "Initialize D3D", \
        MB_YESNO | MB_ICONQUESTION);
```

Теперь, если пользователь выбрал полноэкранный видео режим (я использую 640x480x16), устанавливаем соответствующие параметры, используя стандартные методы, как показано в примерах DX SDK.

```
//Установить видеорежим (в зависимости от оконного или
полноэкранного)
D3DPRESENT_PARAMETERS d3dpp;
ZeroMemory(&d3dpp, sizeof(d3dpp));

//Установить видео настройки, зависящие от того, полноэкранный
// режим или нет
if(Mode == IDYES) {

  ////////////////////////////////////////////////////////
  //Установить полноэкранный формат (задайте свой если хотите)
  ////////////////////////////////////////////////////////
  DWORD Width = 640;
  DWORD Height =480;
  D3DFORMAT Format = D3DFMT_R5G6B5;

  //Установить параметры отображения (для полноэкранного режима)
  d3dpp.BackBufferWidth = Width;
  d3dpp.BackBufferHeight = Height;
  d3dpp.BackBufferFormat = Format;
  d3dpp.SwapEffect = D3DSWAPEFFECT_FLIP;
  d3dpp.Windowed = FALSE;
  d3dpp.EnableAutoDepthStencil = TRUE;
  d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
  d3dpp.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;
  d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_DEFAULT;
} else {
```

Если пользователь выбрал использование оконного видеорежима или флаг ForceWindowed был установлен в TRUE, тогда используется оконный режим вместо полноэкранного. Перед тем как выполнять настройки отображения, как это делалось для полноэкранного режима, клиентская область окна должна быть изменена до 640x480. После этого устанавливаются параметры отображения, как для обычного оконного приложения.

```
    ////////////////////////////////////////////////////
    //Установка оконного формата (установите собственные размеры)
    ////////////////////////////////////////////////////////////

    // Получить размеры окна и клиентской области окна
    RECT ClientRect, WndRect;
    GetClientRect(hWnd, &ClientRect);
    GetWindowRect(hWnd, &WndRect);

    //установить ширину и высоту(установите тут свои размеры)
    DWORD DesiredWidth = 640;
    DWORD DesiredHeight = 480;
    DWORD Width = (WndRect.right - WndRect.left) + \
                  (DesiredWidth - ClientRect.right);
    DWORD Height = (WndRect.bottom - WndRect.top) + \
                   (DesiredHeight - ClientRect.bottom);

    //установка размеров окна
    MoveWindow(hWnd, WndRect.left, WndRect.top, \
               Width, Height, TRUE);

    //Получение формата рабочего стола
    D3DDISPLAYMODE d3ddm;
    pD3D->GetAdapterDisplayMode(D3DADAPTER_DEFAULT,   &d3ddm);

    //Установка параметров отображения (используются оконные)
    d3dpp.BackBufferWidth = DesiredWidth;
    d3dpp.BackBufferHeight = DesiredHeight;
    d3dpp.BackBufferFormat = d3ddm.Format;
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.Windowed = TRUE;
    d3dpp.EnableAutoDepthStencil = TRUE;
    d3dpp.AutoDepthStencilFormat = D3DFMT_D16;
    d3dpp.FullScreen_RefreshRateInHz = D3DPRESENT_RATE_DEFAULT;
    d3dpp.PresentationInterval = D3DPRESENT_INTERVAL_DEFAULT;
}
```

На данный момент вы готовы инициализировать 3D устройство, используя функцию IDirect3D::CreateDevice. В следующем коде вызывается :CreateDevice, в качестве параметра указывая флаг D3DCREATE_MIXED_VERTEXPROCESSING и наряду с ним флаг D3DCREATE_MULTITHREADED, если вы установили MultiThreaded в TRUE при вызове InitD3D.

```
//Создание 3D устройства
DWORD Flags= D3DCREATE_MIXED_VERTEXPROCESSING;
if(MultiThreaded == TRUE)
  Flags |= D3DCREATE_MULTITHREADED;
if(FAILED(hr = pD3D->CreateDevice(
          D3DADAPTER_DEFAULT,
          D3DDEVTYPE_HAL, hWnd, Flags,
          &d3dpp, &pD3DDevice)))
  return hr;
```

```
//Сохранить указатель на 3D устройство
*ppD3DDevice = pD3DDevice;
```

В конце последнего кусочка кода вы можете заметить, что я сохранил результирующий указатель 3D устройства в ppD3Ddevice, который является параметром InitD3D. Далее все просто - вы устанавливаете матрицу проецирования, используя D3DXMatrixPerspectiveFovLH для расчета преобразования и IDirect3DDevice9::SetTransform для его установки.

```
//Установка перспективного проецирования
float Aspect = (float)d3dpp.BackBufferWidth / (float)d3dpp.Back-
BufferHeight;
D3DXMATRIX matProjection;
D3DXMatrixPerspectiveFovLH(&matProjection, D3DX_PI/4.0f, Aspect,
1.0f, 10000.0f);
pD3DDevice->SetTransform(D3DTS_PROJECTION, &matProjection);
```

Наконец, в функции InitD3D вы можете установить состояние освещения, z-буфера и прозрачности. Здесь я отключаю освещение, альфа смешивание (alpha-blending), альфа тест (alpha-test), но включаю z-буферизацию. Также по умолчанию состояние текстур установлено в использование модуляции, и выборка текстур установлена на линейную минимизацию/усиление.

```
//Установка состояния визуализации по умолчанию
pD3DDevice->SetRenderState(D3DRS_LIGHTING,   FALSE);
pD3DDevice->SetRenderState(D3DRS_ZENABLE, D3DZB_TRUE);
pD3DDevice->SetRenderState(D3DRS_ALPHABLENDENABLE,   FALSE);
pD3DDevice->SetRenderState(D3DRS_ALPHATESTENABLE,   FALSE);

//Установка состояний текстур по умолчанию
pD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG1,
D3DTA_TEXTURE);
pD3DDevice->SetTextureStageState(0, D3DTSS_COLORARG2,
D3DTA_DIFFUSE);
pD3DDevice->SetTextureStageState(0, D3DTSS_COLOROP,
D3DTOP_MODULATE);

//Установка тескстурных фильтров
pD3DDevice->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR);
pD3DDevice->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR);

return S_OK;
}
```

Теперь, после того как вы посмотрели код, как насчет того, чтобы научиться пользоваться функцией InitD3D? Я пытался сделать InitD3D как можно проще, так что следующий кусочек кода должен работать для большинства ваших программ.

```
//Объявление 3D устройства и объекта Direct3D
IDirect3D9 *pD3D = NULL;
IDirect3DDevice9 *pD3DDevice = NULL;

//Инициализация видео режима, спросив, хочет ли пользователь
//полноэкранный или нет
InitD3D(&pD3D, &pD3DDevice, hWnd);
```

По мере того как вы будете изучать код демонстрационных программ этой книги, вы увидите, что большинство из них использует приведенный выше код инициализации Direct3D. Только одно приложение испбльзует многопоточность, а другое запускается в оконном режиме. Вы научитесь использовать InitD3D очень быстро.

Давайте перейдем к следующей вспомогательной функции, которая помогает загружать вершинные шейдеры и устанавливать обьявления вершин.

Загрузка вершинных шейдеров

Изучая список вспомогательных функций, вы обнаружите LoadVertexShader. Для всех примеров использования вершинных шейдеров в этой книге вы будете пользоваться этой функцией для загрузки ваших вершинных шейдеров и для подготовки объявлений ваших вершинных шейдеров.

Посмотрите на прототип функции LoadVertexShader:

```
HRESULT LoadVertexShader(
        IDirect3DVertexShader9 **ppShader,
        IDirect3DDevice9 *pDevice,
        char *Filename,
        D3DVERTEXELEMENT9 *pElements = NULL,
        IDirect3DVertexDeclaration9 **ppDecl = NULL);
```

Сам код функции LoadVertexShader очень мал, так что вместо того чтобы разделять его и объяснять части, я приведу его сразу полностью.

```
HRESULT LoadVertexShader(IDirect3DVertexShader9 **ppShader,
        IDirect3DDevice9 *pDevice,
        char *Filename,
        D3DVERTEXELEMENT9 *pElements,
        IDirect3DVertexDeclaration9 **ppDecl)
{
  HRESULT hr;

  //проверка ошибки
  if(!ppShader || !pDevice || !Filename)
    return E_FAIL;

  //Загрузить и транслировать шейдер
  ID3DXBuffer *pCode;
  if(FAILED(hr=D3DXAssembleShaderFromFile(Filename, NULL, \
```

```
                                                NULL, 0, \
                                                &pCode, NULL)))
    return hr;
  if(FAILED(hr=pDevice->CreateVertexShader( \
            (DWORD*)pCode->GetBufferPointer(), ppShader)))

    return hr;
  pCode->Release();

  //Создать интерфейс объявления, если необходимо
  if(pElements && ppDecl)
    pDevice->CreateVertexDeclaration(pElements,  ppDecl);

  //Возвратить успех
  return S_OK;
}
```

После первой проверки, необходимой для проверки правильности переданных параметров, происходит загрузка и транслирование вершинного шейдера с помощью вызова D3DXAssembleShaderFromFile. Используя объект D3DXBUFFER, возвращенный этой функцией, вы используете функцию IDirect3DDevice::CreateVertexShader для создания объекта вершинного шейдера (указатель к которому храниться в параметре функции LoadVertexShader ppShader).

В конце функции LoadVertexShader вы увидите вызов CreateVertexDeclaration, который используется для создания интерфейса IDirect3DVertexDeclaration9 из заданного массива вершин (pElements в прототипе LoadVertexShader). Указатель объекта объявления вершин хранится в указателе ppDecl, который также является параметром LoadVertexShader.

Для использования функции LoadVertexShader, укажите в качестве ее параметров объект IDirect3DVertexShader9, который вы хотите создать, корректный объект IDirect3DDevice9 и имя файла вершинного шейдера. Последние два параметра (pElements и ppDecl) являются дополнительными. Передав корректный массив D3DVERTEXELEMENT9 и указатель на объект IDirect3DVertexDeclaration9, вы можете подготовить ваше объявление вершин для использования с загруженным вершинным шейдером.

Вот небольшой пример использования LoadVertexShader. Сначала я объявляю массив элементов вершин, которые будут использованы для создания объекта объявления вершин.

```
//Объявить элементы объявления вершинного шейдера
D3DVERTEXELEMENT9 Elements[] =
{
  { 0,  0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, \
           D3DDECLUSAGE_POSITION, 0 },
```

```
    { 0, 12, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, \
             D3DDECLUSAGE_NORMAL, 0 },
    { 0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, \
             D3DDECLUSAGE_TEXCOORD, 0 },
  D3DDECL_END()
};
```

Потом я объявляю объекты вершинного шейдера и объявления вершин, после чего вызываю LoadVertexShader

```
//Экземпляры объектов
IDirect3DVertexShader9 *pShader = NULL;
IDirect3DVertexDeclaration9 *pDecl = NULL;

//Загрузить вершинный шейдер и создать интерфейс объявлений
LoadVertexShader(&pShader, pDevice, \
                 "Shader.vsh", \
                 &Elements, &pDecl);
```

Как вы видите, эта функция быстро и легко выполняет всю работу. После этого вы можете установить вершинный шейдер (ассоциированный с pShader), используя функцию IDirect3DDevice9::SetVertexShader. Для установки объявления вершин (ассоциированный с pDecl) необходимо вызвать IDirect3DDevice9::SetVertexShader.

```
//Очистить FVF использование
pD3DDevice->SetFVF(NULL);
pD3DDevice->SetVertexShader(pShader);
pD3DDevice->SetVertexDeclaration(pDecl);
```

Ну хорошо, достаточно инициализации и загрузки вершинных шейдеров, давайте двигаться к вещам поинтересней, как например загрузка и визуализация мешей.

Загрузка мешей

Первая из вспомогательных функций, посвященная мешам, - LoadMesh. На самом деле существует три версии функции LoadMesh. Первая версия используется для загрузки мешей из файла .X с помощью функции D3DXLoadMeshFromX. Это значит, что все меши, находящиеся в файле, объединяются в один меш-обьект, который впоследствии хранится в D3DXMESHCONTAINER_EX.

Все варианты функции LoadMesh содержат указатели на корректный объект 3D устройства, путь к директории, где хранятся текстуры меша, и дополнительный гибкий формат вершин, который функция LoadMesh использует для клонирования мешей после загрузки. Это значит, что загруженный меш может иметь заданный формат вершин!

Вот прототип первой функции LoadMesh:

```
HRESULT LoadMesh(D3DXMESHC0NTAINER_EX **ppMesh,
        IDirect3DDevice9 *pDevice,
        char *Filename,
        char *TexturePath = ".\\",
        DWORD NewFVF = 0,
        DWORD LoadFlags = D3DXMESH_SYSTEMMEM);
```

Первая функция LoadMesh использует указатель на указатель обьекта D3DXMESHCONTAINER_EX, который вы хотите использовать для хранения загруженных данных меша. Заметьте, я сказал указатель на указатель. Функция LoadMesh сама создает нужный объект и сохраняет указатель на него в указателе-параметре функции. Этот подход сходен с тем, как функция InitD3D хранит указатели на объекты Direct3D и 3D устройства.

Также, функции LoadMesh необходимо передавать корректный объект 3D устройства (в качестве указателя pDevice) - это устройство используется для создания меш-объекта и для текстурных буферов. Имя загружаемого меша задается в параметре Filename, директория, в которой находятся текстуры, - TexturePath. Этот путь к директории текстур является префиксом к именам всех загружаемых текстур.

Наконец, есть еще 2 параметра - NewFVF и LoadFlags. Параметр NewFVF используется для вынуждения меша загрузиться с использованием заданного FVF. Например, если бы вы хотели использовать только трехмерные координаты и нормали, то вы бы установили NewFVF в (D3DFVF_XYZ|D3DFVF_NORMAL). Функция LoadMesh использует CloneMeshFVF для копирования меша, применяя заданный вами FVF. Параметр LoadFlag, определяемый в соответствии с документацией DX SDK, используется для задания флагов загрузки функции D3DXLoadMeshFromX. По умолчанию он имеет значение D3DXMESH_SYSTEMMEM, загружая тем самым меш в системную память (в противоположность аппаратной).

Вместо того чтобы показывать полный код функции LoadMesh, я приведу только самые важные части. Для просмотра полного кода первой функции LoadMesh изучите файл Direct3D.cpp, поставляемый с книгой.

Первая LoadMesh-функция использует D3DXLoadMeshFromX как и большинство функций загрузки меша, которые вы, наверное, использовали, как показано далее:

```
//Загрузка меша используя подпрограммы D3DX
ID3DXBuffer *MaterialBuffer = NULL, *AdjacencyBuffer = NULL;
DWORD NumMaterials;
if(FAILED(hr=D3DXLoadMeshFromX(Filename, TempLoadFlags, \
             pDevice, &AdjacencyBuffer, \
             &MaterialBuffer, NULL, \
             &NumMaterials, &pLoadMesh)))

  return hr;
```

Несколько примечаний относительно параметров функции D3DXLoadMeshFromX:

- Filename указывает имя загружаемого .X файла;
- pDevice - это ваш объект 3D устройства;
- AdjacencyBuffer - это объект ID3DXBUFFER, который будет содержать данные смежных граней;

  MaterialBuffer хранит данные материалов (где NumMaterials содержит число загруженных мешей);
- pLoadMesh будет хранить загруженный объект ID3DXMesh.

В предыдущем списке не были упомянуты флаги TempLoadFlags, которые используются для загрузки меша. Если вы используете новый FVF (как указано в NewFVF), тогда меш помещается в системную память, используя флаг D3DXMESHSYSMEMORY (для успешного копирования); иначе меш загружается, используя заданные в LoadFlags-функции LoadMesh-параметры.

Говоря о клонировании меша, если вы зададите ненулевое значение NewFVF, функция LoadMesh попробует скопировать меш в заданном FVF формате. Если клонирование произойдет успешно, его результат заменит указатель pLoadMesh скопированным мешем. Вот маленький пример кода, который демонстрирует возможность копирования:

```
// Сначала конвертируем в новый FVF, если необходимо
if(NewFVF) {
  ID3DXMesh *pTempMesh;

  // Используем CloneMeshFVF для преобразования меша
  if(FAILED(hr=pLoadMesh->CloneMeshFVF(LoadFlags, NewFVF, pDevice,
&pTempMesh))) {
    ReleaseCOM(AdjacencyBuffer);
    ReleaseCOM(MaterialBuffer);
    ReleaseCOM(pLoadMesh);
    return hr;
  }

  // Освободить первоначальный меш и сохранить новый указатель
  ReleaseCOM(pLoadMesh);
  pLoadMesh = pTempMesh; pTempMesh = NULL;
}
```

Кроме вызовов типичной функции загрузки материалов, с которой вы уже должны были познакомиться при загрузке мешей, используя D3DX, у функции LoadMesh существуют два важных аспекта - создание структуры D3DXMESHCONTAINER_EX (с последующим ее заполнением необходимыми данными) и оптимизация мешповерхностей.

Вспомните, что ранее в этой главе я упоминал, что структура D3DXMESHCONTAINEREX содержит разнообразную информацию о меше, такую как его имя, указатель на меш-объект, его тип (обычный, скелетный, прогрессивный), данные смежности граней, данные текстуры и материалов. Следующий код демонстрирует создание объекта D3DXMESHCONTAINER_EX и его инициализацию.

```
// Создание структуры D3DXMESHCONTAINER_EX
D3DXMESHCONTAINER_EX *pMesh = new D3DXMESHCONTAINER_EX();
*ppMesh = pMesh;

//Сохранение имени меша (имя файла), типа и указателя
pMesh->Name = strdup(Filename);
pMesh->MeshData.Type = D3DXMESHTYPE_MESH;
pMesh->MeshData.pMesh = pLoadMesh; pLoadMesh = NULL;

//Сохранение буфера смежности
DWORD AdjSize = AdjacencyBuffer->GetBufferSize();
if(AdjSize) {
  pMesh->pAdjacency = new DWORD[AdjSize];
  memcpy(pMesh->pAdjacency, \
         AdjacencyBuffer->GetBufferPointer(),  AdjSize);
 }
ReleaseCOM(AdjacencyBuffer);
```

На данный момент данные материала загружаются в объект D3DXMESHCONTAINER_EX. (Я не буду приводить код, т. к. это тривиальная задача.) Опять же вы можете проконсультироваться с полным исходным кодом и увидеть, что я просто загружаю материалы и текстуру; используя приемы, которые вы видели уже миллионы раз в демонстрационных программах DX SDK.

Оптимизация граней меша, о которой я упоминал ранее, является важным шагом для того, чтобы быть уверенным, что данные граней меша будут доступны вам, если вы захотите визуализировать полигоны меша самостоятельно (а не используя DrawSubset). Глава 8 "Работа с морфирующей анимацией" описывает использование данных граней более детально, так что сейчас я просто покажу вам, как использовать функцию, оптимизирующую грани для вас.

```
// Оптимизировать меш для лучшего доступа
pMesh->MeshData.pMesh->OptimizeInplace( \
       D3DXMESHOPT_ATTRSORT, NULL, NULL, NULL, NULL);
```

Вот и все о первой функции LoadMesh! Давайте посмотрим, как ее использовать. Предположим, вы хотите загрузить меш (из файла "Mesh.x"), используя только что рассмотренную функцию LoadMesh. Чтобы продемонстрировать возможность задавать новый FVF, зададим, что нам необходимо использовать компоненты XYZ,

нормали и текстурные координаты. Также предположим, что текстуры расположены в поддиректории "\textures". Флаги загрузки меша задавать не будем, так что меш загрузится в системную память (как установлено во флагах по умолчанию в прототипе функции). Вот код:

```
//Экземпляр меш объекта
D3DXMESHCONTAINER_EX *Mesh = NULL;

// Загрузка меша - обратите внимание, указатель на меш
LoadMesh(&Mesh, pD3DDevice, "Mesh.x", "..\\WTextures\\", \
        (D3DFVF_XYZ|D3DFVF_NORMAL|D3DFVF_TEX1));
```

После того как меш был загружен, вы можете обращаться к его объекту, используя указатель Mesh->MeshData.pMesh. Кроме того, данные о материалах хранятся в Mesh->pMaterials, а о текстурах в Mesh->pTextures. Число материалов, используемых мешем, хранится в Mesh->NumMaterials. Чтобы визуализировать загруженный меш, можно использовать следующий код:

```
// pMesh = указатель на объект D3DXMESHCONTAINER_EX

// Перебрать все поднаборы материалов
for(DWORD i=0;i<pMesh->NumMaterials;i++) {

  //Установить материал и текстуру
  pD3DDevice->SetMaterial(&pMesh->pMaterials[i].MatD3D);
  pD3DDevice->SetTexture(0, pMesh->pTextures[i]);

  //Нарисовать поднабор меша
  pDrawMesh->DrawSubset  (i);
}
```

Вторая функция LoadMesh, используемая в этой книге, очень похожа на первую, за исключением того, что весь .X файл загружается не в один объект D3DXMESHCONTAINER_EX, а предусмотрена возможность загрузки отдельных меш-обьектов (используя функцию D3DXLoadSkinMeshFromXof), как указано объектом IDirectXFileDataObject (используемом во время загрузки .X файла). Вот прототип:

```
HRESULT LoadMesh(D3DXMESHCONTAINER_EX **ppMesh,
        IDirect3DDevice9 *pDevice,
        IDirectXFileData *pDataObj,
        char *TexturePath = ".\\",
        DWORD NewFVF = 0,
        DWORD LoadFlags = D3DXMESH SYSTEMMEM);
```

Вы заметите, что здесь, в отличие от первой функции, вместо параметра Filename используется pDataObj. pDataObj является параметром-обьектом IdirectXFileData, который представляет текущий найденный обьект в .X файле. Если вы незнакомы с объектами .X файлов, вы можете узнать о них из главы 3 "Использование формата файла .X"

Вызов второй функции LoadMesh производится так же, как и первой (за исключением того, что используется указатель на объект перечисления), так что я пропущу пример использования второй функции и рассмотрю его позже. А пока давайте перейдем к следующей версии функции LoadMesh.

Третья, и последняя версия функции LoadMesh, - самая расширенная. Она позволяет вам загрузить все меши и фреймы из .X файла одновременно. Все меши хранятся в связанном списке (указанном корневым объектом D3DXMESHCONTAINER_EX, который вы передаете в качестве параметра), и в иерархии объектов D3DXFRAME_EX (указанных корневым объектом, переданном как параметр).

Замечание. *Вы заметите, что я упоминал использование функции D3DXLoadSkinMeshFromXof (вместо D3DXLoadMeshFromXof), которая загружает скелетный меш-объект из заданного вами файла .X. Причина в том, что эта функция загружает как обычные меши (без данных скелета), так и скелетные меши, возвращая корректный указатель на объект ID3DXMesh. Вы познакомитесь с скелетными меш объектами в главе 4.*

Вот прототип третьей функции LoadMesh:

```
HRESULT LoadMesh(D3DXMESHCONTAINER_EX **ppMesh,
        D3DXFRAME_EX**ppFrame,
        IDirect3DDevice9 *pDevice,
       char *Filename,
       char *TexturePath = ".\\",
       DWORD NewFVF = 0,
       DWORD LoadFlags = D3DXMESH_SYSTEMMEM);
```

Третья функция требует небольших разъяснений. Для начала, опишем как ее использовать. Также как и в первой функции LoadMesh вам необходимо задать имя .X файла, из которого вы будете загружать данные, указатель на 3D устройство, путь к текстурам, дополнительные флаги FVF и хранения меша. Также необходимо задать указатель на объект D3DXMESHCONTAINER_EX, который после загрузки хранит связанный список meshes.

Новым для функции LoadMesh (в отличие от предыдущих версий) является указатель на D3DXFRAME_EX, который аналогичен объекту D3DXMESHCONTAINER_EX за исключением одного - вместо того чтобы хранить связанный список meshes, в него записывается информация о фреймах, содержащихся в загружаемом .X файле.

Чтобы вызвать третью функцию LoadMesh, необходимо задать параметры аналогично как и для первого варианта этой функции. Единственным исключением будет добавление указателя на объект D3DXFRAMEEX.

```
//Корневые загружаемые меш-объекты и фреймы
D3DXMESHCONTAINER_EX *pMesh = NULL;
D3DXFRAME_EX *pFrame = NULL;

//Загрузить meshes и фреймы из файла "Mesh.x"
LoadMesh(&pMesh, &pFrame, pD3DDevice, "Mesh.x");
```

После завершения pMainMesh будет содержать указатель на корневой меш-объект (в связанном списке меш-объектов), а pFrame указывать на корневой фрейм иерархии. Для получения дополнительной информации о иерархиях читайте главу 4.

Вы можете также использовать функцию Loadmesh для загрузки нескольких мешей из .X файла, не создавая никаких фреймов. Это можно сделать, указав значение NULL в качестве указателя на фрейм. С другой стороны вы можете вынудить LoadMesh не загружать меш совсем, установив указатель меш в NULL, при вызове функции.

Эта возможность решать, какие компоненты загружать (меш и/или фреймы), делает третий вариант функции LoadMesh самым полезным, но и самым трудным для понимания. Фактически очень трудно объяснить механизм ее работы на данном этапе. Вы увидите, что третья функция LoadMesh использует специальный анализатор .X файлов, созданный с использованием методов, рассмотренных в главе 3.

Вкратце, специализированный анализатор .X файлов просматривает заданный файл для поиска данных меша и фреймов. Как только эти данные найдены анализатор решает - загружать ли меш, используя вторую версию функции LoadMesh (загрузка, используя объект IDirectXFileData), или загружать фрейм (и его матрицу преобразования, если она есть). В любом случае если используются данные, то создается необходимая структура (D3DXMESHCONTAINER_EX или D3DXFRAME_EX) и привязывается к соответствующему связанному списку объектов. Указатель на эти связанные списки возвращается, используя параметры функции LoadMesh (как было рассмотрено ранее).

Как только вы прочитаете об использовании фреймов и специальных анализаторов .X файлов далее в этой книге, вы поймете, что на самом деле третья функция LoadMesh достаточно проста и, опять же, она должна стать для вас наиболее часто используемой функцией для загрузки меша.

Вот и все вспомогательные функции для загрузки меша! После неприятностей с загрузкой мешей пришло время визуализировать их для отображения, но сначала нам необходимо обновить данные вершин, если мы используем скелетные меши. Следующая вспомогательная функция помогает обновить скелетный меш (о которой будет написано в главе 4) очень проста. Посмотрите сами.

Обновление скелетных мешей

Скелетный меш работает так: каждая вершина присоединена к воображаемой кости (которая задана объектом-фреймом). По мере того как двигаются эти фреймы, присоединенные к ним вершины тоже двигаются. Чтобы обновить координаты вершин после движения костей, необходимо вызвать специальную функцию, которая преобразует данные вершин в соответствии с преобразованием костей и сохраняет результат работы в другом меш-обьекте . Эта специальная функция называется ID3DXSkinInfo::UpdateSkinnedMesh.

Когда вы загружаете меш, используя функцию D3DXLoadSkinMeshFromXof (как во второй функции LoadMesh), вы получаете указатель на обьект ID3DXSkinInfo. Этот обьект содержит информацию о том, как вершины соединены с костями. Таким образом, объект знает, какие преобразования к каким вершинам применять.

Перед тем как обновить вершины вам необходимо заблокировать вершинный буфер меша (который содержит начальные координаты вершин) и вершинный буфер меш-назначения. Mesh назначения получит обновленные вершины после их преобразования. После блокирования вам необходимо вызвать UpdateSkinnedMesh, задав несколько матриц преобразования костей (сохраненных в объекте D3DXMATRIX).

Вы поймете все это после работы со скелетными мешами в главе 4. А сейчас просто посмотрите код функции UpdateMesh, чтобы узнать как она работает.

```
HRESULT UpdateMesh(D3DXMESHCONTAINER_EX *pMesh)
{
  //проверка ошибок
  if(!pMesh)
    return E_FAIL;
  if(!pMesh->MeshData.pMesh || !pMesh->pSkinMesh || !pMesh->pSkin-
Info)
    return E_FAIL;
  if(!pMesh->pBoneMatrices || !pMesh->ppFrameMatrices)
    return E_FAIL;

  //Скопировать матрицы костей (должны быть скомбинированы перед
  //вызовом DrawMesh)
  for(DWORD i=0;i<pMesh->pSkinInfo->GetNumBones();i++) {
    // Начать с матрицы смещения кости
   pMesh->pBoneMatrices[i]=(*pMesh->pSkinInfo->GetBoneOffsetMatrix(i));
```

Кроме типичного кода проверки ошибок функция UpdateModel сначала просматривает каждую кость, содержащуюся в объекте ID3DXSkinInfo (хранящемся в объекте D3DXMESHCONTAINER_EX, который вы уже загрузили). Для каждой кости из .X файла берется первоначальная матрица преобразования и сохраняется в массиве матриц, используемых в вызове UpdateSkinnedMesh.

Далее трансформация костей, хранимая в соответствующем фрейм-объекте, накладывается на матрицу преобразований. Этот процесс продолжается, пока все матрицы преобразований не будут установлены.

```
    //Наложить преобразование фрейма
    if(pMesh->ppFrameMatrices[i])
      pMesh->pBoneMatrices[i] *= (*pMesh->ppFrameMatrices[i]);
  }
```

Здесь мы уже готовы заблокировать буфер вершин и вызвать функцию UpdateSkinnedMesh.

```
    // Заблокировать буфер вершин меша
    void *SrcPtr, *DestPtr;
    pMesh->MeshData.pMesh->LockVertexBuffer(D3DLOCK_READONLY, \
                          (void**)&SrcPtr);
    pMesh->pSkinMesh->LockVertexBuffer(0, (void**)&DestPtr);

    // Обновить скелетный меш, используя предоставленное преобразование
    pMesh->pSkinInfo->UpdateSkinnedMesh(pMesh->pBoneMatrices, \
                       NULL, SrcPtr, DestPtr);
```

Функция завершается разблокированием буферов и возвращением кода успешного завершения.

```
    //Разблокировать буфера вершин меша
    pMesh->pSkinMesh->UnlockVertexBuffer();
    pMesh->MeshData.pMesh->UnlockVertexBuffer();

    //Возвратить успех
    return S_OK;
  }
```

Я немного умолчал об особенностях, но это не играет большой роли, пока вы не прочитаете главу 4. После того как вы ее прочитаете, вы поймете, что функция UpdateMesh легко обновляет скелетные меши и подготавливает их к визуализации. .

И еще раз говоря о визуализации, наконец-то пришло время рассказать о вспомогательных функциях, которые я сделал для помещения этих мешей на экран.

Визуализация меша

После того как вы загрузили меш и обновили те скелетные меши, которые этого требовали, пришло время поместить пиксели на экран и наконец-то показать эти меши! Всего я создал четыре функции визуализации меша, и с помощью их я отрисовываю меш в книге.

Вот прототипы четырех функций визуализации мешей, используемых в этой книге:

```
// нарисовать первый меш в связанном списке объектов
HRESULT DrawMesh(D3DXMESHCONTAINER_EX *pMesh);

// нарисовать первый меш в связанном списке объектов
// используя заданный вершинный шейдер и объявления
HRESULT DrawMesh(D3DXMESHCONTAINER_EX *pMesh,
        IDirect3DVertexShader9 *pShader,
        IDirect3DVertexDeclaration9  *pDecl);

// нарисовать все меши в связанном списке объектов
HRESULT DrawMeshes(D3DXMESHCONTAINER_EX *pMesh);

// нарисовать все меши в связанном списке объектов
// используя заданный вершинный шейдер и объявления
HRESULT DrawMeshes(D3DXMESHCONTAINER_EX *pMesh,
        IDirect3DVertexShader9 *pShader,
        IDirect3DVertexDeclaration9  *pDecl);
```

Вы увидите, что функции отображения мешей очень похожи по сути. Первые две используются для визуализации одного меша, который хранится в объекте D3DXMESHCONTAINEREX. Я говорю один меш, потому что меш-объект может содержать связанный список загруженных меш-объектов. Если вы хотите визуализировать определенный меш, тогда используйте функцию DrawMesh.

Я не буду приводить здесь код для функций DrawMesh, потому что он достаточно прост. Ранее в разделе "Загрузка Mesh" я показал небольшой кусочек кода, который демонстрирует визуализацию меша хранимого в объекте D3DXMESHCONTAINER_EX. Функция DrawMesh повторяет этот код за исключением того, что используется альфа-смешивание. Если используется материал со значением альфа, не равным 1, то тогда разрешается альфа-смешивание. Таким образом, вы можете делать части меша прозрачными, используя информацию о материале. Также если объект D3DXMESHCONTAINER_EX содержит скелетный меш, то он визуализируется вместо стандартного.

Что касается второго варианта функции DrawMesh, то он не использует функцию DrawSubset, а визуализирует набор полигональных граней самостоятельно, используя вершинный шейдер и вершинные объявления, которые вы задаете. Эта функция очень полезна, если вы используете вершинные шейдеры для отрисовки ваших мешей.

Оставшиеся две функции имеют в точности такую же функциональность, как и предыдущие две, за исключением того, что все меши в связанном списке меш-объектов визуализируются. Повторюсь, не стесняйтесь использовать полный исходный код функций DrawMesh и других функций. По мере чтения книги и изучения демонстрационных программ вы увидите, какое применение я нашел этим четырем функциям.

Двигаясь дальше по книге

Гмм! Сколько всего - инсталляция DirectX, настройка компилятора, вспомогательный код - как это все возможно запомнить прежде чем двигаться дальше? Просто не спешите, и все будет в порядке.

Как я уже упоминал, вспомогательные функции не делают ничего особенного за исключением того, что вы проделывали тысячи раз. Если вы почувствуете себя более комфортно, изменяйте вспомогательные функции для своих нужд. Например, если вы имеете свой набор вспомогательных функций, просто используйте их вместо используемых в книге. Или можно просто переписать функции. Я уверен, вам просто не терпится добавить код, который отображал бы полный список доступных видео режимов, из которых пользователь мог бы выбирать в демонстрационных программах, да?

Если у вас что-то не получается, просто вернитесь к этой главе как к справочнику по использованию вспомогательного кода, а если уж у вас совсем ничего не получается, просто напишите мне электронное письмо. Я буду рад помочь вам, чем смогу!

Программы на компакт-диске

В директории \common компакт диска вы найдете исходные файлы вспомогательного кода. Эти файлы включают:

- **Direct3D.cpp/.h.** Эти два файла используются практически во всех проектах книги. Файл Direct3D.cpp содержит функции для инициализации Direct3D, загрузки мешей и вершинных шейдеров, визуализации меша. Direct3D.h включает несколько объектов, используемых для содержания иерархии фреймов и данных меша (обычных и скелетных).

- **XFile.cpp/.h.** Также включенная во все проекты этой книги эта пара файлов используется для включения rmxftmpl.h и rmxfguid.h в ваши проекты. Вы спросите, а почему не подключить эти файлы напрямую? Потому что компилятор сгенерирует ошибки, если вы попробуете, так что добавляйте файл XFile.cpp и включайте XFile.h вместо этого!
- **XParser.cpp/.h.** Также используются во всей книге, эти два файла полезны при анализе .X файлов в ваших проектах. Вместе эти файлы определяют базовый класс, который вы можете наследовать для написания любого анализатора. Читайте главу 3 для дополнительной информации об использовании .X файлов и классов, определенных в этих файлах.

Часть II

Основы анимации

2. Синхронизация анимации и движения

3. Использование формата файла .X

Глава 2

Синхронизация анимации и движения

Игры наполнены движением. Здесь бегают персонажи, там летают пули, зачастую существует множество объектов, перемещающихся по игровому миру. Очень важным аспектом, который нельзя проигнорировать, является плавное движение всех этих объектов. Вы никогда не думали использовать анимацию и движение, основанные на времени? Использование движения, основанного на времени, ново, и, чтобы не отставать от остального мира, вы должны уяснить, что оно может помочь вам в ваших игровых проектах. На самом деле вам необходимо не просто понять, как использовать движение, основанное на времени, а что такое движение вообще. Вы думаете, только персонажи перемещаются в вашей игре? Нет, игровые кинематографические камеры также нуждаются в вашем управлении. В этой главе рассмотрено использование движения, основанного на времени, в ваших проектах.

В этой главе вы научитесь:

- Создавать плавную анимацию, основанную на времени;
- Работать с движением, основанным на времени;
- Перемещать объекты вдоль заранее определенных траекторий;
- Создавать внутриигровые кинематографические последовательности.

Использование движения, синхронизированного по времени

Хотя сначала может показаться, что синхронизация не важна, на самом деле она играет важную роль в игровом проекте. Я не говорю о времени суток, я имею ввиду синхронизацию анимации вплоть до миллисекунд. Такая точная синхронизация необходима для плавного анимирования и движения объектов в вашем проекте. Эта тема является основной, поэтому все программисты игр должны хорошо понимать ее.

Когда вам приходится двигать ваши меши, движение на основе времени является лучшим решением. На протяжении всей книги я, в основном, использую движение, синхронизированное по времени, для контролирования скорости разнообразных анимаций. В этой главе я хочу подробно объяснить использование времени (для анимации и движения).

Смысл использования движения на основе времени прост - движение и анимация выполняются всегда за одно и то же время на любой системе, независимо от скорости компьютера. Например, 2-ГГц система обрабатывающая 20-секундную анимацию, безусловно, выполнит все быстро, и в результате получиться очень плавное движение, когда как та же самая анимация, обработанная на 500-МГц системе будет неплавной, но все равно будет длиться 20 секунд.

Более чем вероятно, что более медленный компьютер пропустит некоторые кадры анимации, чтобы не отставать по скорости от более быстрых компьютеров. В этом весь секрет - медленные компьютеры могут пропускать большее количество кадров и при этом поддерживать что-то похожее на фактическое движение.

Я думаю, вы начинаете понимать идею. Хотя анимация, основанная на времени, настолько же проста, насколько звучит, начинающим программистам игр она может показаться тайной. И хотя это книга об использовании приемов современной анимации, вы должны полностью понимать движение, основанное на времени. Поэтому я хочу представить вам (или напомнить) мир движения, основанного на синхронизации по времени. Я начну с краткого обзора того, как можно считать время в Windows.

Считывание времени в Windows

Самым простым способом считывания времени в Windows является использование функции timeGetTime. Эта функции не имеет параметров. Как показывает следующий код, timeGetTime просто возвращает количество миллисекунд, прошедших с момента запуска Windows.

```
DWORD TimeSinceStarted = timeGetTime();
```

Что полезного она делает? Обычно, функция timeGetTime вызывается один раз за кадр, обрабатываемый вашей игрой. Для каждого кадра вы вычитаете время последнего обработанного кадра из текущего времени, чтобы получить время в миллисекундах, прошедшее с предыдущего кадра. Вы можете использовать это прошедшее время для расчета вашего движения на основе времени.

Хранить время обновления последнего кадра так же просто, как и использовать статические переменные. В начале функции обновления кадра вставьте статическую переменную, хранящую текущее время.

```
void FrameUpdate()
{
  static DWORD LastTime = timeGetTime();
```

В конце функции FrameUpdate вы можете сохранить текущее время в переменной LastTime.

```
LastTime = timeGetTime();
```

Таким образом (записывая текущее времени в конце функции обновления кадра), вы сумели сохранить время окончания обновления кадра. В последующие вызовы FrameUpdate переменная LastTime все еще будет содержать время последнего обновления кадра. Используя это значение, вы можете вычислить количество времени, прошедшее с последнего вызова FrameUpdate, вычитая LastTime из текущего времени.

```
DWORD ElapsedTime = timeGetTime() - LastTime;
```

Именно это прошедшее время и время последнего обновления кадра вы будете использовать на протяжении всей книги. А теперь, как насчет тех случаев, когда вам необходимо вычислить количество миллисекунд, прошедших с заданного времени, например, с начала анимации, вместо того чтобы считать прошедшее время или время с первого вызова функции FrameUpdate?

Так же как вы использовали статическую переменную для хранения времени последнего обновления кадра, вы можете хранить время первого вызова функции. Используя статическую переменную, вы можете определить, сколько миллисекунд прошло с момента первого вызова функции.

```
void FrameUpdate()
{
  static DWORD StartTime = timeGetTime();
  DWORD ElapsedTime = timeGetTime() - StartTime;
```

```
    // ElapsedTime - количество миллисекунд,прошедших
с первого вызова
    // функции FrameUpdate.
}
```

Таким же образом, как вышеприведенный код отслеживал количество времени, прошедшее с первого вызова функции FrameUpdate, можно встроить время начала анимации в свои структуры данных. Вы сможете узнать об этом больше в главе 5 "Использование скелетной анимации на основе ключевых кадров".

Если вы читали последующие главы, то вероятно уже знакомы с использованием анимации, основанной на синхронизации по времени. Все демонстрационные программы, поставляемые вместе с книгой, используют анимацию, отсчитываемую в миллисекундах. В сочетании с ключевыми кадрами, анимация на основе синхронизации по времени является превосходным решением для создания плавной анимации. Читайте далее для более подробного рассмотрения использования анимации, основанной на синхронизации по времени.

Анимирование с использованием временных меток

Раньше игры были сделаны так, чтобы анимировать графику основываясь на обработке каждого кадра. Для того чтобы анимация все время выполнялась на одной и той же скорости, эти игры иногда ограничивали количество кадров, обрабатываемых в секунду. Конечно, старые игры были сделаны для компьютеров, которые не могли обрабатывать более 20-30 кадров в секунду, так что было разумно предполагать, что ограничение кадров никогда не будет превосходить 20 или 30 в секунду.

Но это было тогда, а сейчас есть сейчас. Современные компьютеры могут бегать кругами вокруг своих предшественников, и ограничение количества кадров для управления анимацией определенно устарело. Вам необходимо основываться на скорости анимации, на количестве времени, которое прошло с момента начала анимации. Реализовать это не составляет никакого труда, т. к. вы уже знаете, как записывать время начала анимации. Кроме того, для каждого обновляемого кадра вы можете считывать текущее время и вычитать начало анимации. Полученное время является смещением в анимации.

Предположим, вы используете ключевые кадры, основанные на синхронизации по времени, для вашего анимационного движка. Вы можете использовать простую структуру ключевых кадров для хранения времени и матрицы преобразования, как например:

```
typedef struct sKeyframe {
  DWORD  Time;
  D3DMATRIX   matTransformation;
}  sKeyframe;
```

Что типично для ключевых кадров, вы можете хранить массив матриц, каждая из которых ассоциирована со своим уникальным временем. Эти матрицы хранятся в хронологическом порядке, с меньшими временами вначале. Поэтому вы можете создать небольшую последовательность преобразований для ориентирования объекта во времени. (Смотри рис. 2.1).

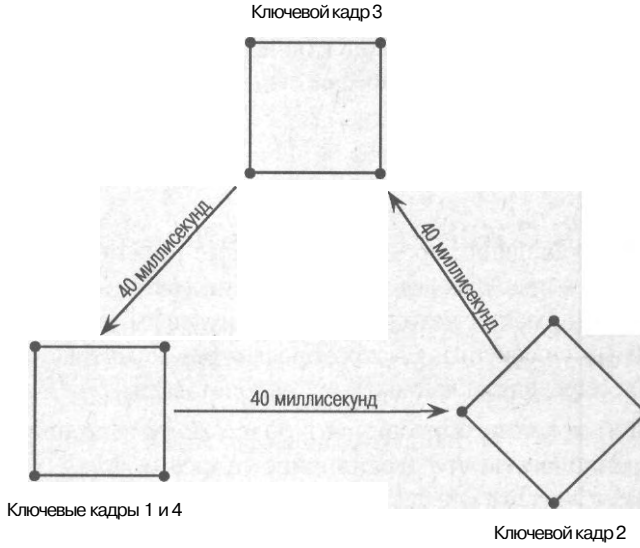


Рис. 2.1. Ключевые кадры отображают ориентацию куба. Кадры разделены между собой 400 миллисекундами, и для определения промежуточных ориентации производится интерполяция

Для отображения ключевых кадров, показанных на рис. 2.1, я создал массив:

```
sKeyframe  Keyframes[4]  =  {
{  0,  1.00000f,  0.00000f,  0.00000f,  0.00000f,
      0.00000f,  1.00000f,  0.00000f,  0.00000f,
      0.00000f,  0.00000f,  1.00000f,  0.00000f,
      0.00000f,  0.00000f,  0.00000f,  1.00000f;  },
{  400,  0.000796f,  1.00000f,  0.00000f,  0.00000f,
       -1.00000f,  0.000796f,  0.00000f,  0.00000f,
       0.00000f,  0.00000f,  1.00000f,  0.00000f,
       50.00000f,  0.00000f,  0.00000f,  1.00000f;  },
{  800,  -0.99999f,  0.001593f,  0.00000f,  0.00000f,
```

```
        -0.001593f, -0.99999f, 0.00000f, 0.00000f,
        0.00000f, 0.00000f, 1.00000f, 0.00000f,
        25.00000f, 25.00000f, 0.00000f, 1.00000f; },
{ 1200, 1.00000f, 0.00000f, 0.00000f, 0.00000f,
         0.00000f, 1.00000f, 0.00000f, 0.00000f,
         0.00000f, 0.00000f, 1.00000f, 0.00000f,
         0.00000f, 0.00000f, 0.00000f, 1.00000f; }
};
```

А теперь самая веселая часть. Используя методы синхронизации, о которых вы читали ранее, вы можете сохранять время начала анимации. И для каждого кадра, обновляющего анимацию, вы можете посчитать время, прошедшее с начала анимации (используя его для нахождения смещения ключевого кадра). Создайте простую функцию обновления кадров, которая будет определять используемое преобразование на основе времени, прошедшего с первого ее вызова.

```
void FrameUpdate()
{
  static DWORD StartTime = timeGetTime();
  DWORD Elapsed = timeGetTime() - StartTime;
```

Имея значение прошедшего времени, вы можете просмотреть ключевые кадры для нахождения двух, в пределах которых оно лежит. Например, если текущее время 60 миллисекунд, то анимация находится между нулевым кадром (0 миллисекунд) и кадром #1 (400 миллисекунд). Быстро просмотрев ключевые кадры, используя прошедшее время, определяем какие из них использовать.

```
DWORD Keyframe = 0 ;  // Начнем с первого кадра
for(DWORD i=0;i<4;i++)  {
  // Если время больше или равно времени ключевого кадра,
  // то обновить используемый ключевой кадр
  if(Time >= Keyframes[i].Time)
    Keyframe = i;
}
```

В конце цикла в переменной Keyframe будет храниться первый из двух кадров, между которыми находится анимационное время. Если Keyframe не последний ключевой кадр в массиве (в котором всего четыре ключевых кадра), то для получения второго ключевого кадра необходимо добавить 1 к Keyframe. Если же Keyframe является последним кадром в массиве, то можно использовать это же самое значение для ваших вычислений.

Намного лучшим решением является использование второй переменной для хранения значения следующего ключевого кадра. Помните, что если Keyframe является последним ключевым кадром в массиве, то новому ключу необходимо присвоить то же самое значение.

```
DWORD Keyframe2 = (Keyframe==3) ? Keyframe:Keyframe + 1;
```

Теперь вам необходимо, используя значения времени, вычислить скаляр на основании разности ключей и расположения ключевых кадров между ними.

```
DWORD TimeDiff = Keyframes[Keyframe2].Time -
                 Keyframes[Keyframe].Time;

// Убедиться, что существует разница во времени
// для исключения деления на 0
if(!TimeDiff)
  TimeDiff=1;
float Scalar = (Time -Keyframes[Keyframe].Time)/TimeDiff;
```

Теперь у вас есть значение скаляра (находящегося в диапазоне от 0 до 1), который используется для интерполяции ключевых матриц преобразования. Чтобы облегчить работу с матрицами преобразований, они приводятся к типу D3DXMATRIX, так что D3DX может выполнить всю работу за вас.

```
// Вычислить разность преобразований
  D3DXMATRIX matInt = \
  D3DXMATRIX(Keyframes[Keyframe2].matTransformation)  - \
  D3DXMATRIX(Keyframes[Keyframe].matTransformation);
matInt *= Scalar; // Scale the difference

  // Прибавить масштабированную матрицу преобразования к матрице
  // первого ключевого кадра.
  matInt += D3DXMATRIX(Keyframes[Keyframe].matTransformation);
```

На данном этапе у вас есть корректно анимированная матрица преобразования, хранимая в matInt. Для просмотра результатов вашей работы установите матрицу matInt в качестве матрицы преобразования мира и визуализируйте ваш анимированный меш.

Как вы видите, использовать анимацию, основанную на синхронизации по времени, достаточно просто. Даже если вы не используете ключевые кадры, вам все равно могут пригодиться некоторые методы работы со временем в ваших программах. После того как вы уже увидели насколько просто использовать анимацию на основе синхронизации по времени, посмотрите, как просто использовать движение на основе синхронизации по времени.

Перемещение, синхронизированное со временем

Движение на основе синхронизации по времени не просто приложение к анимации. Движение является важной частью вашей игры, и использование движения, основанного на синхронизированнии по времени, гарантирует, что ваша игра будет одинаково идти на всех системах, независимо от их производительности.

Наиболее частое применение движения на основе времени находит место при перемещении объекта на определенное расстояние за определенное время. Например, предположим, что игрок переместил джойстик вправо, ваша игра реагирует на это, передвигая персонаж немного вправо, скажем, на 64 единицы за одну секунду, что эквивалентно 0.064 единицам за миллисекунду.

Используя небольшую функцию, вы можете вычислить количество единиц (вещественное число), на которое необходимо сдвинуть объект, основываясь на времени, прошедшем между кадрами.

```
float CalcMovement(DWORD ElapsedTime, float PixelsPerSec)
{
  return (PixelsPerSec / 1000.0f * (float)ElapsedTime);
}
```

Как вы видите, функция CalculateMovement использует следующую формулу:

```
PixelsPerSec / 1000.0f * ElapsedTime;
```

Переменная PixelsPerSec содержит число единиц, на которое вы хотите переместить ваш объект за одну секунду. Величина 1000.0 означает 1000 миллисекунд. В основном, вы вычисляете количество единиц движения за миллисекунду. Наконец, вы умножаете скорость движения в миллисекундах на ElapsedTime для вычисления перемещения.

Эта разновидность движения, основанная на синхронизации по времени, очень проста, однако ее нельзя упускать. Знание принципов движения, основанного на синхронизации по времени, лежит в основе более сложных методов, таких как плавное движение объектов по заранее определенной траектории.

Движение вдоль траекторий

Как было сказано ранее, движение на основе синхронизации по времени определяется как расстояние движения, деленное на 1000 и умноженное на прошедшее время. Я использовал пример, когда пользователь нажал "вправо" на джойстике и его персонаж переместился вправо на заданное число единиц. А как насчет тех случаев, когда вы хотите перемещать объекты без участия пользователя? Например, предположим, игрок нажал кнопку и выпустил пули из большой пушки, которую нес с собой. Эти пули летят по определенной траектории с определенной скоростью. Вы можете установить скорости для каждой пули и не использовать траектории, но как насчет тех супер-пуль, которые могут атаковать через части уровня, по заранее заданной траектории?

Эти специальные случаи требуют установки координат движения заранее, и быстрых вычислений, для определения расположения объекта на этой траектории. А что насчет движения таких объектов как персонажи, платформы, рубильники? Вы угадали, использование траекторий является идеальным решением для всех этих проблем!

Я собираюсь рассказать о двух наиболее используемых типах траекторий - прямолинейных и криволинейных. Начну же свой рассказ с использования прямолинейных траекторий.

Следование прямолинейным траекториям

Прямолинейная траектория - это прямая. Перемещение происходит от начальной точки к конечной без остановок и поворотов. Прямая определяется двумя точками - началом и концом. Чтобы следовать прямолинейной траектории вам просто нужно двигаться по прямой из точки А в точку Б.

Для перемещения объекта по прямолинейной траектории вам необходимо, используя простые формулы, вычислить координаты точки, лежащей на прямой. Рис. 2.2 демонстрирует, что для расчета координатточки, лежащей на отрезке, используя скаляр (лежащий в пределах от 0 до 1), вам необходимо вычислить разность координат концевых точек отрезка, умножить ее на скаляр и добавить к результату координаты начальной точки.

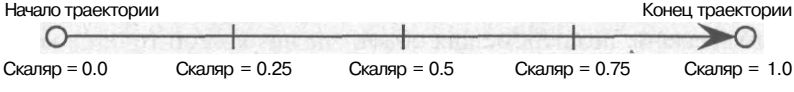


Рис. 2.2. Вы можете определить точку траектории, используя скаляр, который является частью полной длины траектории с началом в 0 и концом в 1

```
// Определим, начальную и конечную точки прямолинейной траектории
// Scalar = положение для вычисления (от 0 до 1)
D3DXVECTOR3 vecStart = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 vecEnd = D3DXVECTOR3(10.0f, 20.0f, 30.0f);
D3DXVECTOR3 vecPos = (vecEnd - vecStart) * Scalar + vecStart;
```

Если вы установите Scalar равным 0.5, то vecPos будет иметь координаты 5.0, 10.0, 15.0, которые являются серединой траектории. Теперь предположим, что вы не хотите использовать скаляр. Что если использовать вместо него трехмерные единицы? Например, предположим, что вместо использования значения скаляра 0.5, вы хотите узнать координаты точки, расположенной в 32 единицах от начала траектории.

Для определения координат, используя в качестве измерения трехмерные единицы, с помощью функции D3DXVec3Length вьиисляется длина траектории и делится на заданное положение для получения скаляра, использовавшегося в предыдущих вычислениях.

Например, для получения координат точки, лежащей в 32 единицах от начала траектории, вы можете использовать следующий код:

```
// Pos = положение (в трехмерных единицах) точки на траектории,
// которую вы хотите определить
// Определение начальной и конечной точки прямолинейной траектории
D3DXVECTOR3 vecStart = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 vecEnd = D3DXVECTOR3(10.0f, 20.0f, 30.0f);

// Вычисляем длину траектории
float Length = D3DXVec3Length(&(vecEnd-vecStart));

// Вычисляем скаляр, деля pos на len
float Scalar = Pos / Length;

// Используем скаляр для вычисления координат
D3DXVECTOR3 vecPos = (vecEnd - vecStart) * Scalar + vecStart;
```

Теперь, когда вы можете вычислять расположение любой точки, лежащей на траектории, можно использовать эти знания для перемещения объекта вдоль траектории. Предположим, вы хотите, применяя теорию движения, основанного на синхронизации по времени, переместить объект из одной точки в другую за 1000 миллисекунд. Следующий код (вызываемый один раз за кадр), поможет вам выполнить это, бесконечно перемещая объект из начальной точки в конечную.

```
// vecPoints[2] = Начальная и конечная координаты траектории
// Используйте этот код каждый кадр для перемещения объекта
// вдоль прямолинейной траектории на основании текущего времени
float Scalar = (float)(timeGetTime() % 1001) / 1000.0f;
D3DXVECTOR3 vecPos = (vecPoints[1] - vecPoints[0]) * \
                     Scalar + vecPoints[0];
// используйте координаты vecPos.x, vecPos.y, and vecPos.z для объекта
```

Передвижение по криволинейным траекториям

В вашей игре траектории не обязательно будут прямолинейными. Ваши объекты могут двигаться по изящным криволинейным траекториям, как например, когда персонаж ходит по кругу. Определить гладкий округлый путь, используя прямые линии практически невозможно, так что вам необходимо разработать второй тип траекторий, который сможет использовать кривые. Надо заметить, что

не все типы кривых могут быть использованы. Помните, что это продвинутая анимация, - мы занимаемся великими делами, так что лучше всего нам подойдут кубические кривые Безье! Как показано на рис. 2.3, для определения кубической кривой Безье необходимы четыре точки (две концевые и две промежуточные).

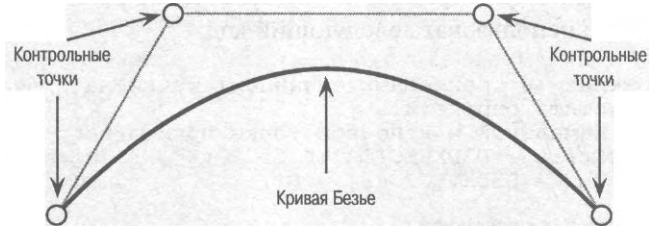


Рис. 2.3. Кубическая кривая Безье использует четыре точки для определения направления движения от начала к концу

Как вы можете видеть, кубическая кривая Безье не просто кривая - она может выгибаться и крутиться множеством различных способов. Управляя этими четырьмя контрольными точками, вы можете создавать действительно полезные траектории для использования в своих проектах. Принцип работы кубических кривых Безье прост, но вот реализовать его практически достаточно сложно.

Чтобы понять теорию использования кубических кривых Безье, посмотрите на рис. 2.4, на котором показано построение кривой по четырем контрольным точкам.

Разбиение линий, соединяющих точки кривой, служит для визуальной помощи и для сглаживания кривой. Чем больше дополнительных разбиений каждой линии вы сделаете, тем глаже будет выглядеть результирующая кривая. Чтобы увидеть, как будет выглядеть кривая, заданная точками, вам необходимо соединить все разбиения с каждой стороны линий как показано на рис. 2.5

Хотя мы и изобразили кривую, компьютер так делать не будет, и нам это не поможет узнать координаты точки, лежащей на кривой. Нам нужно отыскать способ вычисления точных координат любой точки, лежащей на кривой. Таким образом, вы сможете делать с координатами все что угодно - от рисования кривой до вычисления координат расположения вашего объекта на криволинейной траектории! Вот формула для вычисления координат лежащих на кривой:

```
C(s)=P0*(1-s)3+P1*3*s*(1-s)2+P2*3*s2*(1-s)+P3*s3
```

В этой формуле контрольные точки задаются как P0, P1, P2 и P3, которые являются начальной, первой промежуточной, второй промежуточной и конечной точками соответственно. Результирующая точка, лежащая на кривой, определена как C(s), где s - это

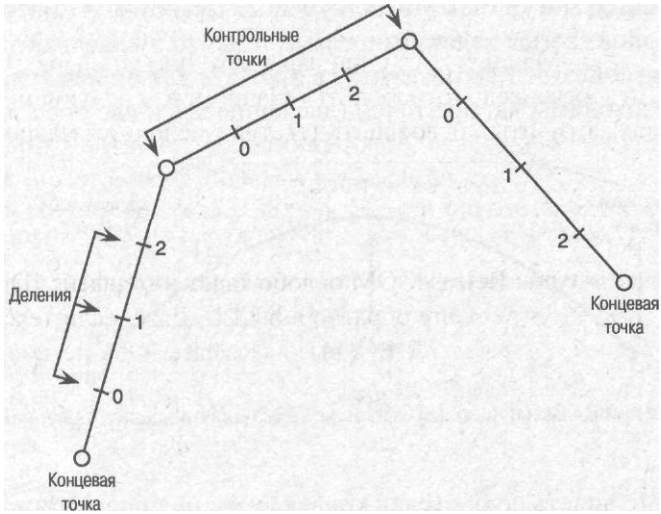


Рис. 2.4. Вы определяете кубическую кривую Безье, соединяя четыре контрольные точки и разбивая соединяющее их линии в заданном соотношении. Каждое разбиение нумеруется для дальнейшего использования

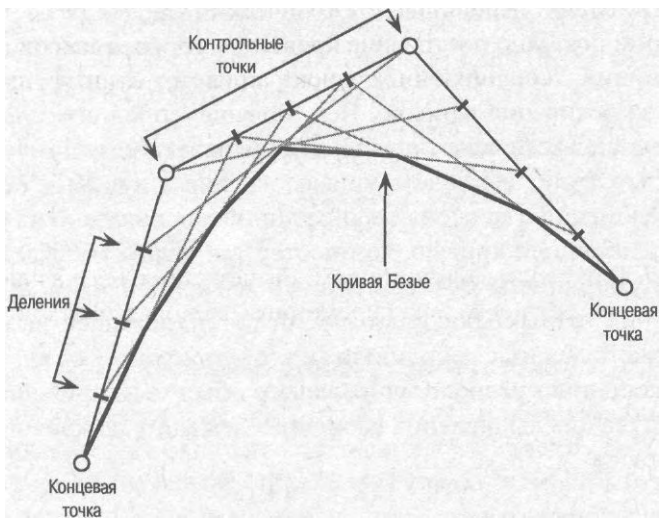


Рис. 2.5. Вы можете нарисовать кубическую кривую Безье, выделив линии, соединяющие пронумерованные разбиения

значение скаляра (или времени) в диапазоне от 0 до 1, который определяет смещение точки, координаты которой мы хотим определить, вдоль кривой.

Значение s=0 определяет начальную точку, в то время как s=1 определяет конечную точку. Любое значение s от 0 до 1 определяет положение между двумя концевыми точками. Таким образом, для вычисления координат середины кривой задайте s=0.5. Положению в четверти кривой соответствует s=0.25 и так далее.

Для упрощения можно создать функцию, параметрами которой были бы четыре контрольных точки (вектора) и скаляр, которая будет возвращать вектор, содержащий координаты точки, лежащей на заданной кривой. Назовем функцию CubicBezierCurve и воспользуемся следующим прототипом для ее определения.

```
void CubicBezierCurve(D3DXVECTOR3 *vecPoint1, // начальная точка
    D3DXVECTOR3 *vecPoint2, // промежуточная точка 1
    D3DXVECTOR3 *vecPoint3, // промежуточная точка 2
    D3DXVECTOR3 *vecPoint4, // Конечная точка
    float Scalar,
    D3DXVECTOR3 *vecOut)
{
```

Теперь приготовтесь записать формулу кубической кривой Безье в программном коде, заменяя соответствующие переменные векторами контрольных точек и скаляром.

```
// C(s) =
*vecOut = \
  // P0 * (1 - s)3 +
   (*vecPoint1)*(1.0f-Scalar)*(1.0f-Scalar)*(1.0f-Scalar)  + \
  // P1 * 3 * s * (1 - s)2 +
   (*vecPoint2)*3.0f*Scalar*(1.0f-Scalar)*(1.0f-Scalar)  + \
  // P2 * 3 * s2 * (1 - s) +
   (*vecPoint3)*3.0f*Scalar*Scalar*(1.0f-Scalar)  + \
  // P3 * s3
   (*vecPoint4)*Scalar*Scalar*Scalar;
}
```

Вот и все! Теперь вы можете вычислять координаты точки, лежащей на кубической кривой Безье, задавая контрольные точки и скаляр. Возвращаясь к примеру с кривой, вы можете параметрически найти ее середину, вызвав функцию CubicBezierCurve:

```
D3DXVECTOR3 vecPos;
CubicBezierCurve(&D3DXVECTOR3(-50.0f, 25.0f, 0.0f), \
                 &D3DXVECTOR3(0.0f, 50.0f, 0.0f), \
                 &D3DXVECTOR3(50.0f, 0.0f, 0.0f), \
                 &D3DXVECTOR3(25.0f, -50.0f, 0.0f), \
                 0.5f, &vecPos);
```

Итак, вы можете использовать координаты, возвращаемые функцией CubicBezierCurve (содержащиеся в векторе vecPos), для перемещения объекта в игре. Постепенно изменяя скаляр от 0 до 1 (с течением заданного времени), вы двигаете объект от начала траектории к ее концу. Например, для перемещения по криволинейной траектории за 1000 миллисекунд вы можете использовать следующий код:

```
// vecPoints[4]=начальная,промежуточная 1,промежуточная 2 и конечная точки
// Каждый кадр использует данный код для расположения объекта на кривой
//используя текущее время
D3DXVECTOR3 vecPos;
float Scalar = (float)(timeGetTime() % 1001) / 1000.0f;
CubicBezierCurve(&vecPoints[0], &vecPoints[1], \
                 &vecPoints[2], &vecPoints[3], \
                 Scalar, &vecPos);
// Используйте координаты vecPos.x, vecPos.y, and vecPos.z для объекта
```

Все это хорошо, но использовать скаляр немного неудобно при работе с реальными трехмерными единицами измерения. Я имею в виду, как узнать, какой скаляр использовать, если вы хотите переместить объект на 50 единиц вдоль кривой? Есть ли способ вычислить длину кривой, чтобы использовать ее, как и прямые линии?

Странно, но нет. Нет простого способа вычисления длины кривой Безье. Так или иначе, можно аппроксимировать длину, используя простые вычисления. Предполагая, что четыре контрольные точки кривой обозначены как p0, p1, p2, и p3, можно найти расстояние между точками p0 и p1, p1 и p2, p2 и p3, разделить результат пополам и добавить расстояние между p0 и p3 (также деленное пополам). В виде кода это будет выглядеть так:

```
// p[4] = четыре контрольные точки - координаты векторов
float Length01 = D3DXVec3Length(&(p[1]-p[0]));
float Length12 = D3DXVec3Length(&(p[2]-p[1]));
float Length23 = D3DXVec3Length(&(p[3]-p[2]));
float Length03 = D3DXVec3Length(&(p[3]-p[0]));
float CurveLength = (Length01+Length12+Length23) * 0.5f + \
                    Length03 * 0.5f;
```

Переменная CurveLength будет содержать приблизительную длину кривой. Вы будете использовать значение CurveLength таким же образом, как и с вычислениями в прямолинейных траекториях для преобразования количества единиц в скаляр для вычисления точных координат точки, лежащей на кривой.

```
// Pos = положение на кривой (от 0-CurveLength)
float Scalar = Pos / CurveLength;
CubicBezierCurve(&vecPoints[0], &vecPoints[1], \
                 &vecPoints[2], &vecPoints[3], \
                 Scalar, &vecPos);
```

Как вы можете видеть, кубические кривые Безье не слишком трудно использовать. Все формулы простые, и я оставляю детали вычислений для математических книжек (или такой хорошей книги как Kelly Dempski's "Рассмотрение кривых и поверхностей" - смотрите приложение А "Книги и веб ссылки"). На данный момент для меня главное - сделать работающий проект игры. Говоря об этом, давайте посмотрим, что можно сделать с новоприобретенными знаниями использования прямолинейных и криволинейных траекторий для созданий маршрутов.

Определение маршрутов

Траектория сама по себе приносит не очень много пользы; бывают случаи, когда вам необходимо соединить вместе несколько траекторий, по которым должен двигаться объект. Я говорю о сложных траекториях, которые частично прямолинейные, а частично криволинейные. По сути, мы больше не обсуждаем траектории, а переходим к следующей теме: маршруты!

Как вы можете видеть на рис. 2.6, маршрут - это набор траекторий, соединенных между собой концевыми точками.

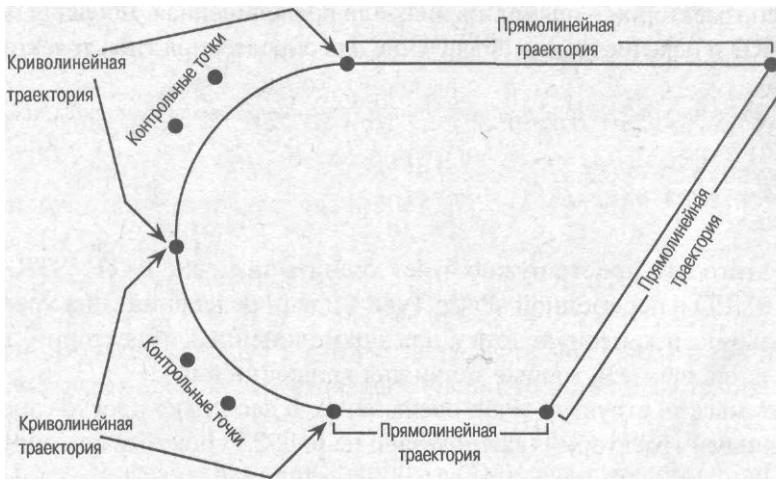


Рис. 2.6. Вы можете создавать сложные маршруты, используя набор прямолинейных и криволинейных траекторий. Как вы можете здесь видеть, траекториям не обязательно соединяться для завершения маршрута

Как вы уже догадываетесь, маршрут может быть определен, используя массив траекторий. Создав основную структуру траектории, вы можете хранить информацию прямолинейных и криволинейных траекторий в одной структуре. Хитрым

ходом является поиск общностей этих двух типов траекторий и их обособление. Например, и прямолинейные, и криволинейные траектории имеют начало и конец. Поэтому мы можем определить в основной структуре траектории два набора координат, соответствующих ее начальной и конечной точкам, как показано:

```
typedef struct {
  D3DXVECTOR3 vecStart, vecEnd;
} sPath;
```

Единственной разницей между этими двумя типами траекторий является то, что криволинейная траектория имеет две дополнительные точки. Добавление двух дополнительных векторов к ранее определенной структуре sPath дает ей возможность прекрасно справиться с хранением этих двух дополнительных контрольных точек.

```
typedef struct {
  D3DXVECTOR3 vecStart, vecEnd;
  D3DXVECTOR3 vecPoint1, vecPoint2;
} sPath;
```

Теперь единственное, что отсутствует в структуре sPath, - это флаг, определяющий тип траектории - прямолинейная или криволинейная. Добавьте переменную типа DWORD и перечисляемое объявление для определения типа траектории.

```
enum { PATH_STRAIGHT = 0, PATH_CURVED };
typedef struct {
  DWORD Type;
  D3DXVECTOR3 vecStart, vecEnd;
  D3DXVECTOR3 vecPoint1, vecPoint2;
} sPath;
```

После этого вам просто нужно будет хранить значение PATH_STRAIGHT или PATH_CURVED в переменной sPath::Type - для определения типа хранимых данных: начальную и конечную точку для прямолинейных траекторий, начальную, конечную и две промежуточные точки для криволинейных.

Создать массив структур sPath очень легко, а насколько просто заполнить этот массив данными траектории (как показано на рис. 2.7) показывает следующий код.

```
sPath Path[3] = {
{ PATH_STRAIGHT, D3DXVECTOR3(-50.0f, 0.0f, 0.0f), \
                 D3DXVECTOR3(-50.0f,  0.0f, 25.0f), \
                 D3DXVECTOR3(0.0f, 0.0f, 0.0f), \
                 D3DXVECTOR3(0.0f, 0.0f, 0.0f) }, \
{ PATH_CURVED, D3DXVECTOR3(-50.0f, 0.0f, 25.0f), \
               D3DXVECTOR3(0.0f,  0.0f, 50.0f), \
               D3DXVECTOR3(50.0f, 0.0f, 0.0f), \
               D3DXVECTOR3(25.0f, 0.0f, -50.0f) }, \
```

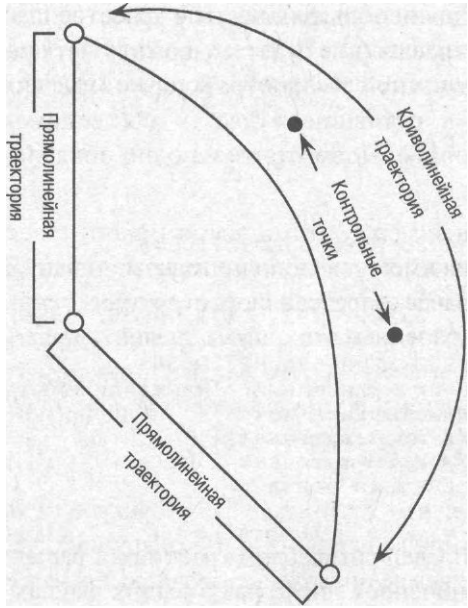


Рис. 2.7. Комбинация двух прямолинейных и одной криволинейной траектории образует сложный маршрут

```
{  PATH_STRAIGHT,  D3DXVECTOR3(25.0f,  0.0f, -50.0f), \
                   D3DXVECTOR3(-50.0f,  0.0f,  0.0f),  \
                   D3DXVECTOR3(0.0f,  0.0f,  0.0f),  \
                   D3DXVECTOR3(0.0f,  0.0f,  0.0f)  }  \
```

Конечно же вы не должны вручную записывать маршруты в ваших проектах; более правильным было бы использование внешнего источника, такого как .X файл для хранения данных вашего маршрута.

Создание анализатора маршрутов .X файла

Самым простым местом, откуда можно получать данные о траектории, как вы уже, наверное, догадались, являются .X файлы! Вы можете создать несколько простых шаблонов, используя специализированный анализатор .X файлов, для получения траекторий, используемых в ваших проектах. Вы даже можете создавать маршруты из ваших шаблонов траекторий для облегчения жизни!

Вы можете скопировать основные структуры траекторий, разработанные в предыдущем разделе, для использования их в качестве шаблонов ваших .X файлов. Основной шаблон траектории будет содержать четыре вектора: первые два являются начальной и конечной точкой траектории (как для прямолинейной, так и для криволинейной), а оставшиеся два - для вспомогательных координат (криволинейных траекторий). Посмотрите на одно объявление шаблона, которое вы можете использовать.

```
// {F8569BED-53B6-4 923-AF0B-59A09271D556}
// DEFINE_GUID(Path,
// 0xf8569bed, 0x53b6, 0x4923,
// 0xaf, 0xb, 0x59, 0xa0, 0x92, 0x71, 0xd5, 0x56);
template Path {
  <F8569BED-53B6-4 923-AF0B-59A09271D556>
  DWORD Type; // 0=прямолинейный, 1=криволинейный
  Vector Start; // начальная точка
  Vector Point1; // промежуточная 1
  Vector Point2; // промежуточная 2
  Vector End; // конечная точка
}
```

После того как вы определили шаблон траектории вашего .X файла, вы можете использовать его неограниченное число раз в ваших файлах данных. Для загрузки этих траекторий вам необходимо создать шаблон маршрута (называемый Route), который позволит вам определять множественные объекты траекторий. Этот шаблон маршрута просто содержит массив объектов Path, как показано ниже:

```
// {18AA1C92-16AB-47a3-B002-6178F9D2D12F}
// DEFINE_GUID(Route,
// 0x18aa1c92, 0x16ab, 0x47a3,
// 0xb0, 0x2, 0x61, 0x78, 0xf9, 0xd2, 0xd1, 0x2f);
template Route {
  <18AA1C92-16AB-47a3-B002-6178F9D2D12F>
  DWORD NumPaths;
  array Path Paths[NumPaths];
}
```

В качестве примера использования шаблона Route давайте посмотрим, как будет выглядеть рассмотренный выше маршрут в .X файле.

```
Route MyRoute {
  3; // 3 траектории
  0; // прямолинейный тип траектории
  -50.0, 0.0, 0.0;
  0.0, 0.0, 0.0;
  0.0, 0.0, 0.0;
  -50.0, 0.0, 25.0;,
```

```
1; // криволинейный тип траектории
  -50.0, 0.0, 25.0;
  0.0, 0.0, 50.0;
  50.0, 0.0, 0.0;
  25.0, 0.0, -50.0;,
  0; // прямолинейный тип траектории
  25.0, 0.0, -50.0;
  0.0, 0.0, 0.0;
  0.0, 0.0, 0.0;
  -50.0, 0.0, 0.0;;
}
```

Вы можете получить доступ к вашему объекту маршрута из .X файла, используя специальный анализатор. Этот анализатор ищет только объекты Route. Когда он их находит, то создает массив структур sPath и считывает данные. Данные маршрутов при этом хранятся в виде связанного списка, так что вы можете загружать несколько маршрутов. Эти данные маршрута используют следующий класс:

```
class cRoute
{
  public:
    DWORD m_NumPaths; // # траекторий в списке
    sPath *m_Paths; // список траекторий
    cRoute *m_Next; // следующий маршрут в связанном списке

  public:
    cRoute() { m_Paths = NULL; m_Next = NULL; }
    ~cRoute() { delete [] m_Paths; delete m_Next; }
};
```

Необходимо также немного улучшить структуру sPath. Нужно добавить в нее длину каждой траектории и начальное положение в наборе. Сделать это очень просто. Длина, как вы могли убедиться ранее, является вещественным числом, а начальное положение траектории - это просто сумма длин всех предыдущих траекторий в списке. Новая структура sPath будет выглядеть так:

```
typedef struct {
  DWORD Type;
  D3DXVECTOR3 vecStart, vecEnd;
  D3DXVECTOR3 vecPoint1, vecPoint2;
  float Start; // Начальное положение
  float Length; // Длина траектории
} sPath;
```

Основной причиной включения длины и начальной позиции траектории в структуру sPath является забота о скорости. Предварительное вычисление значения длин при загрузке данных траектории позволяет вам быстро получать доступ к этим

данным (длине и начальному положению) при определении траектории, на которой сейчас расположен объект, используя для этого смещение в маршруте.

Я знаю, что это звучит странно, но подумайте сами - начальное положение и длина каждой траектории являются подобием ключевых кадров; просто вместо измерения времени вы измеряете длины траекторий. Имея расположение объекта на маршруте (скажем 516 единиц), вы можете просмотреть список траекторий, чтобы определить на какой находится объект.

Предположим, маршрут использует шесть траекторий, и четвертая начинается в 400 единицах. Четвертая траектория имеет длину 128 единиц, т. е. перекрывает значения от 400 до 528. Объект, расположенный в 516 единицах, находится на четвертой траектории; отняв положение объекта (516) от конечного положения траектории (528), вы найдете смещение в траектории, которое можете использовать для вычисления скаляра, необходимого при вычислении координат объекта, лежащего на траектории. В этом случае позиция будет 528-516, или 12 единиц, и скаляр будет 12/128, или 0.09375.

Достаточно разговоров, давайте перейдем к коду! Нижеприведенный класс cXRouteParser наследуется от cXParser, таким образом, вы имеете доступ к данным объекта анализатора. Класс cXRouteParser должен просто находить объекты Route и загружать подходящие данные траекторий в созданный класс Route, который привязан к списку маршрутов.

Посмотрите объявление класса cXRouteParser, который содержит указатель на корневой объект Route и шесть функций (три из которых содержат встроенный код класса).

```
class cXRouteParser : public cXParser
{
  protected:
    BOOL ParseTemplate(IDirectXFileData *pDataObj, \
                       IDirectXFileData *pParentDataObj, \
                       DWORD Depth, \
                       void **Data, BOOL Reference);
  public:
    cRoute *m_Route;

  public:
    cXRouteParser() { m_Route = NULL; }
    - cXRouteParser () { Free(); }
    void Free() { delete m_Route; m_Route = NULL; }
    void Load(char *Filename);
    void Locate(DWORD Distance, D3DXVECTOR3 *vecPos);
```

Основной функцией cXRouteParser является, конечно же, шаблон анализатора данных. Немного позже я покажу вам функцию анализатора. Функция Load просто выполняет настройку перед вызовом Parse, которая преобразует загруженные шаблоны Route в связанный список. Функция Locate вычисляет положение на траектории маршрута, которое вы можете использовать для перемещения объекта. Код функции Locate я покажу вам также немного позже. А пока я хочу вернуться к функции ParseTemplate.

Функция ParseTemplate ищет только один шаблон объекта - Route. Как только он найден, создается класс cRoute, структуры траекторий и загружаются данные. Класс cRoute помещается в список загруженных маршрутов, после чего продолжается разбор .X файла. Вот как выглядит код функции ParseTemplate:

```
BOOL cXRouteParser::ParseTemplate(IDirectXFileData *pDataObj, \
              IDirectXFileData *pParentDataObj, \
              DWORD Depth, \
              void **Data, BOOL Reference)
{
const GUID *Type = GetTemplateGUID(pDataObj);

// Обрабатывать только объекты Route
if(*Type == Route) {
  // получить указатель на данные
  DWORD *DataPtr = (DWORD*)GetTemplateData(pDataObj, NULL);

  // Создать и связать маршрут
  cRoute *Route = new cRoute();
  Route->m_Next = m_Route;
  m_Route = Route;

  // Получить # траекторий в маршруте и создать список
  Route->m_NumPaths = *DataPtr++;
  Route->m_Paths = new sPath[Route->m_NumPaths];

  // Получить данные траектории
  for(DWORD i=0;i<Route->m_NumPaths;i++) {

  // Получить тип траектории
  Route->m_Paths[i].Type = *DataPtr++;

   // Получить векторы
  D3DXVECTOR3 *vecPtr = (D3DXVECTOR3*)DataPtr;
  DataPtr+=12; // skip ptr ahead
  Route->m_Paths[i].vecStart = *vecPtr++;
  Route->m_Paths[i].vecPoint1 = *vecPtr++;
  Route->m_Paths[i].vecPoint2 = *vecPtr++;
  Route->m_Paths[i].vecEnd = *vecPtr++;

  // Вычислить длину траектории на основании ее типа
  if(Route->m_Paths[i].Type == PATH_STRAIGHT) {
    Route->m_Paths[i].Length = D3DXVec3Length( \
```

```
        &(Route->m_Paths[i].vecEnd - \
          Route->m_Paths[i].vecStart));
    } else {
      float Length01 = D3DXVec3Length( \
              &(Route->m_Paths[i].vecPoint1 - \
                Route->m_Paths[i].vecStart));
      float Length12 = D3DXVec3Length( \
              &(Route->m_Paths[i].vecPoint2 - \
                Route->m_Paths[i].vecPoint1));
      float Length23 = D3DXVec3Length( \
              &(Route->m_Paths[i].vecEnd - \
                Route->m_Paths[i].vecPoint2));
      float Length03 = D3DXVec3Length( \
              &(Route->m_Paths[i].vecEnd - \
                Route->m_Paths[i].vecStart));
      Route->m_Paths[i].Length = (Length01+Length12+ \
                          Length23)*0.5f+Length03*0.5f;
    }
    // сохранить начальное положение траектории
    if(i)
      Route->m_Paths[i].Start = Route->._Paths[i-1].Start + \
                          Route->m_Paths[i-1].Length;
    else
      Route->m_Paths[i].Start = 0.0f;
    }
   }

   // анализировать дочерние шаблоны
   return ParseChildTemplates(pDataObj, Depth, Data, Reference);
  }
```

Обычно вы не вызываете ParseTemplate напрямую - функция cXRouteParser::Load вызывает Parse, которая, в свою очередь, вызывает ParseTemplate. Зная это, посмотрите на функцию Load (единственным параметром которой является имя анализируемого .X файла)

```
void cXRouteParser::Load(char *Filename)
{
  Free(); // Освободить загруженные маршруты
  Parse(Filename);
}
```

Коротко и ясно - как я люблю! Функция Load на самом деле является промежуточной и используется для освобождения данных ранее загруженных маршрутов и дальнейшего вызова функции Parse. Вот и все!

После того как вы определили шаблоны, создав специализированный класс, загрузили данные маршрута, пришло время двигать объекты! Для этого вернемся к функции cXRouteParser::Locate. Вы, наверное, уже заметили, что в прототипе

функции Locate используются только вещественное число и вектор. Для упрощения, я буду просматривать только первый маршрут в списке для нахождения расположения объекта.

Совет. *Для просмотра множества маршрутов функцией Locate вы могли бы хранить имя каждого объекта Route, которое можно было бы использовать в качестве параметра при вызове функции Locate, для задания просматриваемого маршрута.*

Имея текущее время и расстояния (в трехмерных единицах), на которое необходимо сдвинуть объект, вы можете просмотреть все траектории для отыскания той, на которой находится объект в заданное время. После этого вы вычисляете, как далеко между началом и концом этой траектории находится объект, и корректируете его положение.

Предположим, имеем объект, движущийся со скоростью 200 единиц в секунду, т.е 0.2 единицы в миллисекунду. Умножив скорость в секунду на общее время движения вдоль траектории, получим положение объекта на маршруте. Используйте это положение как параметр Distance при вызове функции Locate.

Функция Locate берет предоставленное ей расстояние и просматривает каждую траекторию, содержащуюся в маршруте. Помните, вы уже вычислили начальную точку и длину для каждой траектории, так что достаточно простой проверки, чтобы убедиться, что расстояние объекта больше начала траектории и меньше ее длины. Посмотрите код функции Locate, чтобы понять, что я имею ввиду.

```
void cXRouteParser::Locate(float Distance, D3DXVECTOR3 *vecPos)
{
  // Просмотр первого маршрута в списке
  cRoute *Route = m_Route;
  if(!Route)
    return;

  // Просмотр каждой траектории в маршруте
  for(DWORD i=0;i<Route->m_NumPaths;i++) {

// Посмотреть, попадает ли расстояние на текущую траекторию
  if(Distance >= Route->m_Paths[i].Start && \
     Distance < Route->m_Paths[i].Start + \
                Route->m_Paths[i].Length) {
  // Использовать Distance для получения смещения
  // в текущей траектории
  Distance -= Route->m_Paths[i].Start;

  // Calculate the scalar value to use
  float Scalar = (float)Distance/Route->m_Paths[i].Length;

  // Вычислить координаты на основании типа траектории
  if(Route->m_Paths[i].Type == PATH_STRATGHT) {
    *vecPos = (Route->m_Paths[i].vecEnd - \
```

```
                Route->m_Paths[i].vecStart) * \
                Scalar + Route->m_Paths[i].vecStart;
      } else {
        CubicBezierCurve(&Route->m_Paths[i].vecStart, \
                &Route->m_Paths[i].vecPoint1, \
                &Route->m_Paths[i].vecPoint2, \
                &Route->m_Paths[i].vecEnd, \
                Scalar, vecPos);
       }
      }
     }
    }
```

Таким образом, вы имеете великолепный метод использования маршрутов в игровых проектах! Посмотрите демонстрационную программу Route, чтобы увидеть маршруты в действии. (Смотрите конец этой главы для дополнительной информации.) Демонстрационная программа Route медленно двигает объект вдоль траектории, используя заданную скорость, основанную на времени.

Маршруты могут быть также использованы для движения ваших игровых персонажей и других объектов, а как насчет того чтобы использовать маршруты для снятия носок? Одним замечательным применением маршрутов является создание кинематографических последовательностей, контролируя игровые камеры.

Создание внутриигровых кинематографических последовательностей

Использование анимации на основе синхронизации по времени является необходимым для плавного отображения, но что еще можно сделать, используя движение, основанное на синхронизации по времени? Конечно, это замечательно двигать объекты вдоль траекторий, но это все что можно делать? Ответ очевиден - нет! Вы можете намного больше, используя движение, основанное на синхронизации по времени, включая создание внутриигровых кинематографических последовательностей, как например в Silicon Knights' Eternal Darkness: Sanity's Requiem. В Eternal Darkness игроку показывают анимационные последовательности, которые используют трехмерный движок игры.

Обычно игровая камера переключается с точки зрения игрока к другой точке зрения, чтобы показать противостояние игрока и остальных персонажей, расширяя сюжетную линию.

Для использования кинематографической камеры, вы можете опираться на механизмы, описанные ранее в этой книге, и применять заранее вычисленные анимационные последовательности из главы 3. Как показано на рис. 2.8, необхо-

димо просто нарисовать траекторию движения камеры. Совместив ее с полностью вычисленной заранее анимацией, вы получите полноценный игровой кинематографический движок!

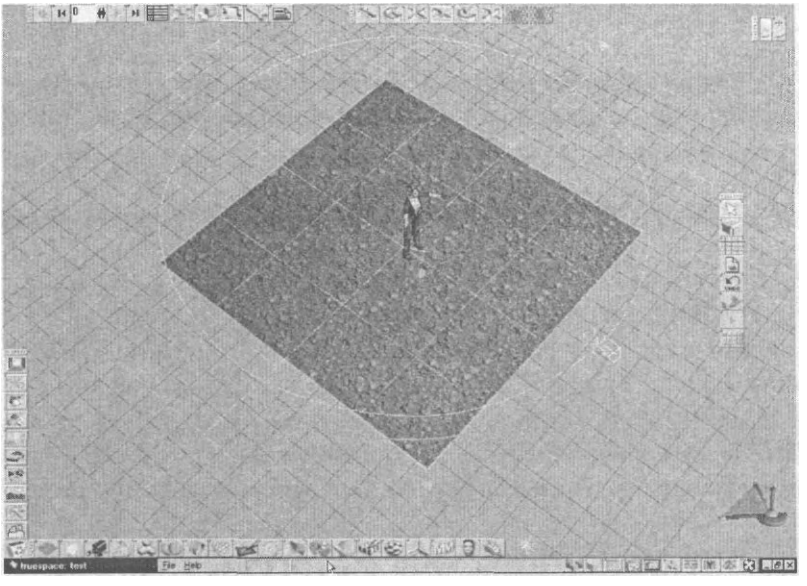


Рис. 2.8. Также как и в кино, камера следует заранее определенной траектории, показывая вычисленную заранее анимацию с различных углов. Углы определяются установкой траектории для камеры и прицела камеры (направление взгляда камеры)

Вместо того чтобы рассматривать заново все, что вы видели в этой главе, я порекомендую вам самостоятельно посмотреть демонстрационную программу Cinematic, которая иллюстрирует небольшой кинематографический пример. Она просто загружает последовательность ключевых точек (используя анализатор траекторий .X файла), которые являются траекторией следования камеры. Расположение камеры пересчитывается каждый кадр, используя ключевые точки, совместно с изменением ориентации смотрового окна.

Посмотрите демонстрационные программы

Эта глава познакомила вас с анимацией, основанной на синхронизации по времени, используя структуры ключевых кадров с движением вдоль траекторий и маршрутов во времени. На компакт-диске вы найдете демонстрационные

программы, которые иллюстрируют то, что вы прочитали в данной главе. Для точной информации о расположении этих программ посмотрите конец этой главы. В следующих нескольких разделах содержится описание работы демонстрационных программ.

TimedAnim

Демонстрационная программа TimedAnim, показанная на рис. 2.9, иллюстрирует использование анимирования объектов (таких как робот) с использованием ключевых кадров. Эта демонстрационная программа выполняется, пока вы ее не закроете.

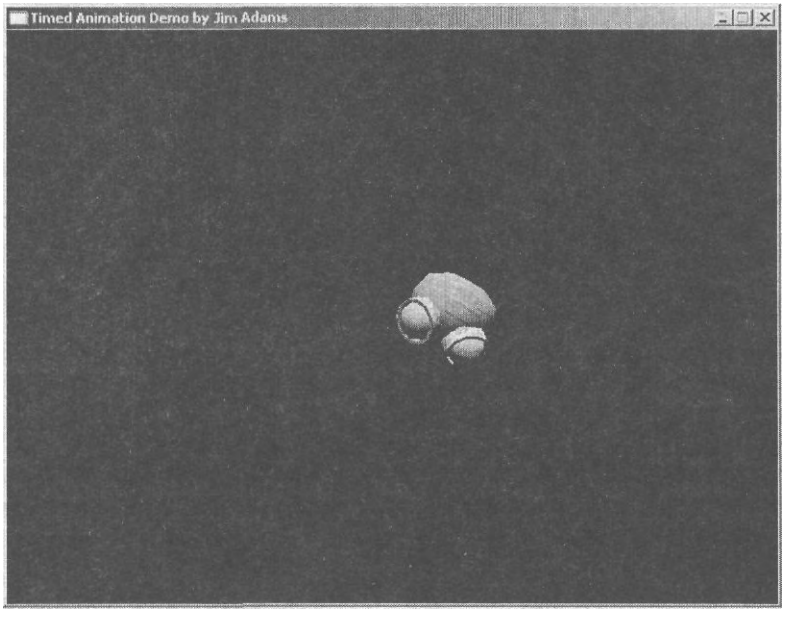


Рис. 2.9. Анимация на основе ключевых кадров в действии! Робот двигается и вращается в соответствии с преобразованиями ключевых кадров, заданными в программе

TimedMovement

Синхронизированное движение настолько же важно, насколько, синхронизированная анимация и TimedDemo показывают вам, как его использовать. Рис. 2.10 иллюстрирует работу демонстрационной программы TimedMovement в действии. Она выполняется, пока вы не закроете ее.

Демонстрационная программа TimedMovement (как и следующие далее Route и Cinematic) показывает, как определять направления движения объектов, используя для этого вектор, в направлении которого объект двигался в предыдущее обновление. Используя этот вектор движения, вы можете вычислить угол для указания правильного направления перемещения.

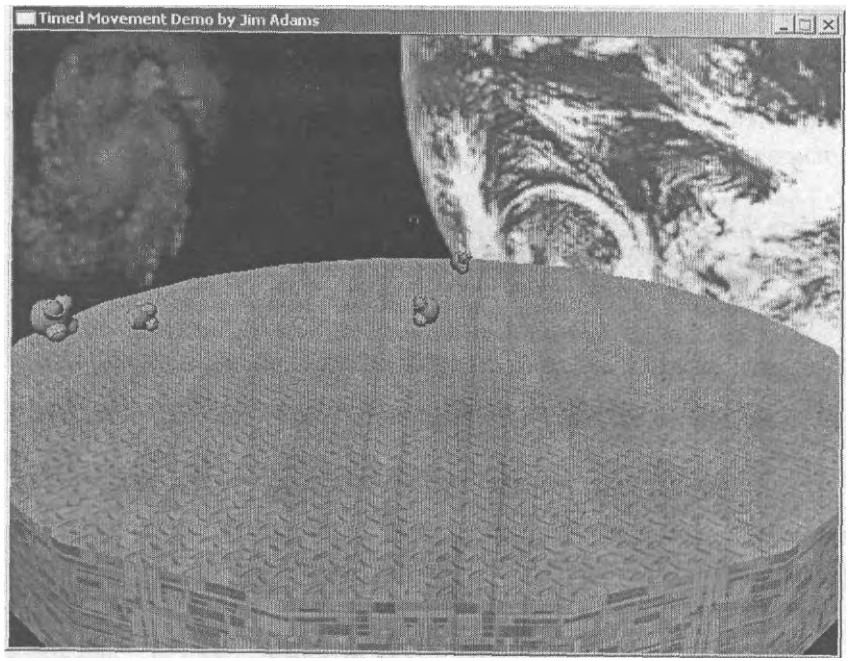


Рис. 2.10. Демонстрационная программа TimedMovement показывает, как двигать набор роботов по прямолинейным и криволинейным траекториям со временем

Route

Демонстрационная программа Route (показанная на рис. 2.11) иллюстрирует применение последовательности прямолинейных и криволинейных траекторий для создании сложного маршрута, вдоль которого вы можете передвигать ваш объект. Это программа выполняется, пока вы не выйдете из нее.

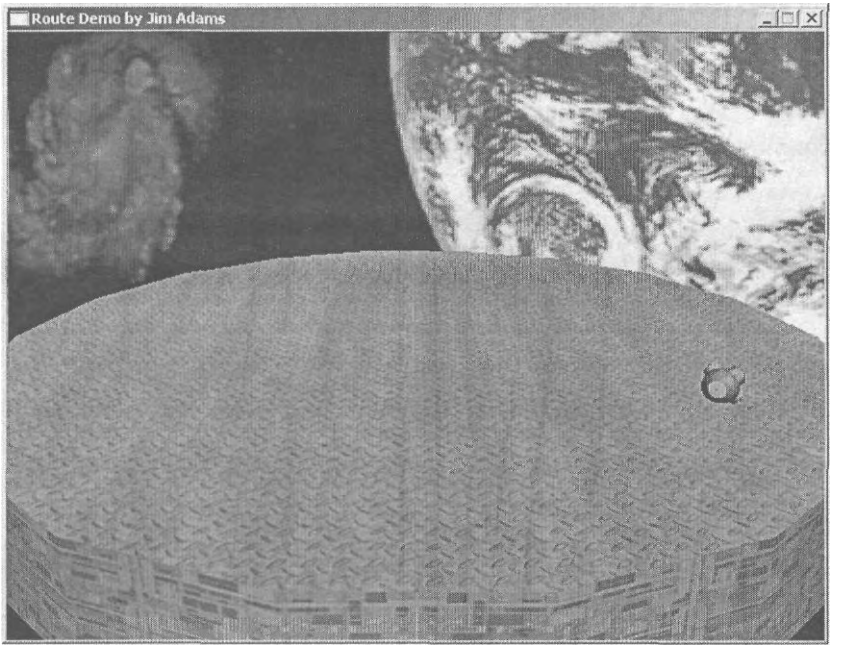


Рис. 2.11. Управляйте вашим роботом, задав ему сложный маршрут движения для перемещения в ваших мирах. В этом примере демонстрируется использование прямолинейных и криволинейных траекторий

Cinematic

Замыкает список демонстрационных программ, используемых в этой главе, Cinematics. Как показано на рис. 2.12, вы можете использовать сложные маршруты для перемещения камеры по трехмерному миру в реальном времени. Эта техника перемещения камеры великолепно подходит для создания системы игровой кинематографии.

Программы на компакт-диске

Компакт-диск содержит четыре демонстрационные программы, которые иллюстрируют технологии анимации, изученные вами в этой главе. Вот краткий обзор этих четырех программ, расположенных в директории "Chapter 2" прилагаемого диска:

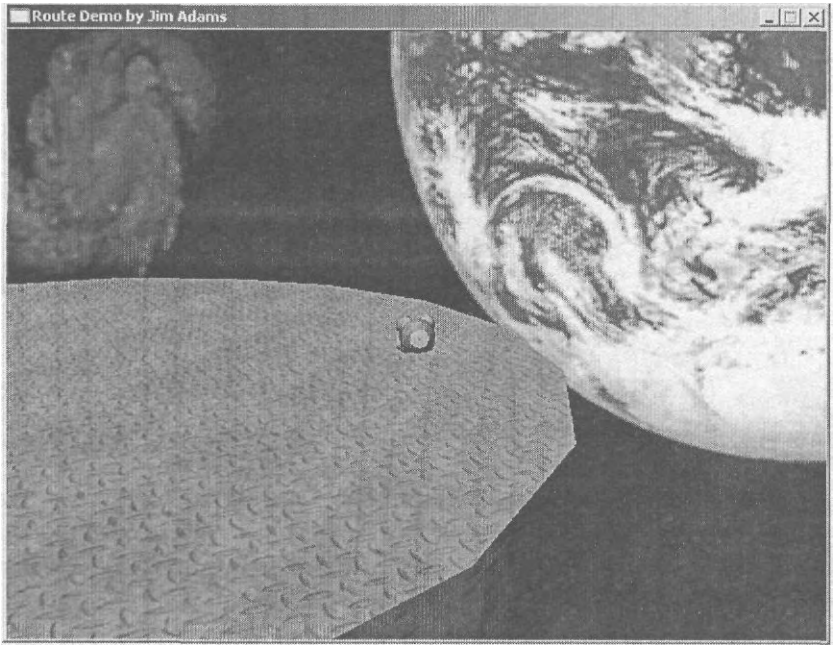


Рис. 2.12. Демонстрационный пример Cinematic добавляет движение камеры к программе Route

- **TimedAnim.** Эта программа иллюстрирует анимацию на основе времени, используя ключевые кадры. Она расположена в \BookCode\Chap02\TimedAnim.
- **TimedMovement.** Эта демонстрационная программа иллюстрирует использование прямолинейных и криволинейных траекторий для перемещения ваших объектов по трехмерному миру. Она расположена в \BookCode\Chap02\TimedMovement.
- **Route.** Эта демонстрационная программа иллюстрирует использование анализатора маршрутов для постепенного передвижения объектов по маршруту. Она расположена в \BookCode\Chap02\Route.
- **Cinematic.** Это пример игровой кинематографической последовательности, включающий камеру, следующую по маршруту, и вычисленную заранее анимацию. Он расположен в \BookCode\Chap02\Cinematic.

Глава 3

Использование формата файла .X

Вашим трехмерным мешам необходимо место обитания... или скорее вам необходимо место для хранения их данных (не беря во внимание остальные данные, требуемые вашим проектом игры). Что же делать разработчику - придумывать собственный формат файла или использовать сторонний? Имея столь широкий ассортимент популярных форматов, легко выбрать нужный, но как насчет ограничений, накладьшаемых некоторыми форматами? Почему бы вам не использовать чей-нибудь формат, изменив его для своих нужд?

Этим кто-нибудь является не кто иная, как Microsoft с ее форматом - .X! Посмотрите правде в глаза-.X файлы очень легко использовать, когда вы их поймете, и эта глава предоставит вам все необходимые сведения.

В этой главе вы научитесь:

- Использовать файлы .X в ваших игровых проектах;
- Определять и анализировать .X шаблоны;
- Создавать объекты из шаблонов;
- Загружать меши и иерархии фреймов;
- Создавать и загружать специализированные данные.

Работа с .X шаблонами и объектами данных

Если вы еще не видели, я хочу показать вам один их тех загадочных .X файлов, поставляемый с DirectX SDK (расположенный в установочной директории DirectX \Samples\Multimedia\Media). Смелее, я вам разрешаю. Более чем вероятно, вы увидите что то вроде этого:

```
xof 0302txt 0032

template Header {
  <3D82AB4 3-62DA-11cf-AB39-0020AF71E433>
  DWORD major;
  DWORD minor;
  DWORD flags;


template Frame {
  <3D82AB4 6-62DA-11cf-AB3 9-0020AF71E433>
   [FrameTransformMatrix]
   [Mesh]


Header {
  1;
  0;
  1;


Frame Scene_Root {
  FrameTransformMatrix {
    1.000000, 0.000000, 0.000000, 0.000000,
    0.000000, 1.000000, 0.000000, 0.000000,
    0.000000, 0.000000, 1.000000, 0.000000,
    0.000000, 0.000000, 0.000000, 1.000000;
  }
  Frame Pyramid_Frame {
   FrameTransformMatrix {
    1.000000, 0.000000, 0.000000, 0.000000,
    0.000000, 1.000000, 0.000000, 0.000000,
    0.000000, 0.000000, 1.000000, 0.000000,
    0.000000, 0.000000, 0.000000, 1.000000;
   }
   Mesh PyramidMesh {
    5;
    0.00000;10.00000;0.00000;,
    -10.00000;0.00000;10.00000;,
    10.00000;0.00000;10.00000;,
    -10.00000;0.00000;-10.00000;,
    10.00000;0.00000;-10.00000;;
    6;
    3;0,1,2;,
    3;0,2,3;,
    3;0,3,4;,


    3;2,4,3;;
    MeshMaterialList
      1;
      6;
      0,0,0,0,0,0;;
```

```
        Material Materia10 {
          1.000000; 1.000000; 1.000000; 1.000000;;
          0.000000;
          0.050000; 0.050000; 0.050000;;
          0.000000; 0.000000; 0.000000;;
          }
        }
      }
    }
  }
```

Выглядит пугающе, не правда ли? На самом деле вы можете разбить каждый .X файл на маленькие, привлекательные, легкоуправляемые компоненты, которые делают файл простым для понимая и обработки. Давайте я объясню, что я имею ввиду. Каждый .X файл начинается с небольшого заголовка, который, для предыдущего примера, выглядит так:

```
xof 0302txt 0032
```

Это маленькое текстовое пояснение сообщает программе, что загружаемый файл действительно является .X файлом. (Кусочек "xof " обозначает .X файл). Он также информирует программу, что файл использует шаблоны DirectX .X версии 3.2 (представленные как "0302"). После версии следует "txt", это означает, что все далее следующие данные представлены в текстовом формате, в отличие от двоичного. Строка заканчивается числом "0032", которое определяет число бит, используемых вещественными числами ("0032" для 32-битных или "0064" для 64-битных).

После заголовка файла идет последовательность наборов данных, называемых шаблонами и объектами данных. Возможность отличать шаблоны от объектов данных реализуется с помощью использования перед шаблонами слова "template". Как вы можете видеть из кода файла .X, шаблон очень похож на определение структуры языка C. Объекты данных являются экземплярами шаблонов.

Вы можете использовать шаблоны для определения информации, хранимой объектом данных в .X файле. (Шаблон определяет расположение объекта данных). Каждый шаблон может содержать любой тип данных, определенный небольшим набором типов данных, и любая комбинация типов данных может быть использована внутри шаблона. Объект данных является просто экземпляром шаблона. Вы можете думать о шаблоне в большей степени как о классе C++, они оба определяют данные, которые может содержать экземпляр объекта.

Замечание. *Второй формат хранения .Xфайлов - двоичный, полезен для представления данных в нечитаемом для человека виде. Я не буду обсуждать этот формат в книге; в любом случае, методы, применяемые для обработки Xфайлов, используемые в этой главе, употребляются и для двоичных Xфайлов, так что вы не пропустите ничего полезного!*

Посмотрев еще раз на пример .X файла, вы можете видеть, что первым встреченным шаблоном является "Header", который является названием класса шаблона. Шаблон "Header" содержит три значения DWORD (наряду с большим числом, называемым GUID, которое заключено в угольные кавычки), которые вы задаете при создании объекта из шаблона. Создание объекта во многом похоже на создание класса или структуры. В ранее упомянутом примере .X файла описание шаблона "Header" выглядит так:

```
Header {
   1; // старший
   0; // младший
   1; // флаги
}
```

Заметьте, что вы должны определять все переменные, содержащиеся в шаблоне "Header" в вашем объекте данных, причем в таком же порядке. Вас, возможно, заинтересовало большое число (шаблонный GUID), определенное в шаблоне. Как оно связано с заданием шаблона? На самом деле никак, потому что DirectX использует это большое число для идентификации шаблона, при его загрузке. Я вернусь к шаблонному GUID (Глобально Уникальное Идентификационное Число) через мгновение.

Совет. *Совсем как в C/C++ вы можете использовать оператор "//" для обозначения комментариев в .X файлах.*

Следующий шаблон, который вы увидите в .X файле, - "Frame". Это специальный шаблон, он не определяет никакого типа данных, зато ссылается на другие классы шаблонов. Другие классы шаблонов, заключенные в квадратные скобки, называются "FrameTransformMatrix" и "Mesh". Используя ссылки на другие шаблоны, вы можете создавать иерархии объектов.

Также объявляя дополнительные шаблоны внутри другого шаблона, вы создаете набор шаблонных ограничений, которые позволяют вам создавать шаблоны, которые, в свою очередь, позволяют вставку заданных объектов в другой объект. В данном случае только объекты типов "FrameTransformMatrix" и "Mesh" могут быть встроенны в объект "Frame". Более подробно о шаблонных ограничениях вы прочитаете далее в этой главе. А сейчас давайте перейдем к рассмотрению оставшейся части .X файла.

После объявления шаблонов (которое должно быть в начале .X файла), следуют объекты данных. Они объявляются совсем как структуры C - вы задаете структуру именем шаблона класса и следующим после него названием экземпляра объекта данных. Имя экземпляра является необязательным, так что не волнуйтесь, когда не найдете имени у некоторых объектов.

В рассматриваемом файле .X первый объект данных имеет имя "Scene_Root". Объект "Scene_Root " принадлежит шаблону класса "Frame". Вы уже видели определение шаблона "Frame". Вернувшись к определению того шаблона, вы можете увидеть, что он не хранит никаких данных, но имеет два необязательных встроенных объекта данных - "FrameTransformMatrix " и " Mesh".

По счастливому стечению обстоятельств оба объекта "FrameTransformMatrix " и "Mesh" встроенны в "Scene_Root". Однако в .X файле отсутствует определение шаблонов для "FrameTransformMatrix " и "Mesh". Как вы сможете узнать, какие данные содержат эти объекты? Оказывается, что не обязательно все определения шаблонов должны содержаться в .X файле, вы можете определить эти шаблоны в вашей программе!

Вы научитесь определять шаблоны внутри вашей программы далее в этой главе. А теперь вернемся к примеру. Объект данных шаблонного класса "Frame-TransformMatrix " встроен в объект "Scene_Root". Он содержит вещественные числа, которые представляют собой матрицу преобразования. Далее следует объект шаблонного класса "Mesh", который содержит информацию о меше.

Достаточно об этом примере, я уверен, что вы поняли суть. Как вы можете видеть, шаблоны полностью определяются пользователем, что означает, что вы можете создавать любые типы шаблонов, содержащие любые данные. Хотите хранить звук в .X файлах? А как насчет хранения в них кардиограммы? Используя .X файлы вы можете хранить звук, кардиограммы, любые типы данных, какие вы только захотите!

Определение шаблонов

Т. к. дизайн .X файлов является открытым, вы должны предопределять каждый используемый с DirectX шаблон, чтобы понять, как получить доступ к данным шаблона. Обычно шаблоны определяются внутри .X файла, хотя вы можете определять их и в вашей программе (как я замечал ранее).

Вы определяете шаблон (содержащийся в .X файле), назначая ему уникальное имя, следующее после слова "template", как я сделал в далее. (Обратите внимание на открывающуюся скобку, обозначающую начало определения шаблона.)

```
template ContactEntry {
```

Хорошо, вы начали объявление шаблона, который будете использовать для хранения контактной информации человека. Назовем класс шаблона "ContactEntry", как вы видите из кода. Даже если вы присвоили вашему шаблону уникальное имя класса, вам необходимо сделать еще один шаг - присвоить уникальное идентификационное число - GUID.

Когда вы будете читать .X файлы в ваших программах, у вас будет доступ только к GUID каждого шаблона, но не к их именам. Имена классов важны только для ваших объектов данных .X файлов; вам необходимо, чтобы ваша программа различала эти объекты данных, используя их шаблонный GUID.

Для определения GUID для вашего шаблона запустите программу "guidgen.exe", поставляемую вместе с компилятором "Microsoft Visual C/C++" (находящуюся в инсталяционной директории \Common\Tools вашего MSVC). После нахождения и запуска программы "guidgen.exe" перед вами появится небольшое диалоговое окно, показанное на рис. 3.1.

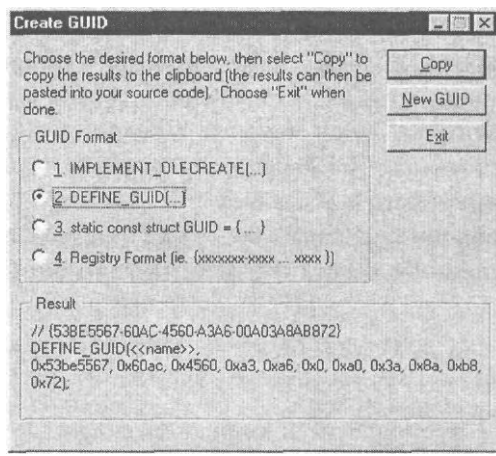


Рис. 3.1. Диалоговое окно "Create GUID" ("Создать GUID") программы "guidgen.exe" позволяет вам создавать уникальные идентификационные числа в различных форматах

Как вы можете видеть на рис. 3.1, диалоговое окно "Create GUID" позволяет вам выбрать формат создаваемого GUID. В данном случае вы должны использовать формат №2 DEFINE_GUID(...). Выберите данную опцию и нажмите кнопку "Copy" ("Копировать").

Теперь абсолютно уникальное идентификационное число находится в буфере обмена, ожидая пока вы вставите его в ваш код. Вернитесь к создаваемому .X файлу и вставьте содержимое буфера обмена в ваше объявление шаблона.

```
template ContactEntry {
// {4C9D055B-C64D-4bfe-A7D9-981F507E4 5FF}
DEFINE_GUID («name»,
0x4c9d055b, 0xc64d, 0x4bfe, 0xa7, 0xd9, 0x98, \
0x1f, 0x50, 0x7e, 0x45, 0xff);
```

Ой! Здесь немного больше текста, чем необходимо для шаблона, поэтому вырежем макрос DEFINE_GUID и вставим его в исходный код проекта. Да, все правильно - каждый определенный вами шаблон требует соответствующего определения GUID (например, используя макрос DEFINE_GUID) в вашем коде. Это означает, что вам необходимо подключить заголовочный файл "initguid.h" в ваш код и использовать DEFINE_GUID, как я сделал тут.

```
#include "initguid.h"
// В начале файла исходного кода добавьте DEFINE_GUIDs
DEFINE_GUID(ContactEntry, \
      0x4c9d055b, 0xc64d, 0x4bfe, 0xa7, 0xd9, 0x98, \
      0x1f, 0x50, 0x7e, 0x45, 0xff);
```

Заметьте, что в макросе DEFINE_GUID я заменил "«name»" на фактическое название класса шаблона, который я определяю. В данном случае я использую "ContactEntry " в качестве названия макроса. После этого макрос "ContactName" будет содержать указатель на GUID моего шаблона (который должен совпадать с GUID шаблона, используемого в .X файле).

Возвращаясь к шаблону "ContactEntry ", вам также необходимо удалить тег комментария из вставленного текста и изменить кавычки GUID на угловые, как я сделал тут:

```
template ContactEntry {
  <4C9D055B-C64D-4bfe-A7D9-981F507E45FF>
```

Теперь вы готовы продолжить и определить данные шаблоны. Шаблоны очень похожи на классы и структуры C; они содержат переменные и указатели на другие шаблоны, наряду с ограничениями доступа. Типы используемых данных практически полностью совпадают с используемыми в C. В таблице 3.1 показаны типы данных, которые вы можете использовать для определения шаблонов, и их аналоги в C/C++.

Таблица 3.1. Типы данных шаблонов .X

Тип данных	Описание
WORD	16-битное число (short)
DWORD	32-битное число (32-битное int или long)
FLOAT	вещественное число IEEE (float)
DOUBLE	64-битное вещественное число (double)
CHAR	8-битное знаковое число (signed char)

Таблица3.1.Типыданных шаблонов .X (Продолжение)

Типданных	Описание
UCHAR	8-битное беззнаковое число (unsigned char)
BYTE	8-битное беззнаковое число (unsigned char)
STRING	строка, оканчивающаяся нулем (char[ ])
Array	Массив определенного типа данных ([ ])

Совсем как в объявлении переменной C/C++ , вы указываете ключевое слово типа данных и имя экземпляра, заканчивая точкой с запятой (означает окончание объявления переменной)

```
DWORD Value;
```

В таблице 3.1 вы могли заметить ключевое слово "array", которое определяет массив элементов типа данных. Для определения массива вы задаете ключевое слово "array", тип данных, имя экземпляра и размер массива (заключенный в квадратные скобки). Например, для объявления массива из 20 элементов типа "STRING" вы можете использовать

```
array STRING Text[20];
```

Замечание. *Замечательным свойством массивов является возможность использования другого типа данных для определения размера массива, как я сделал тут:*

```
DWORD ArraySize;
array STRING Names[ArraySize];
```

Теперь вернемся к шаблону "ContactEntry" и определим имя человека, телефон и возраст. Эти три переменные - две строки (имя и номер телефона) и одно численное значение (возраст) - могут быть определены в шаблоне "ContactEntry" как показано здесь.

```
template ContactEntry {
  <4C9D055B-C64D-4bfe-A7D9-981F507E45FF>
  STRING Name; // имя
  STRING PhoneNumber; // телефон
  DWORD Age; // возраст
}
```

Замечательно! Чтобы закончить объявление шаблона, добавьте закрывающуюся скобку, и он готов к использованию.

Создание объектов данных из шаблонов

После определения шаблона вы можете начать создание объектов и определение их данных. Объекты данных определяются, используя соответствующие шаблоны и произвольное имя экземпляра. Вы можете использовать имя экземпляра для дальнейшего доступа к объекту из вашего .X файла или проекта (об этой особенности будет рассказано далее в этой главе).

Продолжая пример, создайте объект, используя шаблон "ContactEntry". Этот объект будет содержать имя, телефон и возраст.

```
ContactEntry JimsEntry {
  "Jim Adams";
  "(800) 555-1212";
  30;
}
```

Заметьте, что я присвоил экземпляру объекта имя "JimsEntry". Теперь я могу получить доступ к этому объекту, используя его имя, заключенное в скобки, вот так:

```
{JimsEntry}
```

Ссылка на объект таким способом называется адресация данных или просто адресация (как будто вы сами не могли догадаться!), и она позволяет указывать одни объекты другим. Например, шаблон последовательности анимации (AnimationSet) требует указателя на объект "Frame" для последовательности встроенных объектов.

Вы также можете использовать адресацию для копирования данных объекта без его изменения. Это полезно при создании нескольких одинаковых объектов "Mesh" в ваших .X файлах, отличающихся только положением при использовании различных объектов "Frame".

Вставка объектов данных и ограничения шаблонов

Адресация данных имеет один нюанс - ограничения шаблона должны позволять ее использовать. Сначала это может показаться неважным, но вы не можете использовать адресацию данных без корректных ограничений. .X файл представляет собой полную иерархию объектов, которые могут быть либо родственниками либо потомками других объектов. Поэтому для адресации и создания встроенных объектов необходимы корректные ограничения.

В качестве примера рассмотрите следующие три объявления шаблона:

```
template ClosedTemplate {
  <4C9D055B-C64D-4bfe-A7D9-981F507E45FF>
  DWORD ClosedData;
```

```
template OpenTemplate {
  <4C9D055B-C64D-4bff-A7D9-981F507E45FF>
  DWORD OpenData;
   [...]
}

template RestrictedTemplate {
  <4C9D055B-C64D-4c00-A7D9-981F507E45FF>
  DWORD RestrictedData;
   [ClosedTeMplate]
   [OpenTemplate]
}
```

Это стандартное объявление шаблонов, за исключением строк, содержащих квадратные скобки. Информация, содержащаяся в этих скобках очень важна. Первый шаблон, "ClosedTemplate", не имеет квадратных кавычек, так что он считается закрытым. Вы можете создавать и определять значение "ClosedData" только внутри "ClosedTemplate".

В то время как "OpenTemplate" содержит строку "[…]", которая означает что шаблон открытый. Открытый шаблон позволяет вставить любой объект вместо строки "[…]". Например, вы можете создать "OpenTemplate", определить переменную "OpenData" и вставить экземпляр объекта "ClosedTemplate" в "OpenTemplate".

В "RestrictedTemplate" имеется две строки со скобками. Ограничение шаблонов позволяет вставлять шаблоны только указанного типа; в данном случае ими являются "ClosedTemplate" и "OpenTemplate". Попытка вставить объект любого другого типа закончится неудачей (вызывая завершение работы анализатора).

Гмм, возможно вам придется перечитать этот раздел несколько раз, прежде чем вы поймете механизм вставки ограниченных шаблонов в другие шаблоны. Как только вы убедитесь, что твердо усвоили вышеизложенный материал, можно переходить к изучению стандартных шаблонов DirectX, которые поставляются с DirectX SDK.

Работа со стандартными шаблонами DirectX

После того как вы познакомились с шаблонами и объектами данных, пришло время рассмотреть, как можно использовать их в своих проектах. Если вы уже знакомы с DirectX SDK, то заметили, что .X файлы широко используются для хранения информации о меше. Поэтому Microsoft снабдила DirectX некоторым количеством шаблонов, которые я называю: стандартные шаблоны DirectX. Эти шаблоны используются для хранения информации, связанной с мешами.

Стандартные шаблоны полезны, т. к. они определяют почти все аспекты трехмерных мешей, так что уделите немного времени на их изучение. Я не буду очень подробно останавливаться на их описании, потому что DirectX SDK содержит множество информации о них, но я приведу краткий обзор каждого шаблона.

Стандартные шаблоны, приведенные в таблице 3.2, имеют соответствующий макрос, который вы применяете для определения используемого в вашей программе объекта данных. Эти макросы определены (используя DEFINE_GUID) в специальном файле - " rmxfguid.h ". Макросы GUID стандартных шаблонов очень легко запомнить, т. к. они просто имеют префикс перед своим именем - "D3DRM_TID". Например, шаблон "Animation" определяется, используя макрос "D3DRM_TIDAnimation".

Таблица 3.2. Стандартные шаблоны DirectX

Имя шаблона	Описание
Animation	Определяет данные анимации для одного кадра
AnimationKey	Определяет один ключевой кадр для родительского анимационного шаблона
AnimationOptions	Содержит информацию о проигрывании анимации
AnimationSet	Содержит набор шаблонов анимации
Boolean	Содержит логическое значение
Boolean2d	Содержит два логических значения
ColorRGB	Содержит красную, зеленую и синюю компоненты цвета
ColorRGBA	Содержит красную, зеленую и синюю компоненты и значение прозрачности цвета
Coords2d	Определяет два значения координат
FloatKeys	Содержит массив вещественных значений
FrameTransformMatrix	Содержит матрицу преобразования для родительского шаблона "Frame"
Frame	Шаблон кадра ссылок, который определяет иерархию
Header	Заголовок .X файла определяющий версию
IndexedColor	Содержит индексированные значения цвета
Material	Содержит значения цветов материала
Matrix4x4	Содержит матрицу 4x4
Mesh	Содержит данные одного меша
MeshFace	Содержит данные граней меша
MeshFaceWraps	Содержит наложение текстур для граней меша

Таблица 3.2. Стандартные шаблоны DirectX

Имя шаблона	Описание
MeshMaterialList	Содержит список материалов меша
MeshNormals	Содержит нормали, используемые мешем
MeshTextureCoords	Содержит текстурные координаты, используемые мешем
MeshVertexColors	Содержит информацию о цвете вершин меша
Patch	Определяет управляющую сетку
PatchMesh	Содержит сеточный меш (совсем как шаблон "Mesh")
Quaternion	Содержит значение кватерниона
SkinWeights	Содержит массив весов, накладываемых на вершины меша. Используется для скелетных мешей
TextureFilename	Содержит имя текстуры, используемой в материале
TimedFloatKeys	Содержит массив шаблонов "FloatKeys"
Vector	Содержит координаты трехмерной точки
VertexDuplicationIndices	Сообщает, что вы хотите скопировать вершины из других вершин
XskinMeshHeader	Используется скелетным мешем для определения количества костей

Вы видите, что стандартных шаблонов много - слишком много, чтобы обсуждать их в этой книге. К счастью, вы обнаружите, что вам придется иметь дело с небольшим количеством стандартных шаблонов при анализировании .X файлов. Вы поймете о каких шаблонах я говорю, когда дочитаете книгу. А теперь, давайте продолжим, рассмотрев, как получить доступ к .X файлам в ваших проектах.

Доступ к .X файлам

Независимо от используемой версии DirectX (DirectX 8 или 9), методы, применяемые для доступа к .X файлам, одинаковы. На самом деле интерфейсы имеют одинаковые имена в последних версиях DirectX (8 и 9), что делает возможным быстро переносить ваш код версии 8 в новую версию 9 (и наоборот, если необходимо).

Первым шагом к получению доступа к .X файлам является создание интерфейса "IDirectXFile". Для этого вам необходимо вызвать функцию "DirectXFileCreate", как показано в следующем кусочке кода:

```
IDirectXFile *pDXFile = NULL;
HRESULT Result = DirectXFileCreate(&pDXFile);
```

Как вы можете видеть из предыдущих строк кода у функции "DirectXFileCreate" только один параметр - указатель на интерфейс "IDirectXFile". Вы можете быстро определить, был ли удачно создан интерфейс "IDirectXFile", используя макросы SUCCEEDED или FAILED для обработки кода возвращаемого "DirectXFileCreate".

После того как вы успешно создали интерфейс IdirectXFile, вы можете зарегистрировать любые используемые шаблоны (такие как стандартные шаблоны DirectX) и создать объект перечисления, который связывает ваш основной объект данных с .X файлом.

Регистрация специализированных и стандартных шаблонов

Чтобы сократить используемое место и улучшить безопасность, интерфейсы .X позволяют вам убрать все объявления шаблонов из .X файлов и вставить их в исполняемый модуль. Это означает, что вместо того, чтобы определять шаблоны в .X файлах, необходимо определять их в программе. Не волнуйтесь - это не настолько сложно, насколько звучит. Как вы увидите чуть позже, Microsoft взяла на себя всю тяжелую работу, определив стандартные шаблоны в нескольких заголовочных файлах, упростив все насколько это возможно.

Чтобы зарегистрировать стандартный шаблон (или любой другой) в вашей программе, вам необходимо вызвать функцию IDirectXFile::RegisterTemplates.

```
HRESULT  IDirectXFile::RegisterTemplates(
  LPVOID pvData,  // буфер содержащий определение шаблона
  DWORD cbSize);  // количество байт информации
```

Параметр pvData является просто буфером данных, который содержит определение шаблона в таком же формате, как вы видели в .X файле. Например, вы можете определить буфер шаблона так:

```
char *TeMplates = "
     "xof 0303txt 0032 \
     template CustoMTemplate { \
       <4c944580-9e9a-11cf-ab43-0120af71e433> \
       DWORD Length; \
       array DWORD Values[Length]; \
     }";
```

Возвращаясь к функции RegisterTemplates, нужно сказать, что параметр cbSize представляет собой размер буфера данных шаблона, который в настоящем случае вы определяете, используя функцию strlen буфера "Templates". Соединив,

вы можете зарегистрировать шаблон, определенный в буфере "Templates", используя следующий код:

```
pFile->RegisterTemplates(Templates, strlen(Templates));
```

Вернемся к самой теме - регистрирование стандартных шаблонов. Вы видели, как работает функция RegisterTemplates. Для того чтобы зарегистрировать стандартные шаблоны, вам необходимо включить два добавочных файла в ваш проект - "rmxftmpl.h" и "rmxfguid.h". Эти два файла содержат определения и GUID стандартных шаблонов соответственно.

Внутри файла "rmxftmpl.h" вы найдете буфер данных шаблона D3DRMXTEMPLATES и макрос D3DRM_XTEMPLATE_BYTES. Они используются при вызове RegisterTemplates для регистрирования стандартных шаблонов, как вы можете видеть здесь:

```
pFile->RegisterTemplates(D3DRM_XTEMPLATES, \
                         D3DRM_XTEMPLATE_BYTES);
```

Правильно, просто вызвав приведенный выше кусочек кода, вы успешно зарегистрируете стандартные шаблоны, и вы готовы продолжать! Но сначала небольшой совет: как только вы начнете использовать формат .X для ваших специализированных данных и шаблонов, не забывайте, что RegisterTemplates отлично регистрирует и ваши шаблоны!

Замечание. *Отметьте, что в определении шаблона "Templates" используется наклонная черта влево для обозначения новой линии, и после этого следует стандартный заголовок .X файла.*

Совет. *Для того, чтобы запомнить "rmxftmpl.h" и "rmxfguid.h", просто помните, что "rmxf" это .X файл сохраненного режима, "tmpl" означает шаблон и "guid" означает глобально уникальный идентификатор.*

Открытие .X файла

После того как вы создали интерфейс IDirectXFile и зарегистрировали используемые шаблоны, вам необходимо открыть .X файл и начать просмотр содержащихся в нем объектов. Процессы открытия .X файла и создания объекта просмотра объединены в один, соответствующий вызову функции IDirectXFile::CreateEnumObject.

```
HRESULT IDirectXfile::CreateEnumObject(
  LPVOID pvSource, // имя .X файла
  DXFILELOADOPTIONS dwLoadOptions, // параметры загрузки
  LPDIRECTXFILEENUMOBJECT* ppEnumObj}; // интерфейс перечисления
```

Когда вы вызываете функцию CreateEnumObject, укажите ей в качестве параметра pvSource имя загружаемого .X файла, а в качестве ppEnumObj используемый интерфейс. Что же касается dwLoadOptions, вы должны задать значение DXFILELOAD-_FROMFILE, которое сообщает DirectX, что файл загружается с диска. Другие возможные значения dwLoadOptions это DXFILELOAD_FROMRESOURCE, DXFILELOAD_FROMMEMORY и DXFILELOAD_FROMURL. Эти значения говорят DirectX загружать файл из ресурсов, буфера памяти или сетевого адреса соответственно. Да, так и есть - вы можете загружать .X файл прямо из Интернета.

Совет. *Для загрузки .X файла из Интернета, используя адрес, задайте полный сетевой путь в "pvSource". Для загрузки из ресурса или памяти просто укажите дескриптор ресурса или указатель памяти (оба преобразуются к LPVOID) в pvSource.*

Продолжим пример и создадим объект перечисления для .X файла. Следующий код создает объект перечисления, используемый для анализа дискового файла.

```
// Filename = имя загружаемого файла (например "test.x")
IDirectXFileEnumObject *pEnum;
pFile->CreateEnumObject((LPVOID)Filename, \
             DXFILELOAD_FROMFILE, &pEnum);
```

Из комментариев кода видно, что "Filename" указывает на правильное имя файла - в данном случае "test.x". После успешного вызова функция CreateEnumObject возвращает вам правильный объект перечисления (необходим один на один открытый файл), готовый к перечислению объектов находящихся в файле.

Перечисление объектов данных

На данный момент вы открыли .X файл и зарегистрировали используемые шаблоны (такие как стандартные шаблоны DirectX). Создали объект перечисления и теперь готовы к извлечению данных из .X файла.

На данный момент созданный вами объект IdirectXFileEnumObject указывает на первый объект данных в файле, которым обычно является "Header". Все объекты данных верхнего уровня являются родственниками объекта "Header" (или первого объекта в файле). Каждый прочитанный вами объект данных может содержать встроенные (дочерние) объекты или ссылки на другие объекты, которые вы можете получить.

Объект перечисления сам по себе не содержит объектов данных. Вместо этого вам необходимо получить интерфейс IDirectXFileData для доступа к данным. Чтобы получить интерфейс IDirectXFileData, вам необходимо вызвать функцию IDirectXFileEnumObject::GetNextDataObject.

```
HRESULT IDirectXFileEnumObject::GetNextDataObject( \
          LPDIRECTXFILEDATA* ppDataObj);
```

Имея всего лишь один параметр, GetNextDataObject очень проста в использовании. Вам просто необходимо создать экземпляр объекта IDirectXFileData и использовать его при вызове GetNextDataObject.

```
IDirectXFileData *pData;
HRESULT hr = pEnum->GetNextDataObject(&pData);
```

Заметили, как я сохраняю значение, возвращаемое функцией GetNextDataObject? Если возвращаемый результат является ошибкой (что проверяется макросом FAILED), то это означает, что перечисление закончено. Если же вызов функции GetNextDataObject успешен, тогда вы получаете новый интерфейс для доступа к данным объекта!

Прежде чем вы начнете работать с данными объекта, давайте закончим обсуждение перечисления. Пока что вы можете получить первый объект в файле и его интерфейс данных. Что делать, если вы хотите получить остальные объекты из .X файла или встроенные объекты?

После того как вы закончили с одним объектом, вам необходимо освободить его и перейти к следующему объекту. Простой вызов IDirectXFileData::Release освободит интерфейс данных, и повторяющимися вызовами IDirectXFileEnumObject::GetNextDataObject вы можете получить следующий перечисленный объект данных родственника (высокоуровневого). Вы можете представить перечисление всех родственников (получая их интерфейсы данных) с помощью следующего кода:

```
while(SUCCEEDED(pEnum->GetNextDataObject(&pData))) {
  // Сделать что-нибудь с объектом pData

  // Освободить интерфейс данных для продолжения
  pData->Release();
}
```

Все что остается, это добавить возможность получать дочерние (нижнеуровневые) объекты, перечислить их и сделать доступными. Для получения дочернего объекта используйте сначала функцию IDirectXFileData::GetNextObject, чтобы посмотреть содержит ли объект встроенные объекты.

```
HRESULT IDirectXFileData::GetNextObject ( \
          LPDIRECTXFILEOBJECT* ppChildObj);
```

Это еще одна простая функция с одним параметром - указателем на интерфейс IDirectXFileObject. Если вызов GetNextObject был успешным, тогда вам необходимо обработать дочерний объект. После того как вы завершите это, можете освободить его

(вызвав Release) и продолжать вызывать GetNextObject, пока она не вернет ошибку, что означает, что больше не осталось объектов.

Вы можете реализовать циклические вызовы GetNextObject, как я сделал здесь

```
IDirectXFileObject *pObject;

while(SUCCEEDED(pData->GetNextObject(&pObject))) {
  // Дочерний объект существует, необходимо получить его

  // Освободить интерфейс объекта файла
  pObject->Release();
}
```

После того как вы получили правильный интерфейс IDirectFileObject (после вызова GetNextObject), вы можете быстро определить, какой дочерний объект перечисляется (используя приемы описанные ниже). Однако есть небольшое препятствие. Объект может быть как ссылкой, так и экземпляром, и манера доступа к нему зависит от его типа.

Для экземпляров объектов (которые определяются обычно в .X файлах) вы можете получить IDirectXFileObject для интерфейса IDirectXFileData.

```
IDirectXFileData *pSubData;

//Проверить,является ли дочерний объект экземпляром(если нет,то ошибка)
if(SUCCEEDED(pObject->QueryInterface( \
      IID_IDirectXFileData,(void**)&pSubData))) {
  // Дочерний объект существует, сделать что-нибудь с ним

  // Освободить объект
  pSubData->Release();
}
```

Вы видели, как использовать объект IDirectXFileData раннее в этой главе. Используя только что приобретенные навыки, вы можете получать дочерние объекты IDirectXFileData для их собственных встроенных дочерних объектов.

Что касается ссылочных объектов, вам необходимо сначала получить объект IDirectXFileDataReference, а затем переделать ссылку в объект IDirectXFileData. Нижеследующий код преобразовывает ссылки на объекты для вас.

Совет. *Если экземпляр объекта не существует, когда вы его получаете, вызов QueryInterface окончится неудачей. Это быстрый способ определения типа объекта. Таким же образом определяются ссылочные объекты — запрос окончится неудачей, если объект не ссылочный.*

```
IDirectXFileDataReference *pRef;
IDirectXFileData *pSubData;

// Проверить является ли объект ссылочным (если нет, то неудача)
if(SUCCEEDED(pSubObj->QueryInterface( \
```

```
        IID_IDirectXFileDataReference, \
        (void**)&pRef))) {
    // Ссылочный объект существует. Преобразовать ссылку
    pRef->Resolve(&pSubData);

    // Сделать что-нибудь с объектом

    // Освободить используемый интерфейс
    pRef->Release();
    pSubData->Release();
  }
```

Вы поверите мне, если я скажу, что самая сложная часть закончена? Перечисление объектов данных и дочерних объектов очень просто, и если это так же сложно, как оно понимается, тогда вам будет все просто! Чтобы облегчить вашу работу программиста, я предлагаю реализовать перечисление объектов данных в двух простых функциях.

Первая функция (названная Parse) будет открывать .X файл, создавать объект перечисления и просматривать все объекты верхнего уровня. Функция будет брать каждый перечисленный объект и передавать его второй функции (ParseObject), которая будет обрабатывать данные объектов на основе типа их шаблона и просматривать встроенные дочерние объекты. Функция ParseObject будет вызывать себя всякий раз, когда найдет дочерний объект, таким образом обрабатывая встроенные в дочерние объекты.

Вот код функции Parse.

```
// Need to include rmxftmpl.h and rmxfguid.h
// Необходимо подключить "rmxftmpl.h" и "rmxfguid.h"
BOOL Parse(char *Filename)
{
  IDirectXFile *pFile = NULL;
  IDirectXFileEnumObject *pEnum = NULL;
  IDirectXFileData *pData = NULL;

  // Создать объект перечисления, вернуть ошибку
  if(FAILED(DirectXFileCreate(&pFile)))
    return FALSE;

  // Зарегистрировать стандартные шаблоны, вернуть ошибку
  if(FAILED(pFile->RegisterTemplates( \
     (LPVOID)D3DRM_XTEMPLATES, D3DRM_XTEMPLATE_BYTES)))
   return FALSE;

  // Создать объект перечисления, вернуть ошибку
  if(FAILED(pDXFile->CreateEnumObject((LPVOID)Filename, \
            DXFILELOAD_FROMFILE, \
            &pEnum))) {
   pFile->Release();
   return FALSE;
```

```
    // Пройтись по всем верхнеуровневым объектам
    while(SUCCEEDED(pEnum->GetNextDataObject(&pData))) {
      // Анализировать объект, вызвав ParseObject
      ParseObject(pData);

      // Освободить объект
      pData->Release();
    }

    // Освободить использованные COM объекты
    pEnum->Release();
    pFile->Release();

    return TRUE;
}
```

Функция Parse не содержит ничего особенного и, конечно же, не является очень сложной. Я уже объяснил все аспекты, встречающиеся в функции, так что нет нужды повторяться. Вместо этого перейдем к функции ParseObject, которая берет объект и просматривает его дочерние объекты.

```
void ParseObject(IDirectXFileData *pData)
{
  IDirectXFileObject *pObject = NULL;
  IDirectXFileData *pSubData = NULL;
  IDirectXFileDataReference *pRef = NULL;

  // Просмотр встроенных объектов
  while(SUCCEEDED(pData->GetNextObject(&pObject))) {

    // Искать ссылочные объекты
    if(SUCCEEDED(pObject->QueryInterface( \
      IID_IDirectXFileDataReference, (void**)&pRef))) {

     // Преобразовать объект
     pRef->Resolve(&pSubData);

     // Анализировать объект, вызвав ParseObject
     ParseObject(pSubData);

     // Освободить интерфейсы объектов
     pSubData->Release();
     pRef->Release();
    }

    // Искать экземплярные объекты
    if(SUCCEEDED(pObject->QueryInterface( \
       IID_IDirectXFileData, (void**)&pSubData))) {

    // Анализировать объект, вызвав ParseObject
    ParseObject(pSubData);
```

```
      // Освободить интерфейсы объектов
      pSubData->Release  ();
     }

    // Освободить интерфейс для следующего используемого объекта
    pObject->Release();
   }
}
```

Опять же, функция ParseObject не содержит ничего нового. Единственное что вы заметите в этих двух функциях это то, что они на самом деле не делают ничего кроме перечисления всех объектов находящихся в .X файле. Когда придет время работать с данными объекта, что вы будете делать?

Получение данных объекта

Помните, что объекты данных являются контейнерами для данных, и если у вас возникают трудности с перечислением объектов данных, разумно предполагать, что вы имеете данные в каждом их них. После того как вы получили правильный объект IDirectXFileData, который указывает на перечисленный объект данных, вы можете получит имя экземпляра объекта, GUID шаблона и данные, используя три функции.

Первая функция IDirectXFileData::GetName получает имя экземпляра объекта данных.

```
HRESULT IDirectXFileData::GetName(
  LPSTR pstrNameBuf, // Буфер имени
  LPDWORD pdwBufLen); // Размер буфера имени
```

Функция GetName имеет два параметра - указатель на буфер, содержащий имя, и указатель на переменную, содержащую размер буфера имени (в байтах). Прежде чем получать имя, используя функцию GetName, вам необходимо получить его размер, задав значение NULL в качестве pstrNameBuf и указатель на DWORD для pdwBufLen.

```
// pData = загруженный объект IDirectXFileData
// Получить размер имени в байтах
DWORD Size;
pData->GetName(NULL,  &Size);
```

После того как вы получили размер буфера имени, вы можете создать соответствующий буфер и считать имя.

```
// Создать буфер имени и получить его
char *Name = new char[Size];
pData->GetName(Name,  &Size);
```

Хотя имя экземпляра объекта и помогает, на самом деле вам необходим GUID шаблона объекта, для определения, какой шаблон использует объект. Для получения GUID шаблона объекта используйте функцию IDirectXFileData::GetType.

```
HRESULT IDirectXFileData::GetType(
  const GUID ** ppguid);
```

Используя только один параметр - указатель на указатель const GUID, вы можете вызвать функцию GetType, используя следующий код:

```
const GUID *TemplateGUID = NULL;
pData->GetType(&TemplateGUID);
```

После того как вы получили GUID, вы можете сравнить его с внешними GUID (например, с GUID стандартных или ваших специализированных шаблонов) и соответствующим образом обрабатывать данные. Например, для проверки является ли объект данных шаблоном "MeshNormals", вы можете использовать следующий код:

```
// TemplateGUID = GUID проверяемого шаблона
if(*TemplateGUID == TID_D3DRMMeshNormals) {
  // Обработать шаблон MeshNormals
}
```

Конечно, знание GUID шаблона объекта может привести вас только сюда. Настоящим фокусом является получение данных объекта. Никаких проблем! Используя еще одну простую функцию, анализирующие способности будут практически завершены! Последняя используемая функция, позволяющая вам получать доступ к данным объекта, это GetData.

```
HRESULT IDirectXFileData::GetData(
  LPCSTR szMember, // Установите в NULL
  DWORD *pcbSize, // Размер данных
  void **ppvData); // Указатель на данные
```

Для использования функции GetData вам необходимо предоставить указатель для доступа к буферу данных объекта и переменную DWORD, чтобы хранить размер буфера (в байтах). Вот небольшой код, который демонстрирует использование GetData для получения указателя на данные объекта и их размер.

```
char *DataPtr;
DWORD DataSize;
pData->GetData(NULL, &DataSize, (void**)&DataPtr);
```

Указатель на буфер данных теперь указывает на непрерывный блок памяти, который имеет такую же структуру, как и определение шаблона объекта. Вы можете получить доступ к данным как к большому буферу или, если слукавить, можно

создать структуру, соответствующую определению шаблона, для более простого доступа. Например, предположим, что вы нашли стандартный шаблон ColorRGBA, который определен так:

```
template ColorRGBA {
  <35FF44E0-6C7C-11cf-8F52-0040333594A3>
  FLOAT red;
  FLOAT green;
  FLOAT blue;
  FLOAT alpha;
}
```

Чтобы получить доступ к значениям red, green, blue и alpha, вам необходимо получить указатель и преобразовать его к типу float.

```
DWORD DataSize;
float *DataPtr;
pData->GetData(NULL, &DataSize, (void**)&DataPtr);
float red = *DataPtr++;
float green = *DataPtr++;
float blue = *DataPtr++;
float alpha = *DataPtr++;
```

Хотя это нормальный подход, вы можете обрабатывать данные объекта более простым способом, используя соответствующую структуру C.

```
typedef struct {
  float red, green, blue, alpha;
} sColorRGBA;
sColorRGBA *Color;
DWORD DataSize;
pData->GetData(NULL, &DataSize, (void**)&Color);
```

После реализации, предыдущий код дает вам возможность получать доступ к цветам, используя экземпляр структуры.

```
float red = Color->red;
float blue = Color->blue;
float green = Color->green;
float alpha = Color->alpha;
```

Получать доступ к одиночным переменным просто, а как насчет строк и массивов? Массивы, как самый простой их двух случаев, хранятся непрерывно в памяти, что означает, что вы можете просто увеличивать указатель памяти, который содержит данные объекта. Например, нижеследующий код показывает, как получить доступ к массиву вещественных значений, хранимых в объекте, использующем шаблон FloatKeys.

Замечание. *Обратите внимание, что GUID шаблона или имя класса не являются частью получаемых данных, используя функцию IDirectXFileData::GetData.*

```
// Получить размер данных объекта и указатель
DWORD DataSize;
DWORD *DataPtr;
pData->GetData(NULL, &DataSize, (void**)&DataPtr);

// Шаблон FloatKeys имеет значение DWORD вначале
// определяющее, как много вещественных значений в массиве
DWORD NumKeys = *DataPtr++;

// Next, an array of float values follows
for(DWORD i=0;i<NumKeys;i++) {
  float fValue = *(FLOAT*)DataPtr++;
```

Получить доступ к массивам оказалось не слишком сложно, а как насчет получения доступа к строкам? Опять же это простая работа, потому что строки хранятся как указатели на текстовые буферы, к которым вы можете получить доступ, как я сделал в следующем коде. (Я использую шаблон TextureFilename в качестве примера; он хранит имя файла текстуры.)

```
// Получить указатель на данные и их размер
DWORD DataSize;
DWORD *DataPtr;
pData->GetData(NULL, &DataSize, (void**)&DataPtr);

// Получить доступ к текстовому буферу имени файла
char *StringPtr = (char*)*DataPtr;
MessageBox(NULL, StringPtr, "Texture Filename", MB_OK);
```

Простым приведением указателя к типу char мы смогли отобразить имя файла, содержащегося в шаблоне TextureFilename. Я знаю, что теперь вы, наверное, кричите "почему я сразу не понял, как просто это?". Спокойно! Я тоже не сразу понял насколько просто работать с .X файлами. После того как секрет раскрыт, ничто не сможет остановить вас от использования .X файлов в ваших проектах. Вам просто необходимо реализовать всю функциональность анализатора .X в виде класса, еще более упростив работу с .X.

Создание класса .X анализатора

Итак, вы хотите создать класс, который бы реализовывал все аспекты обработки .X файлов, да? Звучит замечательно! В классе анализатора .X файлов вы можете реализовать функции Parse и ParseObject, которые вы видели ранее в этой главе, в разделе "Перечисление объектов данных". Используя код этих двух функций, напишите класс анализатора, чтобы вы могли перегружать функции анализирования объектов, что позволит искать заданные объекты.

Начнем класс анализатора с простого описания.

```
class cXParser
{
  protected:
    // Функция, вызываемая для каждого найденного шаблона
    virtual BOOL ParseObject( \
        IDirectXFileData *pDataObj, \
        IDirectXFileData *pParentDataObj, \
        DWORD Depth, \
        void **Data, BOOL Reference)
      {
       return ParseChildObjects(pDataObj, Depth, \
              Data, Reference);
      }

    // Функция, вызываемая для перечисления дочерних шаблонов
    BOOL ParseChildObjects(IDirectXFileData *pDataObj, \
        DWORD Depth, void **Data, \
        BOOL ForceReference = FALSE);

  public:
     // Функция начала анализирования .X файла
     BOOL Parse(char *Filename, void **Data = NULL);
};
```

Стоп! Я знаю, что я сказал вам начать с простого объявления, а не с того, что я показал тут! Не спорьте со мной, потому что вы быстро осознаете, как прост будет этот класс. Пока что у вас есть эти три функции в вашем новом классе анализизатора .X файлов cXParser. Вы можете использовать эти три функции (ParseObject, ParseChildObjects и Parse) для обработки одного объекта, поиска встроенных объектов или анализирования всего файла соответственно.

Самая простая из функций cXParser::Parse просто повторяет код ранее приведенной функции Parse. Я не стал приводить здесь ее код, но если вы заглянете в него на компакт-диске (\BookCode\Common\XParser.cpp и XParser.h), вы заметите добавление предполагаемого указателя данных и несколько строк кода, содержащих вызовы двух неизвестных функций BeginParse и EndParse. Я расскажу вам о них немного позднее, а пока просто пропустим их.

Вторая функция ParseObject является рабочей лошадкой вашего анализатора .X файлов. ParseObject вызывается для каждого объекта, найденного в .X файле. Вам необходимо перегрузить функцию ParseObject (она является виртуальной), чтобы она могла выполнять что-нибудь полезное. Как вы можете видеть из прототипа функции ParseObject, много чего поменялось, что требует объяснений.

Первым параметром ParseObject является объект IDirectXFileData, который, как вы видели ранее в этой главе, представляет собой объект данных, который просматривается в данный момент. Внутри перегружаемой функции вы можете получить доступ к этому объекту, используя указатель pDataObj.

Второй параметр, pParentDataObj (также объект IDirectXFileData), представляет родителя (объект более высокого уровня) текущего перечисляемого объекта. Он необходим, если вы захотите посмотреть является ли текущий объект потомком какого-либо другого.

Параметр Depth измеряет глубину объекта в иерархии. Объекты самого высоко уровня имеют глубину 0, в то время как у объектов потомков она на один больше. В качестве примера, я показал несколько объектов Frame и их соответствующие глубины.

```
Frame RootFrame { // Depth = 0
  Frame ChildofRoot { // Depth = 1
    Frame ChildofChild { // Depth = 2
    }
  }
  Frame SiblingofRootChild { // Depth = 1
  }
}
Frame RootSibling { // Depth = 0
}
```

Data является четвертым параметром ParseObject. Это определяемый пользователем указатель (или даже указатель на указатель данных) который вы задаете для передачи информации функции анализирования. Например, вы можете создать структуру, содержащую всю необходимую информацию.

```
typedef struct sDATA {
  long NumObjects;
  sDATA() { NumObjects = 0 ; }
} sDATA;
```

Чтобы передать структуру sDATA вашей функции анализатора, вы создаете ее экземпляр и используете при вызове cXParser::Parse, как показано тут:

```
sDATA Data;
cXParser Parser;
Parser.Parse("test.x", (void**)&Data);
```

После чего при каждом вызове ParseObject вы можете преобразовать соответствующий указатель для получения доступа к вашей структуре данных.

```
BOOL cXParser::ParseObject(IDirectXFileData *pDataObj, \
               IDirectXFileData *pParentDataObj, \
               DWORD Depth, \
               void **Data, BOOL Reference)
{
  cDATA *DataPtr = (sDATA*)*Data;
  DataPtr->NumObjects++; // Увеличить количество объектов
  return ParseChildObjects(pDataObj,Depth,Data,Reference);
}
```

Замечание. *Глубина объекта данных очень полезна для сортировки иерархий, таких как иерархии фреймов, используемых в скелетной анимации.*

Я знаю, что я опять забегаю вперед, показывая вам некоторые примеры кода для cXParser, так что давайте вернемся к пятому (и последнему) параметру ParseObject- Reference. Логическая переменная Reference определяет, является ли перечисляемый объект ссылочным или экземпляром. Вы можете использовать переменную Reference для определения, хотите ли вы загрузить данные ссылочного ,объекта или подождать пока создастся экземпляр объекта. Этот параметр полезен при загрузке данных анимации, которым необходимы ссылки на объекты, а не сами экземпляры объектов.

Гмм! После определения функции ParseObject остается определить последнюю функцию ParseChildObjects. К счастью, функция ParseChildObjects очень проста, она просто перечисляет все дочерние объекты передаваемого объекта. Обычно вы вызываете ParseChildObjects в конце вашей функции ParseObject, как я делал в последнем кусочке кода.

Вы можете заметить, что функции ParseChildObjects необходимо передавать текущий объект IDirectXFileData, глубину объекта данных, указатель на данные и флаг ссылки, потому что она ответственна за увеличение глубины и установки соответствующего родительского объекта для следующего вызова ParseObject. Если вы не хотите перечислять дочерние объекты, вы можете пропустить вызов ParseChildObjects и вернуть значение TRUE или FALSE. (TRUE продолжает перечисление, а FALSE останавливает его). Вы увидите примеры использования функций ParseObject и ParseChildObjects далее в этой книге.

После того как мы определились с основами, необходимо немного расширить класс анализатора. Как насчет добавления функций, позволяющих получать имя объекта, GUID и указатель на данные и несколько функций, вызываемых до и после анализирования .X файлов? Посмотрите на нижеследующий код, чтобы увидеть, как должен выглядеть новый класс анализатора.

```
class cXParser
{
  protected:
  // Функции, вызывемые при начале и окончании анализирования
  virtual BOOL BeginParse(void **Data)  { return TRUE; }
  virtual BOOL EndParse(void **Data)  { return TRUE; }

  // Функция, вызываемая при нахождении объекта
  virtual BOOL ParseObject( \
               IDirectXFileData *pDataObj, \
               IDirectXFileData *pParentDataObj, \
```

```
                DWORD Depth, \
                void **Data, BOOL Reference)
        {
         return ParseChildObjects(pDataObj, Depth, \
                                  Data, Reference);
        }

  // Функция, вызываемая для перечисления дочерних объектов
  BOOL ParseChildObjects(IDirectXFileData *pDataObj, \
                DWORD Depth, void **Data, \
                BOOL ForceReference = FALSE);

public:
  // Функция начала просмотра .X файла
  BOOL Parse(char *Filename, void **Data = NULL);

  // Функции, помогающие получить информацию об объекте
  const GUID *GetObjectGUID(IDirectXFileData *pDataObj);
  char *GetObjectName(IDirectXFileData *pDataObj);
  void *GetObjectData(IDirectXFileData *pDataObj,DWORD *Size);
};
```

Вы можете увидеть добавление функций BeginParse, EndParse, GetObjectGUID, GetObjectName и GetObjectData в cXParser. Вы уже видели код трех функций Get, неизвестными являются только виртуальные функции BeginParse и EndParse.

В их текущем виде, обе функции BeginParse и EndParse возвращают значение TRUE, что означает успешное их выполнение. Ваша задача перегрузить эти две функции в унаследованном классе, чтобы вы могли выполнять любые действия перед и после анализирования файла. Например, вы можете захотеть инициализировть какие-нибудь данные или создать указатель на данные в функции BeginParse, после чего удалить все используемые ресурсы в EndParse.

Обе функции BeginParse и EndParse вызываются непосредственно из функции Parse - вам просто необходимо перегрузить их и написать код для них. Вы увидите, как использовать эти функции далее в книге и в предстоящем разделе этой главы "Загрузка иерархии фреймов из .X".

Что же касается трех Get функций, вы используете их, указывая правильный объект IDirectXFileData; в ответ вы получите имя в созданном буфере данных, указатель на GUID шаблона или на буфер данных объекта и размер его данных. Например, следующий код вызывает эти три функции для получения доступа к данным объекта:

```
// pData = предварительно загруженный объект
char *Name = pParser->GetObjectName(pData);
const GUID *Type = pParser->GetObjectGUID(pData);
DWORD Size;
char *Ptr = (char*)pParser->GetObjectData(pData, &Size);
```

```
// сделать что-нибудь с данными и освободить ресурсы по завершении
delete [] Name;
```

Как я уже говорил, использовать новый класс cXParser по-настоящему просто. Далее, до конца этой главы и книги, вы будете использовать класс cXParser для анализирования .X файлов. Продвинутые читатели могут захотеть изменить класс анализатора для собственных нужд. Лично меня класс устраивает в его нынешнем виде.

Для быстрой проверки нового класса cXParser давайте посмотрим, как его наследовать и использовать. Предположим, вы хотите проанализировать .X файл на наличие объектов Mesh и отобразить их имена в окне сообщений. Вот код анализатора, который делает именно это:

```
class cParser : public cXParser
{
  public:
    cParser() { Parse("test.x"); }
    BOOL ParseObject(IDirectXFileData *pDataObj, \
              IDirectXFileData *pParentDataObj, \
              DWORD Depth, \
              void **Data, BOOL Reference)
    {
      if(*GetObjectGUID(pDataObj) == TID_D3DRMMesh) {
       char *Name = GetObjectName(pDataObj);
       MessageBox(NULL, Name, "Mesh template", MB_OK);
       delete [] Name;
      }
      return ParseChildObjects(pDataObj,Depth,Data,Reference);
   }
};
cParser Parser; // Создание экземпляра приведет к запуску анализатора
```

Только не говорите что, это было сложно! Хватит основ, давайте перейдем к использованию .X файлов для чего-нибудь полезного, как например для данных трехмерных мешей.

Загрузка мешей из .X

После того как вы твердо усвоили принцип работы формата файла .X, рассмотрим то, для чего Microsoft изначально их создавала - для хранения информации трехмерных мешей, используемых в ваших играх. Библиотека D3DX содержит функции, которые вы можете использовать для загрузки данных мешей из .X файлов. Добавив анализатор .X файлов, разработанный в этой главе, у вас появляется еще больше возможностей. Посмотрите, как просто работать с D3DX для загрузки данных мешей.

Загрузка мешей с использованием D3DX

Библиотека D3DX определяет удобный объект ID3DXMesh, который хранит и визуализирует трехмерные меши. Кроме этого вы можете использовать собственные специализированные контейнеры для хранения мешей, которые, я думаю, разумно использовать с ID3DXMesh. Этот объект я буду использовать до окончания этой главы (за исключением использования удобного объекта ID3DXSkinMesh, о котором я расскажу позже).

Самым быстрым способом загрузки данных меша при помощи D3DX является использование функций D3DXLoadMeshFromX и D3DXLoadMeshFromXof. Оба этих меша берут данные, хранимые в .X файле, и преобразуют их в объект ID3DXMesh. D3DXLoadMeshFromX сразу загружает весь файл .X (объединяя все меши в один), в то время как D3DXLoadMeshFromXof загружает один меш, указанный объектом IDirectXFileData.

Функция D3DXLoadMeshFromX имеет в качестве параметров имя загружаемого файла, некоторые флаги для контролирования загрузки, указатель на 3D устройство, указатели на буферы, содержащие данные материалов и различные указатели на данные, которые вы можете пока проигнорировать. Посмотрите на прототип функции D3DXLoadMeshFromX.

```
HRESULT D3DXLoadMeshFromX(
  LPSTR pFilename, // Имя загружаемого .X файла
  DWORD Options, // Опции загрузки
  LPDIRECT3DDEVICE9 pDevice, // указатель на трехмерное устройство
  LPD3DXBUFFER* ppAdjacency, // Установить в NULL
  LPD3DXBUFFER* ppMaterials, // Буфер для материалов
  LPD3DXBUFFER* ppEffectInstances, // Здесь не используется - NULL
  PDWORD pNumMaterials, // # загруженных материалов
  LPD3DXMESH* ppMesh); // Указатель на интерфейс меша
```

Приведенные комментарии являются самоописательными, так что сразу перейдем к рассмотрению использования функции D3DXLoadMeshFromX. Для начала вам необходимо создать экземпляр объекта ID3DXMesh.

```
ID3DXMesh *Mesh = NULL;
```

Далее предположим, вы хотите загрузить файл .X, называемый "test.x". Все просто - вам необходимо указать имя файла при вызове функции D3DXLoadMeshFromX ... но подождите! А как же параметр Options? Вы используете его, чтобы сказать D3DX как загружать меш - в системную или видео память, использовать память только для записи и так далее. Для каждой опции есть флаг. В таблице 3.3 приведен список основных макросов.

Таблица 3.3. Флаги D3DXLoadMeshFromX

Макрос	Описание
D3DXMESH_32BIT	Использовать 32-битные индексы (не всегда поддерживается)
D3DXMESH_USEHWONLY	Использовать только аппаратную обработку. Использовать только те устройства, которые точно поддерживают аппаратное ускорение.
D3DXMESH_SYSTEMMEM	Хранить меш в системной памяти
D3DXMESH_WRITEONLY	Установить данные меша только для записи, позволяя Direct3D находить лучшие расположения для хранения данных меша (обычно в видео памяти)
D3DXMESH_DYNAMIC	Использовать динамические буферы (для мешей, изменяющихся со временем)
D3DXMESH_SOFTWAREPROCESSING	Использовать программную обработку вершин, используемую вместо движка аппаратного преобразования и освещения

Из таблицы 3.3 видно, что опций загрузки мешей на самом деле не так уж и много. Вообще-то, я бы рекомендовал вам использовать только D3DXMESH_SYSTEMMEM или D3DXMESH_WRITEONLY. Первая опция D3DXMESH_SYSTEMMEM вынуждает храниться данным вашего меша в системной памяти, что ускоряет доступ к его данным для записи и чтения.

Задание флага D3DXMESH_DYNAMIC означает, что вы собираетесь периодически изменять данные меша. Лучше всего устанавливать этот флаг, если вы собираетесь периодически изменять данные меша (вершины) во время выполнения.

Если вам необходима скорость, то я предлагаю использовать флаг 3DXMESH_WRITEONLY, который говорит D3DX использовать память, из которой нельзя читать. В большинстве случаев это означает использование видеопамяти, потому что она обычно (но не всегда) только для записи. Если вы не будете читать данные вершин меша, тогда используйте этот флаг.

Совет. *Если вы не используете системную память или память только для записи, что остается использовать? Просто задайте в качестве параметра Options 0 при вызове D3DXLoadMeshFromX, и все будет нормально.*

Возвращаясь к параметрам D3DXLoadMeshFromX, вы найдете указатель на интерфейс 3D устройства. Никаких проблем - хотя бы одно должно быть у вас в проекте! Следующий параметр - указатель на ID3DXBUFFER, ppAdjacency. Установите его в NULL, мы не будем его тут использовать.

Следующие три параметра ppMaterials, ppEffectInstance и pNumMaterials содержат информацию о материале, такую как значения цветов, название текстуры и данные эффектов. Если вы используете DirectX 8, вы можете безопасно удалить ссылку ppEffectInstance - она не существует в той версии. Если вы используте DirectX 9, вы можете установить ppEffectInstance в NULL, потому что вам не нужна никакая информация об эффектах.

Указатель ppMaterials указывает на интерфейс ID3DXBuffer, который является простым контейнером данных. pNumMaterials - это указатель на переменную DWORD, которая будет содержать количество материалов в загруженном меше. Вы узнаете, как использовать информацию о материалах немного позже.

Завершает список параметров D3DXLoadMeshFromX указатель на объект ID3DXMesh - ppMesh. Этот интерфейс вы предоставляете для хранения данных загруженного меша. Вот и все! Теперь объединим все это в работающий пример загрузки меша.

Загрузите меш, названный "test.x", используя память для записи. После создания указателя объекта меша вам необходимо создать экземпляр объекта ID3DXBuffer для хранения данных материала и переменную DWORD для хранения количества материалов.

```
ID3DXBuffer *pMaterials = NULL;
DWORD NumMaterials;
```

После этого вызываем D3DXLoadMeshFromX.

```
// pDevice = указатель на правильный объект IDirect3DDevice9
D3DXLoadMeshFromX("test.x", D3DXMESH_WRITEONLY, pDevice, \
      NULL, &pMaterials, NULL, &NumMaterials, &Mesh);
```

Замечательно! Если все прошло, как и было задумано, D3DXLoadMeshFromX вернет код успешного завершения, и ваш меш загрузится в интерфейс ID3DXMesh! Конечно, все меши, содержащиеся в файле, были объединены в один меш, но как насчет тех случаев, когда необходимо получить доступ к каждому отдельно определенному мешу в файле?

Вот здесь то и появляется D3DXLoadMeshFromXof. Вы можете использовать D3DXLoadMeshFromXof в сочетании с вашим анализатором .X для загрузки данных меша из перечисленного объекта Mesh. Посмотрите на прототип функции D3DXLoadMeshFromXof.

```
HRESULT D3DXLoadMeshFromXof(
  LPDIRECTXFILEDATA pXofObjMesh,
  DWORD Options,
```

```
    LPD3DXBUFFER* ppMaterials,
    LPD3DXBUFFER* ppEffectInstances,
    PDWORD pNumMaterials,
    LPD3DXMESH* ppMesh);
```

Подождите-ка! D3DXLoadMeshFromXof выглядит практически так же как D3DXLoadMeshFromX! Единственным отличием является первый параметр, вместо указания имени загружаемого .X файла, D3DXLoadMeshFromXof использует указатель на объект IDirectXFileData. Задав указатель на текущий перечисляемый объект IDirectXFileData, D3DX загрузит все необходимые данные меша! И так как оставшиеся параметры совпадают с D3DXLoadMeshFromX, у вас не возникнет трудностей с использованием D3DXLoadMeshFromXof в вашем классе анализатора .X!

Остановимся на этом, потому что вы увидите, как использовать D3DXLoadMeshFromXof в вашем классе анализатора далее в этой главе, в разделе "Загрузка мешей, используя анализатор .X ".

Независимо от используемой функции для загрузки данных меша (D3DXLoadMeshFromX или D3DXLoadMeshFromXof) все, что вам остается сделать после загрузки меша в объект ID3DXMesh - это обработать информацию о материалах.

Для обработки информации о материалах вам необходимо получить указатель на буфер данных ID3DXBuffer (используемый при вызове D3DXLoadMeshFromX или D3DXLoadMeshFromXof), и преобразовать его к типу D3DXMATERIAL. После этого обработать все материалы, используя количество материалов сохраненных в NumMaterials. После этого вам необходимо создать массив структур D3DMATERIAL9 и интерфейсов IDirect3DTexture9 для хранения данных материалов меша. Используйте следующий код для обработки информации материалов:

```
// Объекты для хранения данных материалов и текстур
D3DMATERIAL9 *Materials = NULL;
IDirect3DTexture9 *Textures = NULL;

// Получить указатель на данные материалов
D3DXMATERIAL *pMat;
pMat = (D3DXMATERIAL*)pMaterials->GetBufferPointer() ;

// Выделить пространство для хранения материала
if(NumMaterials) {
//Loading Meshes from .X
Materials = new D3DMATERIAL9[NumMaterials];
Textures = new IDirect3DTexture9*[NumMaterials];

// Просмотреть все загруженные материалы
for(DWORD i=0;i<NumMaterials;i++) {
  // Скопировать информацию о материале
```

```
    Materials[i] = pMat[i].MatD3D;
    // Копировать рассеянный цвет в окружающий
    Materials[i].Ambient = Materials[i].Diffuse;

    // Загрузить текстуру, если задана
    Textures[i] = NULL;
    if(pMat[i].pTextureFilename) {
     D3DXCreateTextureFromFile(pDevice, \
            pMat[i].pTextureFilename, &Textures[i]);
    }
   }
  } else {
    // Создать материал по умолчанию, если не загрузился
    Materials = new D3DMATERIAL9[1];
    Textures = new IDirect3DTexture9*[1];

    // Установить по умолчанию белый материал без текстуры
    Textureы[0] = NULL;
    ZeroMemory(&Materials[0], sizeof(D3DMATERIAL9));
    Materials[0].Diffuse.r = Materials[0].Ambient.r = 1.0f;
    Materials[0].Diffuse.g = Materials[0].Ambient.g = 1.0f;
   Materials[0].Diffuse.b = Materials[0].Ambient.b = 1.0f;
    Materials[0].Diffuse.a = Materials[0].Ambient.a = 1.0f;
  }
  // Освободить буфер данных материала
  pMaterials->Release();
```

Вы можете видеть в предыдущем коде, что я добавил проверку случая, когда материал не был загружен, а создается материал, используемый по умолчанию. После загрузки материалов вы можете использовать интерфейс меша для визуализации. В главе 1 было показано, как использовать интерфейс меша для его визуализации. А пока вернемся к теме, о которой я говорил ранее - загрузка мешей, используя анализатор .X.

Загрузка мешей, используя анализатор .X

Как я и обещал, пришло время узнать, как присоединить функции загрузки мешей к классу анализатора .X. Т. к. мы будем непосредственно получать доступ к объектам мешей, необходимо использовать функцию D3DXLoadMeshFromXof для загрузки данных меша. Это означает, что вам необходимо анализировать каждый объект, ища при этом объекты "Mesh". Начнем с наследования класса анализатора, с которым будем работать.

```
class cXMeshParser : public cXParser
{
  public:
   ID3DXMesh *m_Mesh;
```

```
  public:
   cXMeshParser() { m_Mesh = NULL; }
   ~cXMeshParser() { if(m_Mesh) m_Mesh->Release(); }

   BOOL ParseObject(IDirectXFileData *pDataObj, \
            IDirectXFileData *pParentDataObj, \
            DWORD Depth, \
            void **Data, BOOL Reference);
};
```

Как вы можете видеть из объявления класса, я объявляю только один объект меша. Если вы хотите больше, вам необходимо создать связанный список (или какой-либо другой список) для хранения объектов меша. Я оставляю это вам, потому что мне просто необходимо продемонстрировать использование функции D3DXLoadMeshFromXof.

Перезагружая функцию ParseObject, позволим ей искать объекты Mesh.

```
BOOL cXMeshParser::ParseObject( \
           IDirectXFileData *pDataObj, \
           IDirectXFileData *pParentDataObj, \
           DWORD Depth, \
           void **Data, BOOL Reference)
{
  // Пропустить ссылочные объекты
  if(Reference == TRUE)
   return TRUE;

  // Убедиться, что анализируемый объект - Mesh
  if(*GetObjectGUID(pDataObj) == D3DRM_TIDMesh) {
   // это меш, использовать D3DXLoadMeshFromXof для его загрузки
   ID3DXBuffer *Materials;
   DWORD NumMaterials;
   D3DXLoadMeshFromXof(pDataObj, 0, pDevice, NULL, \
           &Materials, NULL, &NumMaterials, &m_Mesh);

   // Закончить обработку информации о материалах

   // Вернуть FALSE для завершения анализирования
   return FALSE;
    }
  // Анализировать дочерние объекты
  return ParseChildObjects(pDataObj,Depth,Data,Reference);
}
```

Вот и все! Используя один быстрый вызов, вы загружаете меш из объекта Mesh! Если вы думаете, что это круто, у меня есть для вас кое-что новое - скелетные меши. Правильно, те изящные скелетные меши, о которых вы читали в главе 2, также легко использовать, как и стандартные меши. Читайте далее, чтобы узнать, как загружать скелетные меши из .X файлов.

Загрузка скелетных мешей

Как я упоминал в главе 1, скелетный меш содержит иерархию костей (скелетную структуру), которую вы можете использовать для деформации меша, к которому присоединены кости. Хотя в главе 1 и описывалось использование скелетных мешей, я оставил вопрос загрузки данных скелетных мешей для этой главы, потому что вы можете загружать их данные, только используя класс анализатора .X. К счастью для вас, вы подготовлены!

Загрузка скелетных мешей из .X файлов похожа на загрузку обычных мешей. Перечисляя объекты данных, вы можете определить, из каких из них загружать данные скелетного меша и помещать их в объект ID3DXSkinMesh.

Сюрпризом является то, что данные скелетного меша хранятся в том же объекте, где и обычные меши, - Mesh. Если объект один и тот же, то как же мы можем отличать скелетные меши от обычных?

Единственным способом, чтобы определить содержит ли объект Mesh данные скелетного меша, является использовании функции D3DXLoadSkinMeshFromXof для загрузки меша в объект ID3DXSkinMesh. После загрузки данных меша вы можете получить только что загруженный объект скелетного меша для просмотра, есть ли в нем информация о костях (которая содержится в специально встраиваемом Mesh-объекте). Если информация о костях существует, то это скелетный меш. Если костей не существует, то меш является обычным и может быть преобразован к объекту ID3DXMesh.

Я начинаю забегать вперед, так что вернемся к ID3DXSkinMesh и D3DXLoadSkinMeshFromXof. Совсем как с обычными мешами, необходимо создать экземпляр объекта ID3DXSkinMesh.

```
ID3DXSkinMesh *SkinMesh = NULL;
```

В функции ParseObject необходимо заменить вызов D3DXLoadSkinMeshFromXofl. Вместо вызова D3DXLoadMeshFromXof будем использовать D3DXLoadSkinMeshFromXof.

Замечание. *Пользователи DirectX 8 могут вырезать строку LPD3DXBUFFER* ppEffectInstances из вызова D3DXLoadSkinMeshFromXof.*

```
HRESULT D3DXLoadSkinMeshFromXof(
  LPDIRECTXFILEDATA pXofObjMesh,
  DWORD Options,
  LPDIRECT3DDEVICE9 pDevice,
  LPD3DXBUFFER* ppAdjacency,
  LPD3DXBUFFER* ppMaterials,
  LPD3DXBUFFER* ppEffectInstances,
  DWORD* pMatOut,
  LPD3DXSKINMESH* ppMesh);
```

Я знаю, вы скажите, что D3DXLoadSkinMeshFromXof выглядит почти как D3DXLoadMeshFromXof, и вы будете правы! Загрузка скелетного меша, используя функцию D3DXLoadMeshFromXof, выглядит так:

```
D3DXLoadSkinMeshFromXof(pDataObj, 0, pDevice, NULL, \
        &Materials, NULL, &NumMaterials, &SkinMesh);
```

После того как вы вызвали D3DXLoadSkinMeshFromXof, используя правильный объект Mesh, в вашем распоряжении появляется новый объект ID3DXSkinMesh.

Загрузка иерархии фреймов из .X файла

Системы скелетной анимации требуют иерархии фреймов (которые представлены структурой костей) для ориентации каждой кости при визуализации. Формат файла .X определяет шаблон данных ссылочных фреймов, который вы можете использовать для задания иерархии костей. Этот шаблон, Frame, просто ответственный за типы. Он позволяет встраивать объекты любого типа, так что вы можете ссылаться на ссылочный объект Frame, присваивая ему имя и позволяя адресовать все содержащиеся объекты.

Построение иерархии фреймов включает в себя анализ .X файла, в котором вы связываете каждый объект Frame между собой. Взаимоотношения очень важны - объект Frame может быть как родственником, так и потомком другого объекта Frame и может иметь неограниченное количество родственников и/или потомков, привязанных к нему.

В DirectX 9 вы можете получить доступ к специальной структуре DirectX, называемой D3DXFRAME, предназначенной для хранения каждого кадра в иерархии. Я говорил об этой структуре в главе 1. В этом разделе я использовал D3DXFRAME для хранения иерархии фреймов.

Каким образом вы будете анализировать и создавать объекты Frame, зависит от вас. Вы можете использовать библиотеку D3DX или анализировать/строить иерархию сами, используя специализированные анализаторы .X. Что лучше для вас? Принимая во внимание, что замечательно использовать библиотеку D3DX, я обнаружил, что методы использования библиотеки при загрузке иерархии фреймов сложны, необходимо создавать не менее двух используемых интерфейсов и самостоятельно написанный класс операций с памятью. Зачем волноваться, когда вы можете загружать иерархию, используя один маленький созданный вами класс анализирования .X?

Вместо этого возьмите проверенный анализатор .X (разработанный ранее в этой главе в разделе "Создание класса анализатора .X") и унаследуйте от него версию, которая ищет объекты Frame. Я начну, показав наследуемый класс, который можно использовать.

```
class cXFrameParser : public cXParser
{
  public:
  // объявить расширенную версию D3DXFRAME
  // которая содержит конструктор и деструктор
  struct D3DXFRAME_EX : public D3DXFRAME {
   D3DXFRAME_EX()
   {
    Name = NULL;
    pFrameSibling = NULL; pFrameFirstChild = NULL;
   }
   ~D3DXFRAME_EX()
   {
    delete [] Name;
    delete pFrameFirstChild;
    delete pFrameSibling;
   }
  } D3DXFRAME_EX;
  // создать корневой фрейм иерархии
  D3DXFRAME_EX *m_RootFrame;

 public:
   cXFrameParser() { m_RootFrame = NULL; }
   ~cXFrameParser() { delete m_RootFrame; }

   BOOL BeginParse(void **Data)
   {
    // очистить иерархию
    delete m_RootFrame; m_RootFrame = NULL;
   }

   BOOL ParseObject(IDirectXFileData *pDataObj, \
        IDirectXFileData *pParentDataObj, \
        DWORD Depth, \
        void **Data, BOOL Reference)
   {
    // пропустить ссылочные фреймы
    if(Reference == TRUE)
     return TRUE;

    // убедиться, что анализируемый шаблон - Frame
    if(*GetObjectGUID(pDataObj) == TID_D3DRMFrame) {

     // Создать структуру фрейма
     D3DXFRAME_EX *Frame = new D3DXFRAME_EX();
```

```
      // Получить имя фрейма (назначить новое, если не было найдено)
     if((Frame->Name = GetObjectName(pDataObj)) == NULL)
      Frame->Name = strdup("No Name") ;

      // связать структуру фрейма в список
      if(Data == NULL) {
       // связать как родственник корневого
       Frame->pFrameSibling = m_RootFrame;
       m_RootFrame = Frame;
       Data = (void**)&m_RootFrame;
      } else {
       // связать как потомка текущего фрейма
       D3DXFRAME_EX *FramePtr = (D3DXFAME_EX*)*Data;
       Frame->pFrameSibling = FramePtr->pFrameFirstChild;
       FramePtr->pFrameFirstChild = Frame;
       Data = (void**)&Frame;
      }
    }
    return ParseChildObjects(pDataObj,Depth,Data,Reference);

};
cXFrameParser Parser;
Parser.Parse("frames.x");
// Parser.m_RootFrame теперь указывает на корневой фрейм иерархии
```

Вот и все. После завершения функции cXFrameParser::Parse у вас будет хранящаяся самостоятельно иерархия фреймов, готовая к использованию в ваших проектах. Чтобы лучше понять, как использовать этот класс, посмотрите демонстрационную программу ParseFrame компакт-диска книги (смотрите конец главы для дополнительной информации). Демонстрационная программа ParseFrame загружает выбранный вами.X файл и отображает иерархию объектов в окне списка.

В оставшейся части книги вы увидите, как использовать иерархию фреймов для создания анимации.

Загрузка анимации из .X

Хотя такое базовое понятие как анимация заслуживает своего рассмотрения, я отложу обсуждение загрузки данных анимации до глав по работе с заранее вычисленной анимацией, потому что это больше соответствует порядку книги. Данные анимации могут быть в любом формате (как и все данные), но в оставшихся главах книги вы поймете, как создать свой набор шаблонов анимации для использования в проектах. Не волнуйтесь, все получиться, просто не сдавайтесь!

Загрузка специализированных данных из .X

Как я говорил повсюду в этой главе, формат файла .X является полностью открытым; не существует ограничений на хранимые типы данных. Имея это ввиду, вы можете создать хранилище любого типа данных, и получить доступ к этим данным будет не сложнее, чем к мешам или фреймам, рассмотренным выше.

Вернемся к шаблону ContactEntry и посмотрим, как анализировать .X файл и отобразить все экземпляры ContactEntry. Как и раньше небольшой, унаследованный от cXParser, нам отлично подойдет.

```
// Сначала объявим GUID ContactEntry
#include "initguid.h"
DEFINE_GUID(ContactEntry, \
       0x4c9d055b, 0xc64d, 0x4bfe, 0xa7, 0xd9, 0x98, \
       0x1f, 0x50, 0x7e, 0x45, 0xff);

// Now, define the .X parser class
class cXContactParser : public cXParser

  public:
   BOOL ParseObject(IDirectXFileData *pDataObj, \
             IDirectXFileData *pParentDataObj, \
             DWORD Depth, \
             void **Data, BOOL Reference)

     // пропустить ссылочные объекты
     if(Reference == TRUE)
      return TRUE;

     // убедиться, что анализируемый объект - ContactEntry
     if(*GetObjectGUID(pDataObj) == CONTACTENTRY) {
      // Получить указатель на данные и их размер
      DWORD DataSize;
      DWORD *DataPtr;
      DataPtr = (DWORD*)GetObjectData(pDataObj, &DataSize);
      // Получить имя из объекта данных
      char *Name = (char*)*DataPtr++;

      // Получить номер телефона из объекта данных
      char *PhoneNum = (char*)*DataPtr++;
      // Получить возраст из объекта данных
      DWORD Age = *DataPtr++;
      // Отобразить контактную информацию
      char Text[1024];
      sprintf(Text, "Name: %s\r\nPhone #: %s\r\nAge: %lu", \
         Name, PhoneNum, Age);
      MessageBox(NULL, Text, "Contact", MB_OK);

     return ParseChildObjects(pDataObj,Depth,Data,Reference);
   }
```

```
};
cXContactParser Parser;
Parser.Parse("contacts.x");
```

После единственного вызова cXContactParser::Parse вы увидите список имен людей, телефонных номеров и возрастов - все содержащееся в "contact.x" файле! Разве это не было просто? Видите, вы не должны боятся формата файла .X. Лично я использую этот формат для хранения всех данных, какие только могу.

Я предлагаю, чтобы вы перечитали эту главу несколько раз, прежде чем продолжать; заострите внимание на создании ваших собственных шаблонов и получения доступа к данным объекта. Всюду в оставшейся части книги вы будете опираться на .X формат для хранения специализированных данных, связанных с каждым разделом анимирования, так что я хочу, чтобы вы чувствовали себя уверенно используя .X

Посмотрите демонстрационные программы

В этой главе вы узнали, что такое .X, все - от создания ваших собственных шаблонов до использования этих шаблонов для создания объектов данных, хранящих важные данные вашей игры. Для помощи в демонстрировании полезности .X и для примера использования кода, приведенного в этой главе, я включил две демонстрационные программы (ParseFrame и ParseMesh) в компакт-диск этой книги. Посмотрите, что делает каждая демонстрационная программа.

ParseFrame

Первой демонстрационной программой для этой главы является ParseFrame, которая иллюстрирует использование информации при загрузке иерархии фреймов из .X файла. Как показано на рис. 3.2, ParseFrame имеет кнопку, названную "Select .X File" ("Выберите .X файл"). Нажмите эту кнопку, выберите загружаемый .X файл и нажмите "Open" ("Открыть").

После выбора .X файла для его загрузки нажмите "Open" для отображения иерархии фреймов, содержащихся в .X файле. На рис. 3.2 показана иерархия фреймов, содержащихся в файле "Tiny.x", расположенном в директории DirectX SDK \samples\media. Для облегчения, фреймы расположены в соответствии с их уровнем в иерархии.

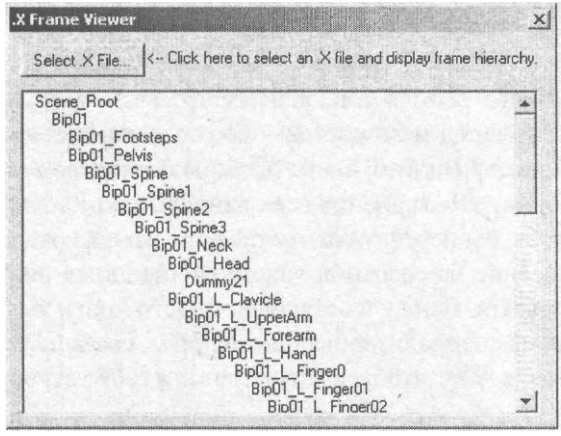


Рис. 3.2. Диалоговое окно демонстрационной программы ParseFrame содержит два элемента: кнопку для загрузки .X фала и окно списка для отображения иерархии фреймов

ParseMesh

Вторая демонстрационная программа, находящаяся в директории главы 3, - ParseMesh. Как и у демонстрационной программы ParseFrame, у нее есть кнопка (как показано на рис. 3.3), нажав на которую вы указываете и открываете .X файл.

В противоположность списку иерархии фреймов ParseFrame, демонстрационная программа ParseMesh перечисляет информацию о найденных мешах в открытых .X файлах. Эти данные включают тип меша (обычный или скелетный), количество вершин, количество граней и (если имеются) количество костей. Вы можете использовать эту программу на собственных файлах, для того чтобы убедиться, что все меши содержат правильную информацию.

Программы на компакт-диске

В директории главы 3 вы найдете два проекта, иллюстрирующие использование класса анализирования .X. Этими двумя проектами являются:

- **ParseFrame.** Эта демонстрационная программа использует класс cXParser для загрузки и создания иерархии фреймов из .X файла. Она расположена в \BookCode\Chap03\ParseFrame.

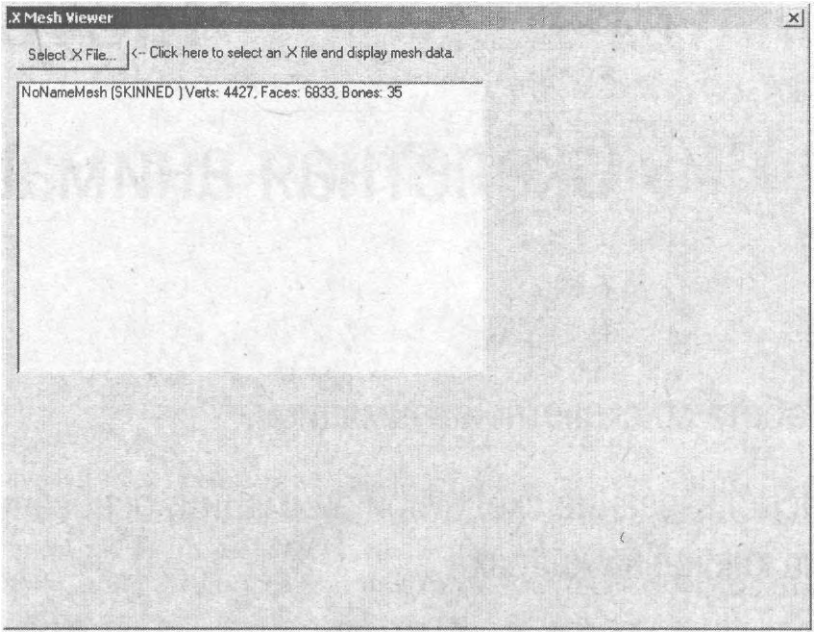


Рис. 3.3. После указания и открытия .X файла вам показываются данные каждого меша, содержащегося в этом файле

- **ParseMesh.** Вы можете использовать эту программу для загрузки и отображения информации о мешах (обычных и скелетных). Она расположена в \BookCode\Chap03\ParseMesh.

Часть III

Скелетная анимация

4. Работа со скелетной анимацией

5. Использование скелетной анимации, основанной на ключевых кадрах

6. Комбинирование скелетных анимаций

7. Создание кукольной анимацией

Глава 4

Работа со скелетной анимацией

Как только темнота окутает мой персонаж, я уже не смогу ему помочь, но я немного хихикнул.

"Эти дураки никогда этого не увидят",- сказал я себе. Как только я нажал джойстик, мой персонаж медленно пополз вперед. Когда он был вуглу, я нажал кнопку, и внезапно он оторвался от стены и сел, ожидая добычу. Сидя в молчаливом ожидании, мой персонаж неторопливо проверяет свои очки ночного видения, оружие и другие причудливые устройства, которые он принес с собой для рискованной попытки.

Вся эта анимация происходит в реальном времени, поэтому нет никаких переходов и остановок во время изменения действий персонажа. От бега до ползания или лазания по стенам и беспечной проверки оборудования - все анимации непосредственно перетекают друг в друга. В своем воображении я могу отлично представить, как скелет персонажа двигается и изгибается в результате всех его действий. Меш персонажа в точности соответствует костям, включая всякие мелочи - от неравномерности мускулов до складок на камуфляжном костюме.

Вы можете создать все эти особенности анимации (и даже намного больше), используя так называемую скелетную анимацию, наверное, самую привлекательную технологию анимирования, которую вы только можете использовать в ваших проектах. Работать с этой технологией определенно проще, чем это может показаться сначала. Посмотрев примеры таких игр как "Splinter Cell" фирмы Tom Clancy , показывающих миру возможности скелетной анимации, вы точно не захотите пропустить эту тему. Данная глава поможет вам начать. Здесь все: от начала работы со структурами скелетов до скелетных мешей.

Вы этой главе вы научитесь:

- Работать со скелетными иерархиями;
- Использовать скелетные меши;
- Загружать иерархии фреймов и скелетные меши из .X файлов;
- Перестраивать иерархии костей и скелетных мешей.

Начало скелетной анимации

Скелетная анимация - два слова, от которых на ум приходят второсортные фильмы ужасов, в которых из могил восстают мертвецы, преследовать живых. Для программистов же эти слова имеют совсем другое значение. Если вы - как я, эта тема даст вам такой прилив адреналина, какой никогда не сможет дать фильм ужасов.

Скелетная анимация быстро становится основной технологией в анимировании, используемой программистами, потому что она быстро обрабатывается и дает невероятные результаты. При использовании скелетной анимации вы можете анимировать любую деталь персонажа. Она позволяет вам контролировать любую часть тела персонажа: от морщин на его коже до неровностей мускулов. Вы можете использовать каждое сочленение, каждую кость, мускул для деформирования меша вашего персонажа.

Использование структур скелетов и иерархий костей

Представьте себе скелетную анимацию так: ваше тело (или по крайне мере кожа) - это меш, дополненный соответствующим набором костей. Когда ваши мускулы толкают, тянут и крутят кости, ваше тело соответствующим образом меняет форму. Вместо того чтобы думать о мускулах, меняющих форму вашего тела, будем думать о костях, меняющих положение каждой части тела.

Если вы поднимаете руку, вращается плечо, поворот которого заставляет вращаться всю руку и заставляет кожу менять форму. Ваше тело (меш) меняет форму для приспособления к изменениям положения костей. Скелетная анимация работает точно так же. Когда соответствующая скелетная структура меняет положение, вращаясь в сочленениях, накладной (overlaid) меш (правильнее называемый скелетный меш) изменяет форму для соответствия скелету.

Как вы можете видеть, есть два отдельных понятия при работе со скелетной анимацией - скелетная структура и скелетный меш. Давайте рассмотрим каждое понятие более подробно, чтобы увидеть, как они взаимодействуют, начав со скелетной структуры.

Использование скелетной структуры и скелетного меша

Скелетная структура, как вы можете представить, является набором соединенных костей, образующих иерархию (иерархию костей, если быть точным). Одна кость, называемая корневой, является центральной точкой для всей скелетной структуры. Все кости присоединяются к корневой кости либо как родственники, либо как потомки.

Слово "кость" означает-объект ссылку на фрейм (объект-фрейм, который представлен структурой DirectX D3DXFRAME или шаблоном Frame внутри файла .X). Если бы вы посмотрели структуру D3DXFRAME, вы бы действительно обнаружили связанный список указателей (D3DXFRAME::pFrameSibling и D3DXFRAME::pFrameFirstChild), образующий иерархию. Указатель D3DXFRAME::pFrameSibling связывает между собой кости одного уровня иерархии, в то время как указатель D3DXFRAME::pFrameFirstChild привязывает дочерние кости, которые расположены на один уровень ниже в иерархии.

Обычно для создания скелетных структур, применяемых в ваших проектах, используются пакеты трехмерного моделирования. Экспортирование иерархии костей из .X файла является отличным примером. Microsoft реализовала экспортеры для 3D Studio Max и Maya, которые позволяют экспортировать данные скелетов и анимации в .X файлы, причем такую возможность экспорта предусматривают многие программы моделирования. Я предполагаю, что у вас есть программа[1], с помощью которой можно экспортировать эти иерархии в .X файл.

Вы найдете множество вещей внутри .X файла, содержащих данные анимации. Первое (и наверное самое главное на данный момент), вы обнаружите иерархию шаблонов Frame, которая на самом деле является замаскированной иерархией костей. Если бы у вас была простая скелетная структура, подобная изображенной на рис. 4.1, тогда ваша иерархия фреймов выглядела бы подобно изображенной на том же рисунке.

Вы должны обнаружить стандартный объект данных Mesh, встроенный в иерархию объектов Frame. Объект Mesh содержит информацию об объекте скелетной анимации и костях, используемых в скелетной структуре. Так и есть - объекты Frame и Mesh содержат информацию о вашей скелетной структуре! Объект Mesh определяет, какие фреймы представляют кости, в то время как объект Frame определяет фактическую иерархию.

1. Можно взять из DirectX Extras с сайта Microsoft.— *Примеч. научн. ред.*

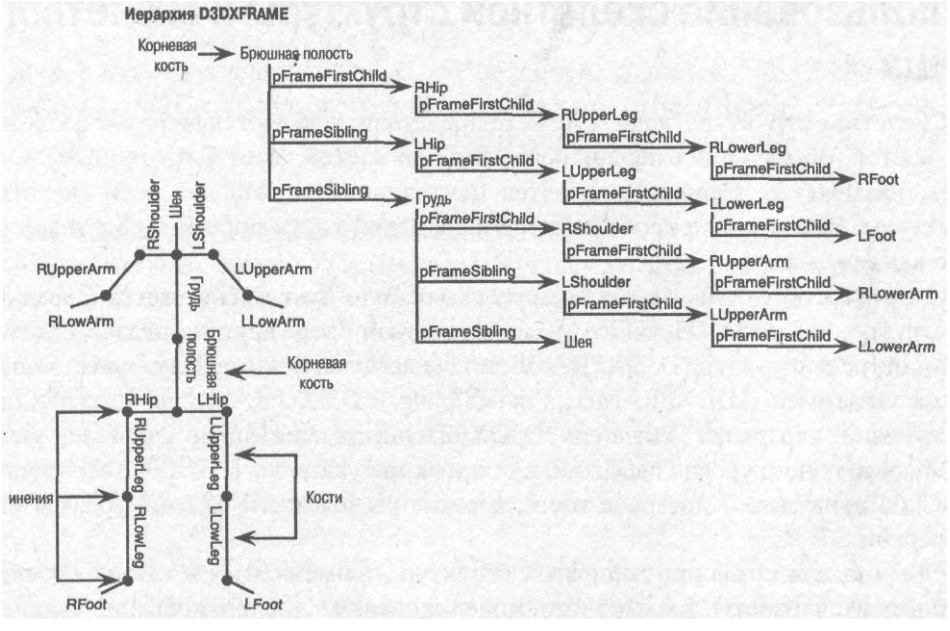


Рис. 4.1. Расположенная слева скелетная структура представлена иерархией справа. Заметьте, что используются указатели D3DXFRAME для образования связанного списка родственных и дочерних кадров

Однако на данный момент кости играют незначительную роль. Т. к. кости зависят от иерархии фреймов, на данном этапе важно сконцентрироваться исключительно на фреймах. Вам просто нужно загрузить иерархию (например, из .X файла) и настроить ее для дальнейшего использования. Читайте далее, чтобы узнать, как загружать иерархии из .X.

Загрузка иерархий из .X

Не следует бить мертвую лошадь (как я могу сделать такую ужасную вещь?), но я хочу слегка коснуться загрузки иерархий фреймов из .X файлов. Хотя в главе 3 детально описаны использование .X файлов и загрузка иерархий фреймов, я хочу еще раз вернуться к этому, используя специальные структуры для хранения иерархии.

Для хранения иерархии фреймов вам необходимо использовать структуру D3DXFRAME (или показанную в главе 1 D3DXFRAME_EX). Как я уже замечал ранее, структура D3DXFRAME (или унаследованная D3DXFRAME_EX) содержит два указа-

теля, предназначенных для создания иерархии - pFrameSibling и pFrameFirstChild. Ваша задача связать каждый загружаемый из .X файла фрейм, используя эти два указателя.

Переберите все объекты данных из заданного .X файла, начав с корневого объекта фрейма. Когда вы обнаружите объект Frame, привяжите его как родственный или дочерний к предыдущему кадру. Продолжайте выполнять эти операции, пока не загрузите все фреймы. В этом примере используйте структуру D3DXFRAME_EX для хранения фреймов в иерархии.

Т. к. в главе 3 содержалось намного больше информации об анализе .X файлов, я просто слегка напомню вам. Обычно вы открываете .X файл и перебираете каждый объект в нем. Для каждого найденного объекта Frame создаете соответствующий объект D3DXFRAME (или D3DXFRAME_EX) и привязываете его в иерархию объектов.

Для обработки .X файлов, вы можете создать класс, который бы выполнял всю работу за вас. Вы можете просто создать функцию ParseObject, принадлежащую классу, которая бы давала доступ к данным всех объектов. Опять же в главе 3 приведено подробное объяснение использования этого класса.

А теперь давайте посмотрим на функцию ParseObject, которая вызывается для всех перечисляемых объектов.

```
BOOL cXFrameParser::ParseObject(
            IDirectXFileData *pDataObj,
            IDirectXFileData *pParentDataObj,
            DWORD Depth,
            void **Data, BOOL Reference)
{
  const GUID *Type = GetObjectGUID(pDataObj);

  // Если объект типа Frame (не ссылочный)
  // тогда добавить его в иерархию
  if(*Type == TID_D3DRMFrame && Reference == FALSE) {

   // создать контейнер фрейма
   D3DXFRAME_EX *pFrame = new D3DXFRAME_EX();

   // Если есть, получить имя фрейма
   pFrame->Name = GetObjectName(pDataObj);

   // Привязать объект в иерархию
   if(Data == NULL) {
    // Привязать как родственника корневого
    pFrame->pFrameSibling = m_RootFrame;
    m_RootFrame = pFrame; pFrame = NULL;
    Data = (void**)&m_RootFrame;
   } else {
    // Привязать как дочерний текущего
    D3DXFRAME_EX *pFramePtr = (D3DXFRAME_EX*)*Data;
    pFrame->pFrameSibling = pFramePtr->pFrameFirstChild;
```

```
      pFramePtr->pFrameFirstChild = pFrame; pFrame = NULL;
      Data =  (void**)&pFramePtr->pFrameFirstChild;
     }
    }

    // Загрузить матрицу преобразования фрейма
    if(*Type==TID_D3DRMFrameTransformMatrix && Reference==FALSE) {
      D3DXFRAME_EX *Frame = (D3DXFRAME_EX*)*Data;
      if(Frame) {
      Frame->TransformationMatrix = *(D3DXMATRIX*) \
             GetObjectData(pDataObj,  NULL);
      Frame->matOriginal = Frame->TransformationMatrix;
     }
    }

    // Анализировать дочерние объекты
    return ParseChildObjects(pDataObj, Depth, Data, Reference);
  }
```

Если вы еще не читали главу 3 (стыдно, если нет!), некоторые части приведенного выше кода могут немного смущать вас. Обычно функция ParseObject вызывается для каждого перечисляемого объекта. Внутри функции ParseObject вы проверяете тип текущего перечисляемого объекта (используя его шаблонный GUID). Если тип является Frame, вы создаете структуру фрейма и загружаете его имя в нее.

Далее вы привязываете фрейм в иерархию фреймов, что и выглядит немного странным. Класс cXFrameParser содержит два указателя - один для созданного корневого фрейм-объекта (m_RootFrame, член класса) и один для объекта данных (Data), который передается в качестве параметра функции ParseObject при каждом ее вызове. Указатель данных содержит данные последнего загруженного фрейм-объекта.

Как только вы начинаете анализировать .X файл, указатель Data установлен в NULL, означая, что не было загружено никаких фрейм-объектов. Когда вы загружаете фрейм-объект, вы проверяете указатель данных, чтобы определить указывает ли он на фрейм. Если нет, то предполагается, что текущий фрейм является родственником корневого фрейма. Если же указатель данных указывает на фрейм, то полагается что текущий перечисляемый фрейм является потомком указываемого указателем данных фрейма.

Знание того, является ли текущий фрейм родственным или дочерним, считается определяющим при создании иерархии. Родственные фреймы связываются, используя указатель pFrameSibling структуры D3DXFRAME, в то время как дочерние фреймы связываются, используя pFrameFirstChild. Сразу же после загрузки фрейма указатель данных устанавливается в новый фрейм или в предыдущий родственный фрейм. В конце концов, все фреймы связываются между собой как родственники или потомки.

Вы заметите также, что функция ParseObject содержит код для загрузки матрицы преобразования (трансформации) фрейма (представленной шаблоном FrameTransformMatrix). Обычно объект FrameTransformMatrix встраивается в объект Frame. Объект FrameTransformMatrix определяет начальное положение загруженного фрейма.

Примененная к скелетной анимации, эта матрица преобразования фрейма определяет начальную позу вашей скелетной структуры. Например, стандартная скелетная структура может находится в позе, в которой тело расположено вертикально, а руки расставлены. Однако предположим, что все ваши анимации рассчитаны на персонаж, находящийся в другой позе, возможно с руками по швам и чуть согнутыми ногами. Вместо того чтобы менять положение всех вершин или костей, чтобы они соответствовали позе, перед сохранением .X файла в вашей программе трехмерного моделирования вы можете изменить преобразования фреймов. Таким образом, все движения костей будут происходить относительно этой позы. Это становится более понятным после того, как вы попробуете управлять положением костей при анимации, так что я пока пропущу эту тему. Просто запомните, что внутри каждой загружаемой структуры фрейма есть место для хранения начальной матрицы преобразования (в объекте D3DXFRAME::TransformationMatrix).

После выполнения всего сказанного иерархия фреймов будет загружена. Корневой фрейм хранится в mRootFrame связанного списка объектов D3DXFRAMEEX в классе загрузки фреймов. Вашей задачей является использование этого указателя в ваших программах. После этого вы можете начать разбираться с положением костей.

Изменение положения костей

После того как вы загрузили иерархию костей, вы можете управлять ими. Для изменения положения кости вам необходимо обнаружить соответствующую ей структуру фрейма, создав функцию, которая бы рекурсивно просматривала фрейм на совпадение с именем кости. Как только он будет найден, у вас будет указатель на него, который вы можете использовать для получения доступа к матрице преобразования. Функция рекурсивного поиска может иметь такой вид:

```
D3DXFRAME_EX *FindFrame(D3DXFRAME_EX *Frame, char *Name)
{
  // Искать только не пустые имена
  if(Frame && Frame->Name && Name) {
   // Вернуть указатель на фрейм, если имя совпадает
   if(!strcmp(Frame->Name, Name))
    return Frame;
  }
```

```
  // попробовать найти заданное имя в родственных фреймах
  if(Frame && Frame->pFrameSibling) {
   D3DXFRAME_EX *FramePtr = \
    FindFrame((D3DXFRAME_EX*)Frame->pFrameSibling, \
          Name);
   if(FramePtr)
    return FramePtr;
  }

  // попытаться найти имя в дочерних фреймах
  if(Frame && Frame->pFrameFirstChild) {
   D3DXFRAME_EX *FramePtr = \
    FindFrame((D3DXFRAME_EX*)Frame->pFrameFirstChild, \
         Name);

   if(FramePtr)
    return FramePtr;
  }

  // ничего не найдено, возвращаем NULL
  return NULL;
}
```

Замечание. *Вы можете применять любые преобразования к кости в иерархии, но рекомендуется использовать только повороты. Почему только повороты? Подумайте об этом так: когда вы сгибаете локоть, он вращается. А что вы скажете, если вместо этого переместить локоть? Получится, что локоть отделится от вашей руки, чего бы вам определенно не хотелось!*

Если вы хотите переместить весь меш, просто перемещайте корневую кость; это преобразование распространится на все остальные кости. А еще лучше использовать преобразование мира, чтобы двигать ваш меш-объект.

Предположим, вы хотите найти кость "Leg" используя функцию FindFrame. Вы просто указываете имя этой кости и указатель на корневой объект фрейм, как показано тут:

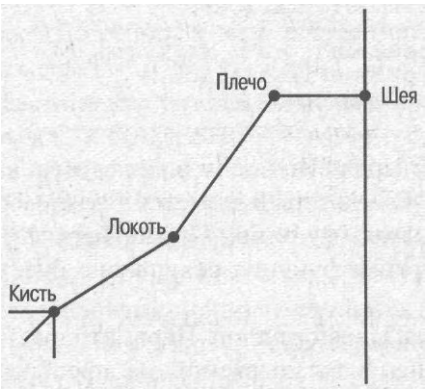
```
// pRootframe = указатель на корневой фрейм D3DXFRAME_EX
D3DXFRAME_EX *Frame = FindFrame(pRootFrame, "Leg");
if(Frame) {
  // Сделать что-нибудь с фреймом, как, например, заменить
  // одну матрицу преобразования     D3DXFRAME_EX::TransformationMa-
trix // на другую.
  // Здесь давайте немного повернем кость
  D3DXMatrixRotationY(&Frame->TransformationMatrix, 1.57f);
}
```

Обновление иерархии

После того как вы изменили матрицу преобразования костей, вам необходимо обновить всю иерархию, чтобы вы могли использовать ее для последующей визуализации. Даже если вы не изменяли матрицу преобразования костей, вам все равно необходимо обновить иерархию для установления некоторых переменных перед визуализацией.

Во время обновления иерархии вам необходимо комбинировать удачные трансформации вниз по иерархии. Начиная с корня, вы накладываете матрицу преобразования костей на комбинированную матрицу преобразования фреймов. Матрица преобразования костей накладывается также на все родственные фреймы корневого. После этого только что посчитанная матрица преобразования накладывается на все дочерние фреймы корневого. Этот процесс распространяется на всю иерархию.

Хотя это и трудно понять сразу, сделайте так: возьмите скелетную структуру, показанную на рис. 4.2, и, начиная с корня, умножайте ее на матрицу преобразования, которая размещает корень в мире.



Кость	Преобразование
Шея	Шеи
Плечо=	Плеча x Шеи
Локоть=	Локтя x Плеча x Шеи
Кисть=	Кисти x Локтя x Плеча x Шеи

Рис. 4.2. Простая скелетная структура слева использует преобразования костей справа для расположения фреймов

Как вы можете видеть на рис. 4.2, комбинированное преобразование корня применяется на все дочерние кости, которые в свою очередь тоже комбинированные. Результаты передаются дочерним костям тех костей. Однако расчет матриц преобразования показанным путем слишком труден, так что необходимы другие пути.

Самым простым способом обновления иерархии фреймов является создание рекурсивной функции, которая бы комбинировала матрицу преобразования фрейма с заданной матрицей преобразования. Далее матрица преобразования передается родственникам, а комбинированная матрица - потомкам. Посмотрите на такую функцию.

```
void UpdateHierarchy(D3DXFRAME_EX *Frame, \
           D3DXMATRIX matTransformation = NULL)
{
  D3DXFRAME_EX *pFramePtr;
  D3DXMATRIX matIdentity;
  // Использовать единичную матрицу, если ничего не было передано
  if(!matTransformation) {
   D3DXMatrixIdentity(&matIdentity);
   matTransformation = &matIdentity;
  }
  // Комбинировать матрицу преобразования с заданной
  matCombined = TransformationMatrix *  (*matTransformation);

  // Комбинировать с родственными фреймами
  if((pFramePtr = (D3DXFRAME_EX*)pFrameSibling))
   pFramePtr->UpdateHierarchy(matTransformation);

  // Комбинировать с дочерними фреймами
  if((pFramePtr = (D3DXFRAME_EX*)pFrameFirstChild))
   pFramePtr->UpdateHierarchy(&matCombined);
}
```

Как вы можете здесь видеть, функция UpdateHierarchy использует в качестве первого параметра объект D3DXFRAME_EX, который является текущим обрабатываемым фреймом. Вам необходимо вызвать функцию UpdateHierarchy только один раз, задав указатель на корневой фрейм; функция рекурсивно будет вызывать сама себя для каждого фрейма.

Второй параметр UpdateHierarchy - matTransformation. Параметр matTransformation является матрицей преобразования, накладываемой на преобразование фрейма. По умолчанию указатель matTransformation является NULL, означая, что будет использоваться единичная матрица при вызове UpdateHierarchy. После того как матрица фрейма комбинируется с заданным преобразованием, результирующее преобразование передается дочерним фреймам в качестве параметра matTransformation при следующих вызовах функции.

Замечание. *Если вы уже читали главу 1 (наверное, читали), тогда должны были заметить, что функция UpdateHierarchy содержится в классе D3DXFRAME_EX, так что вместо вызова функции UpdateHierachy, используя в качестве параметра корневой фрейм, вы можете использовать функцию-член корневого фрейма ::UpdateHierarchy! Для получения дополнительной информации об этой функции-члене читайте главу 1.*

Как я уже упоминал, вам необходимо просто вызвать функцию UpdateHierarchy, используя корневой фрейм. Не указывайте матрицу преобразования в качестве второго параметра, он будет использован при рекурсивных вызовах. Если же вы зададите матрицу преобразования, весь меш будет сдвинут на нее. Это то же самое, что установить матрицу преобразования мира для расположения объекта при визуализации.

```
// pRootFrame = корневой фрейм-объект D3DXFRAME_EX
UpdateHierarchy(pRootFrame);
```

После того как вы познакомились со скелетной структурой и работой с иерархиями костей, пришло время перейти ко второй части анимации - накладным скелетным мешам, которые меняют форму в соответствии с расположением иерархии костей.

Работа со скелетными мешами

В первой половине этой главы вы научились управлять иерархией костей, которые являются основой скелетной анимации. Это все хорошо, но игра с воображаемыми костями не очень занимательна. Людям, играющим в ваши игры, необходимо видеть всю тяжелую работу в виде визуализированных мешей, где и приходит время скелетных мешей.

Скелетные меши очень похожи на обычные, с которыми вы уже знакомы. Используя объект D3DXMESHCONTAINER_EX (как вы видели в главе 1), вы можете хранить данные ваших мешей от вершин и индексов до материалов и данных текстур, все это находится в удобном объекте ID3DXMesh. Что же касается фактических данных скелетного меша, они расположены в специальном объекте ID3DXSkinInfo.

Я пока пропущу описание объекта ID3DXSkinInfo и вместо этого объясню, чем же скелетные меши отличаются от остальных. Скелетные меши меняют форму в соответствии с расположением скелетной структуры. Вершины меша вращаются и поворачиваются вместе с костями. Вершины меша делают скелетный меш уникальным. Вы будете иметь дело с изменением положения вершин при работе со скелетными мешами.

Посмотрите на рис. 4.3, на котором изображен скелет, окруженный простейшим мешем.

На рис. 4.3 каждая вершина присоединена к кости. Когда кость двигается, вместе с ней двигаются и присоединенные к ней вершины. Например, если повернуть кость на 45 градусов по оси x, присоединенные к ней вершины также повернутся на 45 градусов, при этом точка скрепления костей будет является центом вращения.

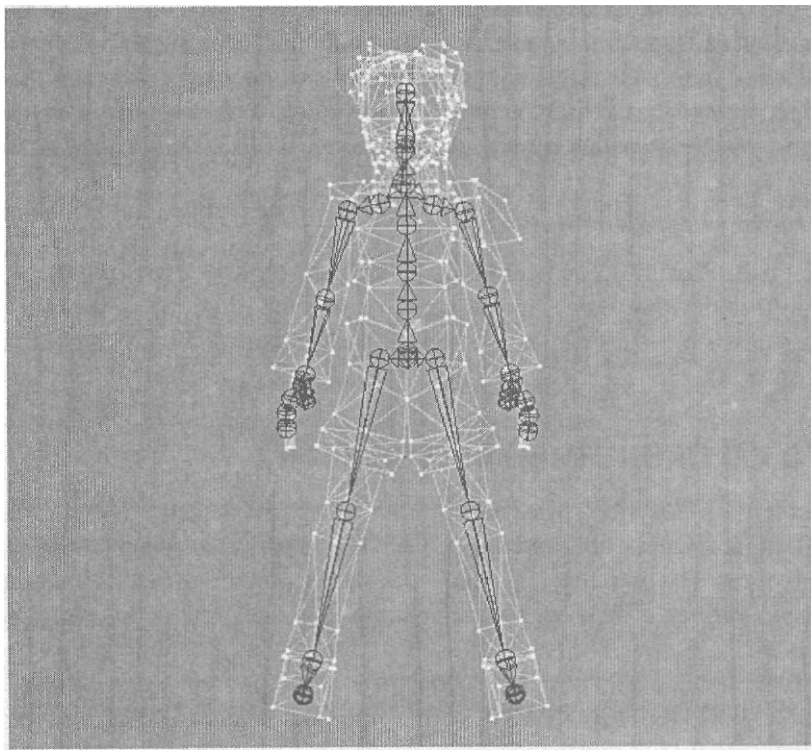


Рис. 4.3. При присоединении к скелетной структуре меш изменяется таким образом, чтобы каждая вершина соединялась с костью

Внимательно посмотрев на рис. 4.3 вы обнаружите, что некоторые вершины присоединены к нескольким костям. Так и есть - вы можете присоединять вершину более чем к одной кости. На самом деле, при использовании DirectX вы можете присоединять вершину к неограниченному числу костей, используя методы, которые вы изучите в этой книге. При движении кости, к которой присоединена вершина, вершина наследует не все движения. Например, если кость поворачивается на 60 градусов по оси z, присоединенная к ней вершина может повернуться только на 25 процентов от этого, т.е вершина повернется только на 15 градусов по оси z.

Точное значение процентного соотношения наследуемого вершиной движения назыпается весом вершины. Для каждой вершины скелетного меша назначается ее вес при присоединении ее к кости. Для вершин, присоединенных только к одной кости, это значение обычно 1.0, это говорит о том, что вершина наследует все движения

кости. Для вершин, присоединенных к нескольким костям, веса делятся на количество костей и, обычно, вычисляются, используя расстояние до каждой кости. (Большинство программ трехмерного моделирования сделает это за вас.) Например, вершина присоединена к двум костям, это означает, что оба веса будут по 0.5. Вершина будет наследовать только 50 процентов движения каждой кости. Заметьте, что сумма весов одной вершины всегда равняется 1.

Цель использования весов скелета достаточно оригинальна. Позволяя определенным костям влиять на заданные вершины, вы можете создавать привлекательные эффекты, такие как морщинистая кожа, неровности мускулов, растяжение одежды, - все это в реальном времени при анимации ваших объектов!

Путь, используемый DirectX для обработки весов вершин, достаточно прост. После загрузки меша, который вы будете использовать как скелетный, и весов вершин (также называемых скелетными весами), вы можете преобразовать вершины, чтобы они соответствовали расположению, костей используя следующие шаги:

1. Перебрать все вершины. Для каждой вершины выполнить шаг 2.
2. Для каждой кости, к которой присоединена вершина, получить преобразование.
3. Каждое преобразование кости умножить на матрицу вершинных весов и наложить полученное преобразование на комбинированное преобразование вершин.
4. Повторить шаг 3 для каждой присоединенной кости, шаги 2-4 повторить для всех вершин. После завершения применить комбинированную матрицу преобразования на заданную вершину (из шага 1).

И все таки, как получить веса вершин? Используя объект ID3DXSkinInfo, о котором я упоминал ранее, вы можете загружать веса из .X файлов. Скелетные веса хранятся в объекте Mesh, в конце его данных.

Каждой кости в скелетной структуре соответствует объект SkinWeights. Внутри объекта SkinWeights находится имя кости и следующее за ним количество присоединенных вершин. Заголовок скелетного меша определяет количество костей, к которым может быть присоединена каждая вершина. Если какая то из вершин присоединена к двум костям, тогда все вершины должны быть присоединены к двум костям. Для того чтобы вершины могли иметь разное количество костей, вы можете присваивать вес 0 ненужным костям.

Как я упоминал ранее, объект SkinWeights включает количество вершин, присоединенных к кости. Он перечисляет массив индексов вершин. После массива индексов вершин расположен массив значений весов вершин. Наконец, далее идет обратное преобразование костей для помощи в расположении вершин относительно соединений костей.

Посмотрите на пример объекта шаблона SkinWeights:

```
SkinWeights {
   "Bip01_R_UpperArm";
   4;
   0, 3449, 3429, 1738;
   0.605239, 0.605239, 0.605239, 0.979129;
   -0.941743, -0.646748, 0.574719, 0.000000,
   -0.283133, -0.461979, -0.983825, 0.000000,
   0.923060, -1.114919, 0.257891, 0.000000,
  -65.499557, 30.497688, 12.852692, 1.000000;;
}
```

В этом объекте используется кость "Bip01_R_UpperArm". К ней присоединены четыре вершины с индексами 0, 3449, 3429 и 1738. Вершина 0 имеет вес 0.605239, вершина 1 - 0.605239 и так далее. Матрица преобразования выравнивает перечисленные вершины относительно соединения костей. Эта матрица очень важна. Без нее вершины будут вращаться относительно центра мира, а не соединения костей.

К счастью, вам не нужно напрямую работать с шаблоном SkinWeights. Эти данные обрабатываются при загрузке скелетных мешей из .X файлов, используя функции D3DX.

Загрузка скелетных мешей из .X

Загрузка скелетных мешей из .X файлов очень похожа на загрузку обычных мешей. Используя специализированный анализатор .X файлов, вы должны перечислить объекты, содержащиеся в .X файле, при помощи функции ParseObject. Когда придет время обрабатывать объект Mesh, вместо вызова функции D3DXLoadMeshFromXof для загрузки данных меша используйте D3DXLoadSkinMeshFromXof, которая имеет дополнительный параметр - указатель на объект ID3DXSkinInfo. Посмотрите на прототип функции D3DXLoadSkinMeshFromXof, чтобы понять о чем я.

```
HRESULT D3DXLoadSkinMeshFromXof(
   IDirectXFileData *pXofObjMesh, // Объект .X файла
   DWORD Options, // Опции загрузки
   IDirect3DDevice9 *pDevice, // Используемое 3D устройство
   ID3DXBuffer **ppAdjacency, // объект буфера смежности
   ID3DXBuffer **ppMaterials, // объект буфера материала
   ID3DXBuffer **ppEffectInstances, // объекты экземпляров эффектов
   DWORD *pMatOut, // # материалов
   ID3DXSkinInfo **ppSkinInfo, // объект информации скелета!!!
   ID3DXMesh **ppMesh); // загруженный объект меш
```

Когда вы готовы загрузить меш из перечисленного шаблона Mesh, вызовите функцию D3DXLoadSkinMeshFromXof вместо D3DXLoadMeshFromXof. Убедитесь, что вы указали объект ID3DXSkinInfo, заданный в прототипе. Неважно, содержит ли шаблон Mesh скелетный меш, функция D3DXLoadSkinMeshFromXof загружает как обычные, так и скелетные меши. Вот пример:

```
// Определить структуры меша и информации скелетного меша
ID3DXMesh *pMesh;
ID3DXSkinInfo *pSkinInfo;

// Определить буферы для хранения данных материалов и смежностей
ID3DXBuffer *pMaterialBuffer = NULL, *pAdjacencyBuffer = NULL;

// DWORD для хранения количества загруженных материалов
DWORD NumMaterials;

// Загрузить скелетный меш из IDirectXFileDataObject pDataObj
D3DXLoadSkinMeshFromXof(pDataObj, D3DXMESH_SYSTEMMEM, \
            pDevice, &pAdjacencyBuffer, \
            &pMaterialBuffer, NULL, \
            &NumMaterials, &pSkinInfo, &pMesh);
```

Использование функции D3DXLoadSkinnedMeshFromXof еще не означает, что загруженный меш является скелетным. Сначала вам необходимо проверить объект pSkinInfo. Если он установлен в NULL, тогда скелетный меш не был загружен. Если же он правильный объект (не NULL), тогда вам необходимо проверить существование костей.

Самым простым способом проверить наличие костей является вызов ID3DXSkinInfo::GetNumBones. Эта функция возвращает количество костей, загруженных из шаблона Mesh. Если количество костей 0, то тогда их нет, и вы можете освобождать объект ID3DXSkinInfo (используя Release). Если же кости существуют, вы можете продолжать использование скелетного меша.

Посмотрите на этот пример, который тестирует, является ли загруженный меш скелетным. Если да, то проверяется наличие в меше костей.

```
// Установить флаг, если меш скелетный и есть кости
BOOL SkinnedMesh = FALSE;
if(pSkinInfo && pSkinInfo->GetNumBones())
  SkinnedMesh = TRUE;
else {
  // Освободить данные объекта информации скелетного меша
  if(pSkinInfo) {
   pSkinInfo->Release();
   pSkinInfo = NULL;
  }
}
```

Если флаг SkinnedMesh установлен в TRUE, то объект pSkinInfo является правильным, и вы готовы работать со скелетным мешем. Следующим шагом является создание еще одного объекта меша, который будет содержать меш, фактически деформируемый при изменении положения костей.

Создание контейнера вторичного меша

После того как вы создали скелетный меш, вам необходимо создать второй контейнер меша. Для чего спросите вы? Загруженный вами с помощью функции D3DXLoadSkinMeshFromXof объект скелетного меша является основной ссылкой на данные вершин меша. Т. к. эти вершины правильно расположены в соответствии с расположением костей, все запутается, если вы начнете изменять эти положения.

Давайте оставим их в покое и вместо этого создадим второй объект меша (ID3DXMesh), который будет содержать точную копию скелетного меша. Вам необходимо считать данные вершин из скелетного меша, наложить разнообразные преобразования костей и записать результирующие данные вершин в дублирующий контейнер меша (который я назвал вторичный меш или вторичный контейнер-меш), который используется при визуализации. Разумно, не правда ли?

Как я уже замечал ранее, вторичный меш в точности совпадает со скелетным мешем, начиная от количества вершин и заканчивая индексами. Самым простым методом копирования объекта скелетного меша является использование функции ID3DXMesh::CloseMeshFVF.

```
HRESULT  ID3DXMesh::CloneMeshFVF(
  DWORD Options,  // Флаги атрибутов меша
  DWORD FVF,  // FVF используемый для копирования
  IDirect3DDevice9 *pDevice,  // Используемое 3D устройство
  ID3DXMesh *ppCloneMesh);  // объект вторичного меша
```

Параметр Options функции CloneMeshFVF абсолютно такой же, как и при вызове функций D3DXLoadMeshFromX, D3DXLoadMeshFromXof и D3DXLoadSkinMeshFromXof, так что выбирайте сами. Я рекомендую установить его в 0, но вы можете свободно изменять его.

Что касается параметра FVF, вам просто необходимо предоставить FVF объекта скелетного меша, используя его функцию GetFVF. Не забывайте также указывать правильный объект IDirect3DDevice9 и указатель на объект ID3DXMesh, который будет вашим контейнером вторичного меша.

Вот небольшой кусочек кода, иллюстрирующий копирование скелетного меша для создания вторичного меша:

```
// pSkinMesh = объект ID3DXMesh
ID3DXMesh *pMesh; // контейнер вторичного меша
pSkinMesh->CloneMeshFVF(0, pMesh->GetFVF(), pDevice, &pMesh);
```

Все эти разговоры о копировании напоминают мне Star Wars Episode II: Attack of the Clones. Хорошей новостью является то, что ваши скопированные меши не будут захватывать вселенную... или будут? Эти копии не способны ничего сделать без небольшой поддержки, так что давайте вернемся к работе и посмотрим, что у нас на очереди.

После создания контейнера вторичного меша пришло время нанести ваши кости на иерархию фреймов. Почему мы не сделали этого ранее, когда я рассказывал о костях и фреймах? Все просто, данные костей были загружены до вызова функции D3DXLoadSkinMeshFromXof!

Сопоставление костей фреймам

Если вы просматривали .X файлы, вы могли заметить сходство объектов Frame и SkinWeights. Для каждой кости в вашей скелетной структуре есть соответствующий объект SkinWeights, встроенный в объект Mesh, который содержит имя объекта Frame (или ссылку Frame). Так и есть - каждая кость называется так же, как и соответствующий ей объект Frame!

После того как вы загрузили скелетный меш, вам необходимо присоединить каждую кость к соответствующему фрейму. Это делается простым перебором всех костей, получая имя каждой их них и находя соответствующий ей фрейм. Каждый соответствующий указатель на фрейм хранится в специальной структуре костей, разрабатываемой вами.

Для этой книги я встроил данные сопоставления костей в структуру D3DXMESHCONTAINER_EX, описанную в главе 1. Структура D3DXMESHCONTAINER_EX (созданная специально для этой книги и описанная ниже) добавляет к структуре D3DXMESHCONTAINER массив объектов текстур, контейнер вторичного меша и данные прикрепления костей.

```
struct D3DXMESHCONTAINER_EX : D3DXMESHCONTAINER
{
  IDirect3DTexture9 **pTextures;
  ID3DXMesh *pSkinMesh;

  D3DXMATRIX **ppFrameMatrices;
  D3DXMATRIX *pBoneMatrices;

  // .. дополнительнные данные и функции
};
```

Для этой главы важными являются переменные ppFrameMatrices и pBoneMatrices. Массив ppFrameMatrices содержит преобразования из иерархии костей; одна матрица преобразования накладывается на каждую вершину, принадлежащую соответствующей кости. Единственной проблемой является то, что матрица преобразования костей хранится не в виде массива, а отдельно для каждой кости в иерархии.

Структура D3DXMESHCONTAINER_EX предоставляет указатель для каждой матрицы преобразования костей, содержащейся в иерархии объектов D3DXFRAME_EX в массиве указателей (ppFrameMatrices). Используя эти указатели, вы можете поместить каждое преобразование в созданный в результате обновления скелетного меша массив pBoneMatrices.

Вы можете создать массивы указателей и матриц после загрузки иерархии костей, взяв количество костей в иерархии и создав указатели и объекты D3DXMATRIX:

```
// pSkinInfo = объект скелетного меша

// Получить количество костей в иерархии
DWORD NumBones = pSkinInfo->GetNumBones();

// Создать массив указателей D3DXMATRIX
// для указания на каждое преобразование костей
D3DXMATRIX *ppFrameMatrices = new D3DXMATRIX*[NumBones];

// Создать массив объектов D3DXMATRIX для хранения фактических
// преобразований, используемых при обновлении скелетного меша
D3DXMATRIX *pBoneMatrices = new D3DXMATRIX[NumBones];
```

После того как вы загрузили скелетный меш, вы можете установить указатели на каждое преобразование костей, получив имя каждой из них из информационного объекта скелетного меша. Используя эти имена, вы можете просмотреть список фреймов на их совпадения. Для каждой кости, у которой совпадает имя, установить указатель на матрицу преобразования фрейма. Когда все кости и фреймы проверены, вы можете перебрать весь список и скопировать матрицы в массив pBoneMatrices.

Сначала я вам покажу, как определять соответствие кости фрейму. Помните, ранее в этой главе я заметил, что кости имеют такие же имена, как и фреймы. Используя функцию ID3DXSkinInfo::GetBoneName вы можете получить имя кости для сравнения с фреймом.

```
// Просмотреть все кости и получить имя каждой
for(DWORD i=0;i<pSkinInfo->GetNumBones();i++) (

  // Получить имя кости
  const char *BoneName = pSkinInfo->GetBoneName(i);
```

После того как вы получили имя кости, вы можете просмотреть список фреймов в иерархии для поиска совпадений. Чтобы сделать это, используйте рекурсивный вызов функции FindFrame, разработанной в разделе "Изменение положения костей" ранее в этой главе.

```
// pRootFrame = корневой объект фрейма D3DXFAME_EX

// Находит совпадающее имя в фреймах
D3DXFRAME_EX *pFrame = pRootFrame->Find(BoneName);
```

Если фрейм с заданной костью именем найден, вы можете привязать комбинированную матрицу преобразования фрейма. Если не найден, то ссылка устанавливается в NULL.

```
// Сравнить фрейм с костью
if(pFrame)
  pMesh->ppFrameMatrices[i] = &pFrame->matCombined;
else
  pMesh->ppFrameMatrices[i] = NULL;
}
```

На данный момент вам могут быть непонятны причины сопоставления костей фреймам, но вам станет все ясно, когда придет время управления скелетными мешами и восстановления меша для его визуализации.

Управление скелетными мешами

Теперь ничто не останавливает вас от сумасшедших экспериментов с манипуляциями над скелетной структурой. Только сначала убедитесь, что вы управляете воображаемой структурой меша, а не вашей собственной, я ненавижу когда случайно принимаю такую позу, изменить которую не могу в течение часа! Шучу, конечно, теперь вы можете изменять положение фреймов в их иерархии. Этими фреймами являются те, которые связаны с костями.

Говоря об изменении положения фреймов, помните, что вы должны только вращать ваши кости; вы никогда не должны перемещать их. Масштабирование приемлемо, но будьте осторожны, помните, что преобразования распространяются по иерархии. Если вы масштабировали верхнюю руку персонажа, нижняя также будетотмасштабирована.

Я рассказывал об изменении положения различных костей ранее в этой главе, в разделе "Изменение положения костей", так что не хочу тут повторяться. После загрузки скелетной структуры и скелетного меша вы можете использовать изложенные выше методы преобразования костей. Когда вы будете готовы, вы можете обновить скелетный меш и подготовить его для визуализации.

Обновление скелетного меша

Когда ваша скелетная структура находится в желаемой позе, пришло время обновить (или восстановить) скелетный меш, чтобы он соответствовал ей. Прежде чем восстанавливать скелетный меш, вам необходимо убедиться, что контейнер вторичного меша создан и иерархия фреймов обновлена. Для повторения того, как создавать контейнер мешей, обратитесь к разделу "Создание контейнера вторичного меша", находящемуся ранее в этой главе. Чтобы освежить ваши воспоминания о том, как обновлять иерархию фреймов смотрите раздел "Обновление иерархии" ранее в этой главе. После того как вы их вспомните, вы можете продолжить.

Для обновления скелетного меша вы должны сначала заблокировать буферы вершин скелетного и вторичного меша. Это является очень важным, потому что DirectX возьмет данные вершин скелетного меша, наложит преобразования костей и запишет результирующие данные вершин в объект вторичного меша.

Однако сначала вам необходимо скопировать преобразования их фреймов в массив матриц (pBoneMatrices), хранимый в контейнере мешей. В то же самое время вам необходимо скомбинировать преобразования с обратным преобразованием костей. Обратное преобразование костей ответственно за перемещение вершин меша к его начальному положению, до того как вы примените фактические преобразования. Чтобы лучше понять это, посмотрите на рис. 4.4

На рис. 4.4 меш составлен их трех костей (фреймов) и некоторого количества вершин. Для применения преобразования к какому либо фрейму вам необходимо передвинуть вершины, принадлежащие фрейму, в начальное положение, а потом накладывать преобразования.

Вы перемещаете вершины относительно начального положения меша, прежде чем применять преобразования, т. к. матрица вращения просто поворачивает вершины относительно начального положения. Если бы вращали вершину, принадлежащую кости, вершина бы вращалась относительно начального положения меша, а не соединения костей. Например, если бы ваше тело было мешем, и вы бы согнули локоть, вершины, образующие меш руки, будут вращаться относительно локтя, а не центра вашего тела. После передвижения вершин к центру меша накладывается преобразование (чтобы поворот вершин соответствовал повороту костей) и, наконец, преобразовывается в положение.

Обычно, преобразования костей обратной матрицы хранятся в .X файле при использовании программы трехмерного моделирования для создания мешей. Если же это информация не содержится в .X файле, вы можете вычислить ее сами, сначала обновив иерархию фреймов, после чего обратив комбинированное преобразование фреймов, используя функцию D3DXMatrixInverse. Вот небольшой пример:

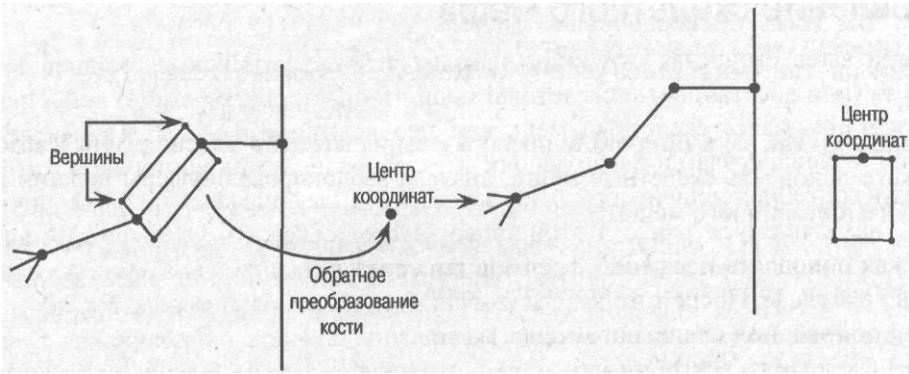


Рис. 4.4. Вершины могут только вращаться вокруг начального положения меша. Прежде чем вы примените преобразования костей, вам необходимо создать преобразование, которое перемещало бы вершины к начальному положению меша

```
// pRoot = корневой объект D3DXFRAME_EX
// pMesh = объект D3DXMESHCONTAINER_EX, содержащий данные меша

// Обновление иерархии фреймов
pRoot->UpdateHierarchy();

// Перебрать все кости для вычисления обратной матрицы
for(DWORD i=0;i<NumBones;i++)  {
  // Получить преобразование, используя матрицу кости
  D3DXMATRIX matBone =  (*pMesh->ppFrameMatrices);
  // Обратить матрицу
  D3DXMatrixInverse(&matBone, NULL, &matBone) ;

// Сохранить где-нибудь матрицу обратного преобразования кости
}
```

Однако вместо того чтобы вычислять все эти матрицы обратного преобразования костей самостоятельно, вы можете использовать объект скелетного меша для получения этой информации. Вызвав ID3DXSkinInfo::GetBoneOffsetMatrix, вы получите указатель на обратную матрицу преобразования кости. Умножив эту матрицу на матрицу преобразования фрейма, вы получите необходимый результат.

Используя только что приобретенные знания, переберите все кости, получите их обратное преобразование, скомбинируйте его с преобразованием фрейма и сохраните результат в массиве pBoneMatrices.

```
for(DWORD  i=0;i<pSkinInfo->GetNumBones();i++)  {

  // Установить обратное преобразование кости
  pMesh->pBoneMatrices[i]=(*pSkinInfo->GetBoneOffsetMatrix(i));
```

```
    // Наложить преобразование фрейма
    if(pMesh->ppFrameMatrices[i])
     pMesh->pBoneMatrices[i]  *=  (*pMesh->ppFrameMatrices[i]);
}
```

После того как вы скопировали преобразования костей в массив pBoneMatrices, вы можете обновлять скелетные меши, сначала заблокировав буферы вершин для скелетного и вторичного меша.

```
// pSkinMesh = контейнер скелетного меша
// pMesh = контейнер вторичного меша

// Заблокировать буферы вершин мешей
void *SrcPtr, *DestPtr;
pSkinMesh->LockVertexBuffer(D3DLOCK_READONLY, (void**)&SrcPtr);
pMesh->LockVertexBuffer(0,  (void**)&DestPtr);
```

После того как вы заблокировали буферы вершин, вам необходимо вызвать ID3DXSkinInfo::UpdateSkinnedMesh, чтобы наложить все преобразования костей на вершины и записать результирующие данные в контейнер вторичного меша. После чего просто разблокируйте буферы вершин, и вы готовы к визуализации!

```
// pSkinInfo = информационный объект скелетного меша

// Обновить скелетный, меш используя заданные преобразования
pSkinInfo->UpdateSkinnedMesh(pBoneMatrices, NULL, \
                  SrcPtr, DestPtr);

// Разблокировать буферы вершин мешей
pSkinMesh->UnlockVertexBuffer();
pMesh->UnlockVertexBuffer();
```

Визуализация скелетных мешей

Пришло время визуализировать вторичный меш и показать миру каково играть с силой ...силой скелетной анимации и скелетных мешей. Вам просто необходимо использовать функции отрисовки обычных мешей для визуализации вторичного меша. Перебрав все материалы, установите материал и текстуру и вызовете функцию ID3DXMesh::DrawSubset. Продолжайте выполнять вышеперечисленные действия, пока все наборы не будут отрисованы.

Если вы используете объект D3DXMESHCONTAINER_EX из главы 1, этот код замечательно визуализирует вторичный меш.

```
// pMesh - объект D3DXMESHCONTAINER_EX с данными материалов
// pMeshToDraw = указатель на вторичный меш
for(DWORD i=0;i<pMesh->NumMaterials;i++) {
```

```
    // Установить материал и текстуру
    pD3DDevice->SetMaterial(&pMesh->pMaterials[i].MatD3D);
    pD3DDevice->SetTexture(0,  pMesh->pTextures[i]);

    // Нарисовать набор меша
    pMeshToDraw->DrawSubset(i);
  }
```

Вот и все, что касается основ скелетной анимации! В следующих нескольких главах вы научитесь использовать скелетную анимацию для работы с заранее вычисленными анимациями, основанными на ключевых кадрах, комбинированием анимаций и кукольной анимацией. Наслаждайтесь!

Посмотрите демонстрационные программы

Не так быстро! Вы же хотите узнать о демонстрационных программах этой главы, расположенных на компакт-диске? Как показано на рис. 4.5, меш SkeletalAnim иллюстрирует изученные в этой главе загрузку скелетного меша (меш "Tiny.x", поставляемый с примерами DirectX SDK) и его визуализацию.

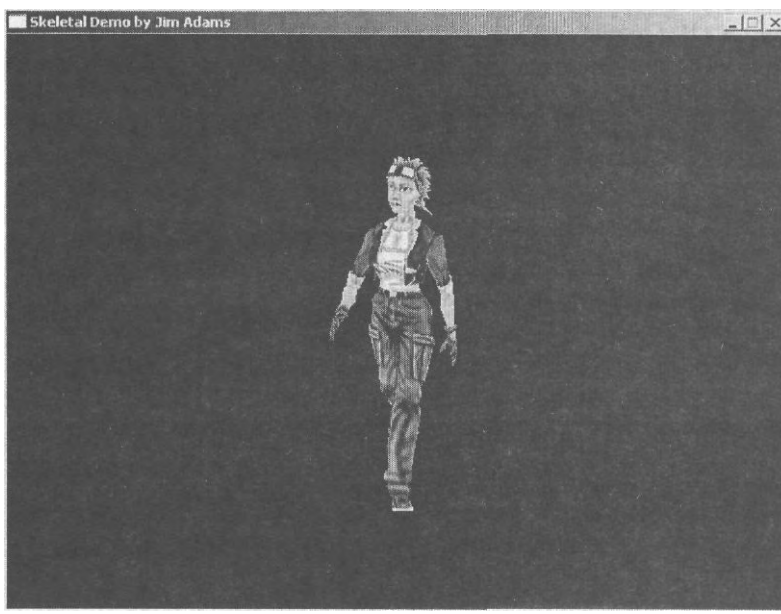


Рис. 4.5. Познакомьтесь с Tiny, скелетным мешем женщины Microsoft. Она создана из одного меша и соответствующей иерархии невидимых костей.

Программы на компакт-диске

В директории, соответствующей главе 4 этой книги, вы обнаружите единственный проект, иллюстрирующий работу со скелетной анимацией:

- **SkeletalAnim.** Эта демонстрационная программа иллюстрирует использование скелетных структур и скелетных мешей. Она расположена в \BookCode\Chap04\SkeletalAnim.

Глава 5

Использование скелетной анимации, основанной на ключевых кадрах

Предварительно вычисленная анимация на основе ключевых кадров является основой современных игровых движков. Приложив небольшие усилия, аниматор может создать полноценную анимационную последовательность, используя популярную программу трехмерного моделирования, а затем экспортировать данные анимации в формате, используемом движком игры. Не прикладывая никаких дополнительных усилий, вы можете изменять или модифицировать эти анимации без изменения игрового кода.

Я знаю, вы видели довольно много таких игр, и я также знаю, что вы хотели бы использовать анимацию, основанную на ключевых кадрах в ваших собственных играх. Эта глава как раз то, что вам нужно!

В этой главе вы научитесь:

- Работать с ключевыми кадрами анимации;
- Загружать данные анимации их .X файлов;
- Использовать наборы скелетной анимации;
- Преобразовывать файлы .MS3D в .X.

Использование наборов скелетных анимаций, основанных на ключевых кадрах

Если вы просматривали примеры DirectX SDK, вы, наверное, встречали небольшую демонстрационную программу SkinnedMesh, которая иллюстрирует использование хранимой в .X файле заранее вычисленной анимации, основанной на ключевых кадрах, для анимирования персонажа. Проблемой является то, что исходный код примера так замысловат и труден для понимания, что у вас голова пойдет кругом. Пример анимирования скелетного меша остается загадкой из-за практически полного отсутствия документации об использовании данных анимации .X файлов.

В главе 4 вы узнали, как работать с мешами, основанными на скелете, так называемыми скелетными мешами, которые связаны с соответствующей иерархией костей (скелетной структурой). Вершины скелетного меша присоединяются к костям и двигаются вместе с ними. Обычно анимирование достигается постепенным наложением набора преобразований на иерархию костей, в результате которых вершины повторяют их движения.

Последовательность таких анимационных преобразований называется ключевым кадром. Ключевые кадры используются для определения необходимого преобразования и времени его использования в анимационной последовательности. Где же взять эти преобразования, необходимые для движения костей? Хотя в вашем распоряжении и имеется множество форматов файлов, но чтобы придерживаться пути DirectX, сосредоточимся на использовании .X файлов.

Если вы посмотрите на .X файл демонстрационной программы SkinnedMesh (Tiny.x) из DirectX SDK, вы заметите, что наряду с обычными шаблонами Frame и Mesh используется шаблон AnimationSet и определенное количество встроенных объектов Animation и AnimationKey. Из этих объектов вы и получаете преобразования, используемые для анимирования иерархии костей скелетного меша. Посмотрите на некоторые из этих объектов анимации, расположенных в .X файле, чтобы понять о чем я говорю.

```
AnimationSet Walk {
 Animation {
  {Bip01}
  AnimationKey {
   4;
   3;
    0; 16; 1.00000, 0.00000, 0.00000, 0.00000,
           0.00000, 1.00000, 0.00000, 0.00000,
           0.00000, 0.00000, 1.00000, 0.00000,
           0.00000, 0.00000, 0.00000, 1.00000;;,
   1000; 16; 1.00000, 0.00000, 0.00000, 0.00000,
```

```
                0.00000, 1.00000, 0.00000, 0.00000,
                0.00000, 0.00000, 1.00000, 0.00000,
                0.00000, 0.00000, 0.00000, 1.00000;;,
    2000; 16; 1.00000, 0.00000, 0.00000, 0.00000,
                0.00000, 1.00000, 0.00000, 0.00000,
                0.00000, 0.00000, 1.00000, 0.00000,
                0.00000, 0.00000, 0.00000, 1.00000;;;
   }
  }
  Animation {
   {Bip01_LeftArm}
   AnimationKey {
     0;
     1;
     0; 4; 1.00000, 0.000000, 0.00000, 0.000000;;;
   }
   AnimationKey {
     1;
     1;
     0; 4; 1.000000, 1.00000, 1.000000;;;
   }
   AnimationKey {
     2;
    1;
    0; 3; 0.000000, 0.00000, 0.000000;;;
   }
  }
}
```

Вы смотрите на простую анимацию двух костей. Каждая анимация определяется в объекте данных AnimationSet; в предыдущем примере у анимации было имя Walk. Два объекта Animation содержат разнообразные ключи анимации, для каждой кости встроенные в объект AnimationSet. Ключи?! О чем это я говорю? Хорошо, мой друг, позвольте мне объяснить концепцию использования ключей при анимации.

Использование ключей при анимации

Ключ (сокращение от анимационный ключ) это временной маркер, который показывает изменение положения и/или ориентации кости. Анимация, использующая ключи, называется анимацией на основе ключевых кадров. Существуют весомые причины использования ключей, при этом одной из главных является экономия памяти.

Вы видите, что анимация является последовательностью движений (в данном случае движений костей) за установленный промежуток времени. Во время анимации изменяется иерархия костей для передачи движения анимации. Сохранение расположения и ориентации костей для каждой миллисекунды анимации является невозмож-

ным; слишком много данных для эффективного хранения. Вместо этого вы можете сохранять движение через более протяженные периоды времени (каждую секунду или две) или, еще лучше, во время значительных изменений положения или ориентации костей. Например, представьте вашу руку... хотя лучше представьте руку, изображенную на рис. 5.1.

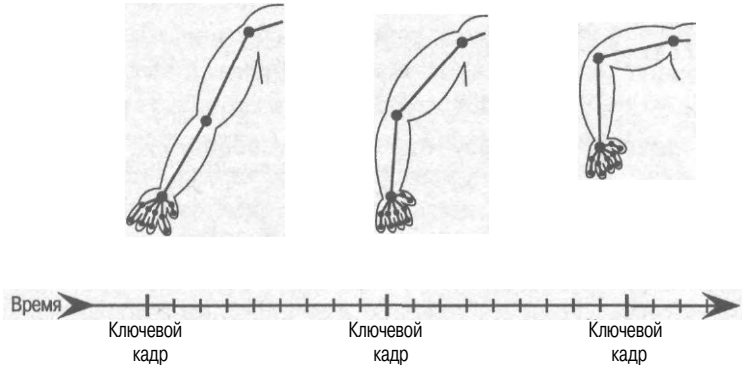


Рис. 5.1. Анимация костей во времени; ключевые отметки описывают каждое движение

Кости, которые на рис. 5.1 образуют руку, в начале анимации расположены на одной линии. Со временем кости изгибаются в воображаемом локте, затем фиксируются, а потом изгибаются на другой угол. Таким образом, в ориентации костей можно выделить три основных изменения: прямое положение (начальное положение), небольшой изгиб и большой изгиб в сочленении. Эти три изменения являются ключами в анимации.

Теперь, вместо того чтобы хранить ориентацию костей каждую миллисекунду, сохраним эти три ключа и точное время (в миллисекундах) достижения костями соответствующих ориентации. Для этого примера положим время начала анимации руки равное 0 миллисекунд, первый ключ (полу-изогнутая рука) в 500 миллисекунд, и последний ключ (полностью изогнутая рука) в 1200 миллисекунд.

Вот где удобно использовать ключевые кадры. Предположим, вы хотите вычислить ориентацию костей в заданное время... скажем в 648 миллисекунд. Это время как раз попадает между вторым и третьим ключами (148 миллисекунд после второго ключа). Далее положим, что для ориентирования каждой кости в анимации используется две матрицы.

```
D3DXMATRIX matKey1, matKey2;
```

Взяв каждый ключ и проинтерполировав их значения, вы получите преобразование, используемое для времени, находящегося между ключами. Например, прошло 648 миллисекунд анимации, вы можете проинтерполировать это преобразование так:

```
// Получить разность матриц
D3DXMATRIX matTransformation = matKey2 - MatKey1;

// Получить время ключей
float Key1Time = 500.0f;
float Key2Time = 1200.0f;

// Получить разность времен ключей
float TimeDiff = Key2Time - Key1Time;

// Получить скаляр, используя время анимации и разность времен
float Scalar = (648.0f - Key1Time) / TimeDiff;

// Вычислить интерполированную матрицу преобразования
matTransformation *= Scalar;
matTransformation += matKey1;
```

Вот и все! Теперь в матрице matTransformation содержится интерполированное преобразование, которое вы применяете к костям для синхронизации анимации! Для увеличения точности интерполяции вы можете использовать значения перемещения, вращения и масштабирования вместо матриц преобразования. Я расскажу как это делать немного позже, а пока давайте вернемся к шаблону Animation, с которым вам предстоит иметь дело.

Для каждой кости в иерархии должен быть соответствующий ей объект Animation. Сразу же после объявления объекта Animation вы увидите ссылочное имя объекта данных. Данные анимации кости, имеющей это имя, хранятся в предшествующем объекте AnimationKey. Это означает, что в предыдущем примере кости Bip01 и Bip01_LeftArm анимированы.

После ссылки на объект данных следует один или более объектов AnimationKey. Объект AnimationKey определяет используемые костью ключи анимации, которые могут включать ключи перемещения, вращения, масштабирования или преобразования. Давайте рассмотрим каждый тип ключа и способы хранения их информации в объекте более подробно.

Работа с четырьмя типами ключей

Пока что вы можете использовать четыре типа ключей в наборах анимаций, определяемых значением от 0 до 4, приведенном в .X файле после ссылки фрейма шаблона AnimationKey. Этими четырьмя значениями ключей являются:

- **Ключи вращения (тип 0).** Это кватернионы, хранимые в виде w, x, y и z.
- **Ключи масштабирования (тип 1).** Вы можете использовать эти ключи для анимирования процесса масштабирования. Ключи масштабирования используют три компонента для масштабирования по осям x, y и z.
- **Ключи перемещения (тип 2).** Эти ключи задают положение в трехмерном пространстве, используя координаты x, y и z. Вы можете хранить эти три значения как вектор.
- **Ключи матрицы преобразования (тип 4).** Вы можете использовать этот ключ для объединения всех преобразований в матрицу. Этот ключ использует 16 вещественных значений, представляющих собой однородную матрицу преобразования 4x4.

Возвращаясь к предыдущим объектам Animation, вы можете увидеть, что самый первый объект AnimationKey (который влияет на кость Bip01) определяет ключ матрицы преобразования (представленный значением 4) так:

```
{Bip01}
AnimationKey {
  4;
  3;
    0; 16; 1.00000, 0.00000, 0.00000, 0.00000,
           0.00000, 1.00000, 0.00000, 0.00000,
           0.00000, 0.00000, 1.00000, 0.00000,
           0.00000, 0.00000, 0.00000, 1.00000;;,
 1000; 16; 1.00000, 0.00000, 0.00000, 0.00000,
           0.00000, 1.00000, 0.00000, 0.00000,
           0.00000, 0.00000, 1.00000, 0.00000,
           0.00000, 0.00000, 0.00000, 1.00000;;,
2000; 16; 1.00000, 0.00000, 0.00000, 0.00000,
           0.00000, 1.00000, 0.00000, 0.00000,
           0.00000, 0.00000, 1.00000, 0.00000,
           0.00000, 0.00000, 0.00000, 1.00000;;;
}
```

Второй объект AnimationKey (влияющий на кость Bip01_LeftArm) использует три ключа: перемещения (значение 2), масштабирования (значение 1) и вращения (значение 0).

```
{Bip01_LeftArm}
AnimationKey {
  0;
  1;
  0; 4; 1.00000, 0.000000, 0.00000, 0.000000;;;
}
AnimationKey {
  1;
  1;
  0; 4; 1.000000, 1.00000, 1.000000;;;
}
```

```
AnimationKey {
  2;
  1;
  0; 3; 0.000000, 0.00000, 0.000000;;;
}
```

Как вы уже могли предположить, объект Animation может иметь неограниченное количество объектов AnimationKey, каждый из которых использует определенный тип ключа. После значения типа ключа (0=вращения, 1=масштабирования, 2=перемещения, 4=матрицы) следует количество ключей, используемых для анимирования заданной кости. Для первого набора костей (Bip01) задано три ключа матричного типа, в то время как для оставшегося объекта AnimationKey (влияющего на Bip01_LeftArm), используется только один ключ для каждого типа преобразования (вращения, масштабирования и перемещения).

Далее следуют данные ключей. Первым для каждого ключа является значение времени, заданное в выбираемом вами формате (секундах, миллисекундах, кадрах или любом другом). В моих примерах я всегда задаю время в миллисекундах. После времени следует число, определяющее количество используемых значений ключа. Например, посмотрите на данные ключа:

```
AnimationKey {
2; // Тип ключа
1; // # ключей
0; // Время ключа
3; // # значений данных ключа
10.00000, 20.00000, 30.00000;;; // данные ключа
}
```

Первое значение, 2, означает, что ключ используется для хранения перемещения. 1 означает, что только один ключ задан. Первый и единственный ключ расположен в момент времени, равный 0. После времени идет значение 3, которое указывает, что после него последует три числа (10,20 и 30). Эти три числа представляют собой координаты, используемые для заданного времени анимации.

Возвращаясь к ранее приведенному примеру, вы можете видеть, что первый ключ анимации (ключ преобразования) содержит три матрицы, используемые в моменты времени, равные 0, 1000 и 2000. Именно в эти моменты при анимации вы будете менять матрицы преобразования для кости Bip01.

Что же касается значений времени, находящихся между ключами, вы должны интерполировать матрицы для получения правильного преобразования. На самом деле, вы можете интерполировать любые типы ключей. Самым простым способом интерполирования является использование матрицы преобразования, значений масштабирования, перемещения или вращения ключа анимации, деленное на разность времен между двумя ключами и умноженное на время. Вы могли заметить, что

в предыдущем разделе я использовал линейную интерполяцию для матрицы преобразования. Вскоре я покажу, как интерполировать значения перемещения, масштабирования и вращения.

Вот и все что касается AnimationKey! Вам просто необходимо считать каждый ключ, содержащийся в объекте AnimationKey, и наложить его на соответствующее преобразование кости, используя для этого интерполирование матриц во времени.

Хорошо, пока достаточно о шаблонах анимации, объектах, ключах и интерполировании, вернемся к коду. Давайте рассмотрим, как загружать данные анимации в ваши игры. После этого я покажу вам, как с ними работать.

Считывание данных анимации из .X файлов

В главе 3 вы научились загружать меши и иерархии фреймов из .X файлов и использовать иерархии фреймов для деформирования (изменения) меша при визуализации. Целью данной главы является научить вас считывать данные анимации, содержащиеся в .X файлах, чтобы вы могли проигрывать анимации, основанные на ключевых кадрах.

Первым шагом для загрузки данных анимации является рассмотрение используемых шаблонов и создание нескольких классов, хранящих данные шаблонов. Посмотрите на шаблоны, с которыми вы будете работать:

```
template AnimationKey {
  <10DD4 6A8-775B-11cf-8F52-0040333594A3>
  DWORD keyType;
  DWORD nKeys;
  array TimedFloatKeys keys[nKeys];
}

template Animation {
  <3D82AB4F-62DA-11cf-AB39-0020AF71E433>
   [AnimationKey]
   [AnimationOptions]
   [...]
}

template AnimationSet {
  <3D82AB50-62DA-11cf-AB3 9-0020AF71E433>
   [Animation]
}
```

В начале списка расположен AnimationKey, который хранит тип данных ключа анимации, количество значений ключей и сами значения ключей, содержащиеся в массиве объектов TimedFloatKey, которые хранят время, и в массиве вещественных значений, хранимых в виде Время:ЧислоЗначений:Значения...

Шаблон Animation содержит объекты типа AnimationKey и AnimationOptions. Заметьте, что шаблон Animation является открытым, потому что ему необходима ссылка на данные фрейма для поиска соответствующей кости.

И, наконец, шаблон AnimationSet, содержащий объекты Animation. Вы можете хранить неограниченное количество анимаций в наборе; обычно для каждой кости используется одна анимация.

Замечание. *Хотя шаблон AnimationOptions и не используется в этой книге, он очень полезен, если вы хотите позволить художнику определять настройки проигрывания анимации. В шаблоне AnimationOptions вы найдете две переменных — openclosed и positionquality. Если openclosed установлен в 0, встроенная в объект анимация не повторяется циклически; значение 1 означает бесконечное повторение анимации. Установка параметра positionquality в 0 означает использование сплайновых положений, в то время как 1 означает использование линейных положений. Обычно вы устанавливаете positionquality в 1.*

Вам захочется использовать специализированные классы для хранения данных анимации; эти классы будут очень похожи на шаблон данных анимации. Сначала необходим класс, содержащий значения различных типов ключей - масштабирования, перемещения, вращения и преобразования. Первые два типа ключей, масштабирования и перемещения, используют вектор, так что для них подойдет общий класс.

```
classcAnimationVectorKey
{
  public:
   float m_Time;
   D3DXVECTOR3 m_vecKey;
};
```

Ключи вращения используют кватернион (четырехмерный вектор)

```
class cAnimationQuaternionKey
{
  public:
   float m_Time;
   D3DXQUATERNION m_quatKey;
};
```

Наконец, ключ преобразования использует матрицу 4x4.

```
classcAnimationMatrixKey
{
  public:
   float m_Time;
   D3DXMATRIXm_matKey;
};
```

Замечание. *Вы можете найти классы анимаций, обсуждаемые в данной главе (также как и полный исходный код) на компакт-диске книги. Посмотрите конец этой главы, чтобы получить дополнительную информацию о демонстрационной программе BoneAnim.*

Пока что неплохо. Помните, что каждая кость в анимации имеет собственный список используемых ключей, который хранятся в шаблоне Animation. Для каждой кости в иерархии есть соответствующий объект Animation. Соответствующий класс анимации должен содержать имя кости, к которой он присоединен, количество ключей каждого типа (перемещения, масштабирования, вращения и преобразования), указатель на связанный список данных и указатель на структуру кости (или фрейма), используемую в иерархии. Вам также необходимо создать конструктор и деструктор, который бы очищал данные класса.

```
class cAnimation
{
  public:
   char *m_Name; // имя кости
   D3DXFRAME *m_Bone; // указатель на кость
   cAnimation *m_Next; // следующий объект анимации в списке
   // # кол-во и массивы ключей каждого типа
   DWORD m_NumTranslationKeys;
   cAnimationVectorKey *m_TranslationKeys;
   DWORD m_NumScaleKeys;
   cAnimationVectorKey *m_ScaleKeys;
   DWORD m_NumRotationKeys;
   cAnimationQuaternionKey *m_RotationKeys;
   DWORD m NumMatrixKeys;
   cAnimationMatrixKey *m_MatrixKeys;

  public:
   cAnimation();
   ~cAnimation();
};
```

Наконец, шаблон AnimationSet содержит объект Animtation для всей иерархии костей. Все, что на данный момент требуется от класса набора анимаций, - это следить за массивом классов cAnimation (помните, что каждая кость в иерархии имеет соответствующий ей класс cAnimation) и за длиной полной анимации.

```
class cAnimationSet
{
  public:
   char *m_Name; // имя анимации
   DWORD m_Length; // длина анимации
   cAnimationSet *m_Next; // следующий набор в связанном списке
   DWORD m_NumAnimations;
   cAnimatTon *m_Animations;

  public:
   cAnimationSet();
   ~cAnimationSet();
};
```

Предположив, что вы хотите загрузить за раз больше одного набора анимации, вы можете создать класс, содержащий массив (или даже связанный список) классов cAnimationSet, означая, что вы можете получить доступ ко всем анимациям используя один интерфейс! Этот класс, называемый cAnimationCollection, наследуется от разработанного в главе 2 класса cXParser, так что вы можете анализировать .X файлы напрямую из класса, хранящего анимации.

Вот объявление класса cAnimationCollection :

```
class cAnimationCollection : public cXParser
{
  public:
   DWORD m_NumAnimationSets;
   cAnimationSet *m_AnimationSets;

  protected:
   // Анализировать объект .X файла
   BOOL ParseObject(IDirectXFileData *pDataObj,
            IDirectXFileData *pParentDataObj,
            DWORD Depth,
            void **Data, BOOL Reference);

   // Найти фрейм по имени
   D3DXFRAME *FindFrame(D3DXFRAME *Frame, char *Name);

  public:
   cAnimationCollection();
   ~cAnimationCollection();

   BOOL Load(char *Filename);
   void Free(};

   void Map(D3DXFRAME *RootFrame);
   void Update(char *AnimationSetName, DWORD Time);
};
```

На данный момент нам не важно подробное описание каждой функции в классе cAnimationCollection, так что вернемся к ним позже. Сейчас нам важно считать данные анимации их .X файла. Специализированный анализатор .X, содержащийся в классе cAnimationCollection, выполняет как раз это — загружает данные из объекта Animation в массив только что виденных вами объектов.

Для каждого объекта AnimationSet, найденного в анализируемом .X файле, вам необходимо создать класс cAnimationSet и добавить его к связанному списку уже загруженных наборов анимации. Последний загруженный объект cAnimationSet хранится в начале связанного списка, что облегчает определение, данные какого набора используются в данный момент.

Далее вы можете соответствующим образом анализировать объекты Animation. Если вы храните последний загруженный объект cAnimationSet в начале связанного списка, каждый следующий анализируемый объект Animation будет принадлежать текущему набору анимаций. То же самое и для объектов AnimationKey — их данные будут принадлежать первому объекту cAnimation в связанном списке.

Я пропущу конструкторы и деструкторы для всех классов, т. к. они необходимы исключительно для очистки и освобождения данных. Нас интересует только пара функций, первая из которых - cAnimationCollection::ParseObject, — имеет дело с каждым объектом, анализируемым в .X файле.

Функция ParseObject начинается с проверки, является ли текущий перечисляемый объект объектом AnimationSet. Если да, то новый объект cAnimationSet размещается и привязывается в список объектов, в то же время объект набора анимаций именуется для дальнейшего использования.

```
BOOL cAnimationCollection::ParseObject( \
          IDirectXFileData *pDataObj, \
          IDirectXFileData *pParentDataObj, \
          DWORD Depth, \
          void **Data, BOOL Reference)
{
  const GUID *Type = GetObjectGUID(pDataObj);
  DWORD i;

  // Проверить, является ли объект объектом AnimationSet
  if(*Type == TID_D3DRMAnimationSet) {

   // Создать и привязать объект cAnimationSet
   cAnimationSet *AnimSet = new cAnimationSet();
   AnimSet->m_Next = m_AnimationSets;
   m_AnimationSets = AnimSet;

   // Увеличить число наборов анимаций
   m_NumAnimationSets++;

   // Установить имя набора анимаций (если его нет, то установить
  // используемое по умолчанию)
  if(!(AnimSet->m_Name = GetObjectName(pDataObj)))
   AnimSet->m_Name = strdup("NoName");
}
```

Как вы можете видеть, здесь не происходит ничего необычного — вы просто создаете объект, который будет содержать данные объекта Animation. Далее вы анализируете объект Animation.

```
// Проверить, является ли объект объектом AnimationSet
if(*Type == TID_D3DRMAnimation && m_AnimationSets)  {
```

```
  // Добавить класс cAnimation на вершину cAnimationSet
  cAnimation *Anim = new cAnimation();
  Anim->m_Next = m_AnimationSets->m_Animations;
  m_AnimationSets->m_Animations = Anim;

  // Увеличить количество анимаций
  m_AnimationSets->m_NumAnimations++;
}
```

Опять же, здесь нет ничего необычного. В предыдущем коде мы просто убедились, что создается объект cAnimationSet и добавляется в начало связанного списка. Если он создан, вы можете создавать объект cAnimation и привязывать его в список объекта cAnimationSet.

Т. к. мы говорим об объекте cAnimation, следующий кусочек кода получает экземпляр фрейма, расположенного в объекте Animation

```
// Проверить, есть ли ссылка на фрейм в объекте анимации
if(*Type == TID_D3DRMFrame && Reference == TRUE && \
            m_AnimationSets && \
            m_AnimationSets->m_Animations) {

  // Убедиться, что родительским является объект Animation
  if(pParentDataObj && *GetObjectGUID(pParentDataObj) == \
              TID_D3DRMAnimation) {

  // Получить имя фрейма и сохранить его в качестве имени анимации
  if(!(m_AnimationSets->m_Animations->m_Name = \
              GetObjectName(pDataObj)))
   m_AnimationSets->m_Animations->m_Name=strdup("NoName");
  }
}
```

Из этого кода вы можете увидеть, что в объекте Animation разрешены только ссылочные объекты фрейма, которые вы можете проверить, используя GUID родительского шаблона. Гмм! Пока что код прост, не так ли? Я не хочу вас расстраивать, но самое сложное впереди! На самом деле самой сложной частью загрузки данных анимации из .X файлов является загрузка данных ключей. Но не пугайтесь; данными ключей является не что иное, как значение времени и массив значений, представляющий собой данные ключа.

Оставшийся код функции ParseObject определяет тип данных ключа, содержащегося в объекте AnimationKey. Код разделяется в зависимости от типа данных и считывает данные в заданные объекты ключей (m_RotationKeys, m TranslationKeys, mScaleKeys и m_MatrixKeys), находящиеся в текущем объекте cAnimation. Приглядитесь внимательно, и вы увидите, насколько код прост на самом деле:

```
// Проверить является ли объект типа AnimationKey
if(*Type == TID_D3DRMAnimationKey && m_AnimationSets && \
           m_AnimationSets->m_Animations) {

  // Получить указатель на верхнеуровневый объект анимации
  cAnimation *Anim = m_AnimationSets->m_Animations;

  // Получить указатель на данные
  DWORD *DataPtr = (DWORD*)GetObjectData(pDataObj, NULL);

  // Получить тип ключа
  DWORD Type = *DataPtr++;

  // Получить количество следующих далее ключей
  DWORD NumKeys = *DataPtr++;
```

В дополнение к проверке существования правильных объектов cAnimationSet и объекта cAnimation, находящегося в начале связанного списка объектов, предыдущий код получает указатель на данные ключа, тип ключа и количество следующих далее ключей. Используя тип ключа, код разделяется при создании объекта ключевого кадра и загрузке данных ключа.

```
// Разделение на основе типа ключа
switch(Type) {
  case 0: // Вращение
   delete [] Anim->m_RotationKeys;
   Anim->m_NumRotationKeys = NumKeys;
   Anim->m_RotationKeys = new \
            cAnimationQuaternionKey[NumKeys];
   for(i=0;i<NumKeys;i++) {
    // Получить время
    Anim->m_RotationKeys[i].m_Time = *DataPtr++;
    if(Anim->m_RotationKeys[i].m_Time > \
             m_AnimationSets->m_Length)
     m_AnimationSets->m_Length = \
             Anim->m RotationKeys[i].m_Time;

    // Пропустить # следующих далее ключей (должно быть 4)
    DataPtr++;

    // Получить значения вращения
    float *fPtr = (float*)DataPtr;
    Anim->m_RotationKeys[i].m_quatKey.w = *fPtr++;
    Anim->m_RotationKeys[i].m_quatKey.x = *fPtr++;
    Anim->m_RotationKeys[i].m_quatKey.y = *fPtr++;
    Anim->m_RotationKeys[i].m_quatKey.z = *fPtr++;
    DataPtr+=4;
}
break;
```

Как вы можете помнить из предыдущей части главы, ключи вращения используют кватернионы. Значения кватернионов хранятся в порядке w, x, y, z; для того, чтобы убедиться, что вы используете корректные значения, вам необходимо считывать их в таком же порядке.

Далее следует код для загрузки ключей перемещения и масштабирования, которые используют вектор для хранения информации.

```
case 1: // Масштабирование
  delete [] Anim->m_ScaleKeys;
  Anim->m_NumScaleKeys = NumKeys;
  Anim->m_ScaleKeys = new cAnimationVectorKey[NumKeys];
  for(i=0;i<NumKeys;i++) {
   // Получить время
   Anim->m_ScaleKeys[i].m_Time = *DataPtr++;
   if(Anim->m_ScaleKeys[i].m_Time > \
            m_AnimationSets->m_Length)
    m_AnimationSets->m_Length = \
            Anim->m_ScaleKeys[i].m_Time;

   // Пропустить количество далее следующих ключей (должно быть 3)
   DataPtr++;

   // Получить значения масштабирования
   D3DXVECTOR3 *vecPtr = (D3DXVECTOR3*)DataPtr;
   Anim->m_ScaleKeys[i].m_vecKey = *vecPtr;
   DataPtr+=3;
  }
  break;

case 2: // Перемещение
  delete [] Anim->m_TranslationKeys;
  Anim->m_NumTranslationKeys = NumKeys;
  Anim->m_TranslationKeys = new \
          cAnimationVectorKey[NumKeys];
  for(i=0;i<NumKeys;i++) {
   // Получить время
   Anim->m_TranslationKeys[i].m_Time = *DataPtr++;
   if(Anim->m_TranslationKeys[i].m_Time > \
            m_AnimationSets->m_Length)
    m_AnimationSets->m_Length = \
            Anim->m_TranslationKeys[i].m_Time;

   // Пропустить количество далее следующих ключей (должно быть 3)
   DataPtr++;

   // Получить значения перемещения
   D3DXVECTOR3 *vecPtr = (D3DXVECTOR3*)DataPtr;
   Anim->m_TranslationKeys[i].m_vecKey = *vecPtr;
   DataPtr+=3;
  }
  break;
```

Последним является код для считывания массива ключей преобразования.

```
case 4: // Матрица преобразования
  delete [] Anim->m_MatrixKeys;
  Anim->m_NumMatrixKeys = NumKeys;
  Anim->m_MatrixKeys = new cAnimationMatrixKey[NumKeys];
  for(i=0;i<NumKeys;i++)   {
   // Получить время
   Anim->m_MatrixKeys[i].m_Time = *DataPtr++;
   if(Anim->m_MatrixKeys[i].m_Time> \
            m_AnimationSets->m_Length)
    m_AnimationSets->m_Length = \
            Anim->m_MatrixKeys[i].m_Time;

   // Пропустить количество далее следующих ключей (должно быть 16)
   DataPtr++;

   // Получить значения матриц
   D3DXMATRIX *mPtr = (D3DXMATRIX *)DataPtr;
   Anim->m_MatrixKeys[i].m_matKey = *mPtr;
   DataPtr += 16;
  }
  break;
 }
}
```

Хорошо, передохните немного и давайте посмотрим, чего мы достигли. Пока что мы обработали каждый объект AnimationSet, Animation и AnimationKey (не беря во внимание ссылочный объект Frame, который содержит названия костей) и загрузили объекты ключей, содержащие данные анимации. Вы почти готовы начать анимирование!

Почти является правильным словом; остался один небольшой шаг - прикрепление объектов анимации к соответствующим им объектам костей.

Прикрепление анимации к костям

После загрузки данных анимации вам необходимо прикрепить классы анимации к соответствующим им костям в иерархии костей. Сопоставление иерархий имеет большое значение, т. к. как только анимация обновляется, вам необходимо быстро получить доступ к преобразованиям костей. Сопоставив, вы можете получить простой метод доступа к костям.

В данном примере иерархия костей будет представлена иерархией D3DXFRAME. Если вы используете DirectX 8, то могли заметить, что не можете использовать объект D3DXFRAME; его структура определена в DirectX 9. Не расстраивайтесь; вспомогательный код Direct3D, используемый во всех демонстрационных программах этой книги, компенсирует недостающую структуру наличием ложной версии D3DXFRAME,

которую вы можете использовать. Вы можете найти поддельную версию структуры D3DXFRAME в файле Direct3D.h, находящемся в директории этой главы компакт-диска.

Структура D3DXFRAME использует два связанных списка указателей, которые применяются для создания иерархии. Из корневой структуры D3DXFRAME вы можете получать доступ к дочерним объектам, используя указатель D3DXFRAME::pFrameFirstChild, и к родственным объектам, используя указатель D3DXFRAME::pFrameSibling.

Следующая функция в классе cAnimationCollection, на которую вам необходимо обратить внимание, - Map. Используйте функцию Map для прикрепления указателя m_Bone анимационной структуры к фрейму, имеющему в иерархии фреймов то же самое имя.

Функция Map просматривает все объекты cAnimationSet и находящиеся в них объекты cAnimation. Название каждого объекта cAnimation сравнивается с названием каждого фрейма; если найдено совпадение, в указатель cAnimation::m_Bone устанавливается адрес фрейма.

Функция Map имеет в качестве параметра корневой фрейм иерархии.

```
void cAnimationCollection::Map(D3DXFRAME *RootFrame)
{
  // Просмотреть все наборы анимаций
  cAnimationSet *AnimSet = m_AnimationSets;
  while(AnimSet != NULL) {

   // Просмотреть все объекты анимаций
   cAnimation *Anim = AnimSet->m_Animations;
   while(Anim != NULL) {

    // Просмотреть все фреймы в поисках совпадения
    Anim->m_Bone = FindFrame(RootFrame, Anim->m_Name);

    // Перейти к следующему объекту анимации
    Anim = Anim->m_Next;
   }

   // Перейти к следующему объекту набора анимаций
   AnimSet = AnimSet->m_Next;
  }
}
```

В то время как функция Map только просматривает все объекты cAnimationSet и cAnimation, функция FindFrame рекурсивно обрабатывает иерархию фреймов в поисках соответствия заданному имени. Когда такое имя найдено, функция FindFrame возвращает указатель на найденный фрейм. Посмотрите на код FindFrame, от которого зависит функция Map.

```
D3DXFRAME *cAnimationCollection::FindFrame(D3DXFRAME *Frame, char
*Name)
{
  D3DXFRAME *FramePtr;

  // Если нет фрейма, вернуть NULL
  if(!Frame)
   return NULL;

  // Вернуть текущий фрейм, если имя не задано
  if(!Name)
   return Frame;

  // Обработать дочерние фреймы
  if((FramePtr = FindFrame(Frame->pFrameFirstChild, Name)))
   return FramePtr;

  // Обработать родственные фреймы
  if((FramePtr = FindFrame(Frame->pFrameSibling, Name)))
   return FramePtr;

  // Ничего не найдено
  return NULL;
}
```

Можете расслабиться. Данные анимации были загружены, и вы прикрепили объекты анимации к иерархии костей. Осталось только обновить анимацию и установить матрицы преобразования для костей.

Обновление анимации

После прикрепления классов анимации к иерархии костей вы можете начать анимировать меши! Все, что необходимо сделать, - это просмотреть ключи анимации для каждой кости, накладывая интерполированные преобразования на преобразования костей перед визуализацией. Это можно сделать простым перебором каждого класса анимации и его ключей для нахождения используемых значений ключа.

Вернувшись к классу cAnimationCollection, вы можете увидеть, что всего одна функция выполнит все это за вас. Предоставив функции cAnimationCollection::Update в качестве параметра название используемого набора анимаций и время в анимации, все матрицы преобразования во всей иерархии прикрепленных костей будут установлены и готовы к визуализации.

Посмотрите на функцию Update, чтобы иметь представление о том, как обновлять данные анимации.

```
void cAnimationCollection::Update(char *AnimationSetName, \
DWORD Time)
{
  cAnimationSet *AnimSet = m_AnimationSets;
  DWORD i, Key, Key2;

  // Искать совпадающее название набора анимации, если установлено
  if(AnimationSetName) {

   // Искать совпадающее название набора анимации
   while(AnimSet != NULL) {

    // Прекратить, если совпадение найдено
    if(!stricmp(AnimSet->m_Name, AnimationSetName))
     break;

    // Перейти к следующему объекту набора анимаций
    AnimSet = AnimSet->m_Next;
   }
  }

  // Вернуть, если набор не был найден
  if(AnimSet == NULL)
   return;
```

Функция Update начинает работу с просмотра наборов анимаций, загруженных в связанный список. Если вы установите значение NULL в качестве AnimationSetName, Update просто будет использовать первый набор анимаций в списке (который обычно является последним загруженным). Если при использовании заданного названия не было найдено совпадений, функция без промедления прекращает работу.

Однако, как только найден соответствующий набор анимаций, функция продолжает работу, просматривая каждый объект cAnimation в нем. Для каждого объекта анимации ищется ключ (перемещения, масштабирования, вращения и преобразования), который необходимо использовать для заданного значения времени.

После того как подходящий ключ обнаружен, значения (вращения, масштабирования, перемещения или преобразования) интерполируются и вычисляется матрица результирующего преобразования. Далее эта матрица хранится в прикрепленной кости (на которую указывает указатель m_Bone).

Вы уже видели, как просматривать список ключей для поиска тех, между которыми находится заданное время, так что я пропущу этот код. Смотрите его на компакт-диске книги; для получения дополнительной информации о демонстрационной программе BoneAnim читайте конец главы.

После вычисления преобразований, накладываемых на каждую кость из данных анимации, вы можете вернуться в игру и визуализировать меш, используя методы, изученные в главе 1. Помните, вы должны наложить матрицы преобразования на

каждую кость соответствующей вершины, и лучшим способом сделать это является использование вершинных тейдеров. Если вам необходимо, вы можете обратиться к главе 1, чтобы вспомнить как визуализировать скелетные меши.

Получение скелетных данных меша из альтернативных источников

Формат файла Microsoft .X является не единственным, используемым для хранения данных мешей и анимаций. Есть еще два формата, сравнимые с ним по простоте использования, которые являются замечательными для хранения данных мешей и анимаций, - форматы chUmbaLum sOft's Milkshape 3D .MS3D и id Software's Quake 2 .MD2.

Формат файла Milkshape .MS3D похож на двоичный .X, за исключением того, что файл .MD3D хранит только один меш и иерархию костей. На самом деле, формат .MS3D является очень простым, т. к. не использует шаблоны; вместо этого в файле хранится заранее определенная последовательность структур.

Формат файла Quake 2 .MD2 является простым набором мешей, помещенных в один файл. Каждый меш представляет собой один кадр анимации из последовательности ее наборов. В то время как файлы .MS3D содержат скелетные анимации, формат .MD2 содержит только наборы морфируемых анимаций.

Замечание. *Морфированная анимация мешей является еще одной привлекательной темой, освещенной в книге, так что я просто указал на двойственность программы MeshConv. Пока что просто игнорируйте любые упоминания морфированной анимации и вернитесь к теме: преобразование наборов скелетных анимаций, основанных на ключевых кадрах из .MS3D в .X.*

Итак, для скелетных анимаций вы можете использовать файлы .MS3D, а для морфируемых анимаций .MD2. Как же конкретно их использовать? Существует множество информации об этих форматах. Посмотрите книжку Focus On 3D Models (Premier Press, 2002) или веб-сайты http://www.gamedev.net или http://nehe.gamedev.net.

Компакт-диск содержит программу MeshConv, которую вы можете использовать для преобразования файлов .MS3D и .MD2 в .X. После запуска программы вы увидите диалоговое окно MeshConv, показанное на рис. 5.2.

Не дайте неказистости кнопок в программе MeshConv напугать вас - она конвертирует все файлы .MS3D и .MD2 files в .X, используя шаблоны, виденные вами в этой главе. Файлы .MS3D сохраняются, используя иерархию фреймов и один объект AnimationSet, в то время как файлы .MD2 сохраняются, используя набор объектов Mesh и MorphAnimationSet, которые хранят названия мешей, используемых для морфированной анимации.

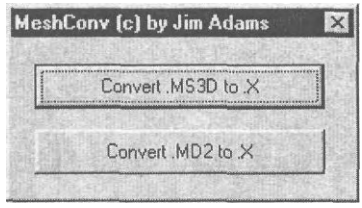


Рис. 5.2. Диалоговое окно MeshConv содержит две кнопки, щелкнув на которые вы можете преобразовать файлы .MS3D и .MD2 в .X

Замечание. *На компакт-диске содержится полностью комментированный исходный код программы MeshConv. Читайте конец этой главы, чтобы подробнее узнать о программах и их местоположениях.*

Для преобразования файла (.MS3D или .MD2) в .X файл щелкните по соответствующей кнопке в диалоговом окне MeshConv. Появится диалоговое окно "Open File". Это диалоговое окно позволяет вам перемещаться по директориям для указания расположения конвертируемого файла. Выберете необходимый для конвертации файл и нажмите "Open".

Появится диалоговое окно "Save .X File". Вы можете использовать его для указания имени и положения файла, в котором вы хотите сохранить данные мешей и анимации. Введите имя файла и нажмите "Save". Через некоторое время вы должны увидеть окно сообщений, говорящее о том, что преобразование было выполнено успешно.

Теперь вы готовы использовать .X файлы при помощи классов, разработанных ранее в этой главе, для загрузки наборов скелетной или морфируемой анимации. Набор скелетных анимаций использует один исходный меш, деформируемый (shaped) иерархией костей; для получения дополнительной информации об использовании скелетных мешей читайте эту главу и главу 2.

Для морфируемых анимаций (из .MD2) используется последовательность объектов Mesh, содержащих все цели морфирования меша из исходного файла. Всего один объект MorphAnimationSet поможет вам загрузить данные анимации в ваш проект, используя ранее изученные классы и технологии.

В качестве примера работы с конвертированным, при использовании программы MeshConv, .X файлом смотрите демонстрационные программы, включенные в эту главу. Вы правильно поняли - программы BoneAnim и MorphAnim используют конвертированные файлы .MS3D и .MD2 для иллюстрирования скелетной и морфируемой анимации. Посмотрите их и получайте удовольствие!

Посмотрите демонстрационные программы

В этой главе вы научились загружать наборы анимаций и использовать их данные для анимирования мешей. Для лучшего иллюстрирования этих концепций анимации я создал программу (SkeletalAnim), которая показывает вашу любимую даму скелетной анимации, Microsoft's Tiny (из примеров DirectX SDK), делающую то, что она может лучше всего, - ходящую! При запуске приложения вы увидите что-то подобное изображенному на рис. 5.3.

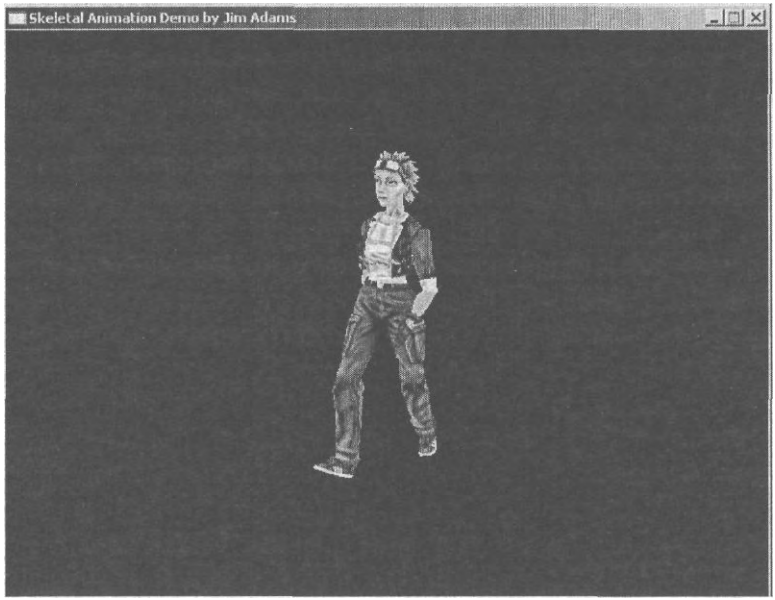


Рис. 5.3. Двигающаяся Tiny в демонстрационной программе SkeletalAnim! Эта программа иллюстрирует использование анимированных скелетных мешей

Программы на компакт-диске

В директории главы 5 компакт-диска этой книги вы найдете следующие две демонстрационные программы, которые вы можете использовать в своих проектах:

- **MeshConv.** Вы можете использовать эту полезную программу для преобразования файлов .MS3D и .MD2 в .X. Она расположена в \BookCode\Chap05\MeshConv.
- **SkeletalAnim.** Эта программа показывает, как считывать и использовать скелетную анимацию, основанную на ключевых кадрах. Она расположена в BookCode\Chap05\SkeletalAnim.

Глава 6

Комбинирование скелетных анимаций

В главе 5 "Использование скелетной анимации, основанной на ключевых кадрах", вы видели, как использовать заранее созданные последовательности анимаций в ваших игровых проектах. Это хорошо; единственная проблема в том, что эти анимации статические. Так и есть, анимации никогда не изменятся и всегда будут оставаться постоянными, сколько раз вы бы их не проигрывали.

На самом деле, проделав небольшую работу, вы можете соединить последовательности плавных анимаций в набор новых динамических анимаций, которые будут уникальными при каждом воспроизведении. Например, комбинируя игровую анимацию передвижения персонажа с его отдельной анимацией удара, вы можете сделать так, чтобы персонаж шел и бил одновременно!

Все правильно, соединив (иногда говорят смешав) движения разнообразных анимаций, вы можете создать сотни новых, используя заранее вычисленные наборы анимаций, основанных на ключевых кадрах. Эта глава покажет, как комбинировать анимации.

В этой главе вы научитесь:

- Комбинировать наборы скелетных анимаций;
- Изменять программный код работы со скелетной анимацией для поддержки комбинированнойанимации.

Комбинирование скелетных анимаций

Как я уже упоминал во введении этой главы, обычно вы используете последовательности заранее вычисленных анимаций на основе ключевых кадров в ваших игровых проектах. Вы создаете эти анимации при помощи программ трехмерного моделирования, таких как 3DStudio Max фирмы Discreet или trueSpace фирмы Caligari. Хотя они и прекрасно справляются со своей задачей, у этих заранее вычисленных последовательностей анимаций отсутствует самое главное - уникальность. Это означает, что анимации всегда одинаковы, независимо от того, сколько раз вы их проиграли.

Двигаясь к миру более динамических анимаций, технология комбинирования анимаций становится более чем актуальной. Как так, вы не знаете что такое комбинирование анимаций? Комбинирование анимаций - это возможность смешивать или комбинировать различные анимации для получения новых.

Например, на рис. 6.1 показано, как вы можете смешать анимацию движения вашего персонажа с анимацией размахивания рукой для создания анимации, в которой персонаж будет идти и махать рукой одновременно!

Вам не надо останавливаться на комбинировании двух анимаций, вы можете соединять три, четыре или даже десять различных анимаций в одну уникальную! С каждой новой добавляемой анимацией возможности комбинирования возрастают экспоненциально. Имея небольшой набор анимаций, вы можете создать сотни новых, используя их комбинирование.

Я не буду вам врать - теория и практическое воплощение комбинирования анимаций мучительно... просты. Так и есть; комбинирование анимаций является одной из тех вещей, о которых вы говорите: почему я не додумался сделать это раньше, это на самом деле просто! Вам тоолько нужно определиться со способом соединения разнообразных преобразований структуры костей.

Соединение преобразований

Как вы видели в главах 4 и 5, скелетная анимация просто является последовательностью матриц преобразования, применяемых к костям скелетной структуры меша. Эти преобразования включают перемещение, масштабирование и вращение. Большей частью преобразования являются вращениями. Кости поворачиваются в сочленениях; обычно только корневая кость может перемещаться по миру, и даже тогда лучше преобразовывать мир (чем напрямую перемещать кости). Принимая во внимание все вышесказанное еще раз заметим, что анимация создается с помощью преобразований.

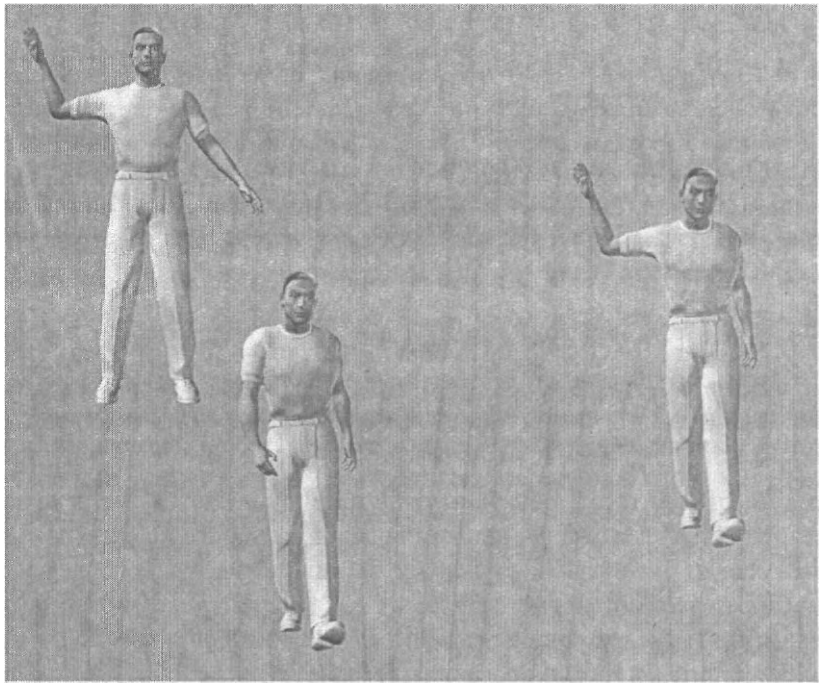


Рис. 6.1. Хотя две анимации, показанные слева, и определены раздельно, вы можете соединить их в одну уникальную анимацию, как показано справа

Как вы можете видеть на рис. 6.2, можно создавать новые позы, добавляя разнообразные преобразования скелетной структуры к уже существующим. Например, чтобы двигать руку скелета, добавьте вращательное преобразование к матрице преобразования кости руки. Постепенное увеличение угла вращения, добавленного к преобразованию кости, создает плавную анимацию.

Вы можете видеть, что анимация создана комбинированием (через перемножение матриц) или непосредственным сохранением набора преобразований анимации и матриц преобразования скелета. Для плавного анимирования меша вы можете использовать линейную интерполяцию, соизмеряя матрицы преобразования набора анимаций со временем.

Так что на самом примитивном уровне вы имеете дело с преобразованием матриц для создания анимации; существует одна матрица преобразования, накладываемая на каждую кость в меше. Набор предварительно вычисленных анимаций является источником матриц преобразования, накладываемых на преобразования кости.

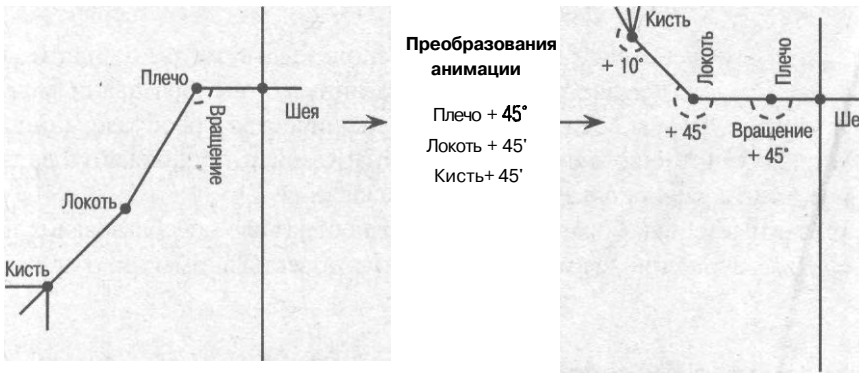


Рис. 6.2. Начальная поза скелета имеет присоединенный набор матриц преобразования; при их комбинировании с матрицами преобразования набора анимаций будут созданы новые позы

Подумайте, вместо того чтобы брать одну матрицу преобразования из набора анимаций (матрицу ключевого кадра или комбинацию из набора ключевых кадров положения, перемещения и вращения), почему бы просто не взять последовательность матриц преобразования, которые действуют на одну и ту же кость, из различных наборов анимаций и не скомбинировать их? В конце концов, ведь мы же используем объединение матриц для комбинирования множественных преобразований, так почему бы просто не добавить еще несколько преобразований из различных анимаций?

Стоп! Вот я и попался, вы не можете просто перемножить матрицы и получить правильный результат. Подумайте: умножение матриц не коммутативное, т. е. имеет значение порядок, в котором вы перемножаете различные преобразования. Если бы вы перемножали два преобразования, оба из которых сначала поворачивают, а потом перемещают, вы бы получили результирующее преобразование, которое бы поворачивало и перемещало, поворачивало и перемещало. Очевидно, что здесь слишком много преобразований для данных кости, которая обычно поворачивается, а затем перемещается.

Для решения этой проблемы вам необходимо складывать преобразования вместо того чтобы перемножать их. Так, для примера, два предыдущих преобразования, которые поворачивают, а потом перемещают, в результате дадут преобразование, которое повернет, а потом сдвинет (вместо того чтобы поворачивать и перемещать, поворачивать и перемещать). Сложение преобразований является идеальным решением!

Сложение двух матриц (представленных объектами D3DXMATRIX) очень просто, что показано на следующем примере:

```
D3DXMATRIX matResult = Matrix1 + Matrix2;
```

Выполнив эти действия, вы можете использовать матрицу matResult для преобразований; она является комбинированным преобразованием матриц Matrix1 и Matrix2. Чтобы соединить большее количество преобразований анимации, просто добавьте ещё одну матрицу к matResult и продолжайте делать это пока не соедините все используемые преобразования.

Теперь вы знаете, как можно соединять разнообразные преобразования наборов анимаций. Для упрощений вы можете расширить объекты анимаций, разработанные в главе 5.

Улучшение объектов скелетной анимации

После того как вы узнали насколько просто комбинировать несколько скелетных анимаций, почему бы вам не применить эти знания и не добавить их к объектам скелетной анимации и коду, рассмотренным в главе 5? Звучит заманчиво: добавив всего одну функцию к классу cAnimationCollection, вы будете комбинировать анимации совсем как профессионалы.

На самом деле, вместо того чтобы смешивать новый код с кодом cAnimationCollection, давайте просто унаследуем от него новый класс, который бы отвечал за комбинированные анимации. Этот новый унаследованный класс cBlendedAnimationCollection будет определяться так:

```
class cBlendedAnimationCollection : public cAnimationCollection
{
  public:
    void Blend(char *AnimationSetName,
         DWORD Time, BOOL Loop,
         float Blend = 1.0f);
};
```

Ничего себе, вот это маленький класс! В классе cBlendedAnimationCollection объявлена только одна функция Blend, которая выполняет обязанности функции cAnimationCollection::Update. Вы спросите, почему просто не унаследовать функцию Update? При помощи cBlendedAnimationCollection вы сможете использовать как обычные наборы анимаций из главы 5, так и только что разработанные наборы комбинированных анимаций.

Давайте рассмотрим поподробнее функцию Blend, после чего я покажу вам, как использовать недавно созданный класс.

```
void cBlendedAnimationCollection::Blend( \
             char *AnimationSetName, \
             DWORD Time, BOOL Loop, \
             float Blend)
{
```

У прототипа функции Blend имеется четыре параметра, первый из которых - AnimationSetName. При вызове Blend вам необходимо передать в качестве AnimationSetName имя набора анимаций, который вы будете смешивать с текущей. Помните, в главе 5, каждый набор анимаций, содержащийся в .X файле имеет собственное уникальное имя (определяемое именем экземпляра объекта AnimationSet). Вам необходимо задать в качестве AnimationSetName соответствующее имя из .X файла.

Я вернусь к наборам анимаций немного позже. А пока продолжим рассмотрение прототипа Blend. Вторым параметром Blend является Time, который представляет собой время в анимации, используемое для комбинирования. Если анимация длится 1000 миллисекунд, то вы можете изменять Time в диапазоне от 0 до 999. Задание значения большего, чем длительность анимации, вынудит функцию Blend использовать последний ключевой кадр для комбинирования анимаций.

А что относительно циклического повтора анимации? За это ответственен третий параметр - Loop. Если вы установите Loop в FALSE, тогда ваша анимация не будет обновляться, если задаваемое время больше, чем длительность анимации. Однако, если Loop установлен в TRUE, функция Blend проверяет величину времени таким образом, чтобы оно всегда попадало в продолжительность анимации.

Предыдущий абзац может показаться непонятным поначалу, поэтому для большей понятности представьте себе такую функцию:

```
void UpdateAnimation(DWORD Elapsed)
{
  static DWORD AnimationTime = 0; // время анимации

  // Вызвать Blend, используя AnimationTime в качестве времени ани-
мации
  AnimationBlend.Blend("Walk", AnimationTime, FALSE, 1.0f);

  // Обновить время анимации
  AnimationTime += ELapsed;
}
```

Функция UpdateAnimation предназначена для отслеживания времени анимации при помощи статической переменной. Во время каждого вызова UpdateAnimation функция Blend используется для комбинирования анимации Walk на основе времени, задаваемого AnimationTime. Положим, что анимация Walk имеет продолжительность 1000 мил-

лисекунд, а время между вызовами функции UpdateAnimation 50 мс, таким образом, анимация закончится после 20 вызовов функции. Это означает, что после 20 вызовов функции UpdateAnimation анимация остановится (т. к. Loop установлен в FALSE).

Вернувшись и установив Loop в TRUE, мы вынудим функцию проверять значения времени, для убеждения, что оно всегда меньше длительности анимации. Когда я говорю проверяем, я имею ввиду модульное вычисление. Я покажу вам, как использовать модульное вычисление немного позже; а пока вернемся к четвертому и последнему параметру.

Последним параметром является Blend, который представляет собой вещественное число, скаляр, используемый для изменения матрицы смешанного преобразования перед ее применением к скелетной структуре. Например, если вы комбинируете анимацию ходьбы, но хотите, чтобы только половина преобразования была применена, тогда установите Blend в 0.5.

Хорошо, достаточно о параметрах; давайте перейдем к коду функции! Если вы использовали функцию cAnimationCollection::Update, вы заметите, что большая часть кода функции Blend такая же. В начале кода вы найдете кусочек, который проверяет связанный список наборов анимаций для нахождения заданного параметром AnimationSetName.

```
cAnimationSet *AnimSet = m_AnimationSets;

// Если задано, искать соответствующее имя набора анимации
if(AnimationSetName) {

  // Найти соответствующее имя набора анимации
  while{AnimSet != NULL)  {

    // Остановиться, если совпадение найдено
    if(!stricmp(AnimSet->m_Name, AnimationSetName))
      break;

    // Перейти к следующему объекту набора анимаций
    AnimSet = AnimSet->m_Next;
   }
  }

// Вернуть, если ничего не найдено
if(AnimSet == NULL)
  return;
```

Если вы установите в качестве параметра AnimationSetName NULL, то функция Blend будет использовать первый набор анимаций, находящийся в связанном списке. Если же вы зададите имя в AnimationSetName и оно не будет найдено в связанном списке, то функция Blend завершит работу.

После того, как вы получили указатель на соответствующий объект набора анимаций, вы можете проверить время, основываясь на заданный параметр Time и значение циклического флага Loop.

```
// Ограничить время длительностью анимации
if(Time > AnimSet->m_Length)
  Time = (Loop==TRUE)?Time % \
         (AnimSet->m_Length+1):AnimSet->m_Length;
```

В самом деле предыдущий маленький изящный кусок кода выполняет две вещи в зависимости от флага Loop. Если Loop установлен в FALSE, то Time сравнивается с продолжительностью анимации (AnimSet->m_Length). Если Time больше длительности анимации, тогда она устанавливается равной продолжительности анимации, таким образом блокируясь на последней секунде (далее - последнем ключевом кадре) анимации. Если Loop установлен в FALSE, то в результате модального вычисления значение Time всегда лежит в пределах длительности анимации (от 0 до AnimSet->m_Length).

После вычисления Time, используемого для анимации, просматривается список костей в скелетной структуре. Для каждой кости вам необходимо отследить все комбинированные преобразования соответствующих ключевых кадров. Для каждого ключевого кадра, найденного в анимации, вам необходимо прибавить (не умножить) его преобразование к преобразованиям скелетной структуры.

```
// Просмотреть все анимации
cAnimation *Anim = AnimSet->m_Animations;
while(Anim) {

  // Обрабатывать, если присоединена к кости
  if(Anim->m_Bone) {

   // Сбросить преобразование
   D3DXMATRIX matAnimation;
   D3DXMatrixIdentity(&matAnimation);

   // Наложить разнообразные матрицы для преобразования
```

После этого вы просматриваете каждый ключевой кадр (основываясь на их типах) и вычисляете преобразование, которое необходимо наложить на скелетную структуру. Для экономии места я только приведу код, который просматривает матричные ключи.

```
// Матрица
if(Anim->m_NumMatrixKeys && Anim->m_MatrixKeys) {
 // Сделать цикл для нахождения матричного ключа
 DWORD Key1 = 0, Key2 = 0;
 for(DWORD i=0;i<Anim->m_NumMatrixKeys;i++) {
  if(Time >= Anim->m_MatrixKeys[i].m_Time)
   Key1 = i;
 }
```

```
// Получить значение второго ключа
Key2 = (Key1>=(Anim->m_NumMatrixKeys-1))?Key1:Key1+1;

// Получить разность времен ключей
DWORD TimeDiff = Anim->m_MatrixKeys[Key2].m_Time-
      Anim->m_MatrixKeys[Key1].m_Time;
if(!TimeDiff)
 TimeDiff = 1;

// Вычислить используемое значение скаляра
float Scalar = (float)(Time - \
 Anim->m_MatrixKeys[Key1].m_Time) / (float)TimeDiff;

// Вычислить интерполированную матрицу
D3DXMATRIX matDiff;
matDiff = Anim->m_MatrixKeys[Key2].m_matKey - \
  Anim->m_MatrixKeys[Key1].m_matKey;
matDiff *= Scalar;
matDiff += Anim->m_MatrixKeys[Key1].m_matKey;

// Соединить с преобразованием
matAnimation *= matDiff;
}
```

Я привел код, показанный в главе 5, так что я не буду объяснять его здесь еще раз. Вкратце, код ищет ключевые кадры и вычисляет соответствующее преобразование. Это преобразование хранится в матрице matAnimation.

Начиная отсюда, код будет совершенно отличен от функции cAnimationCollection::Update. Вместо того чтобы хранить матрицу преобразования (matAnimation) в объекте фрейма скелетной структуры, вычисляется разность между преобразованием matAnimation и начальным преобразованием скелетной структуры (сохраненным в matOriginal при загрузке скелетной структуры). Эта разность масштабируется, используя вещественный параметр Blend, и результирующее преобразование добавляется (не умножается как при concoction) к преобразованию фрейма скелетной структуры. Это позволяет убедиться, что преобразования корректно комбинируются при использовании соответствующих значений.

После этого просматриваются ключевые кадры следующей кости, и цикл продолжается, пока все кости не будут обработаны.

```
// Получить разность преобразований
D3DXMATRIX matDiff = matAnimation - Anim->m_Bone->matOriginal;

// Скорректировать на значение смешивания
matDiff *= Blend;

// Добавить к матрице преобразования
Anim->m_Bone->TransformationMatrix += matDiff;
}
```

```
    // Перейти к следующей анимации
    Anim = Anim->m_Next;
   }
  }
```

Мои поздравления, вы завершили создание функции Blend! Давайте попробуем ее в работе! Предположим, что вы уже загрузили меш и иерархию фреймов, а теперь хотите загрузить последовательность анимаций из .X файла. Далее предположим, что в .X файле (названном Anims.x) содержатся четыре набора анимаций - Stand, Walk, Wave и Shoot. Две из них используются для ног и две для рук и торса. Вот пример кода, загружающего наборы анимаций:

```
// pFrame = корневой фрейм в иерархии фреймов
cBlendedAnimationSet BlendedAnims;
BlendedAnims.Load("Anims.x");

//  Нанести иерархию фреймов анимации
BlendedAnims.Map(pFrame);
```

После того как вы загрузили набор анимаций, вы можете скомбинировать их перед обновлением и визуализацией скелетных мешей. Предположим, вы хотите скомбинировать анимации Walk и Shoot, обе из которых используют сто процентов преобразований. Для начала вам необходимо сбросить преобразования иерархии фреймов к их начальному состоянию. Это означает, что вам необходимо скопировать преобразование D3DXFRAME_EX::matOriginal в преобразование D3DXFRAME_EX::TransformationMatrix. Этот момент очень важен, т. к. является базой, к которой добавляются все преобразования при смешивании.

Замечание. *Объект D3DXFRAME_EX является расширенной версией объекта DirectiD D3DXFRAME. Вы можете прочитать о нем в главе 1.*

```
// Использовать D3DXFRAME_EX::Reset для сброса преобразований
pFrame->Reset();
```

После того как преобразования были сброшены к их начальным состояниям, вы можете комбинировать наборы анимаций.

```
// AnimationTime = время с начала анимации

// смешать анимацию ходьбы
BlendedAnims.Blend("Walk",  AnimationTime, TRUE,  1.0f);

// смешать анимацию стрельбы
BlendedAnims.Blend("Shoot",  AnimationTime, TRUE,  1.0f);
```

После того как вы скомбинировали все используемые наборы анимаций, вам необходимо обновить иерархию фреймов, которая используется для обновления скелетного меша.

```
// Обновить иерархию фреймов
pFrame->UpdateHierarchy();
```

После того как вы обновили иерархию (соединив матрицы преобразований и сохранив результат в D3DXFRAME_EX::matCombined), вы можете обновить скелетный меш и визуализировать его! Я больше не буду углубляться в детали; вам остается посмотреть демонстрационную программу комбинирования анимаций, расположенную на компакт-диске, чтобы увидеть, как использовать полезные функции и объекты, разработанные в главе 1, при комбинировании анимаций.

Посмотрите демонстрационные программы

Хотя в этой главе и одна демонстрационная программа, но зато какая! Демонстрационная программа SkeletaLAnimBlend (смотрите рис. 6.3), иллюстрирующая технологию смешивания анимаций, показывает персонаж Microsoft Tiny во всей ее красе!

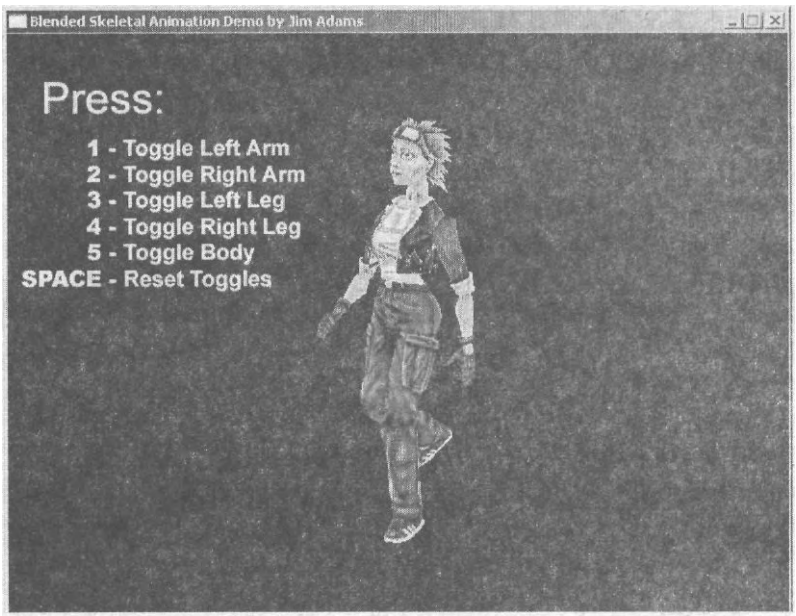


Рис. 6.3. Исследуйте обретенные технологии смешивания скелетных анимаций, комбинируя анимации в реальном времени

Я отредактировал файл "Tiny.x", разбив анимацию на несколько наборов. Существуют наборы анимации для ног, рук и тела. Нажав любую из клавиш, изображенных на экране, вы можете комбинировать соответствующие наборы анимаций. Для примера, нажатие 1 включает комбинирование анимационной последовательности для левой руки. Когда разрешена анимация левой руки, Tiny качает ею при ходьбе. Когда отключена, ее левая рука висит.

Для показания истинной силы смешивания, предположим, что вы добавили новый набор анимаций в файл Tiny.x, при котором Tiny машет рукой вперед назад. Вам просто необходимо выключить смешивание качания и скомбинировать анимацию махания для создания новой и совершенно уникальной анимации.

Программы на компакт-диске.

В директории главы 6 компакт-диска книги вы обнаружите один проект, иллюстрирующий использование смешивания скелетных анимаций. Этим проектом является:

- **SkinAnimBlend.** Этот проект иллюстрирует смешивание множественных анимаций для создания новых. Он расположен в \BookCode\Chap06\SkeletalAnimBlend.

Глава 7

Создание кукольной анимации

Прячась наверху выступа восточной части компаунда, я лежу и жду, не покажется ли неосторожный противник. Мой палец, лежащий на спусковом механизме, тревожно подергивается, ожидая своей очереди приступить к действию. Я уже представляю, как противник выйдет из двери подо мной и я ударю. Мой снаряд из базуки взорвется, и виртуальное тело противника взлетит в воздух, отскакивая от каменных стен бараков компаунда. Мой план идеален. Через некоторое время выясняется, что мой противник также труслив, как и я. Я слышу его зловещий хохот и осматриваюсь, как раз вовремя, чтобы увидеть, как брошенная им граната приземляется прямо рядом со мной. Похоже, что сегодня мое воображаемое тело будет отскакивать от стен. Ну и ладно, всегда можно сыграть еще раз!

Типичный момент из типичной стрелялки от первого лица; в этой истории нет ничего необычного за исключением той части, где персонаж подпрыгивает в результате попадания в него из различных типов игрового оружия. Такие эффекты как уникальные анимации смерти возможны в результате использования технологии, известной как кукольная (rag doll) анимация, в которой ваши персонажи как бы сделаны из маленьких кусочков. Как только ваш персонаж умирает, загружаются короткие, абсолютно уникальные наборы движений - отлетания от препятствий с движением частей тела, как если бы вы бросили настоящую куклу через комнату.

Такие игры как Unreal Tournament 2003 от Epic Games используют кукольную анимацию для последовательностей смертей персонажей, и поверьте мне, если вы не видели этот эффект, вы многое пропустили. Если же вы хотите использовать кукольную анимацию в ваших собственных проектах, то вам повезло - все необходимое содержится в этой главе!

В этой главе вы научитесь:

- Превращать игровых персонажей в кукол;
- Использовать динамику твердого тела в анимациях;
- Накладывать силы, вращательные моменты и импульсы на движения;
- Использовать пружины для соединения частей тела;
- Работать с обнаружением столкновений и реакцией на них.

Создание кукол из персонажей

Кукольная анимация является новой причудой в современной анимации. Как следует из названия, кукольную анимацию можно представить, как если бы мы взяли какой-нибудь объект (например игровой персонаж) и бросили его, при этом бы все его части случайным образом вертелись, подобно тряпичной кукле. Представьте куклу, имеющую размеры человека, которую подбросили в воздух. Ее руки и ноги будут вращаться, а если тело столкнется с каким-нибудь твердым объектом, движения станут еще более непредсказуемыми.

Вот где кукольная анимация проявляется во всей красе, создавая абсолютно уникальные анимации при каждом моделировании. Вы можете учитывать коэффициенты столкновений, гравитации, ветра и других сил для изменения движения куклы.

Игры, подобные Unreal Tournament 2003 иллюстрирует, что такие замечательные технологии как кукольная анимация могут сделать. Нет ничего лучше, чем взорвать персонаж вашего противника из ракетницы, после чего его тело отлетит подобно кукле. Определенно вы захотите использовать эту технологию анимации для своих игр, а если продолжите читать эту главу, то узнаете о ней.

Основная идея кукольная анимации проста. Возьмем ваш игровой персонаж и рассмотрим на его примере, что происходит при кукольной анимации. Этот персонаж (представим, что он выглядит, как персонаж, изображенный на рис. 7.1) состоит из скелетного меша и последовательности костей, образующих скелетную структуру. Здесь никаких проблем - вы узнали о скелетных мешах и скелетных структурах из главы 4.

Представим, что каждую кость (и каждую вершину, принадлежащую этой кости) из рис. 7.1 окружает параллелепипед, ограничивающий параллелепипед, если быть точным. (Хотя это и кажется странным - работать с параллелепипедами вместо фактической скелетной структуры или меша; по мере чтения вы поймете, зачем это необходимо.) Видите ли, на самом деле нас интересуют эти параллелепипеды; они представляют собой части тела персонажа, которые могут крутиться, поворачиваться и болтаться в кукольной анимации.

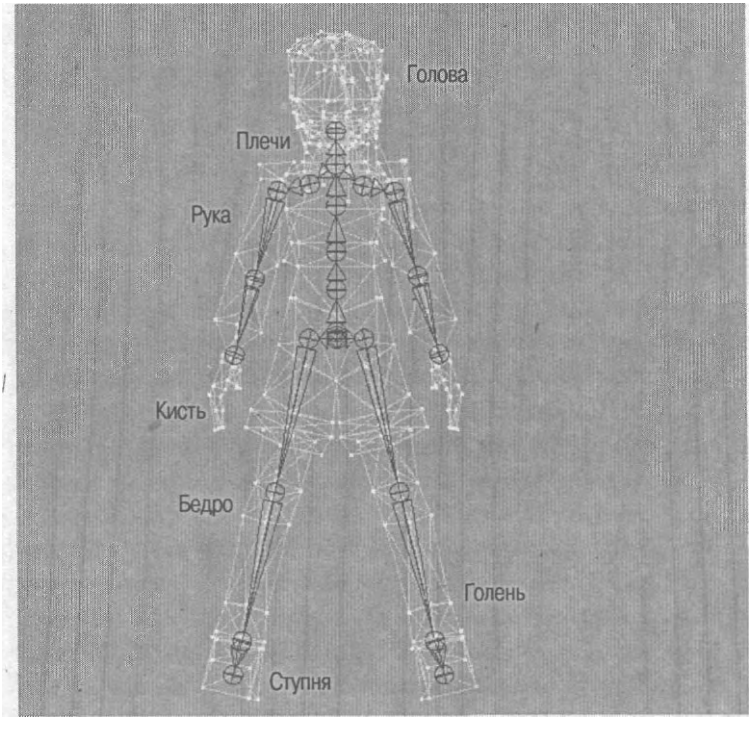


Рис. 7.1. Вы можете разбить демонстрационный персонаж, состоящий из скелетной структуры и скелетного меша, на последовательность отдельных компонентов

На рис. 7.2 показан внешний вид демонстрационного персонажа, как если бы он состоял только из ограничивающих параллелепипедов.

Вашей задачей является обработка и отслеживание движений этих параллелепипедов (а не костей и вершин) во время их перемещений в анимации. Вы можете передвигать и вращать их, как вам захочется. Так как параллелепипеды представляют кости персонажа, каждая кость в скелетной структуре двигается в соответствии с параллелепипедом, к которому она присоединена (даже если движение вызывает разрыв соединений костей, о котором я расскажу чуть ниже).

Как же кости двигаются в соответствии с движением параллелепипедов? Они наследуют преобразования ограничивающих параллелепипедов, которые движутся по вашему трехмерному миру. Так что теоретически, при перемещении параллелепипеда кость двигается соответственно. (Получается, что кость как бы не существует, потому что вы работаете только с параллелепипедами при моделировании.)

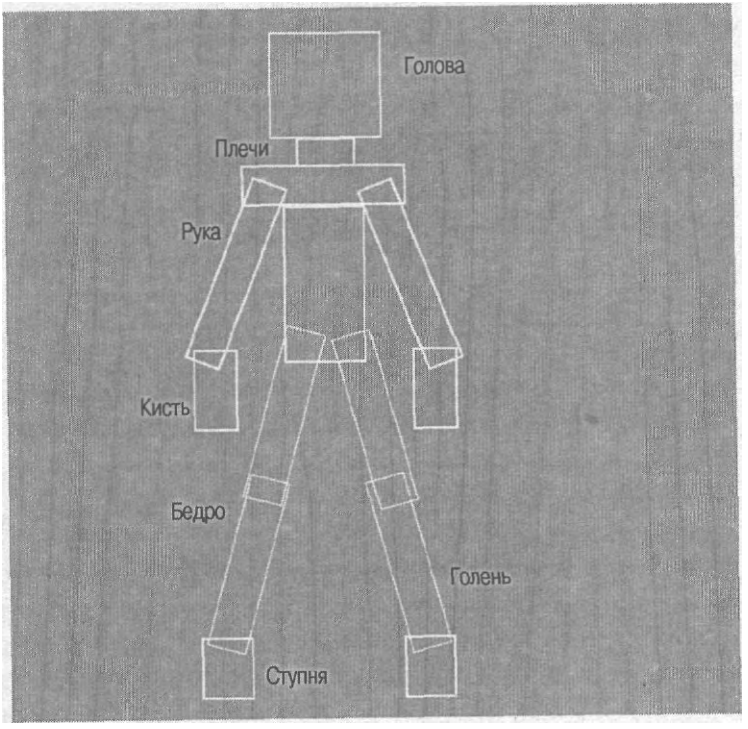


Рис. 7.2. Кости и вершины демонстрационного персонажа были заменены ограничивающими параллелепипедами, которые соответствую пространству, занимаемому костями и вершинами

Постепенно вам должно становиться понятно, почему я выбрал параллелепипеды для представления костей и их вершин. Вы можете представить каждую кость и ее вершины всего восьмью точками (углами ограничивающего параллелепипеда) вместо неизвестного (и зачастую большего) количества вершин в меше. Кроме этого, математически проще отследить перемещение параллелепипеда, чем костей и вершин.

Предположим, вы создали набор параллелепипедов, которые представляют кости и вершины. Как вы можете видеть на рис. 7.3, каждый параллелепипед полностью включает соответствующие вершины кости и точки соединения костей. Эти параллелепипеды преобразовываются таким образом, чтобы их положение и ориентация совпадали с положением и ориентацией соответствующих им костей.

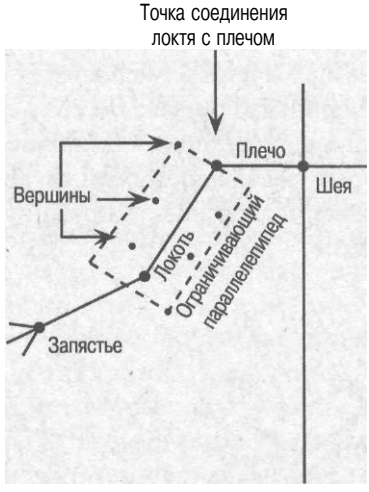


Рис. 7.3. Каждый ограничивающий параллелепипед окружает вершины кости и точки соединения костей

После создания параллелепипедов и помещения их вокруг трехмерного меша вы можете начинать двигать их. Предположим, вы хотите, чтобы руки вашего персонажа крутились. Приложите небольшую силу к нескольким параллелепипедам, и они начнут двигаться. Соответствующие этим параллелепипедам кости также начнут двигаться.

С этой точки зрения параллелепипеды являются отдельными сущностями; на них не распространяются соединения костей, существующие в скелетной структуре (эти же соединения позволяют хранить скелетную структуру в распознаваемом виде). В некотором смысле это хорошо, т. к. позволяет напрямую управлять скелетной структурой персонажа, используя преобразования параллелепипедов вместо комбинированных преобразований костей. Это означает, что вам не надо комбинировать все локальные преобразования костей с родительской для их корректного ориентирования, достаточно просто скопировать преобразование параллелепипеда, и все готово!

Единственной проблемой является то, что параллелепипеды могут смещаться друг относительно друга, что приводит к расчленению персонажа. Должен быть способ усилить соединения костей, используя ограничивающие параллелепипеды, чтобы сохранить персонаж целым в сочленениях (и избавить вас от жуткой сцены летящих в разную сторону частей тела).

На самом деле есть несколько способов убедиться, что ограничивающие параллелепипеды соединяются в тех же точках, что и соответствующие им кости. Метод, который я покажу вам в этой главе, использует пружины для возвращения ограничивающего параллелепипеда при каждом его перемещении. На рис. 7.4 вы можете видеть несколько отсоединившихся ограничивающих параллелепипедов, летающих по сцене, которые расчленяют бедный персонаж на части. Между каждым параллелепипедом вы можете видеть пружину, которая используется для соединения частей тела вместе.

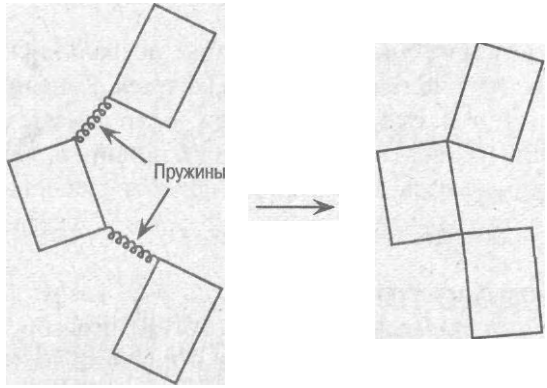


Рис. 7.4. Набор пружин позволяет вернуть отсоединившиеся параллелепипеды

После перемещения параллелепипедов и использования пружин для восстановления точек соединения вам необходимо скопировать ориентацию параллелепипеда в иерархию костей, обновить скелетный меш и визуализировать его. Видите, как просто!

Конечно, это просто в теории, но вот практическая реализация является трудной задачей. Все, что вы прочитали, на самом деле является большой физической проблемой - как отследить ориентацию, движение и взаимодействия параллелепипедов? Используя физику твердого тела, вот как!

Работа с физикой твердого тела

Как вы только что читали, тело вашего персонажа может быть разделено на параллелепипеды, которые представляют собой разнообразные кости, образующие скелетную структуру. Вам необходимо отслеживать изменения ориентации этих параллелепипедов при их перемещении во время анимации и копировать эти ориентации в их начальные кости.

Хорошо, хорошо, но как это связано с физикой твердого тела? Ну, часть "твердого тела" означает, что ваши меши полагаются твердыми объектами - параллелепипедами, представляющими кости, которые никогда не меняют форму и никогда не проникают в пространство других объектов. Поэтому ограничивающие параллелепипеды, представляющие кости, являются твердыми объектами.

Изучение физики твердого тела (часто называемой динамикой твердого тела) отслеживает движение твердых объектов, включая действие сил, таких как гравитации и трения. Взаимодействие также играет важную роль, потому что твердые объекты, представляющие части тела персонажа, должны отскакивать друг от друга и от окружающей территории.

Прежде чем вы возьмете учебник по физике позвольте сказать, что на самом деле физика твердого тела не так сложна, как кажется. Конечно, в ней есть множество формул и вычислений, от которых даже у учителя математики кошмары, но после того как вы разберетесь, не будет никаких причин для волнения.

На самом деле я буду излагать материал физики твердого тела шаг за шагом, чтобы вам было понятнее, начав с создания твердого объекта.

Создание твердого тела

Физика твердого тела - это система слежения за движением и взаимодействие твердых объектов. Для упрощения положим, что твердыми объектами являются трехмерные ограничивающие параллелепипеды, содержащие кости скелетной структуры. В пространстве параллелепипеды состоят из восьми точек (по одной в каждом углу).

Точки являются аналогами вершин, параллелепипед является аналогом меша. Используя простые преобразования (вращение и перемещение), вы можете расположить параллелепипед и его восемь угловых точек где угодно в трехмерном мире. Каждый параллелепипед с его угловыми точками имеет собственное место в системе твердых тел. Параллелепипед представляет точки как целое - что влияет на параллелепипед, то влияет и на точки. Таким образом, если вы переместите параллелепипед, точки переместятся вместе с ним, вам нет необходимости заботиться о точном расположении точек внутри него.

Что касается точек, они не просто помогают определять размер ограничивающего параллелепипеда, они также помогают определять столкновения. Видите ли, если хотя бы одна из этих точек лежит внутри пространства другого объекта, тогда можно сказать, что объекты сталкиваются, и вам необходимо обрабатывать их. Это сильно

облегчает нахождение столкновений; вместо того чтобы проверять каждую вершину меша, попадает ли она в объект, вы можете проверять точки ограничивающего параллелепипеда. Я вернусь к обнаружению столкновений немного позднее; а пока я хочу продолжить определение твердого тела.

Как я уже упоминал, каждый ограничивающий параллелепипед состоит из восьми точек. Для определения положения этих точек вам необходимо полностью окружить каждую кость (и ее вершины, и точки соединения костей) в скелетной структуре параллелепипедом. Я покажу, как это делать далее в этой главе, когда мы начнем использовать твердые тела в анимациях; а пока предположим, что имеем параллелепипед заданной ширины, глубины и высоты.

На рис. 7.5 центр параллелепипеда расположен в начале координат. Размеры параллелепипеда изменяются от -ширина/2, -высота/2, -глубина/2 до +ширина/2, +высота/2, +глубина/2. Используя эти значения (размеров), вы можете вычислить координаты ограничивающего параллелепипеда.

Для хранения этих восьми точек можно использовать два набора из восьми объектов D3DXVECTOR3. Первые восемь объектов будут содержать локальные координаты параллелепипеда, совсем как буфер вершин меша.

```
D3DXVECTOR3 vecLocalPoints[8];
```

Второй набор из восьми точек будет содержать глобальные координаты точек, движущихся в пространстве мира.

```
D3DXVECTOR3 vecWorldPoints[8] ;
```

Второй набор точек используется для хранения преобразованных координат, в то время как первый - для не преобразованных. Для перемещения твердого тела вам необходимо преобразовать координаты первого набора точек и сохранить результат во втором. Опять же, это очень похоже на работу с вершинами меша.

На данный момент вам необходимо хранить координаты восьми углов твердого тела в массиве векторов vecLocalPoints (используя размеры твердого тела).

```
// Width, Height, Depth=3 вещественных числа, содержащие размеры тела
// Заметьте, что все размеры заданы в метрах

// Сохранить размеры тела в вектор
D3DXVECTOR3 vecSize = D3DXVECTOR3(Width, Height, Depth);

// Сохранить половинные размеры в вектор
D3DXVECTOR3 vecHalf = vecSize * 0.5f;

// Сохранить координаты углов
vecLocalPoints[0]=D3DXVECTOR3(-vecHalf.x, vecHalf.y,-vecHalf.z);
```

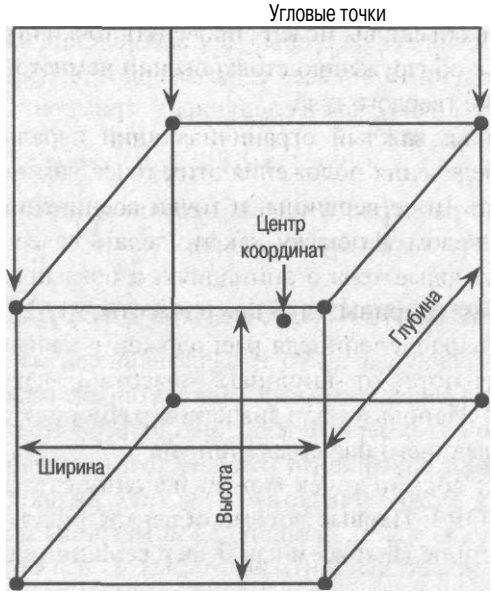


Рис. 7.5. Параллелепипед (который представляет собой твердое тело) определяется расположением восьми точек относительно его центра. Положения этих точек определяются половиной значения ширины, высоты и глубины тела

```
vecLocalPoints[1]=D3DXVECTOR3(-vecHalf.x, vecHalf.y, vecHalf.z);
vecLocalPoints[2]=D3DXVECTOR3(vecHalf.x,vecHalf'y,vecHalf.z);
vecLocalPoints[3]=D3DXVECTOR3( vecHalf.x, vecHalf.y,-vecHalf.z);
vecLocalPoints[4]=D3DXVECTOR3(-vecHalf.x,-vecHalf.y,-vecHalf.z);
vecLocalPoints[5]=D3DXVECTOR3(-vecHalf.x,-vecHalf.y,  vecHalf.z);
vecLocalPoints[6]=D3DXVECTOR3( vecHalf.x,-vecHalf.y, vecHalf.z);
vecLocalPoints[7]=D3DXVECTOR3( vecHalf.x,-vecHalf.y,-vecHalf.z);
```

Замечание. *Из комментариев кодов вы могли заметить, что в качестве единиц измерения я использую метры (в противоположность использованию безразмерных величин, возможно применяемых вами). Этот факт не должен вас беспокоить, потому что все вычисления производятся независимо от единиц измерения.*

Размеры параллелепипеда влияют не только на его объем, но и на массу. О да, твердое тело имеет массу, которая влияет на движение. К объектам с большей массой необходимо приложить большую силу, чтобы сдвинуть их, в то время как к объектам с меньшей массой — меньшую. Я объясню использование массы в вычислениях немного позже; а пока я хочу показать вам, как определять массу объекта.

Вы можете использовать любой метод для определения массы объекта, но, для упрощения, я использовал его размеры. Для вычисления массы параллелепипеда я использовал произведения длин всех его размерностей (хранимых в векторе vecSize из предыдущего кусочка), как в следующем примере:

```
float Mass = vecSize.x * vecSize.y * vecSize.z;
```

После вычисления размеров и массы параллелепипеда, вы можете располагать и ориентировать его в трехмерном мире.

Расположение и ориентирование твердых тел

После создания ограничивающего параллелепипеда из твердого тела пришло время расположить его в мире. Расположение тела определяется при помощи вектора, представляющего собой трехмерные координаты твердого тела. Как и у меша, эти координаты представляют собой центр твердого тела - начало координат.

Вращение объекта можно представить в трех видах: набора углов Эйлера[1] (значения вращения по x, y и z), вращательной матрицы преобразования или кватерниона. Хотя это и может заставить некоторых из вас съежиться, но я выбираю кватернионы для представления вращения тела. Основная причина заключается в том, что кватернионы численно устойчивы и с ними легче работать, чем с остальными методами.

Для тех из вас, кого не устраивает использование кватернионов, позвольте мне бегло рассказать о принципе их работы. Кватернион (или, если быть точным, единичный кватернион) является набором из четырех значений, определяющих вектор и скаляр. Компоненты вектора определяются как x, y, z и скаляр w. Тройка x, y, z может быть обозначена v; w может быть обозначен как s. Итак, существует два способа задания кватерниона

```
q = [s, v]
q = [w, [x, y, z]]
```

В Direct3D кватернион хранится в объекте D3DXQUATERNION, который использует обозначения x, y, z, w.

```
typedef struct D3DXQUATERNION {
  FLOAT x;
  FLOAT y;
  FLOAT z;
  FLOAT w;
} D3DXQUATERNION;
```

1. Угол Эйлера - угол между плоскостями параллелепипеда и мировым центром координат. - *Примеч. науч. ред.*

Как вы можете видеть на рис. 7.6, компоненты x, y, z определяют направляющий вектор. Этот направляющий вектор, v, представляет собой ось вращения.

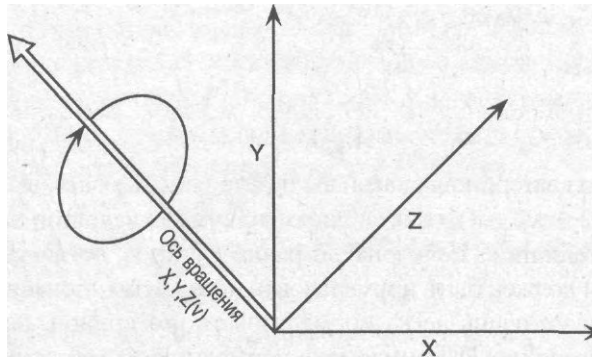


Рис. 7.6. Векторный компонент (v = x, y, z) кватерниона определяет ось вращения

Угол вращения, заданный в радианах, хранящийся в компонентах кватерниона, (w, x, y, z) вычисляется из следующего соотношения:

```
q = [w = cos(q/2), xyz = sin(q/2)xyz]
```

По-русски, все вычисления разделяются на вычисление компоненты w, содержащей косинус угла вращения (q деленное на два) и единичного вектора x, y, z, масштабированного на синус угла вращения (q деленное на два). В виде кода это будет выглядеть так:

```
q = [cos(Angle/2.0f), normalized(x,y,z)*sin(Angle/2.0f)]
```

Говоря другими словами, предположим, вы хотите создать кватернион, представляющий вращение на 45 градусов (0.785 радиан) вокруг оси y. Вы создаете вектор, сонаправленный с положительной осью y, и устанавливаете его амплитуду равной синусу половины угла. После этого вы устанавливаете компоненту w в косинус половины угла.

```
// Экземпляр используемого кватерниона
D3DXQUATERNION quatRotation;

// Создать вектор, представляющий ось вращения
D3DXVECTOR3 vecAxis = D3DXVECTOR3(0.0f, 1.0f, 0.0f);

// Нормализовать его, чтобы установить амплитуду
D3DXVec3Normalize(&vecAxis, &vecAxis);
```

```
// Масштабировать вектор на синус половины угла
vecAxis *= (float)sin(0.785 / 2.0f);

// Сохранить вектор в кватернионе
quatRotation.x = vecAxis.x;
quatRotation.y = vecAxis.y;
quatRotation.z = vecAxis.z;

// Вычислить компоненту w, используя косинус половины угла
quatRotation.w = (float)cos(0.785f / 2.0f);
```

Для единичного кватерниона, какой вы будете использовать, выполняется соотношение (x2 + y2 + z2 + w2 = 1), означающее, что сумма квадратов всех составляющих должна равняться единице. Если она не равна единице, тогда кватернион не единичной длины, и он должен быть нормализован перед использованием. При использовании Direct3D можно очень легко нормализовать кватернион, используя функцию D3DXQuaternionNormalize. Например, для нормализации кватерниона, хранящегося в quatOrientation, вы можете использовать следующий код:

```
D3DXQuaternionNormalize(&quatOrientation, &quatOrientation);
```

Ну и куда все это нас ведет? После определения оси и угла поворота вы можете использовать кватернион для преобразования точек твердого тела. Все правильно; кватернион занимает место повседневно используемых матриц преобразования и углов Эйлера! Ну, в каком то смысле.

Т. к. фактически Direct3D не работает с преобразованиями кватернионов (пока!), вам необходимо привести кватернион к вращательной матрице преобразование, которую может использовать Direct3D. Вы можете выполнить это преобразования при помощи функции D3DXMatrixRotationQuaternion, как показано тут. (Заметьте, что я транспонирую результирующую матрицу, потому что кватернионы используют правосторонние, а я в этой книге левосторонние преобразования.)

```
// quatOrientation = кватернион с установленными вращательными
значениями
D3DXMATRIX matOrientation;
D3DXMatrixRotationQuaternion(&matOrientation, \
          &quatOrientation);

// Конвертировать преобразование в левостороннее
D3DXMatrixTranspose(&matOrientation, &matOrientation);
```

Зачем я раскрывал вам секреты единичных кватернионов, если, в конце концов, мы будем использовать вращательные матрицы преобразования? На самом деле матрица преобразования является вторичным объектом, используемым для преобразования точек ограничивающего параллелепипеда вашего твердого тела

(и еще для нескольких вещей). Под словом вторичная я подразумеваю то, что хотя ориентация тела и хранится в кватернионе, все преобразования выполняются с помощью матрицы.

Вот мы и там, откуда начали, только теперь вы знаете, как работают кватернионы, и поэтому будете использовать их для представления ориентации твердых тел. После вычисления кватерниона, используемого для ориентирования твердого тела и сохранения глобальных координат в векторе положения, вы можете преобразовать локальные точки вашего твердого тела в координаты трехмерного мира.

```
// vecLocalPoints[] = массив точек тела в локальном пространстве
// vecWorldPoints[] = массив точек тела в глобальном пространстве
// quatOrientation = кватернион, содержащий вращение тела
// vecPosition = положение твердого тела

// Создать матрицу преобразования из кватерниона
D3DXMATRIX matOrientation;
D3DXMatrixRotationQuaternion(&matOrientation, \
          &quatOrientation);

// Преобразовать все восемь точек тела
for(DWORD i=0;i<8;i++) {

  // Ориентировать, используя матрицу преобразования
  D3DXVec3Transform(&vecWorldPoints[i], \
          &vecLocalPoints[i], \
          &matOrientation);

// Переместить точки, используя вектор
vecWorldPoints[i] += vecPosition;
}
```

Теперь каждая точка корректно ориентирована в мировом пространстве в соответствии с заданными трехмерными координатами и значениями вращения. Все это хорошо, но на самом деле ничего не происходит - тело остается на месте. Вам необходимо начать трясти этого щенка и заставить тело двигаться!

Обработка движения твердых тел

Движение твердого тела имеет два вида - линейное движение и угловое вращение. Линейное движение это перемещение тела вдоль прямой линии в любом направлении, а угловое вращение это вращение твердого тела вдоль осей. Просто, не правда ли?

Все движения являются результатом действия сил на твердое тело. Если вы прикладываете силу к телу, то оно в результате этого линейно движется и/или вращается. Линейное движение легко вычисляется; просто перемещайте тело

в направлении действия силы. Вращение чем-то похоже - вы прикладываете силу, в результате чего тело вращается. Направление и величина вращения зависят от точки приложения силы.

Давайте рассмотрим каждый тип движения более подробно.

Движение твердых тел

Линейное движение является результатом того, что объект толкнули или потянули в одном направлении. По мере приложения различных сил может меняться направление и скорость движения. Все зависит от силы, приложенной к телу. Для упрощения, все силы, действующие на тело, сводятся к одной, с которой вы и работаете.

Сами силы хранятся в виде векторов (D3DXVECTOR3), которые определяют направление и величину прикладываемой силы (амплитуду). Для наших целей, сила представляет собой значение ускорения, прикладываемого к телу массой 1. А что же с объектами, масса которых не равна 1, или с теми силами, которые ускоряют объекты независимо от их масс? Здесь вам поможет второй закон Ньютона.

```
F = ma
```

Второй закон Ньютона связывает величину силы (F), которую вы должны приложить к телу массой (m) для достижения им заданного ускорения (a). Например, вы хотите определить гравитацию, как силу, ускоряющую все объекты на 9.8 м/с в отрицательном направлении оси y, независимо от их массы. Используя второй закон Ньютона, умножим массу (т) на ускорение (a) для получения силы. Таким образом, для определения гравитации, используя массу объекта, хранимую в переменной Mass, вы можете использовать следующий код:

```
D3DXVECTOR3 vecGravity = D3DXVECTOR3(0.0f, -9.8f, 0.0f) * Mass;
```

А как насчет тех сил, которые ускоряют объекты без компенсирования их масс? Например, вы можете объявить следующую силу, ускоряющую объект массой 1 на 5.0 м/с^2 в положительном направлении оси x:

```
D3DXVECTOR3 vecImplied = D3DXVECTOR3(5.0f, 0.0f, 0.0f);
```

Если вы попробуете приложить ту же силу к объекту массой 2, вы получите ускорение 2.5 м/с^2 в положительном направлении x. Как я определил 2.5 м/с^2? Просто - перевернул формулу F=ma:

```
a = F/m
```

Для расчета ускорения, накладываемого на скорость объекта, вам необходимо разделить вектор силы на массу объекта. Т. к. сила была 5.0 м/с , а масса была 2, тогда ускорение будет a = 5/2.

Подытожим: все прикладываемые силы представляют собой ускорение объекта единичной массы. Если вы хотите убедиться, что ускорение не зависит от массы объекта, умножьте вектор силы на массу объекта.

Двигаясь далее в примере, теперь вы имеете два вектора сил (гравитации и прикладываемой силы), которые вы хотите использовать для перемещения объекта.

```
D3DXVECTOR3 vecGravity = D3DXVECTOR3(0.0f, -9.8f, 0.0f) * Mass;
D3DXVECTOR3 vecImplied = D3DXVECTOR3(5.0f, 0.0f, 0.0f);
```

Прежде чем вы наложите эти две силы на любое тело, соедините их в одну.

```
D3DXVECTOR3 vecForce = vecGravity + vecImplied;
```

Вы хотите работать с этой комбинированной силой? Далее в этой главе вы прочитаете о разнообразных силах и их вычислении. А пока что давайте предполагать, что мы преодолели все трудности и объединили силы в один вектор.

Как я замечал ранее, приложение силы к любой точке тела заставит его двигаться. На самом деле вам не нужно знать точку приложения силы, потому что тело в любом случае будет двигаться в направлении действия силы (или замедлятся, если сила противодействует движению).

Итак, используя две силы, объединенные в одну, вы можете линейно двигать твердое тело, прикладывая объединенную силу к вектору положения объекта. Предположим, что положение твердого объекта хранится в векторе vecPosition.

```
// Добавить вектор силы к вектору положения
yecPosition += vecForce;
```

Я знаю, что некоторые из вас собираются остановить меня здесь. А что же с ускорением, о котором я говорил ранее? После того как я обратил на это внимание, как насчет того, чтобы принять время и скорость во внимание? Я знаю, что упоминал про физику, так что пришло время прояснить ситуацию.

У объекта есть линейная скорость, которая характеризуется быстротой и направлением движения. Разнообразные силы являются причиной ускорения. Модуль силы характеризует величину ускорения, в то время как направление вектора определяет, куда она приложена. Из-за того что объекты имеют массу, для получения "настоящего" значения ускорения необходимо соответственно масштабировать силы (или скорее равнодействующую всех сил). Таким образом, объекты с большей массой ускоряются меньше, чем менее массивные. Помните F=ma и a=F/m?

Возвращаясь к вектору силы тяжести, вы можете увидеть, что требуется ускорение 9.8 м/с , независимо от массы объекта. Это означает, что масштабирование силы на массу объекта, в соответствии с формулой a=F/m, так изменит силу тяжести, что ускорение не будет равным 9.8 м/с . Что же касается прикладываемой силы (5.0 м/с), она также масштабируется в соответствии с a=F/m, так что если объект будет иметь массу отличную от 1, ускорение также будет отличаться от 5.0 м/с^2.

Итак, возвращаясь к проблеме приложения сил к твердому телу, вы делите равнодействующую силу на массу и добавляете результат к линейной скорости твердого тела. Подождите - я забыл использовать время! Вы не просто масштабируете силу на массу, но и умножаете результирующий вектор (ускорения) на время, в которое вычисляется скорость. После этого можно применять эту линейную скорость, вычисленную с использованием времени, к телу для его перемещения (или замедления, в зависимости от направления прикладываемых сил).

Сохраните скорость в векторе vecVelocity и массу в вещественной переменной Mass. Ускорение не требует переменной, потому что вы непосредственно используете ее в скорости в сочетании с массой. Вам еще необходимо учитывать время (также хранимое в вещественной переменной Time), которое определяет изменение скорости за интервал времени (измеряемый в секундах), и сколько скорости применять к телу.

```
// vecVelocity = объект D3DXVECTOR3
// Mass = вещественная переменная
// vecForce = объект D3DXVECTOR3, содержащий вектор действующий силы

// Масштабировать силу (представляющую ускорение) на массу и
// добавить ее непосредственно к скорости
vecVelocity += Time * (vecForce / Mass);

// Применить скорость к перемещению
vecPosition += Time * vecVelocity;
```

Теперь резюмируем все вышесказанное. Найдите равнодействующую всех прикладываемых сил. Масштабируйте ее вектор на массу объекта, умножьте на прошедшее время и добавьте результирующий вектор к вектору скорости объекта. Умножьте скорость на то же самое время и добавьте результат к вектору положения. Вот и все линейное движение!

А теперь пришло время перейти к рассмотрению вращательного движения.

Вращение твердых тел

Как и линейное, вращательное движение имеет скорость, ускорение и импульс, определяющие направление и быстроту вращения твердого тела. Но в отличие от линейного движения, использующего линейные силы для определения направления движения, вращательные силы используют так называемый угловой момент (или, для краткости, момент) для определения вращения тела в зависимости от приложенной силы. Момент, определяемый как вектор, непосредственно влияет на угловую скорость, которая в свою очередь влияет на угловой момент.

Масса объекта влияет на количество скорости, накладываемой на фактическое движение тела, и также используется при вращении объектов. Это объявляет тему - инерция. Инерция определяет величину силы, прикладываемой для вращения объекта вокруг осей относительно точки приложения силы.

Все правильно; та же самая сила, которая являлась причиной линейного движения, является причиной вращательного. Единственное различие состоит в том, что имеет значение точка приложения силы. По мере добавления сил, действующих на тело, вам необходимо следить за точками их приложения и за изменением осей вращения тела.

Например, если толкнуть один угол тела, оно будет вращаться в одном направлении; если же толкнуть другой угол - то в другом. Как же узнать в каком направлении и как быстро будет вращаться тело? Помните, ранее в этой главе я рассказывал об использовании кватернионов? Вы будете использовать их для слежения за направлением и углом объектов! Вы просто сойдете с ума, когда увидите насколько удобно их использовать.

Предположим, вы хотите приложить силу к твердому телу - линейную силу, создающую вращательный момент. Этот момент является направляющим вектором, но вместо того чтобы указывать направление движения, фактически задает направление вращательной оси (совсем как кватернионы используют осевые векторы для ориентирования объектов). По мере того как вы будете добавлять силы, вектор момента может менять направление, таким образом меняя оси. Это замечательно работает, потому что результирующая величина момента (результирующего момента), влияющего на тело, является простой комбинацией всех моментов, совсем как с линейными силами.

Единственным возникающим вопросом является вопрос: а как преобразовать силу в момент? Посмотрите на рис. 7.7, на котором изображено простое твердое тело. Стрелочкой показана линейная сила, приложенная к точке твердого тела.

Конечно же, твердое тело, изображенное на рис. 7.7, будет двигаться в направлении действия приложенной силы, а что же с вращением? Тело определенно будет вращаться, потому что сила приложена не к центральной точке тела.

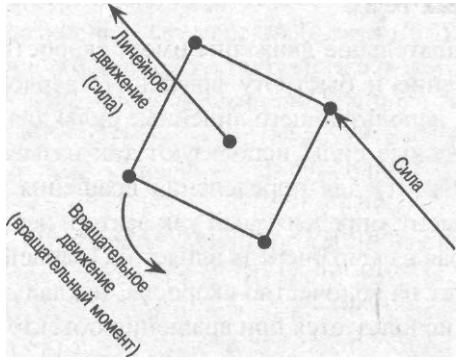


Рис. 7.7. Приложенная сила влияет не только на линейное, но и на вращательное движение

Для расчета вектора момента вычислите векторное произведение линейного вектора силы и вектора, соединяющего центр тела с точкой приложения силы. Результирующий вектор направлен в том же направлении, что и ось вращения, прямо как в кватернионах! Посмотрите на рис. 7.8, чтобы понять, что я имею ввиду. Используя то же самое тело, что и на рис. 7.7, я считаю векторное произведение, которое определяет ось, относительно которой будет вращаться объект.

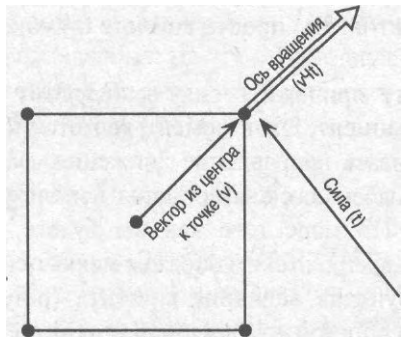


Рис. 7.8. Вектор силы и вектор, соединяющий центр с точкой приложения, используются для вычисления оси вращения путем векторного умножения

После того как вы определили ось вращения, как узнать сколько приложить силы для вращения тела? Совсем как масса объекта определяла величину прилагаемой силы для фактического движения объекта, инерция тела используется для определения вращательного момента.

На самом деле инерция состоит из трех компонентов, по одному для каждой оси вращения (x, y и z). Эти три компонента называются скалярами инерции и используются для определения еще трех значений, обычно называемых моментами инерции.

Эти значения моментов инерции являются аналогами масс тела - чем большее значение, тем меньший момент применяется к телу. Чем меньше масса твердого тела, тем больший момент накладывается на него.

Значения скаляров и моментов инерции зависят от формы и размеров твердого тела. Мы уже решили использовать ограничивающие параллелепипеды для представления твердых тел, так что нам необходимо следить только за шириной, глубиной и высотой тела. Т. к. для хранения размера твердого тела используется вектор (vecSize), можно говорить, что компонента x это ширина, y - высота, z - глубина.

Используя эти три компоненты (x, y и z) и массу объекта (Mass), можно определить скаляры инерции следующим образом:

```
// Скаляр инерции по оси X
float xs = vecSize.x * vecSize.x;

// Скаляр инерции по оси Y
float ys = vecSize.y * vecSize.y;

// Скаляр инерции по оси Z
float zs = vecSize.z * vecSize.z;
```

После этого вы можете вычислить значения моментов инерции, используя только что полученные скаляры инерции. Эти скаляры моментов инерции представляют расширение вашего тела по каждой оси, масштабированное на массу тела. Например, момент инерции для оси x является комбинацией скаляров инерции y и z, умноженной на массу тела. Вот как вычислять значения этих трех моментов инерции:

```
// Ixx = момент инерции по оси x
float Ixx = Mass * (ys + zs);

// Iyy = момент инерции по оси y
float Iyy = Mass * (xs + zs);

// Izz = момент инерции по оси z
float Izz = Mass * (xs + yz);
```

Набор этих трех моментов тензоров инерции известен как тензор момента инерции[2], или для краткости - тензор инерции. Тензор инерции составляется при помощи матрицы 3x3, которая имеет следующий вид:

2. Тензор — некое обобщенное понятие вектора. Вектор — тензор первого порядка, скаляр — тензор нулевого порядка. Величина, которая при повороте системы координат преобразуется как произведение двух компонент вектора, называется тензором второго ранга. Таким образом, тензор инерции является тензором второго порядка. - *Примеч. науч. ред.*

```
Ixx 0 0
  0 Iyy 0
  0 0 Izz
```

Хотя тензор инерции и является важным аспектом определения накладываемого момента инерции, на самом деле вам необходимо использовать обратный тензор (по причинам, о которых я расскажу позже). Обратный тензор инерции имеет следующий вид:

```
1/Ixx 0 0
  0 1/Iyy 0
  0 0 1/Izz
```

Вы же знаете, что вся эта теория инерции рассказывается вам с умыслом, не так ли? Конечно знаете! После объединения всех векторов моментов, действующих на тело, в один результирующий, вы добавляете его к угловому моменту объекта (принимая во внимание значение времени). Этот угловой момент, также как и вращательный, задан в мировых координатах. К сожалению, вы не можете использовать мировые координаты, вам необходимо преобразовать их к локальным координатам тела.

Это одна из причин, по которой я использую обратную матрицу тензора инерции. Видите ли, необходимо задавать угловую скорость в пространстве тела. (Другими словами, тело должно вращаться относительно своего центра, а не центра мира.) Для преобразования координат углового момента (из мировых координат) в координаты угловой скорости (которые в пространстве тела), учитывая момент инерции, вам необходимо умножить обратный тензор инерции на положение твердого тела. После этого умножьте полученную матрицу на транспонированное положение твердого тела, получая таким образом преобразование, учитывающее преобразования пространств тела и мира и моментов инерции. Используя полученное преобразование вы можете перевести вектор углового момента в угловую скорость.

После этого вы используете полученную угловую скорость для вычисления вращения, которое прикладывается к твердому телу. Совсем как линейная скорость масштабируется по времени для вычисления конечной скорости через определенный интервал времени, угловой момент также масштабируется. Вектор, преобразованный в угловую скорость, масштабируется по времени и применяется к значениям углового вращения кватерниона (который представляет вращение тела).

Хорошо, я уже достаточно долго не приводил Никаких кодов, так что позвольте мне рассказать, как создавать обратную матрицу тензора инерции и использовать ее для вращения твердого тела.

```
// vecSize = вектор, содержащий размер ограничивающего параллелепи-
педа твердого тела
// Mass = масса тела

// Вычислить скаляры инерции
float xScalar = vecSize.x * vecSize.x;
float yScalar = vecSize.y * vecSize.y;
float zScalar = vecSize.z * vecSize.z;

// Создать матрицу и вычислить обратный тензор инерции
D3DXMATRIX matInvInertiaTensor;
D3DXMatrixIdentity(&matInvInertiaTensor);
matInvInertiaTensor._11 = 1.0f / (Mass * (yScalar + zScalar));
matInvInertiaTensor._22 = 1.0f / (Mass * (xScalar + zScalar));
matInvInertiaTensor._33 = 1.0f / (Mass * (xScalar + yScalar));
```

Теперь предположим, что у вас есть вектор, содержащий трехмерные координаты тела и кватернион, представляющий его ориентацию. Сперва необходимо приложить силу к одному из углов. Давайте приложим вектор силы (10, 0, 40) к пятой точке твердого тела. Необходимо сразу заметить, что координаты пятой точки являются мировыми, так что используйте вектор положения из массива vecWorldPoints в следующих вычислениях.

```
// vecForce = вектор прикладываемой силы (в данном случае 10,0,40)
// vecMomentum = угловой момент тела
// quatOrientation = кватернион, содержащий ориентацию объекта
// matInvInertiaTensor = обратная матрица тензора инерции

// Получить координаты точки приложения силы (точка 5)
D3DXVECTOR3 vecPos = vecWorldPoints[5];

// Вычислить вектор их центра тела к точке приложения
D3DXVECTOR3 vecPtoC = vecPos - vecPosition;

// Вычислить векторное произведение вектора силы и vecPtoC.
// Результирующий вектор является моментом.
D3DXVECTOR3 vecTorque;
D3DXVec3Cross(&vecTorque, &vecPtoC, &vecForce);

// Добавить момент к угловому моменту
vecMomentum += vecTorque;

// Создать вращательную матрицу преобразования из кватерниона
D3DXMATRIX matOrientation;
D3DXMatrixRotationQuaternion(&matOrientation, \
          &quatOrientation);

// Транспонировать преобразование, чтобы сделать его левосторонним
D3DXMatrixTranspose(&matOrientation, &matOrientation);

// Создать матрицу, переводящую из мирового пространства в
// пространство тела. Используется для вычисления угловой скорости
```

```
D3DXMATRIX matConversion, matTransposedOrientation;
D3DXMatrixTranspose(&matTransposedOrientation,&matOrientation);
matConversion = matOrientation * \
                matInvInertiaTensor * \
                matTransposedOrientation;
```

Теперь вы имеете преобразование, используемое для перевода углового момента в угловую скорость. Вы можете наложить это преобразование на момент для получения скорости:

```
// Использовать матрицу преобразования для перевода момента в скорость
D3DXVec3TransformCoord(&vecAngularVelocity, \
        &vecMomentom, &matConversion);
```

Возвращаясь к работе со временем в предыдущих вычислениях, вам необходимо умножить вращательный момент, накладываемый на момент, на количество прошедшего времени.

```
vecMomentum += (Time * vecTorque);
```

Используя только что посчитанный момент, вычислите угловую скорость, используя виденный вами длинный кусок кода. Имея угловую скорость, вы можете вычислить ориентацию, также масштабированную по времени.

```
// Масштабировать угловую скорость на величину прошедшего времени
D3DXVECTOR3 vecVelocity = Time * vecAngularVelocity;

// Применить скорость к ориентации
quatOrientation.w  -= 0.5f *
    quatOrientation.x * vecVelocity.x +
    quatOrientation.y  * vecVelocity.y +
    quatOrientation.z * vecVelocity.z);
quatOrientation.x += 0.5f *
    quatOrientation.w * vecVelocity.x -
    quatOrientation.z  * vecVelocity.y +
    quatOrientation.y * vecVelocity.z);
quatOrientation.y  += 0.5f *
    quatOrientation.z  * vecVelocity.x +
    quatOrientation.w  * vecVelocity.y -
    quatOrientation.x * vecVelocity.z);
quatOrientation.z  += 0.5f *
    quatOrientation.x * vecVelocity.y -
    quatOrientation.y  * vecVelocity.x +
    quatOrientation.w * vecVelocity.z);
```

Я вычисляю ориентацию, учитывая скорость, основанную на времени. Помните, что скорость была посчитана из момента, а он в свою очередь из вращательного момента. Вращательный момент содержал ось вращения, совсем как кватернион. Обычно вы просто перемножаете скорость и кватернион, после чего разделяете результат на такие части:

```
q1 = 1|2 vt q
```

Наконец, после приложения силы к твердому телу вы можете вычислить угол и ось вращения. Теперь вы знаете достаточно, чтобы двигать и вращать твердые тела!

Хорошо, я признаю, что немного быстро и в вольном стиле излагал материал в данном разделе. Но у меня были веские причины - я не хотел загружать вас множеством формул и вычислений. Раздел физики твердого тела существует уже давно, и по нему написано множество литературы. Я думаю, что немногие используют физику твердого тела из-за сложности ее математики. Я хочу, чтобы любой мог понять основы динамики твердого тела, не углубляясь в математику. По этой причине я пропустил большинство формул, которые могли бы пригодиться знающим читателям. Как я уже замечал, этот раздел существует долгое время, за которое было написано несколько замечательных книг и статей о нем. Особенно полезен набор статей Chris Hecker'a Behind the Screen series on physics. Вы можете найти эти статьи на домашней странице Chris'a, http://www.d6.com/users/checker.

Теперь идет самая забавная часть - создание набора сил, действующих на твердое тело и заставляющих его двигаться.

Использование сил для создания движения

Теперь вы имеете возможность следить за движением и вращением твердого тела. Они появляются, когда сторонние силы начинают действовать на тело. Для упрощения будем полагать, что возможны только такие силы: приложенная к телу, сопротивление воздуха, тяжести и пружина.

Приложенная сила — это такая сила, которая действует непосредственно, как, например, сила взрыва, толкание персонажем объекта или любой другой тип силы, не относящийся к перечисленным ранее. Как вы можете представить, сила тяжести притягивает объект вниз (или вверх, если захотите), в то время как сопротивление воздуха замедляет движение твердого тела. Пружинная сила (сила сжатия пружины) используется для присоединения твердых тел друг к другу или какой-нибудь точке трехмерного мира.

Как я замечал в последних двух разделах, для хранения сил используются векторы. Каждая сила определяется направлением приложения силы и величиной прилагаемой силы. Величина силы хранится в виде ускорения (в метрах в секунду[3] для объекта массой 1). Для углового вращения это значение измеряется в радианах в секунду. Длина вектора силы определяет величину силы. Так, например, для силы тяжести, которая ускоряет объект приблизительно на 9.8 м/с^2 в отрицательном направлении оси y (вниз), принимая во внимание массу, вы можете создать вектор:

```
D3DXVECTOR3 vecGravity = D3DXVECTOR3(0.0f, -9.8f, 0.0) * Mass;
```

А как насчет приложенной силы, ускоряющей тело (массой 1) на 10 метров в секунду в положительном направлении оси x?

```
D3DXVECTOR3 vecApplied = D3DXVECTOR3(10.0f, 0.0f, 0.0f);
```

А как же вычисляется сопротивление воздуха? Вы определяете сопротивление воздуха не как остальные силы; вместо этого, она создается из другой силы. Я имею ввиду, что сопротивление воздуха противодействующая сила, которая вычисляется умножением скорости тела на небольшое число (отрицательное число, если быть точным). Для моделирования сопротивления воздуха в вашей системе твердых тел вы можете создать два вектора, представляющие собой противодействующие силы. Эти силы используются для замедления движения, вызванного линейной скоростью и угловым моментом.

Положив, что скорость вашего твердого тела хранится в векторе vecVelocity, а угловой момент в vecMomentum, вы можете создать два вектора (силы и момента) для дальнейшего использования.

```
D3DXVECTOR3 vecLinearDamping = vecVelocity * LinearDamping;
D3DXVECTOR3 vecAngularDamping = vecMomentum * AngularDamping;
```

Вещественные переменные LinearDamping и AngularDamping отрицательны. Чем больше значения этих переменных[4], тем большее сопротивление воздуха действует на тело и тем больше оно замедляется. Обычно я использую значения -0.5 и -0.4 для LinearDamping и AngularDamping соответственно.

3. Автор подразумевает, но явно не показывает, что прикладываемая сила определяет ускорение в метрах в секунду за секунду. - *Примеч. науч. ред.*

4. Очевидно — больше по модулю. - *Примеч. науч. ред.*

После вычисления векторов, которые представляют силы, противодействующие линейному и вращательному движению, вы можете применять их, как делали это ранее. На самом деле вы можете просто добавить векторы vecLinearDamping и vecAngularDamping к результирующей силе и суммарному моменту.

Хмм, знаете что? Каким то образом я умудрился пропустить объяснение пружинных сил. Это была не ошибка, просто я хотел отложить их рассмотрение до этого момента, чтобы я мог лучше объяснить их. Пружины соединяют твердые тела между собой, гарантируя, что тела, представляющие собой кости персонажа, соединены между собой в правильных точках.

Соединение твердых тел с помощью пружин

Моделирование движения твердого тела и приложенных к нему сил является простой задачей, если вы понимаете основы. Когда же у вас соединяется множество тел, все усложняется. Каждое изменение ориентации твердого тела может вызвать изменения во множестве других тел. Если соединено множество твердых тел, обработка одного перемещения может вызвать обработку еще множества.

Возвратимся к теме. Эти соединенные между собой твердые тела представляют собой кукольный персонаж. Вы знаете, как все устроено - кости кисти присоединены к руке, кость руки присоединена к …. Ну, вы поняли идею. Твердые тела соединяются между собой так же, как и кости в скелетном меше присоединяются к родительским костям (конечно, за исключением корневой кости).

В то время как скелетная структура соединяет кости, используя иерархию фреймов, твердые тела не могут позволить себе такой роскоши; тела полностью свободны и при моделировании перемещаются по всему миру. Необходимо что-то, что держало бы их вместе, и этим чем-то являются пружины.

Как вы можете видеть на рис. 7.9, пружины создаются для каждой точки соединения костей в исходной скелетной структуре.

Когда вы начнете перемещать твердые тела, вам будет необходимо вычислить силы, необходимые каждой пружине для сохранения начальной формы структуры тела. Пружины просто создают силы, двигающие каждый объект в заданном направлении. Но в отличие от упомянутых ранее сил, пружинам необходимо соединить тело не немедленно, а за определенный интервал времени. Это можно реализовать, непосредственно изменяя положение и угловой момент каждого тела.

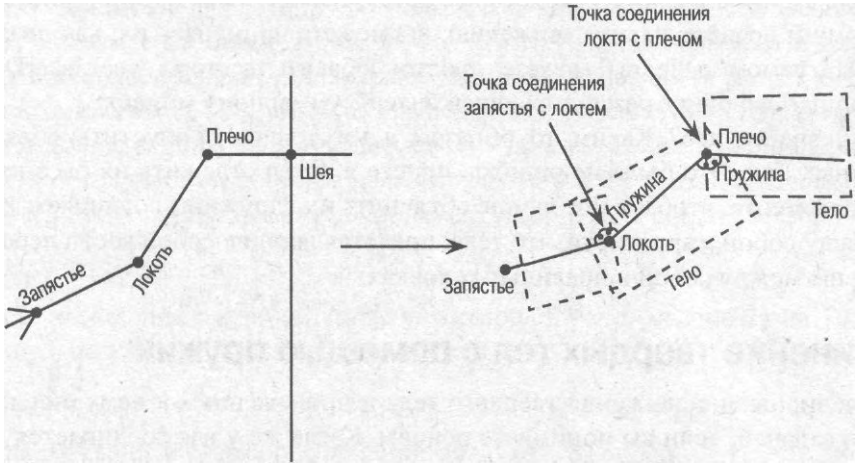


Рис. 7.9. Пружины соединяют твердые тела в точках сочленения костей

Сначала о главном, вам необходимо узнать точки соединения костей. Когда вы создаете ограничивающие параллелепипеды, вам необходимо добавить точку, которая определяет место соединения параллелепипеда с параллелепипедом родительской кости. Вам также необходимо добавить точку, представляющую собой смещение от центра родительской кости до точки соединения. Эти две точки показаны на рис. 7.10.

Помните, ранее в этой главе, вы определили восемь векторов, представляющих собой углы твердого тела? Было два набора по восемь вершин, один для определения локальных координат, а другой для определения этих же самых точек в мировых координатах. Теперь вам необходимо добавить еще две точки в список.

Первая точка является смещением от центра кости до точки соединения с родительской костью. Эта точка, имеющая название точка смещения соединения (или просто смещение соединения), представляет собой один конец создаваемой пружины, которая будет соединять твердые тела после их перемещений.

Вторая точка является смещением от центра родительской кости до точки соединения с родительской костью. Эта вторая точка, называемая точка родительского смещения (или просто родительское смещение), использует преобразования родительской кости для ориентирования, противодействуя преобразованию кости (как вы видели ранее). Вторая точка представляет собой другой конец создаваемой пружины для соединения костей.

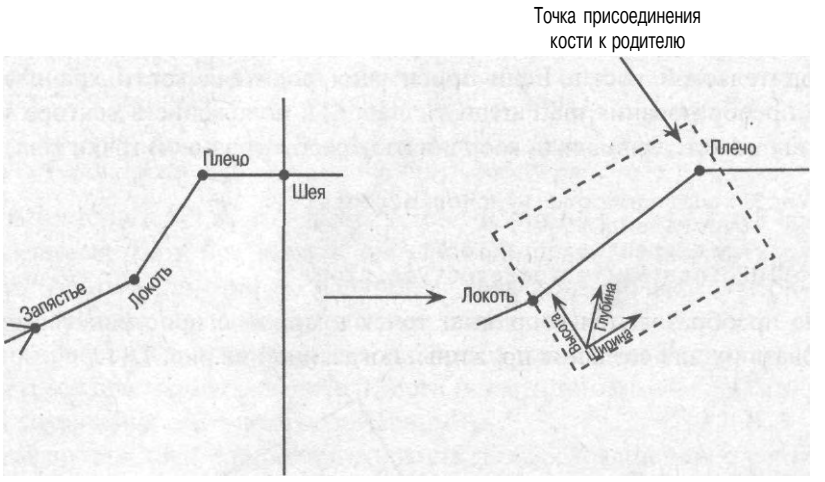


Рис. 7.10. Каждая кость определяется размерами ограничивающего параллелепипеда и точкой соединения с родительской костью.

Т. к. вторая добавляемая точка использует преобразование родительской кости, не обязательно хранить ее в наборе векторов локального пространства; просто храните ее в наборе векторов преобразования. Это дает вам возможность определить два новых набора векторов.

```
D3DXVECTOR3 vecLocalPoints[9];
D3DXVECTOR3 vecWorldPoints[10] ;
```

Вы уже видели, как сохранять координаты угловых точек с помощью векторов, так что я пропущу эту часть и покажу вам, как сохранять значения смещения соединения и родительского смещения. Восьмая точка будет представлять координаты смещения соединения, а точка девять — родительского смещения.

Вектор смещения соединения хранится в объекте D3DXVECTOR3, имеющем название vecJointOffset, в то время как вектор родительского смещения хранится в vecParentOffset. Первое, что необходимо сделать, это сохранить вектор смещения соединения в соответствующем векторе локального пространства.

```
vecLocalPoints[8]  = vecJointOffset;
```

Теперь вы можете преобразовывать девять точек из локального пространства в мировое, как вы делали это ранее. После чего, точка 8 будет содержать координаты в мировом пространстве точки присоединения кости к родителю.

Что же касается вектора родительского смещения, вам необходимо поместить его в один из векторов vecWorldPoints. Чтобы сделать это, используйте преобразования родительской кости. Если ориентация родителя кости хранится в виде матрицы преобразования matParentOrientation, а положение в векторе vecParentPosition, вы можете вычислить координаты преобразованной точки так:

```
D3DXVec3TransformCoord(&vecWorldPoints[9],   \
        &vecParentOffset, \
        &matParentOrientation);
vecWorldPoints[9] += vecParentPosition;
```

После преобразования координат точек в мировое пространство вы можете использовать их для создания пружины, показанной на рис. 7.11.

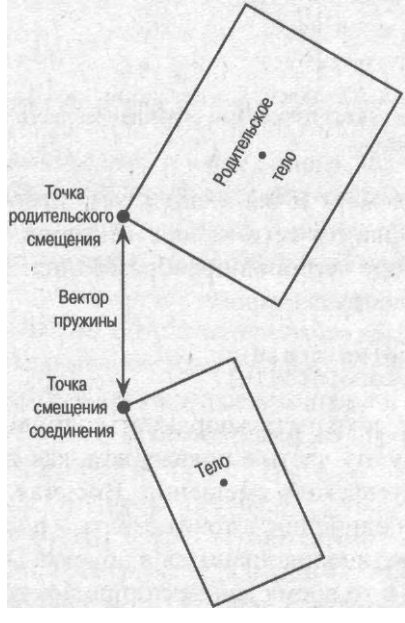


Рис. 7.11. Вы создаете вектор пружины, соединяя точки смещения соединения и родительского смещения. Обе точки задаются в мировых координатах

```
// vecPosition = положение тела

// Получить положение смещения соединения в мировых координатах
D3DXVECTOR3 vecBonePos = vecWorldPoints[8];

// Получить вектор из точки в центр тела
```

```
// Он используется для вычисления момента
D3DXVECTOR3 vecPtoC = vecPosition - vecBonePos;

// Получить положение родительского смещения в мировых координатах
D3DXVECTOR3 vecParentPos = vecWorldPoints[9];

// Вычислить вектор пружины из точек соединения и родительской
D3DXVECTOR3 vecSpring = vecBonePos - vecParentPos;
```

Теперь вектор vecSpring содержит разницу координат, которую вы используете для перемещения кости для соединения ее с родителем. По аналогии с реальным человеком, при соединении с родителем двигаются только дочерние кости. (В качестве примера: ваша рука повторяет движение груди.) Таким образом, при движении корневой кости (представляющей грудь персонажа или центр масс) будут двигаться и все присоединенные к ней кости (вместо того, чтобы двигать корневую кость для сохранения соединения с дочерними).

Т. к. вы просто хотите переместить кость для воссоединения с родительской, вы можете просто добавить вектор vecSpring непосредственно к положению вашего тела (игнорируя таким образом константы пружины). Также вы можете вычислить векторное произведение разницы положения точки и положения смещения вращения и пружины для получения углового момента кости, таким образом повернув ее, чтобы убедиться, что существуют точки касания.

```
vecPosition += vecSpring;
D3DXVECTOR3 vecCross;
D3DXVec3Cross(&vecCross, &vecBtoC, &vecSpring);
vecAngularMoment += vecCross;
```

После изменения момента вы можете преобразовать его, используя обратную матрицу тензора инерции кости, и сохранить результат в векторе угловой скорости.

```
D3DXVECTOR3 vec
D3DXVec3TransformCoord(&vecAngularVelocity, \
        &vecAngularMomentum, \
        &matInvWorldInertiaTensor);
```

Со всеми этими разговорами о векторах смещения, соединения и родительского смещения я забыл упомянуть, откуда получать их значения. При вычислении размера ограничивающего параллелепипеда вам необходимо принимать во внимание координаты кости и ее родителя в мировом пространстве.

Вектор из центра ограничивающего параллелепипеда в координаты кости является смещением соединения. Смещение от координат родительской кости (которые изменяются при помощи обратной матрицы преобразования кости) до координат кости дает вектор родительского смещения. Вы узнаете об этом более подробно чуть позже. Пока что вы научились накладывать силы на линейную скорость и угловой момент и заставлять кости воссоединяться. Что далее? Обработка столкновений, вот что!

Обеспечение обнаружения столкновений и ответной реакции

Т. к. твердые тела состоят из твердых объектов, они никогда не меняют форму и не пересекают пространство других объектов. Столкновение с другими объектами означает, что твердые тела соответствующим образом взаимодействуют, в результате чего они меняют движение и вращение. Это означает, что вам необходимо найти способ обнаружения столкновений твердых тел и соответствующим образом их обрабатывать. Давайте продолжим, подробнее рассмотрев обнаружение столкновений.

Проверка на столкновения

Помните, я упоминал, что использование восьми точек для твердого тела полезно не только для экономии памяти? Если вы еще не догадались, эти точки помогают обнаруживать столкновения! Т. к. ваши твердые тела состоят из восьми точек, вам просто необходимо проверить, попадают ли они в другой объект. Если хотя бы одна из точек попадает в другой объект, вы можете скорректировать движение твердого тела в соответствии с отклонением от сталкиваемого объекта.

Конечно же существует множество способов обнаружения столкновений, обычно использующих проверки грани и точки, точки и точки, грани и грани, но для увеличения скорости обработки я все упрощаю, что означает использование проверки точки и плоскости и точки и сферы.

Другими словами, для каждой из восьми точек твердого тела вы быстро проверяете, попадает ли точка на плоскость (используя скалярное произведение) или в сферу (используя проверку расстояния). Каждая из этих проверок тривиальна: для плоскости вы берете координаты каждой точки и скалярно умножаете их на нормаль плоскости, используя параметр расстояния.

```
// vecPos = D3DXVECTOR3 содержащий координаты точки
// Plane = D3DXPLANE содержащий данные плоскости (нормаль
и расстояние)
                                                    I
// Создать вектор, используя данные плоскости
D3DXVECTOR3 vecPlane = D3DXVECTOR3(Plane.a, Plane.b, Plane.c);
```

```
// Выполнить скалярное умножения для получения расстояния от точки
до плоскости
float Dist = D3DXVec3Dot(&vecPos, &vecPlane) + Plane.d;
```

Т. к. параметр d структуры D3DXPLANE представляет собой расстояние от плоскости до. начала координат, вы можете определить расстояние от точки до плоскости, прибавив это расстояние к скалярному произведению координат точки и нормали плоскости, как я вам показал. Если значение расстояния меньше или равно 0, то точка лежит на плоскости.

На самом деле, точка сталкивается с плоскостью, если расстояние 0, и пересекает ее, если расстояние меньше 0. Какая разница между пересечением и столкновением? При работе с твердыми телами вы имеете дело с двумя типами столкновений - столкновение касания и проникновения. Столкновение касания происходит, когда один объект просто касается (в допустимых пределах) другого. Вследствие этого, один из них отскакивает от другого. Столкновение проникновения происходит, когда один объект проникает в другой, не отскакивая от него.

Мы рассматриваем столкновения касания, т. к. обрабатываем движение, полученное в результате отскакивания объектов друг от друга. Для тел, которые проникают в другие объекты, вы можете либо отследить момент, в который тела соприкоснулись (но не проникли друг в друга), либо просто вытолкнуть тело из пространства другого объекта и продолжать, как будто бы они просто столкнулись (а не проникли друг в друга).

Как вы можете видеть, метод отслеживания времени столкновения является более точным. К сожалению, когда тела начнут взаимодействовать между собой, все может очень сильно замедлиться из-за попыток найти точный момент столкновения. Хотя точность и желаема, но когда решающим фактором является быстродействие, она не действует. По этой причине, я покажу вам как обрабатывать столкновения, выталкивая проникающий объект из пространства объекта, с которым он сталкивается, и продолжая, как будто они никогда не проникали друг в друга.

Возвращаясь к поставленной задаче, вам просто необходимо проверять расстояние для определения столкновения точки со сферой. Однако нам подходят не все проверки, т. к. многие из них используют операцию извлечения квадратного корня. Для увеличения быстродействия вы можете пропустить эту операцию и вместо этого работать с квадратами расстояний от точки до центра сферы и ее радиусом.

Позвольте привести пример. Положение точки хранится в vecPos, а координаты сферы в vecSphere. Радиус сферы хранится в Radius. Чтобы проверить, находится ли точка в пределах радиуса сферы, вы можете использовать следующий код:

```
// vecPos = положение проверяемой точки
// vecSphere = координаты, центра сферы
// Radius = вещественное число, содержащее радиус сферы

// Получить вектор из точки в центр сферы
D3DXVECTOR3 vecDiff = vecSphere - vecPos;

// Получить квадрат расстояния от точки до сферы
float Dist = vecDiff.x * vecDiff.x +
             vecDiff.y * vecDiff.y +
             vecDiff.z * vecDiff.z;

// Проверить является ли distance <= квадрата Radius
if(Dist < (Radius * Radius)) {
  // Точка лежит в сфере!
}
```

Как показывает код, вы создаете вектор из центра сферы в подозрительную точку. Квадрат длины этого вектора сравнивается с квадратом радиуса сферы. Если длина меньше или равна радиусу, тогда точка лежит в сфере.

После определения точки проникновения в объект (сферу или плоскость) вам необходимо вытолкнуть ее из объекта и рассчитать, как точка отскочит от поверхности объекта. Для того чтобы вытолкнуть точку из объекта, вам необходимо взять его нормаль (нормаль плоскости или нормальный вектор из центра сферы в точку) и масштабировать ее в соответствии с тем, насколько далеко необходимо вытолкнуть точку.

Для плоскости, ее нормаль хранится в компонентах a, b и c структуры плоскости. Т. к. эти компоненты уже нормализованы, вам необходимо просто масштабировать вектор на значение скалярного произведения, вычисленного при определении пересечения точкой плоскости.

```
// Plane = структрура плоскости, с которой идет работа
// Dot = скалярное произведение вектора точки и нормали плоскости
// vecPosition = положение точки
D3DXVECTOR vecNormal = D3DXVECTOR3(Plane.a,
                                   Plane.b,
                                   Plane.c);

// Масштабировать нормаль на скалярное произведение
vecNormal *= Dot;

// Добавить масштабированный вектор к положению точки
vecPosition += vecNormal;
```

Для того чтобы вытолкнуть точку за пределы сферы, вам необходимо вычислить разность радиуса и расстояния от точки до центра сферы. Это означает, что вам необходимо взять квадратный корень от уже посчитанного при определении

проникновения значения расстояния. В следующем коде вы вычисляете квадратный корень расстояния и используете радиус сферы для вычисления значения, на которое необходимо масштабировать вектор из середины сферы в точку.

```
// vecDiff = вектор из точки в сферу
// Radius = радиус сферы
// Dist = квадрат расстояния от точки до центра сферы

// Нормализовать вектор
D3DXVec3Normalize(&vecDiff, &vecDiff);

// Получить настоящее значение расстояния, вычислив
// квадратный корень переменной Dist
Dist = (float)sqrt(Dist);

// Вычесть новое значение Dist из Radius, чтобы получить
// расстояние, на которое необходимо вытолкнуть точку из сферы
float ToPush = Radius - Dist;

// Масштабировать нормализованный вектор, используя ToPush
vecDiff *= ToPush;

// Добавить масштабированный вектор к положению точки
vecPosition += vecDiff;
```

Теперь, когда вы знаете, как выполнять проверку расстояний по отношению к плоскостям и сферам и можете использовать ее результаты для обнаружения столкновений объектов (а если происходит проникновение, то и выталкивания объекта), что же осталось? Ну, вам необходимо сделать так, чтобы точка отскочила от поверхности объекта.

Ответная реакция на столкновения

Как вы можете видеть, обнаружение столкновений на данный момент является тривиальной задачей, нас же больше интересует ответная реакция на столкновение. Как только вы определили, что твердое тело сталкивается с другим объектом, вам необходимо выяснить, как это столкновение влияет на движение объекта.

Также как и другие силы (тяжести, пружины) влияют на движение твердых тел, влияет и реакция на столкновения. Как и пружинная сила, реакция на столкновение мгновенно изменяет положение и вращение твердого тела.

Так и есть, Люк; пока мы проигнорируем силу, вместо этого работая с импульсом! Совсем как темная сторона силы (я надеюсь, я не зря трачу метафоры на не фанатов "Звездных войн"), импульс является разновидностью силы. Вместо того чтобы долго и нудно прикладывать силы к объекту и заставлять его двигаться в соответствии с реакцией на столкновение, импульс немедленно перемещает объект от точки столкновения.

Вычисление импульса совершенно такое же, как и вычисление прилагаемой силы. Для столкновений, вычисление включает определение скорости, на которой точка приближается к поверхности объекта столкновения, и использование нормали поверхности для вычисления значения импульса, противодействующего движению твердого тела.

У вас уже есть нормаль к поверхности, относительно которой отскакивает тело, а скорость перемещения тела является суммой линейной и угловой скоростей. Все, что остается, - это определить, какую часть этой скорости использовать для создания отталкивающей силы. Используя так называемый коэффициент возврата (restitution), вы можете масштабировать нормаль объекта на заданное значение, определяемое из скорости сталкивающейся точки, и использовать полученный вектор в качестве силы для отскока твердого тела.

Коэффициент возврата это просто скаляр, который определяет, насколько отскочит тело при столкновении с другим объектом (фактически, сколько силы теряется при столкновении).

Чем меньшее значение коэффициента используется, тем меньше тело отскакивает от объекта. Чем больше значение, тем сильнее. На самом деле, тела могут даже получать энергию при столкновениях, если коэффициент больше 1. Коэффициент возврата хранится в вещественной переменной, которую я назвал Coefficient.

Все начинает усложняться, потому что формулы, используемые для вычисления скорости при отскоке, сложны. Как я ранее замечал, сначала необходимо вычислить скорость точки, используя линейную и угловую скорости тела. Для этого вам необходимо создать два вектора - один, соединяющий центр тела и точку столкновения, и второй, представляющий линейную скорость, прибавленную к векторному произведению предыдущего вектора и угловой скорости.

```
// vecPosition = координаты твердого тела
// vecPointPos = координаты точки столкновения
// vecCollisionNormal = нормаль сталкиваемого объекта

// Получить вектор из центра тела в точку столкновения
D3DXVECTOR3 vecPtoP = vecPosition - vecPointPos;

// Получить векторное произведение vecPtoP и угловой скорости
D3DXVECTOR3 vecCross;
D3DXVec3Cross(&vecCross, &vecAngularVelocity, &vecPtoP);

// Сложить эти два вектора для получения скорости точки
D3DXVECTOR3 vecPointVelocity = vecLinearVelocity + vecCross;
```

Теперь у вас есть вектор скорости точки столкновения, хранящийся в vecPointVelocity. Этот вектор используется для вычисления импульса, который действует непосредственно на положение и угловой момент твердого тела. Для вычисления импульса необходимо сначала посчитать числитель и знаменатель импульса для масштабирования нормали сталкиваемого объекта. Чтобы следующий далее код был читаемым, я буду использовать две функции DotProduct и CrossProduct, которые напрямую работают с кодом вычисления скалярного и векторного произведения двух векторов соответственно.

```
// Вычислить числитель импульса
real ImpulseNumerator = DotProduct(vecPointVelocity, \
                        vecCollisionNormal) * \
                    -(1.0f + Coefficient);

real ImpulseDenominator = (1.0f / Mass) + \
     DotProduct(CrossProduct(matInvWorldInertiaTensor * \
     CrossProduct(vecPtoP, vecCollisionNormal), \
     vecPtoP), CollisionNormal);
```

Как вы и сами могли догадаться, вычисления ImpulseNumerator и ImpulseDenominator очень сложны. Хотя мне и хотелось бы посвятить время и место их полному объяснению, я просто не могу сделать этого. Целые статьи были написаны об их вычислении, так что я оставляю их им. Для примера, одну из таких статей, написанную Chris Hecker для журнала Game Developer, вы можете скачать на его сайте http://www.d6.com/users/checker.

А пока, просто применим эти два значения к сталкиваемой нормали для вычисления вектора импульса, используемого для отскока точки от сталкиваемого объекта.

```
D3DXVECTOR3 vecImpulse = vecCollisionNormal * \
      (ImpulseNumerator/ImpulseDenominator);
```

Теперь у вас есть вектор, который представляет собой противодействующую силу, прикладываемую к сталкиваемому телу. Вы можете добавить этот вектор скорости vecImpulse непосредственно к линейной скорости и угловому моменту твердого тела.

```
// Добавьте силы, для окончания

vecLinearVelocity += vecImpulse;
// Вычислить векторное произведения для создания вектора момента
D3DXVECTOR3 vecCross;
D3DXVec3Cross(&vecCross, &vecPtoP, &vecImpulse);
vecAngularMomentum += vecCross;
```

Однако существует одна проблема, о которой я не упомянул. Как же быть с теми точками, которые сталкиваются с одним и тем же объектом одновременно? Если вы одновременно обработаете эти точки и скорректируете их положение и вращение в соответствии со столкновениями, вы обнаружите, что тела, параллельные сталкиваемому объекту, отскочат очень ненатурально.

Для компенсирования этого факта вы должны обработать каждую точку твердого тела, следя за тем, чтобы все импульсы применялись к линейным скоростям и угловым моментам. После того как вы обработали все точки, надо найти и усреднить векторы импульсов на их количество и добавить результат к скорости и моменту.

Сохранение данных объекта столкновения

Со всеми этими разговорами об объектах столкновений, я забыл упомянуть о том, как мы будем хранить разнообразную информацию, определяющую эти объекты. Т. к. я все упрощаю и использую только плоскости и сферы, вы можете создать пару классов, которые бы содержали один объект столкновений и массив таких объектов, соответственно.

Первый объект, который содержит информацию об одном объекте столкновений, называется cCollisionObject.

```
class cCollisionObject {
  public:
   DWORD m_Type; // Тип объекта

    D3DXVECTOR3 m_vecPos; // Координаты сферы
    float m_Radius; // Радиус сферы
    D3DXPLANE m_Plane; // Значения плоскости

    cCollisionObject *m_Next; // Следующий объект в связанном списке

  public:
   cCollisionObject() { m_Next = NULL; }
   ~cCollisionObject() { delete m_Next; m_Next = NULL; }
};
```

Вы прочитаете более подробно об объекте столкновений в главе 13 "Имитирование одежды и анимация мешей мягких тел". Почему я так тяну с их объяснением, спросите вы? Это все благодаря современному чуду редактирования копий (и моему постоянно меняющемуся мнению) я собирал информацию. Однако я не позволю этому факту остановить вас от просмотра кода, так что давайте быстро пройдемся по классу.

Внутри класса cCollisionObject располагается объект D3DXPLANE, который используется для определения плоскости, в то время как вектор и вещественное значение используются для определения расположения и радиуса сферы. Фактически,

класс cCollisionObject может содержать информацию как о плоскости, так и о сфере. По этой причине в класс введена переменная m_Type, которая определяет тип объекта, содержащегося в классе, и принимает значение COLLISION_SPHERE для сферы или COLLISION_PLANE для плоскости.

Указатель m_Next указывает на следующий объект столкновения в загруженном списке объектов столкновения. Так и есть, объекты столкновения хранятся в связанном списке. По этой причине я создал конструктор и деструктор, которые ответственны за инициализацию и освобождение указателя связанного списка.

Говоря о более чем одном объекте столкновении, я хочу представить вам второй класс - cCollision.

```
class cCollision {
  public:
    DWORD m_NumObjects; // # объектов
    cCollisionObject *m_Objects; // список объектов

  public:
    cCollision();
    ~cCollision();

    void Free();
    void AddSphere(D3DXVECTOR3 *vecPos, float Radius);
    void AddPlane(D3DXPLANE *PlaneParam);
};
```

Класс cCollision содержит связанный список объектов столкновений. Вы можете добавлять объекты, используя функции AddSphere и AddPlane, или очистить связанный список, вызвав Free. Параметры функций AddSphere и AddPlane очевидны: вы задаете центр и радиус добавляемой сферы или параметры добавляемой в связанный список плоскости.

Вы можете сразу перейти к главе 13 и более детально рассмотреть информацию об объектах столкновений или можете посмотреть исходный код, содержащийся на компакт-диске книги (расположенный в файлах Collision.cpp и Collision.h директории главы 7).

А пока что, позвольте показать вам, как создавать объекты плоскости и сферы, используемые во время моделирования.

```
// Экземпляр набора объектов столкновений
cCollision Collision;

// Добавить основную плоскость в центр мира
Collision.AddPlane(&D3DXPLANE(0.0f, 1.0f, 0.0f, 0.0f));

// Добавить сферу в (0,10,40) с радиусом 20
Collision.AddSphere(&D3DXVECTOR3(0.0f, 10.0f, 40.0f), 20.0f);
```

Итак, мой друг, я верю, что вы усвоили все необходимое для создания своей собственной системы кукольной анимации! Я собираюсь быстро показать вам, как применять полученные знания при работе с набором классов, используемых в системах кукольной анимации. Убедитесь, что вы хорошо понимаете все выше изложенное, потому что дальше все пойдет очень быстро!

Создание систем кукольной анимации

После того как вы узнали секреты физики твердого тела, что остается сделать? Создать собственную систему кукольной анимации, вот что! Кукла - это не что иное, как набор связанных твердых тел, созданных из костей персонажа. Создав набор классов, содержащих твердые тела, и класс для управления их движением, вы получите собственную систему кукольной анимации!

Но я хочу рассмотреть целую систему кукольной анимации по частям, начав с определения состояние одного твердого тела.

Определение состояния твердого тела

Ранее в этой главе вы научились определять твердое тело при помощи набора векторов, содержащих положение тела, ориентацию, момент и так далее. Эти вектора определяют текущее состояние твердого тела, которое вы можете хранить в следующей структуре:

```
class cRagdollBoneState
{

  public:
   D3DXVECTOR3 m_vecPosition; // Положение
   D3DXQUATERNION m_quatOrientation; // Ориентация
   D3DXMATRIX m_matOrientation; // Ориентация

   D3DXVECTOR3 m_vecAngularMomentum; // Угловой момент

   D3DXVECTOR3 m_vecLinearVelocity; // Линейная скорость
   D3DXVECTOR3 m_vecAngularVelocity; // Угловая скорость

   // Преобразованные точки, включающие положение присоединения к
родителю и смещение от родителя до кости
   D3DXVECTOR3 m_vecPoints[10] ;

   // Обратная матрица тензора мирового момента инерции
   D3DXMATRIX m_matInvWorldInertiaTensor;
};
```

cRagdollBoneState хранит положение, ориентацию (кватернион и матрицу), линейную, угловую скорости, момент, обратный тензор инерции (в мировых координатах) и преобразованные точки. Эти точки содержат смещение от твердого тела до родительского тела и смещение от родительского тела до текущего тела.

По мере обработки вашей модели состояние костей обновляется в соответствии с действующими силами. Для хранения оставшейся информации о костях, такой как размер, масса и т. д., определим второй класс.

Хранение костей

Оставшиеся данные кости, такие как исходный фрейм кости, размер, масса, коэффициент restitution, сила, момент (просто для перечисления), хранятся в другом классе. Этот класс cRagdollBone определяется так:

```
class cRagdollBone
{
  public:
    // Фрейм, к которому присоединена кость
    D3DXFRAME_EX *m_Frame;

    // Размер ограничивающего параллелепипеда
    D3DXVECTOR3 m_vecSize;

    // Масса и 1/массу (единица деленная на массу)
    float m_Mass;

    // Значение коэффициента возврата (restitution)
    // 0 = нет отскока
    // 1 = "супер" отскок
    // 2 = получать энергию при отскоке
    float m_Coefficient;

    cRagdollBone *m_ParentBone; // указатель на родительскую кость
    // смещения присоединения к родителю и родителя к кости
    D3DXVECTOR3 m_vecJointOffset;
    D3DXVECTOR3 m_vecParentOffset;

    // Линейная сила и угловой момент
    D3DXVECTOR3 m_vecForce;
    D3DXVECTOR3 m_vecTorque;

    // Начальная ориентация кости
    D3DXQUATERNION m_quatOrientation;

    // Уровень разрешения (0-1) для сферической интерполяции
    // Он используется для возвращения костей к начальному
    // ориентированию относительно родителя
    float m_ResolutionRate;
```

```
    // Обратная матрица тензора момента инерции тела
    D3DXMATRIX m_matInvInertiaTensor;

    // Точки (в локальном пространстве), которые образуют
    // ограничивающий параллелепипед и смещение соединения с родителем
    D3DXVECTOR3 m_vecPoints[9];

    // Состояние кости
    cRagdollBoneState m_State;
};
```

Комментарии для каждого члена класса являются достаточно самоописательными, тем более вы читали о большинстве из них ранее в этой главе. Единственное, с чем вы можете быть не знакомы, это кватернион, уровень разрешения и способ использования точек и состояния кости.

Кватернион (m_quatOrientation), содержащийся в классе cRagdollBone, представляет собой относительную разность вращений из родительской кости. Он полезен для поддержания формы персонажа при симуляции. Вместо того чтобы определять границы, в которых может вращаться каждая кость по отношению к родителю, вы можете заставить восстанавливаться ориентацию каждой кости к начальной. Я покажу вам, как это делается немного позже; а пока что вернемся к классу cRagdollBone.

Вы можете видеть, что я встроил класс состояния в cRagdollBone, который представляет состояние кости во время моделирования. Также определено девять точек, представляющих локальные координаты каждой угловой точки и точку присоединения кости к родителю. Хорошо, я пропускаю множество деталей тут, но я еще вернусь к классу cRagdollBone. А сейчас я покажу вам, как набор классов костей используется классом, управляющим всеми аспектами кукольной анимации.

Создание класса управления куклой

Третий и последний класс, используемый для управления системой кукольных анимаций, большой, поэтому я буду показывать его вам по частям. Сначала у вас есть указатель на иерархию фреймов, который используется для создания твердых тел при моделировании. У вас также имеется количество костей, содержащихся в иерархии костей, и массив объектов cRagdollBone, содержащих информацию о каждой кости.

```
class cRagdoll
{
  protected:
    D3DXFRAME_EX *m_pFrame; // Корневой фрейм иерархии

    DWORD m_NumBones; // # костей
    cRagdollBone *m_Bones; // Список костей
```

Пока что класс не выглядит очень пугающе, так к чему суета? Поверьте мне, вы еще не все знаете. Далее идут защищенные функции.

```
protected:
  // Встроенная функция, вычисляющая векторное произведение
  D3DXVECTOR3 CrossProduct(D3DXVECTOR3 *v1, D3DXVECTOR3 *v2);

  // Функция, умножающая вектор на матрицу 3x3 и по желанию
  // добавляющая вектор смещения
  D3DXVECTOR3 Transform(D3DXVECTOR3 *vecSrc,
         D3DXMATRIX *matSrc,
         D3DXVECTOR3 *vecTranslate = NULL);

  // Получить ограничивающий параллелепипед кости фрейма и смещение
  // соединения
  void GetBoundingBoxSize(D3DXFRAME_EX *pFrame,
         D3DXMESHCONTAINER_EX *pMesh,
         D3DXVECTOR3 *vecSize,
         D3DXVECTOR3 *vecJointOffset);

  // Создать кость и установить ее данные
  void BuildBoneData(DWORD *BoneNum,
         D3DXFRAME_EX *Frame,
         D3DXMESHCONTAINER_EX *pMesh,
         cRagdollBone *ParentBone = NULL);

  // Установить силы тяжести, амортизации и соединения
  void SetForces(DWORD BoneNum,
         D3DXVECTOR3 *vecGravity,
         float LinearDamping,
         float AngularDamping);

  // Объединить движение кости для промежутка времени
  void Integrate(DWORD BoneNum, float Elapsed);

  // Обработать столкновения
  DWORD ProcessCollisions(DWORD BoneNum,
           cCollision *pCollision,
           D3DXMATRIX *matCollision);

  // Обработать соединения костей
  void ProcessConnections(DWORD BoneNum);

  // Преобразовать точки состояния
  void TransformPoints(DWORD BoneNum);
```

Давайте рассмотрим эти девять защищенных функций. Начнем с первых двух (CrossProduct и Transform), которые вы используете для вычисления векторного произведения и преобразования векторов, используя матрицу преобразования и дополнительный вектор перемещения. Почему просто не использовать функции

D3DX спросите вы? Ну, функции CrossProduct и Transform используют D3DX, но я хотел иметь возможность вычислять векторное произведение и преобразование векторов напрямую в других кодах, как показано тут:

```
D3DXVECTOR3 vecResult = Transform(&CrossProduct(&vec1, \
                        &vec2), \
             &matTransform);
```

Третья защищенная функция GetBoundingBoxSize используется для вычисления размера ограничивающего параллелепипеда кости. Этот параллелепипед содержит все вершины, присоединенные к кости и точки присоединения кости к родителю и потомкам (если таковые имеются). Следующая функция SetForces используется для установки начальных сил, действующих на тело: тяжести, линейной и угловой амортизации.

Далее в списке функций идет Integrate, которая обрабатывает действующие на кость силу и момент, обновляет скорости и моменты и перемещает точки твердого тела в новое положение в зависимости от движения тела. Далее идут ProcessCollisions (которая выполняет обнаружение столкновений и ответную реакцию на них), ProcessConnections (которая убеждается, что все кости соединены между собой в сочленениях) и TransformPoints (которая преобразует локальные точки в глобальные координаты, используя ориентацию и положение кости, хранящиеся в объекте состояния кости m_State).

Вы обнаружите полный исходный код класса cRagdoll, а также других определенных здесь классов, в файлах Ragdoll.cpp и Ragdoll.h компакт-диска этой книги. Рассмотренные пока функции просто дублировали сказанное ранее в этой главе, поэтому нет нужды в их объяснении. Ну, за исключением функций GetBoundingBoxSize и Integrate. Я вернусь к ним немного позже; а пока, я хочу продолжить, показав вам оставшиеся функции класса cRagdoll.

```
public:
  cRagdoll();
  ~cRagdoll();

  //создать куклу из предоставленного указателя иерархии фреймов
  BOOL Create(D3DXFRAME_EX *Frame,
       D3DXMESHCONTAINER_EX *Mesh,
       D3DXMATRIX *matInitialTransformation = NULL);

  // Освободить данные куклы
  void Free();

  // Обновить куклу, используя гравитацию и амортизацию
  void Resolve(float Elapsed,
     float LinearDamping = -0.04f,
     float AngularDamping = -0.01f,
```

```
        D3DXVECTOR3 *vecGravity = &D3DXVECTOR3(0.0f, -9.8f, 0.0f),
        cCollision *pCollision = NULL,
        D3DXMATRIX *matCollision = NULL);

    // Перестроить иерархию фреймов
    void RebuildHierarchy();
    // Функции, возвращающие количество костей и указатель на
    // заданную кость
    DWORD GetNumBones();
    cRagdollBone *GetBone(DWORD BoneNum);
    };
```

Не беря во внимание конструктор и деструктор класса, которые используются для очистки и освобождения данных класса, вашему вниманию предоставляется шесть функций. Первая функция Create предназначена для установки данных класса. Установка включает в себя просмотр всех костей в иерархии фреймов и создание объекта кости твердого тела (cRagdollBone) для каждой. После этого кости преобразовываются таким образом, чтобы их положение и ориентация соответствовала фреймам. После создания объектов костей, вы можете освободить данные куклы, вызвав Free.

Между вызовами Create и Free вы в основном имеете дело с функцией Resolve. Эта функция принимает в качестве параметров количество обрабатываемых секунд, величину применяемой линейной и угловой амортизации, используемый вектор силы тяжести и указатель на массив объектов столкновений, используемых для проверки столкновений.

После вычисления части симуляции при помощи функции Resolve вы вызываете RebuildHierarchy для обновления иерархии фреймов. После ее обновления вы можете обновлять скелетный меш и визуализировать его!

Последние две функции позволяют узнать количество костей, содержащихся в твердом теле и получить указатель на заданную кость. Эти две функции полезны, если вы хотите использовать кость непосредственно для получения ее положения или скорости.

После того как вы увидели три класса, с которыми вы будете работать, давайте подробнее рассмотрим функции, о которых ранее не было написано, начав с функции, которая создает подходящие данные для каждой кости в иерархии фреймов меша.

Создание данных костей

Я не хочу бить мертвую лошадь, поэтому я упомяну только те части данных, о которых я не рассказывал ранее. Кроме положения и ориентации кости степень разрешения вращения определяет, насколько ваш меш сохраняет свою начальную форму со временем. Чем меньше значение степени разрешения, тем сильнее может искажаться тело. Чем больше разрешение, тем тело становится устойчивее к искажениям.

Я расскажу о степени разрешения немного позже; а пока, я хочу поговорить о других вещах, таких как вычисление начальной ориентации, размера ограничивающего параллелепипеда каждой кости, локальных точек, формирующих углы ограничивающего параллелепипеда, коэффициента restitution и векторов родительского смещения.

Вы уже читали практически обо всех этих переменных, но я ничего не говорил о вычислении размера ограничивающего параллелепипеда и начальной ориентации каждой кости. Я начну с вычисления ориентации.

Помните, давно, в разделе "Расположение и ориентация твердых тел", я сказал, что можно использовать кватернион для ориентации твердого тела? Если в качестве источника кукольных костей вы используете иерархию фреймов, вам необходимо перейти от матрицы преобразования к кватерниону. Как же это сделать?

Здесь приходит на помощь D3DX! Используя комбинированную матрицу преобразования фрейма, вы можете вызвать функцию D3DXQuatemionRotationMatrix, которая преобразует матрицу в кватернион! Например, у вас есть указатель на фрейм D3DXFRAME_EX pFrame. Для создания кватерниона вы можете использовать следующий код:

```
// pFrarae = указатель на фрейм
// quatOrientation = результирующий кватернион
D3DXQuaternionRotationMatrix(&quatOrientation, \
        &m_Frame->matCombined);
D3DXQuaternionInverse(&quatOrientation,  &quatOrientation);
```

Вы заметите, что при преобразовании матрицы в кватернион я добавил вызов функции D3DXQuaternionInverse, которая обращает кватернион. Причиной является то, что кватернионы используют правостороннюю систему координат. Т. к. Direct3D (и мы вместе с ним) использует левостороннюю систему координат, кватернион должен быть соответственно преобразован (обращен).

После того как вы получили начальную ориентацию, с которой можете работать, вы можете вычислить размеры ограничивающего параллелепипеда кости; создайте набор точек, представляющих углы ограничивающего параллелепипеда и точки соединения (точки, присоединяющие кость к родителю); установите массу и коэффи-

циент ограничения; создайте обратный тензор инерции. Исходный код, о котором я говорю, находится в функции cRagdoll::Create. Пожалуйста, проконсультируйтесь с подробно комментированным исходным кодом, чтобы увидеть, как устанавливаются соответствующие данные при создании кукольных костей.

А пока что позвольте мне объяснить, как вычислять размеры ограничивающих параллелепипедов.

Вычисление ограничивающего параллелепипеда кости

Для создания класса кости твердого тела, представляющего каждую кость, сначала необходимо получить обратную матрицу преобразования кости, используя функцию скелетного меша ID3DXSkinInfo::GetBoneOffsetMatrix. Эта обратная матрица преобразования кости ориентирует вершины скелетного меша относительно центра меша (в противоположность центру фрейма).

Помните, в главе 4 я объяснял, как должны быть расположены вершины скелетного меша относительно центра меша, чтобы они корректно вращались относительно центра кости? Весь процесс преобразования вершин заключается в наложении обратной матрицы преобразования с последующим поворотом кости и наложении преобразования перемещения, скомбинированного с родительским преобразованием фрейма.

После того как вы получили обратную матрицу преобразования кости, необходимо перебирать все вершины, присоединенные к ней. При этом каждую перебираемую вершину необходимо преобразовывать, используя обратное преобразование кости. Используя координаты только что преобразованной вершины, вы можете вычислить расширение ограничивающего параллелепипеда (который, в конце концов, будет содержать все вершины и точки соединения костей).

Функция cRagdoll::GetBoundBoxSize вычисляет ограничивающий параллелепипед. В качестве ее параметров задаются указатель на структуру фрейма (который представляет собой кость) и два вектора, содержащие размер ограничивающего параллелепипеда и смещение его центра относительно точки присоединения к родительской кости.

```
void cRagdoll::GetBoundingBoxSize(D3DXFRAME_EX *pFrame,
                D3DXVECTOR3 *vecSize,
                D3DXVECTOR3 *vecJointOffset)
{
```

Я расскажу о векторах размера и смещения немного позже. А пока, поместите в начале функции GetBoundingBoxSize код, создающий и очищающий пару векторов, которые будут содержать координаты расширений твердого тела и в последствии будут использованы для создания восьми угловых точек твердого тела.

```
// Установить минимальные и максимальные координаты по умолчанию
D3DXVECTOR3 vecMin = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
D3DXVECTOR3 vecMax = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
```

Первой задачей GetBoundingBoxSize является поиск кости, имеющей такое же имя, как и заданный фрейм. Эта кость, точнее интерфейс объекта кости скелетного меша (ID3DXSkinInfo), делает запрос о вершинах, присоединенных к ней.

```
// Обрабатывать вершины кости, если кость задана
if(pFrame->Name) {

  // Получить указатель на интерфейс ID3DXSkinInfo
  ID3DXSkinInfo *pSkin = pMesh->pSkinInfo;

  // Найти кость с таким же именем, как и у фрейма
  DWORD BoneNum = -1;
  for(DWORD i=0;i<pSkin->GetNumBones();i++) {
   if(!strcmp(pSkin->GetBoneName(i), pFrame->Name)) {
    BoneNum = i;
    break;
   }
 }

// Обработать вершины, если кость была найдена
if(BoneNum != -1) {
```

После того как вы обнаружили ID3DXSkinInfo для интересующей кости, вы можете сделать ей запрос о количестве присоединенных к ней вершин и создать массивы DWORD и вещественных чисел для хранения индексов вершин и их весов.

```
// Получить количество присоединенных вершин
DWORD NumVertices = pSkin->GetNumBoneInfluences(BoneNum);
if(NumVertices) {

  // Получить влияния костей
  DWORD *Vertices = new DWORD[NumVertices];
  float *Weights = new float[NumVertices];
  pSkin->GetBoneInfluence(BoneNum, Vertices, Weights);
```

После того как вы получили индексы вершин, хранящиеся в буфере Vertices (который вы заполняете с помощью функции GetBoneInfluence), вы можете начать просматривать вершины, преобразовывать их с помощью обратного преобразования кости и использовать преобразованные вершины для вычисления размера ограничивающего параллелепипеда.

```
// Получить тип данных вершин
DWORD Stride = D3DXGetFVFVertexSize( \
      pMesh->MeshData.pMesh->GetFVF());

// Получить обратную матрицу преобразования смещения кости
D3DXMATRIX *matInvBone = \
      pSkin->GetBoneOffsetMatrix(BoneNum);

// Заблокировать буфер вершин и просмотреть все вершины,
// присоединенные к кости
char *pVertices;
pMesh->MeshData.pMesh->LockVertexBuffer( \
      D3DLOCK_READONLY, (void**)&pVertices);
for(i=0;i<NumVertices;i++) {

   // Получить указатель на координаты вершин
   D3DXVECTOR3 *vecPtr = \
       D3DXVECTOR3*)(pVertices+Vertices[i]*Stride);

   // Преобразовать вершины на преобразование смещения кости
   D3DXVECTOR3 vecPos;
   D3DXVec3TransformCoord(&vecPos, vecPtr, matInvBone);

   // Получить минимальные/максимальные значения
   vecMin.x = min(vecMin.x, vecPos.x);
   vecMin.y = min(vecMin.y, vecPos.y);
   vecMin.z = min(vecMin.z, vecPos.z);

   vecMax.x = max(vecMax.x, vecPos.x);
   vecMax.y = max(vecMax.y, vecPos.y);
   vecMax.z = max(vecMax.z, vecPos.z);
  }
  pMesh->MeshData.pMesh->UnlockVertexBuffer();

  // Освободить ресурсы
  delete [] Vertices;
  delete [] Weights;
 }
}
}
```

В конце выполнения этого кусочка кода в созданных в начале функции векторах (vecMin и vecMax) будут храниться размеры ограничивающего параллелепипеда. Массив индексов вершин освобождается (как и весов вершин), и обработка продолжается нахождением точек присоединения кости к родителю и потомкам.

```
// Учесть точки присоединения потомков в размерах
if(pFrame->pFrameFirstChild) {

  // Получить обратное преобразование кости для расположения
  // дочерних соединений
  D3DXMATRIX matInvFrame;

  D3DXMatrixInverse(&matInvFrame,NULL,&pFrame->matCombined);
  // Перебрать все дочерние фреймы, присоединенные к текущему
  D3DXFRAME_EX *pFrameChild = \
      D3DXFRAME_EX*)pFrame->pFrameFirstChild;
  while(pFrameChild) {
   // Получить координаты вершины фрейма и преобразовать их
   D3DXVECTOR3 vecPos;
   vecPos = D3DXVECTOR3(pFrameChild->matCombined._41,
            pFrameChild->matCombined._42,
            pFrameChild->matCombined._43);
   D3DXVec3TransformCoord(&vecPos, &vecPos, &matInvFrame);

   // Получить минимальные/максимальные значения
   vecMin.x = min(vecMin.x, vecPos.x);
   vecMin.y = min(vecMin.y, vecPos.y);
   vecMin.z = min(vecMin.z, vecPos.z);

   vecMax.x = max(vecMax.x, vecPos.x);
   vecMax.y = max(vecMax.y, vecPos.y);
   vecMax.z = max(vecMax.z, vecPos.z);

   // Перейти к следующей дочерней кости
   pFrameChild =  (D3DXFRAME_EX*)pFrameChild->pFrameSibling;
  }
}
```

Обычно для учета точек соединения берутся координаты присоединенной кости в мировом пространстве и преобразуются на обратную матрицу кости. После чего полученные координаты сравниваются с координатами, хранящимися в векторах vecMin и vecMax.

Теперь вы можете закончить функцию, сохранив размер параллелепипеда. Если его размер слишком мал, то он устанавливается в минимальное значение (при помощи макроса MINIMUM_BONE_SIZE, который устанавливает в 1.0)

```
// Установить размер ограничивающего параллелепипеда
vecSize->x = (float)fabs(vecMax.x - vecMin.x);
vecSize->y = (float)fabs(vecMax.y - vecMin.y);
vecSize->z = (float)fabs(vecMax.z - vecMin.z);

// Убедиться, что каждая кость имеет минимальный размер
if(vecSize->x < MINIMUM_BONE_SIZE) {
  vecSize->x = MINIMUM_BONE_SIZE;
  vecMax.x = MINIMUM_BONE_SIZE*0.5f;
}
```

```
if(vecSize->y < MINIMUM_BONE_SIZE) {
  vecSize->y = MINIMUM_BONE_SIZE;
  vecMax.y = MINIMUM_BONE_SIZE*0.5f;
}
ifvecSize->z < MINIMUM_BONE_SIZE) {
  vecSize->z = MINIMUM_BONE_SIZE;
  vecMax.z  = MINIMUM_BONE_SIZE*0.5f;
 }

 // Установать смещение кости в центр, основываясь на половине
 // размера ограничивающего параллелепипеда и максимальном
 // положении
 (*vecJointOffset) = ((*vecSize) * 0.5f) - vecMax;
}
```

В самом конце функции вы наконец то обнаружите вектор vecJointOffset о котором я говорил при создании функции GetBoundingBoxSize. Т. к. кость твердого тела может быть любого размера, а вы отслеживаете ее с помощью координат ее центра, необходимо создать дополнительную точку, которая бы представляла собой точку присоединения ограничивающего параллелепипеда к родителю. Это и есть вектор смещения соединения. Вы узнаете о нем больше при изучении усиления связей костей.

После того как вы вычислили размер ограничивающего параллелепипеда и установили разнообразные данные кости, вы можете прикладывать различные силы для нахождения движения костей.

Установка сил

Для любой прикладываемой силы (решайте сами, какие использовать) вы должны учитывать силу тяжести и амортизации. В классе cRagdoll я определил функцию, которая находит векторы силы и момента одной кости, после чего применяет к ней силы тяжести и амортизации.

```
void cRagdoll::SetForces(DWORD BoneNum,
     D3DXVECTOR3 *vecGravity,
     float LinearDamping,
     float AngularDamping)
{
  // Получить указатель на кость
  cRagdollBone *Bone = &m_Bones[BoneNum];

  // Получить указатель на текущее состояние
  cRagdollBoneState *BCState = &Bone->m_State;

  // Установить гравитацию и очистить момент
  Bone->m_vecForce = ((*vecGravity) * Bone->m_Mass);
  Bone->m_vecTorque = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
```

```
    // Наложить амортизацию на силу и момент
    Bone->m_vecForce += (BCState->m_vecLinearVelocity * \
            LinearDamping);
    Bone->m_vecTorque += (BCState->m_vecAngularVelocity * \
            AngularDamping);
}
```

Вы читали о силах тяжести и амортизации ранее в этой главе, поэтому мне не надо объяснять их опять. Вы заметите, что я масштабирую вектор силы тяжести на массу кости. Помните, что это необходимо для того, чтобы гравитация притягивала все объекты с одинаковой силой, когда вы потом масштабируете силы в соответствии с массой.

После того как вы установили силы, вы можете находить (интегрировать) движение заданной кости.

Объединение костей

После того как вы установили силу и момент кости, вы можете использовать эти векторы для нахождения движения твердого тела. Вы заметите, что на данный момент я работаю с одной костью. Достаточно двигать кости поочередно, в конце концов, объединив их перед визуализированием меша.

В разделе "Обработка движений твердого тела" вы видели, как прикладывать векторы силы и моменты к линейной скорости и угловому моменту для создания движения. В функции cRagdoll::Resolve я просто повторяю все прочитанное вами в том разделе.

Вы заметите, что значения положения, ориентации, скорости и момента хранятся в классе состояния кости cRagdollBoneState. Эти векторы используются при разрешении. Для вызова Integrate необходимо задать в качестве параметров обрабатываемую кость и время обработки (прошедшее время).

```
void cRagdoll::Integrate(DWORD BoneNum, float Elapsed)
{
  // Получить указатель на кость
  cRagdollBone *Bone = &m_Bones[BoneNum];

  // Получить указатели на состояния
  cRagdollBoneState *State = &Bone->m_State;
```

После того как вы получили указатель на класс кости (и указатель на объект состояния кости), вы вычисляете новое положение кости, основываясь на линейной скорости, изменение углового момента, основываясь на моменте, и изменение линейной скорости, основываясь на величине силы (масштабированной на массу объекта).

```
// Сложить положения
State->m_vecPosition += (Elapsed*State->m_vecLinearVelocity);

// Сложить угловой момент
State->m_vecAngularMomentum += (Elapsed * Bone->m_vecTorque);

// Сложить линейную скорость
State->m_vecLinearVelocity += Elapsed * Bone->m_vecForce / \
        Bone->m_Mass;
```

Теперь вычисляется новая ориентация (хранимая в кватернионе), используя угловую скорость, умноженную на прошедшее время.

```
// Сложить ориентацию кватернионов
D3DXVECTOR3 vecVelocity = Elapsed * State->m_vecAngularVelocity;
State->m_quatOrientation.w  -= 0.5f *
      (State->m_quatOrientation.x * vecVelocity.x +
       State->m_quatOrientation.y  * vecVelocity.y +
       State->m_quatOrientation.z * vecVelocity.z);
State->m_quatOrTentation.x += 0.5f *
      (State->m_quatOrientation.w  * vecVelocity.x -
       State->m_quatOrientation.z  * vecVelocity.y +
       State->m_quatOrientation.y * vecVelocity.z);
State->m_quatOrientation.y  += 0.5f *
      (State->m_quatOrientation.z  * vecVelocity.x +
       State->m_quatOrientation.w  * vecVelocity.y -
       State->m_quatOrientation.x * vecVelocity.z);
State->m_quatOrientation.z  += 0.5f *
      (State->m_quatOrientation.x * vecVelocity.y -
       State->m_quatOrientation.y  * vecVelocity.x +
       State->m_quatOrientation.w * vecVelocity.z);

// Нормализовать кватернион (получая единичный кватернион)
D3DXQuaternionNormalize(&State->m_quatOrientation,
          &State->m_quatOrientation);
```

До этого момента я не рассказывал вам, как предотвратить неуправляемое искажение костей кукольного меша. Подумайте, т. к. кости являются твердыми телами, они могут вращаться в любом направлении на любой угол, так что, например, голова вашего персонажа может вращаться через грудь. Таким образом, я представляю вашему вниманию использование фактора разрешения вращения в объявлении каждого кукольного объекта.

После того как вы нашли новую ориентацию кости, вам необходимо медленно вернуть ее в начальное состояние. Вы можете выполнить это, предварительно вычислив разность ориентации кости и ее родителя. Используя эту предварительно вычисленную разность, вы рассчитываете ориентацию, которую должна принять кость.

Чем больший коэффициент разрешения вы установите, тем быстрее кость будет приобретать начальную ориентацию относительно родительской ориентации. Для переориентирования кости к начальному положению вы можете использовать сферическую интерполяцию от текущей ориентации кости к начальной ориентации относительно родительской ориентации. Величина интерполяции, вы угадали, определяется установленным коэффициентом разрешения. Большие значения заставляют не вращаться кости, в то время как маленькие значения заставляют кость постепенно (или никогда) возвращаться к начальной ориентации.

```
// Заставить разрешение вращения
if(BoneNum && Bone->m_ResolutionRate != 0.0f)  {

  // сферическая интерполяция от текущей ориентации к начальной
  D3DXQUATERNION quatOrientation = \
      Bone->m_ParentBone->m_State.m_quatOrientation * \
      Bone->m_quatOrientation;
  D3DXQuaternionSlerp(&State->m_quatOrientation, \
         &State->m_quatOrientation, \
         &quatOrientation, \
          Bone->m_ResolutionRate);
}
```

Двигаемся дальше. Оставшийся код функции Integrate создает матрицу преобразования, которая потом используется для преобразования точек костей твердого тела и создания угловой скорости. Т. к. мы работаем с левосторонней системой координат, после создания матрицы преобразования необходимо ее транспонировать при помощи функции D3DXMatrixRotationQuaternion. Этот шаг обсуждался ранее.

```
// Вычислить новую ориентацию при помощи матрицы преобразования
// полученной из только что посчитанного кватерниона
D3DXMatrixRotationQuaternion(&State->m_matOrientation,
          &State->m_quatOrientation);
D3DXMatrixTranspose(&State->m_matOrientation,
          &State->m_matOrientation);

// Вычислить суммарный тензор мировой инерции
D3DXMATRIX matTransposedOrientation;
D3DXMatrixTransposw(&matTransposedOrientation, &State-
>m_matOrientation);
State->m_matInvWorldInertiaTensor = State->m_matOrientation *
       Bone->m_matInvInertiaTensor *
       matTransposedOrientation;
```

```
// Вычислить новую угловую скорость
State->m_vecAngularVelocity = Transform(&State-
>m_vecAngularMomentum,
           &State->m_matInvWorldInertiaTensor);
}
```

На данный момент мы рассчитали движение твердого тела, а теперь пришло время обрабатывать столкновения.

Обработка столкновений

Как вы помните из предыдущего материала этой главы, столкновения обрабатываются проверкой точек каждого твердого тела на попадание в объекты списка столкновений. Если точка находится внутри пространства сталкиваемого объекта, то тогда она выталкивается из него, а вектор импульса усредняется между всеми объектами таким образом, что тело отскакивает реалистично. Этой цели служит функция ProcessCollisions.

Я не буду приводить здесь полный код функции ProcessCollisions; вместо этого я покажу отдельные фрагменты. Функция ProcessCollisions начинается с прототипа, который принимает в качестве параметров номер проверяемой кости и указатель на корневой объект столкновений.

```
BOOL cRagdoll::ProcessCollisions(DWORD BoneNum, \
       cCollision *pCollision)
{
  // Проверка на ошибки
  if(!pCollision || !pCollision->m_NumObjects || \
    !pCollision->m_Objects)
   return TRUE;

  // Получить указатель на кость
  cRagdollBone *Bone = &m_Bones[BoneNum];

  // Получить указатель на состояние
  cRagdollBoneState *State = &Bone->m_State;

  // Сохранять количество столкновений
  DWORD CollisionCount = 0;

  // Сохранять количество сил столкновений
  D3DXVECTOR3 vecLinearVelocity = D3DXVECTOR3(0.0f,0.0f,0.0f) ;
  D3DXVECTOR3 vecAngularMomentum = D3DXVECTOR3(0.0f,0.0f,0.0f);
```

Итак, функция ProcessCollisions начинает свою работу с получения указателя на объект кости и объект состояния кости. После этого вам необходимо следить за количеством обнаруженных столкновений (для последующего усреднения движений) и определить два вектора, определяющих встроенные импульсы, двигающие и вращающие кости в соответствии со столкновениями.

Двигаясь дальше в коде, вы начинаете просматривать восемь точек костей объекта твердого тела. Для каждой точки вы просматриваете все объекты столкновений. Прежде чем проверять столкновение точки и объекта, необходимо установить флаг, который отвечал бы за три вещи: столкнулась ли точка с объектом, нормаль объекта и расстояние, на которое точка проникает в объект.

```
// Просмотреть все вершины кости в поисках столкновений
for(DWORD i=0;i<8;i++) {

  // Просмотреть все объекты столкновений
  cCollisionObject *pObj = pCollision->m_Objects;
  while(pObj) {

  // Установить флаг, если обнаружено столкновение
  BOOL Collision = FALSE;

  // Нормаль объекта столкновений
  D3DXVECTOR3 vecCollisionNormal;

  // Расстояние, на которое необходимо вытолкнуть точку из объекта
  float CollisionDistance = 0.0f;

  // Обработать сферический объект столкновения
  if(pObj->m_Type == COLLISION_SPHERE) {
```

Я собираюсь вырезать кусочек кода, потому что вы уже видели, как определять столкновения, вычислять нормаль и расстояние, на которое точка проникает в объект. Если вы посмотрите полный исходный код, вы увидите, что я проверяю столкновение точки и сферы и точки и плоскости.

После того как вы выполнили обнаружение столкновений, флаг Collision должен быть установлен в TRUE, если столкновения были, или в FALSE, если их не было. Если флаг установлен в TRUE, тогда точка выталкивается из объекта, и вычисляются корректные векторы импульсов.

```
// Обработать столкновение, если обнаружено
  if (Collision == TRUE) {

    // Вытолкнуть объект на поверхность объекта столкновения
    State->m_vecPosition += (vecCollisionNormal * \
              CollisionDistance);
```

```
    // Получить положение и скорость точки
    D3DXVECTOR3 vecPtoP = State->m_vecPosition - \
           State->m_vecPoints[i];
    D3DXVECTOR3 vecPtoPVelocity = \
                State->m_vecLinearVelocity + \
           CrossProduct(&State->m_vecAngularVelocity, \
                        &vecPtoP);  \

    // Получить скорость точки относительно поверхности
    float PointSpeed = D3DXVec3Dot(&vecCollisionNormal, \
                  &vecPtoPVelocity);

    // Увеличить количество столкновений
    CollisionCount++;

    // Вычислить силу импульса, основываясь на коэффициенте
    // restitution, скорости точки и нормали сталкиваемого объекта
    float ImpulseForce = PointSpeed * \
          (-(1.0f + Bone->m_Coefficient));
    float ImpulseDamping = (1.0f / Bone->m_Mass) + \
          D3DXVec3Dot(&CrossProduct( \
         &Transform(&CrossProduct(&vecPtoP, \
          &vecCollisionNormal), \
         &State->m_matInvWorldInertiaTensor), \
          &vecPtoP), &vecCollisionNormal);
    D3DXVECTOR3 vecImpulse = vecCollisionNormal * \
             (ImpulseForce/ImpulseDamping);

    // Добавить силы
    vecLinearVelocity += vecImpulse;
  vecAngularMomentum += CrossProduct(&vecPtoP, &vecImpulse);
}
```

После последнего кусочка кода проверяются остальные объекты столкновений и обрабатываются оставшиеся точки. В конце, если были обнаружены столкновения, функция ProcessCollisions усредняет векторы импульсов и применяет их к линейной скорости и угловому моменту.

```
// Были ли столкновения
if(CollisionCount) {

  // Добавить усредненные силы к интегрированному состоянию
  State->m_vecLinearVelocity += ((vecLinearVelocity / \
           Bone->m_Mass) / (float)CollisionCount);
  State->m_vecAngularMomentum += (vecAngularMomentum / \
                (float)CollisionCount);

  // Вычислить угловую скорость
  State->m_vecAngularVelocity = Transform( \
           &State->m_vecAngularMomentum, \
           &State->m_matInvWorldInertiaTensor);
}
```

Следующая функция, которую я хочу вам показать, поможет восстановить соединения костей для сохранения формы вашего кукольного персонажа.

Восстановление соединений костей

После того, как вы нашли все силы и обработали столкновения, осталось выполнить последний шаг перед обновлением иерархии фреймов и визуализацией меша кукольного персонажа. Вы должны решить соединения костей.

Вспомните, в разделе "Соединение твердых тел с помощью пружин" вы научились создавать пружины между костями и использовать их для вычисления вектора силы, который немедленно перемещает и вращает тело в корректное положение. Это как раз и выполняет функция ProcessConnections - использует координаты преобразованной точки, представляющей смещение соединения (смещение от центра кости до точки ее присоединения к родителю) и координаты присоединения к родителю (хранящиеся в другой точке) для создания пружины.

```
void cRagdoll::ProcessConnections(DWORD BoneNum)
{
  // Получить указатель на кость и родительскую кость
  cRagdollBone *Bone = &m_Bones[BoneNum];
  cRagdollBone *ParentBone = Bone->m_ParentBone;

  // Если родительской кости нет, то не продолжать
  if(!ParentBone)
   return;

  // Получить указатель на состояние кости
  cRagdollBoneState *BState = &Bone->m_State;

  // Получить указатель на родительское состояние
  cRagdollBoneState *PState = &ParentBone->m_State;

  // Получить положение соединения и вектор к центру
  D3DXVECTOR3 vecBonePos = BState->m_vecPoints[8];
  D3DXVECTOR3 vecBtoC = BState->m_vecPosition - vecBonePos;

  // Получить координаты присоединения к родителю
  D3DXVECTOR3 vecParentPos = BState->m_vecPoints[9];

  // Вычислить вектор пружины из точки к родительской точке
  D3DXVECTOR3 vecSpring = vecBonePos - vecParentPos;

  // Переместить точку в соответствии с родительской и
  // скорректировать угловую скорость и момент
  BState->m_vecPosition -= vecSpring;
  BState->m_vecAngularMomentum -= CrossProduct(&vecBtoC, \
            &vecSpring);
BState->m_vecAngularVelocity = Transform( \
```

```
                &BState->m_vecAngularMomentum,
                &BState->m_matInvWorldInertiaTensor);
}
```

Я знаю, что я излагаю все быстро, но я уже объяснял этот код ранее; я просто хотел повторить основные моменты. Следующая и последняя функция, которую я вам хочу показать, перестраивает иерархию фреймов, позволяя вам обновить скелетный меш.

Перестроение иерархии

После разрешения модели твердого тела единственное, что вам остается сделать, - это перестроить преобразования фреймов. Т. к. кости твердого тела куклы заданы в мировых координатах, нет необходимости просматривать все кости в иерархии и комбинировать все преобразования с родительским. Обычно все, что необходимо вам сделать, - это скопировать ориентацию кости (из состояния объекта) в комбинированную матрицу преобразования фрейма.

Единственной проблемой является то, что кость ориентируется и перемещается относительно координат центра тела, а не относительно координат точки соединения, как определено вектором vecJointOffset. Вам необходимо преобразовать вектор vecJointOffset, используя ориентацию кости, и добавить результирующий вектор к положению кости, что даст вам корректные координаты для расположения кости. Комментарии, приведенные в RebuildHierarchy, должны объяснить все сложные моменты.

```
void cRagdoll::RebuildHierarchy()
{
  if(!m_pFrame)
   return;
  if(!m_NumBones || !m_Bones)
   return;

  // Наложить вращательные матрицы костей на фреймы
  for(DWORD i=0;i<m_NumBones;i++) {

   // Преобразовать смещение соединения в соответствии с положением
фрейма
   D3DXVECTOR3 vecPos;
   D3DXVec3TransformCoord(&vecPos,
          &m_Bones[i].m_vecJointOffset,
          &m_Bones[i].m_State.m_matOrientation);

   // Добавить положение кости
   vecPos += m_Bones[i].m_State.m_vecPosition;
```

```
        // Ориентировать и расположить фрейм
        m_Bones[i].m_Frame->matCombined =
        m_Bones[i].m_State.m_matOrientation;
        m_Bones[i].m_Frame->matCombined._41 = vecPos.x;
        m_Bones[i].m_Frame->matCombined._42 = vecPos.y;
        m_Bones[i].m_Frame->matCombined._43 = vecPos.z;
      }
    }
```

Вот мы и закончили! Все что остается, это найти применение классу кукольной анимации в ваших проектах. Для этого вам просто необходимо создать экземпляр объекта cRagdoll, вызвать функцию Create для создания данных и непрерывно вызывать Resolve и RebuildHierarchy. Демонстрационная программа этой главы показывает, как просто это делается, так что я настоятельно рекомендую посмотреть ее исходный код, прежде чем продолжать.

Посмотрите демонстрационные программы

Как вы можете видеть, системы кукольной анимации являются всего лишь замаскированными моделями твердых тел. Зачем платить тысячи долларов за систему кукольной анимации, если вы видели, как просто создать ее самому? Демонстрационная программа этой главы содержит как раз такую систему кукольной анимации, которую вы можете использовать в ваших проектах.

Эта программа, изображенная на рис. 7.12, берет персонаж и подбрасывает его в воздух. Звучит не очень впечатляюще, так что для большего реализма было добавлено несколько объектов столкновений (сфер), от которых бедный персонаж отскакивает как тряпичная кукла.

Раздел физики твердого тела является, безусловно, привлекательным; если вы захотите получить больше информации, чем я привел тут, есть множество альтернативных источников.

Программы на компакт-диске

Для главы 7 создан только один проект, но поверьте мне, он замечательный! Вы можете обнаружить его в директории главы 7 компакт диска этой книги:

- **Ragdoll.** Этот проект иллюстрирует кукольную анимацию, показывая вам что случается, если бросить персонаж через поле плавающих сфер. Он расположен в \BookCode\Chap07\Ragdoll.

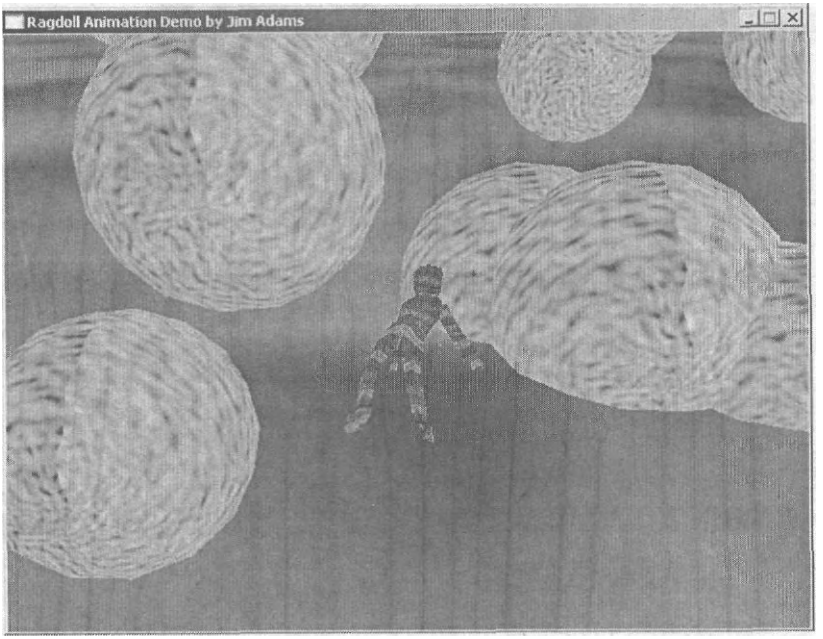


Рис. 7.12. Деревянная кукла встречает ужасную смерть, пролетев по воздуху и отскочив от плавающих сфер

Часть IV

Морфирующая анимация

Глава 8

Работа с морфирующей анимацией

С громким лязгом стена, пересекающая компаунд, начинает скользить. Звуки крошащихся камней перемешиваются со свирепым ревом невиданных злобных созданий, которые притаились за уменьшающимся барьером. Я могу только догадываться, что сейчас обрушится на меня. Как только стена осела, началось нападение. Волна за волной пиксельных демонов лилась вперед только для того, чтобы встретить свою смерть от моей проверенной импульсной винтовки.

Со всеми этими бегающими демонами, скрежещущими зубами, кусочками летающей плоти, я уверен, что технология анимации не стоит во главе списка ваших приоритетов. Но если бы вам пришлось о ней задуматься, я уверен, что вам на ум пришли бы сложные скелетные структуры, иерархии фреймов, скелетные меши, разве нет? А будет ли для вас сюрпризом то, что все вышерассмотренные анимации могут быть воплощены при использовании системы морфирующей анимации? Так и есть, в играх, такого плана как Doom и Quake фирмы id, демонические твари анимируются при помощи морфирующей анимации, технологии, которая обеспечивает плавность и простоту проигрывания даже на медленных системах. Морфирующая анимация настолько проста в использовании, что вы спросите у себя, почему вы не рассмотрели ее раньше. Не расстраивайтесь, потому что эта глава и существует, чтобы помочь вам!

В этой главе вы научитесь:

- Определять морфируемые последовательности;
- Морфировать меши, используя время и скаляры;
- Визуализировать морфируемые меши;
- Улучшать морфирующие анимации при помощи вершинных шейдеров.

Морфинг в действии

Возвратимся в ранние 90-е. Революционная технология компьютерной графики, известная как морфинг, появилась на большой арене благодаря Майклу Джексону. Нет, я не говорю о его неудачной пластической операции, я говорю об использовании морфинга в одном из его видео-клипов. Да, Король Попа использовал технологии морфинга в видео-клипе на свою песню "Black or White" и создал феномен анимации, который существует и по сей день.

В случае если вы не видели этот клип, позвольте мне объяснить. Он включает отрывок, в котором человек танцует под мелодию, а камера направлена на его лицо. Каждые несколько секунд его лицо превращается в лицо какого-нибудь другого человека. И так продолжается, пока лица не превратятся более чем 10 раз. Результат использования морфинга в этом клипе просто потрясающ, и я до сих пор помню его отчетливо!

По мере того как проходили годы, морфинг наконец то добрался до игр. В то время как в старые времена под морфингом понимали цифровую обработку видео, включающую плавное изменение одной картинки на другую (как, например, в "Black or White"), современный же морфинг (или, по крайне мере, обсуждаемый нами) включает в себя плавное изменение трехмерного меша с течением времени.

Наверное, самым известным примером игры, использующей морфинг является id'шный Quake. В Quake все анимационные последовательности движения персонажей созданы из наборов морфируемых мешей. Один меш постепенно принимает форму второго меша, второй меш принимает форму третьего и т. д.

При использовании небольшого интервала времени и достаточного количества морфируемых мешей все последовательности анимаций будут плавными и очень легко обрабатываемыми. Quake может выполняться с приличной скоростью даже на самых медленных компьютерных системах. Причиной этого является простота работы с морфирующими анимациями, что вы и увидите в следующих главах.

Как вы уже могли предположить морфинг, иногда называемый твиннинг (tweening), как, например, в DirectX SDK) - это процесс изменения одной формы в другую с течением времени. Для нас этими формами являются меши. Процесс морфинга меша включает в себя постепенное изменение координат вершин меша, начиная с одной формы и переходя к другой.

Меш, содержащий ориентацию вершин в начале процесса морфинга, называется исходным мешем. Второй меш, ориентацию вершин которого принимает исходный в результате процесса морфинга, называется целевым мешем. Давайте рассмотрим эти меши более подробно, чтобы лучше понимать сам процесс морфинга.

Определение исходного и целевого меша

Исходный и целевой меши, с которыми вы будете работать в этой книге, являются обычными объектами ID3DXMesh. Однако вы не можете просто использовать два меша для морфинга; существуют определенные правила. Во-первых, каждый меш должен иметь одинаковое количество вершин. В процессе морфинга вершины исходного меша просто перемещаются в положение вершин целевого меша. Отсюда следует второе правило: каждой вершине исходного меша должна соответствовать вершина (точнее индекс вершины) в целевом меше. В качестве примера посмотрите на меши, изображенные на рис. 8.1.

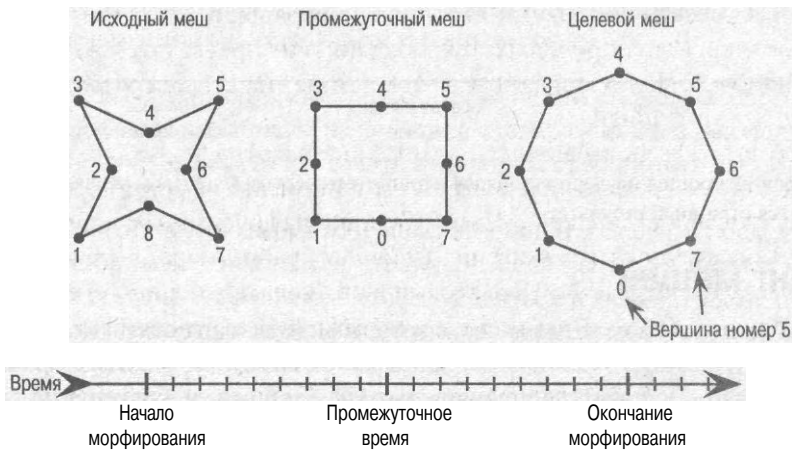


Рис. 8.1. Во время процесса морфинга вершины исходного меша плавно перемещаются в положение вершин целевого меша. Каждая вершина имеет одинаковый индекс как в исходном, так и в целевом меше

Порядок вершин является важным. Вершина, движущаяся из исходного меша, должна двигаться к вершине, имеющей тот же самый индекс в целевом меше. Если вы перенумеруете порядок, вершины будут двигаться в неправильных направлениях при морфинге, производя забавно выглядящие результаты, как показано на рис. 8.2.

Если вы создаете меши, содержащие одинаковое количество вершин и имеющие одинаковый их порядок, то все в нормально. Что касается получения фактических данных меша, я оставляю это вам. Вы можете использовать функцию D3DXLoadMeshFromX или любую функцию, которую вы разработали в главе 1, для загрузки мешей. После того как вы загрузили два меша и подготовили их к использованию, можно их морфировать!

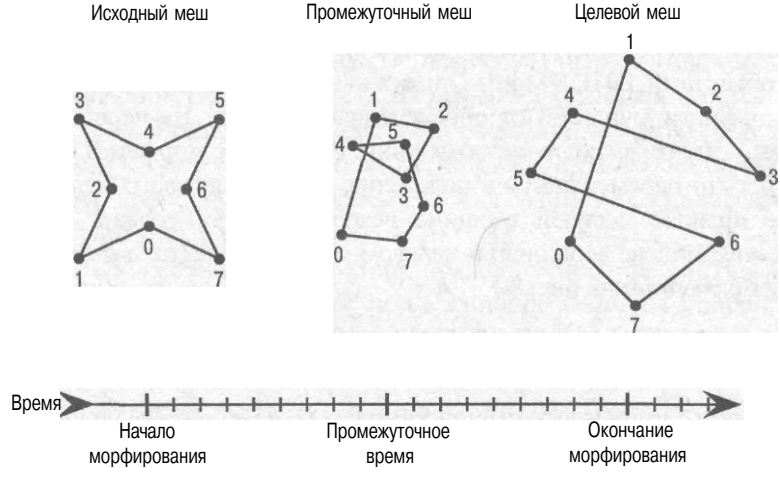


Рис. 8.2. Морфинг прошел неудачно, порядок вершин в исходном и целевом мешах различен, из-за чего получается странный результат

Морфинг мешей

Теперь, когда вы имеете два меша, с которыми будете работать (исходный и целевой), вы можете начинать процесс морфирования. Помните, что морфирование - это процесс изменения одной формы на другую. Необходимо чтобы вершины исходного меша, находящиеся в начальном положении, постепенно двигались к положению вершин целевого меша.

Вы можете измерять интервал времени, используемый для отслеживания движения вершин из координат исходного меша в координаты целевого, при помощи скаляра (находящегося в диапазоне от 0 до 1). При значении скаляра, равного 0, вершины будут расположены в исходном меше, в то время как при значении скаляра 1 вершины будут располагаться в целевом меше. На рис. 8.3 показано, что любое значение скаляра от 0 до 1 расположит вершины между положением исходного и целевого меша.

Вычислять координаты расположения вершины между исходным и целевым мешами очень просто. Возьмите вершину исходного меша и умножьте ее координаты на обратное значение скаляра (1.0-scalar). Использование обратного значения скаляра означает, что координаты вершины будут определяться на 100 процентов координатами ее в исходном меше при значении скаляра, установленного в 0, и на 0 процентов при скаляре, установленном в 1.

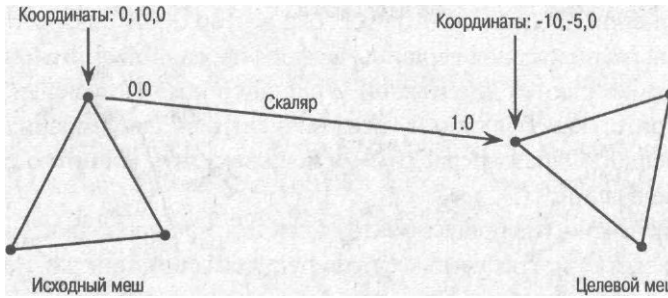


Рис. 8.3. Начиная с координат в исходном меше (и значении скаляра 0), вершина постепенно движется к положению в целевом меше по мере увеличения скаляра

Далее, используя вершину, имеющую тот же самый индекс целевого меша, что и ранее, умножьте ее координаты на значение скаляра. Сложив два получившихся значения, вы получите результирующие координаты вершины в сморфированной анимации.

Сначала концепция умножения координат вершины на скаляр и последующее сложение результатов может показаться странным. Если вы не уверены в математике, проделайте следующие вычисления, чтобы убедиться в правильности полученного результата. Используйте одномерное значение для представления координат вектора. Установите координату исходной точки в 10, а координату целевой точки в 20. Для упрощения, используйте значение скаляра равное 0.5, которое должно дать 15, в качестве результирующей координаты вершины при морфинге.

Умножив исходную координату (10) на 1-0.5 получим 5. Умножив целевую координату (20) на 0.5 получим 10. Сложив два результата (5 и 10) получим 15. Разве не замечательно - получить правильный результат!

Эту процедуру можно записать в виде следующего кода, полагая что координаты исходной вершины хранятся в vecSource, координаты целевой вершины хранятся в vecTarget, а'значение скаляра в Scalar.

```
// vecSource = D3DXVECTOR3 содержащий исходные координаты
// vecTarget = D3DXVECTOR3 содержащий целевые координаты
// Scalar = FLOAT содержащее значение скаляра

// Умножить исходные координаты на исходный скаляр
D3DXVECTOR3 vecSourcePos = vecSource *  (1.0f-Scalar);

// Умножить целевые координаты на скаляр
D3DXVECTOR3 vecTargetPos = vecTarget * Scalar;

// Сложить два получившихся вектора
D3DXVECTOR vecPos = vecSourcePos + vecTargetPos;
```

После выполнения последнего кусочка кода вектор будет содержать координаты, используемые для расположения вершины в сморфированной анимации. Конечно же, эти вычисления повторяются для каждой вершины в исходном меше. В следующем разделе вы увидите, как выполнять эти вычисления для создания собственных морфируемых мешей. Однако перед этим я хочу заметить кое-что о синхронизации морфирующей анимации.

До этого момента я игнорировал фактор времени: как продолжительность анимации (количество времени, требуемого морфирующей анимации для перемещения из исходного в целевое положение), так и точный момент времени из последовательности анимации, в который вычисляются координаты. Полагая, что вы измеряете время в миллисекундах и продолжительность анимации хранится в Length (вещественное значение), а текущее время, в которое вы вычисляете координаты, хранится в Time (также вещественное значение), вы можете вычислить соответствующее значение скаляра, используемого в расчетах, так:

```
Scalar = Time / Length;
```

Используя вычисления, виденные вами в этом разделе, вы можете применить этот скаляр для расчета координат вершин, образующих морфируемый меш.

Уже второй раз я говорю о создании морфируемого меша, так что я больше не буду тянуть. Читайте далее, чтобы узнать, как создавать морфируемый меш, который вы можете использовать для визуализации. Хотя ранее я и замечал об использовании вершинных шейдеров, сначала я покажу вам самый простой путь создания морфируемых мешей - непосредственной обработкой буфера вершин меша.

Создание морфированного меша при помощи обработки

Непосредственная обработка буфера вершин меша является, наверное, самым простым способом работы с морфингом. Для этого метода вам потребуется третий меш, который будет содержать результирующие координаты каждой вершины после морфирования, и именно этот меш вы визуализируете.

Для создания третьего меша, который я назвал результирующим морфированным мешем, вам необходимо скопировать исходный меш и изменять его.

```
// Объявить третий меш, используемый для результирующего
// морфированного меша
ID3DXMesh *pResultMesh = NULL;
```

```
// скопировать меш, используя исходный меш pSourceMesh
pSourceMesh->CloneMeshFVF(0, pSourceMesh->GetFVF(), \
            pDevice, &pResultMesh);
```

После создания результирующего морфированного меша (pResultMesh) вы можете начинать обрабатывать морфируемую анимацию, заблокировав буферы вершин исходного, целевого и результирующего морфированного мешей. Однако прежде чем это сделать вам необходимо объявить общую структуру вершин, которая будет содержать только их координаты, к которым вы получаете доступ при блокировке каждого буфера вершин.

```
typedef struct {
  D3DXVECTOR3 vecPos;
} sGenericVertex;
```

Также вам необходимо вычислить размер каждой структуры вершин, используемых вершинами меша, потому что каждый буфер вершин содержит вершины различных размеров (например, у исходного могут быть нормали, в то время как у целевого нет). Вы можете сделать это, используя функцию D3DXGetFVFVertexSize.

```
// pSourceMesh = объект исходного меша
// pTargetMesh = объект целевого меша
// pResultMesh = объект результирующего морфированного меша
DWORD SourceSize = D3DXGetFVFVertexSize(pSourceMesh->GetFVF());
DWORD TargetSize = D3DXGetFVFVertexSize(pTargetMesh->GetFVF());
DWORD ResultSize = D3DXGetFVFVertexSize(pResultMesh->GetFVF());
```

Теперь вы можете заблокировать буферы вершин и присвоить им указатели.

```
// Объявить указатели вершин
char *pSourcePtr, *pTargetPtr, *pResultPtr;
pSourceMesh->LockVertexBuffer(D3DLOCK_READONLY, \
            (void**)&pSourcePtr);
pTargetMesh->LockVertexBuffer(D3DLOCK_READONLY, \
            (void**)&pTargetPtr);
pResultMesh->LockVertexBuffer(0, (void**)&pResultPtr);
```

Заметили, как я присваиваю буферам вершин указатели char* вместо использования определенной структуры вершин? Вам необходимо поступать таким же бразом, потому что буферы вершин могут быть любого размера, помните? Как только вам необходимо получить доступ к вершине, вы приводите указатель к определенной структуре вершин и получаете доступ к данным. Для того чтобы перейти к следующей вершине, просто добавьте размер структуры вершин к указателю. Поняли? Если нет, не волнуйтесь, следующий ниже код поможет вам сделать это.

После того как вы заблокировали буферы, вы можете просматривать все вершины, получать их координаты и, используя вычисления из предыдущего раздела, рассчитывать положения морфированных вершин. Полагая, что продолжительность анимации хранится в Length, а текущее время в Time, следующий код иллюстрирует выполнение этих вычислений.

```
// Length = вещественное значение, содержащее длительность
// анимации в миллисекундах
// Time = вещественное значение, содержащее текущее время в анимации

// Вычислить скаляр, используемый в вычислениях
float Scalar = Time / Length;

// Перебрать все вершины
for(DWORD i=0;i<pSourceMesh->GetNumVertices();i++) {

  // Привести указатели буферов вершин к общей структуре вершин
  sGenericVertex *pSourceVertex =  (sGenericVertex*)pSourcePtr;
  sGenericVertex *pTargetVertex =  (sGenericVertex*)pTargetPtr;
  sGenericVertex *pResultVertex =  (sGenericVertex*)pResultPtr;

  // Получить исходные координаты и масштабировать их
  D3DXVECTOR3 vecSource = pSourceVertex->vecPos;
  vecSource *= (1.0f - Scalar);

  // Получить целевые координаты и масштабировать их
  D3DXVECTOR3 vecTarget = pTargetVertex->vecPos;
  vecTarget *= Scalar;

  // Сохранить сумму координат в результирующем морфированном меше
  pResultVertex->vecPos = vecSource + vecTarget;

  // Перейти к следующей вершине в каждом буфере
  pSourcePtr += SourceSize;
  pTargetPtr += TargetSize;
  pResultPtr += ResultSize;
}
```

До этого момента я пропускал тему нормалей вершин, потому что используемые для морфирования нормалей значения скаляра и обратного скаляра применяются таким же образом, как и для вершин.

В предыдущем коде вы можете вычислить морфируемые нормали, сначала проверив, существуют ли они у меша. Если да, то при обработке всех вершин возьмите нормали исходной и целевой вершин, умножьте их на скаляр и обратный скаляр и сохраните результат. Посмотрите на код, выполняющий эти действия.

```
// Length = вещественное значение, содержащее длительность
// анимации в миллисекундах
// Time = вещественное значение, содержащее текущее время в анимации
```

```
// Вычислить скаляр, используемый в вычислениях
float Scalar = Time / Length;

// Установить флаг, если используются нормали
BOOL UseNormals = FALSE;
if(pSourceMesh->GetFVF() & D3DFVF_NORMAL && \
   pTargetMesh->GetFVF() & D3DFVF_NORMAL)
  UseNormals = TRUE;

// Перебрать все вершины
for(DWORD i=0;i<pSourceMesh->GetNumVertices();i++) {

   // Привести указатель на буфер вершин к общей структуре вершин
  sGenericVertex *pSourceVertex = (sGenericVertex*)pSourcePtr;
  sGenericVertex *pTargetVertex = (sGenericVertex*)pTargetPtr;
  sGenericVertex *pResultVertex = (sGenericVertex*)pResultPtr;

  // Получить исходные координаты и масштабировать их
  D3DXVECTOR3 vecSource = pSourceVertex->vecPos;
  vecSource *= (1.0f - Scalar);

  // Получить целевые координаты и масштабировать их
  D3DXVECTOR3 vecTarget = pTargetVertex->vecPos;
  vecTarget *= Scalar;

  // Сохранить сумму координат в результирующем морфированном меше
  pResultVertex->vecPos = vecSource + vecTarget;

  // Обработать нормали, если флаг установлен
  if(UseNormals == TRUE) {
   // Настроить соответствующие указатели структур вершин, для
   // получения доступа к нормалям, которые следуют за координатами
   pSourceVertex++; pTargetVertex++; pResultVertex++;

   // Получить нормали и применить скаляр и обратный скаляр
   D3DXVECTOR3 vecSource = pSourceVertex->vecPos;
   vecSource *= (1.0f - Scalar);
   D3DXVECTOR3 vecTarget = pTargetVertex->vecPos;
   vecTarget *= Scalar;
   pResultVertex->vecPos = vecSource + vecTarget;
  }

  // Перейти к следующей вершине в каждом буфере и продолжить
  pSourcePtr += SourceSize;
  pTargetPtr += TargetSize;
  pResultPtr += ResultSize;
}
```

Все выглядит замечательно! Все, что вам остается сделать, - это разблокировать буферы вершин и визуализировать результирующий меш! Я пропущу код, разблокирующий буферы, и сразу перейду к визуализации мешей.

Визуализация морфированных мешей

Если вы создаете морфируемый меш, непосредственно работая с буфером вершин результирующего меша, как показано в предыдущем разделе, то его можно визуализировать также как и используемые вами объекты ID3DXMesh. Например, вы можете перебрать все материалы в меше, установить материал и текстуру и нарисовать текущий поднабор. Нет никакой нужды приводить код здесь - это простая визуализация меша.

С другой стороны, если вы хотите пропустить основы и начать использовать настоящую силу, вы можете создать собственный вершинный шейдер для визуализации морфируемых мешей. Поверьте мне на слово, вы обязательно захотите это сделать. При использовании вершинного шейдера вам потребуется на один меш меньше, потому что в результирующем меше больше нет надобности; в качестве дополнительного плюса можно отметить увеличение скорости визуализации.

Однако прежде чем переходить к рассмотрению использования вершинных шейдеров, необходимо выяснить, как самостоятельно визуализировать поднаборы меша.

Расчленение наборов

Для визуализации морфируемых мешей вам необходимо установить исходный и целевой потоки вершин. Также вам необходимо установить индексы только исходного меша. Это можно сделать простым перебором каждого набора и визуализированием их многоугольников.

Подождите-ка! Как визуализировать меш самостоятельно? Повторив то, что делает функция ID3DXMesh::DrawSubset, конечно же! Функция DrawSubset работает, используя один из двух методов. В первом режиме, применяемом, если меш не оптимизирован для использования таблицы атрибутов, просматривается весь список атрибутов и визуализируются те многоугольники, которые принадлежат одному поднабору. Этот метод может быть немного медленным, потому что он визуализирует меши, использующие множество материалов, маленькими наборами многоугольников.

Второй метод, применяемый после оптимизации меша для использования таблицы атрибутов, просматривает созданную таблицу атрибутов для определения, какие группы граней могут быть нарисованы одновременно. Таким образом, все грани принадлежащие одному поднабору сначала группируются, а потом визуализируются одним вызовом функции DrawPrimitive или DrawIndexedPrimitive. Похоже, это мы и будем использовать!

Чтобы использовать второй метод визуализации, необходимо сначала оптимизировать исходный меш. Вы можете (и должны) сделать это при загрузке меша. Можно безопасно оптимизировать все загруженные меши, используя функцию ID3DXMesh::OptimizeInPlace, как показано тут:

```
// pMesh = только что загруженный меш
pMesh->OptimizeInPlace(D3DXMESHOPT_ATTRSORT, \
       NULL, NULL, NULL, NULL);
```

После оптимизации меша вы можете получить таблицу атрибутов, используемую объектом ID3DXMesh. Таблица атрибутов имеет тип D3DXATTRIBUTERANGE, который определен так:

```
typedef struct _D3DXATTRIBUTERANGE {
  DWORD AttribId;
  DWORD FaceStart;
  DWORD FaceCount;
  DWORD VertexStart;
  DWORD VertexCount;
} D3DXATTRIBUTERANGE;
```

Первая переменная AttribId является номером поднабора, который представляет структура. Каждый материал меша ассоциирован с одной структурой D3DXATTRIBUTERANGE, AttribId которой указывает номер поднабора.

Далее идет FaceStart и FaceCount. Эти две переменные используются для определения, какие грани принадлежат поднабору. Вот где и появляется оптимизация - все грани, принадлежащие одному и тому же поднабору, группируются в буфере индексов. FaceStart представляет собой первую грань, принадлежащую поднабору, в то время как FaceCount содержит количество граней, визуализируемых в данном поднаборе.

Наконец вы обнаружите VertexStart и VertexCount, которые совсем как FaceStart и FaceCount определяют, какие вершины используются при визуализации многоугольников. VertexStart представляет собой первую вершину из буфера вершин, используемую в поднаборе, а VertexCount содержит количество вершин, визуализируемых за один раз. Когда вы оптимизируете меш, основываясь на вершинах, вы заметите, что все вершины запакованы в буфере для уменьшения количества вершин, используемых при визуализации поднабора.

Для каждого поднабора вашего меша существует соответствующая структура D3DXATTRIBUTERANGE. Таким образом, меш, использующий три материала, имеет три структуры атрибутов. После оптимизации меша (используя ID3DXMesh::OptimizeInPlace) вы можете получить таблицу атрибутов, сначала

взяв из объекта меша количество структур атрибутов, используя функцию ID3DXMesh::GetArtribute-Table, как показано тут:

```
// Получить количество атрибутов в таблице
DWORD NumAttributes;
pMesh->GetAttributeTable(NULL, &NumAttributes);
```

Теперь вам необходимо просто создать необходимое количество объектов D3DXATTRIBUTERANGE и вызвать функцию GetAttributeTable, указав, на этот раз, указатель на массив объектов атрибутов.

```
// Выделить память под таблицу атрибутов и получить ее данные
D3DXATTRIBUTERANGE *pAttributes;
pAttributes = new D3DXATTRIBUTERANGE[NumAttributes];
pMesh->GetAttributeTable(pAttributes, NumAttributes);
```

Замечательно! После того как вы получили данные атрибутов, вы можете визуализировать поднаборы, просматривая каждый объект таблицы атрибутов и используя данные каждого из них при вызове DrawIndexedPrimitive. На самом деле вам необходимо сначала получить указатели на буферы вершин и индексов меша.

```
// Получить интерфейс буфера вершин
IDirect3DVertexBuffer9 *pVB;
pMesh->GetVertexBuffer(&pVB);

// Получить интерфейс буфера индексов
IDirect3DIndexBuffer9 *pIB;
pMesh->GetIndexBuffer(&pIB);
```

После того как вы получили оба указателя, вы можете смело устанавливать потоки, вершинные шейдеры и объявления элементов вершин, просматривать каждый набор, устанавливая текстуру, после чего визуализировать многоугольники.

```
// Установить вершинный шейдер и объявления
pDevice->SetFVF(NULL); // Clear FVF usage
pDevice->SetVertexShader(pShader);
pDevice->SetVertexDeclaration(pDecl);

// Установить потоки
pDevice->SetStreamSource(0, pVB, \
         0, pMesh->GetNumBytesPerVertex());
pDevice->SetIndices(pIB);

// Просмотреть каждый поднабор
for(DWORD i=0;i<NumAttributes;i++) {
  // Получить номер материала
  DWORD MatID = pAttributes[i];

  // Установить текстуру поднабора
  pDevice->SetTexture(0, pTexture[AttribID]);
```

```
    // Визуализировать многоугольник, используя таблицу
    pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, 0, \
              pAttributes[i].VertexStart, \
              pAttributes[i].VertexCount, \
              pAttributes[i].FaceStart * 3, \
              pAttributes[i].FaceCount);
}
```

После того как вы визуализировали ваши поднаборы, вы можете освободить полученные ранее интерфейсы буферов вершин и индексов.

```
pVB->Release(); pVB = NULL;
pIB->Release(); pIB = NULL;
```

Когда вы закончите работу с таблицей атрибутов, убедитесь, что тоже освободили ее.

```
delete [] pAttributes; pAttributes = NULL;
```

Замечательно, вы уже кое-что знаете! После того как вы узнали, как самостоятельно визуализировать поднаборы, пришло время перейти к использованию вершинных шейдеров.

Создание морфирующего вершинного шейдера

Такая простая вещь как морфирование мешей требует использования вершинных шейдеров. На самом деле, как только вы увидите насколько просто использовать морфируемые вершинные шейдеры, вы никогда не будете вручную изменять буферы вершин!

Помните, что по сути морфируемый меш состоит из интерполированных координат положения и нормалей, которые вычисляются из исходного и целевого меша. Т. к. координаты этих положений и нормалей являются частью потока вершин, вы можете создать вершинный шейдер, который бы одновременно использовал эти два потока вершин и то же самое значение скаляра, что и раньше, и вычислял бы значения вершин при помощи простых команд.

Все правильно, больше никаких блокировок или перестроений морфируемого меша в каждом кадре. Все происходит в вершинном шейдере! Вам просто необходимо установить источники потоков вершин. Используя методы визуализации, рассмотренные в предыдущем разделе, вы можете визуализировать группы многоугольников, принадлежащих мешу.

Замечание. *Во вспомогательных функциях первой главы вы могли обнаружить функцию DrawMesh, которая использует вершинный шейдер и интерфейс объявления вершин для визуализации меша. Установив целевой меш в первый поток, вы можете вызвать DrawMesh для визуализации морфируемого меша, совсем как в демонстрационной программе вершинного шейдера морфируемого меша этой главы. Для получения дополнительной информации о расположении демонстрационной программы смотрите конец этой главы. Что же касается оставшейся части этой главы, я покажу вам, как визуализировать меш, используя чистый код, — никаких вспомогательных функций!*

Давайте не будем больше тянуть и перейдем к вершинным шейдерам. Начнем с объявления элементов вершин, которые ассоциируют данные вершин с регистрами шейдера.

```
D3DVERTEXELEMENT9 MorphMeshDecl[] =
{
 // первый поток используется для исходного меша
 { 0 , 0 , D3DDECLTYPE_FLOAT3 , D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_POSITION , 0 },
 { 0 , 12 , D3DDECLTYPE_FLOAT3 , D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_NORMAL , 0 },
 { 0 , 24 , D3DDECLTYPE_FLOAT2 , D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_TEXCOORD , 0 },

 // второй поток используется для целевого меша
 { 1 , 0 , D3DDECLTYPE_FLOAT3 , D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_POSITION , 1 },
 { 1 , 12 , D3DDECLTYPE_FLOAT3 , D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_NORMAL , 1 },
 { 1 , 24 , D3DDECLTYPE_FLOAT2 , D3DDECLMETHOD_DEFAULT,
D3DDECLUSAGE_TEXCOORD , 1 },
 D3DDECL_END()
};
```

Как вы можете видеть из объявления элементов вершин, используются два потока. Каждый поток использует набор положений, нормалей и текстурных координат. Первый поток имеет индекс использования 0, в то время как второй - 1.

При установке потоков вершин вы устанавливает буфер вершин исходного меша в качестве потока 0, а буфер вершин целевого меша в качестве потока 1. При этом используется только один буфер индексов, буфер исходного меша, который устанавливается при помощи IDirect3DDevice9::SetIndices. Убедитесь, что исходный и целевой меши используют одинаковый порядок индексов и вершин, иначе морфируемая анимация может искажаться.

Я вернусь к этим потокам чуть позже; а пока я хочу перейти к коду шейдера. В начале вершинного шейдера вы обнаружите версию и привязочные объявления.

```
vs.1.0
; declare mapping
dcl position v0
```

```
dcl_normal v1
dcl_texcoord v2
dcl_position1 v3
dcl_normal1 v4
```

Вы можете видеть, что используются только пять вершинных регистров, а где же шестой? Т. к. оба меша используют одни и те же текстурные координаты, второй набор может быть полностью опущен. Все к лучшему, меньше надо будет делать. Используемые регистры непосредственно привязываются к объявлениям элементов вершин. В объявлении находятся координаты положения исходного меша (v0), нормаль (v), текстурные координаты (v2), координаты положения целевого меша (v3) и его нормаль (v4).

Двигаясь дальше с вершинными шейдерами, вы вычисляете координаты положения вершины, сначала умножая v0 на значение обратного скаляра, находящегося в регистре констант вершинного шейдера c4.x. Помните, что значение этого скаляра изменяется от 0 до 1, при этом 0 означает, что координаты вершины совпадают с координатами исходного меша, а 1 означает, что ее координаты совпадают с координатами целевого меша.

Далее вам необходимо умножить координаты вершины, хранящейся в v3, на значение скаляра, хранящегося в регистре констант вершинного шейдера c4.y. Вот код вершинного шейдера, выполняющий эти вычисления.

```
; применить значения скаляра и обратного скаляра к координатам
mul r0, v0, c4.x
mad r0, v3, c4.y, r0
```

Заметьте, что вы сначала умножаете v0 на обратный скаляр, после чего умножаете v3 на скаляр. Оба полученных результата складываются и сохраняются во временном регистре, который нужен для проецирования, используя транспонированное комбинированное преобразование мир*вид*проекция. Я вернусь к этому чуть позже; а пока, необходимо обработать значения нормалей так же, как вы делали это с координатами вершин.

```
; применить значения скаляра и обратного скаляра к нормалям
mul r1, v1, c4.x
mad r1, v4, c4.y, r1
```

Все что осталось сделать, это преобразовать координаты при помощи транспонированной матрицы преобразования мир*вид*проекция (хранимой в константах вершинного шейдера с c0 до c3), векторно умножить нормаль на обратное направление света (то же самое направление, которое вы используете в структуре D3DLIGHT9), хранимое в константе вершинного шейдера c5, и сохранить текстурные координаты (из регистра v2).

```
; Спроецировать положение
m4x4 oPos, r0, c0

; Векторно умножить нормаль на обратное направление света, для
; получения рассеянного света
dp3 oD0, r1, -c5

; сохранить текстурные координаты
mov oT0.xy, v2
```

Видите, я же говорил, что вершинный шейдер прост. Предположив, что у вас уже есть код загрузки вершинного шейдера, перейдем к тому, как визуализировать меш, используя вершинный шейдер.

Первым шагом к визуализированию меша при помощи вершинного шейдера является установка регистров констант. Ранее в этом разделе вы увидели, что этими константами являются преобразование мир*вид*проекция, направление света и значения скаляров морфинга. Предположим, вы храните преобразования мира, вида и проекция в трех матрицах:

```
// matWorld = матрица преобразования мира  (D3DXMATRIX)
// matView = матрица преобразования вида  (D3DXMATRIX)
// matProj = матрица преобразования проекции(D3DXMATRIX)
```

В случае, если вы не хотите следить за этими тремя матрицами, вы можете получить их при помощи функции IDirect3DDevice9::GetTransform, как показано тут.

```
pDevice->GetTransform(D3DTS_WORLD, &matWorld);
pDevice->GetTransform(D3DTS_VIEW, &matView);
pDevice->GetTransform(D3DTS_PROJECTION, &matProj);
```

После того как вы получили эти преобразования, вы можете скомбинировать их и установить транспонированную матрицу в константы от c0 до c3 вершинного шейдера, как показано далее.

```
D3DXMATRIX matWVP = matWorld * matView * matProj;
D3DXMatrixTranspose(&matWVP, &matWVP);
pDDevice->SetVertexShaderConstantF(0, (float*)&matWVP, 4);
```

После этого установите значение обратного скаляра морфинга в c4.x и значения скаляра в c4.y.

```
// Установить используемые значения скаляра и обратного скаляра
D3DXVECTOR4 vecScalar = D3DXVECTOR4(1.0f - Scalar, \
            Scalar, \
            0.0, 0.0);
pDevice->SetVertexShaderConstantF(4, (float)*&Scalar, 1);
```

Наконец, установите нормализованное направление света в регистр констант c5 вершинного шейдера. Это значение направления является направленным вектором, хранимым в структуре D3DLIGHT, которую вы уже должны были использовать, хотя этот вектор приводится к типу D3DXVECTOR4.

```
// Установить направление света в регистр констант c5
D3DXVECTOR4 vecLight(0.0f, -1.0f, 0.0f, 0.0f);
pDevice->SetVertexShaderConstantF(5, (float*)&vecLight, 1);
```

Заметьте, что координаты света хранятся в пространстве объекта, означая, что свет вращается вместе с объектом. Это необходимо для того, чтобы объект освещался корректно, независимо от направления просмотра. Если вы хотите использовать координаты света в пространстве вида, а не объекта, тогда вам необходимо преобразовать направление света на обратное преобразование вида.

Гмм! Небольшая тема растянулась на много, и теперь вы готовы визуализировать меш при помощи вершинных шейдеров. На данный момент, остается только установить потоки вершин, текстуру, шейдер, объявление вершин и вызвать функцию рисования элементарных объектов. Конечно вам необходимо использовать методы из предыдущего раздела для визуализации отдельных поднаборов исходного меша. Вместо того чтобы повторять код, который вы уже видели, я посоветую вам посмотреть код демонстрационной программы вершинного шейдера для этой главы. Счастливого морфинга!

Посмотрите демонстрационные программы

Как вы можете видеть, морфинг является привлекательной анимационной технологией, которую вы можете внедрить в ваши проекты, затратив минимум усилий. Для этой главы я создал две демонстрационные программы (Morphing и MorphingVS), которые иллюстрируют морфирование, используя технологии изменения буфера вершин и вершинного шейдера.

Эти две программы, в основном, выполняют одно и то же. При запуске любой из них вы увидите экран анимации (показанный на рис. 8.4), на котором дельфин прыгает через анимированные волны океана. Обе программы выполняются, пока вы не выйдите из них.

Программы на компакт диске

Два проекта этой главы иллюстрируют использование морфируемых мешей. Эти два проекта расположены в директории главы 8 компакт-диска книги:

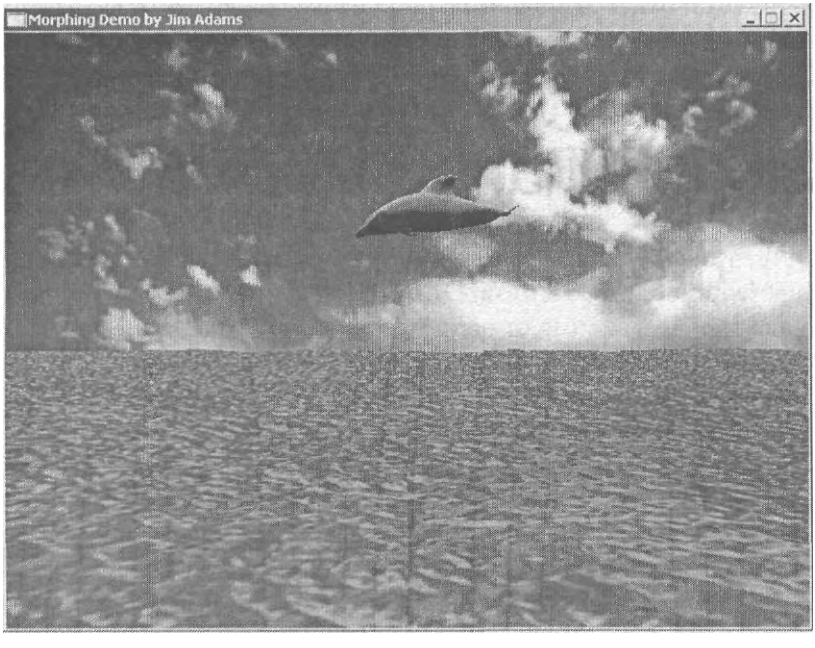


Рис. 8.4. Анимированный дельфин прыгает через волны моря! Оба объекта (дельфин и море) используют технологии морфирующей анимации

- **Morphing.** Этот проект иллюстрирует создание морфируемых мешей непосредственным изменением буфера вершин меша. Он расположен в \BookCode\Chap08\Morphing.
- **MorphingVS.** В этом проекте вы можете узнать, как использовать вершинные шейдеры для экономии памяти и ускорения визуализации. Он расположен в \BookCode\Chap08\MorphingVS.

Глава 9

Использование морфирующей анимации, основанной на ключевых кадрах

Заранее вычисленные анимации являются основой современных игровых движков. Приложив небольшие усилия, аниматор.с помощью популярных программ трехмерного моделирования может создать полноценные последовательности анимаций и экспортировать их данные в формате, используемом движком игры. Программист может изменять или модифицировать эти анимации, не переписывая кода игры.

Я знаю, что вы видели множество таких игр и что вы хотите иметь возможность использовать заранее вычисленные анимации в своих игровых проектах. Эта глава как раз то, что вам нужно!

В этой главе вы научитесь:

- Работать с ключевыми кадрами в анимации;
- Загружать данные морфируемой анимации из .X файлов;
- Работать с наборами морфируемой анимации;
- Преобразовывать файлы .MD2 в .X.

Использование наборов морфируемой анимации

Если вы, хотите использовать морфирующую анимации в своих проектах, вам необходимо разработать средство использования наборов морфируемой анимации. В отличие от наборов скелетной анимации, наборы морфируемой анимации очень просты в использовании. Посмотрите рис. 9.1, на котором вы можете увидеть общую последовательность морфируемой анимации и ее порядок.

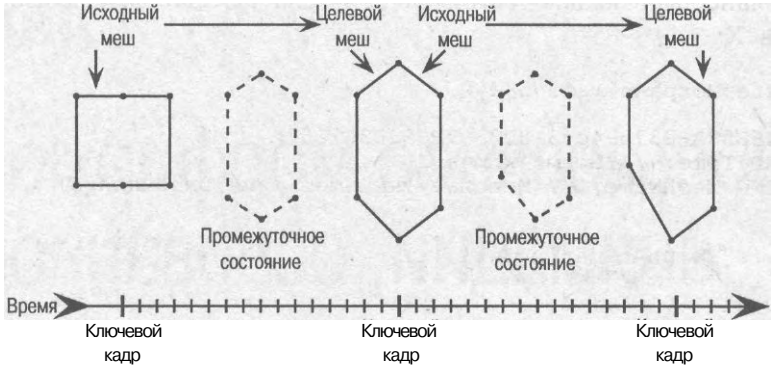


Рис. 9.1. Морфирующая анимация использует набор исходных и целевых морфируемых мешей, разделенных во времени (при помощи ключей анимации), образующих непрерывную морфируемую анимацию

Официально DirectX не содержит шаблонов наборов морфируемой анимации, но это нас не остановит. Вам просто необходимо создать собственные специализированные шаблоны, и все будет в порядке.

Создание шаблонов .X для морфируемой анимации

Данные морфируемой анимации являются чуждыми формату файла .X, они не существуют официально. По этой причине пришло время напрячь мозги и изобрести собственный шаблон данных морфируемой анимации. Выбросите из головы видения истерического хохота злобных ученых при мысли о создании дикого шаблона данных анимации, вам просто необходимо сделать несколько простых шагов для его создания.

Если вы помните, в главе 1 упоминалось, что морфирующая анимация требует двух мешей - исходного и целевого. Значение времени используется для морфинга исходного меша в целевой. Задав диапазон значений времени (ключевых кадров), вы можете использовать интерполяцию для морфинга исходного меша в целевой, который вы визуализируете. Это означает, что вам необходимо определить шаблон ключевого кадра морфируемой анимации, который бы содержал значение времени и меше, используемыи в качестве целевого и исходного (для последующих и предыдущих ключей) для морфинга. Для каждого ключа определяется один меш. Если перед ключом есть другой ключ, то текущий меш является целевым при морфинге. Если после ключа есть еще ключи, то текущий меш определяет исходный меш при морфинге.

Следующие два шаблона должны подойти для хранения данных морфируемой анимации в .X:

```
templateMorphAnimationKey
{
  <2746B58A-B375-4cc3-8D23-7D094D3C7C67>
  DWORD Time; // Время ключа
  STRING MeshName; // Используемый меш (имя экземпляра)
}
templateMorphAnimationSet
{
  <0892DE81-915A-4f34-B503-F7C397CB9E06>
  DWORD NumKeys; // # ключей в анимации
  array MorphAnimationKey Keys[NumKeys];
}
```

Конечно же, каждому шаблону необходимо соответствующее объявление GUID в исходном коде, так что добавьте следующий код в начало основного исходного файла (или куда сочтете нужным):

```
// {2746B58A-B375-4cc3-8D23-7D094D3C7C67}
DEFINE_GUID(MorphAnimationKey,
       0x2746b58a, 0xb375, 0x4cc3,
       0x8d, 0x23, 0x7d, 0x9,
       0x4d, 0x3c, 0x7c, 0x67);

{0892DE81-915A-4f34-B503-F7C397CB9E06}
DEFINE_GUID(MorphAnimationSet,
       0x892de81, 0x915a, 0x4f34,
       0xb5, 0x3, 0xf7, 0xc3,
       0x97, 0xcb, 0x9e, 0x6);
```

Комментарии в каждом из двух шаблонов говорят сами за себя, но давайте рассмотрим их подробнее. В MorphAnimationKey определены две переменные - Time, содержащая значение времени в анимации (время ключа, обычно задаваемое в миллисекундах), и MeshName, типа STRING, которая содержит имя меша, используемого при морфинге.

Замечание. *Вы заметите, что шаблон MorphAnimationKey определяет имя меша как тип STRING, а не ссылку на шаблон. Это означает, что вам необходимо сопоставить имя меша с ключом анимации после загрузки ключей морфирующей анимации.*

Что же касается шаблона MorphAnimationSet, он содержит всего две переменные. Первая их них, NumKeys, является количеством объектов MorphAnimationKeys, содержащихся в анимации. Вторая переменная, Keys, является массивом, содержащим данные каждого объекта MorphAnimationKey, используемого в анимации.

После того как мы объявили шаблоны, давайте посмотрим на то, как использовать их для хранения данных морфируемой анимации. В качестве небольшого примера использования созданных шаблонов определим простую анимацию. В начале необходимо задать меши, с которыми будем работать:

```
Mesh MyMesh1 {
  // Здесь располагаются данные меша
}
Mesh MyMesh2 {
  // Здесь располагаются данные меша
}
Mesh MyMesh3 {
  // Здесь располагаются данные меша
}
```

Для упрощения я не стал приводить данные заданных мешей, а просто показал вам как создавать объекты Mesh для дальнейшего использования. Эти три объекта Mesh называются MyMesh1, MyMesh2, MyMesh3. Для данных анимации я хочу определить две анимации, названные MyAnimation1 и MyAnimation2.

```
MorphAnimationSet MyAnimation1
{
  2;
    0; "MyMesh1";,
500; "MyMesh2";,
}

MorphAnimationSet MyAnimation2
{
  4;
    0; "MyMesh1";,
  500; "MyMesh2";,
 1000; "MyMesh3";,
 1500; "MyMesh2";;
}8,8
```

Первая анимация, MyAnimation1, содержит два ключа. Первый ключ расположен в нулевом моменте времени и использует MyMesh1 в качестве исходного меша. Второй ключ, расположенный во время 500, использует MyMesh2 в качестве целевого меша. Таким образом анимация длится 500 временных единиц.

Вторая анимация, MyAnimation2, содержит четыре ключа. Эти ключи, расположенные в 500 единицах измерения времени друг от друга, используют все три меша. Если вы хотите обновить анимацию в момент времени 700, вам необходимо исполь-

зовать MyMesh2 в качестве исходного меша и MyMesh3 в качестве целевого меша в морфируемой анимации. Значение скаляра, необходимого для морфинга, вычисляется на основе значений времени ключей.

Хорошо, это все просто для понимания. А вот часть, в которой вы фактически загружаете данные анимации, может быть немного тяжела.

Загрузка данных морфируемой анимации

После того как вы определили шаблоны морфируемой анимации, вы можете загружать данные анимации и использовать их. Сначала необходимо определить три класса, которые соответствовали бы данным структур шаблонов, содержащих ключи анимации, наборы и коллекции наборов анимаций.

```
class cMorphAnimationKey
{
  public:
   DWORD m_Time; // Время ключа
   char *m_MeshName; // Имя используемого меша
   D3DXMESHCONTAINER_EX *m_MeshPtr; // Указатель на данные меша

  public:
   cMorphAnimationKey()
   {
    m_MeshName = NULL;
    m_MeshPtr = NULL;
   }

   ~cMorphAnimationKey()
   {
    delete [] m_MeshName;
    m_MeshName = NULL;
    m_MeshPtr = NULL;
  }
};

class cMorphAnimationSet
{
  public:
   char *m Name;  // Название анимации
   DWORD m_Length;  // Длительность анимации
   cMorphAnimationSet *m_Next; // Следующая анимация в связанном
списке
   DWORD m_NumKeys;  // # ключей анимации
   cMorphAnimationKey *m_Keys;  // массив ключей

  public:
   cMorphAnimationSet()
   {
```

```
    m_Name = NULL;
    m_Length = 0;
    m_Next = NULL;
    m_NumKeys = 0;
    m_Keys = NULL;
   }

   ~cMorphAnimationSet()
   {
    delete [] m_Name; m_Name = NULL;
    m_Length = 0;
    m_NumKeys = 0;
    delete [] m_Keys; m_Keys = NULL;
    delete m_Next; m_Next = NULL;
  }
 };

 class cMorphAnimationCollection : public cXParser
 {
  protected:
   DWORD m_NumAnimationSets; // # наборов анимаций
   cMorphAnimationSet *m_AnimationSets; // наборы анимаций
  protected:
    Анализировать .X файл, искать данные пружин и масс
   BOOL ParseObject(IDirectXFileData *pDataObj,
        IDirectXFileData *pParentDataObj,
        DWORD Depth,
        void **Data, BOOL Reference);

  public:
   cMorphAnimationCollection()
   {
    m_NumAnimationSets = 0;
    m_AnimationSets = NULL;
   }

   ~cMorphAnimationCollection()  { Free(); }

   BOOL Load(char *Filename);
   void Free();

   void Map(D3DXMESHCONTAINER_EX *RootMesh);
   void Update(char *AnimationSetName, \
        DWORD Time, BOOL Loop, \
        D3DXMESHC0NTAINER_EX **ppSource, \
        D3DXMESHCONTAINER_EX **ppTarget, \
        float *Scalar);
};
```

Первый приведенный класс, cMorphAnimationKey, хранит время ключа, имя меша и указатель на объект меша (который вы должны установить после загрузки всех мешей из .X файла).

Класс cMorphAnimationSet содержит массив объектов cMorphAnimationKey, которые образуют анимацию. Каждый класс cMorphAnimationSet содержит буфер имени, который заполняется соответствующими данными объекта набора анимации .X файла. Это означает, что вы можете задавать неограниченное количество анимаций (хранящихся в объектах cMorphAnimationSet), имеющих собственные уникальные имена.

Этот список объектов наборов анимации хранится в виде связанного списка, к которому вы получаете доступ при помощи указателя cMorphAnimationSet::m_Next. (Один набор анимации указывает на следующий набор в списке.) Весь список объектов наборов анимации хранится в объекте cMorphAnimationCollection, который содержит только указатель на корневой объект набора анимации. Для получения доступа к любому объекту набора анимации вам необходимо просмотреть все объекты наборов анимации в поисках заданного.

Вы должны быть знакомы с членами объекта cMorphAnimationSet (за исключением переменной m_Length, которая содержит длительность анимации, определяемой временем последнего ключа в массиве объектов cMorphAnimationKey). Вместо того чтобы просматривать весь массив ключей, для определения длительности анимации будет использоваться m_Length.

В то время как класс cMorphAnimationSet содержит одну анимацию, cMorphAnimationCollection ответственен за хранение набора объектов cMorphAnimationSet (при помощи вышеупомянутого связанного списка указателей).

Для загрузки анимаций морфируемого меша вам необходимо унаследовать класс cMorphAnimationSet от cXParser, разработанного в главе 3. Наследование от класса cXParser предоставляет вам возможность получать доступ к функции ParseObject, которую вы будете использовать для получения данных анимации из объекта MorphAnimationSet.

Однако нет никакой нужды напрямую вызывать ParseObject, потому что есть функция Load, которая сделает это сама. Вам просто необходимо вызвать функцию Load, задав в качестве параметра имя .X файла, из которого вы хотите загрузить данные наборов морфируемых анимаций. После того как вы закончили работу с данными анимации, вызов функции Free освобождает все ресурсы, использованные для хранения данных анимации.

В ближайшем рассмотрении конструктор, деструктор, Load и Free являются тривиальными функциями; я предоставляю вам возможность посмотреть самим на их код. Однако вас может заинтересовать код функции ParseObject. В принципе, необхо-

димо, чтобы функция ParseObject искала экземпляры объектов MorphAnimationSet и создавала объекты cMorphAnimationSet, содержащие данные анимации. Это может быть реализовано посредством следующего кода:

```
BOOL cMorphAnimationCollection::ParseObject( \
      IDirectXFileData *pDataObj, \
      IDirectXFileData *pParentDataObj, \
      DWORD Depth, \
       void **Data, BOOL Reference)
{
  const GUID *Type = GetObjectGUID(pDataObj);

  // Прочитать данные набора анимации
  if(*Type == MorphAnimationSet) {

   // Создать и привязать объект cMorphAnimationSet
   cMorphAnimationSet *AnimSet = new cMorphAnimationSet();
   AnimSet->m_Next = m_AnimationSets;
   m_AnimationSets = AnimSet;

   // Увеличить # наборов анимации
   m_NumAnimationSets++;

   // Установить имя набора анимации
   AnimSet->m_Name = GetObjectName(pDataObj);

   // Получить указатель на данные
   DWORD *Ptr = (DWORD*)GetObjectData(pDataObj, NULL);

   // Получить # ключей и создать массив объектов ключевых кадров
   AnimSet->m_NumKeys = *Ptr++;
   AnimSet->m_Keys = new cMorphAnimationKey[AnimSet->m_NumKeys];

   // Получить данные ключа - время и имя меша
   for(DWORD i=0;i<AnimSet->m_NumKeys;i++) {
    AnimSet->m_Keys[i].m_Time = *Ptr++;
     AnimSet->m_Keys[i].m_MeshName = strdup((char*)*Ptr++);
   }

   / Сохранить длительность анимации
   AnimSet->m_Length = AnimSet->m_Keys[AnimSet->m_NumKeys-1].m_Time;
  }

  return ParseChildObjects(pDataObj, Depth, Data, Reference);
}
```

Единственным критическим местом в функции cMorphAnimationCollection::ParseObject является загрузка объекта набора морфируемых анимаций. После создания экземпляра объекта cMorphAnimationSet и привязывания его в связанный список объектов вам необходимо считать количество ключей, используемых в ани-

мации. После считывания количества ключей код циклически считывает время и имя меша каждого ключа. По окончании цикла высчитывается полная продолжительность анимации (используя значение времени последнего ключа).

После того как вы загрузили все наборы анимаций из .X файла, необходимо связать соответствующие объекты мешей с объектами ключей анимации. Это необходимо для получения быстрого доступа к мешу, визуализируемому объектом набора анимаций.

Для сопоставления мешей наборам анимаций создайте функцию, просматривающую каждый набор анимации. Для каждого ключа в наборе просматривайте список мешей в поисках меша, имеющего то же самое имя, какое хранится в ключе. Именно этой цели и служит функция cMorphAnimationCollection::Map.

```
void cMorphAnimationCollection::Map( \
           D3DXMESHCONTAINER_EX *RootMesh)
{
  // Проверка ошибок
  if(!RootMesh)
   return;

  // Перебрать все анимации
  cMorphAnimationSet *AnimSet = m_AnimationSets;
  while(AnimSet != NULL) {

   // Перебрать все ключи в наборе анимаций и сопоставить меши
   // указателю на меш ключа
   if(AnimSet->m_NumKeys) {
    for (DWORD i=0;i<AnimSet->m_NumKeys;i++)
    AnimSet->m_Keys[i].m_MeshPtr = \
        RootMesh->Find(AnimSet->m_Keys[i].m_MeshName);
   }

  // Перейти к следующему объекту набора анимаций
  AnimSet = AnimSet->m_Next;
 }
}
```

В функции Map я использовал функцию D3DXMESHCONTAINER_EX::Find для перебора всех мешей, содержащихся в объекте, и поиска заданного имени меша. Если такой меш найден, указатель на него сохраняется в объекте ключа.

Когда наступает время визуализировать набор морфируемых анимаций, вы быстро просматриваете ключи, находите указатель меша и визуализируете его. Эти же самые последовательности действия могут быть использованы для визуализирования ваших собственных морфируемых анимаций. Просто переходите со мной к следующему разделу, и вы сможете визулизировать меш в мгновение ока!

Визуализации морфированного меша

После того как вы загрузили данные набора анимаций морфируемого меша, визуализировать его просто. Используя время из анимации, вы можете просмотреть массив ключей анимации и найти два из них, между которыми оно находится. Первый ключ содержит указатель на исходный меш, используемый при морфинге, а второй ключ содержит указатель на целевой меш.

Т. к. вы можете загружать неограниченное количество анимаций в объект коллекции, вам необходимо создать функцию, которая бы искала заданную именем анимацию, после чего просматривала бы эту анимацию на соответствующие значения ключей для визуализации. Это делает функция cMorphAnimationCollection::Update.

```
void cMorphAnimationCollection::Update( \
      char *AnimationSetName, \
      DWORD Time, BOOL Loop, \
      D3DXMESHCONTAINER_EX  **ppSource, \
      D3DXMESHCONTAINER_EX **ppTarget, \
      float *Scalar)
```

Хотя функция Update и кажется большой, на самом деле она очень проста в использовании. В качестве параметра AnimationSetName вам необходимо указать имя используемого набора анимации, в качестве параметра Time указать время анимации и повторять ли ее циклически или нет (установите Loop в TRUE или FALSE).

Также вам необходимо предоставить два указателя (ppSource и ppTarget) на объекты исходного и целевого мешей, используемые при морфируемой визуализации, и указатель на переменную, в которую будет помещено значение скаляра, используемого при морфинге.

Функция Update начинается с получения указателя на связанный список набора анимаций и очищения указателей, переданных функции.

```
{
  cMorphAnimationSet *AnimSet = m_AnimationSets;
  // Очистить возвращаемые значения
  *ppSource = NULL;
  *ppTarget = NULL;
  *Scalar = 0.0f;
```

Указатели очищаются на случай возникновения ошибки. Если возникает ошибка, вы можете проверить, установлены ли указатели в NULL. Если они имеют значение отличное от NULL, то это означает, что функция Update отработала корректно.

Двигаясь дальше в функции, вы просматривает список наборов анимаций в поисках той, которая имеет такое же имя, как и заданное в параметре AnimationSetName. Вы можете вынудить функцию Update использовать первый набор анимации в связанном списке, установив AnimationSetName в NULL.

Название анимации должно совпадать с названием, хранимым в .X файле. Название из .X файла является именем экземпляра объекта набора анимации. Например, следующий объект набора анимации имеет имя экземпляра Walk:

```
MorphAnimationSet Walk
{
  2;
    0; "Figure1";,
  500; "Figure2";;
}
```

Для того чтобы получить указатели на меши Figure1 и Figure2, вам необходимо задать Walk в качестве AnimationSetName. Я думаю вы поняли смысл, так что вот код, который просматривает список наборов анимации (прекращая работу функции, если не было найдено набора или если найденный набор не содержит ключей анимации).

```
// Искать заданное имя набора анимации, если используется
if(AnimationSetName) {

  // Найти заданное имя набора анимации
  while(AnimSet !=NULL) {

   // Остановится, если совпадение найдено
   if(!stricmp(AnimSet->m_Name, AnimationSetName))
    break;

   // Перейти к следующему объекту набора анимации
   AnimSet = AnimSet->m_Next;
  }
  }

  // Прекратить, если набор не был найден
  if(AnimSet == NULL)
    return;

  // Прекратить работу, если в наборе нет ключей
  if(!AnimSet->m_NumKeys)
   return;
```

На данный момент вы имеете указатель на объект cMorphAnimationSet, который содержит данные ключей, используемые при морфированной анимации. Используя этот указатель (AnimSet), вы можете проверить, попадает ли время анимации, для которого вы получаете данные ключа, в фактическую продолжительность анимации (хранимой в переменной m_Length набора анимации).

```
// Сравнить время с продолжительностью анимации
if(Time > AnimSet->m_Length)
 Time =  (Loop==TRUE)?Time%(AnimSet->m_Length+1):AnimSet->m_Length;
```

После этого вы можете просмотреть каждый ключ в наборе анимации и определить, какие использовать для получения указателей на исходный и целевой меши. Вы видели как просматривать ключи в главе 5, так что я пропущу объяснения и сразу перейду к коду.

```
// Просмотреть весь набор анимации и найти используемые ключи
DWORD Key1 = AnimSet->m_NumKeys-1;
DWORD Key2 = AnimSet->m_NumKeys-1;
for(DWORD i=0;i<AnimSet->m_NumKeys-1;i++) {
  if(Time >= AnimSet->m_Keys[i].m_Time && \
   Time < AnimSet->m_Keys[i+1].m_Time) (

   // Нашли ключи, устанавливаем указатели и прерываем цикл
   Key1 = i;
   Key2 = i+1;
   break;
 }
}
```

Теперь, когда мы нашли ключи, используемые в морфированной анимации (хранящиеся как Key1 и Key2), можно вычислять значение скаляра морфирования на основе времени этих двух ключей.

```
// Вычислить используемое значение временного скаляра
DWORD Key1Time = AnimSet->m_Keys[Key1].m_Time;
DWORD Key2Time = AnimSet->m_Keys[Key2].m_Time;
float KeyTime =  (float)(Time - Key1Time);
float MorphScale = 1.0f/(float)(Key2Time-Key1Time)*KeyTime;
```

Наконец вы можете установить указатели на целевой, исходный меши и скаляр (которые вы передавали в качестве параметров функции Update) в соответствующие значения.

```
// Установить указатели
*ppSource = AnimSet->m_Keys[Key1].m_MeshPtr;
*ppTarget = AnimSet->m_Keys[Key2].m_MeshPtr;
*Scalar = MorphScale;
}
```

Теперь, когда все классы и функции определены, можно возвращаться к работе! Посмотрите на пример использования только что созданных классов для загрузки и использования коллекции морфированных анимаций из файла MorphAnim.x. Сначала необходимо создать экземпляр объекта cAnimationCollection и загрузить последовательность наборов анимации.

```
cMorphAnimationCollection MorphAnim;
MorphAnim.Load("MorphAnim.x");
```

Также необходимо загрузить меши. Вы можете использовать удобные вспомогательные функции, разработанные в главе 1, для загрузки набора мешей (из упомянутого ранее файла MorphAnim.x).

```
D3DXMESHCONTAINER_EX *Meshes;
LoadMesh(SMeshes, NULL, pDevice, "MorphAnim.x");
```

После того как меши и наборы анимации были загружены, вам необходимо сопоставить наборы анимации мешам при помощи функции cMorphAnimationCollection::Map.

```
MorphAnim.Map(MorphAnim);
```

После этого вы можете использовать функцию Update для получения исходного, целевого мешей и значения скаляра, необходимых для создания последовательной морфируемой анимации. Возвращаясь к тому разделу, где создавались шаблоны наборов морфированных анимаций, положим, что имеется два набора анимации - MyAnimation1 и MyAnimation2.

Мы будем использовать анимацию MyAnimation2, и необходимо определить, как обновить ее в момент времени 700 (задав бесконечное повторение). Следующий код определит используемые исходный, целевой меши и значение скаляра:

```
// pAnimCollection = заранее загруженный объект
// cMorphAnimationCollection
// Time = значение DWORD, содержащее используемое время
// анимации, в данном случае 700

// Указатели на исходный и целевой меши, используемые для
// визуализации
D3DXMESHCONTAINER_EX *pSource, *pTarget;

// Значения скаляра, используемое для морфирования
float Scalar;

// Вызвать Update, указав морфируемый меш и данные скаляра
MorphAnim.Update("MyAnimation2", Time, TRUE, \
          &pSource, &pTarget, &Scalar);
```

Теперь указатели исходного и целевого меша содержат меши, используемые при визуализации морфированного меша, и Scalar, который содержит значение скаляра. Все начинает проясняться; все, что остается сделать, - это нарисовать морфируемый меш, используя технологии, рассмотренные в главе 8. Если хотите,

вы можете посмотреть демонстрационную программу для этой главы, в которой показана морфируемая анимация. Смелее, вы знаете чего хотите. Исходный код просто ждет, чтобы вы использовали его в своих проектах!

Получение данных морфируемого меша из альтернативных источников

Хорошо, создание собственных классов и шаблонов морфирующей анимации потребовало немалых усилий, но что в них полезного, если вы не можете получать данные морфируемой анимации из любых источников, таких как популярный пакеты трехмерного моделирования? Не расстраивайтесь, существует множество способов получения данных набора морфируемых анимаций, которые вы хотели бы видеть в ваших проектах.

Компакт-диск содержит программу MeshConv, которую вы можете использовать для преобразования файлов .MD2 в .X. Что такое файлы .MD2, спросите вы? Разработанные id Software, файлы .MD2 (используемые в таких играх как Quake фирмы id) содержат данные морфируемого меша и анимации. Вы обнаружите сотни файлов .MD2 в Интернете (например на http://www.planetquake.com/polycount), которые вы можете использовать в ваших программах. Используя программу MeshConv, вы можете преобразовывать эти файлы в .X для использования в собственных проектах.

Вы, наверное, помните эту удобную программу из главы 5. В случае, если вы еще не читали ту главу, позвольте мне кратко рассказать о ней. После запуска программы на экране появится диалоговое окно MeshConv, показанное на рис. 9.2.

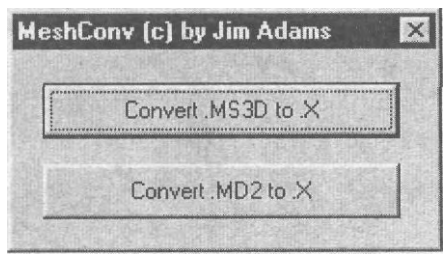


Рис. 9.2. Диалоговое окно MeshConv содержит две кнопки, нажав на которые вы можете преобразовывать файлы .MS3D и .MD2 в .X

Для преобразования файла .MD2 в .X щелкните кнопку "Convert .MD2 to .X" ("Преобразовать .MD2 в .X") диалогового окна MeshConv. Появится диалоговое окно "Open File". Это диалоговое окно позволяет вам указать расположение конвертируемого файла. Выберите подходящий файл и щелкните "Open".

После этого появится диалоговое окно "Save .X File" ("Сохранить файл .X"). Вы можете использовать это диалоговое окно для выбора файла .X, в котором вы хотите сохранить данные меша и анимации. Введите имя файла и нажмите "Save". Через мгновение перед вами появится окно сообщения, содержащее подтверждение успешного преобразования.

Теперь вы готовы использовать файлы .X с одним из классов, разработанным ранее в этой главе. В исходном файле расположен набор шаблонов Mesh, содержащих все используемые целевые меши. Один шаблон MorphAnimationSet поможет загрузить вам данные анимации, используя классы и технологии, изученные ранее в этой главе.

В качестве примера работы с файлами .X, созданными программой MeshConv, посмотрите демонстрационную программу этой главы MorphAnim.

Посмотрите демонстрационные программы

Эта глава содержит два проекта - один, преобразующий файлы .MD2 в .X files, и второй, иллюстрирующий использование наборов морфируемых анимаций в ваших собственных проектах. Запустите демонстрационную программу MorphAnim (показанную на рис. 9.3), чтобы увидеть насколько эффективны наборы морфируемых анимаций.

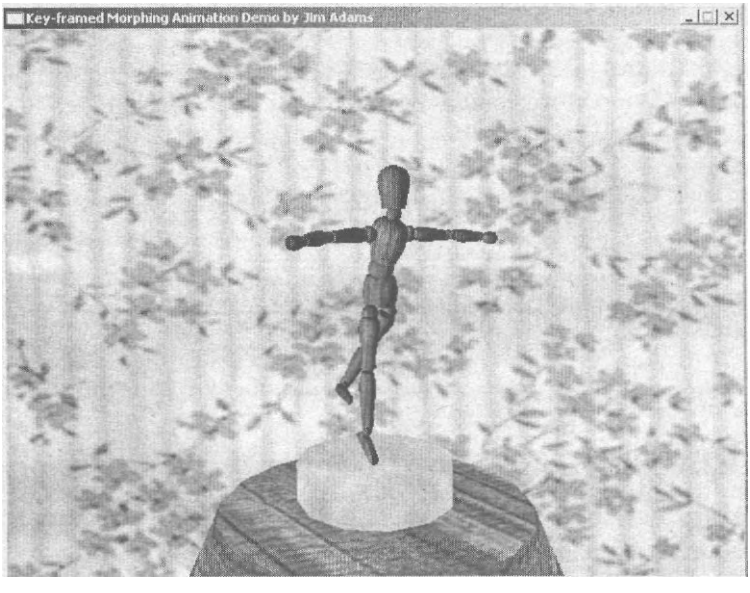


Рис. 9.3. Мечта аниматора становится реальностью, при помощи морфинга анимации балерина из музыкальной шкатулки анимирована

Программы на компакт-диске

В директории главы 9 компакт-диска книги вы обнаружите следующие проекты, которые вы можете использовать в ваших собственных игровых программах:

- MeshConv. Вы можете использовать эту полезную программу для преобразования файлов .MS3D и .MD2 в .X. Исходный код полностью откомментирован, и в нем приведено описание форматов обоих типов файлов. Он расположен в \BookCode\Chap09\MeshConv.
- **MorphAnim.** Эта демонстрационная программа иллюстрирует использование наборов морфируемой анимации. Она расположена в \BookCode\Chap09\MorphAnim.

Глава 10

Комбинирование морфированных анимаций

Удары, отклонения, толчки, ходьба и прыжки - вот множество анимаций, с которыми приходится работать. Вы можете себе представить, сколько вам потребуется времени для кропотливого объявления каждой из этих анимации, и все это, чтобы потом услышать, что вашему начальнику необходимо, чтобы персонаж прыгал и бил одновременно, или отклонялся и толкал, или выполнял любое другое количество комбинированных анимаций? Что вы будете делать?

Хорошо, если вы используете морфированные анимации, тогда все что вам остается сделать, это использовать свои программистские способности и разработать технологию комбинирования морфированных анимаций, в которой меши из ранее созданных анимаций могут быть соединены для образования новых уникальных анимаций в реальном времени. Цель данной главы - показать вам как это делать!

В этой главе вы научитесь:

- Изменять технологию морфирования для реализации комбинирования;
- Работать с базовым мешем в комбинированном морфировании;
- Управлять буферами вершин меша для комбинирования;
- Создавать вершинные шейдеры комбинированного морфирования.

Комбинирование морфированных анимаций

Давно, в главе 6, вы видели, как создавать новые динамические анимации, объединяя, или, скорее, комбинируя разнообразные движения меша, определенные множественными наборами анимаций. В то время как глава 6 была посвящена комбинированию наборов анимаций, основанных на скелетах, эта глава расскажет вам как достичь таких же эффектов комбинирования анимаций для морфируемых

мешей. Все правильно, в этой главе вы увидите, как объединять разнообразные движения ваших морфируемых мешей, основанных на пикселях, для создания новых динамических анимаций в реальном времени!

Комбинирование анимаций морфируемых мешей немного сложнее в сравнении с комбинированием анимаций мешей, основанных на скелетах. Но не волнуйтесь. Оно не на много сложнее, а просто требует другого подхода. При комбинировании морфированной анимации необходимо обеспечить возможность объединения не5скольких морфируемых мешей в один.

Например, предположим, что имеем меш, представляющий собой лицо человека, и две морфированные анимации: одна, открывающая и закрывающая рот меша, и вторая, моргающая глазами меша. На рис. 10.1 показаны циклы каждой анимации.

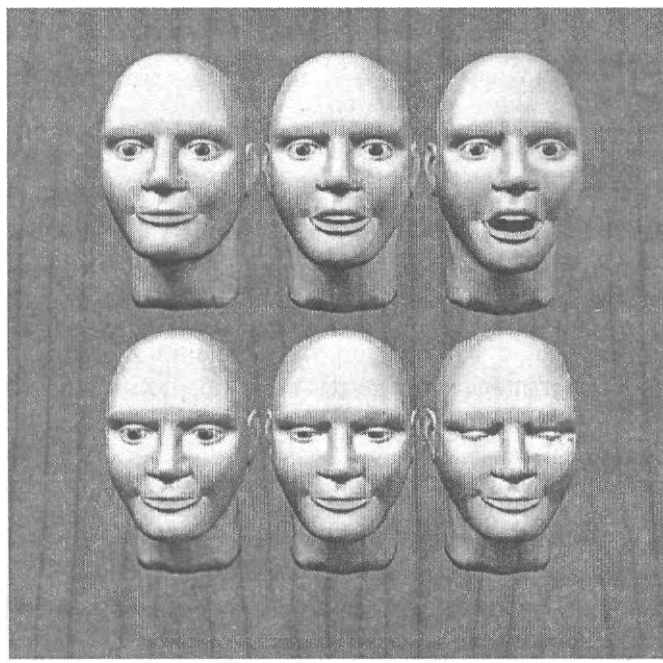


Рис. 10.1. Используя один и тот же меш, вы двигаете (морфируете) разнообразные вершины для создания двух уникальных последовательностей анимаций

Теперь предположим, что вы хотите все перемешать, скомбинировав эти две анимации (как показано на рис. 10.2) так, чтобы меш мог одновременно открывать рот и моргать глазами, с разньми скоростями для каждой анимации. Звучит устрашающе, не правда ли? Ну, на самом деле это не так сложно, когда вы знаете секрет.

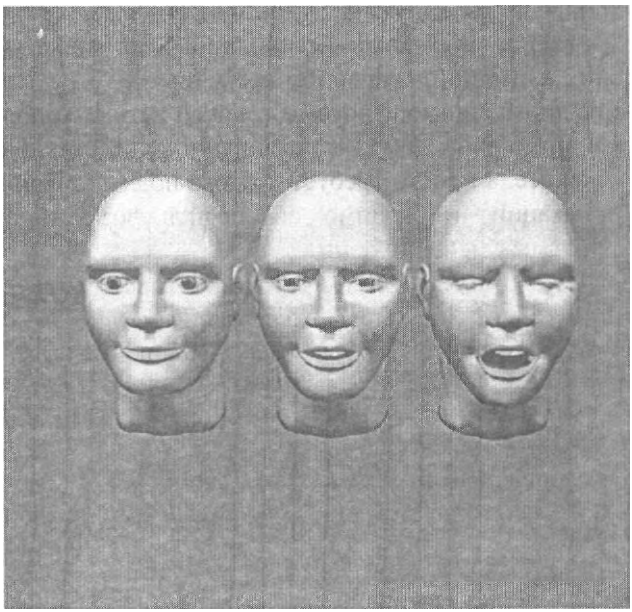


Рис. 10.2. Вы можете объединить две анимации для создания одного комбинированного меша

Весь фокус комбинирования морфированной анимации заключается в изолировании тех вершин, которые фактически движутся в каждой последовательности анимации, и объединении движения каждой вершины в один меш, который и необходимо визуализировать. Определение того, какие вершины движутся при последовательности анимации, кажется сложной задачей, но с помощью дополнительного ссылочного меша (называемого базовым мешем) вы можете решить эту проблему со связанными за спиной руками!

Использование базового меша в комбинированных морфированных анимациях

Настоящим секретом использования комбинированных преобразований является объявление ссылочного меша в уравнении. Базовый меш определяет начальные координаты каждой вершины в меше до применения морфинга. Зачастую базовым мешем является исходный меш операции морфирования. В примере лица человека базовым мешем является лицо человека с закрытым ртом и открытыми глазами.

Когда вы комбинируете два морфированных меша (или любое количество мешей), вы определяете разность координат вершин каждого морфированного меша и базового меша. Вы объединяете эти разности координат в результирующем комбинированном морфируемом меше. Читайте далее, чтобы узнать как вычислять эти разности.

Вычисление разностей

Вычисление значений разностей является простой задачей. Каждая комбинируемая анимация содержит исходный меш (базовый) и целевой меш. Для первой анимации целевым мешем будет тот, у которого рот человека полностью открыт. Целевой меш второй анимации содержит лицо человека с закрытыми глазами. И, конечно же, помните, что базовым мешем является лицо человека с закрытым ртом и открытыми глазами.

Для данного примера этими мешами будут меши, изображенные на рис. 10.2. Я думаю у вас не возникнет проблем при загрузке каждого меша в следующие три объекта:

```
ID3DXMesh *pBaseMesh; //Содержит базовый меш
ID3DXMesh *pTargetMesh1; //Содержит целевой меш первого морфирования
ID3DXMesh *pTargetMesh2; //Содержит целевой меш второго морфирования
```

Наряду с базовым и двумя целевыми мешами вам потребуется еще один для хранения результирующего комбинированного меша. Этот комбинированный меш имеет столько же вершин и граней, что и остальные меши, так что вы можете просто скопировать его из базового меша.

Для копирования базового меша (после его загрузки) в комбинированный меш (объект ID3DXMesh, названный pBlendedMesh) вы можете использовать следующий код:

```
ID3DXMesh *pBlendedMesh;

// Получить FVF базового меша и 3D устройство
DWORD FVF =pBaseMesh->GetFVF();
IDirect3DDevice9 *pMeshDevice;
pBaseMesh->GetDevice(&pMeshDevice);

// Скопировать меш
pBaseMesh->CloneMesnFVF(0,FVF,pMeshDevice,&pBlendedMesh);
```

Итак, вы будете работать с четырьмя объектами меша - базовым мешем, двумя целевыми и комбинированным. Для упрощения, я полагаю, что все меши используют одинаковую структуру вершин и FVF, использующий только положение,

нормаль и пару текстурных координат, как я определил тут. Если необходимо, просто скопируйте все меши, используя одинаковый FVF, как я делал раньше.

```
typedef struct {
  D3DXVECTOR3 vecPos; //Координаты вершины
  D3DXVECTIR3 vecNormal; //Нормаль вершины
  float u,v; //Текстурные координаты
} sVertex;
#define VERTEXFVF (D3DFVF_XYZ |D3DFVF_NORMAL |D3DFVF_TEX1)
```

Когда все меши готовы, вы можете заблокировать их вершинные буферы для получения доступа к данным вершин.

```
// Заблокировать все меши и получить указатели
sVertex *pBaseVertices,*pBlendedVertices;
sVertex *pTarget1Vertices,*pTarget2Vertices;

pBaseMesh->LockVertexBuffer(D3DLOCK_READONLY, \
                            (void**)&pBaseVertices);
pTargetMesh1->LockVertexBuffer(D3DLOCK_READONLY, \
                            (void**)&pTarget1Vertices);
pTargetMesh2->LockVertexBuffer(D3DLOCK_READONLY, \
                            (void**)&pTarget2Vertices);
pBlendedMesh->LockVertexBuffer(0,(void**)&pBlendedVertices);
```

После того как вы получили указатели на буферы вершин всех мешей, вы можете начать перебирать вершины каждого целевого меша, вычитая координаты вершины (и нормали) из соответствующих вершин базового меша (при этом каждая разность храниться во временных регистрах, о которых вы узнаете позднее).

```
// Перебрать все вершины
for(DWORD i=0;i<pBaseMesh->GetNumVertices();i++){

  // Получить разность координат вершин
  D3DXVECTOR3 vecPosDiff1 =pTarget1Vertices->vecPos - \
                           pBaseVertices->vecPos;
  D3DXVECTOR3 vecPosDiff1 =pTarget2Vertices->vecPos - \
                           pBaseVertices->vecPos;

  // Получить разность нормалей
  D3DXVECTOR3 vecNormalDiff1 =pTarget1Vertices->vecNormal - \
                            pBaseVertices->vecNormal;
  D3DXVECTOR3 vecNormalDiff2 =pTarget2Vertices->vecNormal - \
                            pBaseVertices->vecNormal;
```

Вы уже на полпути, но здесь необходимо задержаться! Мы вычислили разности, так что теперь необходимо скомбинировать их.

Комбинирование разностей

На данный момент мы посчитали разности значений координат вершин и нормалей. Следующим шагом является масштабирование каждой из этих разностей на величину комбинирования, используемую для каждого меша. Например, если вы хотите, чтобы первая анимация (открытия рта) использовала только 50 процентов разности (означая, что рот откроется только на половину), вы умножаете значения на 0.5.

Совет. *В дополнение к определению процента разностей для комбинирования вы можете использовать значения комбинирования в качестве коэффициентов при анимировании комбинированных мешей. Постепенно увеличивая значения комбинирования со временем, вы можете получить плавную анимацию.*

Для упрощения, задавайте величину комбинирования в виде двух переменных - одну для величины смешивания первого меша, а вторую для второго меша.

```
float Blend1 =1.0f;//Использовать 100% разностей
float Blend2 =1.of;
```

Для комбинирования разностей вам просто необходимо умножить их на только что определенные значения комбинирования.

```
// Применить значения комбинирования
vecPosDiff1 *=Blend1;vecNormalDiff1 *=Blend1;
vecPosDiff2 *=Blend2;vecNormalDiff2 *=Blend2;
```

После того как вы получили разности, масштабированные на коэффициенты комбинирования, вы можете сложить их, тем самым получив комбинированные значения разностей.

```
// Получить соответствующие значения комбинированных разностей
D3DXVECTOR3 vecBlendedPos =vecPosDiff1 +vecPosDiff2;
D3DXVECTOR3 vecBlendedNormal=vecNormalDiff1+vecNormalDiff2;
```

Последним шагом является прибавление комбинированных разностей к координатам вершин и нормалей базового меша и сохранение результата в буфере вершин комбинированного меша.

```
//Прибавить разности к значениям базового меша и сохранить результат
pBlendedVertices->vecPos =vecBlendedPos + \
                          pBaseVertices->vecPos;

// Нормализовать нормали перед сохранением!
D3DXVECTOR3 vecNormals =BlendedNormal+pBaseVertices->vecNormal;
D3DXVec3Normalize(&pBlendedVertices->vecNormal, &vecNormals);
```

Все, что остается сделать, - это увеличить указатели буферов вершин для обработки следующей вершины в каждом меше, закрыть кодовый блок for...next для завершения обработки вершин и разблокировать буферы вершин.

```
    // Перейти к следующим вершинам
    pBaseVertices++;pBlendedVertices++;
    pTarget1Vertices++;pTarget2Vertices++;
  }   //Следующая итерация цикла

  //Разблокировать буферы вершин
  pBlendedMesh->UnlockVertexBuffer();
  pTarget2Mesh->UnlockVertexBuffer();
  pTarget1Mesh->UnlockVertexBuffer();
  pBaseMesh->UnlockVertexBuffer();
```

Вот и все! Один полостью комбинированный меш готов к визуализации! Вы должны заметить, что комбинирование двух мешей эффектно и просто, на самом деле. А что вы скажете о комбинироЁании четырех или более мешей? Вы подумаете, что я сумасшедший, но если вы посмотрите исходный код демонстрационной программы комбинирования морфируемых мешей, содержащийся на компакт диске, вы увидите, что это выполнимо! Все правильно, полный код комбинирования четырех морфируемых анимаций находится на диске и ожидает, когда вы используете его в своем проекте.

Что это, скажите вы? Вы бы выбрали более быстрый способ комбинирования мешей? Хорошо, мой друг, боги программирования услышали ваши молитвы, и скоро вы увидите как можно использовать вершинные шейдеры для выполнения всей грязной работы комбинирования за вас!

Создание вершинных шейдеров комбинированного морфирования

Вершинные шейдеры являются вашими спасителями при работе с комбинированием морфированных анимаций. Ранее в этой главе, вы видели, что необходимо делать при комбинировании мешей - блокировать каждый буфер вершин, вычислять разности и наконец комбинировать значения, выполняя кадр за кадром этот медленный процесс.

Говоря об основных причинах медленной обработки, особенно если буферы вершин находятся в видео памяти. Использование вершинных шейдеров гарантирует увеличение скорости обработки комбинированных морфированных анимаций, потому что вы можете хранить все данные меша в быстрой видео памяти и позволить вершинному шейдеру обрабатывать все.

Комбинированный вершинный шейдер будет работать аналогично тому, как вы непосредственно комбинировали координаты вершин ранее в этой главе. Для каждого комбинируемого меша вам необходимо получить доступ к его буферу вершин. Используя вершинные шейдеры, необходимо будет ассоциировать каждый буфер вершин с потоком вершин. Вместо того чтобы для получения данных вершин блокировать буфер вершин, вы используете данные потока вершин (как ассоциировано в объявлении вершин) для получения координат и нормалей.

Единственно проблемой является то, что вершинные шейдеры не могут использовать так много потоков. В дополнение, каждый вершинный шейдер имеет доступ к ограниченному количеству вершинных регистров (таких как координаты, нормали и текстурные координаты). DirectX 8 и 9 ограничивает число вершинных регистров 16, так что приходится считать каждый кусочек информации.

Если структура вершин должна содержать по крайне мере координаты вершины, нормаль и текстуру (всего три регистра на один поток), то вы можете использовать только пять мешей, один из которых базовый. В вершинном шейдере вы можете установить до четырех комбинированных мешей. Первым шагом к созданию вершинного шейдера комбинированного морфирования является определение структур и объявлений вершин. Помните, что необходимо сохранять минимальное количество данных (максимум три используемых регистра), так что структура вершины может выглядеть так:

```
typedef struct {
  D3DXVECTOR3 vecPos;  //Координаты вершины
  D3DXVECTOR3 vecNormal; //Нормаль вершины
  float u,v; //Текстурные координаты
} sBlendVertex;
```

Комментарии внутри структуры sBlendVertex говорят сами за себя, так что я пропущу объяснения и перейду к объявлению вершин.

```
// Объявления и интерфейсы вершинного шейдера
D3DVERTEXELEMENT9 g_MorphBlendMeshDecl [] ==
{
  // Первый поток используется для базового меша
  // задать положение, нормаль и текстурные координаты
  {0,0,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT, \
       D3DDECLUSAGE_POSITION,0 },
  {0,12,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_NORMAL,0 },
  {0,24,D3DDECLTYPE_FLOAT2,D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_TEXCOORD,0 },

  // Второй поток используется для первого меша
  {1,0,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
       D3DDECLUSAGE_POSITION,1  },
```

```
    {1,12,D3DDECLTYPE_FLOAT3,D3DDECLMETH0D_DEFAULT,
          D3DDECLUSAGE_NORMAL,1 },
    {1,24,D3DDECLTYPE_FLOAT2,D3DDECLMETH0D_DEFAULT,
          D3DDECLUSAGE_TEXCOORD,1 },

    // Третий поток используется для второго меша
    {2,0, D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
          D3DDECLUSAGE_POSITION,2 },
    {2,12,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
          D3DDECLUSAGE_NORMAL,2 },
    {2,24,D3DDECLTYPE_FLOAT2,D3DDECLMETHOD_DEFAULT,
          D3DDECLUSAGE_TEXCOORD,2 },

    // Четвертый поток используется для третьего меша  '
    {3,0,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
         D3DDECLUSAGE_POSITION,3 },
    {3,12,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
          D3DDECLUSAGE_NORMAL,3 },
    {3,24,D3DDECLTYPE_FLOAT2,D3DDECLMETHOD_DEFAULT,
          D3DDECLUSAGE_TEXCOORD,3 },

    // Пятый поток используется для четвертого меша
    {4,0, D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
        D3DDECLUSAGE_POSITION,4 },
    {4,12,D3DDECLTYPE_FLOAT3,D3DDECLMETHOD_DEFAULT,
          D3DDECLUSAGE_NORMAL,4  },
    {4,24,D3DDECLTYPE_FLOAT2,D3DDECLMETHOD_DEFAULT,
          D3DDECLUSAGE_TEXCOORD,4  },

  D3DDECL_END()
};
```

В g_BlendMorphDecl вы можете видеть объявление пяти потоков, содержащих трехмерное положение, нормаль и двухмерные текстурные координаты. Первый поток имеет индекс использования 0, второй поток имеет индекс использования 1 и т. д. Это означает, что у вершинного шейдера будет доступ к пяти используемым типам для каждой компоненты вершины (положения, нормали и текстурных координат). В таблице 10.1 показаны типы данных, хранимые в каждом вершинном регистре.

После того как вы создали структуру вершин и их объявление, вы можете переходить к созданию самого вершинного шейдера. Шейдер будет дублировать то, что вы делали ранее в этой главе. Для каждой комбинируемой вершины вы вычисляете разность координат целевого морфируемого и базового меша.

Замечание. *Вы можете найти вершинный шейдер комбинированного морфирования (MorphBlend.vsh) на компакт диске. Посмотрите конец этой главы для получения дополнительной информации о демонстрационной программе комбинированного морфирования.*

Таблица 10.4. Назначение вершинных регистров комбинированной морфируемой анимации

Вершинный регистр	Назначение
v0	Трехмерные координаты первого (базового) меша
v1	Нормаль первого (базового) меша
v2	Текстурные координаты первого (базового) меша
v3	Трехмерные координаты второго меша
v4	Нормаль второго меша
v5	Текстурные координаты второго меша
v6	Трехмерные координаты третьего меша
v7	Нормаль третьего меша
v8	Текстурные координаты третьего меша
v9	Трехмерные координаты четвертого меша
v10	Нормаль четвертого меша
v11	Текстурные координаты четветого меша
v12	Трехмерные координаты пятого меша
v13	Нормаль пятого меша
v14	Текстурные координаты пятого меша

Эта разность масштабируется на заданное процентное соотношение (с помощью константы вершинного шейдера, находящейся в диапазоне от 0 до 1), и результат, полученный от каждого комбинируемого меша, добавляется к результирующему набору координат положения, нормали и текстурных координат выходной вершины. Просто, не правда ли?

Посмотрите на код вершин, чтобы увидеть что происходит.

```
;v0 =Положеуие (xyz) базового меша
;v1 =Нормаль (xyz) базового меша
;v2 =Текстурные координаты (xy) базового меша
;
;v3 комбинированное положение (xyz) первого меша
;v4 комбинированная нормаль (xyz) первого меша
;v5 =Комбинированные текстурные координаты (xy) первого меша

;v6 комбинированное положение (xyz) второго меша
;v7 комбинированная нормаль (xyz) второго меша
```

```
;v8 комбинированные текстурные координаты (xy) второго меша
;
;v9 =Комбинированное положение (xyz) третьего меша
;v10 =Комбинированная нормаль (xyz) третьего меша
;v11 =Комбинированные текстурные координаты (xy) третьего меша
;
;v12 =Комбинированное положение (xyz) четвертого меша
;v13 =Комбинированная нормаль (xyz) четвертого меша
;v14 =Комбинированные текстурные координаты (xy) четвертого меша
;
;c0-c3 =матрица мир+вид+проекция
;c4 комбинированные значения 0-1 (mesh1,mesh2,mesh3,mesh4)
;c5 =направление света
vs. 1.0
```

Сначала, вы увидите комментарии, помогающие определить назначение используемых регистров вершин и констант. Перед вызовом вершинного шейдера необходимо установить шесть констант вершин.

- От c0 до c3 должны содержать транспонированную матрицу преобразования мир*вид*проекция.
- c4 содержит значения комбинирования для каждого меша (c4.x для меша 1, c4.y для меша 2, c4.z для меша 3 и c4.w для меша 4).
- c5 содержит вектор направления света.

Я вернусь к этим константам немного позже; а пока я продолжу код вершинного шейдера. После комментариев я добавил необходимую версию вершинного шейдера. Как вы можете видеть, вершинный шейдер комбинированного морфирования требует вершинного шейдера версии 1.0, который должен поддерживаться большинством карт, к тому времени, как вы будете читать эту книгу.

После комментариев и необходимой версии идет объявление ассоциации элементов вершин. Эти объявления используются для привязывания компонент вершин, определенных в массиве D3DVERTEXELEMENT9, к регистрам вершин шейдера. Эти ассоциации идентичны показанным в таблице 10.1, так что это должно быть понятно.

```
;объявить привязывание
dcl_position v0 ;базовый меш
dcl_normal v1
dcl_texcoord v2

dcl_position1 v3 ;первый меш
dcl_normal1 v4
dcl_texcoord1 v5

dcl_position2 v6 ;второй меш
dcl_normal2 v7
dcl_texcoord2 v8
```

```
dcl_position3 v9 ;третий меш
dcl_normal3 v10
dcl_texcoord3 v11

dcl_position4 v12 ;четвертый меш
dcl_normal4 v13
dcl_texcoord4 v14
```

После ассоциирования регистров вершин вы можете получить доступ к данным вершин, используя регистры от v0 до v14. Регистр v0 содержит координаты вершины базового меша; регистр v6 содержит координаты вершины второго меша и т. д.

В начале фактического кода вершинного шейдера координаты вершины базового меша и его нормаль помещаются в два временных регистра (r0 и r1).

```
; Поместить базовые координаты и нормаль в регистры r0 и r1
mov r0,v0 ;координаты (r0)
mov r1,v1 ;нормаль (r1)
```

Шейдер использует эти два регистра только в качестве ссылок, так что они не будут переписаны при вызове последующих функий (по крайне мере до завершения работы шейдера, как вы скоро увидите). Следующий кусочек кода вычисляет разность координат базового меша и меша, заданного в качестве первого комбинируемого, который использует регистры вершин начиная с v3 до v5. Разность масштабируется (умножается) на значение константы c4.x (значение от 0 до 1) и прибавляется к начальным координатам из r0, хранимым в регистре r4. На протяжении оставшейся работы вершинного шейдера r4 будет содержать результирующие координаты вершины комбинированного меша. Точно такой же процесс повторяется для нормалей, а результат хранится в r5. Посмотрите:

```
;Получить разности первого комбинируемого меша и добавить их
;к результату
sub r2,v3,r0 ;Получить разность координат
mad r4,r2,c4.x,r0 ;Поместить результат в r4
sub r3,v4,r1 ;Получить разность нормалей
mad r5,r3,c4.x,r1 ; Поместить результат в r4
```

Тот же самый процесс повторяется еще три раза, по одному для каждого оставшегося комбинируемого меша. Однако начиная с этого момента, инструкции умножения и сложения находятся в соответствующих регистрах, так что разности координат и нормалей могут быть использованы позднее. Вот оставшийся код вычисления используемых значений разностей:

```
;Получить разности второго комбинированного меша и добавить их к
; результату
sub r2,v6,r0 ;Получить разность координат
```

```
mad r4,r2,c4.y,r4 ;Добавить полученные координаты к r4
sub r3,v7,r1 ;Получить разность нормалей

mad r5,r3,c4.y,r5 ;Добавить полученную нормаль к r5
;Получить разности третьего комбинированного меша и добавить их к
;результату
sub r2,v9,r0 ; Получить разность координат
mad r4,r2,c4.z,r4 ; Добавить полученные координаты к r4
sub r3,v10,r1 ; Получить разность нормалей
mad r5,r3,c4.z,r5 ; Добавить полученную нормаль к r5

;Получить разности четвертого комбинированного меша и добавить их к
;результату
sub r2,v12,r0 ; Получить разность координат
mad r4,r2,c4.w,r4 ; Добавить полученные координаты к r4
sub r3,v13,r1 ; Получить разность нормалей
mad r5,r3,c4.w,r5 ; Добавить полученную нормаль к r5
```

После того как вы получили результирующую комбинированную вершину (в r4), вершинный шейдер должен преобразовать ее положение на матрицу объединенного преобразования мира, вида и проекции (хранимой в константах от c0 до c3). Что же касается нормали (сохраненной в r5), вы можете векторно умножить ее на обратное направление света (хранимое в c5) для получения рассеянной компоненты цвета, используемого для затемнения многоугольников.

```
; Спроецировать положение, используя преобразование мир*вид*проекция
m4x4 oPos,r4,c0

;Векторно умножить нормаль на обратное направление света
dp3 oD0,r5,-c5
```

Наконец, текстурные координаты могут быть взяты из регистра вершин базового меша v2 и помещены в выходной регистр текстуры t0.

```
;Сохранить текстурные координаты
mov oT0.xy,v2
```

На этом заканчивается программирование вершинного шейдера. Все, что остается сделать, - это поместить шейдер в проект и выяснить, как заставить его работать.

Использование вершинного шейдера морфируемого комбинирования

После того как вы создали вершинный шейдер, вспомогательные структуры и объявления вершин, вы можете заставить это все работать! Предположим, что имеется базовый меш и четыре целевых меша, уже загруженных в объекты меша.

```
ID3DXMesh  *pBaseMesh;
ID3DXMesh  *pMesh1,*pMesh2,*pMesh3,*pMesh4;
```

Предположим то же самое и для вершинного шейдера и объявления вершин - что вы уже загрузили их и получили их правильные интерфейсы:

```
IDirect3DVertexShader9 *pShader;
IDirect3DVertexDeclaration9 *pDecl;
```

После загрузки вершинного шейдера вы можете установить его для визуализирования комбинированных мешей, с помощью следующих строк:

```
pDevice->SetFVF(NULL); //Очистить использование FVF
pDevice->SetVertexShader(pShader);  //Установить вершинный шейдер
pDevice->SetVertexDeclaration(pDecl);  //Установить объявления
```

Чтобы начать рисовать комбинированный меш, вы должны установить потоки вершин, указывающие на буферы вершин меша. Даже если вы не используете четыре комбинируемых меша, вы должны всегда устанавливать потоки вершин; просто используйте буфер вершин базового меша в качестве потока, если у вас менее четырех комбинируемых мешей.

```
// Получить размер вершины
DWORD VertexStride =D3DXGetFVFVertexSize(pBaseMesh->GetFVF());
//Получить указатели на буферы вершин
IDirect3DVertexBuffer9  *pBaseVB;
IDirect3DVertexBuffer9  *pMesh1VB,*pMesh2VB;
IDirect3DVertexBuffer9  *pMesh3VB,*pMesh4VB;

pBaseMesh->GetVertexBuffer(&pBaseVB);
pMesh1->GetVertexBuffer(&pMesh1VB);
pMesh2->GetVertexBuffer(&pMesh2VB);
pMesh3->GetVertexBuffer(&pMesh3VB);
pMesh4->GetVertexBuffer(&pMesh4VB);
//Установить потоки вершин
pDevice->SetStreamSource(0,pBaseVB,VertexStride);
pDevice->SetStreamSource(1,pMesh1VB,VertexStride);
pDevice->SetStreamSource(2,pMesh2VB,VertexStride);
pDevice->SetStreamSource(3,pMesh3VB,VertexStride);
pDevice->SetStreamSource(4,pMesh4VB,VertexStride);
```

Теперь необходимо получить текущие матрицы преобразования мира, вида и проекции вашего 3D устройства. Эти матрицы объединяются, транспонируются и сохраняются в регистры констант вершинного шейдера с c0 до c3. Следующий код замечательно с этим справляется.

```
//Получить матрицы мира,  вида и проекции
D3DXMATRIX  matWorld,matView,matProj;
pDevice->GetTransform(D3DTS_WORLD,&matWorld);
```

```
pDevice->GetTransform(D3DTS_VIEW,&matView);
pDevice->GetTransform(D3DTS_PR0JECTI0N,&matProj);

//Получить матрицу мир*вид*проекция и установить ее
D3DXMATRIX matWVP;
matWVP =matWorld *matView *matProj;
D3DXMatrixTranspose(&matWVP,&matWVP);
g_pD3DDevice->SetVertexShaderConstantF(0,(float*)&matWVP,4);
```

Для управления величиной комбинирования для каждого меша просто измените значения, хранимые в регистре c4. Регистр x константы c4 представляет собой величину комбинирования для первого меша и меняется в диапазоне от 0 до 1 (или больше, если вы хотите получить преувеличенные результаты).

Замечание. *На самом деле вы не должны вызывать GetTransform для получения различных матриц преобразования. Эти матрицы должны храниться в вашем приложении, по возможности (но не обязательно) на глобальном уровне.*

Аналогично и для c4.y, c4.z, c4.w - каждый меш имеет регистр, хранящий величину комбинирования (y для второго меша, z для третьего, w для четвертого). Пока что установим значения комбинирования в 100 процентов (сохраняя значение 1.0 для каждого значения комбинирования в объекте D3DXVECTOR4) и сохраним эти значения в константе вершинного шейдера c4, используя функцию SetVertexShaderConstantF.

```
//Установить величины комбинирования
D3DXVECTOR4 vecBlending =D3DXVECTOR4(1.0f,1.0f,1.0f,1.0f);
pDevice->SetVertexShaderConstantF(4,(float*)&vecBlending,1);
```

Последнее что нужно сделать, это установить направление света сцены (то же самое, что использовалось в структуре D3DLIGHT9) в константу вершинного шейдера c5. Например, если вы хотите, чтобы свет (расположенный в мировом пространстве) был направлен вниз, вы можете использовать вектор с координатами 0,1,0. (Заметьте, что используете объект D3DXVECTOR4 для сохранения направления вектора, в противоположность объекту D3DXVECTOR3 просто задайте 0 в качестве компоненты w.)

```
//Установить вектор света
D3DXVECTOR3 vecLight =D3DXVECTOR4(0.0f,1.0f,0.0f,0.0f);
pDevice->SetVertexShaderConstantF(5,(float*)&vecLight,1);
```

Т. к. вы получаете данные вершин из набора объектов ID3DXMesh, вам необходимо установить буфер индексов, потому что все объекты ID3DXMesh используют индексированные списки элементарных объектов. Необходимо установить буфер индексов только базового меша, потому что буферы индексов одинаковы у всех мешей. Следующий код устанавливает буфер индексов базового меша:

```
//Установить буфер индексов
IDirect3DIndexBuffer9 *pIndices;
pBaseMesh->GetIndexBuffer(&pIndices);
pDevice->SetIndices(pIndices,0);
```

Наконец, вы можете визуализировать комбинированный меш! Пропуская стандартный код установки текстур и материалов, вы можете визуализировать весь меш, используя следующий код:

```
// Визуализировать меш
pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, \
                              0,pBaseMesh->GetNumVertices(), \
                              0,pBaseMesh->GetNumFaces());
```

Если вы хотите увидеть полнофункциональный пример вершинного шейдера комбинированного морфирования, посмотрите на демонстрационную программу BlendMorphVS, расположенную на компакт-диске. В главе 11 "Морфируемая лицевая анимация" вы увидите как использовать вершинный шейдер для создания привлекательных лицевых анимаций!

Посмотрите демонстрационные программы

Демонстрационные программы главы 10 (BlendMorph и BlendMorphVS) иллюстрируют использование материала данной главы для комбинирования набора мешей в один анимированный меш. Обе программы, хотя и похожи внешне (рис. 10.3), используют различные технологии для визуализирования комбинированных морфируемых анимаций.

В BlendMorph анимации достигаются блокированием, обновлением и разблокированием буферов вершин различных мешей. В BlendMorphVS код блокировки, обновления и разблокирования был убран, зато был добавлен код, иллюстрирующий возможности использования мощи вершинных шейдеров при изменении буферов вершин! В любом случае, обе программы не разочаруют вас!

Программы на компакт-диске

В директории главы 10 компакт-диска книги вы обнаружите два проекта, которые иллюстрируют использование комбинированных морфируемых анимаций. Этими проектами являются:

- **BlendMorph.** В этом проекте вы увидите, как создавать анимацию, комбинируя морфируемые анимации, непосредственно изменяя вершины мешей. Этот проект расположен в \BookCode\Chap10\BlendMorph..

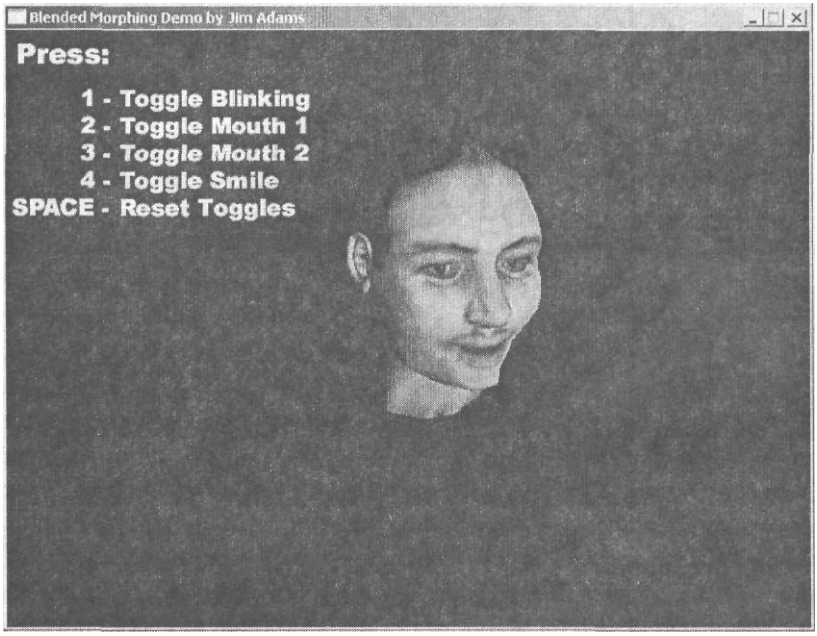


Рис. 10.3. Вводная часть лицевой анимации. Посмотрите, как комбинирование различных мешей может создавать простые анимации лица

- **BlendMorphVS.** Посмотрите, как осуществлять комбинированные морфируемые анимации при помощи вершинного шейдера, разработанного в этой главе. Этот проект расположен в \BookCode\Chap10\BlendMorphVS.

Глава 11

Морфируемая лицевая анимация

Чтобы сделать длинное объяснение коротким, анимация лица - это технология реалистичного анимирования меша, представляющего собой лицо персонажа, включающая движение рта, глаз и бровей. Использование лицевой анимации добавляет вашей игре некий шарм, заставляющий рты игроков Открываться. Ваш персонаж может ожить, если использовать хорошо синхронизированные со звуком движения рта и разнообразные выражения лица. Персонажи больше не деревянные безжизненные куклы, а живущие, дышащие, разговаривающие, кричащие и улыбающиеся жители.

Современные игры, использующие лицевую анимацию, такие как Medal of Honor: Frontline фирмы Electronic Art и Baldur's Gate: Dark Alliance фирмы Interplay повысили планку, и использование лицевой анимации быстро становится стандартом де-факто. Лицевая анимация в обеих этих играх просто потрясает. Добавленное ощущение реальности только увеличивает удовольствие игроков от каждой игры. Разве вы не хотите добавить реализма своему игровому проекту? Если да, тогда эта глава как раз то, что вам нужно.

В этой главе вы научитесь:

- Использовать технологии комбинированных морфируемых анимаций для анимирования лица;
- Создавать лицевые меши для использования в ваших проектах;
- Создавать фонемы и выражения лиц;
- Создавать последовательности анимации, использующие фонемы и выражения лиц;
- Проигрывать и синхронизировать лицевые анимации со звуковым потоком.

Основы лицевой анимации

Вы будете использовать четыре основные части лица: глаза, брови, рот и ориентацию головы. Самой главной из всех четырех частей является ориентация головы. Присмотревшись, вы можете заметить, что во время выполнения повседневных задач большинство людей постоянно двигают головой. Очень редко человек не перемещает голову. Наиболее часто движения головой происходят при разговоре; голова человека постоянно движется, когда он говорит. Ваш движок лицевого анимирования должен повторять те же самые движения.

Следующее, что вы можете заметить, это то, что глаза людей также постоянно двигаются. Вы хотите иметь эту особенность в своем анимационном пакете? Если да, вам придется задуматься куда глядят ваши персонажи. Зачастую у людей есть причина двигать глазами - они смотрят на окружающую обстановку и людей.

Для упрощения необходимо ограничить движения глаз. Если персонажу необходимо осмотреться, вы должны отделить глаза от лица (использовать их в качестве отдельных мешей). Намного проще вращать несколько глазных яблок, чем создавать набор морфируемых мешей, представляющих всевозможные ориентации глаз.

К глазам также относятся веки и ресницы. Как я и вы, для реалистичности ваш персонаж должен периодически моргать глазами. Используя комбинированные анимации, добавление возможности моргать глазами является простой задачей - все что необходимо, это морфировать целевой меш, представляющий базовый меш, в меш с закрытыми глазами. Изменяя величину комбинирования моргающего меша, со временем вы можете создать правдоподобную анимацию моргания.

Я упоминал использование комбинированной морфированной анимации? Вы правы, упоминал! С комбинированной морфированной анимацией очень легко работать, и ее использование замечательно подходит для лицевых анимаций. Давайте подробнее рассмотрим причину использования комбинированной морфированной анимации в вашем собственном движке лицевой анимации.

Использование комбинированного морфирования

Как я упоминал в главе 10, комбинированная анимация лица зависит от используемого базового меша. Базовый меш задает начальную ориентацию, которую используют остальные меши для определения деформируемых вершин при анимации. Только те вершины, положение которых в меше отличается от положения в базовом меше, обрабатываются и используются при визуализации результирующего комбинированного меша.

Базовый меш представляет собой лицевой меш, не содержащий никакого выражения лица. Для каждого выражения, принимаемого лицевым мешем, такого как моргание век, поднятие бровей, разнообразных форм рта, вы просто комбинируете целевой и базовый морфируемые меши.

Посмотрите на набор мешей, показанных на рис. 11.1, в качестве примера использования комбинированного морфирования в лицевых анимациях.

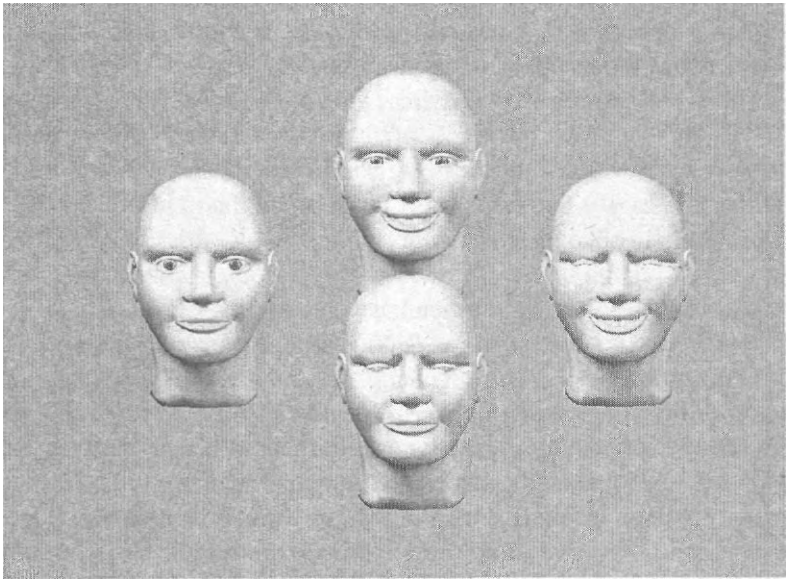


Рис. 11.1. Вы можете комбинировать базовый меш, изображенный слева, со множеством мешей, для создания уникальных анимаций. В данном случае улыбка комбинирована с морганием

Опять же, для анимации каждой детали лицевого меша используются простые цели морфирования. Моргание глаз может быть одной деталью, а ресницы могут быть другой. Вы можете поднять одну или обе брови. Каждая деталь соответствует эмоции персонажа, которую вы можете запрограммировать в своем анимационном пакете. Например, предположим, что имеется флаг, означающий что персонаж разозлен. При визуализации лицевой анимации ваш движок может скомбинировать меш, в котором обе брови опущены. Если персонаж задает вопрос, движок поднимет одну из бровей меша.

Говоря об эмоциях, вы можете использовать не только брови для передачи чувств персонажа. Рот также может изменять форму. Обычно углы рта персонажа повышаются или понижаются в соответствии с изменением эмоций. Расстроенные персо-

нажи опускают углы рта, хмурясь, в то время как счастливые персонажи растягивают рот в улыбке.

Таким образом, вы можете использовать целевые меши для отображения, эмоций меша. Вы можете опустить бровь и немного опустить губы меша для выражения правдоподобной ярости. Как показано на рис. 11.2, нет никакой нужды в применении двух раздельно скомбинированных морфируемых мешей, когда с тем же успехом можно использовать один!

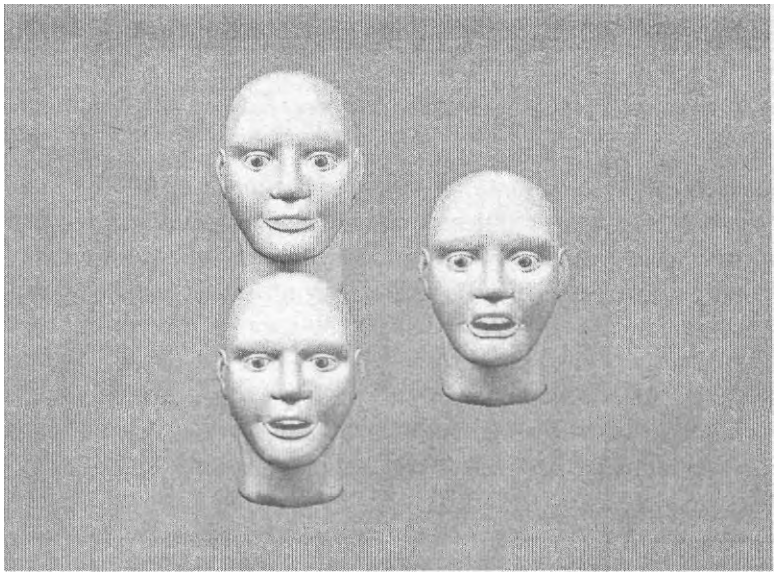


Рис. 11.2. Вместо использования двух целевых морфируемых мешей, вы можете скомбинировать их в один целевой меш, таким образом, сохраняя время и место

Давайте вернемся ко рту. Кроме того, что он помогает отображать эмоции, рот является замечательным средством общения. Губы являются мощной мускульной машиной, которая может принимать множество форм. Во время разговора рот меняет форму, помогая создавать звуки, образующие речь.

Разнообразные формы, которые принимает рот во время разговора, называются виземы (visemes), а произносимые звуки называются фонемы. В этой главе я смешиваю использование визем и фонем, потому что в отношении анимации они обозначают практически одно и то же[1]. Через минуту вы узнаете о фонемах и их использовании в синхронизированной анимации губ.

1. Зачастую даже вместо термина визема используют термин "визуальная фонема". - *Примеч. науч. ред.*

Использования фонем для речи

Как я только что заметил, фонемы являются звуками, произносимыми нами; каждое слово языка (любого языка) состоит из набора фонем. Например, слово "program" состоит из следующих семи фонем: p, r, ow, gh, r, ae, и m.

При произношении каждой из этих фонем рот принимает уникальную форму (визему). Поэтому теперь становится понятным, почему я перемешиваю эти термины - для каждой фонемы есть одна визема. При использовании лицевой анимации необходимо создать целевой морфируемый меш рта, который имеет точно такую же форму, как и ваш рот, при произнесении звуков.

Замечание. *Вместо того чтобы приводить значения уникода IPA в десятичной системе счисления, намного проще (и является стандартом) привести их в шестнадцатеричной системе счисления. Таким образом, в этой главе я буду использовать шестнадцатеричные значения IPA, находящиеся в диапазоне от 0x0000 до 0x0FFF.*

Я скоро вернусь к созданию лицевых мешей, произносящих звуки. А пока я хочу немного подробнее рассмотреть фонемы, чтобы увидеть, как вы можете использовать их в своих проектах. Фонемы создаются набором уникальных символов, при этом для усугубления уникальности каждому символу присваивается уникальное значение. Эти значения, известные как IPA (International Phonetic Alphabet (Международный фонетический алфавит)) значения уникода[2], изменяются в диапазоне от 0 до 1024 (для англоязычных пользователей), при этом каждая группа значений присваивается различным языкам и произношениям (как показано в таблице 11.1).

Таблица 11.5. Группы фонем Unicode IPA

Диапазон значений	Язык
0x0041 to 0x00FF	Стандартная латынь
0x0010 to 0x01F0	Европейская и расширенная латынь
0x0250 to 0x02AF	Стандартные фонемы
0x02B0 to 0x02FF	Символы изменения
0x0300 to 0x036F	Диакритические знаки

Англоговорящие люди используют значения, показанные в таблице 11.1, но большей частью используются значения (и фонемы) таблицы 11.2.

2. Unicode - уникод, 16-битный стандарт кодирования символов, позволяющий представлять алфавиты всех существующих в мире языков. - *Примеч. науч. ред.*

Таблица 11.6. Фонемы американского английского

Значение	Фонема	Пример
0x0069	iy	Feel
0x026A	h	Fill
0x00E6	ae	Carry
0x0251	aa	Father
0x028C	ah	Cut
0x0254	ao	Lawn
0x0259	ax	Ago
0x0065	ey	Ate
0x025B	eh	Ten
0x025A	er	Turn
0x006F	ow	Own
0x028A	uh	Pull
0x0075	uw	Crew
0x0062	b	Big
0x0070	p	Put
0x0064	d	Dug
0x0074	t	Talk
0x0067	g	Go
0x006B	k	Cut
0x0066	f	Forever
0x0076	v	Veil
0x0073	s	Sit
0x007A	z	Lazy
0x03B8	th	Think
0x00F0	dh	Then
0x0283	sh	She
0x0292	zh	Azure
0x006C	l	Length

Таблица 11.6. Фонемы американского английского

Значение	Фонема	Пример
0x0279	r	Rip
0x006A	y	Yacht
0x0077	w	Water
0x0068	hh	Help
0x006D	m	Marry
0x006E	n	Never
0x014B	nx	Sing
0x02A7	ch	Chin
0x02a4	jh	Joy

Значения IPA, показанные в таблице 11.2, являются индексами массива лицевых мешей фонем, используемого при визуализации. Для создания последовательности анимации, соедините значения IPA, образуя таким образом слова и предложения. Вы узнаете больше о создании последовательностей звуков в разделе "Создание последовательностей фонем", расположенном далее в этой главе.

Возвратимся к лицевым анимациям. Для создания законченной системы лицевой анимации необходимо анимировать (или скомбинировать) разнообразные меши, представляющие фонемы и выражения лица.

Создание лицевых мешей

Первым шагом при использовании лицевой анимации является создание набора лицевых мешей, которые бы представляли различные особенности лица игрового персонажа. Т. к. используются технологии комбинирования морфируемых анимаций, необходимо создать только один базовый меш и набор целевых мешей для каждой используемой особенности лица. Например, вам могут потребоваться только меши улыбки персонажа, моргания его глаз и движения рта в соответствии с произносимыми звуками.

Здесь и расположена самая сложная часть воплощения лицевой анимации - создание разнообразных лицевых мешей, используемых движком. Используя различные программы трехмерного моделирования, такие как trueSpace фирмы Caligari или Poser фирмы Curious Labs, вы можете создавать лицевые меши быстро и легко.

TrueSpace фирмы Caligari (версия 5.1 и новее) комплектуется плагином[3] Facial Animator, который очень помогает при создании, текстурировании и анимировании лицевых мешей. Я использовал плагин Facial Animator для создания демонстрационной программы этой главы.

Poser является пакетом, полностью ориентированным на создание персонажей, позволяющим моделировать всего человека. При использовании форм, текстур, одежд и лицевых особенностей Poser определенно является очень полезным трехмерным пакетом.

Независимо от используемого трехмерного пакета все сводится к одной вещи - созданию базового лицевого меша.

Создание базового меша

Оба пакета трехмерного моделирования, о которых я упоминал, поставляются с определенным набором лицевых мешей. Используя trueSpace, вы можете очень просто импортировать ваши собственные меши и подготавливать их для работы с плагином Facial Animator. При работе с Poser вы можете использовать лицевой генератор для получения практически неограниченного числа лиц.

Опять же, независимо от используемого пакета вам необходимо создать базовый меш. Помните, что этот базовый меш не должен содержать никаких выражений: рот должен быть закрыт, а глаза полностью открыты. Для упрощения, я собираюсь использовать один из лицевых, мешей, поставляемых в наборе с Facial Animator программы trueSpace. На рис. 11.3 показан базовый меш, который я буду использовать в этой главе.

После выбора меша, используемого в качестве базового, вам необходимо корректно текстурировать его. Замечательным свойством плагина Facial Animator программы trueSpace является то, что вы можете создавать карты текстур, используемые лицевым мешем, взяв картинки самогосебя, сделанные сбоку и спереди, и натянуть их на меш, используя инструмент Texturize Head. Для этой демонстрационной программы я использовал свое собственное лицо при текстурировании меша.

Замечание. *Лицевой меш Chris (содержащий малое количество граней), который я использую в качестве базового, имеет несколько недостатков, самым заметным их которых является отсутствие глаз. Я использовал инструмент AddFace программы trueSpace для добавления нескольких граней, представляющих собой глаза, что позволило мне наложить карту текстуры на них.*

3. Плагин (plug-in) - подключаемый программный модуль, который расширяет возможности родительской программы. - *Примеч. науч. ред.*

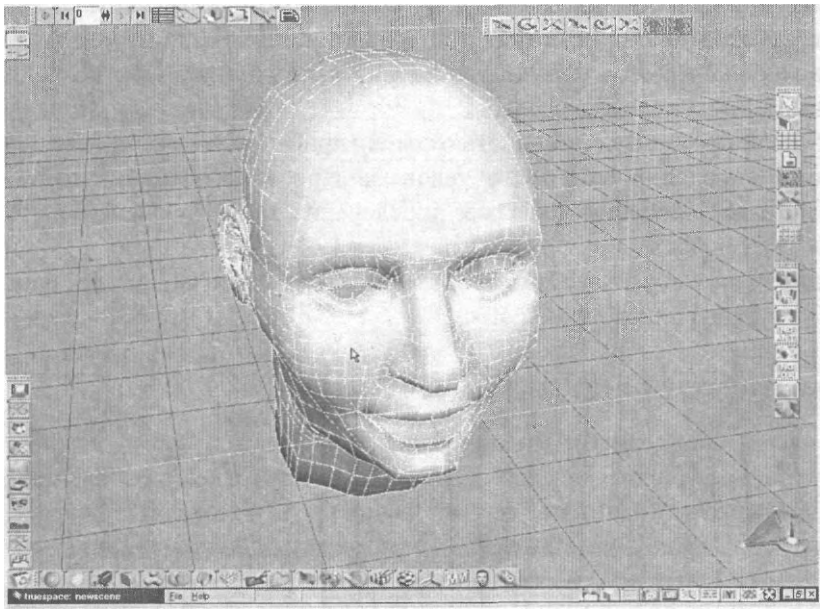


Рис. 11.3. Лицевой меш Chris (содержащий малое количество граней) из программы trueSpace замечательно послужит в качестве базового меша. Помня о лицензии фирмы Caligari, поставляемой для каждого пользователя, вы можете свободно изменять модель Chris в ваших приложениях

После двух небольших шагов базовый меш готов! Я знаю, что пропустил некоторые особенности, такие как моделирование головы, но эта книга не о моделировании, она об анимировании! По правде говоря, оба пакета трехмерного моделирования, о которых я упоминал, выполняют свою задачу и делают процесс моделирования лиц очень простым, так что о создании меша головы вы можете прочитать из документации этих пакетов.

А пока мы имеем готовый базовый меш, и можно начинать создавать выражения лиц, используемые в анимациях.

Создание выражений лица

Прежде чем продолжить, убедитесь, что вы сохранили базовый меш на диск, используя описательное имя, например, Base.x. Помните, вы используете формат .X, так что вы можете захотеть экспортировать меш в качестве .X файла. Если такой возможности нет, экспортируйте меш как файл .3DS. После того как вы выполнили это, вы можете использовать программу Conv3DS.exe, поставляемую с DirectX SDK, для

конвертирования его в .X файл. Обычно программа Conv3DS.exe расположена в директории \Bin\DXUtils DirectX SDK, или вы можете скачать последнюю ее версию с сайта http://www.microsoft.com/DirectX корпорации Microsoft.

После того как вы сохранили базовый меш на диск, вы можете начинать создавать различные виды мимики. Наиболее просто начинать с выражений лица, таких, как например, улыбка, моргание, нахмуривание. Опять же, я хочу сделать все максимально простым, поэтому будем использовать плагин Facial Animator программы trueSpace.

Так получается, что Facial Animator поставляется с набором предопределенных выражений лица, которые вы можете накладывать на лицевой меш простым щелчком мыши! Если задуматься, у Poser имеются точно такие же возможности, так что вы можете создавать выражения лица в любой из этих программ!

Для создания выражений лица меша щелкните закладку Expressions (Выражения) в диалоговом окне Facial Animator. Как вы можете видеть на рис. 11.4, появляется список выражений, которые можно применить к мешу.

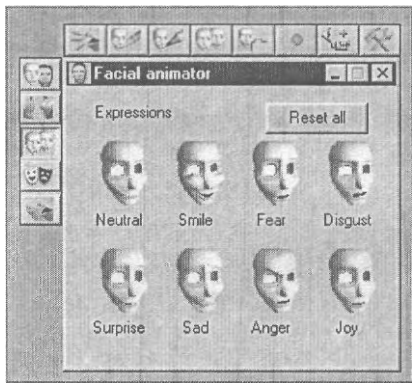


Рис. 11.4. Список Expressions плагина Facial Animator предоставляет вам восемь выражений, которые вы можете выбрать

Я хочу, чтобы демонстрационная программа была проста, и т. к. я достаточно счастливый парень, я хочу использовать выражение Smile (Улыбка). Щелкните на кнопку Smile, и вы увидите, как лицевой меш в трехмерном редакторе изменится в соответствии с выбранным выражением. Если вам интересно, щелкните на кнопках других выражений, чтобы увидеть как они влияют на меш. После того, как вы закончите, щелкните на кнопку Smile, чтобы вернуться к настройке улыбающегося меша.

После того как вы выбрали используемое выражение лица (в данном случае улыбку), экспортируйте меш. Для упрощения, назовите его Smile.x. Поместите файл Smile.x в ту же директорию, что и Base.x. Если вы хотите использовать большее количество выражений, щелкните на соответствующую кнопку выражения в диалогом окне Facial Animator, подождите изменения меша и экспортируйте его в .X файл.

Я не хочу вводить вас в заблуждение тем, что в Facial Animator только восемь выражений, поэтому щелкните на вкладку Gestures (Жесты). Вот! Должно появиться еще четырнадцать выражений (см. рис. 11.5).

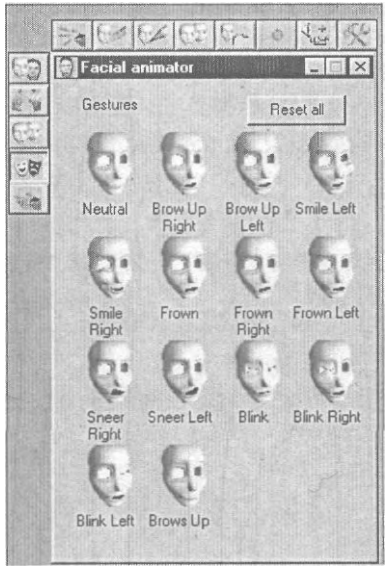


Рис. 11.5. Список Gestures плагина Facial Animator содержит еще 14 выражений, которые вы можете применять к мешу

Прежде чем использовать выражения вкладки Gestures, щелкните один раз кнопку Reset All (Сбросить все). Это вернет лицевой меш к начальной ориентации. Не стесняйтесь использовать набор жестов. Решите, какой жест вы хотите использовать, и экспортируйте меш, использующий его. Для своей демонстрационной программы я использую только жест Blink (Моргать).

После экспортирования всех выражений и жестов, перейдем к созданию разнообразных мешей для последовательностей фонем.

Предупреждение. *После того как вы экспортировали меши лицевой анимации и начали использовать их в своих проектах, убедитесь, что вы не меняете порядок вершин. Это является очень важным для правильного морфирования, как было рассмотрено в главе 8.*

Создание мешей визем

Основная часть создаваемых мешей будет является формами фонем, принимаемых мешем рта. Хотя вы и не используете полный набор мешей, соответствующих каждой мыслимой форме фонемы, полезно создать небольшой набор, представляющий собой основные формы, используемые последовательностями фонем.

Как я ранее замечал, уникальные звуки, используемые для создания слов, называются фонемы. Форма рта и расположение языка, при которых создается звук, называются визимы. Для создания синхронизированной анимации губ необходимо разработать набор viseme, используемых игрой для соответствующих фонем записанного голоса.

В зависимости от того, насколько реалистичной вы хотите сделать синхронизированную анимацию губ, вы можете использовать четыре формы визем для низкокачественных анимаций или вы можете использовать 30 или более форм визем для высококачественных анимаций.

В плагине Facial Animator программы trueSpace вы можете использовать набор из восьми визем, расположенных в разделе Viseme, который можно увидеть, щелкнув вкладку Viseme. Хотя их количество и ограничено, но для знакомства с лицевой анимацией восьми визем достаточно. Я порекомендую вам расширить этот список визем как можно быстрее, используя инструмент Head Geometry Setup (Настройка геометрии головы) плугина Facial Animator. Для того чтобы узнать как добавлять собственные выражения лица и визимы к списку используемых форм, проконсультируйтесь с документацией программы. Для примеров этой книги восемь визем достаточно.

Используя те же технологии, что и в предыдущем разделе, щелкните на каждую визему, используемую в анимации, и экспортируйте каждый меш. Для своей демонстрационной программы я называл каждый экспортируемый файл, основываясь на названии соответствующей визимы - ff.x, consonant.x, ii.x, aa.x, oo.x, uu.x и ee.x. Эти файлы формата .X располагаются в той же директории, что и файлы base.X и Smile.x.

Если вы собираетесь пойти другим путем и создать дополнительные меши визем, вам необходимо время для создания разнообразных форм меша рта. Как я упоминал, вы можете использовать более 30 визем в анимациях, так что вам придется попотеть.

Посмотрев на таблицу 11.2, вы можете увидеть основные фонемы английского языка. Именно для этих фонем вы хотите создать соответствующие лицевые меши. Самым простым способом создания мешей этих фонем является зеркало, в котором вы можете наблюдать за своим ртом во время произношения различных фонем. Смоделируйте точную форму вашего рта, сохраните результат и переходите к следующей фонеме.

Чтобы немного помочь вам, Michael B. Comet (создатель Comet Cartoons, расположенных на http://www.comet-cartoons.com) предоставил разнообразные фонемы, которые вы можете использовать (см. рис. 11.6)

Попытайтесь как можно точнее соблюдать форму при создании собственных мешей. Т. к. анимации проигрываются быстро, некоторые отклонения от формы реальной фонемы не должны быть проблемой. После создания всех используемых мешей фонем пришло время забавной части - соединения этих мешей в ваших собственных анимациях.

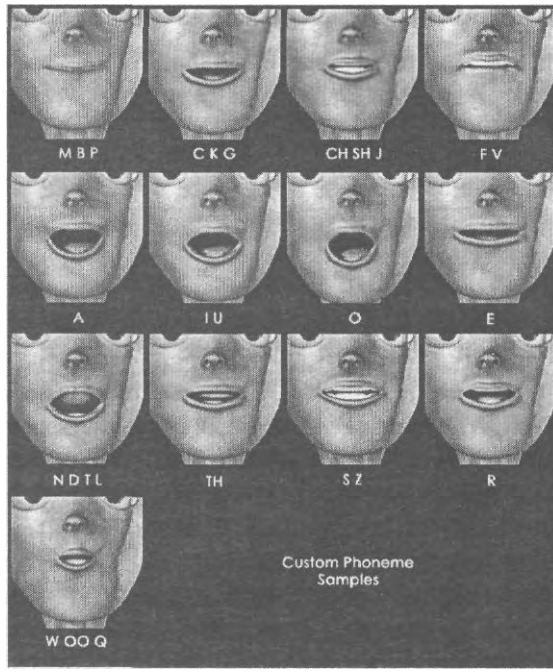


Рис. 11.6. Демонстрационный набор фонем, созданный Michael B. Comet, покажет вам, как необходимо создавать ваши собственные лицевые меши

Создание анимационных последовательностей

Теперь вы готовы перейти к анимациям! На данный момент вы должны иметь базовое понимание сущности фонем, а также должны были создать набор целевых мешей, используемых в комбинированных морфируемых анимациях. Эти меши должны содержать различные подробности, с которыми вы будете работать, такие как моргание глаз, поднятие бровей, изменение формы рта (виземы).

Прежде чем комбинировать анимации необходимо уточнить некоторые вещи. Я начну с самых простых особенностей лицевых анимаций, а потом перейду к более сложным! Что может быть простым при использовании лицевых анимаций, спросите вы? Конечно же автоматизированные функции!

Автоматизирование основных функций

Прежде чем переходить к самой сложной части лицевой анимации - последовательностям фонем - давайте рассмотрим простые вещи. Например, моргание глаз является абсолютно обычной вещью, и вы даже не заметите этого, если только человек, с которым вы разговариваете не перестанет моргать, что в данном случае является очевидным.

То же самое можно сказать и о выражениях лица человека. По мере того как вы разговариваете, ваше лицо меняет выражение в соответствии с тоном голоса. Веселые люди обычно улыбаются, когда разговаривают, а когда люди расстроены, они хмурятся. По мере того как во время разговора меняются ваши эмоции, выражение лица должно меняться соответственно. Малейшее движение губ, глаз, бровей может изменить эмоции, отображаемые на лице. Без этого проявления эмоций мы бы все выглядели как манекены!

Заметьте также, что голова людей всегда движется во время разговора. От небольших кивков и покачиваний до поворотов головы большинства людей постоянно движутся. Не беря в расчет эмоции, небольшие движения головы человека и моргание глаз полагается автоматическим. Другими словами, эти особенности должны быть анимированы без непосредственного их определения вами в последовательностях.

Позвольте мне вернуться к глазам и объяснить эту концепцию немного подробнее. Взрослый человек обычно моргает порядка от 10 до 20 раз в минуту. По мере концентрации на чем-либо, моргание становится менее частым, от 3 до восьми раз в минуту. Также, если вы смотрите вперед и назад, вам приходится моргать чаще.

Другие факторы, такие как эмоции, также изменяют частоту моргания глазами. Нервничающий или возбужденный человек моргает приблизительно в два раза чаще, чем обычно, а разозленные люди моргают реже. Золотое правило: если необходимо сосредоточиться, то частота моргания падает; если требуется много движений, частота моргания возрастает.

Для автоматизирования моргания глазами лицевого меша необходимо создать таймер, который бы комбинировал моргающий меш с требуемым интервалом времени. Если вы хотите поддерживать постоянную частоту моргания, вам необ-

ходимо заставить моргать глаза каждые 3000 миллисекунд (каждые три секунды). Если же вы хотите быть по-настоящему изобретательным, вы можете добавить возможность помечать разделы скрипта для изменения частоты моргания глаз.

Для этой главы я хочу поддерживать постоянную частоту моргания глаз. Т. о. моргающий целевой меш может быть скомбинирован каждые три секунды, причем продолжительность его анимации равна трети секунды. Это означает, что величина смешивания комбинированного морфируемого меша будет находится в пределах от 0 до 1 в течение первых 33 миллисекунд[4] анимации, после чего станет равным 0 на оставшееся время.

А теперь, как насчет движения головы, о котором мы говорили? Опять же, эту особенность легко реализовать, применяя небольшое случайное значение вращения к мешу при визуализации. Ничего сложного, просто убедитесь, что вы не слишком сильно вращаете меш, или все закончится каким-нибудь искаженным второсортным монстром из фильма!

Демонстрационная программа лицевой анимации этой главы иллюстрирует использование автоматического моргания и движения головы; вы можете использовать ее в качестве базы для создания собственных лицевых анимаций. (Для получения дополнительной информации о демонстрационной программе FacialAnim обратитесь к концу этой главы.)

После воплощения автоматических анимаций пришло время перейти к более сложной теме - созданию анимационных последовательностей фонем.

Создание последовательностей фонем

Наряду с используемыми элементарными анимациями, такими как моргание и подрагивание лица, наиболее важным аспектом лицевой анимации является синхронизация губ. Помните, что губы изменяют форму в соответствии с каждой произносимой фонемой, которые вы и хотите воспроизвести в анимации.

При воплощении синхронизированной анимации губ может быть использовано множество методов. На данный момент самым распространенным способом создания синхронизированной анимации губ является использование скриптов (произносимых и написанных) в паре со словарем фонем, используемым для разбиения всех произносимых (и написанных) слов на последовательности фонем.

Скрипт содержит точную произносимую и синхронизируемую с губами последовательность. Например, предположим, что вы хотите, чтобы игровой персонаж произнес "Hello and welcome" синхронно с движением губ. Используя текстовый

4. Видимо — опечатка. 1/3 секунды это 333 миллисекунды. — *Примеч. науч. ред.*

редактор, введите точную фразу и сохраните ее в текстовом файле. Далее, используя любую программу редактирования звука, произнесите и запишите эту фразу. Убедитесь, что вы сохранили ее в стандартном файле .WAV.

После того как вы выполнили это, вы можете разбить фразу на последовательность фонем. Это делается при помощи словаря фонем, который является файлом данных, содержащим последовательности фонем для каждого слова, хранимого в словаре. Соединяя последовательности для каждого слова, вы можете создать последовательности, соответствующие скриптам.

Например, в определении словаря фонем для слова "Hello" будут следующие фонемы: hh, ah, l и ow. Каждая из этих четырех фонем ассоциирована с соответствующим лицевым мешем. По мере обработки каждой фонемы во время лицевой анимации лицевой меш морфирует в соответствии со звуками. Для последовательности "Hello and welcome" будут использованы следующие фонемы hh, ah, l, ow, ae, n, d, w, eh, l, k, ah и m. Совместное использование скрипта и словаря творит чудеса.

Пока что эта хорошая идея, а как она работает? Давайте рассмотрим процесс написания, записи и обработки данных, который в результате станет синхронизированной анимацией губ. Начнем с написания скриптового файла (тестового файла). Этот скрипт должен содержать все слова произносимые вами или дублером.

Далее, используя формат записи с высоким качеством, запишите скрипт слово в слово. Говорите медленно и отчетливо, делая паузы между словами. После того как вы закончили со скриптом, вы можете очистить звуковой файл, сделав максимальную громкость и убрав статические помехи. Также полезным может быть обработка звука с помощью фильтра, чтобы преобразовать звуки между словами в паузы.

На рис. 11.7 изображена волновая форма нескольких слов, которые я записал для иллюстрации технологии синхронизации губ.

Вот здесь и начинается самая веселая часть! Используя словарь фонем, возьмите написанный скрипт и преобразуйте каждое слово в последовательность фонем. Используя звуковой файл, протестируйте каждое слово (по его звуковой волне) и определите их общую длительность. Длительность каждого слова определяется скоростью анимации для каждой последовательности фонем.

Как определить длительность каждого слова звукового файла? Или точнее, как узнать где находится каждое слово? Помните, имеется скрипт, содержащий точную последовательность произносимых слов. С его помощью вы определяете какое слово является каким, а как же определить положение и длину каждого слова?

Помните, как я сказал вам, чтобы вы говорили медленно, добавляя небольшие паузы между словами? Эти паузы являются как бы сигнальными указателями. Пауза означает конец одного слова и начало другого. Посмотрите еще раз на

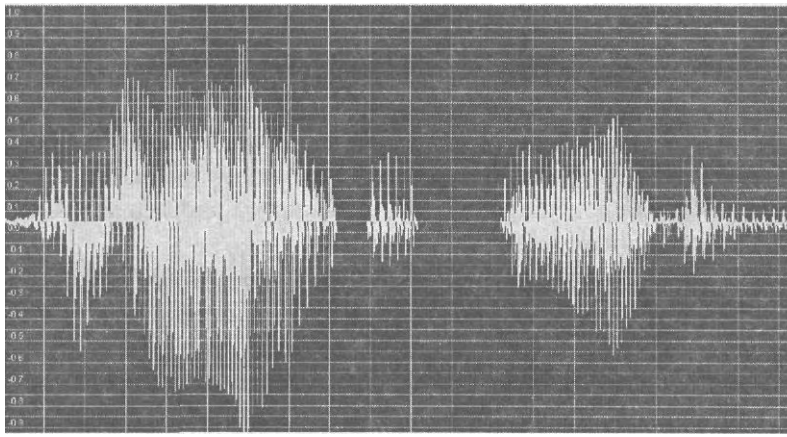


Рис. 11.7. Волновая форма слов "Hello and welcome!"

звуковую волну рис. 11.7. На этот раз я пометил паузы между словами и использовал их для изолирования слов. На рис. 11.8 показана новая звуковая волна, с выделенными словами и паузами.

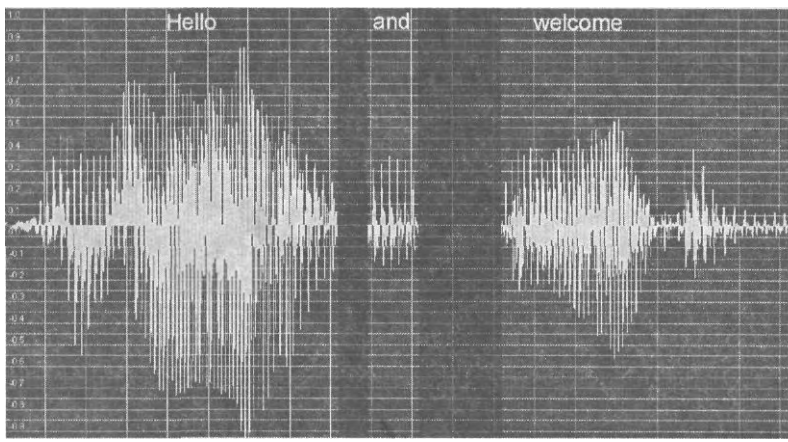


Рис. 11.8. Звуковая волна была разбита на секции, означающие слова и паузы

Теперь, я уверен, вы начинаете понимать значение использования скрипта в сочетании со словарем! Используя значение времени (основанное на частоте воспроизведения и положении звука), вы можете вернуться к последовательности фонем и использовать значения времени для ее анимации. Что может быть проще?

И сразу же в качестве примера, посмотрите на программу, которая работает аналогично только что изученному принципу - Linguistic Information Sound Editing Tool (Инструмент звукового редактирования лингвистической информации) корпорации Microsoft.

Использование лингвистического программного обеспечения корпорации Microsoft

Используя концепции применения скриптов, голосов и словарей фонем, пришло время перейти к реальной программе, которая составляет последовательности фонем. Эта программа, Linguistic Information Sound Editing Tool корпорации Microsoft, или для краткости LISET, анализирует файлы .WAV, с записанными в них голосами. В паре с напечатанным скриптом она создает последовательность фонем, которую вы можете использовать в своих играх.

Программа LISET является частью замечательного программного набора Agent корпорации Microsoft. Agent это полноценный пакет анимационных и Голосовых программ, позволяющий пользователям взаимодействовать с анимированными персонажами, встроенными в приложения и веб-странички. Представьте, вы можете иметь виртуального гида, который бы показывал посетителям ваш веб-сайт, и, более того, выглядел бы и говорил как вы!

Чтообы иметь такие возможности, как полная обработка речи, включая синтез речи из текста, обработка фонем и синхронизация губ, вы определенно должны посмотреть на пакет Agent. Для получения дополнительной информации о нем посмотрите сайт http://www.microsoft.com/msagent корпорации Microsoft.

Установите Agent в вашу систему, если вы еще не сделали этого. Убедитесь, что LISET также установлен, поскольку очень скоро он нам пригодится. После того как вы все проинсталлировали, запустите LISET. (Ищите его в меню "Programs" ("Программы").) На рис. 11.9 показано основное окно программы LISET.

После запуска LISET вы можете загрузить файл .WAV, который содержит произнесенный голосом скрипт. Щелкните пункт меню File и выберите Open. Появится диалоговое окно Open, позволяющее указать расположение файла .WAV. Найдите его, выберите и щелкните Open, чтобы продолжить.

После загрузки файла .WAV он сразу готов к обработке. Однако необходимо указать текстовый скрипт, чтобы LISET мог сопоставить его со звуковой волной. Видите окно редактирования Text Representation (Представление текста) наверху приложения LISET? Именно туда вставляется текстовый скрипт. Однако не спешите, потому что окно редактирования может содержать только ограниченное количество букв. Так что пытайтесь вставлять скрипты блоками по 65,000 слов или менее за раз.

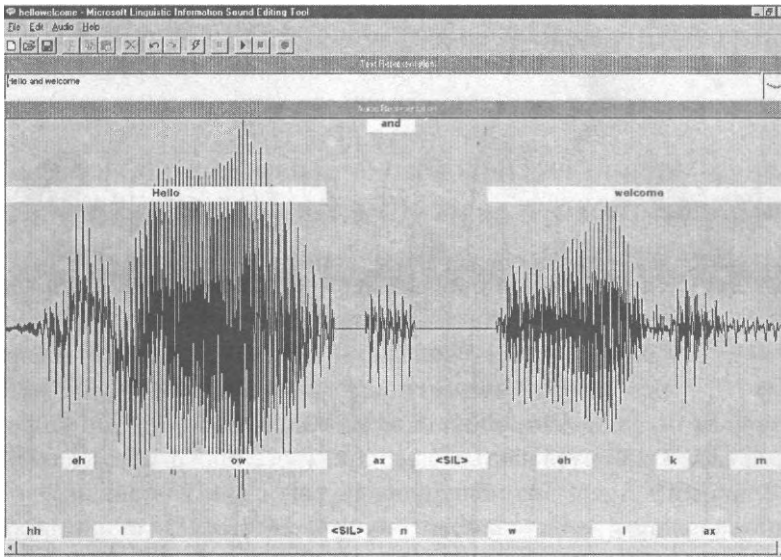


Рис. 11.9. Linguistic Information Sound Editing Tool корпорации Microsoft позволяет вам загружать файлы .WAV и отмечать кусочки звуковых волн маркерами фонем

Посмотрите еще раз на рис. 11.9, где вы можете видеть, что я загрузил файл .WAV и ввел соответствующий текст. Для проверки последовательности фонем скрипта щелкните Audio (Аудио) и выберите Play (Играть). Видите маленький рот в правом верхнем углу LISET? По мере проигрывания звука маленький рот меняет свою форму в соответствии с произносимыми фонемами. Замечательно, не правда ли?

Для получения настоящего удовольствия щелкните Edit и выберите Generate Linguistic Info (Сгенерировать лингвистическую информацию). Через несколько мгновений (и появления окна продвижения) последовательность фонем будет проанализирована, а информация о них будет наложена на звуковую волну. Посмотрите на рис. 11.10, чтобы увидеть, что я имею ввиду.

LISET выполняет приличный объем работы, помещая фонемы в правильные положения в звуковом файле, но если что-то не совсем правильно, вы можете вручную изменять положения и продолжительности каждой фонемы.

Во время перемещения мыши над каждой фонемой вы заметите, что указатель меняется. На правой и левой фанице фонемы курсор меняется на двойной квадрат со стрелочками, означая, что если вы щелкните и переместите мышь, размер фонемы изменится. Щелчок внутри самой фонемы (не на ее краях) выделит небольшую часть звуковой волны.

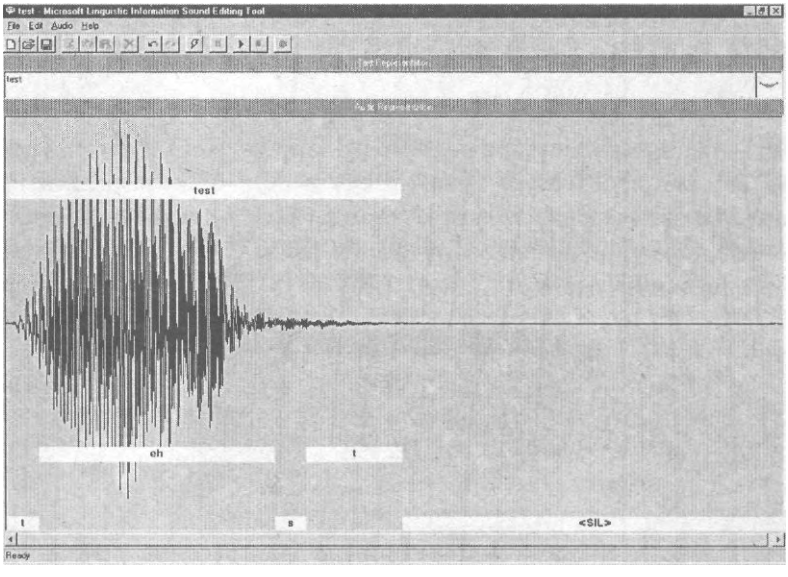


Рис. 11.10. Слово "test" состоит из четырех фонем (t, eh, s и t), которые находятся сверху звуковой волны. Заметьте, что пауза в конце слова также помечена

Вы можете выщелить кусочки звуковой волны, щелкнуть Edit и выбрать Replace Phoneme (Заменить фонемы), Delete Phoneme (Удалить фонему), Insert Phoneme (Вставить фонему) или Insert Word (Вставить слово). Replace Phoneme позволяет вам изменять фонемы на другие, Delete Phoneme удаляет фонему из последовательности. Используя две функции Insert, вы можете вручную помещать новые слова или фонемы в звуковую волну.

После того как вы обработали и разместили последовательность фонем, отображаемых LISET, вы можете сохранить информацию о них. Щелкните File и выберите Save As. Выберите путь и имя сохраняемого файла, и все готово!

В результате использования LISET последовательности фонем хранятся в специальных файлах, имеющих расширение .LWV. Единственной проблемой является то, как использовать эти файлы в собственных программах? Как насчет конвертирования их во что-нибудь более читаемое?

Конвертирование файлов LISET в .X

Формат файла .LWV, используемый приложением LISET, содержит всю информацию о фонемах, необходимую для создания синхронизированной лицевой анимации губ. Единственной проблемой является то, что вы не хотите иметь дело

с форматом файла .LWV в своих проектах. Необходим способ преобразования последовательности фонем из формата .LWV в какой-нибудь более читаемый формат, например .X.

Все правильно, .X опять приходит на помощь! Компакт-диск этой книги содержит вспомогательную программу, называемую ".LWV to .X Converter" (для краткости ConvLWV), которая преобразует файлы .LWV в .X. Посмотрите конец этой главы, чтобы узнать местоположение этой программы. Программа является простой и очень удобной в использовании. Для преобразования файла .LWV в .X все, что необходимо сделать, - это щелкнуть кнопку "Convert .LWV to .X", как показано на рис. 11.11.

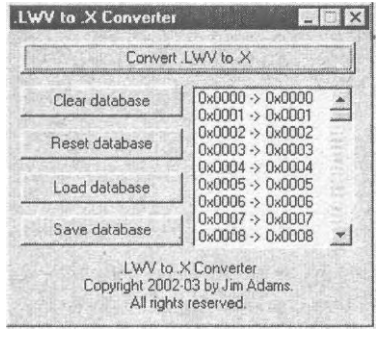


Рис. 11.11. Программа ConvLWV предоставляет шесть управляющих элементов, самым важным из которых является кнопка "Convert .LWV to .X"

После того как вы нажмете кнопку "Convert .LWV to .X", появится диалоговое окно "Select .LWV File for Import" ("Выберите импортируемый файл .LWV"). Используйте управляющие элементы этого окна для указания исходного конвертируемого файла .LWV. После выделения файла нажмите Open.

Далее вы увидите диалоговое окно Select .X File for Export (Выберите экспортируемый файл .X). Введите путь и имя файла, в который будет экспортирована последовательность фонем, и нажмите Save. Фу! Через несколько секунд в вашем распоряжении окажется .X файл, содержащий последовательность фонем!

Формат экспортируемого файла .X выглядит следующим образом:

```
xof 0303txt 0032

// Экспортировано с помощью программы .LWV to .X Converter v1.0 (c)
2002-
//03 by Jim Adams

Header
{
```

```
  1;
  0;
  1;
}
```

Как и обычный .X файл, экспортированная последовательность фонем начинается с определения формата заголовка. После него находится бесстыдная рекламная вставка и объект данных Header. Ничего, чего бы вы не видели раньше, не происходит. Однако далее следует то, на что необходимо обратить внимание.

```
// DEFINE GUID(Phoneme, 0x12b0fb22, 0x4f25, 0x4adb, 0xba, 0x0, 0xe5,
0xb9,
// 0xc1, 0x8a, 0x83, 0x9d)
template Phoneme
{
  <12B0FB22-4F25-4adb-BA00-E5B9C18A839D>
  DWORD PhonemeIndex;
  DWORD StartTime;
  DWORD EndTime;
}
// DEFINE_GUID(PhonemeSequence,
// 0x918dee50, 0x657c, 0x48b0,
// 0x94, 0xa5, 0x15, 0xec, 0x23, 0xe6, 0x3b, 0xc9);
template PhonemeSequence
{
  <918DEE50-657C-48b0-94A5-15EC23E63BC9>
  DWORD NumPhonemes;
  array Phoneme Phonemes[NumPhonemes];
}
```

Последовательность фонем, хранимая в файле .X, использует два специализированных шаблона. Первый шаблон, Phoneme, хранит информацию о каждой фонеме. В нем содержится идентификационное число фонемы (номер меша фонемы, о котором вы скоро прочитаете) и начальное и конечное время (в миллисекундах) анимации фонемы.

Так и есть, шаблон Phoneme фактически является ключевым кадром анимации! Вот здесь то и появляется шаблон PhonemeSequence. Он используется для хранения массива объектов Phoneme, определяющего полную последовательность анимации. Количество ключевых кадров анимации определяется первым значением шаблона PhonemeSequence, после которого следует массив ключевых кадров. Такая реализация структуры делает загрузку анимации фонем быстрой и безболезненной.

Давайте продолжим рассмотрение только что созданного файла .X (или даже того, который я создал для иллюстрирования работы конвертатора). Следующий объект PhonemeSequence, названный PSEQ, содержит небольшую анимацию, состоящую из пяти ключевых кадров.

```
PhonemeSequence PSEQ {
  5;
  116; 0; 30;, // 0x0074
  603; 30; 200;, // 0x025b
  115; 200; 230;, // 0x0073
  116; 230; 270;, // 0x0074
  95; 270; 468;; // 0x0'05f
}
```

Как вы можете видеть, каждая фонема представлена десятичным Unicode значением IPA (для каждого ключевого кадра в комментариях показано шестнадцатеричное значение IPA). Чтобы сделать систему лицевой анимации по настоящему универсальной, вы можете создать лицевой меш для каждой фонемы. Как вы можете догадаться, ни при каких обстоятельствах не используется более нескольких сотен различных мешей для лицевой анимации, так что необходимо ограничить количество Unicode значений IPA, с которыми вы работаете.

Вот здесь-то вспомогательная программа ConvLWV и проявляется во всей красе - переприсваивая известные вам Unicode значения IPA в меньшие, которыми легче управлять в приложениях. Посмотрев на рис. 11.11, вы заметите, что нижняя часть уточняет использование базы данных преобразования. Эта база данных используется для переприсваивания Unicode значений IPA в значения, заданные вами. Например, вместо использования 0x025b для представления фонемы "eh" (например, в словах "ten" и "berry") вы можете использовать значение 0x0001 (которое может также служить индексом меша фонемы в анимационном движке).

Потом эти значения используются в качестве индексов в массиве мешей, представляющих фонемы. Теперь, вместо использования сотен мешей, вы можете работать с маленьким набором мешей, использующихся для множества фонем. Каждый раз, когда вы переприсваиваете Unicode значения IPA и преобразуете файл .LWV в .X, эти Unicode значения IPA преобразуются в объекте PhonemeSequence. Например, после переприсвоения набора Unicode значений IPA предыдущий объект PSEQ может выглядеть так:

```
PhonemeSequence PSEQ (
  5;
  2; 0; 30;, // 0x0074
  3; 30; 200;, // 0x025b
  6; 200; 230;, // 0x0073
  2; 230; 270;, // 0x0074
  0; 270; 468;; // 0x005f
}
```

По умолчанию все Unicode значения IPA переприсваиваются таким же значениям: 0x025b всегда преобразуется в 0x025b, 0x0075 всегда 0x0075 и т. д. Для изменения переприсвоенного Unicode значения IPA найдите его в окне списка, в нижней правой части диалогового окна ConvLWV (как показано на рис. 11.12).

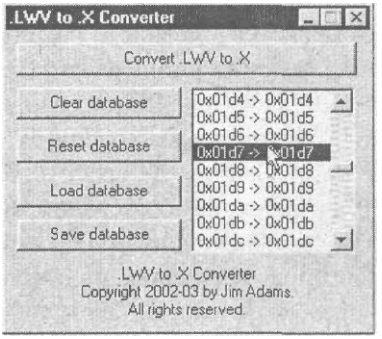


Рис. 11.12. В окне списка щелкните два раза на Unicode значения IPA, которое вы хотите изменить. Слева отображается Unicode значение, а справа расположено переприсваеваемое значение

В качестве примера, предположим, вы хотите переприсвоить фонему "eh" (Unicode значения IPA0x025b) значению 0x0001. В окне списка найдите 0x025b (слева) и щелкните два раза на нем. Появится диалоговое окно Modify Conversion Value (Измените преобразуемое значение), которое позволяет вам ввести новое значение (см. рис. 11.13). Введите 0x0001 и щелкните OK. Значения, содержащиеся в окне списка, обновятся, и вы можете продолжить переприсваивать их.

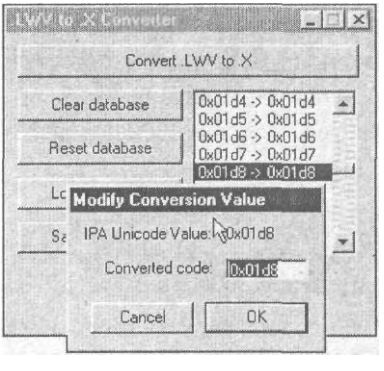


Рис. 11.13. Диалоговое окно Modify Conversion Value позволяет вам изменять Unicode значения IPA. Введите новое используемое значение и нажмите OK

Так что тяжелая работа была проделана не зря, вы имеете возможность сохранять, а потом перезагружать переприсвоенные значения. Это особенно удобно, если вы работаете со множеством различных последовательностей, используя один и те же значения преобразования. Для сохранения базы данных преобразования щелкните Save Database (Сохранить базу данных) и введите имя файла. Все базы данных преобразований сохраняются в файле с расширением .IPA. Для загрузки базы данных преобразования щелкните кнопку Load Database (Загрузить базу данных), укажите загружаемый .IPA файл и нажмите Load.

Последними двумя кнопками программы ConvLWV являются Clear Database (Очистить базу данных) и Reset Database (Сбросить базу данных). Кнопка Clear Database устанавливает все значения преобразования в 0x0000, означая, что всем Unicode значениям IPA переприсваиваются нули. Щелкнув на кнопку Reset Database вы устанавливаете все значения преобразования в соответствующие Unicode значения IPA (таким образом 0x025b преобразуется в 0x025b, 0x0075 в 0x0075 и т. д.)

Вот и все об использовании программы ConvLWV. Смелее, попробуйте все - записывать, обрабатывать, конвертировать файлы .LWV в .X. Когда вы будете готовы, можно переходить к загрузке последовательностей фонем в ваши программы при помощи специализированного анализатора файлов .X.

Использование анализатора файлов .X для последовательностей

После обработки файла .LWV программой ConvLWV в вашем распоряжении окажется файл .X, содержащий пару определений шаблонов и последовательность фонем, основанную на ключевых кадрах. Все, что вам нужно, - это написать анализатор .X, который бы загружал эту последовательность в массив структур, содержащих номера ссылок на меши фонем и начальное и конечное время каждого кадра. Одна структура предназначена для хранения ключевых кадров.

```
typedef struct {
  DWORD Code; // Номер меша фонемы
  DWORD StartTime; // Начальное время морфирования
  DWORD EndTime; // Конечное время морфирования
} sPhoneme;
```

При анализе файла .X, содержащего последовательность фонем, создайте массив структур sPhoneme в соответствии с количеством ключевых кадров, используемых в анимации. Предположим, имеется следующий объект PhonemeSequence (содержащий последовательность фонем):

```
PhonemeSequence PSEQ {
  5;
  2; 0; 30;,
  3; 30; 200;,
  6; 200; 230;,
  2; 230; 270;,
  0; 270; 468;;
}
```

Этот объект содержит пять ключевых кадров анимации, определяемых первым числом объекта. По мере анализа объекта PSEQ необходимо будет создать пять соответствующих структур sPhoneme. После того как вы создали эти структуры, вы можете просматривать список ключевых кадров.

Первый ключевой кадр использует меш фонем номер два и полностью комбинируется за 30 миллисекунд (начало в 0 миллисекундах, а конец в 30). Второй ключевой кадр использует меш номер 3 и полностью комбинируется за 170 миллисекунд (начало в 30 миллисекунд, а конец в 200). То же самое справедливо и для оставшихся ключевых кадров.

Одной из тем, обсуждения которой я избегал до теперешнего момента, является фактический принцип действия комбинирования в лицевой анимации. В главе 8 вы видели, как комбинировать несколько мешей. Для лицевой анимации моргания глаз и движения бровей комбинирование соответствующих мешей на разные доли является превосходным решением. Затруднения появляются при попытках морфировать один меш фонемы в другой. Морфирование базового меша в меш фонемы не вызывает никаких трудностей. Однако попытка морфирования одного меша фонемы в другой невозможна. Если базовый меш является исходным при операции морфирования, а меш фонемы является целевым, то как мы можем комбинировать меши фонем, не задав нового базового меша?

Конечно же, всегда можно задать новый базовый меш для комбинирования, однако проблема в том, что все анимации созданы в соответствии с начальным базовым мешем (в котором рот меша закрыт, а глаза открыты). Использование меша фонемы в качестве нового базового меша вызовет только новые трудности!

В чем же решение? На самом деле оно достаточно просто. Комбинируя базовый меш с двумя различными мешами фонем одновременно, можно сделать, чтобы меш морфировал бы из одного фонемного меша в другой. Все, что необходимо сделать, — это медленно уменьшать значения комбинирования исходного меша фонемы от 1 до 0 и медленно увеличивать значения комбинирования целевого меша фонем от 0 до 1.

Я вернусь к этому решению комбинирования немного позже. А пока, давайте перейдем к анализатору .X и посмотрим на объявление класса.

```
class cXPhonemeParser : public cXParser
{
  public:
   char *m_Name; // Имя последовательности
   DWORD m_NumPhonemes; // Количество фонем в последовательности
   sPhoneme *m_Phonemes; // Массив фонем
   DWORD m_Length; // Длина (в миллисекундах) последовательности

  protected:
   BOOL ParseObject(IDirectXFileData *pDataObj, \
              IDirectXFileData*pParentDataObj,\
              DWORD Depth, \
              void **Data, BOOL Reference);

  public:
   cXPhonemeParser();
   ~cXPhonemeParser();

   // Освободить загруженные ресурсы
   void Free();

   // Найти фонему по заданному времени
   DWORD FindPhoneme(DWORD Time);

   // Получить номер меша и значения масштабирования времени
   void GetAnimData(DWORD Time, \
               DWORD *Phoneme1, float *Phoneme1Time, \
               DWORD *Phoneme2, float *Phoneme2Time);
};
```

Я хочу рассмотреть класс cXPhonemeParser по частям, чтобы вы могли лучше понять, что в нем происходит. Первое, что вам необходимо заметить в классе, это объявление переменных. Каждая анимация последовательности фонем хранится в объекте PhonemeSequence. Помните, что каждому экземпляру объекта может быть присвоено имя. В этом и есть предназначение переменной m_Name - хранить имя экземпляра объекта.

Замечание. *В данном случае, класс cXPhonemeParser хранит данные только одного объекта PhonemeSequence, так что имеется только один буфер, содержащий имя и остальную информацию о фонемах. Если вас это беспокоит, я рекомендую расширить класс, чтобы он содержал множество последовательностей фонем.*

За m_Name следует количество ключевых кадров, содержащихся в последовательности (хранящихся в .X файле), и массив структур sPhoneme, содержащих информацию о последовательности фонем. Переменная mLength представляет собой продолжительность всей последовательности анимации в миллисекундах. Она полезна для проверки временных значений на превышения длины анимации.

После данных в классе cXPhonemeParser располагаются функции, первой из которых является типичная функция ParseObject, которую вы уже должны знать и любить. Помните, функция ParseObject вызывается каждый раз при перечислении объекта данных из файла .X, так что данные последовательности фонем загружаются в ней.

Т. к. мы ищем только объекты PhonemeSequence, функция ParseObject на данный момент будет небольшой. Начав с объявления GUID шаблона PhonemeSequence, можно переходить к фактическому коду ParseTemplate, чтобы увидеть что происходит.

```
// Определить GUID шаблона PhonemeSequence
DEFINE_GUID(PhonemeSequence,
       0x918dee50, 0x657c, 0x48b0,
       0x94, 0xa5, 0x15, 0xec, 0x23, 0xe6, 0x3b, 0xc9);

BOOL cXPhonemeParser::ParseObject( \
         IDirectXFileData *pDataObj, \
         IDirectXFileData *pParentDataObj, \
         DWORD Depth, \
         void **Data, BOOL Reference)
{
  const GUID *Type = GetObjectGUID(pDataObj);

  // Обработать только объекты последовательности фонем
  if(*Type == PhonemeSequence) {
   // Освободить текущую загруженную последовательность
   Free();

   // Получить имя и указатель на данные
   m_Name = GetObjectName(pDataObj);
   DWORD *DataPtr = (DWORD*)GetObjectData(pDataObj, NULL);

   // Получить количество фонем,создать структуры и загрузить данные
   m_NumPhonemes = *DataPtr++;
   m_Phonemes = new sPhoneme[m_NumPhonemes];
```

На данный момент мы получили имена анализируемых экземпляров объектов и количество ключевых кадров фонем последовательности. Далее создается массив структур sPhoneme, и обработка продолжается, просматривая каждый ключевой кадр в последовательности, получая номера мешей фонем, начального и конечного времени.

```
for(DWORD i=0;i<m_NumPhonemes;i++) {
  m_Phonemes[i].Code = *DataPtr++;
  m_Phonemes[i].StartTime = *DataPtr++;
  m_Phonemes[i].EndTime = *DataPtr++;
   }
  m_Length = m_Phonemes[m_NumPhonemes-1].EndTime + 1;
 }
```

```
// Анализировать дочерние объекты
return ParseChildTemplates(pDataObj, Depth, Data, Reference);
}
```

По мере создания функции ParseObject вы можете видеть, что сохраняется продолжительность пбследовательности анимации и перечисляются все дочерние объекты. Помните, что анализируется только первый объект последовательности фонем; любой объект, загружаемый после первого, уничтожает данные предыдущего. Опять же, для своего проекта вы можете захотеть добавить возможность загрузки более чем одной последовательности.

Далее в классе cXPhonemeParser следует функция Free. Эта функция освобождает данные класса, такие как массив ключевых кадров фонем. Я пропущу код этой функции и перейду к следующей, FindPhoneme, которая ответственна за нахождение номера меша фонемы среди ключевых кадров по заданному времени.

```
DWORD cXPhonemeParser::FindPhoneme(DWORD Time)
{
  if(m_NumPhonemes) {
   // Искать по времени
   for(DWORD i=0;i<m_NumPhonemes;i++) {
    if(Time >= m_Phonemes[i].StartTime && \
              Time <= m_Phonemes [i].EndTime)
      return i;
    }
   }
  return 0;
}
```

Функция FindPhoneme просто просматривает массив ключевых кадров фонем, ища тот, в который попадает заданное время. Функция FindPhoneme обычно вызывается функцией GetAnimData, которую вы используете для получения двух мешей фонем, необходимых для комбинирования, и значений комбинирования для них (находящихся в диапазоне от 0 до 1).

```
void cXPhonemeParser::GetAnimData( \
       DWORD Time, \
       DWORD *Phoneme1, float *Phoneme1Time, \
       DWORD *Phoneme2, float *Phoneme2Time)
{
  // Быстрая проверка на окончание анимации
  if(Time >= m_Length) {
   *Phoneme1 = m_Phonemes[m NumPhonemes-1].Code;
   *Phoneme2 = 0;
   *Phoneme1Time = 1.0f;
    *Phoneme2Time = 0.0f;
    return;
  }
```

Как вы можете видеть из последнего кусочка кода, необходимо вставить специальный случай проверки, минула ли анимация свою продолжительность. Если да, только последний ключевой кадр используется, при этом первый меш комбинируется на 100 процентов, а второй на 0.

Двигаясь далее, функция GetAnimData просматривает список фонем в поисках той, которая совпадает с заданным временем.

```
// Найти ключ, используемый в последовательности фонем
  DWORD Index1 = FindPhoneme(Time);
  DWORD Index2 = Index1+1;
  if(Index2 >= m_NumPhonemes)
  Index2 = Index1;

  // Установить номера индексов фонем
  *Phoneme1 = m_Phonemes[Index1].Code;
  *Phoneme2 = m_Phonemes[Index2].Code;
```

Фонема, найденная по заданному времени, используется в качестве первого комбинируемого меша. Второй комбинируемый меш берется из следующий структуры фонем. Опять же, если последовательность анимации находится в конце, используются значения только последнего ключевого кадра.

Вот здесь мы и возвращаемся к использованию трех мешей для морфирования от одной фонемы к другой. По мере того как последовательность анимируется от первого меша фонемы ко второму, значение комбинирования первого меша медленно уменьшается от 1 до 0. Второй меш комбинируется при начальном значении комбинирования, равном 0 и медленно увеличивающем до 1.

Для вычисления коэффициентов смешивания двух мешей необходимо добавить в конец функции GetAnimData несколько вычислений, которые масштабируют текущее время между временами двух используемых ключевых кадров и сохраняют значения комбинирования в качестве коэффициентов масштабированного времени.

```
  // Вычислить значения синхронизации
  DWORD Time1 = m_Phonemes[Index1].StartTime;
  DWORD Time2 = m_Phonemes[Index1].EndTime;
  DWORD TimeDiff = Time2 - Time1;
  Time -= Time1;

  float TimeFactor = 1.0f /  (float)TimeDiff;
  float Timing = (float)Time * TimeFactor;
  // Установить времена фонем
  *Phoneme1Time = 1.0f - Timing;
  *Phoneme2Time = Timing;
}
```

Вот и все с классом cXPhonemeParser! Чтобы его использовать, вам просто необходимо создать его экземпляр и вызвать Parse, используя .X файл, содержащий данные последовательности фонем.

```
cXPhonemeParser PhonemeParser;
PhonemeParser.Parse("Phoneme.x");
```

Для каждого кадра анимации вызывайте функцию GetAnimData для определения мешей и значений комбинирования, используемых для визуализации лицевого меша. Вам необходимо вычислить время в анимации, вызвав такую функцию, как например timeGetTime, и ограничив результат продолжительностью анимации, как я сделал тут:

```
// Получить время анимации
DWORD Time = (DWORD)timeGetTime();
Time %= PhonemeParser.m_Length;

// Получить номера мешей и значения комбинирования
DWORD Mesh1, Mesh2;
float Time1, Time2;
PhonemeParser.GetAnimData(Time,&Mesh1,&Time1,&Mesh2,&Time2);
```

Я думаю, вы уже знаете, что будет дальше. Используя значения, полученные с помощью вызова функции GetAnimData, вы теперь можете визуализировать комбинированный меш. Для получения деталей о рисовании комбинированных мешей обратитесь к главе 8 или к демонстрационной программе лицевой анимации, находящейся на компакт-диске в качестве работающего примера. Посмотрите конец этой главы для получения дополнительной информации о демонстрационной программе лицевой анимации.

Поздравляю! Вы успешно загрузили и проиграли последовательность анимации фонем! Что идет далее в списке? Вы можете комбинировать большее количество мешей для повышения реалистичности анимации, добавив моргание глаз или возможность изменять выражения лица. После того как вы изучили основы, все это легко реализуется.

Проигрывание лицевых последовательностей со звуком

После того как вы записали и воспроизвели последовательности лицевой анимации, пришло время сделать еще один шаг и начать синхронизировать эти анимации со звуком. Если вы используете лицевую анимацию для синхронизации губ со звуком, то это настолько же просто, как проигрывание звукового файла и анимации одновременно.

Имея столько возможностей загрузки и воспроизведения звуковых файлов, что такой программист DirectX как вы, намерен делать? Исключив множество доступных медиа библиотек, я хочу обратиться к лучшей на сегодняшний день - DirectShow

корпорации Microsoft! При ближайшем рассмотрении DirectShow является очень мощной медиа системой. К счастью для нас, она проста в использовании.

Если все, что вам нужно, это проигрывать медиа файлы (аудио медиа файлы, в данном случае), то тогда вам везет, потому что это очень просто выполняется при использовании DirectShow. В данном разделе я устрою вам головокружительный тур проигрывания аудио файлов при помощи DirectShow, таких как .WAV, .MP3, .WMA или любого другого звукового файла, имеющего зарегистрированный в Windows кодек. Конечно же, эти аудио файлы будут содержать записанные диалоги, которые вы хотите синхронизировать с лицевой анимацией.

Для получения детальной информации о DirectShow и используемых объектах проконсультируйтесь с главой 14 "Использование анимированных текстур". Как я уже сказал, это головокружительный тур, так что все будет происходить быстро!

Использование DirectShow для звука

DirectShow является набором интерфейсов и объектов, которые работают с видео и аудио медиа. Я говорю о записи и воспроизведении медиа из любого источника, включая живое видео, потоковое содержимое веб, DVD и предварительно записанные файлы. И как будто этого было недостаточно, DirectShow даже позволяет вам создавать собственные декодеры и кодеры медиа, делающие ее единственной системой, которую выбирают.

Для того чтобы добавить DirectShow в проект, необходимо включить заголовочный файл dshow.h в исходный код.

```
#include "dshow.h"
```

Также убедитесь, что добавили файл strmiids.lib к списку файлов, связываемых в проекте. Он расположен в той же директории, что и остальные библиотеки DirectX (обычно \dxsdk\lib). После того как вы подключили и привязали соответствующие файлы, вы можете создать экземпляры следующих четырех интерфейсов DirectShow для использования в вашем коде:

```
IGraphBuilder *pGraph = NULL;
IMediaControl *pControl = NULL;
IMediaEvent *pEvent = NULL;
IMediaPosition *pPosition = NULL;
```

Первый интерфейс IGraphBuilder, является основным. Он ответственен за загрузку и декодирование медиа файлов. Второй интерфейс, ImediaControl, управляет воспроизведением аудио файлов. Третий интерфейс, ImediaEvent, получает события,

такие как завершение воспроизведения. Последний интерфейс, ImediaPosition, устанавливает и получает положение, в которое происходит воспроизведение. (Например, вы можете проиграть пять секунд аудио файла или можете начать воспроизведение с 20 секунд от начала звука.)

Замечание. *Т. к. используется система COM, необходимо вызвать функцию CoInitialize внутри кода инициализации. После завершения программы вы должны вызвать CoUninitialize, чтобы деинициализировать систему COM.*

Для создания объекта IgraphBuilder (из которого получаются три остальных интерфейса) используйте функцию CoCreateInstance, как я сделал тут:

```
// Инициализировать систему COM
CoInitialize(NULL);

// Создать объект IGraphBuilder
CoCreateInstance(CLSID_FilterGraph,NULL,\
        CLSCTX_INPROC_SERVER, IID_IGraphBuilder, \
        (void**)&pGraph);
```

После создания объекта IGraphBuilder вы можете вызвать IGraphBuilding::RenderFile, чтобы сразу начать использовать его для загрузки аудио файла. (Это называется рендеринг.) По мере того как файл рендерится, DirectShow загружает все необходимые кодеки для декодирования данных.

Функция RenderFile принимает в качестве параметра имя проигрываемого медиа файла в виде широкосимвольной строки, которую можно создать с помощью макроса L. Например, для загрузки файла MeTalking.mp3 необходимо использовать следующий код. (Заметьте, что второй параметр RenderFile всегда устанавливается в NULL.)

```
pGraph->RenderFile(L"MeTalking.mp3",   NULL);
```

После того как вы загрузили медиа файл, вы можете получить оставшиеся три интерфейса из объекта IGraphBuilder, как я сделал здесь:

```
pGraph->QueryInterface(IID_IMediaControl,  (void**)&pControl);
pGraph->QueryInterface(IID_IMediaEvent,  (void**)&pEvent);
pGraph->QueryInterface(IID_IMediaPosition,(void**)&pPosition);
```

Уже почти все! Начать проигрывать звук можно простым вызовом IMediaControl::Run.

```
pControl->Run();
```

Вот и все. Если все прошло, как было запланировано, вы должны услышать проигрываемый звук! Теперь необходимо просто синхронизировать лицевую анимацию с проигрываемым звуком.

Синхронизация анимации со звуком

После того как вы загрузили и воспроизвели звук, пришло время синхронизировать его с анимацией. Т. к. анимация проигрывается в соответствии со временем (миллисекундами в данном случае), вы можете получить с помощью DirectSound точное время проигрываемого звука. Используя это значение времени, вы можете обновлять синхронизированную анимацию губ каждый кадр, делая ее таким образом великолепно синхронизированной со звуком.

Замечание. *Автоматические особенности, такие как моргание глаз мешa, должны выполняться в соответствии с внешним счетчиком, а не временем проигрывания DirectSound. Для получения значения внешнего счетчика (в миллисекундах) вы можете использовать функцию timeGetTime, которая возвращает значение DWORD, содержащее количество миллисекунд, прошедших с момента запуска Windows.*

Для получения времени проигрываемого звука используйте интерфейс IMediaPosition, созданный в предыдущем разделе. Функция, которую мы ищем, называется IMediaPosition::get_CurrentPosition и имеет только один параметр-указатель на REFTIME (вещественный тип данных), как вы можете видеть здесь:

```
REFTIME SndTime; // REFTIME = float
pPosition->get_CurrentPosition(&SndTime);
```

Для получения текущего времени воспроизведения просто умножьте значение REFTIME, полученное с помощью get_CurrentPosition на 1000.0 и преобразуйте результирующее число в переменную DWORD. Это преобразованное значение будет содержать время, используемое для обновления синхронизированной лицевой анимации губ. Следующий кусочек кода демонстрирует преобразования времени, полученного с помощью get_CurrentPosition в миллисекунды.

```
DWORD MillisecondTime = (DWORD)(SndTime * 1000.0f);
```

Вы используете значение времени (MillisecondTime) для нахождения соответствующей последовательности лицевых мешей фонем, которые применяются для морфирования данных анимации, загруженной с помощью анализатора .X файлов. Немного позднее вы увидите демонстрационные программы этой главы, в которых показано, как использовать эти значения времени в анимациях. (Смотрите конец главы для получения дополнительной информации о демонстрационной программе.)

Если вы забежали вперед и пробовали воспроизвести синхронизированные звук и анимацию, вы могли заметить, что что-то не так. В реальном мире, прежде чем произнести звук, человек двигает ртом. В текущем состоянии метод син-

хронизации не учитывает этот фактор. Однако не волнуйтесь, вам просто необходимо немного сместить время анимации для решения этой проблемы. Просто вычтите небольшое количества времени из анимации, я предлагаю от 30 до 80 миллисекунд, но вы можете попробовать варьировать это значение для повышения точности синхронизации.

Зацикливание воспроизведения звуков

Кажется чего-то не хватает... Хм, что же это может быть? Конечно же IMediaEvent, третий дополнительный интерфейс, который вы получили из объекта IgraphBuilder. Интерфейс IMediaEvent становится действительно полезным, если вам необходимо узнать, закончилось ли воспроизведение звука. (Это все, что нас заботит на данный момент.) Он полезен, если вы хотите определить, должен ли звук начаться сначала или должно ли произойти другое событие при окончании проигрывании звука.

События представлены командным кодом и двумя параметрами, приведенными к типу данных long.

```
long Code, Param1, Param2;
```

Для получения кода события и параметров вы можете вызвать IMediaEvent::GetEvent, как показано тут:

```
HRESULT hr = pEvent->GetEvent(&Code, &Param1, &Param2, 1);
```

Запись значения, возвращаемого IMediaEvent::GetEvent, является очень важным. Значение S_OK соответствует корректному получению события, а любое другое значение ошибки означает, что больше нет обрабатываемых событий. Т. к. обработки может ожидать любое количество событий, вам необходимо непрерывно вызывать GetEvent, пока все события не закончатся. Вы можете увидеть этот процесс здесь, содержащийся в цикле while.

Замечание. *Четвертым параметром функции IMediaEvent::GetEvent является количество миллисекунд, которые необходимо подождать перед получением события. Я всегда жду одну миллисекунду прежде чем продолжить.*

```
while(SUCCESS(pEvent->GetEvent(&Code,&Param1,&Param2,1))) {
  // Обработать событие завершения проигрывания
  if(Code == EC_COMPLETE) {
   // Сделать что-нибудь по завершению проигрывания
  }

  // Освободить данные события
  pEvent->FreeEventParams(Code,  Param1,  Param2);
}
```

Из комментариев к предыдущему коду вы можете увидеть, что я проверяю тип кода события, полученного вызовом GetEvent. Единственным интересующим нас событием является завершение проигрывания, которое представлено макросом EC_COMPLETE. Внутри блока условия вы можете делать все, что пожелаете. Например, вы можете переместиться в начало звука и начать проигрывать его, как я сделал тут:

```
// Обработать событие завершения проигрывания
if(Code == EC_COMPLETE) {
  // Переместиться в начало звука и проиграть его опять
  pPosition->put_CurrentPosition(0.0f);
  pControl->Run();
}
```

Вы также могли заметить вызов функции IMediaEvent::FreeEventParams. Вы должны всегда вызывать IMediaEvent::FreeEventParams, чтобы позволить DirectSound освободить все ресурсы, которые были выделены для получения событий при вызове функции GetEvent.

Реализовав это, функция проигрывания звука завершена! Ну, почти завершена. После завершения работы со звуком, вы можете вызвать IMediaControl::Stop для остановки проигрывания всех звуков и освободить все интерфейсы COM при помощи функции Release.

Посмотрите демонстрационные программы

Гмм! Лицевую анимацию невероятно просто использовать, если вы знаете как, а эффект от ее использования просто потрясающ. На компакт-диске книги вы обнаружите две программы, связанные с лицевой анимацией - FacialAnim и ConvLWV.

Первая программа, FacialAnim, является обычным проектом, поставляемым с исходным кодом. Демонстрационная программа FacialAnim, показанная на рис. 11.14, иллюстрирует мощь движка лицевой анимации, разработанного в этой главе.

Последняя программа, ConvLWV, помогает вам создавать собственные последовательности фонем, используемых в пакете лицевой анимации, разработанном в этой главе. Посмотрите текст этой главы для получения дополнительной информации об использовании ConvLWV.

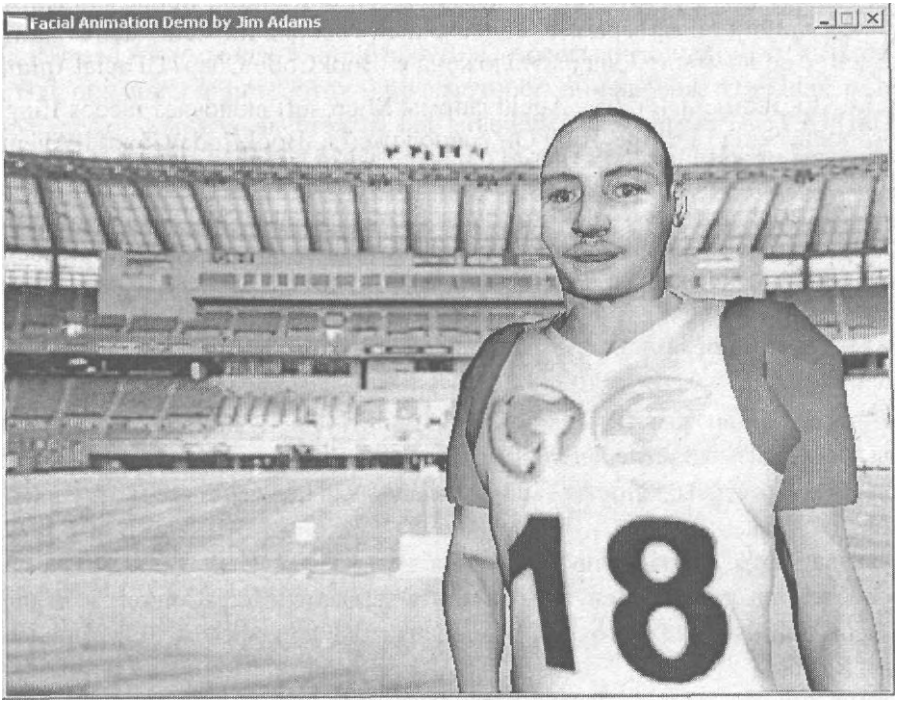


Рис. 11.14. Прослушайте футбольный репортаж игрового персонажа, с полностью синхронизированными губами, в демонстрационной программе FacialAnim!

После того как вы полностью завершите систему лицевой анимации, я предлагаю вам улучшить технологии, рассмотренные здесь, добавив персонажу новые детали, такие как зубы, волосы и брови. Добавьте анимацию глазам меша или увеличьте набор фонем. После того как вы закончите лицевые меши, поиграйте с текстурами. Текстуры являются очень важными в системе, потому что текстурированная модель всегда выглядит лучше обычной.

Программы на компакт-диске

Директория главы 11 компакт-диска включает проект (демонстрационную программу) FacialAnim и две программы (программный пакет Agent фирмы Microsoft и программу ConvLWV). В директорию входят:

- **FacialAnim.** Посмотрите на пример лицевой анимации, приведенный в этой демонстрационной программе, содержащей говорящий меш, меняющий свое поведение во времени. Она расположена в \BookCode\Chap11\FacialAnim.
- **Agent.** Программный пакет Agent фирмы Microsoft включает в себя Linguistic Information Sound Editing Tool. Он расположен в \BookCode\Chap11\Agent.
- **ConvLWV.** Эта программа преобразует файлы .LWV в .X и сохраняет последовательность фонем в объектах, легко импортируемых игровым проектом. По причинам лицензирования исходный код для этого проекта недоступен. Программа расположена в \BookCode\Chap11\ConvLWV.

Часть V

Прочие типы анимации

12. Использование частиц в анимации

13. Имитирование одежды и анимация мешей мягких тел

14. Использование анимированных текстур

Глава 12

Использование частиц в анимации

Яркий взрыв, клочья травы, валящиеся деревья, клубы дыма, кричащие пешеходы - что они могут иметь общего? Факт того, что вы можете нарисовать все это, используя частицы, вот что! Частицы являются основным специальным эффектом игр, позволяя отображать множество вещей, начиная от капель дождя и заканчивая сверкающими взрывами разлетевшейся шрапнели. Применяя небольшие ноу-хау и помощь этой главы, вы можете использовать эти простые маленькие частицы для существенного улучшения внешнего вида игры!

В этой главе вы научитесь:

Рисовать частицы, используя три различные технологии;

- Передвигать частицы с использованием скорости;
- Создавать классы, управляющие частицами, в вашей игре.

Работа с частицами

Сначала о главном - что такое частицы? Частица является простым графическим объектом, используемым в качестве средства улучшения внешнего вида. Обычно частицы используются для отображения небольших графических эффектов, таких как огонь, дым, небольшие искрящиеся огоньки, оставляемые за собой магической ракетой (см. рис. 12.1). Конечно же, ваша игра может работать и без частиц, но зато их присутствие делает ее намного ярче.

На самом деле, частицы применяются не только для маленьких искрящихся огоньков и клубов дыма. Некоторые игры используют частицы неожиданным образом. Например, серия игр Tekken фирмы Namco использует частицы для представления летящих кусков травы в некоторых уровнях.

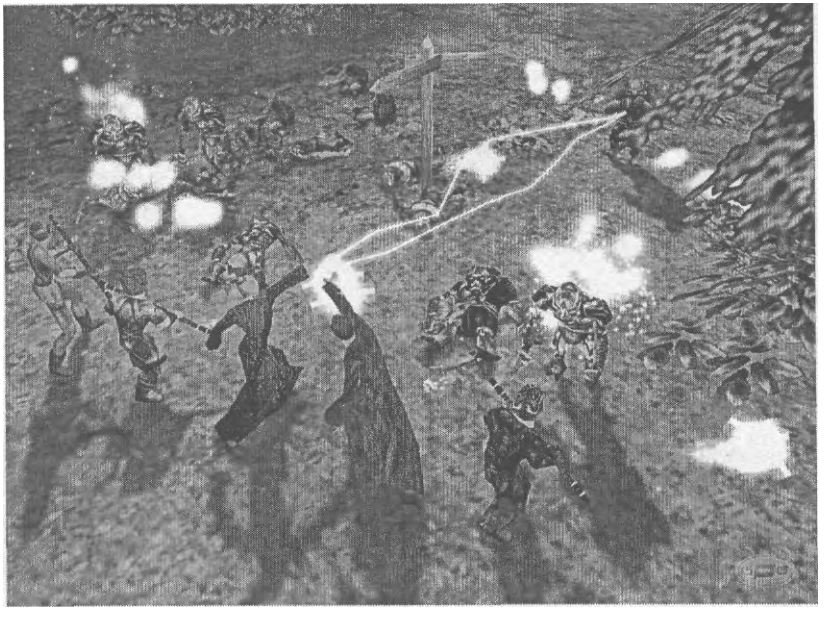


Рис. 12.1. Взрыв, вызванный заклинанием, создает ливень частиц в игре Dungeon Siege фирмы Gas Powered Games

Какие еще нестандартные применения могут быть найдены частицам? Представьте - ваша новейшая игра состоит из гигантских монстров, которые опустошают различные города нации. В одном из уровней игры какой-нибудь из монстров может сломать дамбу. После нескольких точных попаданий дамба рушится и вода начинает вытекать, отскакивая от чешуйчатой спины титанического создания и заливая город. Где же в этом сценарии частицы?

Уверен, вы скажете вода, да? Вы правы, но здесь также присутствуют кусочки отлетающей дамбы, испуганные жители, покидающие городские здания, взрывающиеся трансформаторы затопленных силовых линий и, наверное даже, случайная пуля, выпущенная в монстра храбрым гражданином. Я не говорю о тех частях уровня, в которых присутствуют биллборды[1], используемые для визуализации разнообразных городских знаков, деревьев, машин, которые также являются частицами.

1. Биллборд - графический примитив, вставляемый в трехмерную сцену так, чтобы одна его сторона всегда была повернута к наблюдателю. - *Примеч. науч. ред.*

Вот и все - вы можете добавить набор простых частиц в вашу игру, таким образом улучшив ее внешний вид и при этом прикладывая минимум усилий. Замечательным является то, что вы можете управлять большей частью частиц одним или двумя небольшими классами, которые очень просто добавить в игровые проекты.

Однако прежде чем перейти к использованию классов для управления частицами, необходимо усвоить основы. Сначала необходимо посмотреть, как рисовать частицы, а потом уже двигаться дальше.

Основы

Частицы обычно представлены в виде квадратных полигонов, которые визуализируются, используя небольшой набор текстур. Используя текстурированные квадратные полигоны, вы можете обманывать людей, заставляя их думать, что частицы на самом деле являются трехмерным мешем; этот эффект создает карта текстуры. Это очень важно, потому что если вы собираетесь показывать только одну сторону меша, то вы можете визуализировать его в карту текстуры, а потом наложить ее на квадрат.

Т. к. частица визуализируется при использовании двух треугольников (образующих квадрат), как показано на рис. 12.2, необходимо основываться на визуализации биллборда, чтобы убедиться, что многоугольники всегда направлены к смотрящему.

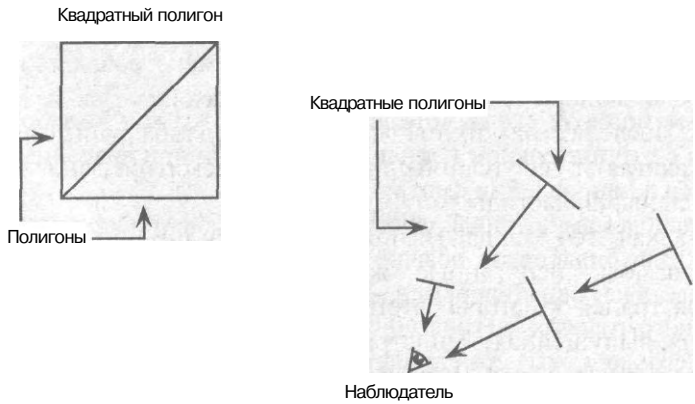


Рис. 12.2. Два треугольника соединены вместе, образуя квадрат. Смотря сверху вниз, вы можете видеть, что использование биллбординга позволяет увериться, что многоугольники всегда направлены к зрителю

В случае, если вы не слышали этого термина ранее, биллбординг это технология ориентирования объекта (такого как многоугольник) таким образом, чтобы он всегда был направлен на смотрящего. Сначала объект биллбординга создается направленным в отрицательном направлении оси z (как показано на рис. 12.3). По мере того, как смотрящий двигается, объект биллбординга вращается таким образом, чтобы быть всегда направленным к смотрящему.

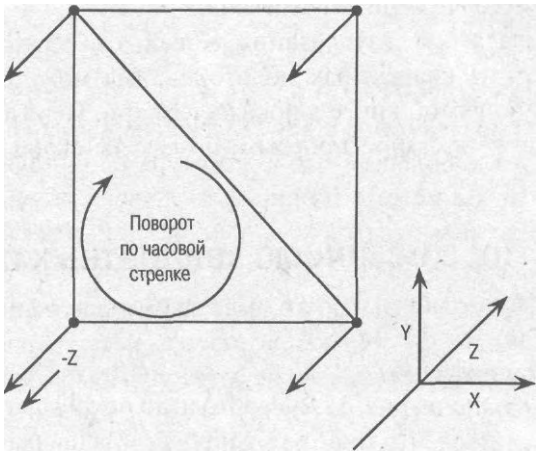


Рис. 12.3. Квадратный полигон слева, который используется как биллборд, изначально ориентирован в отрицательном направлении оси z, и при визуализации он просто поворачивается в направлении смотрящего

Квадратный полигон слева, который используется как биллборд, изначально ориентирован в отрицательном направлении оси z, и при визуализации он просто поворачивается в направлении смотрящего.

Причины использования биллбордов очень просты - экономия памяти и ускорение визуализации. Для биллбордов обычно используется растровая картинка того, как должны выглядеть частицы (например, клуб дыма для частицы дыма), и эта картинка рисуется на многоугольник частицы. Далее, вместо визуализации трехмерного меша, представляющего дым, вы можете нарисовать многоугольник частицы, используя картинку дыма в качестве текстуры.

Однако стоит сказать, что визуализации трехмерного меша частицы не является плохой мыслью. На самом деле, частица может состоять из любого типа примитива, от пикселей и линий до многоугольников и целого трехмерного меша. Можно полагать, что бывают случаи, когда простого квадратного полигона недостаточно и придется использовать трехмерные меши.

Возвращаясь к игре с монстрами, можно использовать частицы для представления маленьких машин, движущихся по улицам города. Простого текстурированного (с использование биллбординга) квадратного полигона будет достаточно для этих машин, а что если вместо этого использовался бы трехмерный меш? Он просто бы потребовал визуализации нескольких дополнительных многоугольников в каждом кадре, и, поверьте мне, эффект от этих маленьких машин, движущихся по городу, стоил бы дополнительного времени визуализации.

Хорошо, если не касаться визуализации мешей, то большая часть рисуемых частиц будет состоять из квадратных полигонов. Вы можете рисовать их различными способами, но в этой книге я покажу вам три. Первый метод рисования частиц, наверное, является самым простым, но как вы скоро увидите, он имеет свои недостатки.

Рисование частиц с помощью квадратных полигонов

Рисование частиц настолько же просто, насколько и рисование многоугольников, потому что частица - это просто меняющий размеры текстурированный квадратный полигон. На рис. 12.4 показан внешний вид двух треугольников, которые используются для создания частиц. На этом рисунке я показал частицу, имеющую размер 10 единиц.

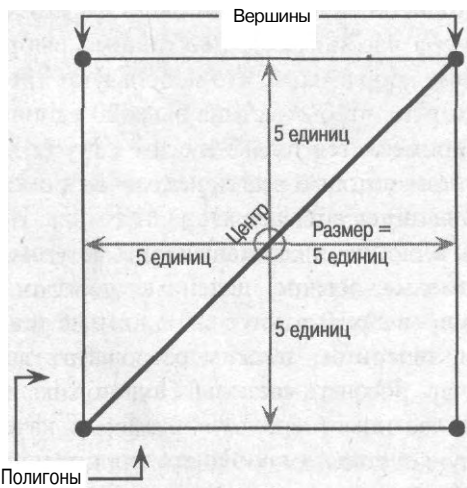


Рис. 12.4. Частица, имеющая размер 10 единиц, имеет протяженность 5 единиц в направлениях осей x и y

В дополнение к координатам вершин, используемым при рисовании частицы, необходима еще пара вещей. Во-первых, необходима текстура, используемая для улучшения внешнего вида частицы. Является ли эта текстура дымом, сгустком света или испуганным человеком, вам необходимо хранить картинку в объекте текстуры. Кроме текстур многоугольникам необходима информация о текстурных координатах для наложения текстуры на многоугольник.

Для упрощения, я буду использовать только одну текстуру для каждого типа частиц в этой главе. Другими словами, если имеются частицы дыма и огня, я загружу две текстуры. Любой экземпляр любой частицы будет использовать собственную текстуру при визуализации. Возможно вы захотите объединить все текстуры частиц в одну, для улучшения частиц.

Следующее, что вам необходимо для создания данных многоугольника частиц, это рассеянная (diffuse) составляющая цвета. Возможность изменять цвет частиц во время выполнения очень полезна, потому что изменения цвета могут представлять различные циклы жизни частиц. Например, огонь медленно охлаждается и меняет свой цвет по мере того, как он удаляется от источника тепла. Вместо использования множества текстур для представления различных уровней нагрева вы можете использовать одну и ту же текстуру и просто менять рассеянный цвет частицы.

Последнее, что вам нужно, это размер частицы. В общем, частицы могут иметь любой размер, от небольшого пятна пыли до метеора, крушащего Землю. Для определения размера частицы вам необходимо выбрать габариты частицы, используя те же мировые координаты, что используют трехмерные меши.

Предположим, вы хотите, чтобы частица была 20 единиц в ширину и 50 единиц в высоту. Этот размер применяется только к осям x и y (т. к. на самом деле частица является плоским объектом, который всегда направлен к смотрящему). При создании частицы ее центр располагается в начале координат мира. Используя размеры частицы, вы можете создать вершины, представляющие ее углы. Эти вершины располагаются, используя размеры частицы, деленные пополам, в качестве смещения. Например, частица размеров 20x50 лежит от -10 до 10 на оси x и от -25 до 25 на оси y.

Вот и все об этом, теперь вы должны располагать достаточным количеством информации, чтобы начать рисовать частицы! Подытожим: необходимо иметь размер и координаты вершин частицы (используя размер в качестве базы), текстурные координаты и рассеянную составляющую цвета для каждой вершины. Смелее, объедините все эти компоненты в одну структуру вершин, как я сделал тут:

```
typedef struct {
  D3DXVECTOR3 vecPos; // Координаты вершины частицы
  D3DCOLOR Diffuse; // Рассеянный цвет
```

```
  float u, v; // Текстурные координаты
} sVertex;
```

Не забудьте определять объявление FVF для только что созданной структуры.

```
#define VERTEXFVF (D3DFVF_XYZ | D3DFVF_DIFFUSE | D3DFVF_TEX1)
```

Используя структуру вершин и FVF, вы можете создать небольшой буфер вершин, который содержал бы достаточно вершин для рисования одной частицы. Для частицы используются два треугольника, это означает, что потребуется создать шесть вершин (три для каждого треугольника). Используя полосу треугольников, можно сократить количество вершин до четырех.

Буфер вершин можно создать, используя следующий код:

```
IDirect3DVertexBuffer9 *pVB = NULL;
pDevice->CreateVertexBuffer(4*sizeof(sVertex), \
         D3DUSAGE_WRITEONLY, \
         VERTEXFVF, D3DPOOL_DEFAULT, \
         &pVB, NOLL);
```

После того как вы создали буфер вершин (он создается один раз в программе), вы можете заполнить его данными частицы. Предположим, вы хотите нарисовать частицу, имеющую размер 10 единиц (и в направлении оси x и в направлении оси y), используя всю поверхность текстуры, имеющую белый цвет. Для этого вам необходимо использовать следующий код:

```
// Size = размер частицы, в данном случае 10.0
float Size = 10.0;

// Получить размер,деленный пополам, для установки координат вершин
float HalfSize = Size / 2.0f;

// Заблокировать буфер вершин и заполнить его данными вершин
sVertex *Vertex = NULL;
pVB->Lock(0, 0, (void**)&Vertex, 0);

// Верхний левый угол, вершина номер 0
pVB[0].vecPos = D3DXVECTOR3(-Size, Size, 0.0f);
pVB[0].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
pVB[0].u = 0.0f; pVB[0].v = 0.0f;

// Верхний правый угол, вершина номер 1
pVB[1].vecPos = D3DXVECTOR3(Size, Size, 0.0f);
pVB[1].Diffuse = D3DCOLOR_RGBA(255,255,255,255) ;
pVB[1].u = 1.0f; pVB[1].v = 0.0f;

// Нижний левый угол, вершина номер 2
pVB[2].vecPos = D3DXVECTOR3(-Size, -Size, 0.0f);
pVB[2].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
pVB[2].u = 0.0f; pVB[2].v = 1.0f;
```

```
// Нижний правый угол, вершина номер 3
pVB[3].vecPos = D3DXVECTOR3(Size, -Size, 0.0f);
pVB[3].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
pVB[3].u = 1.0f; pVB[3].v = 1.0f;

// Разблокировать буфер вершин
pVB->Unlock();
```

Теперь вершинный буфер готов к визуализации. Однако есть небольшая тонкость. Вы заметите, что координаты вершин располагают многоугольники в центре координат трехмерного мира, расширяясь в направлении осей x и y. Т. к. точка обзора может находится где угодно в мире, необходимо расположить многоугольники, используя преобразование мира перед визуализацией.

Также необходимо повернуть многоугольники таким образом, чтобы они были направлены на смотрящего, что, как вы помните, является целью биллбординга. Необходимо вычислить биллборд-преобразование, чтобы повернуть многоугольники к смотрящему. Используя это преобразование, вы добавляете координаты частицы в место, где она должны быть нарисована (в мировых координатах).

Предупреждение. *Использование функции GetTransform для получения преобразования Direct3D сильно замедляет работу и не всегда возможно. Лучше всего хранить глобальные преобразования мира, вида и проекции, которые можно потом использовать. Однако пока я последую плохим путем и буду использовать GetTransform в учебных целях.*

Для создания биллборд-преобразования необходимо получить матрицу преобразования вида и вычислить обратную ей матрицу (таким образом меняя порядок содержащихся в ней преобразований на обратный), используя функцию D3DXMatrixInverse, как показано тут:

```
// Получить матрицу преобразования вида и обратить ее
D3DXMATRIX matView;
pDevice->GetTransform(D3DTS_VIEW, &matView);
D3DXMatrixInverse(&matView, NULL, &matView);
```

Использование обратного преобразования позволит вращать вершины частицы в направлении, обратном направлению вида, таким образом выравнивая координаты к смотрящему. После получения обратного преобразования вида необходимо непосредственно добавить координаты частицы для расположения ее в трехмерном мире. Вы можете сделать это, сохранив координаты x, y и z в элементах _41, _42 и _43 соответственно только что вычисленной обратной матрице преобразования и установив результирующую матрицу преобразования в качестве мирового преобразования.

```
// Положим ParticleXPos,ParticleYPos и ParticleZPos содержат мировые

// координаты рисуемой частицы. Добавить координаты рисуемой частицы
matView._41 = ParticleXPos;
matView._42 = ParticleYPos;
matView._43 = ParticleZPos;

// Установить результирующую матрицу в качестве преобразования мира
pDevice->SetTransform(D3DTS_WORLD, &matView);
```

После того как вы установили матрицу преобразования мира, можно визуализировать многоугольники. Для того чтобы частицы комбинировались со сценой, необходимо убедиться, что используется z-буфер и альфа-тестирование. Использование z-буфера позволяет убедиться, что частицы корректно отображаются в сцене, а альфа-тестирование позволяет убедиться в том, что прозрачные части текстуры частицы не рисуются (позволяя видеть оставшуюся геометрию через эти прозрачные части). Также вы можете включить альфа смешивание, если хотите создать некоторые эффекты смешивания цветов.

Пропустив установку z-буфера (вы должны были сделать ее ранее), вы можете включить альфа тестирование и альфа смешивание так:

```
// Включить альфа тестирование
pDevice->SetRenderState(D3DRS_ALPHATESTENABLE,   TRUE);
pDevice->SetRenderState(D3DRS_ALPHAREF,   0x08);
pDevice->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);

// Включить альфа смешивание (простого добавочного типа)
pDevice->SetRenderState(D3DRS_ALPHABLENDENABLE,   TRUE);
pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);
pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_DESTCOLOR);
```

Что касается альфа-тестирования, я выбрал использования правила сравнения больше или равно. Это означает, что если пиксели текстуры частицы имеют значение альфа большее или равное 8, то они визуализируются, в противном же случае они пропускаются. Использование альфа-тестирования означает необходимость использования функции D3DXCreateTextureFromFileEx для загрузки текстур, задав цветовой режим, использующий альфа каналы (такой как D3DFMT_A8R8G8B8), и цветовой ключ непрозрачного черного (D3DCOLOR_RGBA(0,0,0,255)), как показано в следующем кусочке кода:

```
D3DXCreateTextureFromFileEx(
    pDevice,
    "Particle.bmp",
    D3DX_DEFAULT, D3DX_DEFAULT, D3DX_DEFAULT,
    0, D3DFMT_A8R8G8B8, D3DPOOL_DEFAULT,
    D3DX_DEFAULT, D3DX DEFAULT,
    D3DCOLOR_RGBA(0,0,0,255), NULL, NULL,
    &pTexture);
```

После того как вы установили соответствующие альфа состояния, вы можете установить FVF, потоки, текстуры и визуализировать сцену!

```
// Установить вершинный шейдер и источники потоков
pDevice->SetVertexShader(NULL);
pDevice->SetFVF(PARTICLEFVF);
pDevice->SetStreamSource(0, pBuffer, 0, sizeof(sVertex));

// Установить текстуру
pDevice->SetTexture(0, &pTexture);

// Нарисовать частицу
pDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
```

Хорошо, уже что-то получается! Вы нарисовали частицу! Т. к. обычно одновременно визуализируется более одной частицы, нет нужды повторять весь этот код для каждой частицы. После того как вы установили поток и текстуру, вы можете периодически блокировать буфер вершин, устанавливать данные, разблокировать буфер и визуализировать частицу. Если размер частицы не меняется, вы можете избежать блокировки и разблокировки буфера вершин, а просто хранить координаты частицы в обратной матрице преобразования, установив соответствующую текстуру, после чего визуализировать частицу.

После того как вы ознакомитесь с кодом визуализации частицы, заметьте несколько вещей. Во-первых, существует множество обрабатываемых данных вершин. Во-вторых, постоянное блокирование, заполнение, разблокирование буфера вершин может сильно снизить производительность. Вам необходимо визуализировать частицы таким образом, чтобы использовать меньше памяти и делать это более быстро. Вам необходимы точечные спрайты!

Замечание. *Демонстрационная программа Particles, расположенная на компакт-диске, иллюстрирует визуализацию большого количества частиц, используя только-что рассмотренную технологию. Смотрите конец этой главы для получения дополнительной информации о программе Particles.*

Работа с точечными спрайтами

Как вы можете видеть, частицы являются важной частью игр — многие игры используют их для создания разнообразных спецэффектов. Корпорация Microsoft прекрасно об этом знала и добавила возможность визуализации биллбордных квадратных полигонов в Direct3D при использовании точечных спрайтов. Точечный спрайт является простым биллдным квадратным полигоном, содержащим минимальный набор данных. Каждая частица представлена одной трехмерной точкой

(соответствующей ее центру) и размером (как он вводился в предыдущей части, за исключением того, что он одинаков для обоих осей x и y). Это позволяет сэкономить огромное количество памяти по сравнению с использованием квадратов.

Помните, ранее в этой главе упоминалось, что квадратная частица требует четырех вершин. В предыдущей структуре sVertex использовалось двадцать вещественных значений и четыре значения DWORD на одну частицу. Итого: 96 байт данных для визуализации одной частицы! А как насчет точечных спрайтов?

Точечные спрайты используют следующую структуру вершин:

```
typedef struct {
  D3DXVECTOR3 vecPos; // Координаты центра частицы
  float Size; // Размер частицы
  D3DCOLOR Diffuse; // Рассеянный  цвет
} sPointSprite;
```

Выглядит очень похоже, разве нет?

Большим отличием является то, что для одной частицы необходима только одна структура sPointSprite. Так и есть, структура sPointSprite использует только 20 байт данных, что меньше чем sVertex на 76 байт. Какая экономия!

Так в чем же хитрость точечных спрайтов? Вы же знаете, что должны быть какие-нибудь недостатки, не так ли? Недостатком точечных спрайтов является ограниченность их размера. При использовании квадратов вы можете создавать частицы любого размера, точечные спрайты ограничены в максимальном размере, определяемом переменной D3DCAPS9::MaxPointSize. Это означает, что максимальный размер точечных спрайтов зависит от видео карты конечного пользователя.

Другим недостатком является то, что драйвер видео карты должен поддерживать точечные спрайты. Часто я встречал плохие драйверы, при использовании которых точечные спрайты при визуализации моргали либо имели неправильный размер. Вам остается только надеяться, что конечный пользователь обновит свои драйверы, что гарантирует корректное использование точечных спрайтов.

Замечание. Для *того чтобы убедиться, что вы можете аппаратно визуализировать точечные спрайты, необходимо проверить возможности аппаратного драйвера при помощи функции IDirect3D9::GetDeviceCaps. Если значение D3DCAPS9::MaxPointSize, полученное после вызова этой функции, установлено в 1.0, то тогда точечные спрайты не поддерживаются аппаратно.*

Не будем брать во внимание недостатки, - точечные спрайты сохраняют огромное количество памяти; для тех видео карт, которые могут их использовать, это является превосходным решением для рисования частиц. Как вы видели в структуре вершин sPointSprite, точечные спрайты используют только четыре вещественных значения и одно DWORD для хранения координат, размера и рассеянного цвета частицы соответственно.

Замечание. *Точечные спрайты используют всю поверхность текстуры, на которую рисуются. Это означает, что вам необходима одна текстура на каждую используемую картинку частицы. Это также означает, что вам не надо задавать текстурные координаты в структуре вершин точечного спрайта.*

Точечные спрайты используют следующее объявление FVF:

```
#define POINTSPRITEFVF (D3DFVF_XYZ|D3DFVF_PSIZE|D3DFVF_DIFFUSE)
```

При вызове функции CreateVertexBuffer необходимо указать флаг D3DUSAGE_POINTS, как показано в следующем кусочке кода:

```
pDevice->CreateVertexBuffer(8 * sizeof(sPointSprite), \
        D3DUSAGE_POINTS | D3DUSAGE_WRITEONLY, \
        POINTSPRITEFVF, D3DPOOL_DEFAULT, \
        &pBuffer, 0);
```

После того как вы создали буфер вершин (не забыв указать количество содержащихся в буфере вершин), вы можете начать заполнять его данными частиц, которые вы хотите визуализировать. Предположим, вы хотите визуализировать восемь частиц, каждая из которых имеет размер 10 единиц (занимая по 5 единиц в направлениях осей x и y) и использует белый рассеянный цвет. Используя только что созданный буфер вершин, вы можете заблокировать, заполнить и разблокировать его, используя следующий код:

```
float Size = 10.0f; // Сделать размер частиц равным 10 единицам

sPointSprite PointSprites[8] = {
  // Частица #0
  { D3DXVECTOR3(0.0f,0.0f,0.0f), Size,
    D3DCOLOR_RGBA(255,255,255,255) },
  // Частица #1
  { D3DXVECTOR3(10.0f,0.0f,0.0f), Size,
    D3DCOLOR_RGBA(255,255,255,255) },
  // Частица #2
  { D3DXVECTOR3(22.0f,0.0f,0.0f), Size,
    D3DCOLOR_RGBA(255,255,255,255) },
  // Частица #3
  { D3DXVECTOR3(30.0f,0.0f,0.0f), Size,
    D3DCOLOR_RGBA(255,255,255,255) },
  // Частица #4
  { D3DXVECTOR3(-10.0f,0.0f,0.0f), Size,
    D3DCOLOR_RGBA(255,255,255,255) },
  // Частица #5
  { D3DXVECTOR3(-20.0f,0.0f,0.0f), Size,
    D3DCOLOR_RGBA(255,255,255,255) },
  // Частица #6
  { D3DXVECTOR3(-30.0f,0.0f,0.0f), Size,
    D3DCOLOR_RGBA(255,255,255,255) },
```

```
    // Частица #7
    { D3DXVECTOR3(-40.0f,0.0f,0.0f), Size,
      D3DCOLOR_RGBA(255,255,255,255) },
};

// Заблокировать буфер вершин
sPointSprite *Ptr;
pBuffer->Lock(0,0,(void**)&Ptr,0);

// Скопировать данные вершин в буфер
memcpy(Ptr, PointSprites, sizeof(PointSprites));

// Разблокировать буфер вершин
pBuffer->Unlock();
```

После того как вы создали и заполнили буфер вершин данными точечного спрайта, можно визуализировать его. Подождите! Я забыл упомянуть об установке важных состояний визуализации. Direct3D должно знать, что вы используете точечные спрайты, расположенные в трехмерном пространстве (в противоположность экранному пространству). Для этого вы должны установить соответствующие состояния визуализации, как показано тут:

```
// Использовать всю текстуру для визуализации точечного спрайта
pDevice->SetRenderState(D3DRS_POINTSPRITEENABLE,   TRUE);

// Масштабировать в пространстве камеры
pDevice->SetRenderState(D3DRS_POINTSCALEENABLE,   TRUE);
```

Также необходимо установить минимальный размер точечного спрайта и насколько большим его делать, если в объявлении вершин отсутствует размер. На данный момент я установлю, чтобы точечные спрайты использовали размер 1, если в объявлении вершин отсутствуют необходимые данные, и чтобы минимальный размер был 0.

```
// Установить минимальный и определяемый по умолчанию размер
точечного спрайта
pDevice->SetRenderState(D3DRS_POINTSIZE, FLOAT2DWORD(1.0f));
pDevice->SetRenderState(D3DRS_POINTSIZE_MIN, FLOAT2DWORD(0.0f));
```

Наконец необходимо установить несколько масштабируемых коэффициентов затухания, основанных на расстоянии. Direct3D сможет изменять размер частиц в зависимости от их удаленности от смотрящего. Здесь не нужно ничего необычного, так что значения, установленные по умолчанию (заданные в документации DirectX SDK) подойдут. Эти коэффициенты (которые фактически являются состояниями визуализации) устанавливаются при помощи следующего кода:

```
// Определить функцию, преобразующую float в DWORD
inline DWORD FLOAT2DWORD(FLOAT f) { return *((DWORD*)&f); }
```

```
// Установить значения затухания для масштабирования
pDevice->SetRenderState(D3DRS_POINTSCALE_A, FLOAT2DWORD(1.0f));
pDevice->SetRenderState(D3DRS_POINTSCALE_B, FLOAT2DWORD(0.0f));
pDevice->SetRenderState(D3DRS_POINTSCALE_C, FLOAT2DWORD(0.0f));
```

Вы заметите кое-что забавное в последнем кусочке кода - использование функции FLOAT2DWORD. Как вы знаете, функция SetRenderState принимает только DWORD в качестве второго параметра. Значения коэффициентов затухания являются вещественными, так что приходится приводить их к типу DWORD. Вот здесь то и появляется FLOAT2DWORD. Используя эту функцию, вы можете указывать любое вещественное значения в качестве параметра функции SetRenderState и быть уверенным, что оно правильно преобразуется в DWORD.

Наконец можно визуализировать точечные спрайты! Помните, точечные спрайты являются обычными буферами вершин, использующими примитив типа точечный спрайт, указываемый флагом D3DPT_POINTLIST при вызове функции DrawPrimitive. Не отвлекаясь, вот код, позволяющий включить альфа-тестирование и смешивание, установить FVF и источники потоков и вызвать функцию DrawPrimitive.

```
// Включить альфа-тестирование
pDevice->SetRenderState(D3DRS_ALPHATESTENABLE,   TRUE);
pDevice->SetRenderState(D3DRS_ALPHAREF,   0x08);
pDevice->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);

// Включить альфа-комбинирование (простого добавочного типа)
pDevice->SetRenderState(D3DRS_ALPHABLENDENABLE,   TRUE);
pDevice->SetRenderState(D3DRS_SRCBLEND,  D3DBLEND_SRCCOLOR);
pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_DESTCOLOR);

// Установить вершинный шейдер и источник потока
pDevice->SetVertexShader(NULL);
pDevice->SetFVF(POINTSPRITEFVF);
pDevice->SetStreamSource(0, pBuffer, 0, sizeof(sPointSprite));

// Визуализировать точечный спрайт (восемь спрайтов)
pDevice->DrawPrimitive(D3DPT_POINTLIST, 0, 8);
```

Как вы можете видеть, работать с точечными спрайтами очень легко. По крайней мере намного проще, чем иметь дело с биллбордными квадратными полигонами, в спрайтах используется меньше данных. Единственной проблемой является то, что точечные спрайты ограничены в размерах и не полностью поддерживаются всеми видео-картами. Было бы очень хорошо, если бы была возможность использовать большие частицы и скорость визуализации точечных спрайтов, не правда ли? Замечательные новости - вы можете получить и то и другое при использовании вершинных шейдеров!

Улучшения визуализации частиц при помощи вершинных шейдеров

Что бы я мог показать такого, что бы улучшило возможности визуализации частиц? Скажу вам прямо: используя вершинные шейдеры, вы можете смешать простоту использования квадратов и скорость визуализации точечных спрайтов.

Ранее, в разделе "Рисование частиц с помощью квадратных полигонов", я показывал, как рисовать частицу, устанавливая координаты четырех ее вершин, образующих квадрат, и потом используя матрицу обратного преобразования вида, смешанную с мировыми координатами частицы, таким образом визуализируя многоугольники. За один раз рисовалась одна частица, т. е. каждый раз приходилось блокировать, заполнять и разблокировать буфер вершин.

Даже если частица не меняла свой размер, т. е. не было необходимости блокировать и разблокировать буфер вершин для каждой рисуемой частицы, все равно приходилось каждый раз изменять и устанавливать матрицу преобразования, чтобы убедиться, что частицы находятся в правильном положении в трехмерном мире перед визуализацией. Подумайте об этом - тысяча частиц означает тысячу вызовов функции SetTransform!

Т. к. вершинные шейдеры работают параллельно с конвейером визуализации, нет нужды каждый раз иметь дело с преобразованиями. Все правильно - больше нет нужды рисовать только одну частицу за один раз! Используя вершинные шейдеры, вы можете заполнить буфер вершин каким угодно количеством вершин и рисовать сразу набор частиц за один вызов. Это означает, что скорость частиц буфера вершин будет такой же, как и скорость точечных спрайтов!

При использовании вершинных шейдеров структуры вершин очень похожи на структуры точечных спрайтов. На рис. 12.5 и следующей далее структуре вершин показаны координаты центра частицы в трехмерном мире, рассеянный цвет и текстурные координаты. Единственной разницей между следующей далее структурой вершин и используемой в точечных спрайтах является хранимый немного по другому размер частицы.

```
typedef struct {
  D3DXVECTOR3 vecPos; // Координаты частицы
  D3DXVECTOR2 vecOffset; // Координаты смещения вершины
  DWORD Diffuse; // Рассеянный цвет частицы
  float u, v; // Текстурные координаты
} sShaderVertex;
```

Рис. 12.5. При использовании вершинного шейдера частица состоит из четырех вершин, определяемых центральной точкой, смещением от центра, рассеянным цветом и текстурными координатами

Рис. 12.5 показывает, что каждая вершина определяется расстоянием от центра частицы, называемым смещением. Смещение хранится в объекте D3DXVECTOR2, причем каждая компонента вектора относится к соответствующей оси (vecOffset.x для оси x и vecOffset.y для оси y).

Значения вектора смещения являются аналогом переменных размера в стандартных многоугольниках и точечных спрайтах; они определяют размер частицы по каждой оси. Например, значение смещения по оси x 10 означает, что частица лежит от -10 до 10 по оси x (при этом результирующая ширина частицы равна 20 единицам). То же самое справедливо и для оси y; значение смещения может быть любым числом, определяющим размер частицы по оси y. Обычно эти значения устанавливаются одинаковыми по обеим осям для получения квадратных частиц.

Для создания частицы, используя только что объявленную структуру, создается буфер вершин, содержащий достаточно вершин для каждой визуализируемой частицы. При использовании списков треугольников, когда каждая частица использует шесть вершин и буферов индексов, вы можете сократить количество вершин до четырех.

Я вернусь к буферу индексов немного позднее. А пока я хочу поговорить о буферах вершин. Т. к. вы используете вершинные шейдеры для визуализации частиц, для создания буфера вершин необходимо создать объявление элементов вершин и привязать компоненты структуры вершин к соответствующим регистрам шейдера. Для пользователей DirectX9 это означает создание массива структур D3DVERTEXELEMENT9, как показано тут:

```
D3DVERTEXELEMENT9 ParticleDecl[] =
{
  // Координаты вершины - D3DXVECTOR3 vecPos
  { 0, 0, D3DDECLTYPE_FLOAT3, D3DDECLMETHOD_DEFAULT, \
          D3DDECLUSAGE_POSITION, 0 },
  // Смещение угла вершины D3DXVECTOR2 vecOffset
  { 0, 12, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, \
           D3DDECLUSAGE_POSITION, 1 },
  // Рассеянный цвет - DWORD Diffuse
  { 0, 20, D3DDECLTYPE_D3DCOLOR, D3DDECLMETH0D_DEFAULT, \
           D3DDECLUSAGE_C0L0R, 0 },
  // Текстурные координаты - float u, v
  { 0, 24, D3DDECLTYPE_FLOAT2, D3DDECLMETHOD_DEFAULT, \
           D3DDECLUSAGE_TEXCOORD, 0 },
  D3DDECL_END()
};
```

Объявление элементов вершин ParticleDecl является простым - оно связывает компоненты структуры вершины и соответствующие части вершинного шейдера. Во-первых, две компоненты положения (трехмерные координаты и вектор смещения угла). После них следует рассеянная составляющая цвета и текстурные координаты. Каждая компонента использует индекс #0, за исключением вектора смещения угла, использующего индекс #1 в типе положения.

Элементы вершин станут более понятны, когда вы увидите вершинный шейдер. А пока, просто создайте буфер вершин и заполните его данными частицы. Предположим, вы хотите разместить четыре частицы, которые образуют шестнадцать вершин.(четыре вершина на каждую частицу, причем каждая вершина соответствует углу частицы). Следующий кусочек кода создает буфер вершин, используя объявленные выше элементы вершин:

```
IDirect3DVertexBuffer9 *pVB = NULL;
pDevice->CreateVertexBuffer(16 * sizeof(sShaderVertex), \
         D3DUSAGE_WRITEONLY, 0, \
         D3DPOOL_DEFAULT, &pVB, 0);
```

Как я замечал ранее, для визуализации частиц необходимо также создать буфер индексов. Т. к. каждая частица использует два многоугольника (каждый из которых содержит по три вершины), необходимо шесть индексов на каждую частицу. В случае, если имеется четыре частицы, необходимо создать буфер индексов, содержащий 24 элемента.

```
IDirect3DIndexBuffer9 *pIB = NULL;
pDevice->CreateIndexBuffer(24 * sizeof(short), \
         D3DUSAGE_WRITEONLY, D3DFMT_INDEX16, \
         D3DPOOL_DEFAULT, &pIB, 0);
```

```
unsigned short *IBPtr;
pIB->Lock(0, 0, (void**)&IBPtr, 0);
for(DWORD i=0;i<4;i++) { // # частиц
  IBPtr[i*6+0] = i * 4 + 0;
  IBPtr[i*6+1] = i * 4 + 1;
  IBPtr[i*6+2] = i * 4 + 2;
  IBPtr[i*6+3] = i * 4 + 3;
  IBPtr[i*6+4] = i * 4 + 2;
  IBPtr[i*6+5] = i * 4 + 1;
}
pIB->Unlock();
```

Все, что осталось сделать на данном этапе, это заблокировать буфер вершин, заполнить данными, разблокировать и визуализировать его! При заполнении буфера вершин убедитесь, что вы устанавливаете правильные значения вектора смещения угла для каждого из углов визуализируемой частицы и координаты центра в каждой вершине. Следующий код демонстрирует установку четырех частиц, имеющих случайное положение и размер:

```
// Заблокировать буфер вершин
sShaderVertex *VPtr = NULL;
pVB->Lock(0, 0, (void**)VPtr, 0);

for(DWORD i=0;i<4;i++) {
  // Получить случайное положение частицы
  float x = (float)(rand()%20)-10.0f;
  float y = (float)(rand()%20)-10.0f;
  float z = (float)(rand()%20)-10.0f;

  // Получить случайный размер частицы
  float Size = (float)(rand()%10)+1.0f;

  // Получить размер, деленный пополам
  float HalfSize = Size / 2.0f;

  // Верхняя левая вершина
  pVB[0].vecPos = D3DXVECTOR3(x, y, z);
  pVB[0].vecOffset = D3DXVECTOR2(-HalfSize, HalfSize);
  pVB[0].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
  pVB[0].u = 0.0f; pVB[0].v = 0.0f;

  // Верхняя правая вершина
  pVB[1].vecPos = D3DXVECTOR3(x, y, z);
  pVB[1].vecOffset = D3DXVECTOR2(HalfSize, HalfSize);
  pVB[1].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
  pVB[1].u = 1.0f; pVB[0].v = 0.0f;

  // Нижняя левая вершина
  pVB[2].vecPos = D3DXVECTOR3(x, y, z);
  pVB[2].vecOffset = D3DXVECTOR2(-HalfSize, -HalfSize);
  pVB[2].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
  pVB[2].u = 0.0f; pVB[0].v = 1.0f;
```

```
    // Нижняя правая вершина
    pVB[3].vecPos = D3DXVECTOR3(x, y, z) ;
    pVB[3].vecOffset = D3DXVECTOR2(HalfSize, -HalfSize);
    pVB[3].Diffuse = D3DCOLOR_RGBA(255,255,255,255);
    pVB[3].u = 1.0f; pVB[0].v = 1.0f;

    // Перейти к следующем четырем вершинам
    pVB+=4;
  }
```

После того как вы установили буфер вершин, можно визуализировать частицы. Ну, хотя это не совсем правда - все еще нужно создать и загрузить вершинный шейдер; создайте интерфейс объявления; установите источники вершин, текстуры, альфа-тестирование и смешивание; установите константы вершинного шейдера.

Загрузка вершинного шейдера и создание интерфейса объявления стандартны в DirectX9, поэтому я их пропущу. (Если вам необходима помощь, можете посмотреть код демонстрационной программы этой главы ParticlesVS или вспомогательные функции первой главы.) Вы также видели, как устанавливать источники вершин, текстуры, альфа-тестирование и смешивание, так что я пропущу их тоже. А теперь я хочу показать вам фактический вершинный шейдер, используемый для визуализации частиц.

Помните, что каждая частица состоит из четырех вершин. Ранее в этой главе, вы узнали, что необходимо было располагать каждую из этих вершин при помощи матрицы обратного преобразования вида, прежде чем визуализировать частицу. Однако в вершинном шейдере все будет немного по-другому. Прежде чем продолжить, я хочу показать вам компоненты направленного вектора преобразования вида.

Чтобы не быть обманщиком, скажу: компоненты направленного вектора описывают направление вида - вперед, вверх и вправо (как показано на рис. 12.6)

Чтобы лучше понять концепцию направленных компонент, встаньте и посмотрите вперед. Направление вашего взгляда называется вектор глаза, который описывает направление, куда вы смотрите. Мысленно проведите линию от пяток до верхней части головы. Направление этой линии называется вектор вверх, он указывает вверх от вашей текущей ориентации. Наконец, поднимите правую руку на 90 градусов и вытяните палец. Направление, которое вы указываете, называется вектор вправо, он всегда указывает направление вправо относительно текущей ориентации.

Эти векторы (глаза, вверх и вправо) определяют, какие направления относительны текущей ориентации. Например, если вы движетесь вперед, тогда вектор глаза является направлением движения; если вы идете назад, то обратный вектор глаза определяет направление движения. Аналогично, если вы движетесь вправо, то вы перемещаетесь в направлении вектора вправо. Идите влево, и тогда вы

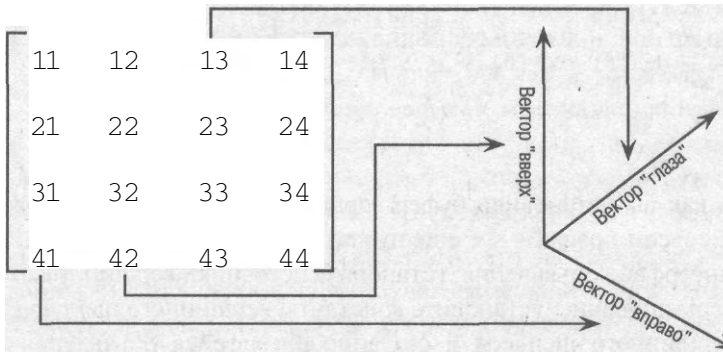


Рис. 12.6. Преобразование вида указывает его направление: какие направления справа, а какие сверху от его ориентации

будете перемещаться по направлению обратного вектора вправо. То же самое относится и к вектору вверх, если вы движетесь вверх или вниз по воображаемой линии от ноги до головы.

Теперь вы знаете, что весь этот материал о компонентах вектора был не просто так, не так ли? Хорошо, преобразования вида фактически содержатся в трех направленных векторах, как я только что заметил. Каждая компонента хранится в столбце преобразования вида. Как показано на рис. 12.7, вектор вправо является столбцом 1, вектор вверх столбцом 2, вектор глаза столбцом 3, четвертый столбец не задействован.

Преобразование вида

$$\begin{bmatrix} Right \cdot X & Up \cdot X & Eye \cdot X & 14 \\ Right \cdot Y & Up \cdot Y & Eye \cdot Y & 24 \\ Right \cdot Z & Up \cdot Z & Eye \cdot Z & 34 \\ Right \cdot W & Up \cdot W & Eye \cdot W & 44 \end{bmatrix}$$

Рис. 12.7. Каждый столбец преобразования вида содержит вектор направления, который вы можете использовать для расположения вершин частиц

Теперь я хочу вернуться к сути. Используя эти три вектора направления (или фактически нормальные векторы направлений), вы можете расположить угловые вершины частицы в соответствующих направлениях - вверх, вниз, влево или вправо, как

определено векторами направлений вверх и вправо. Эти векторы масштабируются на координаты смещения частицы.

Говоря по простому, сначала вы помещаете каждую вершину в начало координат мира. Далее вы передвигаете каждую вершину влево или вправо, используя нормальный вектор направления вправо, масштабированный на смещение частицы по x (sShaderParticle::vecOffset.x). После этого вы перемещаете вершину вверх или вниз, используя нормальный вектор направления вверх, масштабированный на смещение частицы по y (sShaderParticle::vecOffset.y). Наконец вы добавляете координаты центра частицы для получения результирующих координат вершин. Смыть, намылить и повторять для каждой вершины. Теперь вершины находятся в корректных положениях в мире, и можно их визуализировать!

Я покажу вам, как получить эти компоненты векторов направления из преобразования вида немного позднее; а пока я хочу вернуться к вершинному шейдеру. В начале шейдера, в комментариях, я привел список используемых привязок, констант и необходимую версию:

```
; v0 = Координаты частицы
; v1 = Смещения X/Y, необходимые для расположения вершин
; v2 = Рассеянный цвет частицы
; v3 = Текстурные координаты

; c0-c3 = Матрица вид*проекция
; c4 = Нормальный вектор направления вправо
; c5 = Нормальный вектор направления вверх
vs.1.1
```

Как вы можете видеть, используемые константы являются комбинированной матрицей вида и проекции, которую вы помещаете в константы от c0 до c3. Далее следуют нормальные векторы направлений вправо и вверх, о которых я говорил ранее. Они помещаются в константы c4 и c5. Я вернусь к установке констант немного позднее; а пока, я хочу продолжить рассмотрение кода шейдера.

Далее следует фактическая привязка объявлений вершинных регистров, необходимая для того, чтобы можно было получить доступ к соответствующим компонентам структуры вершин через вершинный шейдер.

```
dcl_position v0
dcl_position1 v1
dcl_color v2
dcl_texcoord v3
```

А теперь начинается веселье! Помните, ранее в этом разделе я сказал, что сначала необходимо расположить вершину в начале координат мира и передвинуть ее вдоль масштабированных векторов вверх и вправо? Хорошо, этой цели служит приведенный ниже код вершинного шейдера.

```
; Масштабировать смещения углов на векторы вправо и вверх
mov r2, v1
mad r1, r2.xxx, c4, v0
mad r1, r2.yyy, c5, r1
```

Предыдущий кусочек кода берет нормальные векторы масштабирования (расположенные в регистрах c4 и c5) и умножает их на значения смещения вершинной структуры. После чего результаты добавляются к координатам центра частицы для получения результирующих координат вершины. Все, что вам необходимо сделать, это применить преобразование вид*проекция к координатам вершины и сохранить рассеянный цвет и текстурные координаты.

```
; Наложить преобразование вид*проекция
m4x4 oPos, r1, c0

; Сохранить рассеянный цвет
mov oD0, v2

; Сохранить текстурные координаты
mov oT0.xy, v3
```

Ничего себе, вот это маленький и эффективный шейдер! Все, что остается сделать после создания и загрузки нового вершинного шейдера, это установить его константы и визуализировать! Эти константы содержат преобразование вид*проекция и векторы направлений вправо и вверх. Давайте разберемся с преобразованием и перейдем к векторам направлений.

Я полагаю, что матрицы преобразования вида и проекции хранятся в двух отдельных объектах D3DXMATRIX. Все, что вам необходимо сделать, это просто перемножить их, транспонировать результирующую матрицу и сохранить результат в константах, используя функцию SetVertexShaderConstantF.

```
// matView = матрица преобразования вида
// matProj = матрица преобразования проекции
D3DXMATRIX matViewProj = matView * matProj;
D3DXMatrixTranspose(&matViewProj, &matViewProj);
pDevice->SetVertexShaderConstantF(0, (float*)matViewProj, 4);
```

Теперь перейдем к векторам направлений. Помните, я сказал, что векторы направлений хранятся в столбцах преобразования вида? Т. к. у нас уже есть объект матрицы преобразования вида, то можно непосредственно получить эти компоненты

и нормализовать их одновременно с помощью следующего кода. (Заметьте, что используются только компоненты вправо и вверх, не учитывая вектор глаза.)

```
// Получить нормальные векторы вправо/вверх из преобразования вида
D3DXVECTOR4 vecRight, vecUp;

// Вектор вправо является первым столбцом
D3DXVec4Normalize(&vecRight, \
       &D3DXVECTOR4(matView._11, \
            matview._21, \
            matView._31,  0.0f));

// Вектор вверх является вторым столбцом
D3DXVec4Normalize(&vecUp, \
       &D3DXVECTOR4(matView._12, \
            matview._22, \
            matView._32,  0.0f));
```

После того как вы нормализовали векторы, вы можете сохранить их в константах c4 и c5.

```
pDevice->SetVertexShaderConstantF(4, (float*)&vecRight, 1);
pDevice->SetVertexShaderConstantF(5, (float*)&vecUp, 1);
```

Наконец-то! Все готово, и теперь вы можете визуализировать частицы с помощью замечательного вершинного шейдера! Убедитесь, что вы установили потоки, текстуру, шейдер, объявление и что вы используете индексные методы рисования примитивов, т. к. используются буферы индексов. Посмотрите на все это в следующем коде:

```
// pShader = интерфейс вершинного шейдера
// pDec = интерфейс объявления элементов вершин

// Установить вершинный шейдер, объявление и источники потоков
pDevice->SetFVF(NULL);
pDevice->SetVertexShader(pShader);
pDevice->SetVertexDeclaration(pDecl);
pDevice->SetStreamSource(0, pVB, 0, sizeof(sShaderVertex));
pDevice->SetIndices(pIB);

// Включить альфа-тестирование
pDevice->SetRenderState(D3DRS_ALPHATESTENABLE,   TRUE);
pDevice->SetRenderState(D3DRS_ALPHAREF,   0x08);
pDevice->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);

// Включить альфа-смешивание (простого добавочного типа)
pDevice->SetRenderState(D3DRS_ALPHABLENDENABLE,   TRUE);
pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);
pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_DESTCOLOR);

// Визуализировать частицы вершинного шейдера (четыре)
pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST,0,0,16,0,8);
```

Как вы видите, визуализация частиц с помощью вершинных шейдеров является лучшим способом. Когда вы разберетесь, как вершинный шейдер работает с данными частицы, посмотрите демонстрационную программу этой главы, иллюстрирующую использование большого числа частиц в ваших проектах.

После того как вы усвоите основы рисования частиц, можно двигаться дальше и посмотреть, как можно оживить их в ваших проектах.

Оживление частиц

После того как вы узнали, как рисовать частицы, вы можете использовать эти знания для создания, управления и уничтожения множества экземпляров частиц в трехмерном мире. Первое, что необходимо сделать, это создать класс частиц, который бы содержал всю необходимую информацию о частице.

Не беря во внимание данные вершин, которые хранятся отдельно от данных частиц, скорее всего вы будете хранить положение, цвет и тип частицы. Вы также можете хранить направление, скорость ее движения и время жизни частицы, после которого движок удалит ее.

Для этой книги я решил использовать следующую информацию о частице:

- **Type.** Частицы могут быть различных типов, и эта переменная определяет тип частицы. Например, я могу хранить частицы огня и дыма вместе, используя одинаковую структуру частиц.
- **Position.** Это набор трехмерных координат, определяющих положение частицы в мире.
- **Velocity.** Скорость и направление частицы хранятся в векторе. Компоненты x, y и z определяют насколько быстро движется частица в направлении образуемой ими оси.
- **Life.** Частица может существовать в трехмерном мире долго; эта часть данных позволяет вашему движку узнать, когда необходимо удалять частицу из списка активных визуализируемых частиц. Однако не обязательно удалять частицу, вы можете сделать, чтобы она существовала неограниченно долго.
- **Size.** За свой жизненный цикл частица может менять свой размер. Эти данные содержат значение, используемое в качестве размера частицы (в трехмерных единицах). Помните, частица имеет одинаковый размер по осям x и y, так что задание размера равного 10 сделает частицу, имеющую 20 единиц в ширину и 20 единиц в высоту (от -10 до 10 по каждой оси).
- **Color.** Для моделирования изменений частицы вы можете использовать модификатор цвета, чтобы изменять красную, зеленую и синюю компоненты цвета многоугольников при визуализации. Например, частица огня может менять цвет от белого до красного и желтого с течением времени.

Вы можете объединить все эти данные в один простой класс, как показано тут:

```
class cParticle
{
  public:
    DWORD m_Type;  // Тип частицы

    D3DXVECTOR3 m_vecPos;  // Положение частицы
    D3DXVECTOR3 m_vecVelocity; // Скорость частицы

    DWORD m_Life; // Время жизни частицы в мс
    float m_Size;  // Размер частицы

    DWORD m_Color; // Рассеянный цвет частицы
};
```

Каждой частице необходим собственный экземпляр класса cParticle. Вы можете объединить набор этих классов в массив для облегчения работы. Или, как я вам покажу позже, вы можете использовать связанный список частиц. Я расскажу вам, как иметь дело с более чем одной частицей далее в этой главе. А пока, для упрощения, будем рассматривать одну частицу.

Первым шагом является заполнение класса cParticle информацией о частице: ее тип, местоположение, цвет и т. д. Следующий кусочек кода устанавливает тип частицы в 1 и помещает ее в начало координат мира. Размер является номинальным (установлен в 5), а цвет устанавливается в ярко белый (исходный цвет карты текстуры). Скорость пока не имеет никакого значения, поэтому обнулим все ее компоненты.

```
// Создадим экземпляр частицы и заполним его данными
cParticle Particle;
Particle.m_Type = 1;
Particle.m_vecPos = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
Particle.m_Size = 5.0f;
Particle.m_Color = D3DCOLOR_RGBA(255,255,255,255) ;
Particle.m_vecVelocity = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
```

Вот и все, что необходимо для создания частицы! Конечно, частица просто стоит на месте и ничего не делает, так что давайте заставим ее летать, придав ей скорость.

Передвижение частиц при помощи скорости

После того как вы создали частицу в трехмерном мире, необходимо придать ей движение, применив метод обновления ее положения в мире. Каждая частица имеет особенный способ обновления своих координат. Для частицы огня необходимо, чтобы она медленно поднималась и меняла свой цвет с самого горячего на

холодный (например с красного на оранжевый). Частицы дыма обычно дрейфуют по ветру, так что необходимо найти близлежащий источник ветра и использовать его для управления этими частицами.

Большинство частиц являются очень простыми по природе, и они двигаются в заданном направлении, изменяемом внешними силами. Частица имеет скорость, определяющую насколько быстро или медленно она движется. Используемые внешние силы могут увеличивать или уменьшать скорость частицы со временем, таким образом, ускоряя или замедляя ее движение.

Для каждого обрабатываемого кадра вы можете посчитать силы, действующие на скорость частицы. Эти силы могут иметь любое происхождение, будь то ветер, гравитация, сопротивление или толчок. Однако не все силы необходимы в вашем движке, они просто увеличивают реалистичность. Ваш движок должен содержать хотя бы силу тяжести и толчка, чтобы частицы могли двигаться и останавливаться на земле.

Вы можете добавить функции, которые бы применяли силы к классу частиц. Или, если вы используете другой класс для управления частицами (похожий на тот, что вы увидите ниже), вы можете поместить функции туда. А пока я просто покажу вам, как вычислять вектор силы из множества источников, прикладываемых к скорости частицы (заметьте, что силы измеряются в миллисекунду).

```
// силы, прикладываемые к частице(ам)
D3DXVECTOR3 vecForce = D3DXVECTOR3(0.0f, 0.0f, 0.0f);

// Добавить силу тяжести равную 0.02, каждую миллисекунду
vecForce += D3DXVECTOR3(0.0f, -0.02f, 0.0f);

// Добавить силу ветра равную 0.01f, каждую миллисекунду
vecForce += D3DXVECTOR3(0.01f, 0.0f, 0.0f);
```

После того как вы вычислили силу, прикладываемую к каждой частице, необходимо обновить скорость. Обновление скорости частицы является простым сложением векторов силы и скорости. Помните, что силы измеряются в миллисекундах, означая, что вы должны умножить вектор силы на количество прошедших миллисекунд с последнего движения частицы. Предположим, что величина прошедшего времени хранится в вещественной переменной TimeElapsed.

```
// TimeElapsed=время,в миллисекундах,прошедшее с последнего движения
// частицы.
Particle.m_vecVelocity += (m_vecForce * TimeElapsed);
```

В добавление к использованию вектора силы для изменения скорости частицы можно использовать альтернативные методы для обновления частицы, используя более интеллектуальные методы.

Использование интеллекта при обработке

Пока что я рассмотрел простые движения частиц, использующие силы. А что насчет бегущих прохожих, о которых я говорил в начале этой главы? Эти испуганные люди, спасающие свои жизни от гигантского монстра, разрушившего городскую дамбу. Никто ведь не скажет, что нельзя использовать немного интеллекта при обработке частиц, не правда ли?

Вместо приложения набора сил к частицам вы могли бы интеллигентно двигать их, используя минимум времени и обработки. Предположим, ваш игровой город поделен сеткой. Каждая линия сетки представляет собой улицу, по которой могут ездить машины. Каждая машина имеет заданное направление движения; как только она попадает на пересечение линий сетки, она может выбрать новое случайное направление движения.

Разнообразные пересечения помечаются как опасные; если монстр атакует город, начиная с этого момента, машины будут пытаться уехать как можно дальше от этих опасных точек с удвоенной скоростью. В конце концов, все машины либо покинут город, либо будут раздавлены чудовищем, либо застрянут где-нибудь и будут оставлены. Таким образом, со временем все частицы могут быть удалены, освобождая время обработки для других вещей.

Частицы, использующие интеллект, точно знают с какой скоростью и в каком направлении перемещаться. Вместо того чтобы медленно изменять скорость в заданном направлении с течением времени, они меняют скорость мгновенно.

Этот тип интеллектуальной обработки замечательно подходит для ваших игровых проектов, и его очень легко реализовать, используя технологии, описанные в этой главе. В демонстрационной программе, содержащейся на компакт-диске, вы увидите, как я использовал интеллектуальную обработку в своем движке частиц.

В демонстрационной программе присутствует группа частиц, которая представляет людей. Эти люди просто стоят и ничего не делают, пока над их головами не начинает пикировать вертолет, заставляя людей искать укрытие!

Чтобы узнать больше о методах использования интеллектуальной обработки частиц, посмотрите на игру War of the Monsters фирмы Ingoc для PlayStation 2. Помните сценарий о терроризируемых жителях, о котором я упоминал ранее? В War of the Monsters огромные звери опустошают город, и сотни маленьких людей, состоящих из частиц, бегут, спасая свою жизнь. Определенно замечательное использование частиц!

Создание и уничтожение частиц

Частицы обычно используются для улучшения внешнего вида игры. Игровой движок может использовать сотни полигонов для одного эффекта. По этой причине очень важным является использование как можно меньшего количества частиц и периодическое удаление старых частиц, которые уже больше не используются.

Например, попадание в стену из лазерной винтовки может породить группу частиц, представляющих летящие осколки. Эти осколки быстро отлетают и рассеиваются. Это типично для большинства используемых частиц - они существуют только в течение короткого интервала времени, а потом исчезают. Конечно есть исключения, когда частицы существуют всю игру. Например, возьмите биллбордные деревья или знаки, которые наполняют ландшафт. Эти частицы никогда не двигаются и не исчезают (если только их не растопчет монстр). Лучшим способом управления этими долго существующими объектами является рисование только тех, которые находятся в пределах определенного расстояния от смотрящего, таким образом уменьшая количество частиц, рисуемых за каждый кадр. Для демонстрационных программ, содержащихся на компакт-диске, я использовал частицы деревьев и людей, которые не уничтожаются. Также я использовал частицы клубов дыма - эти частицы уничтожаются после непродолжительного времени.

Реализовать список активных частиц легко с помощью связанного списка. Для каждой активной частицы существует соответствующая структура в связанном списке частиц. Как только вы хотите обновить частицы вашего движка, вы просматриваете весь связанный список и обновляете частицы, содержащиеся в нем. Если частица больше не нужна, ее структура освобождается и удаляется из связанного списка.

Возвращаясь к элементарному классу cParticle, созданному ранее в этой главе, вы можете добавить несколько указателей для создания связанного списка частиц. Также вы можете добавить в класс конструктор и деструктор, которые бы управляли этими указателями при создании и удалении экземпляра класса.

```
class cParticle
{
  public:
   // Предыдущие данные частицы, такие как тип, положение и т.д.
   cParticle *m_Prev; // Предыдущая частица в связанном списке
   cParticle *m_Next; // Следующая частица в связанном списке
  public:

   // Конструктор и деструктор для очистки и освобождения данных
   cParticle () { m_Prev = NULL; m_Next = NULL; }
   ~cParticle () { delete m_Next; m_Next=NULL; m_Prev=NULL; }
};
```

Т. к. используется связанный список, одна частица становиться корневой и к ней привязываются все остальные. Т. к. эта частица может уничтожаться, корневая частица может меняться со временем. Вы можете установить все добавляемые в связанный список частицы в качестве корневых, а потом привязать старую корневую частицу в качестве следующей (используя указатели m_Prev и m_Next).

```
// pRoot = текущая корневая частица списка

// Создать новый экземпляр используемого класса частиц
cParticle *pParticle = new cParticle();

// Связать новую частицу в качестве корня и привязать к старому
корню
pParticle->m_Prev = NULL; // Очистить указатель на предыдущий объект
pParticle->m_Next = pRoot; // Привязать новую частицу к старой
корневой
pRoot->m_Prev = pParticle; // Привязать корневую частицу к новой
pRoot = pParticle; // Переприсвоить корневую частицу новой
```

Для удаления заданной частицы, вы можете использовать указатели связанного списка, чтобы соединить связанные частицы и освободить ресурсы, удаляемой частицы.

```
// pParticle = указатель удаляемой частицы
// pRoot = корневая частица

// Привязать предыдущую частицу к следующей
if(pParticle->m_Prev) {
   pParticle->m_Prev->m_Next = pParticle->m_Next;
}  else {
   // Если предыдущей частицы нет, значит текущая частица является
корневой.
   // Теперь необходимо сделать корневой следующую частицу
   pRoot = pParticle->m_Next;
 }

// Привязать следующую частицу к предыдущей
if(pParticle->m_Next)
pParticle->m_Next->m_Prev = pParticle->m_Prev;

// Очистить указатели частицы и освободить ресурсы
pParticle->m_Prev = pParticle->m_Next = NULL;
delete pParticle;
```

Использование связанного списка является обычным программистским приемом, так что нет никакой нужды углубляться в детали. Я уверен, что существуют намного более эффективные способы управления списком частиц, но чтобы быть предельно честным (и я уверен, что получу немало электронных писем на этот счет!), использование связанного списка очень быстро и просто.

Я оставляю вам самостоятельно посмотреть исходный код демонстрационной программы Particles, чтобы увидеть, как использовать связанный список для хранения большого числа частиц. А пока, читайте, как использовать связанный список частиц для их визуализации.

Рисование частиц

Что еще можно сказать, чего вы еще не знаете? Ну, способ рисования частиц является важным, если их тысячи. Т. к. мы не хотим иметь множество больших буферов вершин, использующих память, необходимо управлять ими, прежде чем двигаться дальше.

Управление буферами вершин означает, что необходимо блокировать и заполнять их кадр за кадром. Я знаю, я говорил, что вы не должны этого делать, но если сделать это правильно, можно эффективно управлять большим количеством частиц при использовании буферов вершин без возникновения задержек при их блокировании, загрузке и разблокировании.

Чтобы эффективно управлять буферами, необходимо следить за количеством рисуемых частиц и количеством вставляемых вершин при визуализации. Для каждой рисуемой частицы вы вставляете соответствующие вершины в буфер вершин. Если после вставки буфер полностью заполнен, вы можете его визуализировать и начинать по новой со следующей частицы.

После обработки всех частиц вы проверяете были ли визуализированы все частицы. Другими словами, вы проверяете, есть ли еще данные вершин в буфере. Если да, вы визуализируете последний набор частиц и двигаетесь дальше.

Для каждого визуализируемого набора частиц необходимо разблокировать буфер вершин и визуализировать многоугольники. После визуализации необходимо по новой заблокировать буфер и продолжить. Предположим, вы используете метод визуализации частиц при помощи вершинного шейдера и частицы содержатся в связанном списке, на который указывает объект cParticle (pRoot). Следующий код иллюстрирует управление буферами вершин, визуализируя частицы группами.

```
// pRoot = корневой объект cParticle, содержащий все визуализируемые
// частицы
// matView, matProj = преобразования вида и проекции
// pTexture = объект текстуры, используемый при визуализации частиц
// pShader, pDecl = объекты вершинного шейдера и объявления
// pVB, pIB = объекты буферов вершин и индексов
// NumParticles = количество частиц, которые могут находится
в буфере вершин
```

```
// Установить вершинный шейдер, объявления и источники потоков
pDevice->SetFVF(NULL);
pDevice->SetVertexShader(pShader);
pDevice->SetVertexDeclaration(pDecl);
pDevice->SetStreamSource(0, pVB, 0, sizeof(sShaderVertex));
pDevice->SetIndices(pIB);

// Включить альфа-тестирование
pDevice->SetRenderState(D3DRS_ALPHATESTENABLE,   TRUE);
pDevice->SetRenderState(D3DRS_ALPHAREF,   0x08);
pDevice->SetRenderState(D3DRS_ALPHAFUNC, D3DCMP_GREATEREQUAL);

// Включить альфа-комбинирование (простого добавочного типа)
pDevice->SetRenderState(D3DRS_ALPHABLENDENABLE,   TRUE);
pDevice->SetRenderState(D3DRS_SRCBLEND, D3DBLEND_SRCCOLOR);
pDevice->SetRenderState(D3DRS_DESTBLEND, D3DBLEND_DESTCOLOR) ;

// Установить текстуру
pDevice->SetTexture(0, pTexture);

// Сохранить транспонированную матрицу вид*проекция в константах
D3DXMATRIX matViewProj =  (*matView)  *  (*matProj);
D3DXMatrixTranspose(&matViewProj, &matViewProj);
pDevice->SetVertexSnaderConstantF(0, (float*)&matViewProj, 4);

// Получить нормальные векторы вверх и вправо из преобразования вида
D3DXVECTOR4 vecRight, vecUp;

// Вектор вправо является первым столбцом
D3DXVec4Normalize(svecRight, \
       &D3DXVECTOR4(matView._11, \
            matView._21, \
            matView._31,  0.0f));

// Вектор вверх является вторым столбцом
D3DXVec4Normalize(&vecUp, \
       &D3DXVECTOR4(matView._12, \
            matView._22, \
            matView._32,  0.0f));
// Сохранить векторы в константах
pDevice->SetVertexShaderConstantF(4, (float*)&vecRight, 1);
pDevice->SetVertexShaderConstantF(5, (float*)&vecUp, 1);
```

До этого момента идет обычный код установки буфера вершин, потоков, альфа-состояний, текстуры и регистров констант (содержащих значения преобразования и векторов направлений). Важным является следующий кусочек кода. Он блокирует буфер вершин, чтобы подготовить его к добавлению вершин частиц. Отсюда с помощью цикла обрабатываются все частицы.

Замечание. *Т. к. здесь буфер вершин постоянно блокируется, изменяется и разблокируется, возможно вы захотите использовать флаг D3DUSAGE_DYNAMIC при вызове IDirect3DDevice9::CreateVertexBuffer. Это позволит сообщить Direct3D, что с буфером будет проводиться много работы, чтобы разместить его в легко доступной и эффективно используемой памяти.*

```
// Начать с первой частицы в списке
cParticle *Particle = pRoot;

// Установить количество для сброса буфера вершин
DWORD Num = 0;

// Заблокировать буфер вершин
sShaderVertex *Ptr;
pVB->Lock(0, 0, (void**)&Ptr, D3DLOCK_DISCARD);

// Просмотреть все частицы
while(Particle != NULL) {
```

После начала цикла вы можете скопировать данные текущей вершины в буфер вершин

```
// Скопировать данные частиц в буфер вершин
float HalfSize = Particle->m_Size / 2.0f;

Ptr[0].vecPos = Particle->m_vecPos;
Ptr[0].vecOffset = D3DXVECTOR2(-HalfSize, HalfSize);
Ptr[0].Diffuse = Particle->m_Color;
Ptr[0].u = 0.0f;
Ptr[0].v = 0.0f;
Ptr[1].vecPos = Particle->m_vecPos;
Ptr[1].vecOffset = D3DXVECTOR2(HalfSize, HalfSize);
Ptr[1].Diffuse = Particle->m_Color;
Ptr[1].u = 1.0f;
Ptr[1].v = 0.0f;
Ptr[2].vecPos = Particle->m_vecPos;
Ptr[2].vecOffset = D3DXVECTOR2(-HalfSize, -HalfSize);
Ptr[2].Diffuse = Particle->m_Color;
Ptr[2].u = 0.0f;
Ptr[2].v = 1.0f;
Ptr[3].vecPos = Particle->m__vecPos;
Ptr[3].vecOffset = D3DXVECTOR2(HalfSize, -HalfSize);
Ptr[3].Diffuse = Particle->m_Color;
Ptr[3].u = 1.0f;
Ptr[3].v = 1.0f;

Ptr+=4; // перейти к следующим четырем вершинам
```

После копирования всех вершин в буфер вершин вы можете увеличить количество частиц, содержащихся в нем. Если их количество достигает максимального значения, которые может содержать буфер, тогда он разблокируется, визуализируется и блокируется заново, подготавливаясь таким образом принять новый набор частиц.

```
// Увеличить количество вершин и очистить буфер, если он заполнен
Num++;
if(Num >= NumParticles) {

  // разблокировать буфер и визуализировать полигоны
  pVB->Unlock();
  ppDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, \
            0, 0, Num*4, 0, Num*2);

  // Еще раз блокировать буфер вершин
  pVB->Lock(0, 0, (void**)&Ptr, D3DLOCK_DISCARD);

  // Обнулить количество вершин
  Num=0;
 }

 // Перейти к следующей частице
 Particle = Particle->m_Next;
}
```

После того как вы просмотрели все частицы, необходимо последний раз разблокировать буфер вершин. Если в нем все еще находятся вершины, необходимо визуализировать их.

```
// Разблокировать буфер вершин
pVB->Unlock();

// Визуализировать полигоны, если они остались
if(Num)
  pDevice->DrawIndexedPrimitive(D3DPT_TRIANGLELIST, \
            0, 0, Num*4, 0, Num*2);
}
```

Вот и весь эффективный способ визуализации неограниченного количества частиц при помощи маленького буфера вершин! А теперь давайте объединим все рассмотренное ранее и создадим пару классов, которые помогали бы полностью управлять частицами.

Управление частицами с помощью класса

Ранее в этой главе, в разделе "Оживление частиц", вы видели, как создавать класс, содержащий данные частицы. Этот класс, cParticle, является великолепной отправной точкой для создания пары вспомогательных классов. На самом деле необходимо создать по крайне мере два класса. Класс cParticle содержит информацию об одной частице, такую как положение, размер и тип. В этом классе нет ничего такого, за исключением того, что мы хотим хранить в нем данные и указатели на связанный список частиц.

Второй класс отвечает за весь список частиц и управляет их созданием и уничтожением во времени. Это позволяет визуализировать все частицы, содержащиеся в связанном списке этого класса. Этот тип класса называется излучателем частиц, потому что он ответственен за излучение частиц. Я определил один из классов излучения так:

```
class cParticleErnitter
{
  protected:
    IDirect3DDevice9 *m_pDevice; // Родительское 3-D устройство

    // Тип излучателя
    DWORD m_EmitterType;

    // Буферы вершин и индексов, содержащих вершины
    IDirect3DVertexBuffer9 *m_VB;
    IDirect3DIndexBuffer9 *m_IB;

    // Максимальное количество частиц в буфере
    DWORD m_NumParticles;

    // Расположение излучателя (в трехмерном пространстве)
    D3DXVECTOR3 m_vecPosition;

    // Корневой объект связанного списка частиц
    cParticle *m_Particles;

    // Количество ссылок класса
    static DWORD m_RefCount;
    static IDirect3DVertexShader9 *m_pShader; // Вершинный шейдер
    static IDirect3DVertexDeclaration9 *m_pDecl; // Объявление вершин
    static IDirect3DTexture9 **m_pTextures; // Текстуры

  public:
    cParticleEmitter();
    ~cParticleEmitter();

    BOOL Create(IDirect3DDevice9 *pDevice,
         D3DXVECTOR3 *vecPosition,
         DWORD EmitterType,
         DWORD NumParticlesPerBuffer = 32);
    void Free();

    void Add(DWORD Type, D3DXVECTOR3 *vecPos, float Size,
         DWORD Color, DWORD Life,
         D3DXVECTOR3 *vecVelocity);
    void ClearAll();
    void Process(DWORD Elapsed);

    // Функции, подготавливающие частицу к визуализации
    BOOL Begin(D3DXMATRIX *matView, D3DXMATRIX *matProj);
    void End();
    void Render();
};
```

Ничего себе - множество вопросов в которых надо разбираться! Давайте рассмотрим этот класс по частям, чтобы лучше понять, что он делает. Сначала имеется набор защищенных переменных. В каждом классе излучателя частиц содержится указатель на 3D устройство, буферы вершин и индексов, используемые для визуализации частиц.

Т. к. каждый излучатель предназначен для разных целей (один может излучать частицы огня, другой осколки), существует переменная типа частиц (m_EmitterType). Значение этой переменной зависит от типа используемых частиц. Для демонстрационной программы этой книги я использовал следующие типы излучателей частиц:

```
// Тип излучаемых частиц
#define EMITTER_CLOUD 0
#define EMITTER_TREE 1
#define EMITTER_PEOPLE 2
```

В зависимости от типа излучаемых частиц вы передаете определяемое макросом значение, функции cParticleEmitter::Create и, кроме этого, используемый объект 3D устройства и количество частиц, которые могут находится в буфере вершин. Также необходимо указать в качестве параметра вектор, определяющий положение излучателя частиц в трехмерном мире.

Далее в списке защищенных переменных следует корневой объект частицы (m_Particles), который используется для хранения связанного списка частиц, создаваемых излучателем. Как вы можете видеть из объявления класса cParticleEmitter, существует только одна функция (Add), которая позволяет вам добавлять частицы в сцену. Чтобы добавить частицу, просто вызовете Add, указав тип частицы.

Я создавал типы частиц на основе используемых текстур. Например, я имею три текстуры для трех типов излучателей. Одна содержит изображение частицы огня, другая содержит изображения частицы дыма, и третья содержит изображение частицы вспышки. Тип частиц определяется так:

```
#define PARTICLE_FIRE 0
#define PARTICLE_SMOKE 1
#define PARTICLE_FLASH 2
```

Возвращаясь к функции Add, вы задаете положение (в мировом пространстве) в сцене добавляемой частицы. Каждая частица имеет собственную продолжительность жизни, цвет, размер и стартовую скорость, которые вы устанавливаете при вызове функции Add. Продолжительность жизни измеряется в миллисекундах, цвет измеряется значением D3DCOLOR, размер является вещественным значением,. а скорость - объектом D3DXVECTOR.

Обычно функция Add не используется напрямую для добавления частиц - это является задачей функции Update излучателя (хотя это не должно останавливать вас от использования Add, когда это необходимо). На самом деле функция Update служит двум целям: обновлять все частицы, содержащиеся в связанном списке, и определять, необходимо ли добавить еще частиц в связанный список.

Несколько указателей на объекты завершают список защищенных переменных. Этими указателями являются вершинный шейдер, объявление элементов вершин и массив текстур, используемый для визуализации частиц. Заметьте, что каждый из этих объектов является статическим, т. е. все экземпляры излучателей частиц разделяют их, что помогает сохранить память.

Количество ссылок, содержащихся в переменной m_RefCount, предназначено для слежения за тремя статическими объектами. Когда создается излучатель (с помощью вызова Create), количество ссылок увеличивается; когда излучатель уничтожается (вызовом Free или деструктором класса), количество ссылок уменьшается. Вершинный шейдер и текстуры загружаются при первом вызове инициализации излучателя (внутри функции Create); когда освобождается последний излучатель (количество ссылок равно 0), все объекты освобождаются.

Пока что я описал все, за исключением четырех функций - ClearAll, Begin, End и Render. Функция ClearAll очищает список частиц излучателя, предоставляя вам новый список. Вы можете вызвать эту функцию где угодно, чтобы заставить излучатель удалить все частицы.

Что касается функций Begin, End и Render, они используются совместно для визуализации частиц. Когда вы используете частицы, основанные на вершинном шейдере, должны быть некоторые состояния визуализации и другие настройки, которые одинаковы для всех излучателей. Вы можете сэкономить время, установив сначала эти данные, потом визуализировав частицы, после чего закончить, переустановив соответствующие состояния и данные визуализации. Этой цели и служат эти три функции.

В функции Begin, используемой для частиц, основанных на вершинных шейдерах, установите FVF в NULL, после чего установите используемые вершинный шейдер и объявление, сохраните транспонированное преобразование вид*проекция, сохраните векторы направлений вверх и вправо. После того как вы вызвали функцию Begin (которая в качестве параметров использует матрицы преобразования вида и проекции), вы можете вызывать функцию Render для визуализации частиц. Этот процесс одинаков для всех частиц, созданных из одного класса, т. к. используются одни и те же текстуры и вершинный шейдер. Как только вы закончите визуализировать частицы, просто вызовете End для очистки вершинного шейдера и объявления.

Класс излучателя частиц по своей конструкции является очень простым: он просто хранит список частиц и визуализирует их. Для каждого кадра игры вызы-

вается функция Update, что позволяет классу решать, какие частицы добавить, какие обновить, а какие удалить. Это все простые манипуляции со связанным списком, так что я пропущу код, который вы можете найти на компакт-диске.

На самом деле, вы уже видели все, связанное с излучателями частиц, в этой главе: от связанного списка частиц до обработки движения и столкновений и их визуализации. На данный момент осталось не так уж и много повторить. Если вы посмотрите код демонстрационных программ этой главы, вы увидите, как я создал класс излучателя частиц, который управляет всеми типами частиц, излучателей и визуализирует частицы при помощи трех методов, рассмотренных в самом начале этой главы.

Использование излучателей в проектах

Хорошо-хорошо, достаточно кода класса; давайте посмотрим, как со всем этим работать. Для создания излучателя необходимо создать экземпляр класса излучателя так:

```
cParticleEmitter Emitter;
```

Теперь вызовите функцию Create излучателя, чтобы поместить излучатель в центр трехмерного мира. Также задайте используемый тип излучателя.

```
Emitter.Create(pDevice, \
        &D3DXVECTOR3(0.0f,  0.0f,  0.0f),  \
         EMITTER_CLOUD);
```

После того как вы создали излучатель, можно начинать добавлять в него частицы с помощью функции Add, или можете вызвать функцию Update, которая добавит частицы за вас.

```
Emitter.Add(PARTICLE_SMOKE, \
        &D3DXVECTOR3(0.0f,  0.0f,  0.0f),  \
         10.0f, 0xFFFFFFFF, 2000, \
        &D3DXVECTOR3(0.0f,  1.0f,  0.0f));
```

После того как вы добавили частицы в сцену, необходимо обновить их с помощью функции Update и визуализировать частицы.

```
Emitter.Update(ElapsedTime);
Emitter.Begin(&matView,  &matProj);
Emitter.Render();
Emitter.End();
```

Это является, в каком то роде, полупопыткой показать, как использовать классы частиц, но я рекомендую посмотреть демонстрационные программы, располо-

женные на компакт-диске книги. Эти демонстрационные программы намного лучше показывают, что вы можете делать с помощью частиц в своих проектах.

Создание движков частиц в вершинных шейдерах

Хорошо, вы знали, что кто-то должен был создать их! На самом деле существует способ создать движок частиц, который бы полностью выполнялся внутри вершинного шейдера. Вы можете найти этот вершинный шейдер на сайте Nvidia http://developer.nvidia.com/view.asp?IO=Particle_System.

В основе работы этого вершинного шейдера лежит то, что вы определяете структуру вершин, содержащую скорость частицы; используя эту скорость, вы определяете, на сколько переместить частицу, основываясь на прошедшем времени. Для визуализации частицы используются точечные спрайты, т. е. вы можете представить каждую частицу одной вершиной (находящейся в буфере вершин).

Я не хочу изобретать колесо, но при помощи информации, полученной в этой главе, вы должны понять, что происходит в вершинном шейдере частиц. Для каждой частицы, которую получает для визуализации шейдер (точечный спрайт), вычисляются положение, цвет и размер. Эти вычисления основываются на переменной прошедшего времени, устанавливаемой через константы шейдера. Вы уже видели, как перемещать частицы, основываясь на их скорости - это простое умножение вектора скорости на величину прошедшего времени. То же самое происходит для цвета и размера - они вычисляются умножением соответствующих величин на количество прошедшего времени.

Посмотрите демонстрационные программы

На компакт-диске этой книги вы обнаружите три проекта, которые я создал для иллюстрации управления частицами. Эти проекты содержат код для добавления, удаления, визуализации частиц разнообразных типов, который вы можете использовать в собственных проектах для получения привлекательных эффектов частиц.

Результат выполнения всех трех программ одинаков (см. рис 12.8). Вертолет пролетает над лесом и группой людей. В этой демонстрационной программе используются частицы: пучки дыма, выдуваемые с земли перед вертолетом, деревья и люди. На самом деле используются три различных типа деревьев и два типа людей. Излучатель частиц имеет несколько дополнительных функций, которые добавляют частицы дыма и изменяют внешний вид людей во время выполнения демонстрационной программы. Как только вертолет пролетает над человеком, частица, представляющая человека, меняет свой тип. Когда вертолет отлетает от

человека, частица опять меняется на другую. Посмотрите демонстрационную программу, чтобы понять, что я имею ввиду.

Как вы можете видеть, использование частиц зависит от проекта. То, что подходит для одного проекта, может не сработать для другого. По этой причине в этой главе было дано не так много, за исключением основной теории и основных знаний. Небольшое пожелание: убедитесь, что посмотрели демонстрационные программы, содержащиеся на компакт-диске.

Рис. 12.8. Apache пикирует над головами любителей деревьев

Программы на компакт диске

В директории главы 12 компакт-диска книги, вы обнаружите три проекта, которые иллюстрируют использование трех методов рисования частиц, показанных в этой главе. Этими проектами являются:

- **Particles.** Этот проект иллюстрирует рисование частиц, использующих квадраты, основанное на первой рассмотренной в этой главе технологии. Он расположен в \BookCode\Chap12\Particles.
- **ParticlesPS.** Этот проект иллюстрирует использование точечных спрайтов для рисования частиц. Он расположен в \BookCode\Chap12\ParticlesPS.
- **ParticlesVS.** Этот последний проект иллюстрирует использование вершинного шейдера для визуализации частиц. Он расположен в \BookCode\Chap12\ParticlesVS.

Глава 13

Имитирование одежды и анимация мешей мягких тел

Анимация в ваших проектах не должна быть такой жесткой и предопределенной. Персонажи, встречающиеся в игре (вы знаете, о чем я говорю), должны выглядеть более реалистично. Их одежда такая жесткая, их тела такие твердые. Что программисту делать с этими проблемами? Как их устранить?

Я скажу, что вам нужно - анимация одежды и мешей мягких тел! Дайте вашим персонажам реалистичную одежду, которая облегает их тела и развевается на ветру, или пышные волосы, развевающиеся во время бега за новой добычей. Позвольте вашим персонажам принимать удар за ударом противника, заставляя их тела сгибаться, а потом распрямляться вновь. Правильно, вы можете использовать реалистичную одежду и меши мягких тел. Все, что вам нужно, находится в этой главе!

В этой главе вы научитесь:

Работать с имитацией одежды и мешей мягких тел;

- Создавать одежду и меши мягких тел при помощи точек, масс и пружин;
- Накладывать силы и столкновения;
- Интегрировать движение во времени;
- Деформировать и восстанавливать форму мешей мягких тел;
- Создавать классы управления одеждой и мешами мягких тел.

Имитация одежды в ваших проектах

По сути, одежда является трехмерным мешем, состоящим из вершин, индексов и граней. Хотя они все выполняют важную задачу при визуализации, вершины имеют дополнительную задачу при имитации одежды, потому что на них влияет физика. Таким образом, вершины одежды называются точками одежды (или просто точками, для краткости). Изменяя координаты точек одежды со временем (прикладывая силы, такие как сила тяжести, ветер, или любая другая), вы можете создать эффект развевающегося элегантного кусочка материала.

Чтобы точно эмулировать реальный кусочек одежды, все точки одежды должны иметь заданную массу, которая определяет их движение в зависимости от прикладываемой силы. Точки, имеющую большую массу, требуют большей силы для сообщения ускорения, в то время как точки с меньшей массой получают большее ускорение при действии внешней силы. Это закон движения, гласящий, что сила, необходимая для перемещения объекта равна произведению массы тела на сообщаемое ему ускорение (F=ma).

Однако точкам не обязательно двигаться, они могут быть "скреплены", как если бы они были приклеены к части уровня или присоединены к мешу. Для целей этой главы я принял, что эти точки не имеют массы. (На самом деле необходимо полагать, что точки имеют бесконечную массу, потому что тогда величина силы, которую необходимо приложить для их движения, также будет бесконечно велика.)

Ребра меша одежды также играют важную роль - они соединяют точки одежды. Ну, на самом деле ребра не держат вместе точки, вместо этого, как показано на рис. 13.1, они представляют собой набор маленьких пружин, которые скрепляют точки. По мере того, как точки одежды движутся, каждая пружина пытается сохранить состояние статического равновесия, выталкивая и втягивая точки, пока не будут скомпенсированы все силы. Визуально это выражается в том, что точки как бы всегда находятся на одном и том же расстоянии между собой.

Расстояния, которые пытаются сохранить пружины, являются начальными расстояниями между точками в самом начале имитации. Каждая пружина имеет собственный способ "возвращения" к начальному положению, основанный на значениях жесткости и амортизации пружины. Жесткость позволяет определить, сколько силы используется при попытке вернуть пружину в начальное состояние, в то время как амортизация уменьшает величину создаваемой пружиной силы, позволяя сделать движения точек гладкими при воздействии на них сил пружины.

Нас интересует взаимодействие точек одежды, массы и пружины; при моделировании в основном вся работа происходит с ними. Давайте рассмотрим более подробно, как имитировать движение одежды, используя эти точки и пружины.

Определение точек одежды и пружин

Точка одежды, являющаяся аналогом вершины, - это просто точка в трехмерном пространстве. Поэтому можно определить ее координаты при помощи объекта D3DXVECTOR3. Этот объект изначально содержит точно такие же координаты, как и у соответствующей вершины меша одежды, и массу точки одежды.

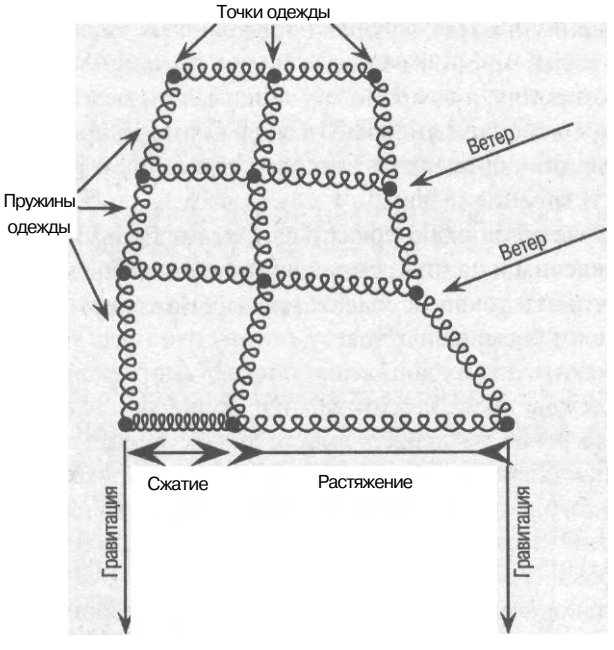


Рис. 13.1. По мере того как внешние силы действуют на точки одежды, пружины выталкивают и втягивают точки назад в форму, таким образом сохраняя общую форму меша одежды

На самом деле необходимо использовать два значения, относящиеся к массе - первое для определения фактического значения массы и второе для определения значения единицы, деленной на массу. (Второе значение используется в некоторых вычислениях.)

Я объясню, зачем необходимо использовать два значения, немного позднее; а пока я хочу показать вам, как определять точки одежды. Т. к. меш одежды может состоять из множества точек, будет разумно использовать массив векторов и вещественных значений. Создав класс, содержащий данные каждой точки и массив из объектов этого класса, мы сможем хранить точки меша одежды.

```
class cClothPoint {
  D3DXVECTOR3 m_vecPos; // трехмерные координаты точки
  float m_Mass; // Масса точки (0=прикрепленная)
  float m_OneOverMass; // 1 / Массу (0=прикрепленная к месту)
};
cClothPoint *ClothPoints = new cClothPoint[NumPoints];
```

Что же касается пружин, необходимо хранить два значения индексов точек (из массива точек), к которым присоединена пружина. Каждая пружина также имеет начальное расстояние между точками, называемое длиной покоя пружины. Значение длины покоя пружины является очень важным, т. к. оно используется для определения, была ли пружина растянута или сжата во время моделирования.

Наконец, необходимо определить значения жесткости и амортизации пружины. Эти значения очень важны, так как они определяют характер реакции пружины на действие внешней силы. Я скоро покажу вам, как использовать эти два значения; а пока просто определим их в виде двух вещественных переменных.

Вы можете определить класс, содержащий данные пружины и массив используемых объектов класса пружины, так:

```
class cClothSpring {
  DWORD m_Point1; // первая точка пружины
  DWORD m_Point2; // вторая точка пружины
  float m_RestingLength; // длина покоя пружины
  float m_Ks; // жесткость пружины
  float m_Kd; // значение амортизации пружины
};
cClothSpring *ClothSprings = new cClothSpring[NumSprings];
```

Определение массивов точек и пружин не вызывает никаких трудностей, особенно если они не содержат никакой важной информации о точках и пружинах меша одежды. В следующем разделе я расскажу вам, как получать данные пружин и точек.

Получение данных одежды из мешей

Предположим, что вы уже загрузили меш в объект ID3DXMesh, и из него вы собираетесь создавать объект одежды. В предыдущем разделе я определили две переменные, определяющие размеры массивов точек и пружин - NumPoints и NumSprings. В том разделе эти переменные были не определены, но теперь, имея корректный объект меша (предположим, что он называется pClothMesh), вы можете получить из него информацию, необходимую для вычисления этих переменных так:

```
// pClothMesh = предварительно загруженный объект ID3DXMesh
NumPoints = pClothMesh->GetNumVertices();
NumSprings = pClothMesh->GetNumFaces() * 3;
```

После того как вы определили количество точек и пружин, содержащихся в массивах одежды, можно начинать перемещение данных вершин для конструирования массива. Первым шагом к получению данных вершин меша является вычисление размера каждой его вершины, при помощи D3DXGetFVFVertexSize, как показано тут:

```
DWORDVertexStide=D3DXGetFVFVertexSize(pClothMesh->GetFVF());
```

Когда вы знаете размер вершины (называемый шагом вершины), вы можете заблокировать буфер вершин меша, получив таким образом указатель на его данные, через который можно обращаться к вершинам. Используя указатель на данные буфера вершин, вы можете просмотреть все вершины меша и получить их координаты (первые три вещественных значения каждой частицы).

Так зачем же необходимо вычислять размер каждой вершины? Когда вы просматриваете вершины, находящиеся в буфере вершин, вы не знаете, сколько данных содержит каждая вершина. Т. к. нас интересуют только координаты вершин, можно использовать их размер для пропуска ненужных данных и перехода к следующей вершине в списке.

Замечание. *На самом деле использование количества граней для определения количества создаваемых пружин одежды является не очень хорошей идеей. Т. к. каждая грань меша может разделять любое количество вершин, то получается множество повторяющихся пружин. Далее в этой главе я покажу вам способ создания пружин, который позволяет избежать повторений.*

После того как мы выяснили, для чего используется размер вершины, попробуем заблокировать буфер вершин, получить указатель на данные и просмотреть список вершин.

```
// Создать универсальную структуру вершин для получения координат
typedef struct {
  D3DXVECTOR3 vecPos;
} sVertex;

// Заблокировать буфер вершин
BYTE *pVertices;
pClothMesh->LockVertexBuffer(D3DLOCK_READONLY, \
            (BYTE**)&pVertices);

// Просмотреть список вершин и получить их координаты
for(DWORD i=0;i<NumPoints;i++) {

  // Преобразовать к структуре универсальной вершины
  sVertex *pVertex = (sVertex*)pVertices;

  // Сохранить координаты точки одежды
  ClothPoints[i].m_vecPos=pVertex->vecPos;
```

Также необходимо определить массу точки. Т. к. нет никакого источника этого значения (т. к. меш не содержит значений массы), можно просто установить какое-нибудь значение по умолчанию, скажем 1. То же самое относится и к другим значениям массы (m_OneOverMass) - установите их в 1, деленную на массу.

```
// Присвоить массу, равную единице, и 1/масса
ClothPoints[i].m_Mass  = 1.0f;
ClothPoints[i].m_OneOverMass = 1.0f / ClothPoints[i].m_Mass;
```

После установки двух значений массы можно переходить к следующей вершине и точке и продолжать так то тех пор, пока все точки одежды не будут инициализированы соответствующими данными. После этого можно разблокировать буфер вершин и продолжать, как показано тут:

```
// Перейти к следующей вершине в списке
pVertices += VertexStride;
}

// Разблокировать буфер вершин
pClothMesh->UnlockVertexBuffer();
```

Замечательно, мы уже на полпути к получению необходимых данных одежды, используемых при симуляции! Далее необходимо преобразовать ребра меша в пружины, которые бы скрепляли одежду. На этот раз необходимо заблокировать буфер индексов меша и получить три индекса, образующих каждую грань. Три точки соединяются, образуя грани, при этом каждая грань является пружиной.

Получить доступ к буферу индексов намного проще, чем получить доступ к буферу вершин. Я полагаю, что используются 16-битные индексы, потому что 32-битные индексы пока не очень широко распространены. Необходимо получить следующие подряд 16-битные значения индексов для каждой грани и создать из них три пружины, используя все сочетания соединений каждых двух имеющихся индексов.

Можно начать, заблокировав буфер индексов.

```
unsigned short *pIndices;
pClothMesh->LockIndexBuffer(D3DLOCK_READONLY, \
 (BYTE**)&pIndices);
```

Теперь необходимо просмотреть все грани и получить три индекса, образующие каждую из них. (Также необходимо объявить переменную, которая бы следила за создаваемой в данный момент пружиной.)

```
DWORD SpringNum = 0;
for(i=0;i<pClothMesh->GetNumFaces();i++) {
  unsigned short Index1 = *pIndices++;
  unsigned short Index2 = *pIndices++;
  unsigned short Index3 = *pIndices++;
```

Используя эти три индекса, создайте три пружины, представляющие собой грани.

```
  // Создать пружину от 1->2
  ClothSprings[SpringNum].m_Point1 = Index1;
  ClothSprings[SpringNum].m_Point2 = Index2;
  SpringNum++; // Увеличить число пружин
```

```
    // Создать пружину от 2->3
    ClothSprings[SpringNum].m_Point1 = Index2;
    ClothSprings[SpringNum].m_Point2 = Index3;
    SpringNum++; // Увеличить число пружин

    // Создать пружину от 1->3
    ClothSprings[SpringNum].m_Point1 = Index1;
    ClothSprings[SpringNum].m_Point2 = Index3;
    SpringNum++; // Увеличить число пружин
  }
```

После того как вы закончите обрабатывать все индексы и пружины, необходимо еще раз просмотреть список пружин, вычислив длину покоя каждой из них. Эта длина является расстоянием между точками, вычисляемым при помощи никому не известной[1] теоремы Пифагора, которая гласит, что сумма квадрата гипотенузы равна сумме квадратов катетов. Нет нужды искать эту теорему и код, ее реализующий, в книжках по математике, вместо этого вы можете использовать D3DX, как в следующем коде:

```
  for(i=0;i<NumSprings;i++) {
    // Получить индексы каждой точки пружины
    unsigned short Point1 = ClothSprings[i].m_Point1;
    unsigned short Point2 = ClothSprings[i].m_Point2;

    // Вычислить векторную разность точек
    D3DXVECTOR3 vecDiff = ClothPoints[Point2].m_vecPos - /
                ClothPoints[Point1].m_vecPos;

    // Использовать D3DX для вычисления длины вектора разности
    ClothSprings[i].m_RestingLength = D3DXVec3Length(&vecDiff);
  }
```

Наконец вы закончили создавать точки и пружины одежды! Теперь можно переходить к рассмотрению их движения!

Приложение сил для создания движения

Одежда пока является очень статичной, потому что вы на самом деле ничего с ней не сделали, кроме как сохранили координаты ее точек и данные пружин. Чтобы заставить ее двигаться и развеваться, необходимо приложить силу.

На каждую точку одежды действует отдельная сила или даже набор прилагаемых и естественных сил. Эти силы могут иметь любой источник: ветер, гравитация, трение или кто-то просто потянул одежду. Пружины выступают в роли другого вида сил; как только две точки, соединенные пружиной, движутся навстречу или друг от

1. Автор шутит. Или нет? Впрочем, после окончательного реформирования нашего образования мы уже не будем задаваться таким вопросом. - *Примеч. науч. ред.*

друга, пружина расширяется или сжимается в соответствии с законом Гука, т. е. пока сила растяжения (сжатия) пружины не скомпенсирует внешнюю силу. Другими словами, пружины пытаются создать достаточную силу для поддержания статического равновесия.

Другим аспектом сил и движения одежды является скорость. По мере того как силы, такие как гравитация, действуют на точки одежды, эти точки приобретают скорость и момент. Это означает, что точки будут продолжать двигаться в направлении действия силы, пока что-нибудь, как например сила трения или противодействия, не остановит их. Использование скорости и ускорения делает движения одежды более реалистичными, а реализм как раз то, что мы хотим видеть в своих проектах.

Замечание. *Момент вычисляется умножением массы на вектор ускорения (F=ma). Момент точки, а не векторы силы, увеличивает скорость точки, в конце концов, приводя ее в движение.*

При моделировании одежды силы, такие как гравитация и скорость, представлены направленными векторами, которые используются для перемещения точек одежды. Длина каждого вектора определяет величину действующей силы. Чтобы корректно эмулировать эти силы, необходимо добавить два связанных вектора в класс точки одежды.

```
class cClothPoint {
  D3DXVECTOR3 m_vecPos; // трехмерные координаты точки
  float m_Mass; // Масса точки (0=прикрепленная)
  float m_OneOverMass; // 1 / Mass (0=прикрепленная к месту)
  D3DXVECTOR3 m_vecForce; // Вектор силы (ускорения)
  D3DXVECTOR3 m_vecVelocity; // Вектор скорости
};
```

Изначально сила и скорость точки устанавливаются в 0, означая, что не происходит направленного движения и приложения сил. Для каждого кадра анимации необходимо сбросить векторы силы в 0 и приложить внешние силы, такие как ветер и гравитация, и внутренние, такие как сила пружин. Равнодействующая всех сил хранится в векторе m_vecForce.

Для того чтобы очистить вектор силы, просто просмотрите их список и сбросьте значения, как я сделал тут:

```
for(DWORD i=0;i<NumPoints;i++)
  ClothPoints[i].m_vecForce  = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
```

В начале имитации необходимо очистить скорость каждой точки, используя следующий код:

```
for(DWORD i=0;i<NumPoints;i++)
  ClothPoints[i].m_vecVelocity  = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
```

После того как вы очистили векторы силы и скорости, к точкам можно прикладывать различные силы. Давайте начнем с самых простых - сил тяжести и ветра.

Применение гравитации и ветра

Гравитация является естественной силой, которая притягивает объекты друг к другу. Более тяжелые объекты притягивают более маленькие. В отношении Земли это означает, что все объекты падают на землю (еще один пример, когда здоровяки побеждают). Меш одежды не является исключением. Чтобы корректно симулировать движения одежды, каждая точка ее меша должна иметь возможность упасть на землю (за исключением тех точек одежды, которые закреплены или присоединены к другим объектам, что не позволяет им двигаться).

В реальной жизни, по мере того как объекты падают, у них увеличивается скорость. Для всех объектов ускорение является более или менее постоянной величиной, равной 9.8 м/с^2. Через секунду, после того как вы уронили объект, его скорость будет 9.8 м/с. Через две секунды, скорость объекта станет равной 19.6 м/с. Причина того, что некоторые объекты падают быстрее (развивая большую скорость), чем другие, является то, что на объекты, имеющие разную форму, действует разная сила сопротивления воздуха (противодействующая сила), которая и уменьшает их ускорение. Предельная скорость - это максимальная скорость, которую может развить объект, прежде чем сила сопротивления воздуха не прекратит увеличение ускорения тела.

При моделировании гравитацию можно представить направленным вектором. Этот направленный вектор будет добавляться к вектору силы каждой точки, чтобы она могла двигаться. Для определения вектора гравитации можно использовать следующий код (полагая, что гравитация действует в отрицательном направлении оси y):

```
// Создать вектор силы тяжести (величина притяжения равна -9.8)
D3DXVECTOR vecGravity = D3DXVECTOR3(0.0f, -9.8f, 0.0f);
```

После того как вы определили вектор гравитации, вы можете добавить его к каждой точке одежды. (Не забудьте сначала очистить векторы сил точки.) В следующем кусочке кода можно заметить, что вектор гравитации масштабируется на значение массы. Это очень важно, т. к. все силы, в конце концов, масштабируются на соответствующие массы. (Это необходимо для вычисления момента, помните, что сила, необходимая для движения объекта равна массе, умноженной на ускорение.) На данный момент масштабирование вектора гравитации позволяет убедиться, что все объекты будут падать с правильной скоростью независимо от их массы.

```
for(DWORD i=0;i<NumPoints;i++)
  ClothPoints[i].m_vecForce +=  (vecGravity * \
       ClothPoints[i].m_Mass);
```

Ветер, - это другая сила, которую можно часто встретить на Земле. Вы знаете, как одежда любит поймать ветер и развеваться вместе с ним? Хорошо, именно сила ветра заставляет точки одежды двигаться. Вы можете легко создать этот же эффект при моделировании одежды. Ветер также представлен с помощью направленного вектора.

Совет. *Т. к. вектор гравитации является направленным, вы можете изменить его так, чтобы гравитация действовала вверх, влево, вправо или в любом направлении, в котором вы захотите. Представьте использование обратной гравитации, которая бы заставляла взлетать одежду!*

```
// Сделать ветер, дующий в направлении оси x
D3DXVECTOR3 vecWind = D3DXVECTOR3(0.5f, 0.0f, 0.0f);
```

Сначала вам может показаться, что силу ветра необходимо прикладывать так же, как и гравитационные силы - прибавляя направленный вектор к силе каждой точки. Чтобы приложить силу ветра, необходимо просмотреть все грани одежды и, основываясь на нормали каждой грани, вычислить вектор направления силы, создаваемой ветром.

Как вы можете видеть на рис. 13.2, каждая грань меша имеет нормаль, которая является направлением грани. На рис. 13.2 также показано направление приложения ветра и угол между силой ветра и нормалью грани.

Угол между вектором нормали грани и вектором силы очень важен - он позволяет определить величину силы, прикладываемой к точкам. Чтобы лучше это понять, представьте, что вы едете на машине и высунули руку в окно. Если рука будет расположена параллельно земле, то ветер будет обтекать ее, не отклоняя назад. Однако, если повернуть руку, величина силы (силы ветра), которая отталкивает вашу руку назад, увеличится.

Как вы можете видеть, угол между векторами определяет величину прикладываемой силы ветра. Чтобы вычислить этот угол, используется скалярное произведение нормализованного вектора нормали грани и вектора ветра. Единственной проблемой является то, где взять эту нормаль грани?

При использовании объекта ID3DXBaseMesh вы можете получить список индексов вершин, используемых каждой гранью меша. Если вы сначала получите эти индексы, а потом используете их для получения текущих координат вершин, вы можете вычислить их векторное произведение, чтобы получить нормаль грани.

Чтобы получить индексы до начала моделирования одежды, вы можете создать массив 16-битных значений (или 32-битных, в зависимости от настроек меша) и заполнить его значениями из буфера индексов меша.

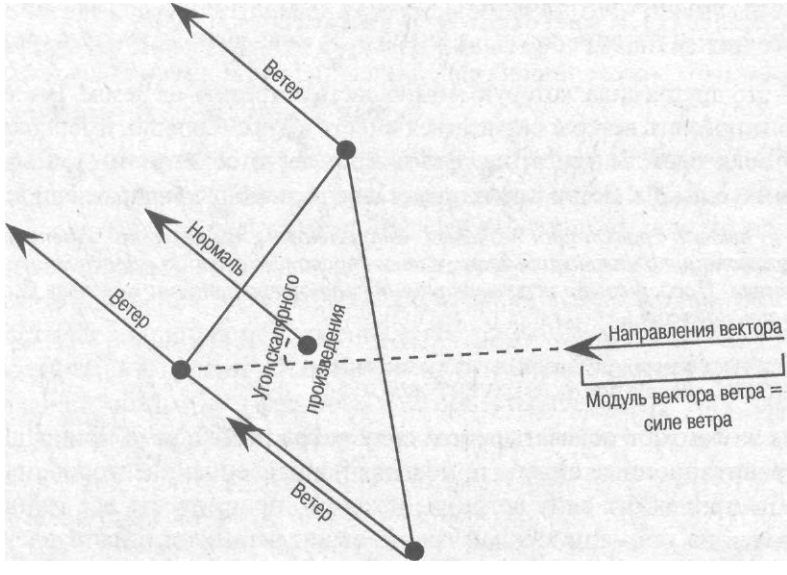


Рис. 13.2. Угол между нормалью грани и вектором ветра используется для вычисления величины силы, прикладываемой к каждой точке

```
// pClothMesh = предварительно загруженный объект ID3DXMesh

// Буфер индексов, который будет содержать индексы граней меша
unsigned short *FaceIndices = NULL;

// Создать массив индексов, основываясь на количестве граней,
содержащихся
//в объекте меша.Помните,что каждая грань имеет 3 индекса.Выделяется
// место под 32-битные индексы.
DWORD NumFaces = pClothMesh->GetNumFaces();
FaceIndices = new DWORD[NumFaces * 3];

// Теперь заблокируем меш, чтобы получить индексы
unsigned short *pIndices;
pClothMesh->LockIndexBuffer(D3DLOCK_READONLY,(void**)&pIndices);

// Просмотреть все индексы и сохранить их
for(DWORD i=0;i<NumFaces*3;i++)
  FaceIndices[i] = (unsigned short)*Indices++;

// Разблокировать буфер индексов
pClothMesh->UnlockIndexBuffer();
```

После выполнения предыдущего кусочка кода у вас будет массив, содержащий индексы - по три индекса на каждую грань. Используя эти индексы, вы можете просмотреть весь список граней и вычислить нормаль для каждой.

```
for(i=0;i<m_NumFaces;i++) {
  // Получить три вершины, образующие грань
  DWORD Vertex1 = FaceIndices[i*3];
  DWORD Vertex2 = FaceIndices[i*3+1];
  DWORD Vertex3 = FaceIndices[i*3+2];

  // Вычислить нормаль грани
  D3DXVECTOR3 vecV12 = ClothPoints[Vertex2].m_vecPos - \
      ClothPoints[Vertex1].m_vecPos;
  D3DXVECTOR3 vecV13 - ClothPoints[Vertex3].m_vecPos - \
      ClothPoints[Vertex1].m_vecPos;
  D3DXVECTOR3 vecNormal;
  D3DXVec3Cross(&vecNormal, &vecV12, &vecV13);
  D3DXVec3Normalize(&vecNormal, &vecNormal);
```

После того как мы получили нормаль грани (сохраненную в объекте vecNormal), вычисляется ее скалярное произведение с вектором ветра для определения угла между ними.

```
// Вычислить скалярное произведение нормали и ветра
float Dot = D3DXVec3Dot(&vecNormal, vecWind);
```

Имея результат скалярного произведения, можно масштабировать вектор нормали грани для вычисления величины силы, прикладываемой к каждой точке этой грани. (Помните, что каждой вершине соответствует точка.)

```
// Масштабировать нормаль на результат скалярного произведения
vecNormal *= Dot;
// Применить нормаль к вектору силы точки
ClothPoints[Vertex1].m_vecForce += vecNormal;
ClothPoints[Vertex2].m_vecForce += vecNormal;
ClothPoints[Vertex3].m_vecForce += vecNormal;
}
```

После того как вы просмотрели все индексы грани, вы получаете векторы сил точек, заполненные соответствующими значениями, которые позволяют одежде развеваться на ветру! Теперь вы готовы перейти к вычислению следующего важного набора сил, используемого при моделировании одежды — сил растяжения пружин одежды.

Применение сил пружин

Каждая пружина меша одежды первоначально имеет длину покоя. Эта длина покоя равна расстоянию между двумя точками, которые пружина соединяет. При обработке пружин необходимо вычислить текущую длину каждой пружины и определить, каким образом каждая пружина влияет на силы, действующие на точки.

Для пружин, которые стали короче их длины покоя, необходимо растянуть пружины и оттолкнуть присоединенные к ним точки друг от друга. Для пружин, которые стали длиннее длины покоя, необходимо сжать пружины, таким образом подтолкнув соединяемые пружиной точки друг к другу.

Растяжения и сжатия пружин происходят в соответствии с законом Гука, который гласит, что сила растяжения тела пропорциональна изменению длины этого тела. Это означает, что сила, создаваемая пружиной, пропорциональна длине пружины (или даже разности текущей длины пружины и ее длины покоя). Математически, закон Гука выглядит так:

$$F = к*x$$

Где F представляет собой действующую силу, x - изменение размера пружины (разницу текущей длины пружины и ее длины покоя, как показано на рис. 13.3), к - константу пружины; она определяет насколько величина изменения длины пружины влияет на силу. Чем больше константа пружины, тем большая действует сила.

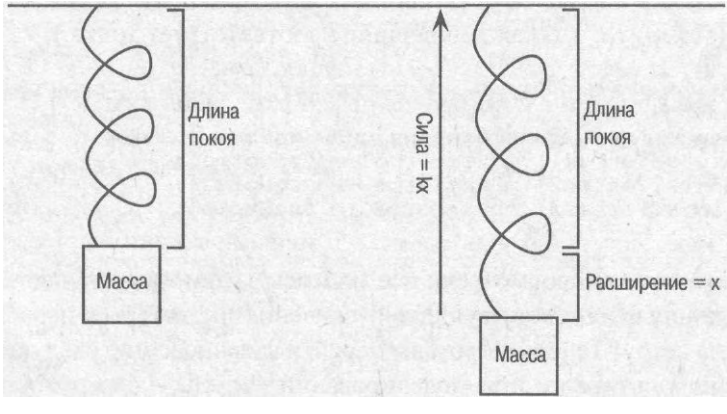


Рис. 13.3. Пружина, изображенная слева, находится в покое (равновесии), в то время как пружина справа — растянута. Величина силы пружины определяется из изменения длины пружины и ее константы

Например, вы хотите вычислить величину силы растяжения пружины. Эта пружина имеет длину покоя 128 единиц и в данный момент растянута до 200 единиц. Вычитая длину покоя из текущей длины, получим 200-128=72; это значение является изменением размера пружины x. Умножим полученное значение на константу пружины k. В результате получим величину прилагаемой силы (F) равной 0.4x72=28.8.

28.8 - это значение силы, используемой для сближения точек пружины. На каждую точку, находящуюся на концах пружины, действуют силы одинаковые по модулю и противоположные по направлению. После того как вы получили величину силы, вы можете добавить ее к остальным силам точки. Позже эта рассчитанная равнодействующая сила будет использована для вычисления ускорения, которое в свою очередь изменяет скорость точек.

Единственное, о чем я еще не сказал, это откуда брать значения константы пружин, используемых в вычислении силы. Помните, ранее мы определили вещественное значение, представляющее жесткость пружины? Ну, значение жесткости это и есть константа пружины! Вы можете использовать значение константы/жесткости пружины, обычно обозначаемое символом k или ks, для масштабирования прикладываемой силы пружины.

Я уверен, вас интересует, какие допустимые значения жесткости пружины можно использовать. Ну, простого ответа не существует. Чем больше жесткость, тем большая сила производится пружиной. Обычно этого достаточно - установка большого значения жесткости будет замечательно работать. Единственной проблемой при моделировании является то, что большое значение жесткости создаст слишком большую прикладываемую силу, в результате чего одежда может двигаться непредсказуемым образом. Если же используется слишком маленькое значение жесткости, то одежда вытянется и будет выглядеть как кусок резины.

Опять же, какого значения жесткости будет достаточно? Для примеров этой главы я использовал значение жесткости равной 4. При моделировании я предлагаю вам самостоятельно устанавливать жесткость в разнообразные значения и посмотреть какое значение подойдет лучше всего.

Чтобы рассчитать пружины при моделировании, необходимо просмотреть весь их список в меше, используя цикл for...next, и получить текущие их длины (расстояния между двумя их точками).

```
for(DWORD i=0;i<NumSprings;i++) {

   // Получить значения индексов двух точек
  DWORD Point1 = ClothSprings[i].m_Point1;
  DWORD Point2 = ClothSprings[i].m_Point2;
```

```
// Получить вектор между точками
D3DXVECTOR3 vecSpring = ClothPoints[Point2].m_vecPos - \
    ClothPoints[Point1].m_vecPos;

// Получить длину вектора между двумя точками
float SpringLength = D3DXVec3Length(&vecSpring);
```

Текущее расстояние между двумя точками пружины является очень важным. Помните, ранее в этой главе, мы говорили, что пружина может растягиваться или сжиматься. Вычислив сначала вектор между двумя точками пружины, вы можете потом определить текущую длину пружины, используя функцию D3DXVec3Length. Для вычисления силы пружины используйте закон Гука. Он включает в себя умножение величины изменения длины пружины (длина покоя, вычтенная из текущей длины) на значение жесткости пружины для получения скаляра силы пружины.

```
// Вычислить скаляр прикладываемой силы
float SpringForce = Spring->m_Ks *
     (SpringLength - Spring->m_RestingLength);
```

Вектор vecSpring также является очень важным при определении направления движения точек. Нормализовав вектор текущей длины пружины и умножив его на SpringForce, вы можете быстро вычислить вектор силы, действующий на каждую точку, присоединенную к пружине.

```
// Нормализовать вектор пружины
vecSpring /= SpringLength;

// Умножить на значение силы
vecSpring *= SpringForce;
```

Таким образом получим вектор, представляющий собой величину силы, прикладываемой к каждой точке пружины. Добавим его к вектору суммарной силы первой точки и вычтем его из вектора суммарной силы второй точки для перемещения их в правильном направлении (т. о. либо сближая, либо отдаляя их друг от друга)

```
// Приложить силу к вектору сил точек
ClothPoints[Point1].m_vecForce += vecSpring;
ClothPoints[Point2].m_vecForce -= vecSpring;
}
```

Вот и все, что касается пружин одежды! На данный момент вы почти готовы обновить скорость и положение каждой точки на основе результирующей силы, но сначала необходимо приложить самую важную силу - трение.

Замедление движения с помощью трения

Чтобы все выглядело реалистично, одежда должна придерживаться законов трения. По мере того как одежда движется в среде, такой как воздух, она замедляется. Трение фактически является противодействующей силой, независимо оттого является ли она гравитационной силой, турбулентностью воздуха или трением. Для упрощения, я буду называть комбинацию этих сил просто трением.

Если вы не примените трение к точкам одежды, у вас возникнет проблема - одежда никогда не перестанет двигаться, и моделирование одежды будет нестабильным (из-за скоростей ваш меш выйдет из-под контроля). Конечно, сила тяжести будет тянуть тело вниз, но горизонтальное движение никогда не прекратиться. Применив трение, вы инициируете медленное замедление движения, если против трения не действует большая сила. Также, используя амортизированное трение пружин, вы можете увеличить стабильность, сократив величину силы пружины.

Чтобы применить трение (в данном случае называемое линейной амортизацией) к точкам одежды, необходимо уменьшить вектор силы каждой точки на небольшой процент вектора ее скорости. (Амортизация на самом деле является коэффициентом пропорциональности силы к скорости, F=kV.) Это процентное соотношение, заданное в виде вещественного числа, обычно находится в диапазоне от -0.04 до -0.1. (Отрицательный диапазон означает, что вы уменьшаете прикладываемую силу, вместо того чтобы увеличивать ее.) Чем больше это значение, тем большая величина трения применяется. Например, при использовании значения -1 создается впечатление, что одежда находится в чане с маслом!

Предположим, что используется значение линейной амортизации, равное -0.04. Вы можете просмотреть все точки прямо сейчас. Для каждой точки необходимо прибавить вектор скорости, масштабированный на значение амортизации к вектору силы точки, как показано в следующем коде:

```
for(DWORD i=0;i<NumPoints;i++) {
  ClothPoints[i].m_vecForce+= (-0.04f * \
       ClothPoints[i].m_vecVelocity);
}
```

Что же касается трения, действующего на пружины (называемого амортизирующим трением), необходимо добавить значение амортизации в вычисление силы пружины, как было показано ранее. Значение амортизации, как вы уже, наверное, догадались, было определено в классе, содержащем данные пружины! Значение трения должно быть немного больше, чем значение линейной амортизации, и обычно лежит в диапазоне от 0.1 до 0.5. В примерах я буду использовать значение 0.5.

Чтобы применить амортизацию к пружинам, вернитесь к вычислению их сил.

```
for(DWORD i=0;i<NumSprings;i++) {

  // Получить номера индексов двух точек
  DWORD Point1 = ClothSprings[i].m_Point1;
  DWORD Point2 = ClothSprings[i].m_Point2;

  // Получить вектор, соединяющий точки
  D3DXVECTOR3 vecSpring = ClothPoints[Point2].m_vecPos - \
      ClothPoints[Point1].m_vecPos;

  // Получить длину вектора, соединяющего точки
  float SpringLength = D3DXVec3Length(&vecSpring);

  // Вычислить скаляр прикладываемой силы
  float  SpringForce = Spring->m_Ks *
       (SpringLength - Spring->m_RestingLength);
```

Теперь необходимо вычислить значения скаляра амортизации пружины на основе нормализованной относительной скорости двух точек (разности их скоростей) и значения амортизации пружины. Чтобы вычислить нормализованную относительную скорость двух точек, необходимо просто посчитать разность двух векторов скоростей и поделить ее на текущую длину пружины.

```
// Получить относительную скорость точек
D3DXVECTOR3 vecVelocity = CState2->m_vecVelocity - \
    CState1->m_vecVelocity;
float Velocity = D3DXVec3Dot(&vecVelocity, \
    &vecSpring) / SpringLength;
// Применить значение амортизации пружины (m_Kd)
float DampingForce = Spring->m_Kd * Velocity;
```

Далее идет оставшийся код вычисления силы пружины, за исключением того, что к вектору пружины применяется не только сила пружины (SpringForce), но и сумма скаляров силы пружины и силы амортизации (DampingForce).

```
// Нормализовать вектор пружины
vecSpring /= SpringLength;
// Умножить на скаляры силы  (скаляры пружины и амортизации)
vecSpring *=  (SpringForce + DampingForce);

// Применить силу к вектору силы точки
ClothPoints[Point1].m_vecForce += vecSpring;
ClothPoints[Point2].m_vecForce -= vecSpring;
}
```

После того как вы скорректировали силы в соответствии со значением трения, вы можете прикладшать эти силы к скорости каждой точки и вычислять их движение.

Приложение сил и передвижение точек во времени

На данный момент силы каждой точки должны быть вычислены и сохранены в векторе силы. Эти векторы сил представляют собой величины ускорений, применяемые к скоростям каждой с течением времени. Векторы скоростей также имеют направление и определяют направление и скорость движения каждой точки при моделировании.

Итак, если имеется вектор силы равный 0, -9.8, 1, ускорение будет равно 9.8м/с в отрицательном направлении оси y и 1 м/с в положительном направлении оси z. Этот перевод силы в ускорение (использование одного и того же вектора для ускорения и силы) является немного неправильным, но зато замечательно работает при моделировании, так что пока мы проигнорируем законы физики.

Единственное, о чем необходимо побеспокоится сейчас, - это время. Видите ли, каждый раз, когда вы обновляете имитацию, необходимо получить значение прошедшего времени, чтобы точки одежды могли перемещаться в соответствии с ним. Предположим, вы хотите рассчитать движение точек только для 400 миллисекунд. Как это можно сделать, не рассчитывая имитацию 400 раз, т. е. раз в миллисекунду? Необходимо получить имитацию одежды в реальном времени, черт побери!

Теперь волнуйтесь, мой друг! Используя так называемое явное интегрирование, вы можете определить ускорение, достигнутое за промежуток времени, и как это ускорение влияет на скорость точки за тот же период времени. Так что вместо того, чтобы выполнять моделирования каждую миллисекунду, вы можете выполнять его каждые 40 миллисекунд, таким образом значительно ускоряя расчеты!

Совет. *Если вы хотите задать время в миллисекундах, а не в секундах, как показано в коде примера, просто добавьте дробное десятичное значение. Чтобы его вычислить, разделите 1 на 1000 и умножьте на количество миллисекунд. Например, одна миллисекунда это 0.001, десять миллисекунд это 0.01, а сто миллисекунд это 0.1.*

В данном примере я использовал интегрирование вперед, которое позволяет масштабировать вектор или значение на набор дискретных величин (время). Используя силу точки (которая представляет собой ее ускорение), вы можете вычислить, насколько изменится скорость за определенный период времени, как показано далее:

```
// TimeElapsed = количество прошедших секунд
ClothPoints[Num].m_vecVelocity+= TimeElapsed * \
     ClothPoints[Num].m_vecForce;
```

Используя тот же самый период времени, вы можете вычислить насколько переместится точка, используя скорость.

```
ClothPoints[Num].m_vecPos += TimedElapsed * \
     ClothPoints[Num].m vecVelocity;
```

Итак, интегрирование вперед является не чем иным как масштабированием векторов. Просто, не так ли? Единственное, о чем я не упомянул, это то, что вам необходимо принять во внимание массы точек. Помните, чем большую массу имеет объект, тем большую силу необходимо приложить, чтобы сдвинуть его. Я имею ввиду момент, т. е. то, что комбинированная сила точки (представленная вектором силы) должна быть масштабирована на массу, чтобы вычислить фактически применяемое ускорение. Масштабирование вектора силы на значение массы (на самом деле на значение 1/масса) при интегрировании замечательно с этим справляется.

```
ClothPoints[Num].m_vecVelocity += TimeElapsed * \
     ClothPoints[Num].m_OneOverMass * \
     ClothPoints[Num].m_vecForce;
```

Подытожив, необходимо умножить вектор силы каждой точки на массу (1/масса) и прошедшее время, после чего добавить результирующий вектор к вектору скорости точки. Вот отрывок кода, который выполняет все это для каждой точки одежды:

```
// TimeElapsed = количество обрабатываемых секунд
for(DWORD i=0;i<NumPoints;i++) {

  // Интегрировать скорость
  ClothPoints[Num].m_vecVelocity += TimeElapsed * \
       ClothPoints[Num].m_OneOverMass * \
       ClothPoints[Num].m_vecForce;

  // Интегрировать положение
  ClothPoints[Num].m_vecPos += TimedElapsed * \
       ClothPoints[Num].m_vecVelocity;
}
```

Вот и все, мой друг! Мы успешно вычислили скорость и обновили положение каждой точки меша одежды! Далее необходимо взять данные точек и использовать их для воссоздания и визуализации меша одежды.

Воссоздание и визуализация меша одежды

Т. к. точки одежды изменили свое положение, необходимо перестроить начальный меш так, чтобы изменения были заметны. Т. к. вы уже загрузили исходный меш из .X файла в объект ID3DXMesh, все, что остается сделать, это заблокировать буфер вершин меша, сохранить в него координаты точек и пересчитать нормали (если исходный меш их использует).

Еще раз создадим универсальную структуру, содержащую только вектор. Этот вектор используется для получения доступа к данным координат вершин меша.

```
typedef struct {
  D3DXVECTOR3 vecPos;
} sVertex;
```

После этого необходимо просто заблокировать буфер вершин меша и просмотреть все точки одежды (опять!), сохраняя их координаты.

```
// Заблокировать буфер вершин меша
BYTE *pVertices;
pClothMesh->LockVertexBuffer(0, (BYTE**)&pVertices);

for(DWORD i=0;i<NumPoints;i++) {

  // Преобразовать указатель на вершину
  sVertex *pVertex = (sVertex*)pVertices;

  // Сохранить координаты вершины
  *pVertex = ClothPoints[i].m_vecPos;

  // Перейти к следующей вершине
  pVertices += VertexStride;
 }

// Разблокировать буфер вершин
pClothMesh->UnlockVertexBuffer();
```

После того как вы воссоздали меш, вы можете визуализировать его обычным образом, просмотрев список материалов меша и используя функцию ID3DXMesh::DrawSubset для визуализации каждого поднабора. Если вы хотите повторно использовать исходный меш, например, для сброса имитации одежды, необходимо восстановить исходные координаты его вершин.

Восстановление исходного меша

Вы, наверное, заметили, что данные точек одежды очень динамичны. Не существует простого способа восстановить начальную ориентацию каждой точки меша одежды. Зачем вообще необходимо восстанавливать исходный меш? Возможно, чтобы начать заново моделирование или использовать меш одежды в его исходном состоянии.

Самым простым способом восстановления данных точек исходного меша одежды является добавление еще одного вектора в класс cClothPoint.

```
class cClothPoint {
  D3DXVECTOR3 m_vecOriginalPos; // Начальные координаты
  D3DXVECTOR3 m_vecPos; // Трехмерные координаты точки
  float m_Mass; // Масса точки (0=прикрепленная)
  float m_OneOverMass; // 1 / Массу (0=прикрепленная к месту)
  D3DXVECTOR3 m_vecForce; // Вектор силы (ускорения)
  D3DXVECTOR3 m_vecVelocity; // Вектор скорости
};
```

Дополнительный вектор (vecOriginalPos) хранит начальные координаты точек одежды. При создании массива точек одежды убедитесь, что вы сохранили начальное положение в этом новом векторе.

```
//...После загрузки точек одежды в массив
for(DWORD i=0;i<NumPoints;i++)
  ClothPoints[i].m_vecOriginalPos = ClothPoints[i].m_vecPos;
```

Когда вы захотите восстановить данные исходных точек одежды, просто скопируйте данные их этого нового вектора в вектор положения точки.

```
for(DWORD i=0;i<NumPoints;i++)
  ClothPoints[i].m_vecPos = ClothPoints[i].m_vecOriginalPos;
```

Использование начального положения замечательно подходит для восстановления данных исходного меша и также полезно при определении расстояния между текущим и начальным положением точек, как при моделировании мешей мягких тел, которые будут рассмотрены далее в этой главе.

А пока, просто продолжим улучшение имитации одежды, добавив дополнительные пружины в меш, таким образом улучшая стабильность одежды при имитировании.

Добавление дополнительных пружин

До этого момента я говорил о пружинах, как если бы они были созданы из ребер меша. На самом деле это очень неаккуратный подход к созданию пружин из данных меша, если достоверное моделирование одежды является вашей целью. Например, посмотрите на рис. 13.4. Мы думаем, что показанные пружины будут скреплять меш.

На рис. 13.4 все выглядит замечательно, пока вы не запустите моделирование. Проблемой является то, что меш одежды использует ребра многоугольников в качестве пружин, что может вызвать его сворачивание по прошествии некоторого времени, как если бы он был сделан из очень тонкого материала. Это может быть приемлемым для большинства случаев, но как насчет тех случаев, когда необходимо, чтобы одежда была жестче и с трудом изменяла форму?

Весь секрет в том, что чем больше пружин содержит меш, тем жестче становится одежда. Все правильно - добавив еще несколько пружин, вы можете заставить одежду мяться и изгибаться вместо того, чтобы распадаться на непрочные группы вершин. Конечно же, важным является положение пружин, так что посмотрите еще раз на рис. 13.4, чтобы понять, что необходимо сделать. На рис. 13.5 вы видите новый меш, в который добавлено несколько новых пружин.

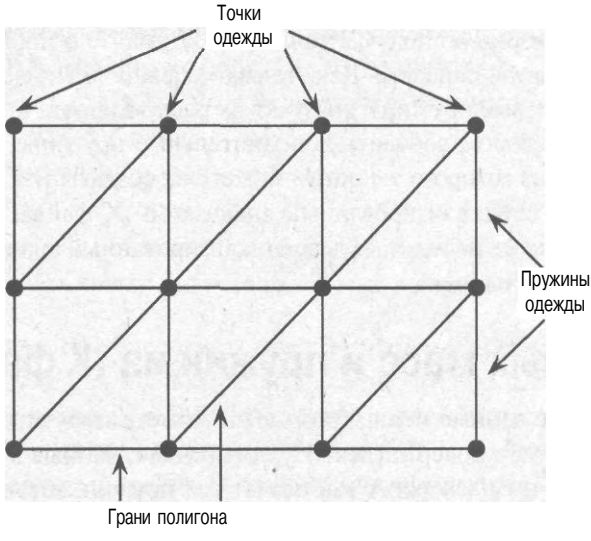


Рис. 13.4. Набор пружин, созданный из ребер многоугольников изображенного меша

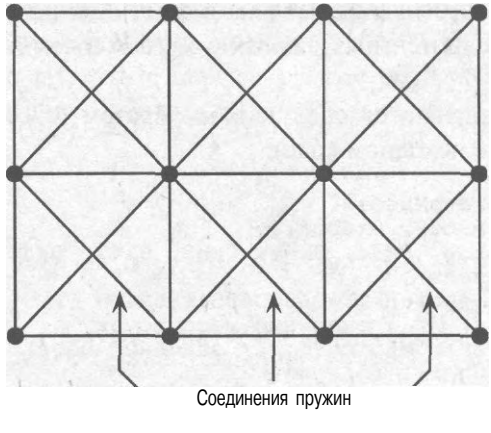


Рис. 13.5. Теперь меш одежды имеет набор соединенных между собой пружин, которые расположены на его граня

Не забудьте, что пружины сближают или отдаляют две точки друг от друга, в зависимости от расстояния между ними. Внимательно посмотрев на рис. 13.5, вы можете увидеть, что правильно расположив несколько дополнительных пружин, вы можете усилить структуру одежды, потому что эти пружины будут вынуждать перемещаться

точки от других точек, если одежда попытается свернуться. Сначала очень трудно представить, о чем я говорю, но подумайте на этим. Думайте о новых пружинах как о средстве от образования складок. Как только одежда начинает сворачиваться и образуются складки, новые пружины сразу же их распрямляют.

Самым простым способом добавить дополнительные пружины в меши является составления их списка, из которого вы потом можете их создать. Вы можете получить этот список, используя специализированный анализатор .X файла. И пока вы этим заняты, почему бы заодно не использовать специализированный анализатор .X файлов для получения данных масс точек!

Загрузка данных масс и пружин из .X файла

Не беря во внимание данные меша, которые вы можете загрузить из .X файла, что еще полезного может в нем содержаться? Ну, для начала, данные меша не содержат информации о массе точек одежды. А как насчет тех пружин, которые возможно вы захотите добавить в одежду для обеспечения дополнительной жесткости?

Хорошо, поводов для беспокойства нет - .X замечательно подходит для хранения дополнительных данных одежды, таких как значения масс точек и используемые пружины. Добавив два небольших шаблона в файл .X и набор классов в исходные файлы, получение дополнительных данных о массах точек и пружинах становится очень простым.

Два шаблона определяются следующим образом (с определением макроса GUID, используемым в исходном коде):

```
// DEFINE_GUID(ClothMasses,
// 0xf5ad0f93, 0x9bf2, 0x4bcf,
// 0xb7, 0xcf, 0xd6, 0x8c, 0xd6, 0xb5, 0x41, 0x56);
template ClothMasses {
  <F5AD0F93-9BF2-4bcf-B7CF-D68CD6B54156>
  DWORD NumPoints; // # значений масс точек
  array FLOAT Mass[NumPoints]; // значения масс
}

// DEFINE_GUID(ClothSprings,
// 0x8c08b088, 0x728e, 0x46c8,
// 0xbe, 0x87, 0x72, 0x67, 0x2b, 0x81, 0xdb, 0x11);
template ClothSprings {
  <8C08B088-728E-46c8-BE87-72672B81DB11>
  DWORD NumSprings; // # загружаемых пружин
  DWORD NumVertices; // NumSprings * 2
  array DWORD Vertex[NumVertices]; // используемые точки пружин
}
```

Первый шаблон ClothMasses содержит две переменные. Первая переменная NumPoints соответствует количеству точек, содержащихся в меше, и количеству значений масс, содержащихся в объекте. Второй объект Mass является массивом значений масс точек, которые могут меняться от 0 (точка прикреплена к месту) до 1.

Вот пример объекта данных, который использует шаблон ClothMasses для хранения шести значений масс точек:

```
ClothMasses {
  6;
  1.0, 0.0, 0.0, 1.0, 1.0, 1.0;
}
```

Список значений масс соответствует точкам меша одежды - первое значение массы используется для первой точки одежды, второе для второй и т. д. В моем примере, первое, четвертое, пятое и шестое значения масс установлены в 1, в то время как второе и третье установлены в 0.

Второй приведенный шаблон ClothSprings используется для определения точек меша одежды, которые соединяют пружины. Переменная NumSprings используется для задания количества определяемых пружин, а переменная NumVertices задает, на сколько вершин (точек) влияют пружины. Num Vertices должно всегда равняться удвоенному количеству пружин. Наконец, Vertex является массивом значений индексов, определяющих соединяемые точки.

Предположим, что вы хотите определить три пружины, соединяющие шесть вершин. Вот пример объекта данных, который соединяет 0 и 1, 1 и 2, 2и0 точки.

```
ClothSprings {
  3;
  6;
  0,1,
  1,2,
  2,0;
}
```

После того как вы определили шаблоны и объекты данных в файле .X, вы можете использовать специализированный анализатор .X файлов, чтобы считать соответствующие значения в игру. При создании анализатора .X данных одежды, вы можете даже включить код моделирования в унаследованный класс. В демонстрационной программе одежды, поставляемой с книгой, я сделал это - объединил анализатор и имитатор одежды в один класс.

Создание анализатора .X данных одежды

В который раз, класс анализатора .X cXParser из главы 3 приходит нам на помощь. Здесь вы унаследуете класс от класса cXParser, который будет искать два различных шаблона - один, содержащий значения массы одежды (ClothMasses), и другой, который будет хранить информацию о пружинах (ClothSprings).

Для хранения данных обоих объектов замечательно подходят классы точки и пружины, определенные ранее в этой главе, - все, что необходимо, это класс анализатора для перечисления объектов и получения их данных. Вот пример класса анализатора .X, который сделает все для вас:

```
class cClothMesh : cXParser
{
  protected:
   DWORD m_NumPoints; // # точек в одежде
   cClothPoint *m_ClothPoints; // данные точек
   DWORD m_NumSprings; // # пружин в одежде
   cClothSpring *m_ClothSprings; // данные пружин

  protected:
   // Анализирует файлы .X и получает данные масс и пружин
   BOOL ParseObject(IDirectXFileData *pDataObj,
        IDirectXFileData *pParentDataObj,
        DWORD Depth,
        void **Data, BOOL Reference)
   {
    const GUID *Type = GetObjectGUID(pDataObj);
    DWORD *DataPtr = (DWORD*)GetObjectData(pDataObj, NULL);

    // Считать массы точек одежды
    if(*Type == ClothMasses) {

     // Получить количество присваиваний масс
     DWORD NumPoints = *DataPtr++;

     // Скопировать значения масс
     float MassPtr = (float*)DataPtr;
     for(DWORD i=0;iXNumPoints;i++) {
      m_ClothPoints[i].m_Mass = *MassPtr++;

      // Вычислить 1/масса
      m_ClothPoints[i].m_OneOverMass = \
            (m_ClothPoints[i].m_Mass==0.0f) ? \
           0.0f:(1.0f/m_ClothPoints[i].m_Mass);
     }
    }

    // Считать данные пружин
    if(*Type == ClothSprings) {
```

```
        // Освободить предыдущие данные пружин
        delete [] m_ClothSprings; m_ClothSprings = NULL;

        // Получить новое количество пружин и вершин
        DWORD NumSprings = *DataPtr++;
        DWORD NumVertices = *DataPtr++;

        // Выделить память под пружины
        m_ClothSprings = new cClothSpring[NumSprings];

        // Загрузить данные каждой пружины
        for(DWORD i=0;i<NumSprings;i++) {
         m_ClothSprings[i].m_Point1 = *DataPtr++;
         m_ClothSprings[i].m_Point2 = *DataPtr++;
        }
       }

       return ParseChildObjects(pDataObj, Depth, \
              Data, Reference);
    }
  }
```

Только что показанный код класса cXParser является очень простым, так что я не буду подробно его объяснять. Все, что делает функция ParseObject, - это загрузка любых данных пружины или масс точек в массив классов, используемых в программе. Чтобы существенно повысить функциональность класса анализатора, вы можете объединить его с классом, создающим и управляющим мешем одежды.

Чтобы задействовать объект анализатора .X, просто вызовете cClothMesh::Parse, задав в качестве параметра имя используемого файла. Вы заметите, что массивы точек одежды и данных пружин встроены в класс анализатора, поэтому убедитесь, что вы выделили под них память, прежде чем анализировать файл .X. Позже в этой главе вы увидите, как создать завершенный класс меша одежды, наследуемый от класса анализатора .X для загрузки данных масс и пружин.

А пока, пришло время оживить имитацию, добавив обнаружение столкновений и реакцию на них.

Работа с обнаружением столкновений и реакцией на них

Важным аспектом, о котором я еще не говорил, являются столкновения. Помните, что меш считается твердым объектом. Т. е. он взаимодействует с другими объектами в виртуальном мире и должен соответствующим образом реагировать. Например, плащ персонажа скатится с его плеч и будет развеваться у него за спиной. Плащ никогда не должен пересекать меш персонажа.

Возможно, одежда висит на жерди. Ваш персонаж (или любой другой объект, который заденет одежду) вынудит одежду отклониться от жерди и развеваться по мере оседания. Поговорим о некоторых замечательных идеях - используя обнаружение столкновений и реакцию на них, перед вами открываются привлекательные возможности при моделировании одежды!

Работать с обнаружением столкновений и реакцией на них подозрительно просто. После того как вы примените скорость каждой точки, вы проверяете, находится ли точка внутри другого твердого объекта. Например, вы можете проверить, лежит ли точка одежды в сфере, используя простую проверку расстояния, или можете выполнить проверку точки и плоскости, чтобы узнать с какой стороны плоскости лежит точка (см. рис. 13.6).

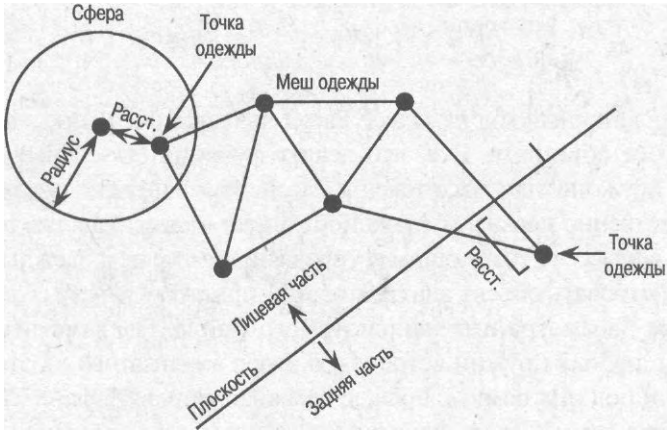


Рис. 13.6. Точка сталкивается со сферой, если она расположена ближе, чем радиус сферы, или если она расположена за плоскостью

Итак, при обнаружении столкновений используются два простых объекта: сфера и плоскость. Необходимо создать класс, который бы отвечал за сталкиваемые объекты.

Определение объектов столкновений

Для хранения данных, относящихся к объекту столкновений, вы можете использовать следующий класс (и два макроса):

```
// макрос, определяющий тип сталкиваемы  объектов
#define COLLISION_SPHERE 0
#define COLLISION_PLANE 1
```

```
class cCollisionObject {
  public:
    DWORD m_Type; // Тип объекта

    D3DXVECTOR3 m_vecPos; // координаты сферы
    float m_Radius; // Радиус сферы
    D3DXPLANE m_Plane; // Параметры плоскости

    cCollisionObject *m_Next; // Следующий объект в связанном списке

  public:
    cCollisionObject() { m_Next = NULL; }
    ~cCollisionObject() { delete m_Next; m_Next = NULL; }
};
```

В классе cCollisionObject содержаться четыре основные переменные. Необходимо установить переменную т_Type с помощью макроса, представляющего тип объекта, содержащегося в классе, - либо сфера (COLLISION_SPHERE), либо плоскость (COLLISION_PLANE).

Необходимо хранить координаты объекта столкновения сферы в m_vecPos. Эти координаты определяют положение в трехмерном пространстве, совсем как у трехмерных мешей. Также, если используется объект сфера, убедитесь, что вы установили ее радиус в переменной m_Radius.

Однако, если вы используете объект столкновения плоскость, вам только необходимо установить ее соответствующие параметры в объекте mPlane. Параметры mPlane.a, m_Plane.b и m_Plane.c являются нормализованными векторами направления плоскости, в то время как m_Plane.d является смещением по вектору направления, определяющему положение плоскости.

Далее в классе располагается указатель m_Next, который используется для хранения структуры связанного списка. Используя связанный список, вы можете создать полный набор объектов столкновений, используемых при моделировании одежды. Далее в этом разделе вы увидите, как его использовать.

Покончив с объявлениями переменных, перейдем к рассмотрению функций cCollisionObject. Класс содержит только две функции, являющиеся конструктором и деструктором, которые используются для очистки связанного списка указателей при инициализации и для его освобождения при уничтожении.

Кроме класса cCollisionObject вы можете создать еще один класс, который бы содержал связанный список набора объектов. Я рекомендую использовать отдельный класс, позволяющий хранить множество списков объектов столкновений, например, один для статичной геометрии, а другой — для динамичной. Объект столкновения может перемещаться, взаимодействуя с объектом одежды и вынуждая ее развеваться.

Второй класс, названный cCollision, определяется так:

```
class cCollision {
  public:
   DWORD m_NumObjects; // # объектов
   cCollisionObject *m_Objects; // список объектов

  public:
   cCollision() { m_NumObjects = 0; m_Objects = NULL; }
   ~cCollision() { Free(); }

   void Free()
   {
    // Удалить связанный список объектов
    delete m_Objects; m_Objects = NULL;
    m_NumObjects = 0 ;
   }

   void AddSphere(D3DXVECTOR3 *vecPos, float Radius)
   {
    // Создать новый объект
    cCollisionObject *Sphere = new cCollisionObject ();

    // Установить данные сферы
    Sphere->m_Type = COLLISION_SPHERE;
    Sphere->m_vecPos = (*vecPos);
    Sphere->m_Radius = Radius;

    // Добавить сферу в связанный список и увеличить счетчик
    Sphere->m_Next = m_Objects;
    m_Objects = Sphere;
    m_NumObjects++;
   }

   void AddPlane(D3DXPLANE *PlaneParam)
   {
    // Создать новый объект
    cCollisionObject *Plane = new cCollisionObject();

    // Установить параметры плоскости
    Plane->m_Type = COLLISION_PLANE;
    Plane->m_Plane = (*PlaneParam);

    // Добавить плоскость в связанный список и увеличить счетчик
    Plane->m_Next = m_Objects;
    m_Objects = Plane;
    m_NumObjects++;
   }
};
```

Объявление класса cCollision включает две переменные (m_NumObjects и m_Objects), которые содержат количество загруженных объектов столкновений и их связанный список соответственно. Также в вашем распоряжении находятся

пять функций, первые из которых конструктор и деструктор. Они используются для очистки данных класса и вызова функции Free соответственно.

Функция Free предназначена для удаления связанного списка объектов и обнуления количества загруженных в него объектов столкновений. Функции AddSphere и AddPlane используются для создания нового объекта столкновения и добавления его в связанный список, хранящийся в классе cCollision.

Чтобы добавить в связанный список сферу, вызовите AddSphere, задав в качестве параметров координаты центра сферы и ее радиус. Чтобы добавить плоскость, вызовите AddPlane и передайте в качестве параметра объект D3DXPLANE, который бы задавал нормаль плоскости и ее смещение от начала координат.

Чтобы использовать данные объекта столкновения, необходимо создать функцию, которая бы просматривала все точки одежды и для каждой точки проверяла бы, сталкивается ли точка с объектом столкновения. Это выполняется простой проверкой расстояния, как вы увидите далее.

Обнаружение и реакция на столкновения

Создаваемая функция проверки столкновений должна просмотреть все точки одежды и для каждой точки проверить, происходит ли ее столкновение с каким-либо объектом столкновений, содержащимся в связанном списке объектов. Вызовете функцию CheckCollision; в качестве параметров задайте указатель на объект cCollision для проверки столкновений точки и объекта и матрицу преобразования, которая используется для расположения и ориентирования объектов столкновений в трехмерном мире.

```
void CheckCollisions(cCollision *pCollision, \
     D3DXMATRIX *matTransform)
{

  // Просмотреть все точки
  for(DWORD i=0;i<NumPoints;i++) {

  // Не обрабатывать точки с нулевой массой
  if(ClothPoints[i].m_Mass != 0.0f) {
```

В предыдущем кусочке кода вы начали просматривать все точки одежды. Я полагаю, что количество точек определяется переменной NumPoints, а данные самих точек определены в массиве ClothPoints. При просмотре необходимо сначала убедится, что точка имеет ненулевую массу, т. е. что она может перемещаться. Если точка может перемещаться, тогда необходимо просмотреть все объекты столкновений и определить, сталкивается ли точка с ними.

```
// Просмотреть каждый объект столкновений
cCollisionObject *pObject = pCollision->m_Objects;
while(pObject) {

// Проверить, сталкивается ли точка с объектом сферой
if(pObject->m_Type == COLLISION_SPHERE) {
```

Первый тип объекта, который проверяется на столкновение, - это сфера. Помните, что сфера располагается при помощи трехмерного вектора, а ее радиус определяется вещественным значением, которое вы установили при вызове AddSphere. Как показано на рис. 13.7, точка сталкивается со сферой, если расстояние от центра сферы до просматриваемой точки меньше ее радиуса.

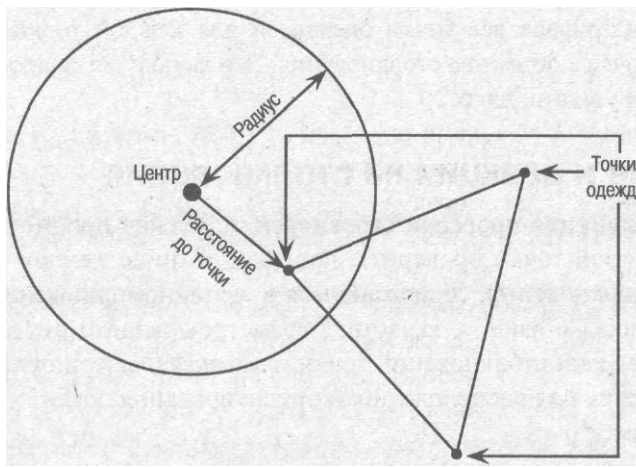


Рис. 13.7. Точка сталкивается со сферой, если расстояние от центра сферы до точки меньше, чем радиус сферы

Т. к. для расположения объекта столкновения используется преобразование, необходимо применить часть преобразования, ответственного за перемещение, к вектору положения сферы, прежде чем проверять столкновения.

```
// Сохранить координаты сферы в локальный вектор
D3DXVECTOR3 vecSphere = pObject->m_vecPos;

// Переместить сферу, если необходимо
if(matTransform) {
 vecSphere.x  += matTransform->_41; // Translate x
 vecSphere.y  += matTransform->_42; // Translate y
 vecSphere.z  += matTransform->_43; // Translate z
 }
```

После того как вы получили вектор, представляющий положение сферы, вычислите вектор, который равняется длине от точки до центра сферы.

```
// Вычислить вектор расстояния
vecDist = vecSphere - ClothPoints[i].m_vecPos;
```

Теперь вы можете сравнить длину этого вектора с радиусом сферы. Чтобы избежать использования sqrt при вычислении расстояний, просто сравните квадраты значений.

```
// Получить квадрат расстояния разности
float Length = vecDist.x * vecDist.x + \
      vecDist.y * vecDist.y + \
      vecDist.z * vecDist.z;

// Проверить, меньше ли длина разницы радиуса
if(Length <= (pObject->m_Radius*pObject->m_Radius)) {

// Произошло столкновение!
```

После того как вы определили, что произошло столкновение (расстояние между точкой и сферой меньше или равно радиусу сферы), вы можете обрабатывать столкновение. Для упрощения, я собираюсь отбросить физику и просто вытолкнуть точку из сферы. Также полагаем, что такая мягкая вещь, как кусочек одежды, не отскочит определенным образом от объекта столкновения, поэтому я также пропущу все вычисления коэффициентов возврата.

Чтобы вытолкнуть точку из сферы, необходимо масштабировать нормализованный вектор расстояния (вектор, представляющий расстояние от точки до центра сферы) на разность расстояний между точкой и краем сферы, а потом вычесть этот вектор из положения точки (и скорости, чтобы замедлить точку).

```
// Нормализовать значение расстояние и вектор
Length = (float)sqrt(Length);
vecDist /= Length;

// Вычислить разность расстояний точки и границы сферы
float Diff = pObject->m_Radius - Length;

// масштабировать вектор на эту разность
vecDist *= Diff;

// Вытолкнуть точку и скорректировать скорость
ClothPoints[i].m_Pos -= vecDist;
ClothPoints[i].m_Velocity -= vecDist;
}
}
```

Вот и все, что касается проверки столкновений точки и сферы! Далее следует проверка столкновения точки и плоскости.

```
// Проверить, сталкивается ли точка с объектом плоскостью
if(pObject->m_Type == COLLISION_PLANE) {
```

Чтобы проверить пересекается ли точка с плоскостью, необходимо сначала преобразовать плоскость, после чего взять скалярное произведение ее нормали и положения точки. Это скалярное произведение, скомбинированное с компонентой смещения плоскости, определяет, расположена ли точка перед плоскостью (столкновения нет) или за плоскостью (столкновение есть). Точки, находящиеся за плоскостью должны быть вытолкнуты из нее.

Чтобы преобразовать плоскость, необходимо сначала обратить и транспонировать матрицу преобразования. Это необходимо сделать только один раз в функции, потому что вы можете использовать эту матрицу для всех плоскостей. В демонстрационной программе этой главы показано, как создать преобразование только однажды. А пока, я буду вычислять его каждый раз. Используя обратную транспонированную матрицу, вы можете вызвать D3DXPlaneTransform для преобразования плоскости.

```
// Сохранить плоскость в локальной переменной
D3DXPLANE Plane = pObject->m_Plane;

// Преобразовать плоскость, если необходимо
if(matTransform) {

  // Обратить и транспонировать матрицу преобразования
  D3DXMATRIX matITTransform;
  D3DXMatrixInverse(&matITTransform, NULL, matTransform);
  D3DXMatrixTranspose(&matITTransform, &matITTransform);

  // Преобразовать плоскость
  D3DXPlaneTransform(&Plane,  &Plane, &matITTransform);
 }
```

После того как вы преобразовали плоскость, необходимо взять ее нормаль и использовать ее для вычисления скалярного произведения с вектором положения точки.

```
// Получить вектор нормали
D3DXVECTOR3 vecNormal = D3DXVECTOR3(Plane.a, \
    Plane.b, \
    Plane.c);

// Посчитать скалярное произведение нормали плоскости и
// положения точки
float Dot = D3DXVec3Dot(&ClothPoints[i].m_vecPos, \
      &vecNormal) + Plane.d;
```

Вы заметите, что я добавил компоненту смещения плоскости (d) к результату скалярного произведения. Это гарантирует корректное вычисление расстояния от точки до плоскости. Если результирующее значение векторного произведения

меньше 0, то точка сталкивается с плоскостью и должна быть вытолкнута. Чтобы вычислить вектор, который необходимо прибавить к векторам положения и скорости точки, необходимо просто умножить нормаль плоскости на значение скалярного произведения и прибавить результат к векторам положения и скорости.

```
// Проверить, находится ли точка за плоскостью
if(Dot < 0.0f)  {

 // Масштабировать нормаль плоскости на модуль скалярного произведения
 vecNormal *=  (-Dot);

 // Переместить точку и скорректировать скорость на вектор нормали
 ClothPoints[i].m_vecPos += vecNormal;
 ClothPoints[i].m_vecVelocity += vecNormal;
 }
}
```

После этого вы можете переходить к обработке следующего объекта столкновения и закончить цикл, который бы просматривал все остальные точки одежды.

```
  // Перейти к следующему объекту столкновения
  pObject = pObject->m_Next;
 }
}
}
}
```

Чтобы использовать функцию CheckCollisions, загрузите меш одежды и начните имитацию. После того как вы обработали силы одежды и обновили ее точки, вызовите CheckCollisions. В качестве примера приведем небольшой кусочек кода:

```
// Создать экземпляры объекта столкновений и матрицы преобразования
cCollision Collision;
D3DXMATRIX matCollision;

// Добавить в список столкновений сферу и установить матрицу в
единичную
Collision.AddSphere(&D3DXVECTOR3(0.0f,   0.0f, 0.0f), 40.0f);
D3DXMatrixIdentity(&matCollision);

// Обработать силы меша одежды и обновить ее точки

// Обработать столкновения
CheckCollisions(&Collision, &matCollision);
```

Хотя методы, которые я только что показал, и не являются самыми точными для определения столкновений объектов и реакции на них, они в целом должны подойти для использования в игровых проектах. Тем из вас, кому необходима абсолютная точность, как например, значение точного момента времени столкновения точки

и объекта столкновения, следует рассмотреть такие методы, как back-stepping time or time-stepping. Я рассмотрел time-stepping в главе 7, так что вы можете использовать те же самые технологии для моделирования одежды.

А пока я хочу показать вам, как использовать полученные вами значения для создания полностью законченного класса меша одежды, который бы управлял моделированием.

Создание класса меша одежды

Я уверен, вам не терпится добавить имитацию одежды в ваши проекты, после того как вы увидели ее в действии (пускали слюни от результата). Используя знания, полученные в этой главе, вы можете очень легко создать класс, который бы управлял одним мешем одежды. Я взял на себя смелость создать такой класс или даже набор классов, которые вы можете использовать в собственных проектах.

Для класса меша одежды будут использованы три класса:

- cClothPoint, который содержит информацию о каждой точке одежды, содержащейся в меше;
- cClothSpring, который содержит информацию о пружинах, включая информацию о соединенных вершинах, длины покоя каждой пружины, значения жесткости и амортизации;
- cClothMesh, который содержит рдин меш, массив точек одежды, список пружин и массив, содержащий индексы всех граней меша. Также в нем содержаться функции для загрузки меша, добавления пружины, установки масс точек, приложения сил и воссоздания меша.

Замечание. Вы *можете найти классы меша одежды на компакт-диске книги. (Посмотрите раздел "Программы на компакт диске" в конце это главы для получения дополнительной информации.) Чтобы их использовать, просто вставьте соответствующий исходный файл в проект.*

Единственный класс, который вы будете непосредственно использовать - это cClothMesh, который использует классы CClothPoint и cClothSpring для хранения данных точек и пружин одежды соответственно. Давайте посмотрим на объявления классов cClothPoint и cClothSpring.

```
// Класс, содержащий информации о точках одежды
class cClothPoint
{

  public:
   D3DXVECTOR3 M_vecOriginalPos; // Исходное положение точки
   D3DXVECTOR3 m_vecPos; // Текущее положение точки
```

```
    D3DXVECTOR3 m_vecForce; // Сила, приложенная к точке
    D3DXVECTOR3 m_vecVelocity; // Скорость точки
    float m_Mass; // Масса объекта (0=прикрепленная)
    float m_OneOverMass; // 1/Массу
};

// Класс, содержащий информацию о пружинах
class cClothSpring
{

  public:
    DWORD m_Point1; // Первая точка пружины
    DWORD m_Point2; // Вторая точка пружины
    float m_RestingLength; // Длина покоя пружины

    float m_Ks; // Значение константы пружины
    float m_Kd; // Значение амортизации пружины

    cClothSpring *m_Next; // Следующая пружина в связанном списке
  public:

    cClothSpring() { m_Next = NULL; }
    ~cClothSpring() { delete m_Next; m_Next = NULL; }
};
```

Как вы можете видеть, классы cClothPoint и cClothSpring используют те же данные, что и структуры cClothPoint и cClothSpring, о которых вы читали ранее в этой главе, так что здесь нет ничего нового. На самом деле, код этих двух классов распределен в этой главе, так что я пропущу его и перейду к классу cClothMesh.

Класс cClothMesh объявлен так:

```
// Класс, полностью содержащий меш одежды (с встроенным анализатором .X)
class cClothMesh : public cXParser
{

  protected:
    DWORD m_NumPoints; // # точек в одежде
    cClothPoint *m_Points; // точки

    DWORD m_NumSprings; // # пружин в одежде
    cClothSpring *m_Springs; // Пружины
```

Пока что меш содержит массив точек и корневую пружину связанного списка пружин. Далее определяется оставшаяся информация, такая как количество граней меша, массив их индексов и размер вершины (в байтах).

```
DWORD m_NumFaces; // # граней в меше
DWORD *m_Faces; // грани

DWORD m_VertexStride; // Размер вершины
```

За защищенными членами-данными следует одна защищенная функция, которая используется для анализа объектов данных файла .X и загрузки соответствующих данных масс точек и пружин. Функциональность анализатора .X используется для обработки данных масс точек и пружин, как было показано в разделе "Создание анализатора .X данных одежды".

```
protected:
  // Анализировать .X файл в поисках данных масс и пружин
  BOOL ParseObject(IDirectXFileData *pDataObj,
       IDirectXFileData *pParentDataObj,
       DWORD Depth,
       void **Data, BOOL Reference);
```

Все оставшиеся функции являются открытыми! В начале списка открытых функций находятся типичные конструктор и деструктор, которые используются для установки данных классов, и функции Create и Free.

```
public:
  cClothMesh();
  ~cClothMesh();

  // Создать одежду из предоставленного указателя меша
  BOOL Create(ID3DXMesh *Mesh, char *PointSpringXFile = NULL);

  // Освободить данные одежды
  void  Free();
```

Используя функцию Create вы можете преобразовать исходный меш и дополнительно предоставляемый файл .X, содержащий данные масс точек и пружин, в данные одежды. Функция Free предназначена для освобождения ресурсов, выделенных для хранения данных одежды; она должны вызываться только при окончании работы с классом меша одежды.

Далее в списке открытых функций располагаются те, которые позволяют вам задавать силы, действующие на точки одежды при моделировании, обновлять точки на основе этих сил и обрабатывать столкновения.

```
// Установить силы, действующие на точки
void SetForces(float LinearDamping,
     D3DXVECTOR3 *vecGravity,
     D3DXVECTOR3 *vecWind,
     D3DXMATRIX *matTransform,
     BOOL TransformAllPoints);

// Обработать силы
void ProcessForces(float Elapsed);

// Обработать столкновения
void ProcessCollisions(cCollision *Collision,
     D3DXMATRIX *matTransform);
```

Вы уже видели код функции ProcessCollisions; я же хочу показать вам две другие функции. Ну, на самом деле, я все еще хочу показать вам объявление класса, так что я вернусь к этим функциям немного поже. Следующие две функции класса воссоздают меш, после того как вы обработали точки одежды (включая вычисление нормалей меша, если это необходимо, при помощи функции D3DXComputeNormals), и восстанавливают точки одежды в их исходные положения.

```
// Воссоздать меш одежды
void RebuildMesh(ID3DXMesh *Mesh);

// Сбросить точки к их исходному положению, а также сбросить силы
void Reset();
```

Вы не должны рассчитывать, что файл .X будет хранить данные пружин или масс точек - имеется пара функций, которая позволяет добавлять пружины в список пружин и устанавливать массу заданной точки.

```
// Добавить пружину в список
void AddSpring(DWORD Point1, DWORD Point2,
     float Ks = 8.0f, float Kd = 0.5f);

// Установить массу точки
void SetMass(DWORD Point, float Mass);
```

В целях экономии памяти и исключения повторений пружин меша одежды функция AddSpring проверяет существующий список на наличие добавляемой пружины. Если найдено совпадение с добавляемой пружиной, то она будет проигнорирована. Функция SetMass принимает в качестве параметра индекс изменяемой точки и новое значение массы. Она также вычисляет новое значение 1/масса.

В завершение, в классе содержаться функции, которые позволяют вам получить количество точек, пружин, граней и указатели на массивы точек, пружин и граней меша. Вы можете использовать эти функции для увеличения полезности класса меша одежды позднее.

```
// Функции получения данных точек/пружин/граней
DWORD GetNumPoints();
cClothPoint *GetPoints();

DWORD GetNumSprings();
cClothSpring *GetSprings();

DWORD GetNumFaces();
DWORD *GetFaces();
};
```

После того как мы объявили класс cClothMesh, пришло время посмотреть на код каждой из его функций. Вы уже видели код ParseObject и ProcessCollisions, поэтому я не буду приводить его здесь. Начав сначала, я приведу код трех важных функций: Create, SetForces и ProcessForces. (Код остальных функций вы можете посмотреть на компакт-диске.)

Функция Create используется для создания меша одежды из указанного объекта ID3DXMesh.

```
BOOL cClothMesh::Create(ID3DXMesh *Mesh, char *PointSpringXFile)
{
  DWORD i;

  // Освободить предыдущий меш
  Free();

  // Проверка ошибки
  if(!Mesh)
   return FALSE;

  // Вычислить шаг вершины (размер ее данных)
  m_VertexStride = D3DXGetFVFVertexSize(Mesh->GetFVF());


  ////////////////////////////////////////////////////
  // Вычислить информацию пружин из загруженного меша
  ////////////////////////////////////////////////////

  // Получить количество граней и создать массив
  m_NumFaces = Mesh->GetNumFaces();
  m_Faces = new DWORD[m_NumFaces*3];

  // Заблокировать буфер индексов и скопировать данные (16-битные
индексы)
  unsigned short *Indices;
  Mesh->LockIndexBuffer(0, (void**)&Indices);
  for(i=0;i<m_NumFaces*3;i++)
   m_Faces[i] = (DWORD)*Indices++;
  Mesh->UnlockIndexBuffer();

  // Получить количество точек меша и создать структуры
  m_NumPoints = Mesh->GetNumVertices();
  m_Points = new cClothPoint[m_NumPoints]();

  // Заблокировать буфер вершин и поместить данные в точки одежды
  char *Vertices;
  Mesh->LockVertexBuffer(0, (void**)&Vertices);
  for(i=0;i<m_NumPoints;i++) {

   // Получить указатель на координаты вершин
   sClothVertexPos *Vertex = (sClothVertexPos*)Vertices;
```

```
        // Сохранить положение, скорость, силу и массу
        m_Points[i].m_vecOriginalPos = Vertex->vecPos;
        m_Points[i].m_vecPos = m_Points[i].m_vecOriginalPos;
        m_Points[i].m_Mass = 1.0f;
        m_Points[i].m_OneOverMass = 1.0f;

        // Установить состояния точек
        m_Points[i].m_vecVelocity = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
        m_Points[i].m_vecForce = D3DXVECTOR3(0.0f, 0.0f, 0.0f);

        // Перейти к следующей вершине
        Vertices += m_VertexStride;
      }
      Mesh->UnlockVertexBuffer();

      // Создать список пружин из вершин граней
      for(i=0;i<m_NumFaces;i++) {

        // Получить вершины, образующие грань
        DWORD Vertex1 = m_Faces[i*3];
        DWORD Vertex2 = m_Faces[i*3+1];
        DWORD Vertex3 - m_Faces[i*3+2];

        // Добавить пружины из 1->2, 2->3 и 3->1
        AddSpring(Vertex1, Vertex2);
        AddSpring(Vertex2, Vertex3);
        AddSpring(Vertex3, Vertex1);
      }

      // Получить массы одежды и пружины из файла
      if(PointSpringXFile)
        Parse(PointSpringXFile);

      return TRUE;
    }
```

Функция Create начинается с освобождения соответствующих данных при помощи вызова Free. Далее происходит вызов D3DXGetFVFVertexSize, необходимый для определения размера данных одной вершины, после чего определяется количество граней и создается массив, содержащий индексы. Далее индексы граней считываются в этот массив и создается массив точек одежды.

Заблокировав буфер вершин, вы копируете координаты вершин в соответствующие точки одежды, при этом устанавливая значения их масс, скоростей, сил и 1/масса. После того как вы просмотрели все вершины и создали данные точек одежды, вы заканчиваете созданием пружин и анализом .X файла. (Вы создаете пружину, получая три индекса, образующие грань, и вызывая три раза функцию AddSpring, для соединения трех точек.)

Следующей важной функцией, на которую я бы хотел обратить внимание, является SetForces, которая просматривает все точки одежды и устанавливает соответствующие силы, подготавливая их к интегрированию.

```
void cClothMesh::SetForces(float LinearDamping,
      D3DXVECTOR3 *vecGravity,
      D3DXVECTOR3 *vecWind,
      D3DXMATRIX *matTransform,
      BOOL TransformAllPoints)
{
  DWORD i;

  // Проверка ошибок
  if(!m_NumPoints || m_Points == NULL)
   return;
```

Функция SetForces начинается с прототипа и нескольких строчек кода, выполняющих проверку для убеждения, что были заданы точки одежды. В качестве параметров функции SetForces передаются значения линейной амортизации (установленное в отрицательное значение -0.05f), векторы, представляющие направление и величину сил гравитации и ветра, матрица преобразования, применяемая к точкам, и флаг, определяющий, какая точка преобразуется.

Вы уже читали о линейной амортизации и векторах сил (гравитации и ветра), но я еще ничего не упоминал о преобразовании координат точек. В чем была бы польза меша одежды, если бы он не мог перемещаться по трехмерному миру? Передав матрицу преобразования SetForces, вы можете преобразовать точки перед вычислением сил.

Вы спросите, а какие точки преобразовывать? Это зависит от флага TransformAllPoints. Если вы установите TransformAllPoints в TRUE, что вы и должны сделать самый первый раз для расположения и ориентации точек меша одежды в трехмерном мире, тогда все точки используют матрицу преобразования. Установка TransformAllPoints в FALSE означает, что матрица преобразования влияет только на точки с нулевой массой (т. е. точки с ненулевой массой будут двигаться в соответствии с действующими силами, догоняя преобразованные точки).

Функцию SetForces можно разбить на три основных куска. Первый кусок подготавливает точки, очищая их векторы сил, преобразуя точки на основе заданной матрицы преобразования, применяя силу тяжести и линейной амортизации.

```
// Очистить силы, применить преобразование, установить гравитацию и
// применить линейную амортизацию
for(i=0;i<m_NumPoints;i++) {

  // Очистить силы
  m_Points[i].m_vecForce = D3DXVECTOR3(0.0f, 0.0f, 0.0f);
```

```
  // Переместить точки, используя преобразование, если задано
  if(matTransform && (TransformAllPoints == TRUE || \
   m_Points[i].m_Mass == 0.0f)) {

  D3DXVec3TransformCoord(&m_Points[i].m_vecPos, \
          &m_Points[i].m_vecOriginalPos, \
           matTransform);
  }

  // Применять гравитацию и линейную амортизацию только к точкам
с ненулевой
  // массой
  if(m_Points[i].m_Mass != 0.0f) {

   // Применить гравитацию, если задана
   if(vecGravity != NULL)
    m_Points[i].m_vecForce += (*vecGravity) * \
                m_Points[i].m_Mass;

   // Применить линейную амортизацию
   m_Points[i].m_vecForce += (m_Points[i].m_vecVelocity * \
                LinearDamping);
  }
 }
```

Ранее в этой главе я уже объяснял, как применять силы линейной амортизации и гравитации, так что я не буду опять загружать вас деталями. Как я упоминал, есть код, который преобразует точки одежды, используя заданную матрицу преобразования. Следующий кусок кода применяет силы ветра к точкам одежды, используя технологии, показанные ранее в этой главе:

```
// Применить ветер
if(vecWind != NULL && m_NumFaces) {

  // Просмотреть все грани и применить ветер ко всем их  вершинам
  for(i=0;i<m_NumFaces;i++) {

   // Получить три вершины, образующие грань
   DWORD Vertex1 = m_Faces[i*3];
   DWORD Vertex2 - m_Faces[i*3+1];
   DWORD Vertex3 = m_Faces[i*3+2];

   // Вычислить нормаль грани
   D3DXVECTOR3 vecV12 = m_Points[Vertex2].m_vecPos - \
   m_Points[Vertex1].m_vecPos;
   D3DXVECTOR3 vecV13 = m_Points[Vertex3].m_vecPos - \
   m_Points[Vertex1].m_vecPos;
   D3DXVECTOR3 vecNormal;
   D3DXVec3Cross(&vecNormal, &vecV12, &vecV13);
   D3DXVec3Normalize(&vecNormal, &vecNormal);
```

```
    // Получить скалярное произведение нормали и ветра
    float Dot = D3DXVec3Dot(&vecNormal, vecWind);

    // Увеличить нормаль на величину скалярного произведения
    vecNormal *= Dot;

    // Применить нормаль к вектору силы точки
    m_Points[Vertex1].m_vecForce += vecNormal;
    m_Points[Vertex2].m_vecForce += vecNormal;
    m_Points[Vertex3].m_vecForce += vecNormal;
  }
 }
```

Последний, и самый важный, кусок кода применяет силы пружин.

```
// Обработать пружины
cClothSpring *Spring = m_Springs;
while(Spring) {

  // Получить текущий вектор пружины
  D3DXVECTOR3 vecSpring;
  vecSpring = m_Points[Spring->m_Point2].m_vecPos - \
              m_Points[Spring->m_Point1].m_vecPos;

  // Получить текущую длину пружины
  float SpringLength = D3DXVec3Length(&vecSpring);

  // Получить относительную скорость точек
  D3DXVECTOR3 vecVelocity;
  vecVelocity = m_Points[Spring->m_Point2].m_vecVelocity - \
                m_Points[Spring->m_Point1].m_vecVelocity;
  float Velocity = D3DXVec3Dot(&vecVelocity, &vecSpring) / \
        SpringLength;

  // Вычислить скаляры силы
  float SpringForce = Spring->m_Ks * (SpringLength - \
        Spring->m_RestingLength);
  float DampingForce = Spring->m_Kd * Velocity;

  // Нормализовать пружину
  vecSpring /= SpringLength;

  // Вычислить вектор силы
  D3DXVECTOR3 vecForce = (SpringForce + DampingForce) * \
      vecSpring;

  // Применить силу к векторам
  if(m_Points[Spring->m_Point1].m_Mass != 0.0f)
   m_Points[Spring->m_Point1].m_vecForce += vecForce;

  if(m_Points[Spring->m_Point2].m_Mass != 0.0f)
   m_Points[Spring->m_Point2].m_vecForce -= vecForce;

  // Перейти к следующей пружине
```

```
    Spring = Spring->m_Next;
  }
}
```

Опять же здесь нет ничего, чего бы вы не видели. После того как вы вызвали SetForces необходимо обработать силы и переместить точки одежды в соответствии с накопленными силами.

```
voidcClothMesh::ProcessForces(floatElapsed)
{
  // Проверка ошибок
  if(!m_NumPoints || !m_Points)
   return;

  // Рассчитать силы точек
  for(DWORD i=0;i<m_NumPoints;i++) {

   // Точки с нулевой массой не движутся
   if(m_Points[i].m_Mass != 0.0f) {

    // Обновить скорость
    m_Points[i].m_vecVelocity+= (Elapsed * \
                m_Points[i].m_OneOverMass* \
                m_Points[i].m_vecForce);

    // Обновить положение
    m_Points[i].m_vecPos += (Elapsed *
                m_Points[i].m_vecVelocity);
  }
 }
}
```

Коротко и ясно, функция ProcessForces делает то, что и должна - просматривает список точек одежды и для каждой точки линейно интегрирует скорость и положение на основе сил точки и величины прошедшего времени (которую вы задаете при вызове ProcessForces). Заметьте, что параметр Elapsed представляет собой количество интегрируемых секунд. Если вы хотите задать миллисекунды, необходимо задать дробное значение, как я уже замечал ранее.

Что вы думаете о том, чтобы показать вам, как использовать класс cClothMesh в ваших собственных проектах? Сначала убедитесь, что имеется корректный объект ID3DXMesh, после чего вызовете функцию Create для создания необходимых данных одежды.

Замечание. *Вы не должны обрабатывать (интегрировать) более 10-20 миллисекунд за один раз. Если вы хотите обработать более 20 миллисекунд за один раз, вы должны сделать несколько вызовов SetForces и ProcessForces, каждый раз задавая небольшой интервал времени. Например, вы можете вызвать ProcessForces, указав в качестве параметра оба раза 20 миллисекунд, чтобы в общем обработать 40 миллисекунд.*

```
// pMesh = предварительно загруженный объект ID3DXMesh
cClothMesh ClothMesh;

// Создать данные одежды, вызвав Create
ClothMesh.Create(pMesh);
```

После того как вы создали данные меша одежды, вы можете использовать объект класса для имитации движения меша одежды в каждом кадре вашей игры. Предположим, у вас имеется функция, которая вызывается при обновлении каждого кадра игры. В этой функции вы можете следить за количеством миллисекунд, прошедших с последнего обновления. Это прошедшее время используется для интегрирования точек одежды. Сразу после установки сил и обновления одежды вы можете воссоздать меш и визуализировать его.

```
void FrameUpdate()
{
  static DWORD LastTime = timeGetTime()-1;
  DWORD ThisTime = timeGetTime();
  DWORD Elapsed;

  // Вычислить прошедшее время
  Elapsed = ThisTime - LastTime;
  LastTime = ThisTime;

  // Установить векторы гравитации и тяжести
  D3DXVECTOR3 vecGravity = D3DXVECTOR3(0.0f, -9.8f, 0.0f);
  D3DXVECTOR3 vecWind = D3DXVECTOR3(0.0f, 0.0f, 1.0f);

  // Установить силы одежды
  ClothMesh.SetForces(-0.05f,&vecGravity,&vecWind,NULL,FALSE);

  //Обработать силы одежды,основываясь на прошедшем времени. Убедитесь,
  //что время задано в миллисекундах,для чего разделите прошедшее время
  //на 1000
  ClothMesh.ProcessForces((float)Elapsed / 1000.0f);

  // Обработать столкновения, если они есть

  // Воссоздать меш
  ClothMesh.RebuildMesh(pMesh);

  // Визуализировать меш одежды любым удобным способом
}
```

Чтобы облегчить использование cClothMesh, вы можете запустить демонстрационную программу с компакт-диска. Она очень подробно откомментирована, и иллюстрирует возможности класса меша одежды. Наслаждайтесь!

Использование мешей мягких тел

Меши мягких тел становятся все более популярными в современных играх. Они позволяют представляться объектам мягкими, эластичными и/или растягивающимися. Объекты могут изгибаться, скручиваться, подрагивать множеством различных способов, после чего принимать исходную форму.

Что могут сделать меши мягких тел в вашей игре? Ну, давайте посмотрим на современные игры, использующие их. Baldur's Gate: Dark Alliance фирмы Interplay использует меши мягких тел при анимировании персонажей, придавая женским персонажам дополнительные...м... подрагивания при ходьбе.

Меши мягких тел также могут быть использованы для моделирования волос персонажа. Если волосы персонажа взметнулись вверх, со временем они восстанавливают начальную форму. Использование мешей мягких тел также может быть полезно в картах и уровнях игры. Представьте уровень игры, который изгибается и закручивается в соответствии с действиями персонажа, как если бы персонаж шел по горячему воздушному шару. Привлекательно!

Я знаю, что вам интересно, почему я поместил моделирование одежды и анимацию мешей мягких тел в одну главу. Ответ прост - меши мягких тел являются мешами одежды! Да, вы меня правильно расслышали - меши мягких тел практически идентичны мешам одежды, за единственным исключением. Меши мягких тел всегда восстанавливают начальную форму со временем, вместо того, чтобы висеть подобно частям ткани.

Восстановление мешей мягких тел

В отличие от своего двоюродного брата, меша одежды, меш мягкого тела медленно восстанавливает начальную форму со временем. Если вы подумаете об этом, то поймете, что восстановление меша мягкого тела настолько просто, насколько необходимо рассчитать силы, нужные, чтобы вернуть вершины меша в начальное положение. Так что вместо того, чтобы рассчитывать пружины меша, необходимо рассчитывать его точки.

В отличие от сил, создаваемых пружинами одежды, меш мягкого тела напрямую изменяет положение каждой точки и векторы скорости. Просмотрев все точки меша, вы можете вычислить вектор пружины, который представляет расстояние от текущего положения точки до начального.

Помните, мы сохраняли начальное положение каждой точки? Применив константу пружины к расстоянию между начальным положением точки и конечным, можно определить силу, которую необходимо прибавить к векторам положения и скорости каждой точки.

Без дальнейших разговоров перейдем к коду, вычисляющему силу, необходимую для перемещения каждой точки в ее начальное положение:

```
// SpringStiffness = вещественное значение, содержащее жесткость,
// необходимую для возвращения точек в начальное положение
for(DWORD i=0;i<NumPoints;i++) {
  // получить вектор разности между текущим и начальным положениями
  D3DXVECTOR3 vecDiff = ClothPoints[i].m_vecInitialPos - \
  ClothPoints[i].m_vecPos;

  // Применить значение жесткости к вектору пружины
  vecDiff *= SpringStiffness;

  // Применить вектор пружины к векторам скорости и положения
  ClothPoints[i].m_vecPos += vecDiff;
  ClothPoints[i].vecVelocity += vecDiff;
```

Быстро и легко предыдущий код вычисляет вектор разности между текущим и начальным положениями точки одежды. Этот вектор разности потом масштабируется на заданное значение жесткости пружины. В отличие от пружин точек, показанных ранее, значение жесткости пружины мягкого тела должно оставаться маленьким, скажем от 0.05 до 0.4. Величина жесткости зависит от величины времени, заданного в качестве параметра функции базового класса UpdateForces. Лично я использую значение жесткости 0.2 для каждых обрабатываемых 20-30 миллисекунд.

Т. к. мы говорим об использовании одного и того же кода для моделирования мешей мягких тел, что и для моделирования одежды, вы можете добавить предыдущий код, который возвращает точки меша в исходное положение, в функцию, которая рассчитывает силы одежды и передвигает точки.

Создание класса меша мягкого тела

Класс меша мягкого тела до мельчайших подробностей идентичен классу меша одежды, за исключением того, что в классе меша мягкого тела имеется дополнительная функция, которая восстанавливает исходную форму меша. Т. к. вы просто добавляете одну функцию, вы можете пропустить большую часть объявления класса мягкого тела и сразу перейти к функции, восстанавливающей меш мягкого тела.

```
// Создать производный класс мягкого тела
class cSoftbodyMesh : public cClothMesh


  public:
   void Revert(float Stiffness, D3DXMATRIX *matTransform)
   {
```

```
        // Проверка ошибок
        if(!m_NumPoints || m_Points == NULL)
         return;

        // Обработать силы мягкого тела (восстанавливающие форму)
        for(DWORD i=0;i<m_NumPoints;i++) {

         // Обрабатывать только те точки, которые могут двигаться
         if(m_Points[i].m_Mass != 0.0f) {

          // Преобразовать исходные координаты, если необходимо
          D3DXVECTOR3 vecPos = m_Points[i].m_vecOriginalPos;
          if(matTransform)
           D3DXVec3TransformCoord(&vecPos, &vecPos, matTransform);

          // Создать вектор пружины из исходного положения точки
(преобразованного)
          // до ее текущего положения
          D3DXVECTOR3 vecSpring = vecPos - m_Points[i].m_vecPos;

          // Масштабировать пружину на значение жесткости
          vecSpring *= Stiffness;

          // Непосредственно изменить скорость и положение
          m_Points[i].m_vecVelocity += vecSpring;
          m_Points[i].m_vecPos += vecSpring;
         }
        }
       }
};
```

Как вы можете видеть из объявления класса cSoftbodyMesh, функция Revert просматривает все точки одежды, передвигает их к начальному положению и соответствующим образом корректирует скорость точки. Вы заметите, что значение жесткости, матрицу преобразования, применяемую к начальному положению точек до вычисления вектора пружины, можно задать при вызове Revert. Это позволяет точкам занимать корректные положения, а мешу - двигаться по миру.

Используя те же самые технологии, что и в классе меша одежды, вы можете работать с классом меша мягкого тела. Загрузка, установка сил, визуализация - у обоих классов они одинаковы. Единственным различием является то, что вам необходимо вызывать cSoftBodyMesh::Revert сразу после вызова cSoftBodyMesh::Resolve. Посмотрите демонстрационную программу меша мягкого тела, находящуюся на компакт-диске, чтобы увидеть, как использовать класс cSoftBodyMesh в ваших проектах.

Посмотрите демонстрационные программы

Фу, это была длинная глава, но безусловно, содержащаяся в ней информация стоила прочтения! После того, как вы увидели, как легко работать с моделированием одежды, я уверен, вам не терпится применить эти полученные знания в ваших проектах игр. Чтобы помочь вам понять, как использовать меши одежды и меши мягких тел в ваших проектах, я включил две демонстрационные программы в эту книгу.

Первый проект (ClothMesh), показанный на рис. 13.8, иллюстрирует один из примеров использования одежды - развевающийся плащ супергероя.

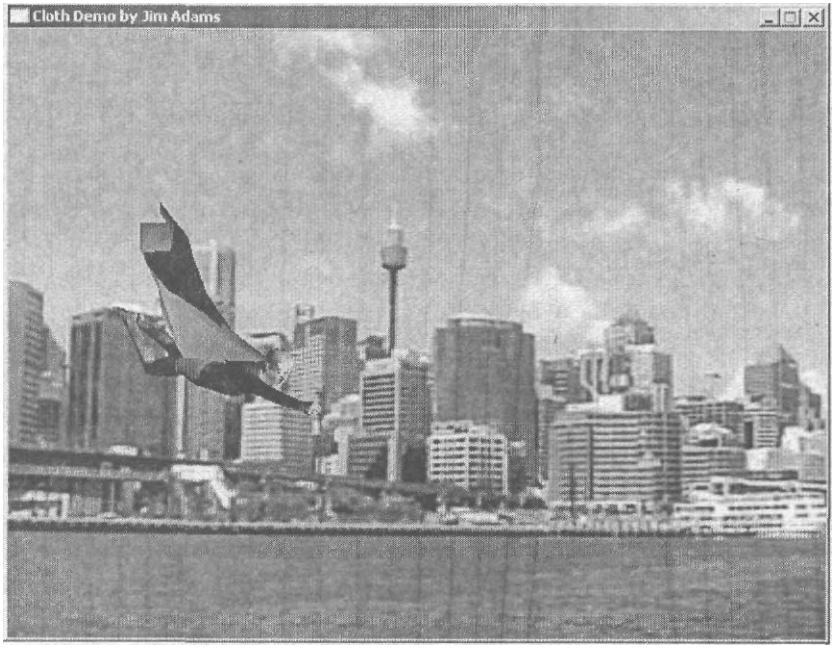


Рис. 13.8. Супергерой летит по воздуху, а его плащ развевается у него за спиной

Второй проект, поставляемый с книгой, Softbody, иллюстрирует использования мешей мягких тел для придания персонажам дополнительного bounce при ходьбе. Посмотрите демонстрационную программу, о которой я говорю, или рис. 13.9.

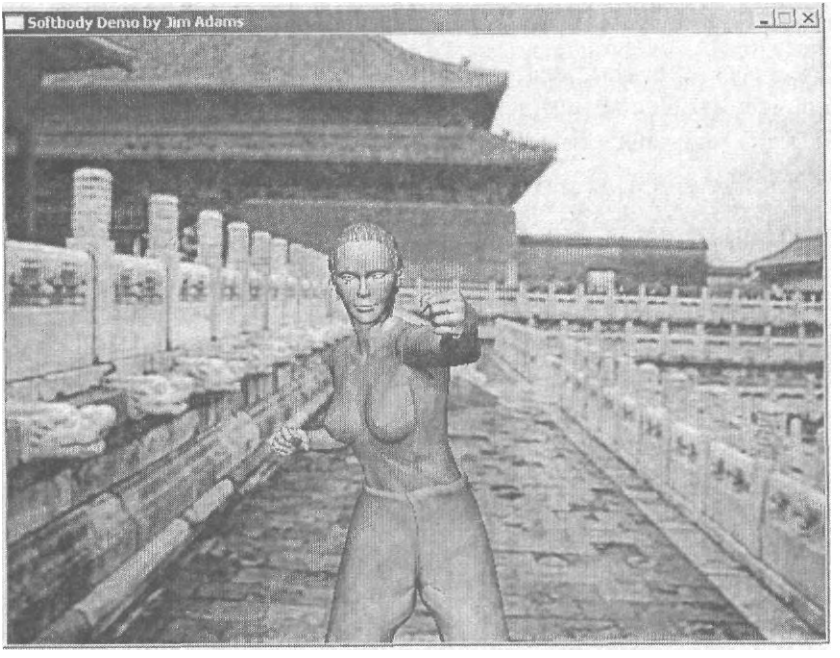


Рис. 13.9. Урок карате! Меш мягкого тела придает дополнительные подрагивания при атаках мастера

Программы на компакт-диске

Классы меша мягкого тела и меша одежды этой главы находятся в двух файлах проекта, расположенных в директории главы 13 компакт-диска этой книги. Этими проектами являются:

- **ClothMesh.** Эта программа иллюстрирует использование класса меша одежды, разработанного в этой главе. Она расположена в \BookCode\Chap13\ClothMesh.
- **SoftBody.** Эта демонстрационная программа иллюстрирует использование класса меша мягкого тела в ваших проектах. Она расположена в \BookCode\Chap13\SoftBody.

Глава 14

Использование анимированных текстур

Трехмерная анимация включает в себя управление вершинами, полигонами и мешами. По правде говоря, ваша законченная система анимации может выполнять все эти анимации, но вы не используете одну из самых замечательных технологий, имеющихся на сегодняшний день - анимацию текстур.

Вы даже не представляете себе, насколько может стать живее мир, если использовать анимацию текстур. Используя разнообразные методы, от простого преобразования текстур до современных технологий проигрывания медиа файлов, вы можете создать привлекательные эффекты, такие как динамический задний фон уровня, текущая вода и внутриигровые видеовставки.

В этой главе вы научитесь:

- Использовать преобразования для изменения текстур;
- Работать с DirectShow;
- Создавать фильтры DirectShow;
- Накладывать видео образцы на текстуры.

Использование анимации текстур в ваших проектах

Обычная анимация включает манипулирование вершинами трехмерного меша. Большей частью этих манипуляций над вершинами достаточно для игры. А как насчет анимации поверхности полигона? Возможно, вы захотите изменить внешний вид полигона со временем, проиграть видеофайл на поверхности полигона или просто плавно переместить текстуру по поверхности полигона. Я говорю о том, чтобы заставить ваши полигоны танцевать! Я говорю об анимации текстур.

При использовании анимации текстур вы работаете на уровне полигонов (изменяя источник изображения или координаты текстур, используемые многоугольником), а не с вершинами, как при обычной анимации. Для этой книги я выбрал две наиболее популярные на сегодняшний день технологии анимации: преобразование текстур и видео анимация текстур.

Давайте начнем с самой простой технологии и посмотрим, как можно использовать преобразования текстур в игровом проекте.

Работа с преобразованиями текстур

Одним из самых недооцениваемых (и, наверное, самых простых) способов анимации текстур является их преобразование. Совсем как преобразование мира изменяет координаты вершин модели перед ее визуализацией, преобразование текстур изменяет данные текстурных координат модели перед ее визуализацией.

Какая может быть польза от преобразований, текстур? Предположим, необходимо анимировать несколько полигонов. Например, вы хотите создать воду, падающую в виде водопада и медленно текущую по руслу. Возможно ли добиться такого эффекта, используя обычные преобразования вершин? Возможно, но это будет очень не просто, я уверяю вас.

Используя анимацию текстур, вы можете плавно перемещать текстуру воды по полигонам, создавая таким образом водопад и поток. Не изменяя меша, вы оживляете вашу сцену! Однако вы можете не ограничиваться перемещением текстур - вы можете также вращать их. Представьте текстуру воды, движущуюся и вращающуюся одновременно! Секрет использования преобразования текстур заключается в… преобразованиях.

Создание преобразования текстур

В отличие от трехмерных эквивалентов, преобразования текстур являются двухмерными и используют матрицы преобразования 3x3. Больше нет нужды перемещать по оси z и вращать по осям x и y. Остается перемещение по осям x/y (при этом ось y перевернута и направлена вниз) и вращение по оси z. Не беспокойтесь, потому что этого будет достаточно!

Самое простое преобразование текстуры, которое может быть выполнено, - это перемещение. Перемещения текстур в Direct3D задаются значениями, находящимися в диапазоне от 0 до 1, совсем как в текстурных координатах. Например, если ваша текстура имеет размер 256 пикселей, и вы хотите переместить ее на 64 пикселя вправо, задайте значение перемещения равным 0.25 (четверть размера текстуры, или 64 пикселя).

По умолчанию текстуры облегают многоугольник при любом преобразовании. Это означает, что если переместить текстуру на 64 пикселя вправо, последние 64 колонки пикселей переместятся в левую сторону текстуры. Это облегание текстур нас полностью устраивает, поэтому будем его использовать.

Замечание. *DirectX проверяет все значения, задаваемые к качестве перемещения текстур, чтобы они попадали в диапазон от 0 до 1, так что в этой части от вас не требуется никаких действий. Таким образом, вы можете все время увеличивать или уменьшать значения перемещения и быть уверенными, что они всегда будут лежать в диапазоне от 0 до 1.*

Для упрощения вы можете использовать объекты матриц D3DX для создания преобразований текстур. Я знаю, я сказал, что преобразования текстур используют матрицу 3x3, но если вы ограничитесь использованием перемещения по осям x/y и вращением по оси z, вы можете с небольшими изменениями использовать матрицы D3DX 4x4, как показано в следующем разделе "Установка матриц преобразования текстур".

Имея возможность использовать матрицу D3DX, вы можете создать матрицу перемещения так:

```
// Прототип функции D3DXMatrixTranslation из DX SDK
D3DXMATRIX *D3DXMatrixTranslation(D3DXMATRIX *pOut,
            FLOAT x, FLOAT y, FLOAT z);

D3DXMATRIX matTranslation;
D3DXMatrixTranslation(&matTranslation,   0.5f,  0.5f, 0.0f);
```

В этом коде приведен прототип D3DXMatrixTranslation, которая будет вами использоваться, и небольшая иллюстрация установки преобразования, перемещающего текстуру влево и вверх на половину ширины и высоты текстуры (в пикселях). Заметьте, что перемещение по z равно 0.0, каким оно и должно быть для всех преобразований текстур.

Вспомните, я говорил, что перемещение зависит от размера текстуры. Хотя задаваемые значения и представляют собой процентные соотношения размеров текстуры, на которые необходимо ее переместить (при этом 0 означает 0%, а 1.0 - 100%), если вам нужна абсолютная точность, необходимо работать с пикселями. Например, если вы применяете преобразования, приведенные выше (значение 0.5, которое означает половину размеров текстуры по осям x и y), для текстуры размером 256x128 она переместиться на 128 пикселей влево и 64 пикселя вправо. Если вы запомните принцип работы значений перемещения, то все будет нормально.

Для вращательных преобразований текстуры вы можете использовать функцию D3DXMatrixRotationZ. Помните, что текстуры могут вращаться только относительно оси z. As long as you keep to the z-axis, вы можете использовать матрицу 4x4. Вот пример использования функции D3DXMatrixRotationZ для вращения текстуры на 1.57 радиан.

```
// Прототип функции D3DXMatrixRotationZ
D3DXMATRIX *D3DXMatrixRotationZ(D3DXMATRIX *pOut, FLOAT Angle);

D3DXMATRIX matRotation;
D3DXMatrixRotationZ(&matRotation, 1.57f);
```

Вращение происходит относительно центра координат текстуры, который находится в ее левом верхнем углу. Если вы хотите повернуть текстуру относительно другой точки, необходимо преобразовать текстуру, повернуть ее и снова преобразовать в начальное положение. Вот пример вращения текстуры относительно ее центра на 0.47 радиан:

```
D3DXMATRIX matTrans1, matTrans2, matRotation;
D3DXMatrixTranslation(&matTrans1,  -0.5f, -0.5f,  0.0f);
D3DXMatrixTranslation(&matTrans2,   0.5f,  0.5f,  0.0f);
D3DXMatrixRotationZ(&matRotation, 0.47f);

// Скомбинировать матрицы в одно преобразование
D3DXMATRIX matTransformation;
matTransformation = matTrans1 * matRotation * matTrans2;
```

Как вы можете видеть из кода примера, возможно комбинировать любое число матриц для получения результирующей матрицы преобразования. После того как вы получили ее, необходимо передать ее Direct3D и визуализировать текстуру.

Установка матриц преобразования текстуры

После того как вы получили корректную матрицу D3DXMATRIX, содержащую все преобразования, вы можете визуализировать преобразованную текстуру. Однако сначала необходимо преобразовать матрицу 4x4 в матрицу 3x3, которая используется Direct3D для преобразования текстур. Это может сделать небольшая функция, как например эта:

```
void Matrix4x4To3x3(D3DXMATRIX *matOut, D3DXMATRIX *matIn)
{
  matOut->_11 = matIn->_11; // Скопировать первую строчку
  matOut->_12 = matIn->_12;
  matOut->_13 = matIn->_13;
  matOut->_14 =  0.0f;
```

```
  matOut->_21 = matIn->_21; // Скопировать вторую строчку
  matOut->_22 = matIn->_22;
  matOut->_23 = matIn->_23;
  matOut->_24 = 0.0f;

  matOut->_31 = matIn->_41; // Скопировать нижнюю строчку,
  matOut->_32 = matIn->_42; // используемую для перемещения
  matOut->_33 = matIn->_43;
  matOut->_34 = 0.0f;

  mat0ut->_41 = 0.0f; // Очистить нижнюю строчку
  matOut->_42 = 0.0f;
  matOut->_43 = 0.0f;
  matOut->_44 = 1.0f;
}
```

Вызов функции Matrix4x4To3x3 происходит так же просто, как и вызов любой функции D3DX для работы с матрицами. Все, что вам необходимо, это предоставить указатель на результирующую матрицу, как я сделал тут. (Заметьте, что исходная и результирующая матрицы могут быть одинаковыми.)

```
// Преобразовать матрицу в матрицу 3x3
Matrix4x4To3x3(&matTexture, &matTexture);
```

После того как вы задали в качестве параметра функции Matrix4x4To3x3 матрицу преобразования текстуры, вы можете установить результирующую матрицу 3x3 преобразования текстуры, используя функцию IDirect3DDevice9::SetTransform, установив флаг D3DTS_TEXTURE0.

```
// Установить преобразование в конвейер Direct3D
pDevice->SetTransform(D3DTS_TEXTURE0, &matTexture);
```

На данный момент Direct3D почти готов к преобразованию текстур! Единственное, что осталось сделать, это сообщить Direct3D, что при вычислении преобразований используются двухмерные координаты текстуры. Вы можете сделать это следующим вызовом IDirect3DDevice9::SetTextureStageState:

```
pDevice->SetTextureStageState(0,
         D3DTSS_TEXTURETRANSFORMFLAGS,
         D3DTTFF_COUNT2);
```

Вот теперь все готово! С этого момента к каждой визуализируемой текстуре (нулевого уровня) будет применяться преобразование. После того как вы обработаете все текстуры, вы можете отключить преобразования, вызвав SetTextureStageState, но на этот раз установив D3DTSS_TEXTURETRANSFORMFLAGS в D3DTTFF_DISABLE.

```
pDevice->SetTextureStageState(0,
         D3DTSS_TEXTURETRANSFORMFLAGS,
         D3DTTFF_DISABLE);
```

Замечание. *Показанный здесь код позволяет использовать преобразования текстур только в нулевом уровне. Если вы используете текстуры другого уровня, просто замените 0 на используемый уровень.*

Использование преобразования текстур в проектах

Как вы можете видеть, использовать преобразования текстурных координат очень просто. Самым сложным является отслеживание разнообразных преобразований. Я рекомендую вам создать какой-нибудь менеджер, следящий за всеми преобразованиями текстур. (Возможно, вам подойдет использование массива перемещений и вращений.)

В качестве примера использования преобразований текстур в ваших проектах вы можете посмотреть демонстрационную программу преобразования текстур, расположенную на компакт-диске. (Ищите ее в директории главы 14 или посмотрите конец этой главы для получения дополнительной информации.) В демонстрационной программе показано, как создать эффект текущего водопада с помощью простых перемещений. Пусть эта программа послужит отправной точкой в мир анимации преобразования текстур!

Использование файлов видео в качестве текстур

Хотя текстуры и улучшают внешний вид трехмерной графики, они являются статичными картинками. Даже использование преобразований текстур не может этого изменить. Вам необходима возможность накладывать видео файлы на поверхности полигонов.

Все правильно! Вы действительно можете проигрывать видео файлы на поверхности трехмерных моделей, что позволяет достичь невероятных результатов! Представьте, вы можете записать что-нибудь, а потом проиграть это в вашей игре. Например, для спортивной игры может быть использована толпа фанатов, болеющих за любимую команду. Вы можете нарисовать это живое действие на поверхности пустых сидений, заполнив таким образом стадион тысячами восхищающихся болельщиков. Поговорим о реализме!

Однако не ограничивайтесь использованием видео текстур. Используя эту технологию, вы можете сделать гораздо больше. Как насчет внутриигровых видеовставках? Просто создайте полигон, размером на весь экран, который имеет в качестве текстуры видео последовательность. В чем хитрость? Ни в чем - вы используете те же самые технологии, которые использовались для текстурирования лицевого меша персонажа!

Если преобразования текстур начали вас раздражать, я уверен, что использование видео текстур вдохновит вас! Как же они работают, спросите вы? Все начинается с DirectShow корпорации Microsoft.

Импорт видео при помощи DirectShow

DirectShow является частью DirectX, которая предназначена для управления воспроизведением видео. Это очень полезный набор компонент, позволяющих работать с множеством разнообразных типов медиа, включая различные аудио форматы, такие как .mp3 и .wma, и видео форматы, такие как mpg, .avi и даже закодированные файлы DVD!

Т. к. нас интересует только часть, относящаяся к воспроизведению видео, вы можете игнорировать большую часть компонент DirectShow и сосредоточиться только на тех, которые работают с видео. Хм - после некоторых размышлений, я понял что самым простым способом работы с видео является создание специализированного фильтра, который бы обрабатывал данные видео, импортируемые DirectShow. Давайте посмотрим, как можно использовать фильтры для анимации текстур.

Замечание. *Те из вас, кто изучал DirectX SDK самостоятельно, видели демонстрационную программу Texture3D, которая иллюстрирует использование видео файла для текстурирования полигонов. В этой главе я расширю функциональность этой программы, разработав способ использования неограниченного количества видео файлов в трехмерных сценах.*

Использование фильтров в DirectShow

DirectShow работает с различными медиа форматами посредством фильтров. Каждый фильтр предназначен для кодирования или декодирования медиа данных при их записи или считывании из файла. Фильтры могут быть объединены, при этом один фильтр ответственен за декодирование одного типа данных медиа, а другой фильтр обрабатывает эти данные и отображает их (см. рис. 14.1).

Вы можете начать с создания собственного фильтра DirectShow и вставки его в поток декодирования видео. Как только вы начнете декодировать видео файл с помощью DirectShow, фильтр обработает и отправит данные этого файла непосредственно на поверхность текстуры, используемой при визуализации полигонов. Звучит просто, не так ли?

Проблема заключается в том, что последние версии DirectShow используют более 70 фильтров и более 70 интерфейсов! Ничего себе - этого достаточно, чтобы заставить поежиться самых опытных программистов! Что такой программист, как вы или я, может сделать с этим огромным количеством интерфейсов? Я скажу, что вы можете сделать...вы можете использовать базовые классы.

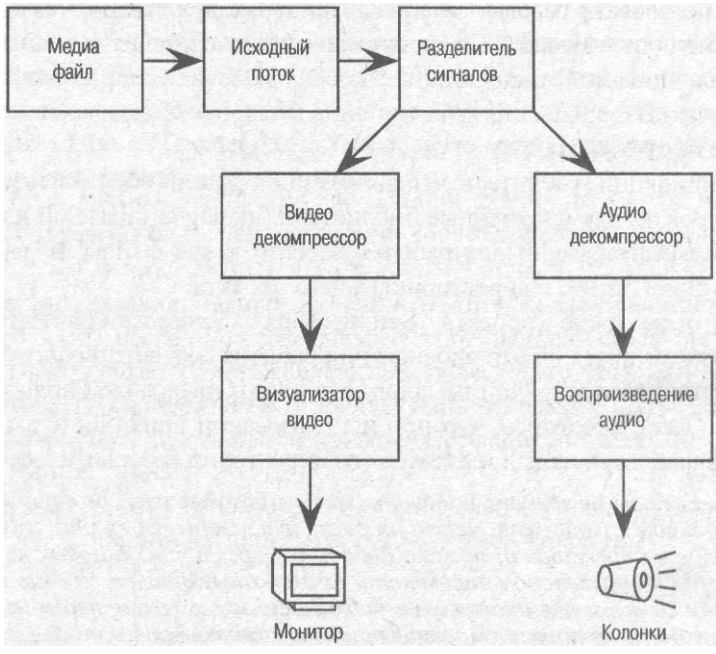


Рис. 14.1. Медиа файл может пройти через множество фильтров, прежде чем будет проигран

Использование базовых классов

Вот здесь то и приходит на помощь корпорация Microsoft! Зная, что непосредственная работа с декодерами видео и разнообразными интерфейсами DirectShow может быть затруднительна, корпорация Microsoft создала набор классов, которые облегчают разработку фильтров. Эти классы расположены в инсталляционной директории DirectX SDK в\Samples\Multimedia\DirectShow\BaseClassesи называют базовые классы DirectShow.

Совет. *Чтобы изменить настройки компилятора Visual C/C++, в главном меню выберите Build (Построить), а затем Set Active Configuration (Установить текущую конфигурацию). В диалоговом окне Set Active Project Configuration (Установка текущей конфигурации проекта) выберите желаемую конфигурацию (либо отладочную(debug), либо рабочую(release) из списка Project Configurations (Настройки проекта) и нажмите кнопку OK.*

Чтобы использовать базовые классы в проектах, необходимо сначала откомпилировать их. Запустите Visual C/C++ и откройте проект BaseClasses, который расположен в инсталляционной директории DirectX SDK в \Samples\Multimedia\DirectShow\BaseClasses\BaseClasses.dsw. Скомпилируйте проект базовых классов, используя отладочную и release настройки.

После компиляции (убедитесь, что вы скомпилировали обе конфигурации!) необходимо скопировать две полученные библиотеки (\debug\strmbasd.lib для отладочной версии и \release\strmbase.lib для рабочей версии) и все файлы .h, расположенные в директории BaseClasses, в директорию вашего проекта.

В диалоговом окне проекта Settings/Link (Установки/Привязка) добавьте strmbasd.lib или strmbase.lib (отладочные или рабочие соответственно, в зависимости от конфигурации) к списку Object/Library Modules (Объектные/Библиотечные модули) проекта. Также убедитесь, что при использовании DirectShow вы подключаете заголовочные файлы streams.h и dshow.h (из директории базовых классов).

Замечание. *Если вы не хотите помещать файлы базовых классов в директорию своего проекта (чтобы сэкономить место на диске или чтобы избежать огромного количества файлов в директории), просто добавьте директорию базовых классов в список используемых компилятором директорий заголовочных файлов. Убедитесь, что проделали то же самое и для директории библиотек; или вместо этого непосредственно привяжите библиотеки в настройках привязки компилятора.*

Теперь вы готовы к действиям! Чтобы импортировать видео данные, необходимо сначала создать собственный фильтр.

Создание специализированного фильтра

Ранее вы узнали как DirectShow использует фильтры для обработки потоков кодируемых или декодируемых данных. В данном случае потоки представляют собой кадры видео. В DirectShow включено несколько фильтров обработки видео, которые вы можете использовать в собственных проектах, но, к сожалению, ни один из этих фильтров нам не пригодится. Необходимо создать собственный специализированный фильтр DirectShow, который берет входящие данные видео и вставляет их на поверхность текстуры, таким образом создавая видео анимацию текстур.

Создать собственный специализированный фильтр не так уж и сложно. Вся тонкость заключается в наследовании базового класса фильтра DirectShow, присвоении ему уникального идентификационного номера и перегрузке соответствующих функций, обрабатывающих поступающие видео данные. Звучит не очень сложно, не так ли? В следующем разделе рассмотрены шаги, необходимые для создания специализированного фильтра.

Наследование класса фильтра

Первым шагом к созданию собственного специализированного фильтра DirectShow является наследование класса фильтра от базовых классов DirectShow. Мы собираемся унаследовать класс, обрабатывающий видео образцы из входного потока, от класса CbaseVideoRenderer. Унаследуйте класс (названный cTextureFilter), который будет использоваться для фильтра.

```
class cTextureFilter : public CBaseVideoRenderer
{
```

Не беря в расчет используемые функции (о которых вы прочитаете чуть ниже), необходимо объявить несколько переменных-членов, используемых для хранения указателя на 3D устройство (IDirect3DDevice9), объект текстуры (IDirect3DTexture9), формат текстуры (D3DFORMAT) и исходную ширину, высоту и шаг видео картинки. Вы можете объявить эти шесть переменных в классе cTextureFilter.

```
public:
  IDirect3DDevice9 *m_pD3DDevice; // интерфейс 3D устройства
  IDirect3DTexture9 *m pTexture; // Объект текстуры
  D3DFORMAT m_Format; // Формат текстуры

  LONG m_lVideoWidth; // Ширина видео, в пикселях
  LONG m_lVideoHeight; // Высота видео, в пикселях
  LONG m_lVideoPitch; // Шаг поверхности видео
```

Первые три переменные m_pD3DDevice, m_pTexture и m_Format являются обычными для хранения текстуры. Среди них интерфейс 3D устройства, используемый в проекте, объект текстуры, содержащий анимированную текстуру, и описание цветового формата текстуры.

Последние три переменные, m_lVideoWidth, m_lVideoHeight и m_lVideoPitch описывают размеры источника видео. DirectShow импортирует видео данные, создавая в памяти растровую поверхность и копируя эти данные на нее. Поверхность имеет собственную высоту (m_lVideoHeight), ширину ((m_lVideoWidth) и шаг поверхности (m_lVideoPitch), определяющий размер строки видео данных в байтах.

Пока хватит о переменных. (Мы вернемся к ним позднее в разделе "Работа со специализированным фильтром".) А сейчас давайте рассмотрим, как присвоить уникальной идентификационный номер фильтру текстур.

Определение уникального идентификационного номера

Чтобы DirectShow мог отличить ваш фильтр от всех остальных, необходимо присвоить ему уникальный идентификационный номер. Этот уникальный номер (номер идентификации класса, если быть точным) представлен в виде UUID, который

вы можете определить с помощью следующего кода. (Обычно необходимо включать этот код в начале исходного файла специализированного фильтра.)

Замечание. *UUID фильтра может быть любым; запустив программу guidgen.exe корпорации Microsoft, вы получите число GUID, которое может быть использовано. Я взял на себя смелость использовать показанный UUID во всех демонстрационных программах этой главы. Если вы хотите посмотреть, как создавать собственные GUID, обратитесь к главе 3.*

```
struct __declspec( \
 uuid("{61DA4980-0487-11d6-9089-00400536B95F}") \
) CLSID_AnimatedTexture;
```

После того как вы определили UUID, необходимо просто использовать его внутри конструктора специализированного фильтра. Необходимо зарегистрировать UUID в системе фильтров DirectShow, чтобы она знала где искать ваш фильтр. Интересно заметить, что как только вы установите фильтр и зарегистрируете его в системе (о чем мы поговорим немного позднее), он все время будет находиться в памяти. Как только загружается медиа файл, фильтр декодирует его. Чтобы ваш фильтр мог декодировать данные медиа, необходимо перегрузить несколько основных функций.

Перегрузка основных функций

После того как вы объявили класс специализированного фильтра и присвоили ему уникальный идентификационный номер, пришло время определить функции, используемые фильтром. Не беря во внимание стандартный конструктор класса, необходимо перегрузить три основные функции DirectShow, которые одинаковы для всех фильтров.

Первая функция, CheckMediaType: определяет, поддерживает ли фильтр заданный тип данных медиа. Т. к. DirectShow может обрабатывать почти любой медиа файл, фильтр будет использовать CheckMediaType для проверки: являются ли входные данные видео данными, сохраненными в используемом фильтром формате. Посмотрите на прототип функции CheckMediaType.

```
virtual HRESULT CBaseRenderer::CheckMediaType(
  const CMediaType *pmt) PURE;
```

У функции CheckMediaType только один параметр - интерфейс CmediaType, определяющий входные данные. Используя функцию FormatType интерфейса CmediaType, вы можете получить тип медиа данных, содержащихся в интерфейсе (видео, звук или что-нибудь еще).

Т. к. нас интересуют только видео файлы, мы будем искать только значение FORMAT_VideoInfo, возвращаемое CMediaType::FormatType. Если данные медиа действительно являются видео, то необходимо выполнить еще одну проверку для определения их формата. Эта проверка заключается в сравнении значения GUID с типом и подтипом видео.

Снова, нам необходим видео медиа тип (представленный MEDIATYPE_Video макросом GUID) а необходимый подтип - макрос GUID цветовой глубины MEDIASUBTYPE_RGB24 (означающий, что используется 24-битная глубина цвета, составленная из красной, зеленой и синей компоненты цвета.)

После того как вы проверили видео поток, вы можете вернуть значение S_OK из вашей функции CheckMediaType, чтобы сообщить DirectShow, что поток может использоваться. Если заданные данные медиа не поддерживаются, вы можете вернуть E_FAIL.

Кажется, что для проверки типа данных требуется выполнить множество операций, но, поверьте мне, это не так. Посмотрите на перегруженную функцию CheckMediaType.

```
HRESULT cTextureFile::CheckMediaType( \
        const CMediaType *pMediaType)
{
  // Принимать только видео тип медиа
  if(*pMediaType->FormatType() != FORMAT_VideoInfo)
   return E_INVALIDARG;

  // Убедиться, что данные используют 24-битный RGB формат цвета
  if(IsEqualGUID(*pMediaType->Type(), MEDIATYPE_Video) && \
     IsEqualGUID(*pMediaType->Subtype(),MEDIASUBTYPE_RGB24))
   return S_OK;

  return E_FAIL;
}
```

Второй перегружаемой функцией является SetMediaType, которая используется фильтром для настройки внутренних переменных и подготовки к обработке внешних данных. Эта функция также может отвергнуть внешние данные, в зависимости от их формата.

Функция SetMediaType имеет только один параметр const CMediaType, также как и CheckMediaType. Внутри функции SetMediaType необходимо получить разрешение видео данных (ширину, высоту и шаг поверхности) и сохранить его в переменных класса.

Также необходимо создать 32-битную поверхность текстуры (которая по умолчанию может быть в 16-битном формате, если 24-битные режимы недоступны) для хранения данных видео по мере их обработки. После того как вы создали текстуру, необходимо проверить ее формат, который должен быть либо 32-либо 16-битным, и сохранить цветовую информацию для дальнейшего использования.

Значение S_OK, возвращаемое функцией SetMediaType, указывает, что фильтр готов к использованию данных видео; в противном случае необходимо вернуть значение ошибки (такое как E_FAIL).

Вы можете создать функцию SetMediaType, получающую разрешение видео и создающую поверхность текстуры, используя следующий код. (Объяснение всех содержащихся операций дано с помощью комментариев.)

```
HRESULT cTextureFilter::SetMediaType(const CMediaType *pMediaType)
{
  HRESULT hr;
  VIDEOINFO *pVideoInfo;
  D3DSURFACE_DESC ddsd;

  // Получить размер этого типа медиа
  pVideoInfo = (VIDEOINFO *)pMediaType->Format();
  m_lVideoWidth = pVideoInfo->bmiHeader.biWidth;
  m_lVideoHeight = abs(pVideoInfo->bmiHeader.biHeight);
  m_lVideoPitch = (m_lVideoWidth * 3 + 3) & ~(3);

  // Создать текстуру, которая соответствует этому типу медиа
  if(FAILED(hr = D3DXCreateTexture(m_pD3DDevice, \
            m_lVideoWidth, m_lVideoHeight, \
            1, 0, D3DFMT_A8R8G8B8, \
            D3DPOOL_MANAGED, &m_pTexture)))
    return hr;

  // Получить описание текстуры и сверить установки
  if(FAILED(hr = m_pTexture->GetLevelDesc(0, &ddsd)))
   return hr;
  m_Format = ddsd.Format;
  if(m_Format != D3DFMT_A8R8G8B8 && m_Format != D3DFMT_A1R5G5B5)
   return VFW_E_TYPE_NOT_ACCEPTED;

  return S_OK;
}
```

Последней перегружаемой и наиболее важной из всех является функция DoRenderSample. DoRenderSample вызывается фильтром при обработке данных каждого кадра, содержащегося в медиа файле. В данном случае Данными является видео клип, вставляемый на поверхность текстуры.

Прототип перегружаемой функции DoRenderSample выглядит так:

```
virtual HRESULT CBaseRenderer::DoRenderSample(
        IMediaSample *pMediaSample);
```

DoRenderSample имеет только один параметр, интерфейс ImediaSample, который содержит информацию об одном медиа экземпляре (растровом изображении, представляющем один кадр видео). Вам необходимо взять этот указатель на данные и вставить их на поверхность текстуры.

Начнем с объявления перегруженной функции DoRenderSample и получения указателя на данные медиа с помощью функции IMediaSample::GetPointer.

```
HRESULT cTextureFilter::DoRenderSample( \
         IMediaSample *pMediaSample)
{
// Получить указатель на буфер видео образца
BYTE *pSamplePtr;
pMediaSample->GetPointer(&pSamplePtr);
```

После этого необходимо заблокировать поверхность текстуры и скопировать данные пикселей из данных видео. Для этого вы можете использовать функцию LockRect объекта текстуры.

```
// Заблокировать поверхность текстуры
D3DLOCKED_RECT d3dlr;
if(FAILED(m_pTexture->LockRect(0, &d3dlr, 0, 0)))
  return E_FAIL;

// Получить шаг текстуры и указатель на данные текстуры
BYTE *pTexturePtr = (BYTE*)d3dlr.pBits;
LONG lTexturePitch = d3dlr.Pitch;
```

После того как вы заблокировали поверхность текстуры и получили ее шаг и указатель на ее данные, можно начинать копировать данные видео. Однако сначала необходимо переместить указатель на нижнюю строчку текстуры, потому что по некоторым странным причинам данные видео хранятся снизу вверх.

```
// Сместить указатель текстуры на нижнюю строчку,т.к. видео хранится
// вверх ногами в буфере
pTexturePtr += (lTexturePitch * (m_lVideoHeight-1));
```

Замечательно, теперь мы готовы скопировать данные видео. Помните, при создании поверхности текстуры цветовой формат текстуры мог быть либо 32- либо 16-битным? Теперь необходимо учитывать глубину цвета при копировании данных видео, потому что за один раз мы можем копировать либо 32 либо 16 бит.

В начале следующего кода стоит оператор switch, который содержит блок, копирующий видео данные, по 32 бита информации за один раз (конечно, если используется 32-битный цветовой формат).

```
// Скопировать информацию, используя заданный видео формат
int x, y, SrcPos, DestPos;
switch(m_Format) {
  case D3DFMT_A8R8G8B8: // 32 бита
```

Следующий кусочек кода просматривает все строки видео образца и копирует каждый пиксель на поверхность текстуры.

```
// Просмотреть каждую строку, копируя информацию
for(y=0;y<m_lVideoHeight;y++) {

  // Скопировать каждый столбец
  SrcPos = DestPos = 0;
  for(x=0;x<m_lVideoWidth;x++) {
   pTexturePtr[DestPos++] = pSamplePtr[SrcPos++];
   pTexturePtr[DestPos++] = pSamplePtr[SrcPos++];
   pTexturePtr[DestPos++] = pSamplePtr[SrcPos++];
   pTexturePtr[DestPos++] = 0xff;
  }

  // Переместить указатель на следующую строку
  pSamplePtr += m_lVideoPitch;
  pTexturePtr -= ITexturePitch;
}
break;
```

Второй оператор switch копирует данные видео, использующие 16-битную глубину цвета. Этот код в целом похож на код копирования 32-битных пикселей, за исключением того, что копируются 16-битные пиксели.

```
case D3DFMT_A1R5G5B5: // 16-бит

  // Просмотреть все строчки, копируя информацию
  for(y=0;y<m_lVideoHeight;y++) {

   // Скопировать каждый столбец
   SrcPos = DestPos = 0;
   for(x=0;x<m_lVideoWidth;x++) {
    *(WORD*)pTexturePtr[DestPos++]  = 0x8000 +
        ((pSamplePtr[SrcPos+2] & 0xF8) << 7)  +
        ((pSamplePtr[SrcPos+1] & 0xF8) << 2)  +
        (pSamplePtr[SrcPos] >> 3);
    SrcPos += 3;
   }

   // Переместить указатель на следующую строку
   pSamplePtr += m_lVideoPitch;
   pTexturePtr -= ITexturePitch;
  }
  break;
}
```

В завершение функции DoRenderSample необходимо просто разблокировать поверхность текстуры и вернуть код успешного окончания или ошибки.

```
// Разблокировать текстуру
if (FAILED(m_pTexture->UnlockRect(0)))
  return E_FAIL;

 return S_OK;
}
```

Каждая из трех только что написанных перегруженных функций (CheckMediaType, SetMediaType и DoRenderSample) вызывается непосредственно DirectShow. Другими словами, вам не надо непосредственно вызывать эти функции: вместо этого позвольте DirectShow вызывать их по мере необходимости. Именно по этой причине все фильтры DirectShow все время находятся в памяти. Это означает, что после создания фильтра, вы никогда не должны самостоятельно удалять его из памяти. DirectShow самостоятельно выполнит это.

Завершение класса фильтра

После того как фильтр принимает внешние данные медиа, для его завершения необходимо добавить всего лишь еще несколько функций. Этими функциями являются конструктор и функция, возвращающая указатель на объект поверхности текстуры. Каждая функция объявляется так (обе функции объявлены открытыми в классе cTextureFilter):

```
class cTextureFilter {
  // ... объявление предыдущих членов

  public:
   cTextureFilter(IDirect3DDevice9 *pD3DDevice, \
            LPUNKNOWN pUnk = NULL, \
            HRESULT *phr = NULL);
   IDirect3DTexture9 *GetTexture();
};
```

Конструктор класса cTextureFilter регистрирует фильтр в системе DirectShow и объявляет интерфейс объекта 3D устройства. Пусть прототип конструктора вас не пугает. Наряду с указателем на объект 3D устройства, в качестве параметров конструктор принимает указатель на объект Iunknown (который можно установить в NULL) и указатель на переменную типа HRESULT, используемый для возвращения результата выполнения функции. Вот код конструктора класса cTextureFilter:

```
cTextureFilter::cTextureFilter(IDirect3DDevice9 *pD3DDevice, \
                LPUNKNOWN pUnk, HRESULT *phr)
        :CBaseVideoRenderer(  uuidof(CLSID AnimatedTexture),
```

```
        NAME("ANIMATEDTEXTURE"), pUnk, phr)
{

  // Сохранить указатель на устройство
  m_pD3DDevice = pD3DDevice;

  // Возвратить успешное окончание выполнения функции
  *phr = S_OK;
}
```

Чтобы зарегистрировать фильтр, необходимо вызвать конструктор CbaseVideoRenderer, как показано в вызове конструктора вашего фильтра. В качестве параметров конструктор CbaseVideoRenderer принимает UUID созданного фильтра и имя, присваиваемое фильтру (в данном случае ANIMATEDTEXTURE). Другие две переменные просто передаются из параметров конструктора фильтра.

Внутри конструктора необходимо просто сохранить указатель объект 3D указателя в переменной класса (m_pD3DDevice) и код успешного завершения в указателе на HRESULT. Вот и весь конструктор класса фильтра!

Вторая (и последняя) добавляемая к классу фильтра функция возвращает указатель на объект поверхность текстуры. Вспомните, объект текстуры был объявлен в классе как m_Texture, поэтому вернуть указатель на него можно при помощи следующего кода:

```
IDirect3DTexture9  *cTextureFilter::GetTexture()
{
  return m_Texture;
}
```

После создания последней функции класс фильтра готов к использованию!

Работа со специализированным фильтром

Хорошо, становится все интереснее! На данный момент специализированный фильтр готов к использованию. Единственное, чего не хватает, это класса, который был создавал соответствующие объекты DirectShow (включая ваш фильтр), загружал бы видео файл и управлял бы воспроизведением видео.

На самом деле, вам необходимо использовать один из основных интерфейсов DirectShow- IGraphBuilder. Интерфейс IGraphBuilder, называемый построитель графов, создает и содержит список фильтров (называемый графом фильтров), которые используются при декодировании медиа файла. Построитель графов загружает медиа файл и настраивает соответствующие фильтры, необходимые для обработки данных этого файла. Давайте рассмотрим этот очень важный объект поближе.

Использование построителя графов

Объект IGraphBuilder очень похож на все остальные интерфейсы COM. Чтобы использовать IGraphBuilder, необходимо создать его экземпляр и вызвать CoCreateInstance для создания фактического объекта и получения его интерфейса.

```
IGraphBuilder *pGraph;
CoCreateInstance(CLSID_FilterGraph,NULL,\
        CLSCTX_INPROC_SERVER,IID_IGraphBuilder, \
         (void**)&pGraph);
```

После создания интерфейса IGraphBuilder вы можете использовать его для регистрации вашего фильтра, чтобы он мог быть использован для загрузки медиа файлов. Однако прежде чем регистрировать фильтр, необходимо создать его копию, которую построитель графов будет держать в памяти при воспроизведении медиа. Для создания экземпляров класса фильтра вы можете использовать оператор new, после чего вы можете зарегистрировать его с помощью функции IGraphBuilder::AddFilter.

Совет. *Прежде чем использовать COM-объекты, необходимо убедиться, что система COM инициализирована. Вы можете инициализировать систему COM, вставив следующую строчку кода в начало вашего приложения (обычно, в начало функции WinMain):*

```
CoInitialize(NULL);
```

При выходе из приложения необходимо деинициализировать систему COM, используя следующую строчку кода:

```
CoUninitialize();
```

Замечание. *Функция CoCreateInstance возвращает S_OK, если она была выполнена успешно. Любое другое возвращаемое значение означает, что при создании COM-объекта произошла ошибка. Для получения дополнительной информации о значении возвращаемых кодов обратитесь к Win32 SDK.*

Вот пример кода, создающего фильтр и регистрирующего его в графопостроителе:

```
// pD3DDevice=указатель на предварительно инициализированный объект 3D
// устройства

// Создать экземпляр фильтра
cTextureFilter *pTextureFilter = \
        new cTextureFilter(pD3DDevice, NULL, &hr);

// Добавить фильтр в построитель графов
IBaseFilter *pFilter = (IBaseFilter*)pTextureFilter;
pGraph->AddFilter(pFilter,  L"ANIMATEDTEXTURE");
```

Функция AddFilter построителя графов принимает в качестве параметров указатель на фильтр (преобразовываемый к интерфейсу класса IBbaseFilter, от которого наследуются все фильтры) и присваиваемое ему имя. В предыдущем примере построителю графов были переданы указатель на фильтр текстур (pFilter) и его имя (ANIMATEDTEXTURE). После этого имя фильтра больше не используется; оно необходимо для ваших собственных целей и для внешнего просмотрщика фильтров, проверяющего все фильтры DirectShow.

После того как вы зарегистрировали фильтр, вы можете использовать его, выбрав исходный видео файл.

Выбор исходного файла

После того как вы зарегистрировали фильтр в DirectShow (при помощи интерфейса IGraphBuilder), необходимо указать DirectShow импортируемый видео файл. Функция IGraphBuilder::AddSourceFilter предназначена для установки исходного файла и создания соответствующих фильтров для импортирования видео данных.

```
HRESULT  IGraphBuilder::AddSourceFilter(
  LPCWSTR lpwstrFileName,
  LPCWSTR lpwstrFilterName,
  IBaseFilter **ppFilter);
```

Функция AddSourceFilter имеет три параметра: имя загружаемого медиа файла, имя исходного фильтра (lpwstrFilterName, о котором вы прочитаете чуть ниже) и указатель (ppFilter) на объект фильтра IBaseFilter, который используется для получения доступа к объекту фильтра видео.

К чему все эти разговоры об исходном фильтре? Я знаю, что ранее говорил, что для декодирования медиа файла используется набор фильтров. Как показано на рис. 14.2, исходный фильтр на самом деле является способом доступа ко всем остальным фильтрам, используемым графопостроителем.

Создав исходный фильтр (используя интерфейс IBaseFilter) и назвав его (с помощью параметра lpwstrFilterName), вы можете быстро получить доступ к разнообразным фильтрам построителя графов через один интерфейс исходного фильтра.

Теперь чувствуется разница, не так ли? Для создания базового фильтра и загрузки медиа файла вызовите AddSourceFilter. Базовый фильтр будет иметь имя SOURCE (преобразованной к Unicode строке с помощью макроса L"") и требует созданного экземпляра интерфейса IBaseFilter.

```
// Filename = имя используемого видео файла
IBaseFilter *pSrcFilter;

// Преобразовать их ASCII текста в Unicode:
```

Рис. 14.2. Исходный фильтр позволяет использовать один интерфейс для представления набора объектов фильтров

```
WCHAR wFilename[MAX_PATH];
mbstowcs(wFilename, Filename, MAX_PATH);

// Добавить исходный фильтр
pGraph->AddSourceFilter(wFilename, L"SOURCE", &pSrcFilter);
```

После вызова AddSourceFilter вы получите указатель на исходный фильтр. Этот указатель pSrcFilter используется для соединения выходной pin источника видео и входной pin фильтра текстуры.

Замечание. *Яконвертировалмультибайтнуютекстовуюстроку Filenateвуникодовскую строку, потому что функция AddSourceFilter (как и большинство функций DirectShow) принимаетвкачествепараметровсимволыуникода.*

Соединение разъемов

О чем это я говорю тут - что за разъемы (pins) и что они делают? Каждый фильтр имеет набор разъемов, которые в каком-то смысле являются трубами, выдающими и принимающими данные медиа в и из фильтра. Данные медиа поступают на входной разъем первого фильтра. Данные обрабатываются фильтром и передаются следующему через выходной разъем. Процесс продолжается до тех пор, пока не закончатся данные, в настоящем случае представленные используемой поверхностью текстуры. Посмотрите на рис. 14.3, чтобы лучше понять, что я имею ввиду.

Каждый фильтр выполняет с данными все необходимые операции и предает их следующему фильтру. Что же касается нашего фильтра, он просто ожидает данных медиа и вставляет их на поверхность текстуры. Опять же, зачем необходимы разъемы?

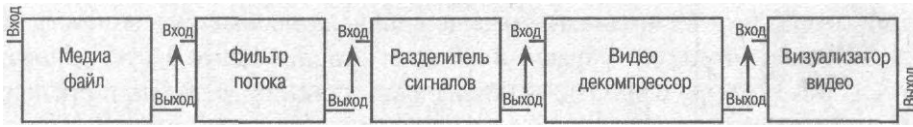


Рис. 14.3. Демонстрационный набор из четырех фильтров используется для получения данных медиа файла, их декодирования и визуализации

Следующим шагом к использованию видео текстуры является нахождение входного разъема (in pin) вашего фильтра (разъема, который принимает входящие данные), выходного разъема (out pin) исходного фильтра (того, из которого данные выходят) и их соединение.

Чтобы найти эти два разъема (входной разъем вашего фильтра и выходной разъем фильтра источника), необходимо использовать функцию FindPin.

```
HRESULT  IBaseFilter::FindPin(
  LPCWSTR Id,  // Имя искомого разъема
  IPin **ppPin);  // Объект интерфейса pin
```

Я знаю, что говорил вам, что мы не будем иметь дела с множеством интерфейсов DirectShow, но я обещаю вам, их осталось немного. Функция FindPin имеет два параметра: имя искомого разъема (в данном случае либо In, либо Out, причем имена представлены Unicode символами) и интерфейс IPin.

В данном случае интерфейсы IPin нам не важны; они нам необходимы только для того, чтобы графопостроитель мог соединить их. Сначала посмотрите код, ищущий оба разъема, а потом переходите к их соединению.

```
// Найти входной и выходной разъемы и соединить их
IPin  *pFilterPinIn;
IPin  *pSourcePinOut;
pFilter->FindPin(L"In",  &pFilterPinIn);
pSourceFilter->FindPin(L"Output",  &pSourcePinOut);
```

Не может быть ничего проще. Сделав вызов еще одной функции, вы можете соединить разъемы.

```
pGraph->Connect(pSourcePinOut,  pFilterPinIn);
```

На данный момент вы загрузили медиа файл и соединили фильтры, подготовив их к работе! Осталось только начать проигрывание видео потока. Проблемой является то, что построитель графов не содержит функций управления проигрыванием. Что за бездельник! Однако не расстраивайтесь, потому что с помощью построителя графов вы можете получить интерфейсы, управляющие проигрыванием видео потоков.

Получение интерфейсов

Чтобы управлять проигрыванием видео данных, необходимо из построителя графов получить интерфейс управления медиа - ImediaControl. Интерфейс управления медиа может проигрывать, останавливать, приостанавливать и продолжать проигрывание видео потока.

Для определения завершения проигрывания видео необходимо получить из графопостроителя интерфейс медиа событий - ImediaEvent. Наконец, для определения текущего положения или даже времени воспроизведения видео необходимо получить еще один дополнительный интерфейс.

Используйте следующий код для получения необходимых интерфейсов:

```
// Создать экземпляры объектов медиа
IMediaControl *pMediaControl;
IMediaPosition *pMediaPosition;
IMediaEvent *pMediaEvent;

// Получить интерфейсы
pGraph->QueryInterface(IID_IMediaControl, \
        (void **)&pMediaControl);
pGraph->QueryInterface(IID_IMediaPosition, \
        (void **)&pMediaPosition);
pGraph->QueryInterface(IID_IMediaEvent, \
        (void **)&pMediaEvent);
```

Не сдавайтесь - мы уже почти закончили! Осталось только начать воспроизведение видео, определять текущее положение видео по мере его воспроизведения и наблюдать за различными возникающими событиями медиа.

Управление проигрыванием и обработка событий

Последним шагом к получению анимированной текстуры является начало воспроизведение видео источника при помощи только что созданного объекта IMediaControl. Интерфейс IMediaControl содержит две интересующие нас функции: IMediaControl::Run и IMediaControl::Stop.

Функция IMediaControl::Run начинает воспроизведение видео с текущего положения, что в свою очередь инициирует передачу данных вашему фильтру и, в конце концов, поверхности текстуры. Функция Run не имеет никаких параметров и может быть использована так:

```
pMediaControl->Run();
```

Во время воспроизведения видео вы можете остановить его, вызвав функцию IMediaControl::Stop, которая также не имеет параметров. Следующий код останавливает воспроизведение видео:

```
pMediaControl->Stop();
```

Замечательным является то, что вы можете использовать функции Run и Stop для приостановки и продолжения видео воспроизведения. Вызов функции Stop приостанавливает воспроизведение, в то время как последующий вызов Run продолжает его.

Чтобы переместиться в заданное положение в видео (например, в начальное положение, когда вы хотите "перемотать" его) используется интерфейс IMediaPosition. Чтобы переместиться на заданное время в видео файле, вызовите функцию IMediaPosition::put_CurrentPosition.

```
HRESULT  IMediaPosition::put_CurrentPosition(
  REFTIME llTime);
```

При ближайшем рассмотрении параметр REFTIME функции putCurrentPosition оказывается вещественным значением; время llTime устанавливается в любое значение для перемещения видео потока. (Например, 2.0 означает переместиться на 2.0 секунды в потоке.) Функция put_CurrentPosition в основном используется для перемотки видео и воспроизведения его сначала, как я сделал тут:

```
// Переместиться в начало потока видео  (0 секунд)
pMediaPosition->put_CurrentPosition(0.0f);
```

Последним, в трио используемых медиа объектов, идет IMediaEvent, который получает все возникающие события. Нас интересует только событие, означающее окончание воспроизведения видео, чтобы его можно было начать заново, process some function to stream in another video или сделать еще что-нибудь.

Чтобы получить событие с помощью интерфейса IMediaEvent, необходимо использовать функцию IMediaEvent::GetEvent.

```
HRESULT  IMediaEvent::GetEvent(
  long *lEventCode,  // Код события
  long *lParam1, // Первый параметр события
  long *lParam2, // Второй параметр события
  long msTimeout); // Время ожидания события
```

Функция GetEvent имеет четыре параметра, первые три из которых являются указателями на переменные типа long, а последний сам является переменной типа long. После вызова GetEvent первые три переменные (lEventCode, lParam1 и lParam2) будут содержать информацию о текущем событии, такую как код события и два его

параметра. Четвертую переменную msTimeout вы устанавливаете равной величине времени, равной задержке между проверкой на возникновение события, или вы можете установить 0, чтобы ждать бесконечно.

Единственное событие, которое вы хотим получить, это окончание видео, представленное макросом EC_COMPLETE. Не надо бесконечно ждать события; вместо этого необходимо подождать несколько миллисекунд, а потом продолжать получать события, пока их не останется. Как узнать, что больше не осталось событий? Как только функция GetEvent вернет ошибку, можно смело полагать, что больше событий нет.

Вот пример кода, который циклически получает текущее событие из интерфейса событий медиа (останавливаясь, когда событий больше не осталось), и небольшой блок кода, обрабатывающий событие, - EC_COMPLETE.

```
// Обработать все ожидающиеся события
long lEventCode, lParam1, lParam2;
while(1) {
  // Получить события и подождать миллисекунду
  if(FAILED(pMediaEvent->GetEvent(&lEventCode, &lParam1, &lParam2,
1)))
   break;

  // Обработать события окончания видео, вызвав специальную функцию
  if(lEventCode  == EC_COMPLETE)
   EndOfAnimation(); // Здесь может быть любая функция

  // Освободить ресурсы события
  pMediaEvent->FreeEventParams(lEventCode, lParam1, lParam2);
}
```

Ха! Я попытался обмануть вас, использовав вызов новой функции интерфейса ImediaEvent - FreeEventParams. Каждый раз, когда вы получаете события с помощью функции GetEvent, необходимо сопровождать их соответствующим вызовом FreeEventParams, чтобы объект события медиа мог освободить все ресурсы, используемые для события.

Вот и все. Теперь вы можете воспроизводить, останавливать, приостанавливать, продолжать и изменять положение воспроизведения видео текстуры, а также проверять завершение воспроизведения (по достижении которого вы можете начать воспроизведение заново или сделать что-либо другое).

Создание менеджера анимированных текстур

Не то чтобы использование анимированных текстур было сложным (с помощью этой книги конечно), но ваша жизнь могла бы стать намного проще, если бы вы потратили немного времени и создали менеджер, который бы управлял фильтрами

и текстурами. Я взял на себя смелость и объединил все, что было сказано о текстурах в этой главе, в один класс, который вы можете использовать в своих игровых проектах.

Вы можете найти этот класс, названный cAnimationTexture, на компакт-диске, в директории \BookCode\Chap14\TextureAnim\. Объявление класса выглядит так:

```
class cAnimatedTexture
{
  protected:
    IGraphBuilder *m_pGraph; // граф фильтра
    IMediaControl *m_pMediaControl; // Управление проигрыванием
    IMediaPosition *m_pMediaPosition; // Управление положением
    IMediaEvent *m_pMediaEvent; // Управление событиями
    IDirect3DDevice9 *m_pD3DDevice; // 3D устройство
    IDirect3DTexture9 *m_pTexture; // Объект текстуры

  public:
    cAnimatedTexture();
    ~cAnimatedTexture();

    // Загрузить и освободить объект анимированной текстуры
    BOOL Load(IDirect3DDevice9 *pDevice, char *Filename);
    BOOL Free();

    // Обновить текстуру и проверить цикл
    BOOL Update();

    // Вызывается в конце анимации
    virtual BOOL EndOfAnimation();

    // Функции проиграть и остановить
    BOOL Play();
    BOOL Stop();

    // Начать анимацию заново или перейти к заданному времени
    BOOL Restart() ;
    BOOL GotoTime(REFTIME Time);

    // Вернуть указатель на объект текстуры
    IDirect3DTexture9 *GetTexture();
};
```

Большая часть членов класса cAnimatedTexture должна быть вам знакома. В нем содержаться обычные интерфейсы DirectShow, используемые для воспроизведения событий и информации о положении, пара объектов Direct3D, применяемые для указания на используемое 3D устройство, и объект поверхности текстуры, указывающий на поверхность текстуры фильтра.

Наряду с переменными, в классе определено 11 функций. Среди которых - конструктор и деструктор класса, которые инициализирует его переменные и освобождают используемые ресурсы соответственно. Далее идет функция Load, которая

загружает видео файл и подготавливает к использованию фильтр. Для загрузки файла функции Load необходимо предоставить только указатель на используемый объект IDirect3DDevice9 и имя загружаемого видео файла, используемого в качестве текстуры! Функция Load создаст экземпляр класса cTextureFilter и будет использовать его для загрузки данных видео. После завершения работы с объектом класса cAnimated-Texture вызовите функцию Free для освобождения всех используемых им ресурсов и интерфейсов.

Далее следует функция Update, которая должна вызываться при каждой итерации вашего цикла сообщений (каждый кадр игры). Функция Update опрашивает объект события медиа, есть ли в нем какие-либо события; если есть, то они обрабатываются. Если воспроизведение видео было окончено, тогда вызывается функция EndOfAnimation.

Вы заметите, что функцию EndOfAnimation можно перегружать, что позволяет вам создавать собственные функции, определяющие, что необходимо сделать по окончании воспроизведения видео. Например, вы можете вызвать функцию Restart, которая запускает воспроизведение заново с самого начала, или можете вызвать Goto-Time, которая принимает в качестве параметра значение REFTIME (вещественное значение), означающее смещение воспроизведения от начала.

Наконец, есть еще три функции. Start начинает воспроизведение видео с текущего положения, а Stop останавливает воспроизведение. Существует возможность приостановить воспроизведение, вызвав Stop, а потом продолжить его, вызвав Play.

И, наконец, функция GetTexture, которая аналогична функции фильтра GetTexture; обе возвращают указатель на интерфейс объекта текстуры, который позволяет вам установить текстуру с помощью вызова IDirect3DDevice9::SetTexture.

Я не собираюсь приводить код класса cAnimatedTexture, т. к. большую его часть вы уже видели в этой главе. Опять же, я рекомендую вам посмотреть исходный код проекта TextureAnim на компакт-диске книги. Чтобы увидеть функциональность класса cAnimatedTexture, давайте рассмотрим небольшой пример работы с анимированными текстурами.

Применение анимированных медиа текстур

После того как вы закончили движок анимированных видео текстур, пришло время протестировать его и создать что-нибудь стоящее. В этом примере вы создадите квадратный полигон (полигон заданный четырьмя точками) и примените к нему анимированную текстуру. Чтобы не загружать пример, я пропущу код инициализации Direct3D и сразу перейду к примеру, сначала создав буфер вершин, содержащий данные полигона.

Замечание. *При использовании фильтров DirectShow, разработанных в этой главе, необходимо задать флаг D3DCREATE_MULTITHREADED при вызове IDirect3D9::CreateDevice. Это позволяет сообщить Direct3D, что используется многопоточное приложение (в котором фильтры являются отдельными потоками), и позволяет получить доступ к фильтрам данных поверхности текстуры. Также необходимо использовать многопоточные динамические библиотеки (устанавливаемые в настройках компилятора).*

Создание буфера вершин

Чтобы использовать анимированную текстуру, необходимо сначала наложить ее в качестве обычной текстуры на полигон (или набор полигонов). Чтобы продемонстрировать анимированную текстуру, создайте простой квадратный полигон помощью двух треугольников. Вы можете сделать это при помощи буфера вершин и четырех вершин (задав ленту треугольников[1]).

```
// pDevice = предварительно инициализированное устройство Direct3D
typedef struct {
  float x, y, z; // 3D координаты
  float u, v; // Текстурные координаты
} sVertex;
#define VERTEXFVF (D3DFVF_XYZ | D3DFVF_TEX1)

// определить данные прямоугольника (два треугольника в ленте)
sVertex Verts[4] = {
  { -128.0f, 128.0f, 0.0f, 0.0f, 0.0f },
  { 128.0f, 128.0f, 0.0f, 1.0f, 0.0f },
  { -128.0f, -128.0f, 0.0f, 0.0f, 1.0f },
  { 128.0f, -128.0f, 0.0f, 1.0f, 1.0f }
};
// Создать буфер вершин
IDirect3DVertexBuffer9 *pVertices;
pDevice->CreateVertexBuffer(sizeof(Verts), \
        D3DUSAGE_WRITEONLY, VERTEXFVF, \
        D3DPOOL_MANAGED, &pVertices);

// Установить данные вершин
BYTE *VertPtr;
pVertices->Lock(0, 0, (BYTE**)&VertPtr, D3DLOCK_DISCARD);
memcpy(VertPtr, Verts, sizeof(Verts));
pVertices->Unlock();
```

В этом коде создается буфер вершин, содержащий четыре вершины, которые расположены в strip треугольниках, образующих квадрат. Здесь используется обычный буфер вершин, содержащий координаты вершин и один набор текстурных координат.

1. Лента треугольников (triangle strip) - один из способов задания полигона. Более подробно смотри справку по DirectX SDK.

После того как вы создали буфер вершин, пришло время загрузить анимированную текстуру.

Совет. *Если вы используете двухмерное проигрывание видео (в противоположность визуализации текстуры на несколько полигонов), вы можете заменить непреобразованные трехмерные координаты на набор преобразованных двухмерных (задав флаг FVFD3DFVF_WYZRHW).*

Загрузка анимированных текстур

Вы подготовили буфер вершин и, конечно же, файл анимированной текстуры. Все, что осталось сделать, - это загрузить текстуру и визуализировать сцену! Для этого примера унаследуйте от класса cAnimatedTexture класс, который бы циклически проигрывал видео, перегружая функцию EndOfAnimation так, чтобы она вызывала Reset при окончании видео. Вот унаследованный класс, который вы можете использовать:

```
class cAnimTexture : public cAnimatedTexture
{

  public:
   BOOL EndOfAnimation() { Restart(); return TRUE; }
};
```

Далее просто создайте экземпляр класса cAnimTexture и при помощи функции Load загрузите видео.

```
cAnimTexture *g_Texture = new cAnimTexture();
g_Texture->Load(pDevice, "Texture.avi");
```

На данный момент текстура загружена, и вы готовы к визуализации сцены.

Подготовка к визуализации

Пришло время подготовится визуализировать полигоны, использующие анимированную текстуру. Выполнить это можно с помощью обычного кода - ничего необычного здесь не происходит. Просто установите источник потока вершин, шейдер, материал и текстуру. Все правильно, вам просто необходимо установить поверхность текстуры, содержащей видео данные; DirectShow сделает все остальное!

Следующий кусочек кода взят из примера этой главы и используется для установки источника потока вершин, FVF шейдера, материала и текстуры:

```
pDevice->SetFVF(VERTEXFVF);
pDevice->SetStreamSource(0, pVB, sizeof(sVertex));
pDevice->SetMaterial(&Material);
pDevice->SetTexture(0, g_Texture->GetTexture());
```

После того как вы установили источник, FVF шейдер, материал и текстуру, все, что необходимо сделать, - это нарисовать примитив.

Рисование и представление сцены

Опять же, чтобы нарисовать графические примитивы и представить сцену не надо делать ничего необычного. Т. к. DirectShow управляет потокам видео данных в фоновом режиме, вы можете рисовать примитивы, как если бы вы рисовали обычную сцену.

Следующий код визуализирует буфер вершин, освобождает использование источников текстур и вершин (для ликвидации утечек памяти) и представляет сцену.

```
pDevice->DrawPrimitive(D3DPT_TRIANGLESTRIP, 0, 2);
pDevice->SetStreamSource(0, NULL, 0);
pDevice->SetTexture(0,  NULL);
pDevice->Present(NULL, NULL, NULL, NULL);
```

Вот и все - полностью анимированная текстура, наши поздравления DirectShow и Вам! После того как вы увидели насколько просто работать с анимированными текстурами, вы можете добавлять различные эффекты в ваши игры, такие как лицевые текстуры, полностью анимированные фоновые изображения и даже кинематографические последовательности, проигрываемые 3D движком.

Посмотрите демонстрационные программы

Для иллюстрирования технологий анимации текстур, о которых вы прочитали в этой главе, создано две демонстрационные программы. Обе эти программы отображают горный водопад (см. рис. 14.4), показываю лишь небольшую часть того, что может быть создано с помощью анимации текстур.

Первая демонстрационная программа Transformations использует преобразование текстур для плавного перемещения текстуры воды, создавая эффект водопада. Вторая демонстрационная программа TextureAnim использует фильтры видео текстур, разработанные в этой главе для анимации поверхности воды.

Окончание современной анимации

Грустно.... К сожалению, мой друг, мы достигли конца пути. Вы закончили последнюю главу. Однако не расстраивайтесь, потому что эта книга только верхушка айсберга. Существует множество еще более продвинутых технологий, которые вы можете изучить! Если вам интересно, посмотрите приложение А "Ссылки на книги и сайты", чтобы ознакомиться с доступными ресурсами. Уверен, что дальнейшее изучение вебсайтов поможет вам реализовать все ваши смелые фантазии,

Итак, вперед! Миру продвинутой анимации еще есть, чем вас удивить. Удачи!

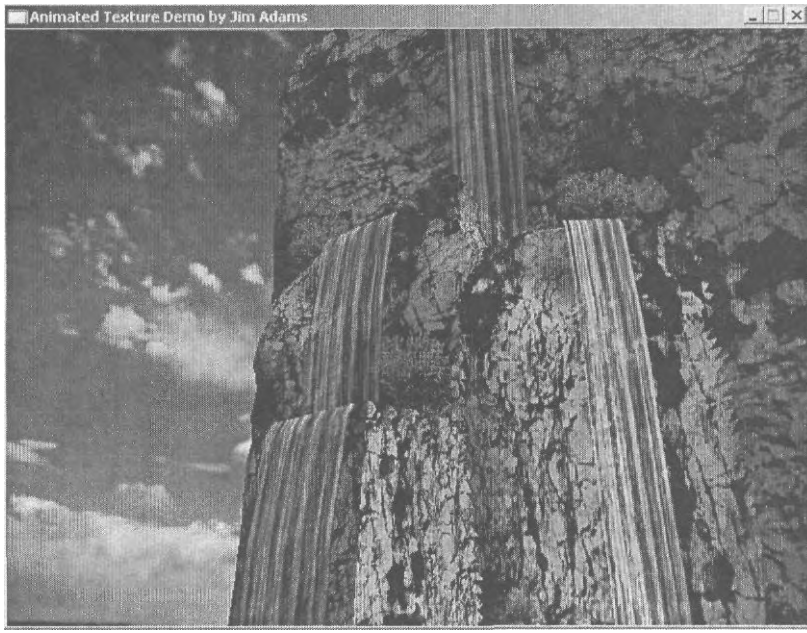


Рис. 14.4. Водопад, имеющий невидимый источник, находящийся в горах, в демонстрационных программах TextureAnim и Transformations

Программы на компакт-диске

Директория главы 14 компакт-диска содержит два проекта:

- **Transformations.** Этот проект иллюстрирует использование преобразования текстур для анимации воды сцены. Он расположен в \BookCode\Chap14\Transformations.
- **TextureAnim.** Этот исходный файл использует класс фильтра текстур, разработанный в этой главе для иллюстрации сцен, использующих анимированные текстуры. Он расположен в \BookCode\Chap14\TextureAnim.

Часть VI

Приложения

Приложение А

Ссылки на книги и сайты

В наше время существует множество разнообразной информации, к которой легко получить доступ, надо просто знать, где ее искать. В этом приложении я привел список веб-сайтов и книг, которые могут быть полезными при изучении продвинутой анимации.

Веб-сайты

Найти какую-нибудь информацию в сети достаточно сложно; необходимо просмотреть тысячи страничек, что превращает поиск информации в сложное путешествие. Куда обращаться программистам, когда им нужна помощь или какая-нибудь информация при возникновении сложностей с проектами? Хорошо, я привел небольшой список сайтов, которые считаю полезными. Надеюсь, они помогут вам также, как помогли мне.

The Collective Mind

http://www.theCollectiveMind.net

Я начну со своего собственного сайта, где вы можете найти сведения о моих книгах, загрузить последние обновления кода, почитать статьи, посвященные программированию игр, загрузить демонстрационные программы и многое другое. Если вам будет нужна помощь в изучении какой-либо из моих книг или проектов, зайдите на мой сайт и посмотрите, не было ли обновлений, или можете написать мне электронное письмо.

GameDev.net

http://www.GameDev.net

Здесь содержится ВСЕ, что необходимо разработчику игр. Общайтесь с другими разработчиками в форумах, читайте статьи, участвуйте в соревнованиях по кодингу, загружайте демонстрационные программы и игры, созданные посетителями сайта GameDev. Если не брать во внимание мой собственный сайт, это лучшее место, где вы можете меня найти, общающегося в форумах.

Premier Press Books

http://www.PremierPressBooks.com

Здесь вы можете найти подборку замечательных книг, посвященных программированию игр, и анонсы выпускаемых книг издательства Premier. Наряду с программированием игр, Premier предлагает полный набор технических книг, посвященных множеству тем, связанных с компьютерами, таким как моделирование, цифровое изображение, сети и безопасность и еще множество всего.

Microsoft

http://www.microsoft.com

Если вы читаете эту книгу, то почти наверняка вы уже знаете веб-сайт корпорации Microsoft. Особенно его странички, посвященные DirectX и Agent. Увидев обилие статей, демонстрационных программ, вы наверняка станете постоянным посетителем этого сайта.

Caligari

http://www.caligari.com

Небольшая цена и высокое качество - именно так можно охарактеризовать продукцию компании Caligari. Именно эта компания является разработчиком пакета трехмерного моделирования trueSpace. Этот сайт обязательно необходимо посетить тем, у кого небольшой бюджет, но требуется качественное программное обеспечение. Если вы нуждаетесь в таких вещах как лицевая анимация (вы читали о ней в этой книге), нелинейное редактирование, создание текстур и, наконец, возможность экспорта непосредственно в формат файла .X, вы обязательно должны посетить сайт компании Caligari.

NVIDIA

http://www.nvidia.com

На сайте создателя линейки видео карт GeForce NVIDIA содержится огромное количество документов о DirectX и OpenGL, а также демонстрационные программы, иллюстрирующие их. Если вы хотите получить информацию о таких технологиях как пиксельные и вершинные шейдеры, собственный высокоуровневый язык программирования шейдеров NVIDIA, называемый Cg, то вам обязательно надо посетить этот сайт.

ATI Technologies, Inc.

http://www.atitech.com

Компания ATI является создателем линейки видео карт Radeon. Как и на сайте NVIDIA, на сайте ATI содержится множество документации, утилит, множесство демонстрационных программ, которые предназначены в помощь разработчикам, использующим DirectX и OpenGL. На этом сайте также содержится информация о программном пакете RenderMonkey, который компания ATI разработала для программирования шейдеров.

Curious Labs, Inc.

http://www.curiouslabs.com

Бегло упомянутый в этой книге сайт компании Curious Labs является домашней страницей привлекательного программного пакета трехмерного моделирования и анимации персонажей Poser. Вы определенно захотите посетить сайт компании Curious Labs, программный пакет которой предоставляет вам возможность располагать, визуализировать, анимировать и экспортировать трехмерные персонажи в множество популярных форматов.

Polycount

http://www.polycount.com

Сайт Polycount является основным источником трехмерных моделей персонажей, которые используются в таких популярных играх как Quake II и III, Half-Life и Grand Theft Auto 3. Если вам необходимы модели для игры или художник для проекта, то это как раз то место, где вы можете их найти!

Flipcode

http://www.flipcode.com

Еще один неизвестный сайт посвященный программированию. Здесь вы можете найти последние новости, посвященные программированию, общаться с другими пользователями и даже посмотреть галерею лучших картинок дня (image of the day). Когда вы будете на этом сайте, обязательно посмотрите ссылки Kurt'a на некоторые интересные научные разработки и инновации!

Chris Hecker and Jeff Lander Physics

Chris Hecker: http://www.d6.com/users/checker

Jeff Lander: http://www.darwin3d.com

Целью этой книги было помочь вам в короткие сроки научится создавать привлекательные эффекты анимации. Я признаю, что иногда я немного небрежно относился к физике, но целью было показать вам, как можно достичь тех или иных эффектов. Для получения дополнительной информации я рекомендую вам изучить статьи или демонстрационные программы Chris Hecker или Jeff Lander, посвященные использованию физики в играх.

Рекомендуемые книги

Каждый великий программист имеет собственный набор используемых книг. Я не являюсь исключением, поэтому предлагаю вашему вниманию обзор книг, которые тематически пересекаются с моей книгой.

Programming Role Playing Games with DirectX by Jim Adams (Premier Press, 2002)

ISBN: 1-931841-09-8

Еще одна безобразная реклама меня! На самом деле моя книга по программированию ролевых игр является великолепным пособием тем, кто хочет делать игры с нуля. Эта книга не ограничивается созданием ролевых игр, она показывает ключевые компоненты, используемые в игровом движке, такие как графический октантный движок, управление персонажем и инвентарем. Она также показывает, как создать ядро библиотек, существенно упрощающих разработку игровых проектов.

FocusOn3DModels byEvanPipho(PremierPress,2002)

ISBN: 1-59200-033-9

Масло моего Хлеба, Рахат моего Лукума - эта книга является путеводителем по основным форматам трехмерных моделей, используемых в современных играх. Эта книга является отличным помощником тем, кто хочет использовать не только форматы файлов .X, .MD2 и .MS3D, описанные в этой книге.

Real-TimeRenderingTricksandTechniquesinDirectX byKellyDempski(PremierPress,2002)

ISBN: 1-931841-27-6

Kelly Dempski показывает как перейти к некоторым продвинутым методам визуализации, например вершинным шейдерам, для получения замечательных эффектов. Эта книга пригодится вам, если вы начинаете изучать вершинные и пиксельные шейдеры, она также содержит разделы, посвященные улучшению игр.

Direct3DShaderX:VertexandPixelShaderTipsandTricks byWolfgangF.Engel(Wordware,2002)

ISBN: 1-556220-41-3

Это еще одна замечательная книга об использовании пиксельных и вершинных шейдеров. Она поможет вам улучшить графику, используя такие эффекты как пузырьковая визуализация и волны на воде. Применяя технологии, описанные в книге, вы можете создавать привлекательные анимационные эффекты!

Приложение Б

Содержимое компакт-диска

Вы прочитали книгу, теперь пришло время посмотреть программное обеспечение! Компакт-диск этой книги содержит исходный код каждой демонстрационной программы, о которой упоминалось в книге, и, кроме того, полезные утилиты и приложения, используемые для создания эффектов, упоминаемых в книге.

Если вы еще этого не сделали, я рекомендую вам распечатать компакт-диск и вставить его в привод для чтения компакт-дисков. Если автозапуск (autorun) разрешен, вы увидите интерфейс, с которым будете работать при изучении содержимого компакт-диска. Если автозапуск запрещен, вы можете запустить интерфейс, выполнив следующие шаги:

1. Вставьте компакт-диск в устройство для чтения компакт-дисков.
2. Правой кнопкой щелкните на My Computer (Мой компьютер) и в появившемся меню выберите Open (Открыть).
3. Щелкните на устройство для чтения компакт-дисков в списке устройств.
4. В списке содержимого компакт-диска дважды щелкните на файле Start_Here.html. После прочтения лицензионного соглашения нажмите I Agree (Я согласен), если вы принимаете условия (или I Disagree (я не согласен), чтобы выйти из интерфейса).

Если вы приняли лицензионной соглашение и щелкнули кнопку I Agree, вы увидите пользовательский интерфейс Premier Press. Из него вы можете просматривать содержимое компакт-диска или инсталлировать приложения. Вот список приложений, находящихся на компакт-диске.

DirectX 9.0 SDK

Основным приложением является DirectX 9 SDK корпорации Microsoft. Первым делом вы должны установить SDK, т. к. он используется на протяжении всей книги. Для получения помощи в установке DirectX 9.0 SDK проконсультируйтесь с главой 1 "Подготовка к изучению книги".

GoldWave Demo

Все звуковые файлы, которые используются в этой книге, были созданы с помощью GoldWave - небольшой программы редактирования звуков. Эта небольшая программа, созданная GoldWave Inc., позволяет одновременно работать с несколькими звуками. Вы не найдете другой программы, в которой бы запись, воспроизведение иизменение звуков происходили так легко, как в этой. С помощью GoldWave вы можете:

- Редактировать звуковые файлы размером до 1Гб;
- В реальном времени просматривать амплитуды, спектры и т. д. звуковых файлов;
- Осуществлять быструю навигацию, используя средства быстрой прокрутки вперед и назад;
- Сохранять и загружать файлы, сохраненные во множестве форматов, как .WAV, .AU, .MP3, .OGG, .AIFF, .VOX, .MAT, .SND и .VOC;

Выделять кусочки звука при помощи мыши.

Paint Shop Pro Trial Version

Бюджетное решение для редактирования изображений. Используя Paint Shop Pro, вы можете редактировать изображения, как профессионалы, не платя при этом! Эта 30-дневная версия программы редактирования изображений фирмы Jasc позволяет вам открывать множество графических форматов, работать с несколькими слоями, использовать плагины и ретушировать изображения.

TrueSpace Demo

Хотите иметь мощные возможности трехмерного редактирования по приемлимой цене? Тогда программа трехмерного моделирования traeSpace фирмы Caligari как раз то, что вам нужно. Эта демонстрационная программа позволяет вам изучить разнообразные возможности новой версии trueSpace. Вы обязательно захотите посмотреть эту программу, потому что она имеет такие особенности как система лицевой анимации, гибридная радиосити (radiosity) визуализация и нелинейное редактирование!

Microsoft Agent and LISET

Пакет Agent корпорации Microsoft содержит очень полезные средства языковой разработки, которые позволяют создавать речь на основе текста, распознавание речи и приложения синхронизированной анимации губ. Используя программу LISET, распространяемую с Agent, вы можете создавать собственную синхронизированную анимацию губ, используемую в игровых проектах, как показано в этой книге.

Предметный указатель

D

E

F

И

W

X-Z

Б

В

Г

Д

З

И

К

Л

М

H

O

П

Ф

Содержание