ЧАСТЬ I

Введение в программирование трехмерных игр

В этой части...

ГЛАВА 1

Основы программирования трехмерных игр

В этой главе...

Для того чтобы разогреть ваш интерес, в этой главе мы рассмотрим некоторые общие темы программирования игр, такие как игровые циклы и различия между двумерными и трехмерными играми. В конце мы создадим небольшую трехмерную игру, чтобы правильно настроить компилятор и DirectX. Если вы уже прочитали мою первую книгу *Программирование игр для Windows. Советы профессионала*, то, наверное, можете бегло просмотреть эту главу и сосредоточиться на материале, изложенном в самом конце. Если же нет, вам определенно стоит прочитать ее — даже если вы игровой программист среднего или высокого уровня. Итак, вот содержание главы:

- краткое введение в программирование игр;
- элементы двумерных и трехмерных игр;
- общие советы по программированию игр;
- использование инструментов;
- пример игры для Windows: Raiders 3D.

Краткое введение

Эта книга на самом деле представляет собой второй том серии (вероятно, трехтомной), посвященной программированию двумерных и трехмерных игр. В первом томе *Программирование игр для Windows. Советы профессионала* рассматриваются в первую очередь следующие темы.

- Программирование для Windows
- Интерфейс прикладного программирования (API) Win32
- Основы DirectX
- Искусственный интеллект
- Основы физического моделирования
- Звук и музыка
- Алгоритмы
- Программирование игр
- Двумерная растровая и векторная графика

Эта книга продолжает предыдущую. Тем не менее, я попытался написать ее так, что если вы не читали первую, то все равно сможете получить из данной книги массу информации по трехмерной графике реального времени и ее применению для создания трехмерных игр. Следовательно, идеальный читатель данной книги — это тот, кто прочел первую книгу серии и интересуется программированием трехмерных игр, либо тот, кто уже умеет делать двумерные игры и стремится освоить трехмерные методы с точки зрения написания программ и алгоритмов.

Помня об этом, я собираюсь сосредоточиться в этой книге на трехмерной математике и графике, и в меньшей степени касаться всего, что относится к программированию игр. Я исхожу из того, что это вы уже умеете, — если нет, я в очередной раз советую вам прочитать *Программирование игр для Windows. Советы профессионала* (либо любую другую книгу по программированию игр) и посидеть за компьютером (до ощущения усталости), изучая Windows, DirectX и игровое программирование в целом.

С другой стороны, даже если вы не читали предыдущую книгу и ничего не знаете о программировании игр, вы все равно кое-что почерпнете для себя из этой книги — хотя бы потому, что здесь много иллюстраций. Мы собираемся заниматься от главы к главе созданием трехмерного игрового процессора, однако чтобы сэкономить время (и около 1500 страниц), мы начнем с базового процессора DirectX, разработанного в первой книге. Конечно, этот процессор использовал DirectX версий 7 и 8; сейчас DirectX изменился и в версии 8.0+ поддержка двумерных приложений стала сложнее, поскольку DirectDraw теперь объединен с Direct3D. Мы собираемся продолжить работу с интерфейсами DirectX 7 и 8, но компилировать наши приложения с DirectX 9.0.

НА ЗАМЕТКУ

Это книга о программном обеспечении и алгоритмах, а не о DirectX. Поэтому я мог бы написать программу под DOS, и такой материал был бы вполне уместен. Нам необходимы только в меру сложная графика, система ввода/вывода и работа со звуком — для этих целей более чем достаточно DirectX 8.0+. Другими словами, если вы приверженец Linux, перенос игрового процессора под SDL пройдет без проблем!

Таким образом, если вы уже прочитали предыдущую книгу, игровой процессор и его функции уже знакомы вам. Если нет, его следует считать “черным ящиком” (код кото-

рого, впрочем, прилагается), поскольку в данной книге мы будем лишь добавлять трехмерный аспект к тому, что уже было сделано ранее.

Не беспокойтесь, в следующей главе мы полностью рассмотрим API и структуру двумерного игрового процессора, а также все функции, которые он выполняет. Я также покажу вам ряд демонстрационных программ, чтобы заинтересовать вас в использовании базового процессора DirectX, созданного в первой книге.

В этой книге я хочу попытаться представить максимально общую, "виртуальную" точку зрения на графическую систему. Хотя код будет основан на игровом процессоре из первой книги, главное в том, что этот игровой процессор не делает ничего, кроме настройки графической системы с двойной буферизацией со стандартной линейной адресацией.

Таким образом, если вы хотите перенести такой код на платформу Mac или Linux, он должен нормально заработать — после нескольких часов работы (максимум — до недели). Моя цель — научить вас работать с трехмерной графикой и математикой в целом. Так сложилось, что доминирующей вычислительной системой в настоящий момент является DirectX на платформе Windows, поэтому именно на ней я и построил низкоуровневую обработку. И мы посвятим наше время рассмотрению именно этих концепций верхнего уровня.

Мне уже приходилось писать о двумерных и трехмерных игровых процессорах, и у меня на жестком диске есть соответствующий материал. Однако когда я пишу книгу, я предпочитаю создавать новый игровой процессор, т.е. я пишу игровой процессор для книги, а не книгу для игрового процессора. В этой книге я не использую трехмерный игровой процессор повторно, а создаю его. Поэтому я не вполне уверен, что мне удастся эта книга! Мне даже интересно узнать, что же выйдет из этой затеи. Вы узнаете все необходимое для того, чтобы создать собственный игровой процессор Quake, но я могу по ходу дела переключиться на процессор для игр на открытом воздухе или какой-то еще — кто знает, куда меня занесет? Я исхожу из того, что работа над процессором гораздо полезнее, чем код с примечаниями автора.

Наконец, читатели моей первой книги могут заметить, что часть материала одинакова в обеих книгах. В самом деле, я не могу сразу начать с 3D-графики без использования материала из предыдущей книги. Я не могу рассчитывать, что у каждого есть моя первая книга и не могу заставить купить ее. В любом случае, в первых нескольких главах материал будет несколько перекликаться с первой книгой — по крайней мере, в отношении DirectX, игрового процессора и Windows.

Элементы двумерных и трехмерных игр

Для начала давайте посмотрим, чем видеоигра отличается от любого другого вида программ. Видеоигры — очень сложный вид программ, писать которые труднее всего. Конечно, написать что-нибудь типа MS Word труднее, чем игру типа *Asteroids*, но написать что-то вроде *Unreal*, *Quake Arena* или *Halo* труднее, чем любую программу, которую можно себе представить, включая военную программу управления вооружениями!

Это означает, что вам нужно изучить новый способ программирования, который более подходит для написания моделирующих приложений реального времени, чем программ, управляемых событиями, или последовательных логических программ. Видеоигра — это непрерывный цикл, в котором исполняется логический сценарий и рисуется картинка на экране — обычно с частотой не менее 30-60 кадров в секунду. Это напоминает фильм, но фильм, которым можно управлять.

Давайте начнем с рассмотрения упрощенной схемы игрового цикла (рис. 1.1). Вот краткое описание стадий цикла.

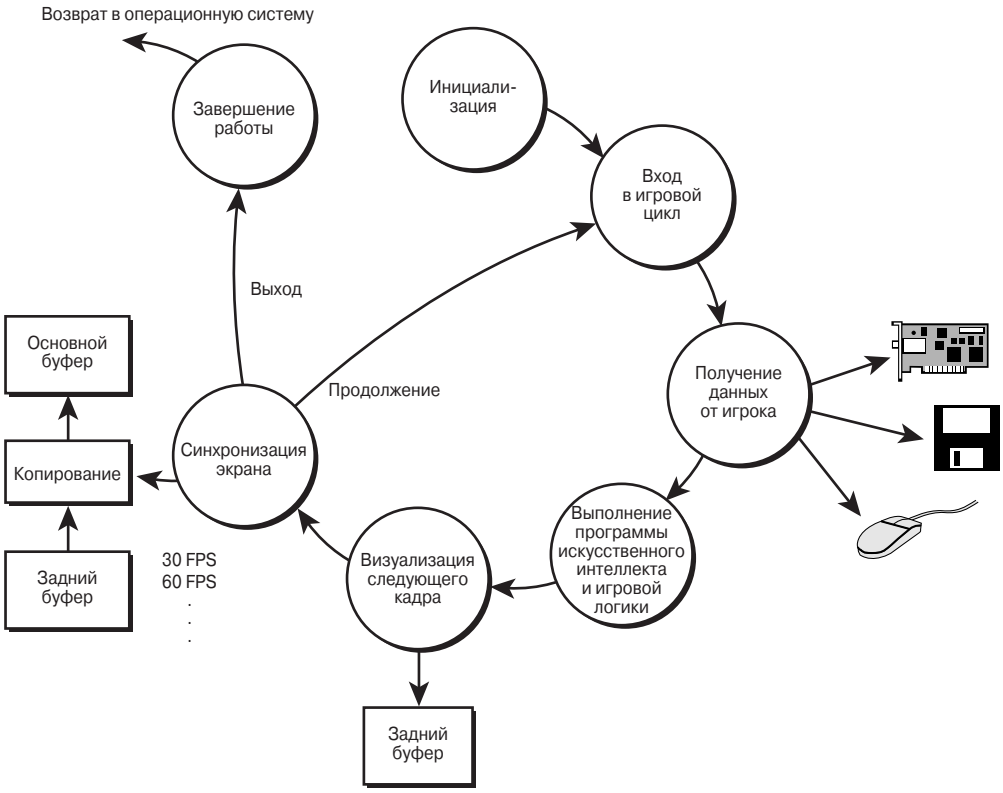


Рис. 1.1. Общая архитектура цикла игры

Стадия 1. Инициализация

На этой стадии выполняются стандартные операции, характерные для любой программы, такие как выделение памяти, получение ресурсов, загрузка данных с диска и т.п.

Стадия 2. Вход в игровой цикл

На этой стадии выполнение кода достигает входа в игровой цикл. Здесь начинается действие игры, и оно продолжается, пока пользователь не выйдет из основного цикла.

Стадия 3. Получение данных от игрока

На этой стадии данные, введенные игроком, обрабатываются и/или заносятся в буфер для дальнейшего использования на стадии исполнения программы искусственного интеллекта и логики.

Стадия 4. Выполнение программы искусственного интеллекта и игровой логики

Эта стадия содержит основную часть игрового кода. Исполняются программы искусственного интеллекта, физических систем, общей игровой логики. Результаты используются для формирования следующего кадра изображения.

Стадия 5. Визуализация следующего кадра

На этой стадии данные, полученные от игрока, и результаты исполнения программы искусственного интеллекта и логики используются для формирования следующего кадра игрового изображения. Это изображение обычно формируется во внеэкранной буферной зоне. Его визуализацию невозможно увидеть. Затем происходит быстрое копирование изображения на экран, в результате чего возникает анимационный эффект. В случае трехмерного программного игрового процессора происходит визуализация тысяч (в некоторых случаях миллионов) многоугольников, формирующих игровую сцену. В случае трехмерного игрового процессора с аппаратным ускорением, основанного на OpenGL или Direct3D, значительная часть работы перекладывается на аппаратный ускоритель.

Стадия 6. Синхронизация экрана

Скорость, с которой компьютер исполняет игровой код, зависит от уровня сложности игры. Например, если на экране находится 1000 движущихся объектов, нагрузка на центральный процессор будет значительно больше, чем в случае только с 10 объектами. Следовательно, частота кадров в игре будет меняться, что неприемлемо. Поэтому нужно обеспечить постоянную частоту кадров в игре с помощью функций для работы со временем и функций ожидания. В настоящее время 30 fps (кадров в секунду, frame per second) считается минимально допустимым значением, а 60 fps — идеальным. Частоты выше 60 fps вряд ли оправданны, поскольку мозг с трудом обрабатывает информацию, поступающую с такой высокой скоростью.

НА ЗАМЕТКУ

Хотя в ряде игр и достигается идеальная скорость 30-60 кадров в секунду, она может упасть с увеличением уровня сложности визуализации. Тем не менее, с помощью контролируемых по времени расчетов в разделах игровой логики, физики и искусственного интеллекта можно, по крайней мере, попытаться обеспечить согласование сцены во времени, т.е., например, при меньшей частоте обновления кадров объекты будут проходить большее расстояние между двумя кадрами.

Стадия 7. Начало нового цикла

Эта стадия довольно проста — всего лишь возврат к началу игрового цикла и повторение его заново.

Стадия 8. Завершение работы

Это конец игры, в том смысле, что пользователь выходит из основного раздела кода или игрового цикла и желает возвратиться в операционную систему. Однако, прежде чем сделать это, необходимо освободить все ресурсы и очистить систему так же, как это делается в случае исполнения любой программы.

Следует отметить, что приведенное объяснение немного упрощенное, однако оно верно отображает суть происходящего. В действительности игровой цикл в большинстве случаев является конечным автоматом, содержащим ряд состояний. Например, ниже приведен более подробный код, иллюстрирующий, как может выглядеть игровой цикл в реальном C/C++ коде.

```
// Состояния цикла игры
#define GAME_INIT    // Инициализация игры
#define GAME_MENU    // Режим меню
#define GAME_START   // Подготовка к запуску
#define GAME_RUN     // Работа игры запущена
#define GAME_RESTART // Подготовка к перезапуску
#define GAME_EXIT    // Выход из игры
```

```
// Глобальные переменные игры
int game_state = GAME_INIT; // Начинаем с этого состояния
int error   = 0;          // Используется для возврата
                    // сообщения об ошибке
                    // операционной системе


// Основная программа - Функция main
void main()
{
  // Реализация основного цикла игры
  while (game_state!=GAME_EXIT)
  {
    // Состояние игры
    switch(game_state)
    {
      case GAME_INIT: // Инициализация игры
        {
          // Выделение памяти и ресурсов
          Init();
          // Возврат в состояние меню
          game_state = GAME_MENU;
        } break;

      case GAME_MENU: // Режим меню
        {
          // Вызов основного меню и изменение
          // состояния игры
          game_state = Menu();
          // Примечание: здесь можно перейти в
          // состояние RUN
        } break;

      case GAME_START:  // Игра готова к запуску
        {
          // Это состояние необязательно, но
          // обычно оно используется для настройки
          // состояния готовности. Здесь можно
          // заняться вспомогательными мероприятиями
          Setup_For_Run();

          // Переключаемся в состояние запуска
          game_state = GAME_RUN;
        } break;

      case GAME_RUN:  // Игра запущена
        {
          // Здесь содержится полный цикл
          // логики игры

          // Очистка экрана
          Clear();

          // Получение входных данных
```

```
            Get_Input();

            // Выполнение программы логики и
            // искусственного интеллекта
            Do_Logic();

            // Отображение следующего кадра анимации,
            // визуализация двумерного или
            // трехмерного мира
            Render_Frame();

            // Стабилизируем частоту вывода
            Wait();

            // Единственный способ изменить
            // состояние - это взаимодействие с
            // пользователем в разделе ввода данных
            // или, возможно, прерывание игры
          } break;

        case GAME_RESTART: // Перезапуск игры
          {
            // Здесь выполняется очистка для решения
            // всех проблем для нового запуска игры
            Fixup();
            // Возврат в меню
            game_state = GAME_MENU;
          } break;

        case GAME_EXIT: // Выход из игры
          {
            // Если игра находится в этом состоянии,
            // пришло время освободить ресурсы и
            // выйти из игры
            Release_And_Cleanup();

            // Устанавливаем код ошибки
            error = 0;

            // Примечание: переключать состояние не
            // требуется, т.к. на следующей стадии
            // код выходит из основного цикла
            // и возвращается в ОС
          } break;

        default: break;
      } // switch

    } // while

    // Возврат кода ошибки операционной системе
    return(error);

} // main
```

Хотя этот код нефункционален, я думаю, что изучив его, вы поймете идею структуры реального цикла игры. Все циклы игр — не имеет значения, трехмерных или двумерных, — довольно точно повторяют приведенную структуру. На рис. 1.2 показана диаграмма переходов между состояниями игрового цикла. Как видно, переходы между состояниями довольно последовательны.

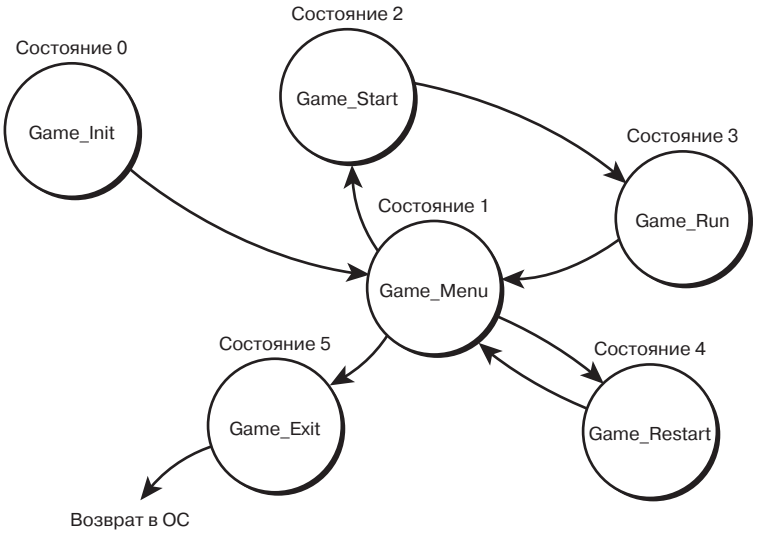


Рис. 1.2. Диаграмма переходов между логическими состояниями цикла игры

Общие советы по программированию игр

Следующее, о чем я хочу поговорить, — это общепринятые методы и философия программирования игр, о которых вам следует подумать и которые следует принять (если, конечно, вы это сможете). Они значительно облегчают процесс программирования.

Начнем с того, что видеоигры — это компьютерные программы, требующие высокой производительности компьютера. Это означает, что для разделов кода с высоким требованием к машинному времени и памяти больше нельзя использовать высокоуровневые API. В большинстве случаев следует самому писать все, что касается внутреннего цикла игрового кода, иначе игра будет работать ужасающе медленно. Конечно, это не означает, что нельзя полагаться на такие API, как DirectX, поскольку DirectX специально написан так, чтобы быть максимально быстрым при минимальном размере кода. Но в целом следует избегать вызовов высокоуровневых функций Win32 API. Например, вы можете решить, что функция memset() достаточно быстрая, однако она заполняет память побайтово. Значительно лучше использовать версию функции, которая заполняет память сразу по четыре байта (одно слово). Вот пример функции на встроенном ассемблере, которую я использую для заполнения памяти по четыре байта.

```
inline void Mem_Set_QUAD(void *dest, UINT data, int count)
{
  // Заполнение памяти по 32 бита. count— размер памяти
  // в четырехбайтовых словах
```

```
  _asm
  {
    mov edi, dest  ; edi указывает адрес памяти
    mov ecx, count ; Количество 32-битовых слов
    mov eax, data  ; 32-битовые данные
    rep stosd      ; Перенос данных
  }  // asm

}  // Mem_Set_QUAD
```

А вот версия для двухбайтового слова.

```
inline void Mem_Set_WORD(void *dest, USHORT data, int count)
{
  // Заполнение памяти по 16 битов. count— размер памяти
  // в двухбайтовых словах

  _asm
  {
    mov edi, dest  ; edi указывает адрес памяти
    mov ecx, count ; Количество 16-битовых слов
    mov ax, data   ; 16-битовые данные
    rep stosw      ; Перенос данных
  }  // asm

}  // Mem_Set_WORD
```

Эти несколько строк кода могут в некоторых случаях ускорить игру в два или четыре раза! Поэтому вам обязательно следует знать, что находится внутри функции API, если вы собираетесь ее использовать.

НА ЗАМЕТКУ

Кроме того, Pentium III, 4 и последующие версии поддерживают SIMD-команды (Single Instruction Multiple Data — один поток команд и множество потоков данных), которые позволяют распараллеливать обработку простых математических операций, поэтому здесь открывается большое поле деятельности для оптимизации базовых математических операций, таких как векторная математика и умножение матриц. К этому мы вернемся позднее.

Давайте взглянем на перечень приемов, о которых следует помнить во время программирования.

СЕКРЕТ

Не бойтесь использовать глобальные переменные. Многие видеоигры не используют передачу параметров в критических по времени выполнения функциях, вместо этого применяя глобальные переменные. Рассмотрим, например, следующую функцию.

```
void Plot(int x, int y, int color)
{
  // Вывод точки на экран
  video_buffer[x + y*MEMORY_PITCH] = color;
} // Plot
```

В данном случае тело функции выполняется гораздо быстрее, чем ее вызов, вследствие необходимости внесения параметров в стек и снятия их оттуда. В такой ситуации более эффективным может оказаться создание соответствующих глобальных переменных и передача информации в функцию путем присвоения им соответствующих значений.

```
int gx, gy, gcolor; // Глобальные переменные

void Plot_G(void)
{
  // Вывод точки на экран
  video_buffer[gx + gy*MEMORY_PITCH] = gcolor;
} // Plot_G
```

СЕКРЕТ

Используйте встраиваемые функции. Предыдущий фрагмент можно еще больше улучшить с помощью директивы inline, которая полностью устраняет код вызова функции, указывая компилятору на необходимость размещения кода функции непосредственно в месте ее вызова. Конечно, это несколько увеличивает программу, но скорость для нас гораздо важнее[1].

```
inline void Plot_I(int x, int y, int color)
{
  // Вывод точки на экран
  video_buffer[x + y*MEMORY_PITCH] = color;
} // Plot_I
```

Заметим, что здесь не используются глобальные переменные, поскольку компилятор сам заботится о том, чтобы для передачи параметров не использовался стек. Тем не менее, глобальные переменные могут пригодиться, если между вызовами изменяется только один или два параметра, поскольку при этом не придется заново загружать старые значения.

СЕКРЕТ

Всегда используйте 32-битовые переменные вместо 8- или 16-битовых. Pentium и более поздние процессоры — 32-битовые, а это означает, что они хуже работают со словами данных размером 8 и 16 битов и их использование может замедлить работу в связи с эффектами кэширования и другими эффектами, связанными с адресацией памяти. Например, вы можете создать структуру, которая выглядит примерно следующим образом.

```
struct CPOINT
{
  short x, y;
  unsigned char c;
} // CPOINT
```

Хотя использование такой структуры может показаться стоящей идеей, на самом деле это вовсе не так! Эта структура имеет размер 5 байтов: (2*sizeof(short)+sizeof(unsigned char)) = 5. Это не очень хорошо в силу особенностей адресации памяти у 32-битовых процессоров. Гораздо лучше использовать следующую структуру.

```
struct CPOINT
{
  int x, y;
  int c;
} // CPOINT
```

Такая структура гораздо лучше. Все ее элементы имеют одинаковый размер — 4 байта, а следовательно, все они выровнены в памяти на границу DWORD. Несмотря на выросший размер данной структуры, работа с ней осуществляется эффективнее, чем с предыдущей.

[1] Теоретически возможна ситуация, когда из-за использования встроенных функций внутренний цикл может вырасти и перестать полностью помещаться в кэше процессора, что приведет к снижению производительности по сравнению с использованием обычных функций. Злоупотреблять встроенными функциями не следует, и не только по этой причине. — *Прим. ред.*

В действительности вы можете доводить размер всех своих структур до величины, кратной 32 байтам, поскольку это оптимальный размер для стандартного кэша процессоров класса Pentium. Довести размер до этого значения можно путем добавления дополнительных искусственных членов структур либо посредством соответствующих директив компилятора. Конечно же, такой рост размера структур приведет к перерасходу памяти, но он может оказаться оправданным увеличением скорости работы.

C++

struct в C++ представляет собой аналог class, у которого все члены по умолчанию открыты (public).

СЕКРЕТ

Тщательно комментируйте ваш код. Программисты, работающие над играми, пользуются дурной славой в связи с тем, что не комментируют свой код. Не повторяйте их ошибку. Ясный, хорошо комментированный код стоит лишней работы с клавиатурой.

СЕКРЕТ

Программируйте в стиле RISC. Другими словами, делайте ваш код как можно более простым. В частности, в силу особенностей архитектуры процессоры класса Pentium предпочитают простые инструкции сложным, да и компилятору работать с ними и оптимизировать их легче. Например, вместо кода

```
if ( (x += (2*buffer[index++])) > 10 )
{
   // Выполняем некоторые действия
} // if
```

используйте код попроще

```
x += 2*buffer[index];
index++;

if (x > 10)
{
   // Выполняем некоторые действия
} // if
```

На то есть две причины. Во-первых, такой стиль позволяет при отладке вставлять дополнительные точки останова между разными частями кода. Во-вторых, такой подход облегчает работу компилятора по оптимизации этого кода.

СЕКРЕТ

Вместо умножения целых чисел на степень двойки, используйте побитовый сдвиг. Поскольку данные в компьютере хранятся в двоичном виде, сдвиг влево или вправо эквивалентен, соответственно, умножению или делению, например:

```
int y_pos = 10;

// Умножаем y_pos на 64
y_pos = (y_pos << 6);  // 2^6 = 64

// Делим y_pos на 8
y_pos = (y_pos >> 3);  // (1/3)^3 = 1/8
```

Вы еще встретитесь с подобными советами в главе, посвященной оптимизации.

СЕКРЕТ

Используйте эффективные алгоритмы. Никакой ассемблерный код не сделает алгоритм $O(n^2)$ более быстрым. Лучше использовать более эффективные алгоритмы, чем пытаться оптимизировать никуда негодные.

СЕКРЕТ

Не оптимизируйте ваш код в процессе программирования. Обычно это просто пустая трата времени. Перед тем как приступать к оптимизации, завершите написание если не всей игры, то по крайней мере, основной ее части. Тем самым вы сохраните силы и сэкономите время. Только когда игра готова, наступает время для ее профилирования и поиска проблемных участков кода, которые следует оптимизировать.

СЕКРЕТ

Не используйте слишком сложные структуры данных для простых объектов. Не стоит использовать связанные списки для представления массива, количество элементов которого точно известно заранее, только в силу того, что такие списки — это очень круто. Программирование видеоигр на 90% состоит из работы с данными. Храните их в как можно более простом виде с тем, чтобы обеспечить как можно более быстрый и простой доступ к ним. Убедитесь, что используемые вами структуры данных наиболее подходят для решения ваших задач.

СЕКРЕТ

Разумно используйте C++. Не пытайтесь применять множественное наследование только потому, что вы знаете, как это делается. Используйте только те возможности, которые реально необходимы и результаты применения которых вы хорошо себе представляете.

СЕКРЕТ

Если вы видите, что зашли в тупик, остановитесь и без сожалений вернитесь назад. Лучше потерять 500 строк кода, чем получить совершенно неработоспособный проект.

СЕКРЕТ

Регулярно делайте резервные копии вашей работы. При работе с игровой программой достаточно легко восстановить какую-нибудь простую сортировку, но восстановить систему искусственного интеллекта — дело совсем другое.

СЕКРЕТ

Перед тем как приступить к созданию игры, четко организуйте свою работу. Используйте понятные и имеющие смысл имена файлов и каталогов, выберите и придерживайтесь последовательной системы именования переменных, постарайтесь разделить графические и звуковые данные по разным каталогам.

Использование инструментов

Раньше для написания видеоигр обычно не требовалось ничего, кроме текстового редактора и, возможно, кустарных графических и звуковых программ. Сегодня ситуация намного усложнилась. Для написания трехмерной игры необходим, как минимум, компилятор C/C++, программа создания двумерных изображений, программа обработки звука, а также какой-нибудь минимальный разработчик трехмерных объектов (если только вы не собираетесь вводить все 3D-модели как ASCII-данные). Кроме того, понадобится музыкальный секвенсор, если вы собираетесь использовать какие-либо MIDI-данные.

Давайте рассмотрим некоторые наиболее популярные программы и их возможности.

- **Компиляторы C/C++**. Для работы под Windows 9x/Me/XP/2000/NT нет лучшего компилятора, чем MS VC++ (рис. 1.3). Он делает все, что может вам потребоваться, и даже больше того. Генерируемые им `.EXE`-файлы обладают максимально быстрым кодом. Компилятор фирмы Borland также хорошо работает, однако имеет меньше возможностей. В любом случае вам не нужна полная версия — достаточно студенческой версии, способной генерировать Win32 `.EXE`-файлы.
- **Программы для двумерной растровой графики**. Сюда относятся программы для рисования, черчения и обработки изображений. Программы для рисования позволяют рисовать изображения попиксельно с помощью примитивов и редактировать их. Насколько мне известно, Paint Shop Pro фирмы JASC

(рис. 1.4) является лидером по показателю соотношения цены и производительности. Но все же несомненным фаворитом большинства компьютерных художников является Adobe Photoshop — он обладает гораздо большей мощностью, чем та, которая когда-либо может вам понадобиться.

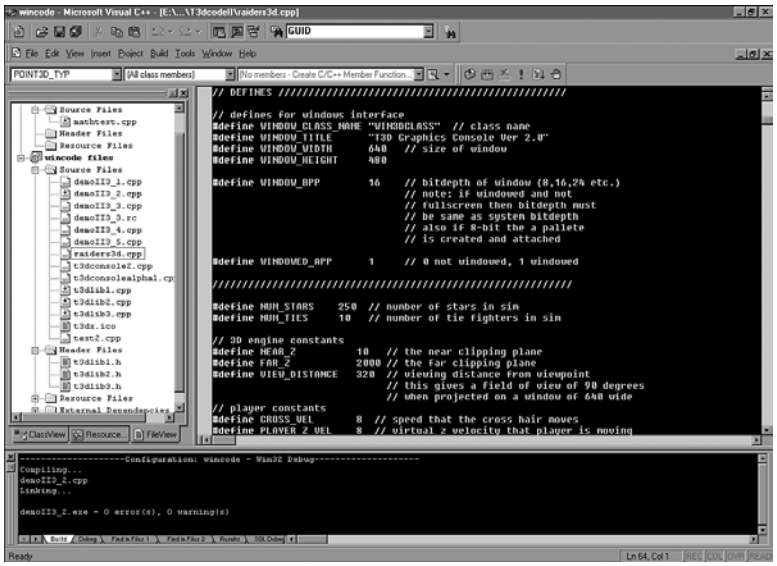


Рис. 1.3. Среда Microsoft VC++

- **Программы для двумерной векторной графики**. Эти программы позволяют создавать изображения, которые состоят в основном из кривых, прямых и двумерных геометрических примитивов. Этот тип программ не столь полезен, однако вам потребуется одна из них. Adobe Illustrator — это то, что вам нужно.
- **Программы для завершающей обработки изображений**. Редакторы изображений — это финальный класс программ для двумерной графики. Эти программы предназначены больше для завершающей обработки, чем для создания изображений. В этой области фаворитом многих пользователей является Adobe Photoshop, однако мне больше по душе Corel Photo-Paint. Решать вам.
- **Программы для обработки звука**. 90% всех звуковых эффектов, используемых в играх, — это оцифрованный звук. Для работы со звуковыми данными этого типа вам понадобится программа обработки звука. Одна из лучших программ этого рода — Sound Forge (рис. 1.5). Несомненно, это одна из самых сложных программ обработки звука. Я все еще открываю в ней новые и новые замечательные возможности! Еще одна довольно мощная программа — Cool Edit Pro, но с ней я работал меньше.
- **Программы разработки трехмерных объектов**. Здесь все зависит от ваших финансовых возможностей. Такие программы могут стоить десятки тысяч долларов. Тем не менее, опыт показывает, что последние версии не очень дорогих редакторов трехмерных объектов могут в буквальном смысле создавать кино. Для разработки простых и среднего уровня сложности трехмерных моделей я использую, главным образом, Caligari trueSpace (рис. 1.6). Это самый лучший 3D-редактор по такой цене — он стоит всего несколько сот долларов. Кроме того, я считаю, что у него самый удобный интерфейс.

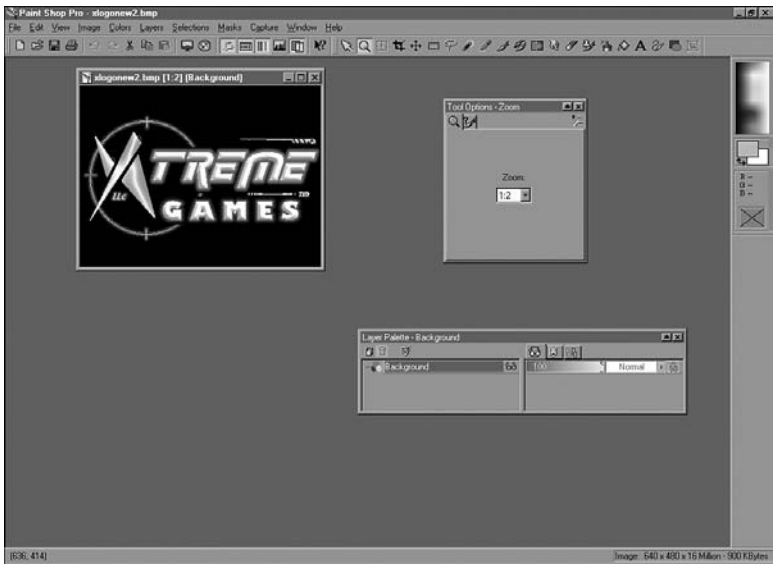


Рис. 1.4. Paint Shop Pro фирмы JASC Software

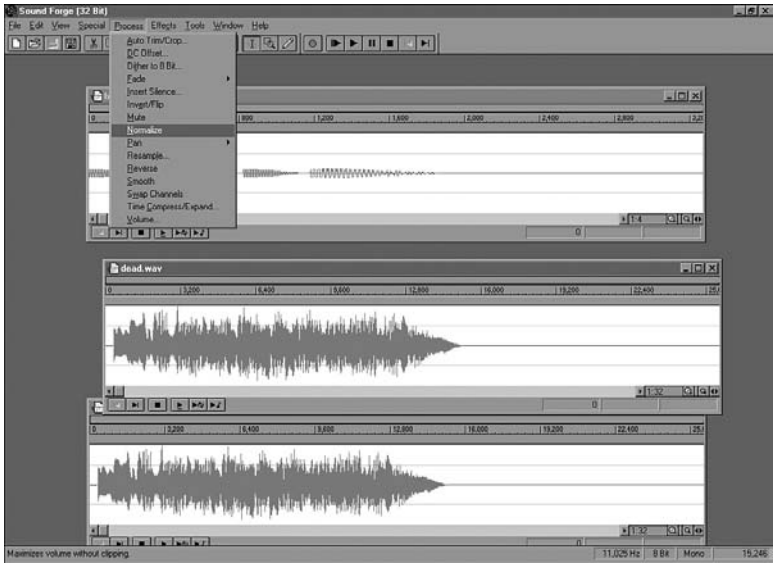


Рис. 1.5. Sound Forge в работе

Если же вам недостаточно этих возможностей и требуется полный фотореализм — используйте 3D Studio Max (рис. 1.7). Однако он стоит 2500 долл., так что тут есть о чем задуматься. В то же время, поскольку мы собираемся использовать эти программы в большинстве случаев лишь для создания трехмерных сеток, а не для визуализации, все эти "рюшечки и финтифлюшечки" нам не нужны, и наилучшим выбором будет trueSpace или, возможно, какая-то бесплатная или условно-бесплатная программа.

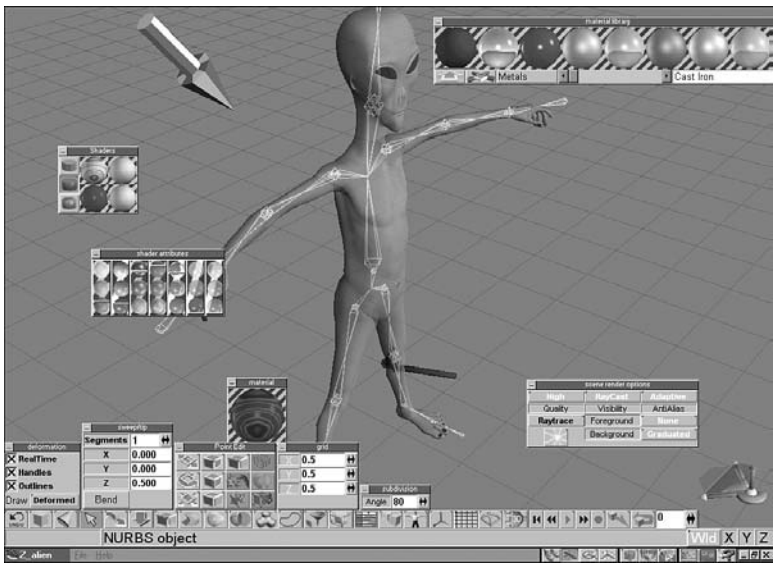


Рис. 1.6. Редактор Caligari trueSpace

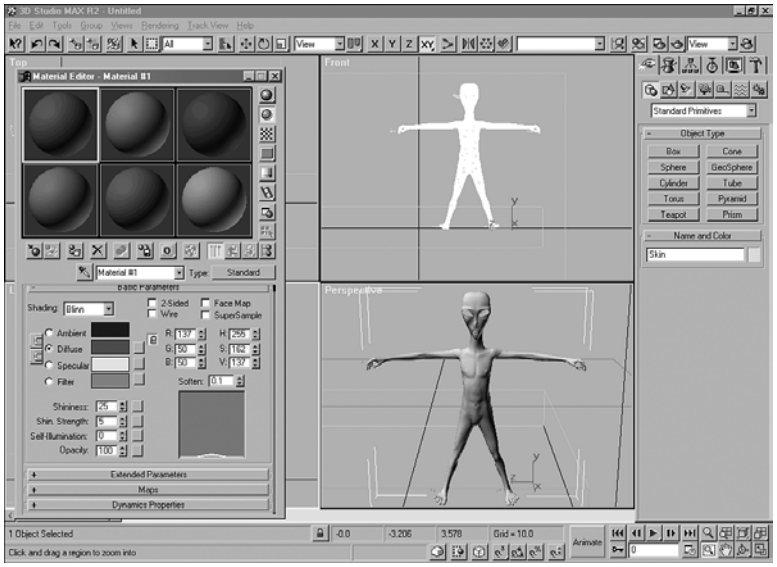


Рис. 1.7. Тяжеловес 3D Studio Max

НА ЗАМЕТКУ

В Internet можно найти множество бесплатных или условно-бесплатных 3D-редакторов, которые иногда имеют такие же возможности, как и коммерческий продукт. Поэтому, если денежные средства ограничены — ищите, и, возможно, вы найдете то, что нужно.

Редакторы трехмерных уровней

В этой книге мы собираемся создать программный трехмерный игровой процессор. Однако мы не намерены создавать сложные инструменты для моделирования трехмер-

ного интерьера. Трехмерный мир можно создать с помощью редактора 3D-объектов, однако есть программы, гораздо лучше приспособленные для этого, такие как WorldCraft (рис. 1.8). Следовательно, самый правильный подход при написании трехмерной игры и игрового процессора — это использовать файловый формат и структуру данных, совместимые с наиболее доступным из редакторов, таким как WorldCraft (или подобным). В этом случае для создания миров вы сможете воспользоваться инструментами сторонних разработчиков. Конечно, файловый формат большинства таких редакторов основан преимущественно на разработках фирмы id Software и игрового процессора Quake. Тем не менее, будет ли это файловый формат Quake или какой-либо иной, в любом случае очевидно, что этот формат проверен и работает прекрасно.

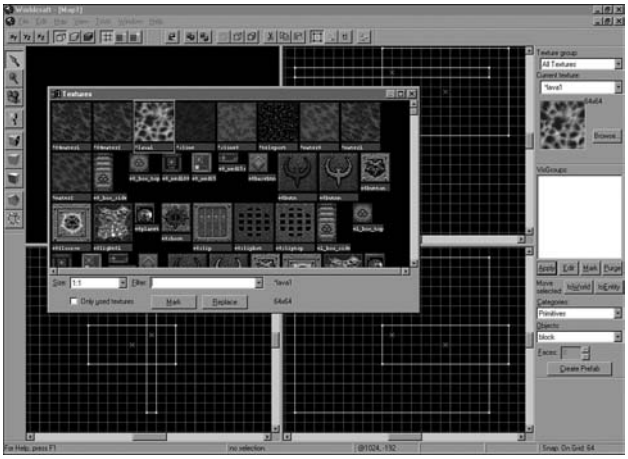


Рис. 1.8. Редактор уровней WorldCraft

- **Музыкальные программы и MIDI-секвенсоры**. В современных играх есть два типа музыки: цифровая (как на компакт-дисках) и MIDI-музыка (Musical Instruments Digital Interface — цифровой интерфейс музыкальных инструментов), представляющая собой синтетический объект, основанный на нотных данных. Для управления MIDI-данными необходим секвенсор. Один из лучших — CakeWalk (рис. 1.9). Одним из достоинств CakeWalk является приемлемая цена. Поэтому, если вы намерены работать с MIDI-музыкой, я весьма рекомендую познакомиться с этой программой. Мы поговорим о MIDI-данных, когда речь пойдет о DirectMusic.

НА ЗАМЕТКУ

А сейчас кое-то на десерт... Некоторые производители программ, упоминавшихся выше, разрешили мне записать на компакт-диск свои условно-бесплатные или ознакомительные версии. Ознакомьтесь с ними!

Использование компилятора

Один из наиболее напряженных моментов в изучении программирования трехмерных игр — это использование компилятора. В большинстве случаев вы настолько переполнены желанием начать работу, что входите в интегрированную среду разработки и запускаете компилятор. И получаете миллионы ошибок компилирования и связывания! Чтобы решить проблему, рассмотрим несколько основных принципов компилирования.

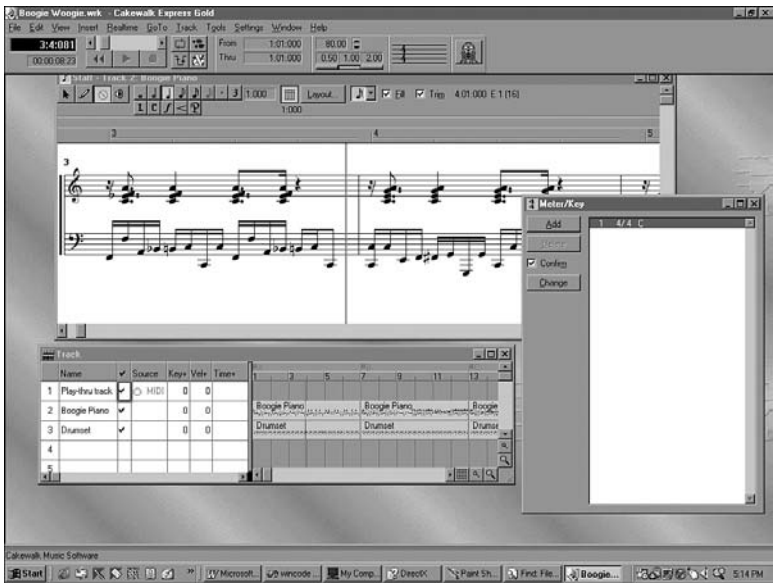


Рис. 1.9. Программа CakeWalk

НА ЗАМЕТКУ

Излагаемые здесь сведения относятся к Visual C++, однако на принципиальном уровне они касаются всех компиляторов.

1. Прочтите руководство пользователя компилятора полностью — пожалуйста, я умоляю вас!
2. Вы должны установить в системе DirectX 9.0 SDK. Чтобы сделать это, зайдите в каталог DirectX SDK на компакт-диске, прочтите `README.TXT` и выполните все, что там написано, т.е. щелкните на файле запуска установки DirectX SDK. И помните, в этой книге мы используем DirectX 9.0 (однако пользуемся интерфейсами версий 7 и 8), так что вам нужно использовать DirectX SDK с компакт-диска. Хотя — если вы действительно этого хотите — можете компилировать все с помощью DirectX 8.0.
3. Мы собираемся создавать Win32 `.EXE`-модули приложений, а не `DLL`, не компоненты ActiveX, не консольные приложения и не что-либо еще (если только я не скажу вам об этом отдельно). Поэтому первое, что нужно сделать, это создать новый проект и установить в качестве выходного файла Win32 `.EXE`. На рис. 1.10 показано, как сделать этот шаг в компиляторе Visual C++ 6.0.

НА ЗАМЕТКУ

Есть два типа `.EXE`-модулей, которые можно компилировать: `Release` и `Debug`. Окончательный (`Release`) обладает более быстрым и оптимизированным кодом, тогда как отладочный (`Debug`) медленнее и содержит контрольные точки отладчика. Советую во время разработки программы использовать `Debug`-версию, а затем, когда программа заработает, переключить компилятор в режим `Release`.

4. Для добавления исходных файлов в проект используйте команду Add Files из главного меню или из самого узла проекта. Этот шаг для Visual C++ 6.0 показан на рис. 1.11.

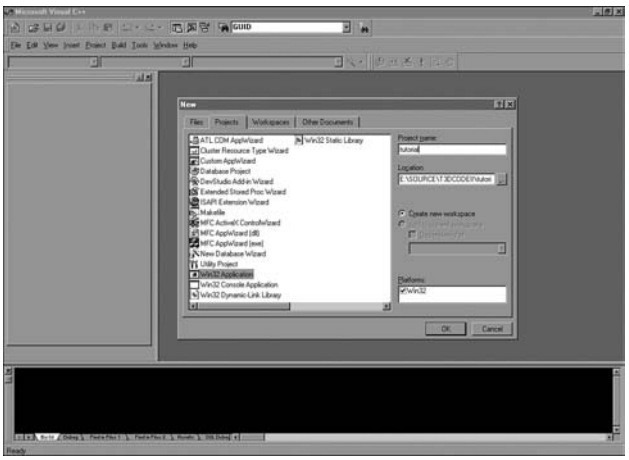


Рис. 1.10. Создание Win32 .EXE-модуля с помощью Visual C++ 6.0

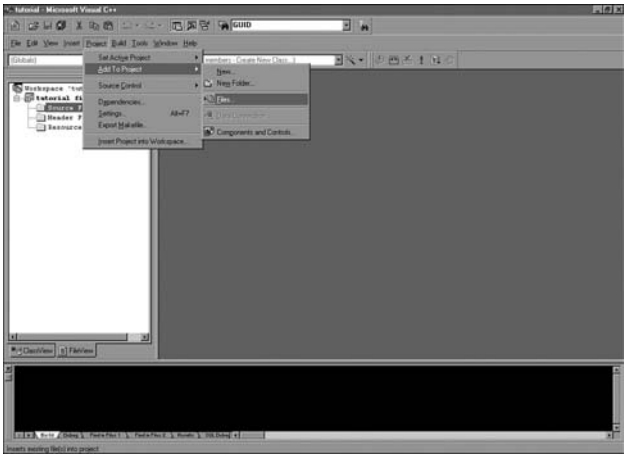


Рис. 1.11. Добавление файлов в проект в среде Visual C++ 6.0

5. Когда вы компилируете что-то, для чего нужен DirectX, необходимо указать в пути поиска файлов компилятором путь к месту нахождения заголовочных файлов DirectX, а также .LIB-файлов DirectX. Это можно сделать с помощью пункта меню Options, Directories в главном меню интегрированной среды разработки. Затем сделайте соответствующие добавления в переменные Include Files и Library Files. Этот шаг показан на рис. 1.12 (конечно, следует убедиться, что в них указан путь к конкретному месту инсталляции DirectX, а также новейших версий библиотек и заголовочных файлов).

НА ЗАМЕТКУ

Проверьте, чтобы узлы поиска DirectX были первыми в списке(ах). Вы же не хотите, чтобы компилятор нашел старые версии файлов DirectX, которые могли быть установлены вместе с компилятором?

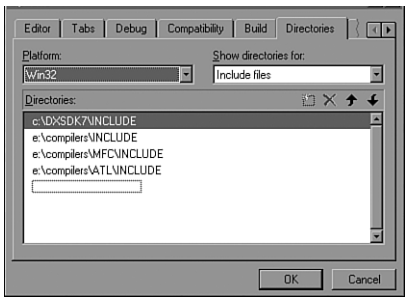


Рис. 1.12. Настройка пути поиска файлов в Visual C++ 6.0

6. Кроме того, вам обязательно нужно включить в проект библиотеки импорта COM-интерфейса DirectX, перечисленные ниже и показанные на рис. 1.13.
   - DDRAW.LIB
   - DSOUND.LIB
   - DINPUT.LIB
   - DINPUT8.LIB
   - А также другие, которые я упоминаю в конкретных примерах.

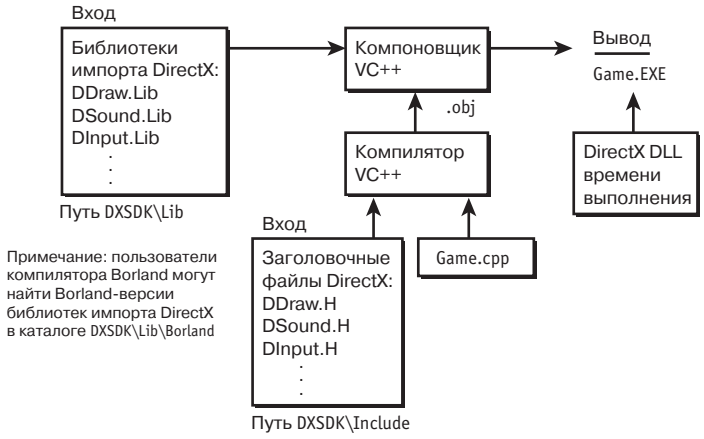


Рис. 1.13. Ресурсы, необходимые для создания приложения Win32 DirectX

Эти .LIB-файлы DirectX расположены в каталоге LIB\, находящемся там, куда был установлен DirectX SDK. Вам необходимо добавить эти .LIB-файлы в свой проект. Однако нельзя просто добавлять каталог LIB\ в путь поиска — тем самым компилятору/компоновщику указывается место поиска, но не говорится о том, что нужно использовать именно DirectX .LIB-файлы. Я получил тысячи (действительно тысячи!) электронных писем от людей, которые не сделали этого и получили проблемы. Повторю еще раз: вам необходимо вручную включить .LIB-файлы DirectX в список связывания компилятора вместе с другими библиотеками или в проект вместе с вашими .CPP-файлами. Это можно сделать в подменю Project, Settings диалогового окна Link, General в списке Object/Library-Modules (рис. 1.14).

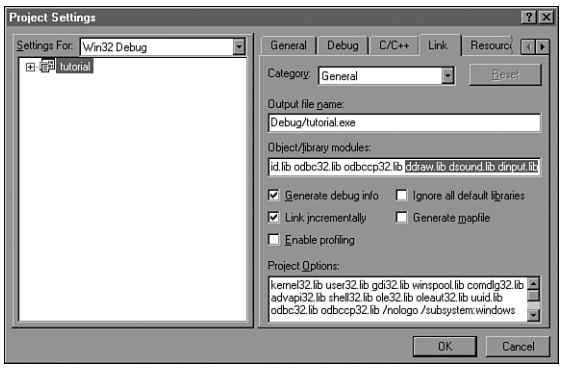


Рис. 1.14. Добавление .LIB-файлов DirectX в список связывания Visual C++ 6.0

НА ЗАМЕТКУ

Если вы используете Visual C++, можете добавить в проект библиотеку расширений Multimedia Windows (`WINMM.LIB`). Этот файл находится в каталоге `LIB\` в месте установки компилятора Visual C++. Если его там нет, воспользуйтесь командой Find меню Start. Найдя файл, добавьте его в список связывания.

7. Теперь вы готовы к компиляции программ.

НА ЗАМЕТКУ

Для пользователей Borland в DirectX SDK есть каталог `BORLAND\`. Убедитесь, что добавлены именно эти `.LIB`-файлы, а не файлы DirectX, установленные вместе с Visual C++.

Если у вас есть еще вопросы по этой теме, не волнуйтесь — на протяжении книги при компилировании программ я буду возвращаться к этим операциям много раз. Однако если я получу письмо с вопросами по компилированию и меня будут спрашивать о том, о чем только что говорилось, я за себя не ручаюсь!

НА ЗАМЕТКУ

Я уверен, что вы слышали о новой системе Visual .NET. Технически это компилятор и технология программирования Microsoft под новой маркой. Система нормально работает только на Windows XP/2000 и избыточна для наших целей, поэтому мы остановимся на Visual C++ 6.0. Тем не менее, все работает и при использовании .NET: Win32 есть Win32.

Пример трехмерной игры: Raiders 3D

Прежде чем у нас начнут плавиться мозги от разговоров о математике, программировании трехмерных игр и графике, я бы хотел сделать паузу и показать уже готовую трехмерную космическую игру — простую, но все же игру. При этом вы увидите, что такое настоящий игровой цикл, некоторые вызовы графических функций и компиляция. Звучит неплохо, не правда ли?

Проблема состоит в том, что это только первая глава, поэтому я не могу использовать материал из последующих глав — это было бы нечестно, согласны? И я еще не показал вам игровой процессор из предыдущей книги. Поэтому я решил приучить вас пользоваться для программирования игр API как “черным ящиком”. И эта простая игра позволит вам привыкнуть к идее использования такого API, как DirectX и игрового процессора из первой книги.

НА ЗАМЕТКУ

Хотя в 70-х, 80-х годах и в начале 90-х годов прошлого столетия можно было все делать самому, в 21 веке в компьютере функционирует слишком много подсистем оборудования и программного обеспечения, что делает практически невозможным написание всего кода одним человеком. Увы, мы всего лишь разработчики игр, и наш удел — использовать чужие API, такие как Win32, DirectX и т.п.

Основываясь на принципе "черного ящика", я поставлю вопрос так: каков абсолютный минимум средств для создания 16-разрядной трехмерной каркасной космической игры?

Все, что нам необходимо от API — это следующая функциональность:

- переключение в любой графический режим с помощью DirectX;
- рисование цветных линий и пикселей на экране;
- получение ввода с клавиатуры;
- проигрывание звуковой записи из .WAV-файла, находящегося на диске;
- проигрывание MIDI-музыки из .MID-файла, находящегося на диске;
- синхронизация игрового цикла с помощью функций работы со временем;
- вывод на экран строк цветного текста;
- копирование на экран двойного буфера (внеэкранной страницы визуализации).

Вся эта функциональность, конечно, содержится в библиотеке игровых модулей T3DLIB*, созданной в предыдущей книге и состоящей из шести файлов: T3DLIB1.CPP¦H, T3DLIB2.CPP¦H, и T3DLIB3.CPP¦H. Модуль 1 предназначен для базовой работы с DirectX, модуль 2 — с DirectInput и DirectSound, а модуль 3 — в основном с DirectMusic.

Основываясь на использовании минимального набора функций из игровой библиотеки T3DLIB*, я написал игру под названием Raiders 3D, которая демонстрирует ряд концепций, уже обсуждавшихся в данной главе. Кроме того, поскольку это каркасно-трехмерная игра, массу кода вообще не имеет смысла писать — и это хорошо! Хотя я буду кратко объяснять используемые математику и алгоритмы, не тратьте много времени, пытаясь понять, что к чему: эта информация дается только для того, чтобы позволить шире взглянуть на вещи.

Raiders 3D иллюстрирует все основные компоненты реальной трехмерной игры, включая игровой цикл, вычисления, искусственный интеллект, определение столкновений, звук и музыку. На рис. 1.15 показана копия экрана работающей игры. Конечно, это не Star Wars, однако совсем неплохо для нескольких часов работы!

Прежде чем показать исходный код игры, я хочу, чтобы вы получили представление о том, из каких компонентов состоит проект (рис. 1.16). Как видно из рисунка, игра состоит из следующих файлов, являющихся частью проекта:

- RAIDERS3D.CPP — основной блок логики игры, который использует функциональность T3DLIB и создает минимальное Win32-приложение;
- T3DLIB1.CPP — исходные файлы библиотеки T3DLIB;
- T3DLIB2.CPP;
- T3DLIB3.CPP;
- T3DLIB1.H — заголовочные файлы библиотеки;
- T3DLIB2.H;
- T3DLIB3.H;
- DDRAW.LIB — DirectDraw — компонент двумерной графики в составе DirectX. Для создания приложения требуется библиотека импорта DDRAW.LIB. Она не содержит

код DirectX; это промежуточная библиотека, обеспечивающая обращение к динамической библиотеке DDRAW.DLL, которая выполняет реальную работу. Этот файл можно найти в DirectX SDK в каталоге LIB\;

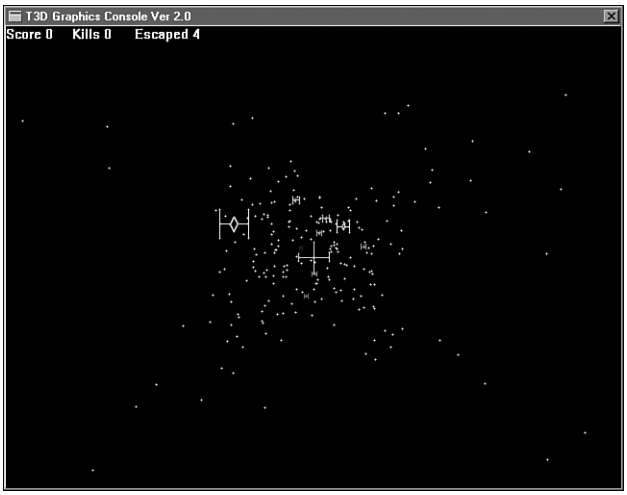


Рис. 1.15. Копия экрана Raiders 3D

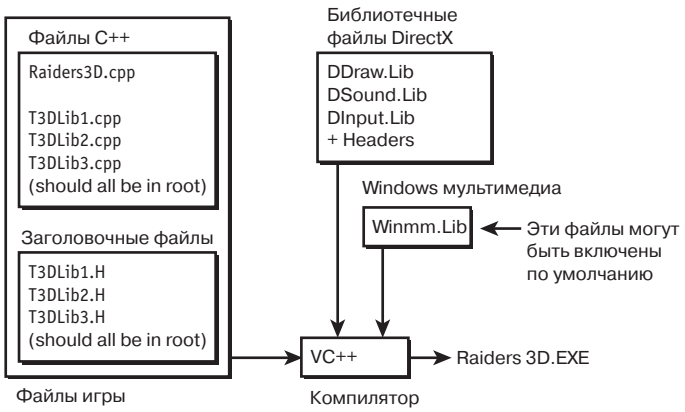


Рис. 1.16. Структура кода Raiders 3D

- DINPUT.LIB/DINPUT8.LIB — DirectInput — компонент пользовательского ввода в составе DirectX. Для создания приложения требуются библиотеки импорта DINPUT.LIB и DINPUT8.LIB;
- DSOUND.LIB — DirectSound — компонент цифрового звука в составе DirectX. Для создания приложения требуется библиотека импорта DSOUND.LIB.

НА ЗАМЕТКУ

Отметим, что не существует файла DMUSIC.LIB, хотя DirectMusic используется в T3DLIB. Это так, поскольку DirectMusic — это чистый COM-объект. Иными словами, нет библиотеки импорта, содержащей интерфейсные функции для обращения к DirectMusic — вам нужно все делать самому. К счастью, я уже сделал этот для вас!

Следующие файлы не нужны компилятору или компоновщику, но они являются выполняемыми DLL-файлами DirectX, которые загружаются при запуске игрового приложения:

- DINPUT.DLL/DINPUT8.DLL — это динамически компонуемые библиотеки DirectDraw, которые содержат COM-реализацию функций интерфейса DirectInput, вызываемых через библиотеку импорта DINPUT.LIB. Здесь вам не нужно ни о чем беспокоиться, проследите лишь, чтобы были установлены исполняемые файлы DirectX;
- DDRAW.DLL — это динамически компонуемая библиотека DirectDraw, которая содержит COM-реализацию интерфейсных функций DirectDraw, вызываемых через библиотеку импорта DDRAW.LIB;
- DSOUND.DLL — это динамически компонуемая библиотека DirectSound, которая содержит COM-реализацию интерфейсных функций DirectSound, вызываемых через библиотеку импорта DSOUND.LIB;
- DMUSIC.DLL — это динамически компонуемая библиотека DirectMusic, которая содержит COM-реализацию интерфейсных функций DirectMusic, вызываемых непосредственно через COM-обращения.

Основываясь на нескольких вызовах библиотеки, я создал игру RAIDERS3D.CPP, показанную в нижеприведенном листинге. Игра запускается в оконном режиме с 16-разрядной графикой, поэтому проследите, чтобы экран находился в 16-битовом цветовом режиме.

Хорошо изучите представленный код: главный цикл игры, трехмерную математику, а также вызовы других функций.

```
// Raiders3D - RAIDERS3D.CPP - наша первая трехмерная игра
// ПРОЧТИТЕ ЭТО!
// Перед компиляцией убедитесь, что включили в список
// связывания проекта файлы DDRAW.LIB, DSOUND.LIB,
// DINPUT.LIB, WINMM.LIB, а также исходные модули
// T3DLIB1.CPP,T3DLIB2.CPP и T3DLIB3.CPP в проект!!!
// Включите заголовочные файлы T3DLIB1.H,T3DLIB2.H и
// T3DLIB3.H в рабочий каталог, с тем чтобы компилятор
// мог найти их.

// Для запуска игры убедитесь, что экран установлен в
// 16-битовый цветовой режим с разрешением 640x480 или выше

// INCLUDES ///////////////////////////////////////////////

#define INITGUID  // Гарантируем доступность COM-интерфейсов
          // Вместо этого можно подключить .LIB-файл
          // DXGUID.LIB

#define WIN32_LEAN_AND_MEAN

#include <windows.h> // Подключаем функциональность Windows
#include <windowsx.h>
#include <mmsystem.h>
#include <iostream.h> // Подключаем функциональность C/C++
#include <conio.h>
#include <stdlib.h>
```

```
#include <malloc.h>
#include <memory.h>
#include <string.h>
#include <stdarg.h>
#include <stdio.h>
#include <math.h>
#include <io.h>
#include <fcntl.h>

#include <ddraw.h>   // Подключаем DirectX
#include <dsound.h>
#include <dmksctrl.h>
#include <dmusici.h>
#include <dmusicc.h>
#include <dmusicf.h>
#include <dinput.h>
#include "T3DLIB1.h"  // Подключаем библиотеку T3D
#include "T3DLIB2.h"
#include "T3DLIB3.h"

// DEFINES ////////////////////////////////////////////////

// Определяем интерфейс windows
#define WINDOW_CLASS_NAME "WIN3DCLASS" // Имя класса
#define WINDOW_TITLE   "T3D Graphics Console Ver 2.0"
#define WINDOW_WIDTH   640          // Размер окна
#define WINDOW_HEIGHT   480

#define WINDOW_BPP    16    // Битовая глубина цвета окна
                  // (8,16,24 и т.д.)
// Примечание: если работа идет в окне и не используется
// полноэкранный режим работы, то битовая глубина цвета
// должна быть такой же, как в системе. То же самое в
// случае создания и подключения 8-разрядной палитры

#define WINDOWED_APP 1 // 0 - не оконный режим, 1 - оконный

////////////////////////////////////////////////////////////

#define NUM_STARS  250  // Число звезд в модели
#define NUM_TIES   10   // Число боевых кораблей в модели

// Константы 3D-игрового процессора
#define NEAR_Z     10    // Ближняя плоскость отсечения
#define FAR_Z      2000   // Дальняя плоскость отсечения
#define VIEW_DISTANCE 320 // Расстояние видимости для данной
                  // точки обзора. Оно дает размер
                  // поля зрения при угле зрения 90°
                  // в случае проекции на окно
                  // шириной 640 пикселей
// Константы игрока
#define CROSS_VEL    8  // Скорость, с которой движется
                // перекрестие прицела
```

```
#define PLAYER_Z_VEL  8 // виртуальная z-скорость игрока
                        // для имитации перемещения

// Константы модели боевого корабля
#define NUM_TIE_VERTS   10
#define NUM_TIE_EDGES   8

// Данные о взрыве
#define NUM_EXPLOSIONS  (NUM_TIES) // Общее число взрывов

// Состояния игры
#define GAME_RUNNING   1
#define GAME_OVER      0

// ТИПЫ ///////////////////////////////////////////////

// Трехмерная точка
typedef struct POINT3D_TYP
{
  USHORT color;   // 16-битовый цвет точки
  float x,y,z;    // Координаты точки
} POINT3D, *POINT3D_PTR;

// 3D-линия, два индекса в списке вершин
typedef struct LINE3D_TYP
{
  USHORT color; // 16-битовый цвет линии
  int v1,v2;    // Индексы конечных точек в списке вершин

} LINE3D, *LINE3D_PTR;

// Боевой корабль
typedef struct TIE_TYP
{
  int state;      // Состояние боевого корабля:
                  // 0=мертв, 1=жив
  float x, y, z;  // Координаты корабля
  float xv,yv,zv; // Скорость корабля
} TIE, *TIE_PTR;

// Базовый 3D-вектор, используемый для скорости
typedef struct VEC3D_TYP
{
  float x,y,z;    // Координаты вектора
} VEC3D, *VEC3D_PTR;

// Каркасный взрыв
typedef struct EXPL_TYP
{
  int state;      // Состояние взрыва
  int counter;    // Счетчик взрывов
  USHORT color;   // Цвет взрыва
```

```
   // Взрыв - это совокупность ребер/линий,
   // основанная на модели взрывающегося корабля
   POINT3D p1[NUM_TIE_EDGES]; // Начальная точка ребра n
   POINT3D p2[NUM_TIE_EDGES]; // Конечная точка ребра n
   VEC3D  vel[NUM_TIE_EDGES]; // Скорость осколков

} EXPL, *EXPL_PTR;

// ПРОТОТИПЫ ////////////////////////////////////////////

// Консоль игры
int Game_Init(void *parms=NULL);
int Game_Shutdown(void *parms=NULL);
int Game_Main(void *parms=NULL);

// Функции игры
void Init_Tie(int index);

// ГЛОБАЛЬНЫЕ ПЕРЕМЕННЫЕ ////////////////////////////////

HWND main_window_handle = NULL; // Дескриптор окна
HINSTANCE main_instance = NULL; // Сохраняем экземпляр
char buffer[256];              // Используется для
                        // вывода текста

// Корабль - это совокупность вершин, соединенных
// линиями, которые образуют форму

POINT3D tie_vlist[NUM_TIE_VERTS]; // Список вершин модели
                        // боевого корабля
LINE3D tie_shape[NUM_TIE_EDGES];  // Список ребер корабля
TIE    ties[NUM_TIES];          // Боевые корабли

POINT3D stars[NUM_STARS];       // Звездное поле

// Некоторые цвета мы не можем создать, пока не знаем формат
// цвета — 5.5.5 или 5.6.5, поэтому мы подождем немного
// и сделаем это в функции Game_Init()
USHORT rgb_green,
     rgb_white,
     rgb_red,
     rgb_blue;

// Переменные игрока
float cross_x = 0, // Перекрестие прицела
    cross_y = 0;

int cross_x_screen  = WINDOW_WIDTH/2, // Перекрестие прицела
   cross_y_screen  = WINDOW_HEIGHT/2,
   target_x_screen = WINDOW_WIDTH/2,
   target_y_screen = WINDOW_HEIGHT/2;

int player_z_vel = 4; // Виртуальная скорость
```

```
                // наблюдателя/корабля
int cannon_state = 0; // Состояние лазерной пушки
int cannon_count = 0; // Счетчик выстрелов из пушки

EXPL explosions[NUM_EXPLOSIONS]; // Взрывы

int misses = 0; // Число кораблей, которые
           // не удалось поразить
int hits  = 0;  // Число попаданий
int score = 0;  // Счет


// Музыкальные и звуковые данные
int main_track_id = -1, // Идентификатор музыкальной дорожки
   laser_id     = -1, // Звук лазерной вспышки
   explosion_id  = -1, // Звук взрыва
   flyby_id     = -1; // Звук пролетающего корабля

int game_state = GAME_RUNNING; // Состояние игры

// ФУНКЦИИ ////////////////////////////////////////////

LRESULT CALLBACK WindowProc(HWND hwnd,
                    UINT msg,
                    WPARAM wparam,
                    LPARAM lparam)
{
   // Обработчик сообщений системы
   PAINTSTRUCT  ps; // Используется в WM_PAINT
   HDC        hdc; // Контекст устройства

   // Какое сообщение получено?
   switch(msg)
   {
      case WM_CREATE:
         {
            // Инициализация данных
            return(0);
         } break;

      case WM_PAINT:
         {
            // Начало рисования
            hdc = BeginPaint(hwnd,&ps);

            // Окончание рисования
            EndPaint(hwnd,&ps);
            return(0);
         } break;

      case WM_DESTROY:
         {
            // Закрываем приложение
```

```
            PostQuitMessage(0);
            return(0);
          } break;

      default:break;

    } // switch

    // Обработка необработанных сообщений
    return (DefWindowProc(hwnd, msg, wparam, lparam));

} // WinProc

// WINMAIN ////////////////////////////////////////////

int WINAPI WinMain(  HINSTANCE hinstance,
                HINSTANCE hprevinstance,
                LPSTR lpcmdline,
                int ncmdshow)
{

    WNDCLASS winclass;  // Создаваемый класс
    HWND     hwnd;     // Обобщенный дескриптор окна
    MSG      msg;      // Обобщенное сообщение
    HDC      hdc;      // Дескриптор контекста устройства
    PAINTSTRUCT ps;    // Структура вывода

    // Заполняем структуру класса
    winclass.style       = CS_DBLCLKS | CS_OWNDC |
                     CS_HREDRAW | CS_VREDRAW;
    winclass.lpfnWndProc  = WindowProc;
    winclass.cbClsExtra   = 0;
    winclass.cbWndExtra   = 0;
    winclass.hInstance    = hinstance;
    winclass.hIcon        = LoadIcon(NULL, IDI_APPLICATION);
    winclass.hCursor      = LoadCursor(NULL, IDC_ARROW);
    winclass.hbrBackground=
              (HBRUSH)GetStockObject(BLACK_BRUSH);
    winclass.lpszMenuName = NULL;
    winclass.lpszClassName = WINDOW_CLASS_NAME;

    // Регистрируем класс окна
    if (!RegisterClass(&winclass))
      return(0);

    // Создаем окно. Обратите внимание на проверку
    // для выбора надлежащего флага окна
    if (!(hwnd = CreateWindow(WINDOW_CLASS_NAME,// Класс
                      WINDOW_TITLE,    // Заголовок
                      (WINDOWED_APP ?
                      (WS_OVERLAPPED | WS_SYSMENU) :
                      (WS_POPUP | WS_VISIBLE)),
                      0,0,           // x,y
```

```
                WINDOW_WIDTH,    // Ширина
                WINDOW_HEIGHT,   // Высота
                NULL,  // Дескриптор родителя
                NULL,  // Дескриптор меню
                hinstance,       // экземпляр
                NULL)))          // Параметры
  return(0);

// Сохраняем дескриптор и экземпляр
// окна в глобальной переменной
main_window_handle = hwnd;
main_instance     = hinstance;

// Изменим окно, чтобы клиентская область имела
// размер width x height
if (WINDOWED_APP)
{
  // Изменим размер окна так, чтобы клиентская область
  // имела именно тот размер, который указан в запросе,
  // поскольку, если приложение работает в окне, там
  // могут быть рамки и панели органов управления.
  // Если приложение работает не в окне, это не имеет
  // значения
  RECT window_rect = {0,0,WINDOW_WIDTH,WINDOW_HEIGHT};

  // Вызов для настройки window_rect
  AdjustWindowRectEx(&window_rect,
            GetWindowStyle(main_window_handle),
            GetMenu(main_window_handle) != NULL,
            GetWindowExStyle(main_window_handle));

  // Сохраним глобальные переменные смещения клиента,
  // необходимые для DDraw_Flip()
  window_client_x0 = -window_rect.left;
  window_client_y0 = -window_rect.top;

  // Изменим размер окна с помощью вызова MoveWindow()
  MoveWindow(main_window_handle,
        CW_USEDEFAULT, // Координата x
        CW_USEDEFAULT, // Координата y
        // Ширина и высота
        window_rect.right - window_rect.left,
        window_rect.bottom - window_rect.top,
        FALSE);

  // Выводим окно
  ShowWindow(main_window_handle, SW_SHOW);
} // if windowed

// Выполняем инициализацию параметров игровой консоли
Game_Init();

// Отключаем CTRL-ALT-DEL, ALT-TAB. Закомментируйте
```

```
  // эту строку, если она вызывает зависание системы
  SystemParametersInfo(SPI_SCREENSAVERRUNNING,TRUE,NULL,0);

  // Входим в главный цикл событий
  while(1)
  {
    if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
      // Проверяем, не следует ли выйти
      if (msg.message == WM_QUIT)
        break;

      // Транслируем горячие клавиши
      TranslateMessage(&msg);

      // Посылаем сообщение обработчику окна
      DispatchMessage(&msg);
    } // if

    // Основная работа игры выполняется здесь
    Game_Main();

  } // while

  // Выходим из игры и освобождаем все ресурсы
  Game_Shutdown();

  // Включаем CTRL-ALT-DEL, ALT-TAB, закомментируйте
  // эту строку, если она приводит к зависанию системы
  SystemParametersInfo(SPI_SCREENSAVERRUNNING,
                FALSE,NULL,0);

  // Возвращаемся в Windows
  return(msg.wParam);

} // WinMain

// ФУНКЦИИ КОНСОЛИ ИГРЫ T3D II ////////////////////////////

int Game_Init(void *parms)
{
  // Инициализация параметров игры

  int index;

  Open_Error_File("error.txt");
  // Запускаем DirectDraw (заменяем параметры по желанию)
  DDraw_Init(WINDOW_WIDTH, WINDOW_HEIGHT, WINDOW_BPP,
        WINDOWED_APP);

  // Инициализируем DirectInput
  DInput_Init();
```

```
// Запрашиваем клавиатуру
DInput_Init_Keyboard();

// Инициализируем DirectSound
DSound_Init();

// Загружаем звуковые файлы
explosion_id = DSound_Load_WAV("exp1.wav");
laser_id     = DSound_Load_WAV("shocker.wav");

// Инициализируем DirectMusic
DMusic_Init();

// Загружаем и запускаем музыкальный трек
main_track_id = DMusic_Load_MIDI("midifile2.mid");
DMusic_Play(main_track_id);

// Добавляем вызовы для ввода данных из других
// устройств DirectInput

// Прячем мышь
ShowCursor(FALSE);

// Инициализируем генератор случайных чисел
srand(Start_Clock());

// Создаем системные цвета
rgb_green = RGB16Bit(0,31,0);
rgb_white = RGB16Bit(31,31,31);
rgb_blue  = RGB16Bit(0,0,31);
rgb_red   = RGB16Bit(31,0,0);

// Создаем звездное поле
for (index=0; index < NUM_STARS; index++)
{
   // В беспорядке располагаем звезды в цилиндре,
   // вытянутом от точки наблюдателя (0,0,-d) в
   // направлении отсекающей плоскости (0,0,far_z)
   stars[index].x = -WINDOW_WIDTH/2 +
              rand()%WINDOW_WIDTH;
   stars[index].y = -WINDOW_HEIGHT/2 +
              rand()%WINDOW_HEIGHT;
   stars[index].z = NEAR_Z +
              rand()%(FAR_Z - NEAR_Z);

   // устанавливаем цвета звезд
   stars[index].color = rgb_white;
} // for index

// Создаем модель боевого корабля

// Список вершин боевого корабля
POINT3D temp_tie_vlist[NUM_TIE_VERTS] =
```

```
    // Цвет, x,y,z
  {
  {rgb_white,-40, 40,0},  // p0
  {rgb_white,-40,  0,0},  // p1
  {rgb_white,-40,-40,0},  // p2
  {rgb_white,-10,  0,0},  // p3
  {rgb_white,  0, 20,0},  // p4
  {rgb_white, 10,  0,0},  // p5
  {rgb_white,  0,-20,0},  // p6
  {rgb_white, 40, 40,0},  // p7
  {rgb_white, 40,  0,0},  // p8
  {rgb_white, 40,-40,0}}; // p9

  // Копируем модель в глобальные массивы
  for (index=0; index<NUM_TIE_VERTS; index++)
    tie_vlist[index] = temp_tie_vlist[index];

  // Список ребер боевого корабля
  LINE3D temp_tie_shape[NUM_TIE_EDGES] =
    // цвет, вершина 1, вершина 2
  {
  {rgb_green,0,2   },  // l0
  {rgb_green,1,3   },  // l1
  {rgb_green,3,4   },  // l2
  {rgb_green,4,5   },  // l3
  {rgb_green,5,6   },  // l4
  {rgb_green,6,3   },  // l5
  {rgb_green,5,8   },  // l6
  {rgb_green,7,9   } };// l7

  // Копируем модель в глобальные массивы
  for (index=0; index<NUM_TIE_EDGES; index++)
    tie_shape[index] = temp_tie_shape[index];

  // Инициализируем положение каждого боевого
  // корабля и его скорость
  for (index=0; index<NUM_TIES; index++)
  {
    // Инициализируем корабль
    Init_Tie(index);
  } // for index

  // Возвращаем код успешного завершения
  return(1);

} // Game_Init

//////////////////////////////////////////////////////////

int Game_Shutdown(void *parms)
{
  // Эта функция выполняется при выходе из игры
  // и освобождении всех выделенных ресурсов
```

```
  // Завершаем все процессы

  // Освобождаем все ресурсы, выделенные для игры

  // Закрываем DirectSound
  DSound_Stop_All_Sounds();
  DSound_Shutdown();

  // DirectMusic
  DMusic_Delete_All_MIDI();
  DMusic_Shutdown();

  // DirectInput
  DInput_Shutdown();

  // DirectDraw завершаем последним
  DDraw_Shutdown();

  // Возвращаем код успешного завершения
  return(1);
} // Game_Shutdown

//////////////////////////////////////////////////////////

void Start_Explosion(int tie)
{
  // Взрыв, основанный на модели пораженного корабля

  // Можно ли произвести взрыв
  for (int index=0; index < NUM_EXPLOSIONS; index++)
  {
    if (explosions[index].state==0)
    {
      // Начинаем взрыв с учетом индекса корабля

      explosions[index].state  = 1; // Состояние взрыва
      explosions[index].counter = 0;// Сброс счетчика

      // Цвет взрыва
      explosions[index].color = rgb_green;

      // Делаем копию списка ребер
      for (int edge=0; edge < NUM_TIE_EDGES; edge++)
      {
        // Начальная точка ребра
        explosions[index].p1[edge].x = ties[tie].x +
              tie_vlist[tie_shape[edge].v1].x;
        explosions[index].p1[edge].y = ties[tie].y +
              tie_vlist[tie_shape[edge].v1].y;
        explosions[index].p1[edge].z = ties[tie].z +
              tie_vlist[tie_shape[edge].v1].z;
```

```
            // Конечная точка ребра
            explosions[index].p2[edge].x = ties[tie].x +
                    tie_vlist[tie_shape[edge].v2].x;
            explosions[index].p2[edge].y = ties[tie].y +
                    tie_vlist[tie_shape[edge].v2].y;
            explosions[index].p2[edge].z = ties[tie].z +
                    tie_vlist[tie_shape[edge].v2].z;

            // вычисляем траекторию вектора ребра
            explosions[index].vel[edge].x = ties[tie].xv-
                    8+rand()%16;
            explosions[index].vel[edge].y = ties[tie].yv-
                    8+rand()%16;
            explosions[index].vel[edge].z = -3+rand()%4;

          } // for edge

          return;
        } // if

    } // for index

} // Start_Explosion

///////////////////////////////////////////////////////////

void Process_Explosions(void)
{
    // Обрабатываются все взрывы

    // Цикл по всем взрывам и выполняем их визуализацию
    for (int index=0; index<NUM_EXPLOSIONS; index++)
    {
      // Проверяем, активен ли взрыв
      if (explosions[index].state==0)
         continue;

      for (int edge=0; edge<NUM_TIE_EDGES; edge++)
      {
        // Должен быть взрыв, обновляем ребра
        explosions[index].p1[edge].x+=
             explosions[index].vel[edge].x;
        explosions[index].p1[edge].y+=
             explosions[index].vel[edge].y;
        explosions[index].p1[edge].z+=
             explosions[index].vel[edge].z;

        explosions[index].p2[edge].x+=
             explosions[index].vel[edge].x;
        explosions[index].p2[edge].y+=
             explosions[index].vel[edge].y;
        explosions[index].p2[edge].z+=
             explosions[index].vel[edge].z;
```

```
      } // for edge

      // Проверяем окончание взрыва
      if (++explosions[index].counter > 100)
         explosions[index].state =
               explosions[index].counter = 0;

   } // for index

} // Process_Explosions

///////////////////////////////////////////////////////////

void Draw_Explosions(void)
{
   // Этот блок кода рисует все взрывы

   // Цикл по всем взрывам и визуализация
   for (int index=0; index<NUM_EXPLOSIONS; index++)
   {
      // Проверяем, является ли взрыв активным
      if (explosions[index].state==0)
         continue;

      // Выполняем визуализацию взрыва
      // Каждый взрыв составлен из ряда ребер
      for (int edge=0; edge < NUM_TIE_EDGES; edge++)
      {
         POINT3D p1_per, p2_per;

         // Находится ли ребро вблизи плоскости отсечения
         if (explosions[index].p1[edge].z < NEAR_Z &&
            explosions[index].p2[edge].z < NEAR_Z)
            continue;

         // Шаг 1: аксонометрическое преобразование
         // каждой конечной точки
         p1_per.x = VIEW_DISTANCE*
               explosions[index].p1[edge].x/
               explosions[index].p1[edge].z;
         p1_per.y = VIEW_DISTANCE*
               explosions[index].p1[edge].y/
               explosions[index].p1[edge].z;
         p2_per.x = VIEW_DISTANCE*
               explosions[index].p2[edge].x/
               explosions[index].p2[edge].z;
         p2_per.y = VIEW_DISTANCE*
               explosions[index].p2[edge].y/
               explosions[index].p2[edge].z;

         // Шаг 2: вычисляем экранные координаты
         int p1_screen_x = WINDOW_WIDTH/2 + p1_per.x;
         int p1_screen_y = WINDOW_HEIGHT/2 - p1_per.y;
```

```
        int p2_screen_x = WINDOW_WIDTH/2 + p2_per.x;
        int p2_screen_y = WINDOW_HEIGHT/2 - p2_per.y;

        // Шаг 3: рисуем ребро
        Draw_Clip_Line16(p1_screen_x, p1_screen_y,
                  p2_screen_x, p2_screen_y,
                  explosions[index].color,
                  back_buffer, back_lpitch);

      } // for edge

   } // for index

} // Draw_Explosions

///////////////////////////////////////////////////////////

void Move_Starfield(void)
{
   // Перемещаем звезды

   int index;

   // Технически звезды неподвижны, но мы собираемся
   // перемещать их, чтобы имитировать движение наблюдателя
   for (index=0; index<NUM_STARS; index++)
   {
      // Движемся к следующей звезде
      stars[index].z-=player_z_vel;

      // Проверяем ближайшую плоскость отсечения
      if (stars[index].z <= NEAR_Z)
         stars[index].z = FAR_Z;

   } // for index

} // Move_Starfield

///////////////////////////////////////////////////////////

void Draw_Starfield(void)
{
   // Рисуем звезды трехмерными с помощью
   // аксонометрического преобразования

   int index;

   for (index=0; index<NUM_STARS; index++)
   {
      // Рисуем следующую звезду
      // Шаг 1: аксонометрическое преобразование
      float x_per = VIEW_DISTANCE*stars[index].x/
                    stars[index].z;
```

```
      float y_per = VIEW_DISTANCE*stars[index].y/
                    stars[index].z;

      // Шаг 2: вычисляем координаты экрана
      int x_screen = WINDOW_WIDTH/2 + x_per;
      int y_screen = WINDOW_HEIGHT/2 - y_per;

      // Отсекаем до экранных координат
      if (x_screen>=WINDOW_WIDTH || x_screen < 0 ||
         y_screen >= WINDOW_HEIGHT || y_screen < 0)
      {
         // Продолжаем движение к следующей звезде
         continue;
      } // if
      else
      {
         // Иначе выполняем визуализацию в буфер
         ((USHORT *)back_buffer)
            [x_screen + y_screen*(back_lpitch >> 1)]
               = stars[index].color;
      } // else

   } // for index

} // Draw_Starfield

///////////////////////////////////////////////////////////

void Init_Tie(int index)
{
   // Функция создает боевой корабль на дальнем конце
   // вселенной и посылает его к нам!

   // Располагаем каждый корабль в области обзора
   ties[index].x = -WINDOW_WIDTH + rand()%(2*WINDOW_WIDTH);
   ties[index].y = -WINDOW_HEIGHT+rand()%(2*WINDOW_HEIGHT);
   ties[index].z = 4*FAR_Z;

   // Инициализируем скорость боевого корабля
   ties[index].xv = -4+rand()%8;
   ties[index].yv = -4+rand()%8;
   ties[index].zv = -4-rand()%64;

   // Запускаем боевой корабль
   ties[index].state = 1;
} // Init_Tie

///////////////////////////////////////////////////////////

void Process_Ties(void)
{
   // Обрабатываем координаты боевых кораблей и выполняем
   // блок искусственного интеллекта
```

```
   int index;

   // Перемещаем каждый боевой корабль
   // в поле обзора наблюдателя
   for (index=0; index<NUM_TIES; index++)
   {
      // Может, он мертв?
      if (ties[index].state==0)
         continue;

      // Перемещаем
      ties[index].z+=ties[index].zv;
      ties[index].x+=ties[index].xv;
      ties[index].y+=ties[index].yv;

      // Проверяем ближайшую плоскость отсечения
      if (ties[index].z <= NEAR_Z)
      {
         // Переустанавливаем параметры данного корабля
         Init_Tie(index);
         misses++;

      }

   } // for index

} // Process_Ties

//////////////////////////////////////////////////////////

void Draw_Ties(void)
{
   // Рисуем боевые корабли в виде трехмерных каркасов

   int index;

   // Используется для вычисления ограничивающего
   // прямоугольника корабля для обнаружении попаданий
   int bmin_x, bmin_y, bmax_x, bmax_y;

   // Рисуем каждый боевой корабль
   for (index=0; index < NUM_TIES; index++)
   {
      // Рисуем следующий боевой корабль

      // Этот корабль мертв?
      if (ties[index].state==0)
         continue;

      // Ограничивающий прямоугольник становится
      // нереальным
      bmin_x = 100000;
      bmax_x = -100000;
```

```
bmin_y = 100000;
bmax_y = -100000;

// Основываясь на расстоянии по оси z, затеняем
// корабль, нормализуем расстояние от 0 до max_z,
// затем масштабируем (чем ближе, тем ярче)
USHORT rgb_tie_color =
   RGB16Bit(0,(31-31*(ties[index].z/(4*FAR_Z))),0);

// Каждый корабль состоит из ряда ребер
for (int edge=0; edge < NUM_TIE_EDGES; edge++)
{
   POINT3D p1_per, p2_per;
   // Шаг 1: аксонометрическое преобразование
   // каждой конечной точки. Заметим, что
   // трансляция каждой точки в положение боевого
   // корабля, который является моделью,
   // выполняется относительно положения корабля
   p1_per.x =
      VIEW_DISTANCE*(ties[index].x+
      tie_vlist[tie_shape[edge].v1].x)/
      (tie_vlist[tie_shape[edge].v1].z+
      ties[index].z);

   p1_per.y = VIEW_DISTANCE*(ties[index].y+
      tie_vlist[tie_shape[edge].v1].y)/
      (tie_vlist[tie_shape[edge].v1].z+
      ties[index].z);

   p2_per.x = VIEW_DISTANCE*(ties[index].x+
      tie_vlist[tie_shape[edge].v2].x)/
      (tie_vlist[tie_shape[edge].v2].z+
      ties[index].z);

   p2_per.y = VIEW_DISTANCE*(ties[index].y+
      tie_vlist[tie_shape[edge].v2].y)/
      (tie_vlist[tie_shape[edge].v2].z+
      ties[index].z);

   // Шаг 2: вычисляем экранные координаты
   int p1_screen_x = WINDOW_WIDTH/2 + p1_per.x;
   int p1_screen_y = WINDOW_HEIGHT/2 - p1_per.y;
   int p2_screen_x = WINDOW_WIDTH/2 + p2_per.x;
   int p2_screen_y = WINDOW_HEIGHT/2 - p2_per.y;

   // Шаг 3: рисуем ребро
   Draw_Clip_Line16(p1_screen_x, p1_screen_y,
               p2_screen_x, p2_screen_y,
               rgb_tie_color,back_buffer,
               back_lpitch);

   // Обновляем ограничивающий прямоугольник
   // следующим ребром
```

```
      int min_x = min(p1_screen_x, p2_screen_x);
      int max_x = max(p1_screen_x, p2_screen_x);

      int min_y = min(p1_screen_y, p2_screen_y);
      int max_y = max(p1_screen_y, p2_screen_y);

      bmin_x = min(bmin_x, min_x);
      bmin_y = min(bmin_y, min_y);

      bmax_x = max(bmax_x, max_x);
      bmax_y = max(bmax_y, max_y);

    } // for edge

    // Проверяем, был ли этот корабль поражен лазерами
    if (cannon_state==1)
    {
      // Простой тест экранных координат
      // ограничивающего прямоугольника, который
      // содержит лазерную цель
      if (target_x_screen > bmin_x &&
        target_x_screen < bmax_x &&
        target_y_screen > bmin_y &&
        target_y_screen < bmax_y)
      {
        // Этот корабль мертв!
        Start_Explosion(index);

        // Запускаем звук
        DSound_Play(explosion_id );

        // Увеличиваем счет
        score+=ties[index].z;

        // Добавляем одно очко
        hits++;

        // Последняя переустановка
        // параметров этого корабля
        Init_Tie(index);

      } // if

    } // if

  } // for index

} // Draw_Ties

//////////////////////////////////////////////////////////

int Game_Main(void *parms)
{
```

```
// Рабочий блок игры, он будет постоянно вызываться
// в режиме реального времени. Подобен функции main()
// в языке C

int index;

// Запускаем часы
Start_Clock();

// Очищаем поверхность рисования
DDraw_Fill_Surface(lpddsback, 0);

// Получаем ввод с клавиатуры и других устройств
DInput_Read_Keyboard();

// Блок логики игры...

if (game_state==GAME_RUNNING)
{
  // Перемещаем перекрестие прицела
  if (keyboard_state[DIK_RIGHT])
  {
    // Перемещаем перекрестие прицела вправо
    cross_x+=CROSS_VEL;

    if (cross_x > WINDOW_WIDTH/2)
      cross_x = -WINDOW_WIDTH/2;

  } // if
  if (keyboard_state[DIK_LEFT])
  {
    // Перемещаем перекрестие прицела влево
    cross_x-=CROSS_VEL;

    if (cross_x < -WINDOW_WIDTH/2)
      cross_x = WINDOW_WIDTH/2;
  } // if

  if (keyboard_state[DIK_DOWN])
  {
    // Перемещаем перекрестие прицела вниз
    cross_y-=CROSS_VEL;

    if (cross_y < -WINDOW_HEIGHT/2)
      cross_y = WINDOW_HEIGHT/2;
  } // if

  if (keyboard_state[DIK_UP])
  {
    // Перемещаем перекрестие прицела вверх
    cross_y+=CROSS_VEL;

    if (cross_y > WINDOW_HEIGHT/2)
```

```
        cross_y = -WINDOW_HEIGHT/2;
    } // if

    // Управление скоростью корабля
    if (keyboard_state[DIK_A])
       player_z_vel++;
    else
       if (keyboard_state[DIK_S])
          player_z_vel--;

    // Проверяем, стреляет ли игрок из лазерной пушки
    if (keyboard_state[DIK_SPACE] && cannon_state==0)
    {
       // Стреляем из пушки
       cannon_state = 1;
       cannon_count = 0;

       // Сохраняем последнее положение прицела
       target_x_screen = cross_x_screen;
       target_y_screen = cross_y_screen;

       // Выводим звук
       DSound_Play(laser_id);

    } // if

}

// Процесс работы пушки - простой конечный автомат:
// готовность-выстрел-охлаждение

// Фаза выстрела
if (cannon_state == 1)
   if (++cannon_count > 15)
      cannon_state = 2;

// фаза охлаждения
if (cannon_state == 2)
   if (++cannon_count > 20)
      cannon_state = 0;

// Перемещаем звездное поле
Move_Starfield();

// Перемещаем и выполняем блок искусственного
// интеллекта для кораблей
Process_Ties();

// Обработка взрывов
Process_Explosions();

// Закрываем задний буфер
DDraw_Lock_Back_Surface();
```

```
// Рисуем звездное поле
Draw_Starfield();

// Рисуем боевые корабли
Draw_Ties();

// Рисуем взрывы
Draw_Explosions();

// Рисуем перекрестие прицела
// Вначале вычисляем экранные координаты перекрестия
// прицела. Обратите внимание на знак по оси y
cross_x_screen = WINDOW_WIDTH/2 + cross_x;
cross_y_screen = WINDOW_HEIGHT/2 - cross_y;

// Рисуем перекрестие прицела в экранных координатах
Draw_Clip_Line16(cross_x_screen-16,cross_y_screen,
            cross_x_screen+16,cross_y_screen,
            rgb_red,back_buffer,back_lpitch);

Draw_Clip_Line16(cross_x_screen,cross_y_screen-16,
            cross_x_screen,cross_y_screen+16,
            rgb_red,back_buffer,back_lpitch);

Draw_Clip_Line16(cross_x_screen-16,cross_y_screen-4,
            cross_x_screen-16,cross_y_screen+4,
            rgb_red,back_buffer,back_lpitch);

Draw_Clip_Line16(cross_x_screen+16,cross_y_screen-4,
            cross_x_screen+16,cross_y_screen+4,
            rgb_red,back_buffer,back_lpitch);

// Рисуем лазерные лучи
if (cannon_state == 1)
{
  if ((rand()%2 == 1))
  {
    // Правый луч
    Draw_Clip_Line16(WINDOW_WIDTH-1,WINDOW_HEIGHT-1,
                -4+rand()%8+target_x_screen,
                -4+rand()%8+target_y_screen,
                RGB16Bit(0,0,rand()),
                back_buffer,back_lpitch);
  } // if
  else
  {
    // Левый луч
    Draw_Clip_Line16(0, WINDOW_HEIGHT-1,
                -4+rand()%8+target_x_screen,
                -4+rand()%8+target_y_screen,
                RGB16Bit(0,0,rand()),
                back_buffer,back_lpitch);
```

```
    } // if

  } // if

  // Визуализация завершена, открываем
  // поверхность заднего буфера
  DDraw_Unlock_Back_Surface();

  // Выводим текстовую информацию
  sprintf(buffer, "Score %d   Kills %d   Escaped %d",
        score, hits, misses);
  Draw_Text_GDI(buffer, 0,0,RGB(0,255,0), lpddsback);

  if (game_state==GAME_OVER)
    Draw_Text_GDI("G A M E O V E R", 320-8*10,240,
              RGB(255,255,255), lpddsback);

  // Проверяем, закончилась ли музыка,
  // если да - перезапускаем
  if (DMusic_Status_MIDI(main_track_id)==MIDI_STOPPED)
    DMusic_Play(main_track_id);

  // Меняем поверхности
  DDraw_Flip();

  // Синхронизация 30 кадров в секунду
  Wait_Clock(30);

  // Проверяем переключатель состояния игры
  if (misses > 100)
    game_state = GAME_OVER;

  // Проверяем, выходит ли пользователь из игры
  if (KEY_DOWN(VK_ESCAPE) || keyboard_state[DIK_ESCAPE])
  {
    PostMessage(main_window_handle, WM_DESTROY,0,0);

  } // if

  // Возвращаем код завершения
  return(1);
} // Game_Main

//////////////////////////////////////////////////////////
```

Очень немного для трехмерной игры, правда? Это реальная трехмерная игра Win32/DirectX.

Прежде чем мы займемся анализом кода, я хочу, чтобы вы сами скомпилировали его. Я запрещаю вам двигаться дальше, пока компиляция не завершится успехом! Надеюсь, я понятно выразился. Итак, займитесь настройкой компилятора в соответствии с описанными выше инструкциями для создания Win32 .EXE-приложений, указывая пути поиска и создавая списки связывания, настроенные для DirectX. Затем, когда проект будет готов, подключите исходные файлы T3DLIB1.CPP, T3DLIB2.CPP, T3DLIB3.CPP, RAIDERS3D.CPP.

Конечно же, заголовочные файлы T3DLIB1.H, T3DLIB2.H, T3DLIB3.H должны быть в рабочем каталоге компилятора. И наконец, необходимо быть абсолютно уверенным, что вы включили в проект .LIB-файлы DirectX вместе с .CPP-файлами или включили их в список связывания. Вам необходимы лишь следующие .LIB-файлы DirectX: DDRAW.LIB, DSOUND.LIB, DINPUT.LIB, DINPUT8.LIB.

Вы можете назвать .EXE-файл как угодно — возможно, TEST.EXE или RAIDERS3D_TEST.EXE — однако не идите дальше, пока вы не сможете его скомпилировать.

Цикл событий

Главная точка входа для всех программ Windows — функция WinMain(), точно так же, как main() — главная точка входа для программ DOS/UNIX. В любом случае WinMain() создает окно для Raiders3D и затем входит прямо в цикл событий. WinMain() начинает с создания и регистрации класса Windows. Затем создается окно игры, после чего производится вызов функции Game_Init(), которая выполняет инициализацию игры. После завершения инициализации выполняется вход в стандартный цикл событий Windows, который считывает сообщения. Если сообщение найдено, вызывается процедура Windows WinProc, которая обрабатывает его. В противном случае вызывается функция игры Game_Main(). Именно здесь происходит реальное действие игры.

НА ЗАМЕТКУ

Читатели предыдущей книги могут заметить, что в разделе инициализации функции WinMain() появился дополнительный код для обработки оконной графики и изменения размера окна. Эта возможность, а также поддержка 16-битового цвета являются частью новой версии игрового процессора T3DLIB. Тем не менее, большая часть кода в этой книге по-прежнему поддерживает 8-битовую графику, поскольку в общем случае скорость 16-битовых программ все еще слишком мала.

При желании вы можете войти в бесконечный цикл Game_Main() и никогда больше не возвращаться в основной цикл событий WinMain(), но это было бы плохо, поскольку в этом случае Windows не будет получать сообщения. Нам нужно рассчитать и вывести один кадр анимации, а затем вернуться в WinMain(). При этом Windows продолжит работу и будет обрабатывать сообщения. Этот процесс показан на рис. 1.17.

Внутренняя логика игры

После выполнения блока игровой логики в функции Game_Main() производится визуализация изображения во внеэкранную рабочую область (двойной буфер, или на жаргоне DirectX “задний буфер” (back buffer)). Завершающим этапом является вывод изображения на экран в конце цикла с помощью вызова DDraw_Flip(), что создает иллюзию анимации. Игровой цикл состоит из стандартных разделов, определенных ранее в элементах двумерных или трехмерных игр. Теперь я хочу сосредоточиться на 3D-графике.

Логика искусственного интеллекта врага очень проста. Вражеский корабль создается в случайной точке трехмерного пространства на расстоянии, превышающем видимость. Рисунок 1.18 показывает пространство вселенной Raiders3D. Как видно из рисунка, камера или наблюдатель расположены в точке на отрицательной оси z с координатами (0,0,-zd), где -zd — расстояние от наблюдателя до виртуального окна, куда проецируется изображение. Здесь используется левая система координат (положительная ось z направлена в экран).

После того как вражеский корабль создан, он следует по заданному вектору траектории, пересекающей поле зрения игрока, т.е. он идет более или менее встречным курсом. Вектор и начальное положение корабля генерируются функцией Init_Tie(). Вашей целью как игрока является прицелиться во врага и выстрелить.

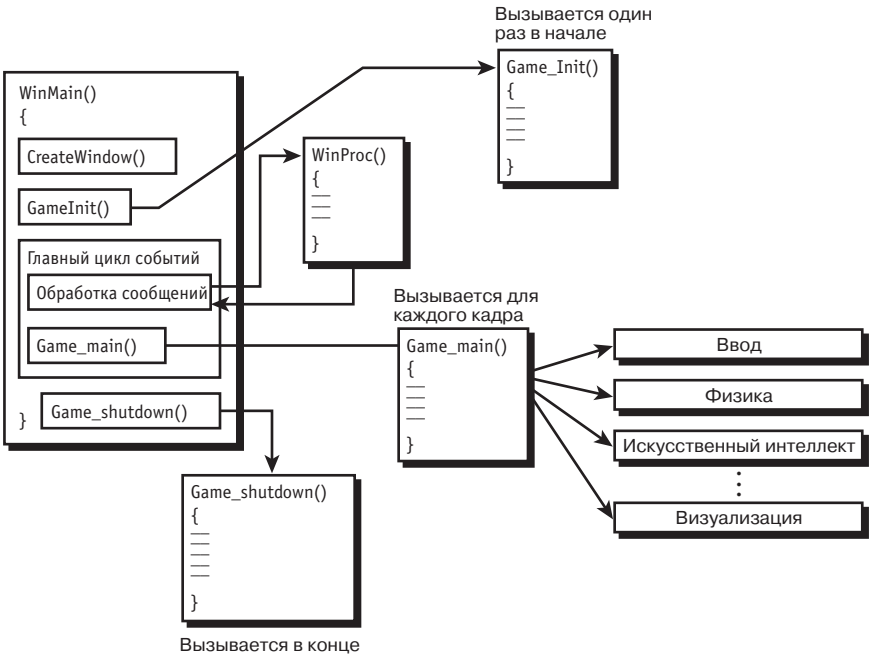


Рис. 1.17. Схема системы обработки сообщений

Итак, как же генерируется трехмерная картинка? Вражеские корабли — это не что иное, как многоугольные объекты (совокупность линий, которые образуют контур 3D-объекта) — т.е. они скорее двумерные, а не полностью трехмерные. Ключевым моментом трехмерности является перспектива. На рис. 1.19 показана разница между ортогональной и аксонометрической проекцией. Это два основных типа проекций, используемых в системах трехмерной графики.

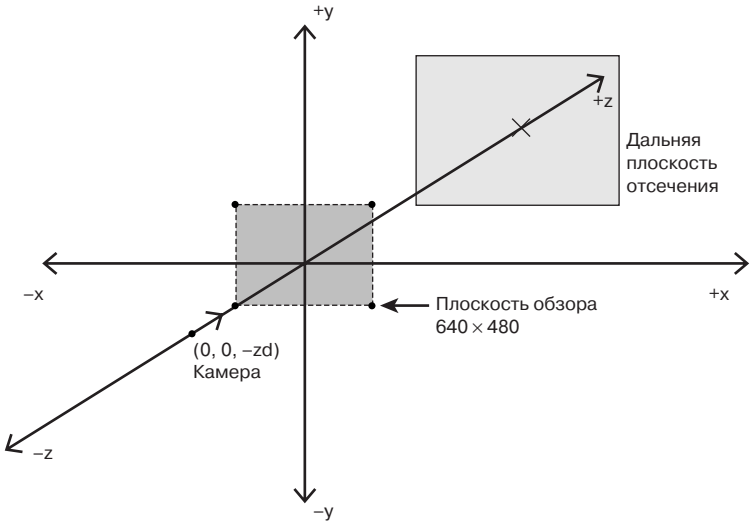


Рис. 1.18. Вселенная Raiders3D

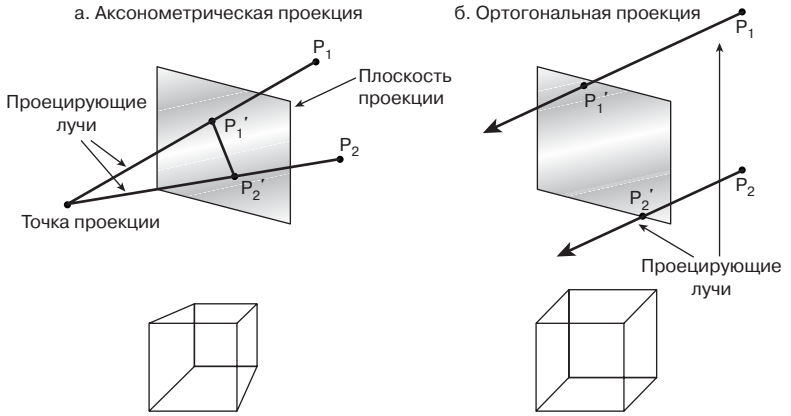


Рис. 1.19. Аксонометрическая и ортогональная аксонометрическая проекции

Трехмерные проекции

Ортогональная проекция хорошо подходит для технических чертежей и изображений, где аксонометрическое искажение нежелательно. Математика ортогональной проекции очень проста — по сути, происходит простое изменение координаты z каждой точки в соответствии со следующим уравнением.

Уравнение 1.1. Ортогональная проекция

Для точки с координатами (x, y, z)

$$x_{ortho} = x,$$

$$y_{ortho} = y.$$

Аксонометрическая проекция немного сложнее, и на данном этапе я не хочу слишком вдаваться в ее объяснение. В целом, для получения двумерной проекции на экране с координатами (x_{per}, y_{per}) нам необходимо учитывать координату z, а также расстояние от наблюдателя. Математика аксонометрической проекции показана на рис. 1.20. Она основана на вычислении координат подобных треугольников.

Уравнение 1.2. Аксонометрическая проекция

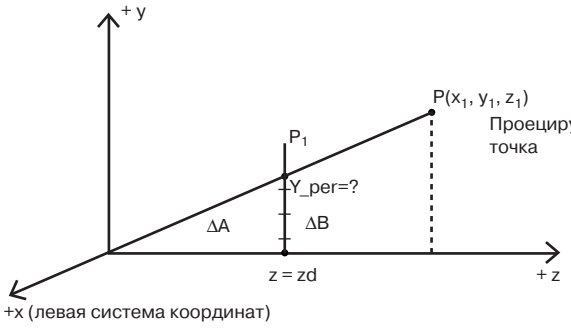
Для точки с координатами (x, y, z) с расстоянием до наблюдателя zd

$$x_{per} = \frac{zd \cdot x}{z},$$

$$y_{per} = \frac{zd \cdot y}{z}.$$

С помощью такого простого уравнения объекты, состоящие из многоугольников, в трехмерном пространстве можно перемещать так же, как и двумерные объекты. Применяя к объектам аксонометрическое преобразование до выполнения визуализации, мы достигаем корректный вид и перемещение в трехмерном пространстве. Конечно, здесь есть несколько своих тонкостей, но сейчас они не важны. Все, что нужно знать, — это то, что трех-

мерные объекты проецируются на двумерное поле зрения (экран) по законам аксонометрической проекции.

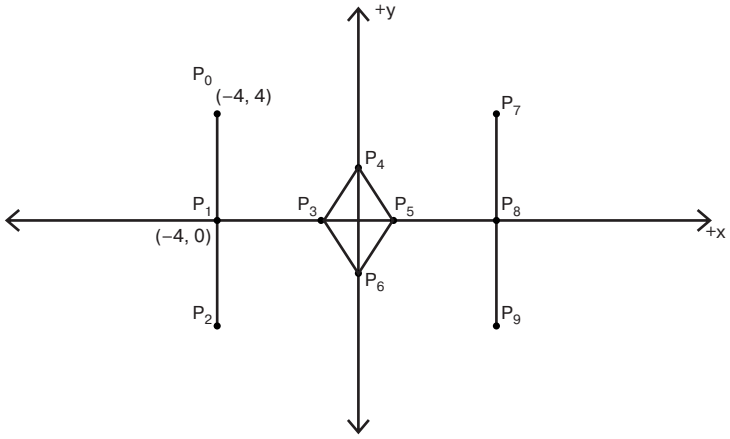


Используя подобие треугольников, можно записать:

$$\frac{Y\_per}{Y_1} = \frac{zd}{z},$$

$$Y\_per = Y_1 \cdot \frac{zd}{z}.$$

Рис. 1.20. Аксонометрическое преобразование



	Список вершин $P_0 \ldots P_9$
P_0	(–4, 4)
P_1	(–4, 0)
P_2	(–4, –4)
P_3	(–1, 0)
P_4	(0, 2)
P_5	(1, 0)
P_6	(0, –2)
P_7	(4, 4)
P_8	(4, 0)
P_9	(4, –4)

	Список ребер
Ребро 0	P_0–P_2
Ребро 1	P_1–P_3
Ребро 2	P_3–P_4
Ребро 3	P_4–P_5
Ребро 4	P_5–P_6
Ребро 5	P_6–P_3
Ребро 6	P_5–P_8
Ребро 7	P_7–P_9

Рис. 1.21. Каркасная модель боевого корабля

Итак, мы преобразуем каждый вражеский корабль с помощью аксонометрической проекции и выполняем его визуализацию на экран. Рис. 1.21 показывает каркасную мо-

дель вражеского корабля, которая используется в качестве основы для визуализации. Таблица положений каждого корабля хранится в следующем массиве.

```
// Боевой корабль
typedef struct TIE_TYP
{
  int state;     // Состояние корабля: 0=мертв, 1=жив
  float x, y, z;  // Координаты корабля
  float xv,yv,zv; // Скорость корабля
} TIE, *TIE_PTR;
```

Когда приходит время рисовать вражеские корабли, используются данные в виде структуры TIE и мы получаем более-менее реалистичное трехмерное изображение движущегося объекта.

Вы также заметите, что корабли по мере приближения становятся ярче. Этот эффект легко реализовать. Главный принцип состоит в том, что ось z используется в качестве масштабирующего коэффициента при изменении яркости корабля.

Звездное поле

Звездное поле — это не что иное, как совокупность одиночных точек, генерируемых некоторым источником в пространстве. Кроме того, после ухода из поля зрения игрока они появляются заново. Точки подвергаются визуализации как полноценные 3D-объекты в соответствии с перспективным преобразованием. Однако их размер — 1×1×1 пиксель, поэтому это "настоящие" точки и всегда выглядят как одиночные пиксели.

Лазерные пушки и обнаружение попаданий

Лазерные пушки, из которых стреляет игрок, — это не что иное, как двумерные линии, выходящие из углов экрана и сходящиеся на перекрестии прицела. Обнаружение попаданий выполняется путем наблюдения за двумерными проекциями кораблей на поле зрения и проверкой того, попадают ли лазерные лучи в ограничивающий прямоугольник каждой проекции. Это процесс схематически показан на рисунке 1.22.

Этот алгоритм работает, поскольку лазерные лучи распространяются со скоростью света. Не имеет значения, находится ли цель на расстоянии 10 метров или 10000 километров, — если вы прицеливаетесь и лазерный луч пересекает проекцию трехмерного объекта, произойдет попадание.

Взрывы

Взрывы в этой игре очень впечатляют размером кода. При попадании лазерного луча в корабль, линии, образующие 3D-модель корабля, копируются во вторичную структуру данных. Затем линии беспорядочно расходятся в трехмерном пространстве, что напоминает обломки корабля. Движение линий происходит несколько секунд, после чего взрыв прекращается. Эффект очень реалистичен и реализован меньше чем 100 строками кода.

Как играть в Raiders3D

Чтобы запустить игру — просто сделайте щелчок мышью на файле RAIDERS3D.EXE на компакт-диске, и программа тотчас запустится. Клавиши управления:

- клавиши управления курсором — перемещение перекрестия прицела;
- пробел — огонь из лазерных пушек;
- Esc — выход из игры.

Игра использует DirectDraw, DirectInput, DirectSound и DirectMusic, поэтому убедитесь, что в системе установлен DirectX.

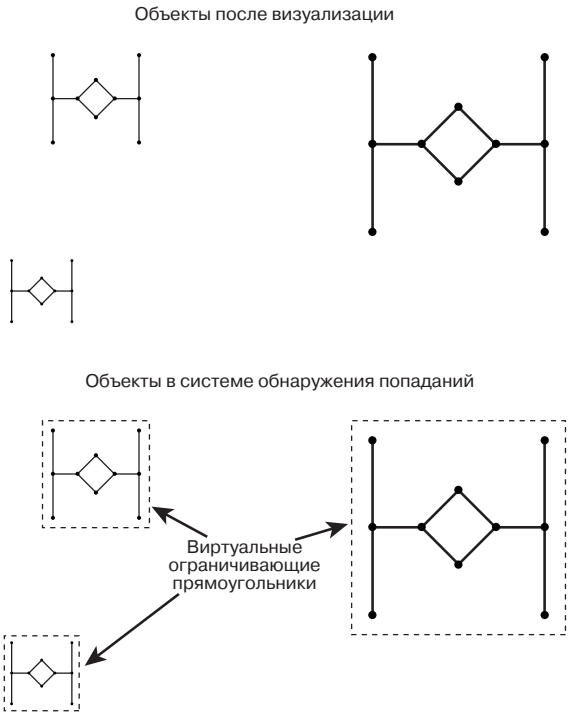


Рис. 1.22. Использование двумерных ограничивающих прямоугольников для обнаружения попаданий

Резюме

Эта глава представляет собой лишь начало введения в программирование трехмерных игр. Самым важным здесь является компилирование DirectX-программ. Кроме того, мы рассматриваем основы программирования игр, игровой цикл, основы аксонометрического преобразования 3D-объектов, а также использование компилятора. Здесь вкратце рассмотрена библиотека `T3DLIB`, разработанная в предыдущей книге *Программирование игр для Windows. Советы профессионала*. Она позволит нам сосредоточиться на программировании трехмерных игр, а не на создании поверхностей, изучении интерфейса и загрузке звука. Теперь, когда вы уже умеете компилировать, поэкспериментируйте с игрой — добавьте противников, астероиды, что-нибудь еще — и приступайте к чтению новой главы.