СЕРГЕЙ ГАЙДУКОВ

bhv

OpenGL

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ ТРЕХМЕРНОЙ ГРАФИКИ НА C++

НОВЫЕ ВОЗМОЖНОСТИ OpenGL

СРЕДСТВА NVIDIA OpenGL SDK

ПРОГРАММИРОВАНИЕ ИГР

ЭКСПОРТ МОДЕЛЕЙ ИЗ 3ds max

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

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

Гайдуков С. А.

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

ISBN 5-94157-363-4

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

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

Для программистов

УДК 681.3.068+800.92C++
ББК 32.973.26-018.1

Группа подготовки издания:

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

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.03.04.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 59,34.
Тираж 3000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

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

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

ISBN 5-94157-363-4

Содержание

Глава 2

Библиотека GLH

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

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

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

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

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

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

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

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

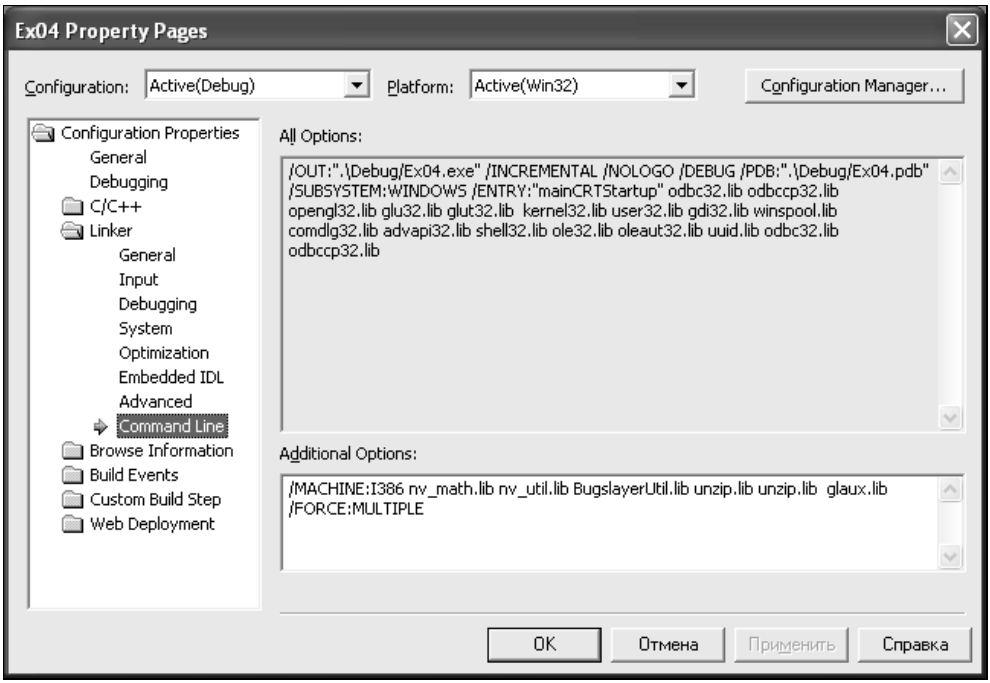

Рис. 2.1. Свойства компановщика

2.1. Математическая библиотека GLH_LINEAR

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

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

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

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

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

2.1.1. Классы для работы с векторами

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

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

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

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

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

Тип `real` определен следующим образом:

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

Следовательно, он является аналогом типа `float`.

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

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

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

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

Класс vec4 имеет пять конструкторов, определенных следующим образом:

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

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

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

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

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

Для изменения значения вектора используется метод `set_value`:

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

Для получения значений компонентов вектора используется метод `get_value`:

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

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

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

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

```
T v[N]
```

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

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

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

Таблица 2.1. *Основные методы класса* `vec`

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

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

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

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

Листинг 2.1

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

using namespace std;
using namespace glh;

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

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

        cout<<c;

        getchar();
};
```

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

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

2.1.2. Класс *line*

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

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

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

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

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

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

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

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

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

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

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

Таблица 2.2. *Методы класс* `line`

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

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

Листинг 2.2

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

```
using namespace std;
using namespace glh;

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

       vec3f p1;
       vec3f p2;

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

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

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

2.1.3. Работа с матрицами

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

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

Класс `matrix4` имеет четыре конструктора:

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

Внимание

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

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

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

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

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

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

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

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

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

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

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

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