OpenGL. Шейдеры

Содержание

1. Определение шейдера

Шейдер (от англ. *shader*), шейдерная программа – небольшая программа, выполняемая на стороне видеокарты или другого аппаратного или программного устройства рендеринга, которая позволяет производить отдельные элементы цикла рендеринга объекта особым, отличным от стандартного, образом. Иначе говоря, шейдер – программа, выполняющая некоторую часть цикла рендеринга.

2. Разновидности шейдеров. Их цели, задачи, возможности

На данный момент, шейдеры бывают **пиксельными** и **вершинными**. Часто их еще называют *пиксельными и вершинными программами*. Такое деление справедливо для большинства популярных языков программирования шейдеров, специфичные языки программирования шейдеров могут использовать другую классификацию.

В каждый момент выполнения программы активна лишь одна пара, состоящая из вершинного и пиксельного шейдера. В случае, если программист не установил активной пары шейдеров, работает стандартный шейдер, который обеспечивает всю стандартную функциональность графической библиотеки (OpenGL или DirectX). Обычно пару вершинных и фрагментных шейдеров называют просто *шейдером*.

Вершинные шейдеры имеют целью и задачей обработку каждой вершины и нахождение её координат с учетом матрицы моделирования и прочих, зависящих от задумки программиста, условий. Для этого вершинный шейдер получает все параметры вершины (координаты, цвет, текстурные координаты и т.д.). Также основная программа может передать шейдеру любые другие, определяемые программистом, параметры, включая совершенно произвольные, имеющие смысл только для выполнения общей задачи. Так, к примеру, шейдеру может быть передан параметр «сила ветра», который совершенно не привязан ни к какому способу представления трехмерных объектов, ни к какому графическому API. Это просто параметр, который будет обрабатываться внутри шейдера.

Фактически, при обработке каждой вершины запускается вершинный шейдер.

Пиксельные шейдеры имеют своей целью расчет цвета каждого пикселя и значения глубины (обычно, глубину рассчитывать необязательно) или вынесение решения о том, что пиксель выводит не надо. Пиксельный шейдер получает на входе текстурные координаты, соответствующие обрабатываемой точке, примешиваемый цвет и, возможно, некоторые другие параметры. На основании этих данных он расчитывает цвет точки и завершает работу.

Пиксельный шейдер вызывается при обработке каждой двумерной точки.

Необходимо четко понимать, что шейдеры работают раздельно друг от друга, ими можно подменить не все части рендеринга и это накладывает значительные ограничения на возможные в реализации шейдерами алгоритмы.

Рис. 1 иллюстрирует, какие этапы рендеринга подменяют собой пиксельные и вершинные шейдеры.

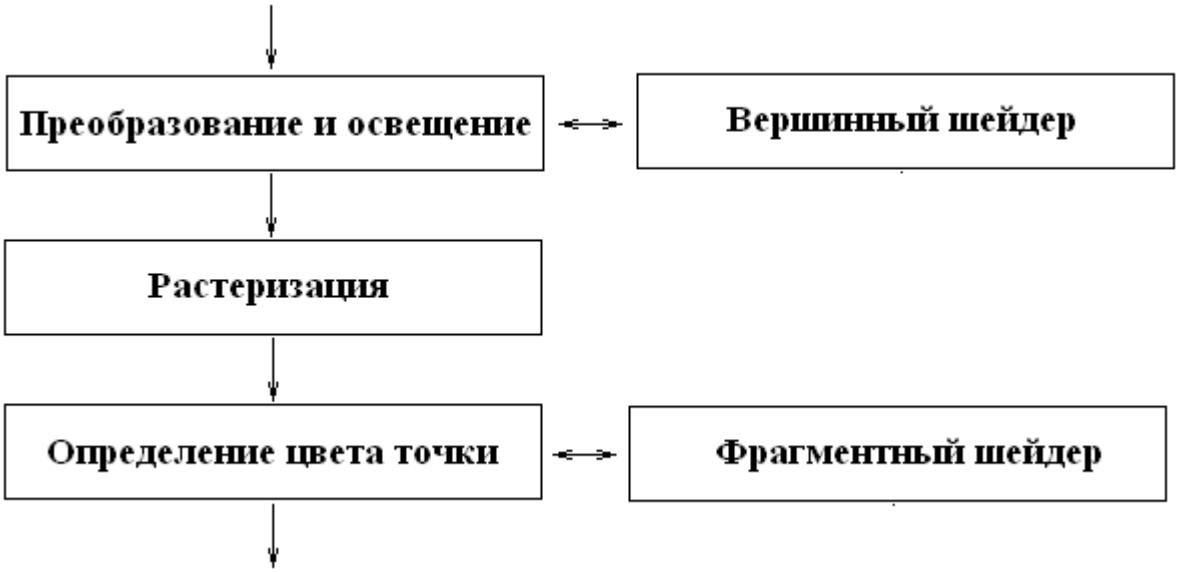

Рис. 1. Этапы рендеринга, подменяемые шейдерами.

3. Шейдерные языки, их сравнение, плюсы и минусы.

3.1. Язык шейдеров RenderMan

3.1.1. Что такое RenderMan?

Это стандарт описания трехмерных данных для их последующей визуализации (официальное определение: "это стандартный интерфейс между программами моделирования и рендеринга для создания фотореалистичного качественного изображения").

Также наиболее часто под этим под словом RenderMan "скрывается" продукт, осушествляющий рендериг из RI, то есть реализация фирмы Pixar "Pixar's PhotoRealistic RenderMan product", сокращенно PRMan.

RenderMan-стандарт используется с 1987 года многими известными корпорациями, поэтому о нем и говорят, как о стандарте в современном мире рендеринга.

3.1.2. Язык шейдеров RSL

Одной из основополагающих идей RenderMan стала идея необходимости специального программируемого освещения для достижения возможности рендеринга высокофотореалистичных изображений. Результатом разработок в этой области стало описание языка RenderMan™ Shading Language, опубликованное в 1990 году в ACM Computer Graphics.

RSL — это основанный на C язык, специально созданный для написания шейдеров. Из-за того, что он основан на C, он сохранил многие свойства этого языка. RSL очень

похож на современные языки программирования шейдеров и для него существуют сотни готовых шейдеров, которые находятся в свободном доступе в сети интернет.

- Для тех, кому стало интересно…

RSL имеет следующие виды данных:

1) Число с плавающей точкой – что интересно, в RSL нет целых чисел, они тоже представляются числами с плавающей точкой.
2) Данные о цвете – трехмерный вектор из трех чисел с плавающей запятой. Каждое из трех чисел представляет собой интенсивность соответствующей компоненты цвета в цветовом пространстве RGB.
3) Векторы, точки и нормали. Ранее, все три типа объединялись в один, но в новых версиях появилось разделение, в основном оно базируется на смысле того, что представляет собой триада чисел с плавающей запятой.
4) Матрицы преобразования. Матрицы преобразования удобны для перехода из одной системы координат в другую. Также они полезны для преобразования систем координат и для построения проекций.
5) Строки. Чаще всего строки выполняют роль именованных констант.
6) Массивы. Массивы полезны для передачи шейдеру целого списка однородных параметров.

Глобальные переменные.
В зависимости от своего типа, каждый шейдер имеет определенный набор глобальных переменных. Это переменные, которые в обязательном порядке нужны для работы шейдера, например вектор направления камеры. Некоторые из них read-only, другие же вы можете перезаписать своими значениями.

Сравните язык RSL и язык GLSL, описанный ниже. Что их роднит и что отличает?

3.2. Язык шейдеров Gelato

3.2.1. Что такое Gelato?

NVIDIA Gelato представляет собой оригинальную гибридную систему рендеринга изображений и анимации трехмерных сцен и объектов, использующую для расчетов центральные процессоры и аппаратные возможности профессиональных видеокарт серии Quadro FX.

Основополагающим принципом, которого неукоснительно придерживаются разработчики, является бескомпромиссное качество финального изображения, не ограниченное ничем, в том числе - современными возможностями видеокарт. Как производственный инструмент, способный создавать конечный продукт высокого качества, Gelato предназначен для профессионального использования в таких областях как кино, телевидение, промышленный дизайн и архитектурные визуализации.

Gelato является не просто программой с предопределенным набором функций рендеринга. Это гибкая расширяемая программируемая система, которая может быть интегрирована в существующий производственный конвейер.

3.2.2. Язык шейдеров Gelato

Для программирования шейдеров Gelato обладает собственным специализированным языком GSL - Gelato Shading Language. Создание кода шейдеров полностью описано в документации.

Все материальные шейдеры Gelato условно делятся на три группы - шейдеры свойств поверхности, displacement-шейдеры и шейдеры объемных эффектов. Инновационной возможностью является возможность задания многоуровневых шейдеров для более точных эффектов…

3.3. OpenGL-расширения GL_ARB_VERTEX_PROGRAM, GL_ARB_FRAGMENT_PROGRAM

Эти расширения OpenGL ввели возможность написания вершинных и пиксельных шейдеров на языке, схожем с языком ассемблера.

У таких шейдеров были очевидные минусы:

- Сильное ограничение на общее количество команд в шейдере
- Низкий уровень языка разработки
- Недостаточные возможности языка

3.4. Язык шейдеров CG от NVidia

Язык шейдеров CG появился в 2002 году. Отличительной чертой является использование C-подобного синтаксиса. Поддерживаются циклы, условный оператор, функции. Кроме того, главным новшеством была трансляция шейдеров, написанных на CG, на тот низкоуровневый язык шейдеров, который поддерживает конкретная видеокарта, будь то расширения GL_ARB_VERTEX_PROGRAM, GL_ARB_FRAGMENT_PROGRAM или специфичные расширения видеокарты.

Очевидные преимущества:

- Язык высокого уровня
- Может применяться как с OpenGL, так и с DirectX
- Может применяться на самых различных видеокартах за счет компиляции в низкоуровневый шейдерный язык видеокарты.

Недостатки:

- Использование CG требует поставлять вместе с основной программой и динамические линкуемые библиотеки NVidia CG, которые имеют приличный размер.
- Язык не стал стандартом.
- Язык заточен главным образом под видеокарты NVidia

3.5. Язык шейдеров GLSL

Язык шейдеров GLSL был введен в обращение стандартом OpenGL 2.0. Он очень похож на CG, в том числе тем, что этап компиляции также присутствует. Однако это стандарт, который вынуждены соблюдать все производители видеокарт. Более того, видеокарты любого нового крупного игрока на рынке видеокарт будут вынуждены поддерживать GLSL. Как в этих изменившихся условиях поведет себя NVidia и насколько оптимально она сможет реализовать в CG поддержку новых видеокарт сторонних производителей – большой вопрос.

Очевидные преимущества:

- Язык высокого уровня
- Также может применяться на самых различных видеокартах.
- Поддержан стандартом OpenGL 2.0
- Поддержка GLSL включается в стандартные драйверы OpenGL. Их применение не требует использования больших дополнительных библиотек.

Недостатки:

- Нет поддержки DirectX

3.6. Языки шейдеров для DirectX

В DirectX в свое время также существовал низкоуровневый язык программирования шейдеров. Сейчас в DirectX поддерживается высокоуровневый язык шейдеров HLSL. Он похож на CG и GLSL и является стандартом для DirectX. Совсем недавно корпорация ATI выпустила в свет утилиту, конвертирующую шейдеры, написанные на языке HLSL, в шейдеры на GLSL. Этот факт говорит о совершенной идентичности функциональности обоих языков.

4. Язык программирования шейдеров GLSL

4.1. Основные типы данных. Объявление переменных

Основные типы данных GLSL перечислены в табл. 1.

bool	Аналогично C/C++
int	Аналогично C/C++, но гарантируемый стандартом размер 2 байта (может быть больше)
float	Аналогично C/C++
vec2	2-мерный массив вещественных чисел
vec3	3-мерный массив вещественных чисел
vec4	4-мерный массив вещественных чисел
bvec2	2-мерный массив булевых переменных
bvec3	3-мерный массив булевых переменных
bvec4	4-мерный массив булевых переменных
ivec2	2-мерный массив целых чисел
ivec3	3-мерный массив целых чисел
ivec4	4-мерный массив целых чисел
mat2	матрица 2 на 2 вещественных чисел
mat3	матрица 3 на 3 вещественных чисел
mat4	матрица 4 на 4 вещественных чисел
sampler1D	ссылка на одномерную текстуру
sampler2D	ссылка на двумерную текстуру
sampler3D	ссылка на трехмерную текстуру
samplerCube	ссылка на кубическую текстуру
sampler1DShadow	ссылка на одномерную текстуру глубин
sampler2DShadow	ссылка на двумерную текстуру глубин

Табл.1. Основные типы данных GLSL

В GLSL как и в C/C++ могут задаваться массивы, структуры, функции. Объявление переменных идентично C/C++. Глобальные переменные с одинаковыми именами в обоих шейдерах считают идентичными. Кроме типа переменной у переменной может быть объявлен один из описателей (см. табл. 2).

const	константа
attribute	переменная, передаваемая вершинному шейдеру с каждой вершиной
uniform	переменная, передаваемая шейдеру из основной программы
varying	переменная, передаваемая из вершинного шейдера в фрагментный, интерполируется вдоль грани с учетом перспективы

Табл. 2. Основные описатели

Описатель пишется перед типом переменной, например

```
varying vec3 normal;
```

4.2. Основные конструкции языка.

GLSL поддерживает почти все распространенные в C/C++ арифметические, логические и битовые операторы. Поддерживаются конструкторы, например

```
vec2 v = vec2(1.0, 2.0);
```

Если в параметрах конструктора указано только одно число, то им инициализируются все компоненты вектора.

```
vec2 = vec2(1.0);
```

Для доступа к компонентам векторов (vec2, vec3, vec4) используются три разных пространства имен. (x, y, z, w) – удобно для векторов в пространстве, (r, g, b, a) – удобно для векторов цвета, (s, t, p, q) – удобно для доступа к векторам текстурных координат.

```
vec4 v;
v.x = 1.0;
v.y = 1.0;
v.z = 1.0;
v.w = 2.0;
vec3 v2 = v.rgb;
```

Также можно обращаться к компонентам вектора индексно, например v[0] будет идентично v.x.

```
vec2 v;
v[0] = 1.0;
v.y = 2.0;
```

Если m – это матрица, то m[0] будет соответствовать 0-му столбцу матрицы, а m[i, j] – элементу матрицы (i, j).

```
mat4 m;
m[2] = vec4(1.0, 2.0, 3.0, 4.0);
m[0][0] = 1.0;
```

Для работы с пикселями в текстуре используются специальные функции. С помощью функции vec4 texture2D(sampler2D sampler, vec2 coord) можно получить цвет пикселя в текстуре по заданным координатам. Вы можете задать любые текстурные координаты, совершенно не зависящие от реальных размеров текстуры (например, они могут оказаться где-то между реальных пикселей текстуры). Тем не менее, описываемая функция возвратит вам приближенный цвет пикселя, каким бы он в этой точке должен был быть, если бы текстура не состояла из дискретных пикселей, а была бы единой и монолитной.

```
uniform sampler2D texture1;

void main (void)
{
      vec3 tempColor = texture2D(texture1, gl_TexCoord[0].xy).rgb;
```

```
}
```

GLSL поддерживает оператор условия, циклы for, while, do while, команды переходов continue и break, а также оператор discard, прекращающий выполнение программы и запрещающий обработку пикселя. Также поддерживаются функции.

```
float value = gl_TexCoord[0].x;
if (value > 0.5)
{
     discard();
}
```

Для возвращения результата работы вершинного шейдера существуют предопределенные переменные gl_Position (vec4, содержит координаты вершины), gl_PointSize (float, содержит размер точки), gl_ClipVertex (vec4, координаты для проверки на отсечение).

Для возвращения результата работы фрагментного шейдера в простейшем случае используют предопределенную переменную gl_FragColor (vec4, содержит цвет пикселя). Также довольно часто рассчитывается глубина, которая заносится в переменную gl_FragDepth.

Также в вершинный шейдер передаются следующие предопределенные переменные: gl_Color (vec4, цвет для вершины), gl_SecondaryColor (vec4, цвет для вторичной вершины), gl_Normal (vec4, нормаль к вершине), gl_Vertex (vec4, координаты вершины), glMultiTexCoord0 (vec4, текстурные координаты для блока 0) и т.д.

Также в шейдер передаются и некоторые другие переменные, а язык имеет некоторые другие богатые возможности, например, различные дополнительные математические операции. Однако приведенного достаточно для оценки мощи языка в обзорных целях.

4.3. Стандартный цикл работы шейдера.

Для обработки каждой вершины вызывается текущий вершинный шейдер. Он производит перерасчет координат вершины, а также расчитывает значения varying-переменных, которые будут интерполироваться по грани. Видеокарта производит расчет текстурных координат (также с помощью интерполяции), находит текущих значения определенных программистом и предопределенных varying-переменных для каждого пикселя и вызывает (также для каждого пикселя) фрагментный шейдер. Он производит с учетом всех рассчитанных параметров расчет конечного цвета пикселя. Несмотря на простоту рабочего цикла шейдера, шейдеры позволяют воплощать сложные и интересные эффекты.

4.4. Пример простейшего шейдера

```
// Simple vertex shader
void main (void)
{
     gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}

// Simple fragment shader
uniform sampler2D texture1;

void main (void)
{
     vec3 tempColor = texture2D(texture1, gl_TexCoord[0].xy).rgb;
```

```
        gl_FragColor = vec4(tempColor, 1.0);
}
```

Разберем, что делает этот шейдер. Единственная строка вершинного шейдера рассчитывает координаты трехмерной вершины и заносит их в переменную типа vec4 gl_Position. Это происходит путем перемножения вектора gl_Vertex, содержащего координаты вершины, и матрицы gl_ModelViewProjectionMatrix.

В пиксельном шейдере первой строчкой мы заносим значение цвета из текстуры texture1 по текущим текстурным координатам в переменную tempColor типа vec3. Второй строчкой мы заносим этот цвет в переменную gl_FragColor, но т.к. она типа vec4, добавляем, что alpha-значение цвета = 1. Таким образом мы рассчитываем цвет текущего пикселя.

4.5. Подключение шейдеров на GLSL к основной программе

```
void LoadShaderSource(GLhandleARB shader, const char fileName[])
{
        std::string shaderSrc = LoadTextFile(fileName);

        const char* shaderstring = shaderSrc.c_str();
        const char** pshaderstring = &shaderstring;

        glShaderSourceARB(shader, 1, pshaderstring, NULL);
}

GLhandleARB vxShader, frShader, prObject;

vxShader = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
frShader = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);

LoadShaderSource(vxShader, vxFileName);
LoadShaderSource(frShader, frFileName);

glCompileShaderARB(vxShader);
glCompileShaderARB(frShader);

prObject = glCreateProgramObjectARB();

glAttachObjectARB(prObject, vxShader);
glAttachObjectARB(prObject, frShader);

glLinkProgramARB(prObject);
```

Так выглядит простейший пример компиляции шейдеров. В целях увеличения понимаемости в примере обработка ошибок не производится. vxFileName и frFileName содержат имена файлов, содержащих исходный код соответственно вершинного и фрагментного шейдеров.

Компиляция состоит в следующем: сначала мы создаем объекты GL_VERTEX_SHADER_ARB и GL_FRAGMENT_SHADER_ARB, затем связываем их с исходным кодом, затем компилируем каждой по отдельности, создаем объект GL_PROGRAM_OBJECT_ARB, соотносим с ним откомпилированные вершинный и фрагментный шейдеры и линкуем этот объект. После этого программа может сделать шейдер активным. Делается это так:

```
glUseProgramObjectARB(prObject);
```

Отключить шейдеры (т.е. заменить установленные шейдеры на стандартный):

```
glUseProgramObjectARB(0);
```

И удалить шейдеры из памяти:

```
glDeleteObjectARB(vxShader);
glDeleteObjectARB(frShader);
glDeleteObjectARB(prObject);
```

Также основная программа может передать шейдеру какие-либо параметры, например, с помощью таких процедур

```
bool SetUniformFloat(const char* name, float value)
{
	int loc = glGetUniformLocationARB(prObject, name);

	if (loc < 0)
	{
		return false;
	}

	glUniform1fARB(loc, value);

	return true;
}

bool SetUniformTexture(const char* name, int texUnit)
{
	int loc = glGetUniformLocationARB(prObject, name);
	if (loc < 0)
	{
		return false;
	}

	glUniform1iARB(loc, texUnit);

	return true;
}

bool SetUniformVector4D(const char* name, const CVector4D& value)
{
	int loc = glGetUniformLocationARB(prObject, name);
	if (loc < 0)
	{
		return false;
	}

	glUniform4fvARB(loc, 1, value);

	return true;
}
```

В параметрах процедур name – строка с именем параметра, value – его значение, texUnit – номер текстурного уровня (вспомните понятие *мультитекстурирования*). Эти процедуры устанавливают параметры для активного шейдера.

5. Область применения шейдеров

Основное применение шейдеров заключается в реализации специфичных моделей освещения и в реализации различных физических эффектов.

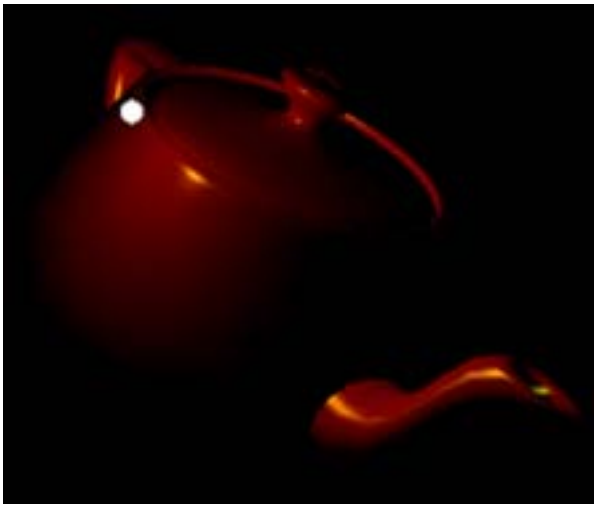

Рис. 2. Phong mapping – освещение по модели Фонга. На поверхности объекта мы можем наблюдать блики, которых нельзя добиться при использовании стандартного освещения OpenGL.

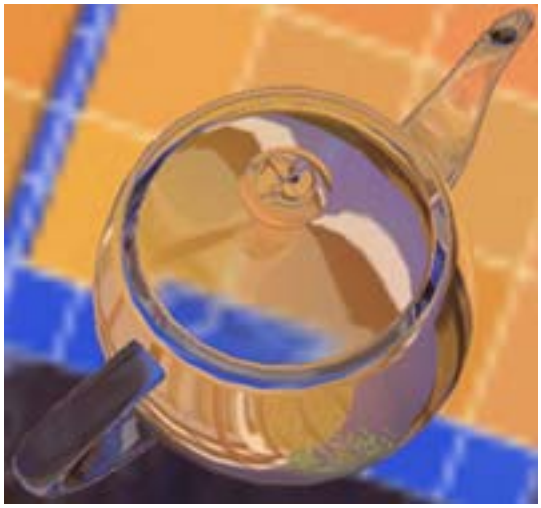

Рис.3. Эффект стеклянного объекта. Мы можем наблюдать искажение текстурных координат, соответствующее преломлению луча в стеклянном чайнике, чего стандартными средствами достичь нельзя.

Рис. 4. Bump mapping – имитация микронеровностей объекта. Эффект заключается в том, что торус на самом деле абсолютно гладкий, однако для него существует карта нормалей, которая соответствует наблюдаемым микронеровностям. Сам эффект создается за счет освещения, зависящего от направления нормали.

Рис. 5. Эффект водной поверхности. Имеет множество вариантов реализации на шейдерах, чаще всего его реализуют с помощью кубических текстурных карт, содержащих окружающую водную поверхность среду, и некоторой шумовой функции от времени, от значения которой зависят искажения на водной поверхности.

Рис. 2, 3, 4, 5 иллюстрируют некоторые из специфичных моделей освещения и физических эффектов, которые часто реализуются с помощью шейдеров. Как известно, стандартный OpenGL поддерживает по сути только две модели освещения: модель

Ламберта или так называемый flat shading (у всей грани одинаковая освещенность) и модель Гуро или так называемый Gouraud shading (освещенность равномерно интерполируется по грани). С помощью тумана можно также добиться третьего варианта освещения – освещения по Z-буферу.

Однако, во-первых, моделей освещения гораздо больше. Одна из самых популярных моделей освещения – phong shading – стандартным OpenGL не поддерживается (на пиксельном уровне). Не поддерживаются и другие, интересные модели освещения.

Во-вторых, OpenGL вносит ограничение на общее число источников света – не более 8. В своих реализациях на шейдерах вы можете использовать и большее количество источников света.

Также очень часто шейдеры используются для реализации так называемых **postprocessing-эффектов**. Они получили свое название из-за того, что реализация самого эффекта производится после рендеринга сцены. Обычно процесс реализации выглядит следующим образом:

1. Сначала рендерится 3д-сцена.
2. Отрендеренная сцена берется в текстуру, это можно сделать как минимум 4 способами:
   - через glReadPixels (читает видеобуфер в память, наиболее неоптимально)
   - через glCopyTexImage2D (функция читает видеобуфер в текстуру)
   - через использование p-buffer'а (второй по оптимальности способ)
   - через использование расширения GL_EXT_FRAMEBUFFER_OBJECT (наиболее оптимальный способ, но требует свежих драйверов)
3. Затем во весь экран выводится прямоугольник с наложенной на него полученной текстурой, а шейдер в этот момент реализует какой-либо из 2д-фильтров.

Рис. 6. Эффект старого кино

Рис. 7. Blur (смазывание)

Рис. 8. Sepia (Сепия)

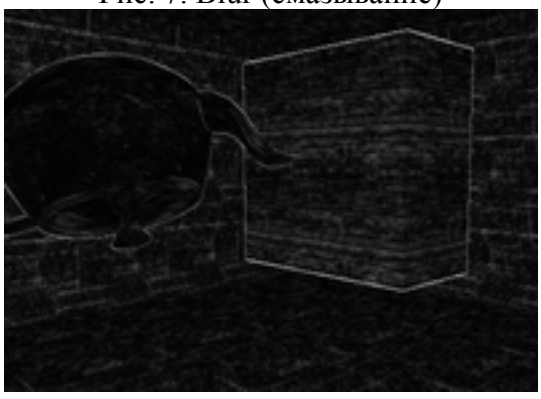

Рис. 9. Edge detect (выделение краев)

Рис. 6, 7, 8, 9 иллюстрируют некоторые из наиболее часто применяющихся postprocessing-эффектов, основанных на шейдерах.

Увеличение быстродействия – также одна из наиболее эффективных сфер применения шейдеров. Это связано с тем, что пропускная способность шины, через которую данные идут на видеокарту, не так велика, как того хотелось бы.

В связи с этим, имеет смысл как можно больше данных держать на стороне видеокарты и как можно меньше данных отправлять каждый новый кадр на видеокарту. В частности можно рассмотреть следующие примеры оптимизации с применением шейдеров.

1. Система частиц. Обычно системы частиц рассчитываются основной программой, а потом набор вершин и граней направляется в видеокарту на обработку. Оказывается, выйгрыш в скорости можно получить, если разместить первичный набор вершинных данных в дисплейных списках или VERTEX_BUFFER_OBJECT-е, которые располагаются в памяти видеокарты, а изменять положения частиц с помощью шейдеров.

2. Морфинг также часто реализуется с помощью шейдеров. Новые координаты соответствующей вершины располагают в каких-либо вспомогательных данных, например, в координатах нормали. Набор вершин располагается в памяти видеокарты, а интерполяция между фазами морфинга реализуются на шейдерах.

3. Скелетная анимация также может быть реализована на шейдерах (правда, с некоторыми ограничениями).

4. Любые искажения объекта, например, по синусу или косинусу, также легко реализуются с помощью шейдеров.

Все приведенные примеры показывают, что увеличение скорости может быть достигнуто за счет уменьшения объема пересылаемых данных, за счет размещения части данных в памяти видеокарты как статичных и изменении в реальном времени их с помощью шейдеров.

6. Методы оптимизации шейдеров

Применение сложных шейдеров может значительно замедлить работу графического конвейера. В связи с этим возникает проблема оптимизации шейдеров. Многие методы оптимизации обычных программ подойдут и для шейдеров, но есть и некоторые специфические:

- Никогда не рассчитывайте при каждом запуске шейдера то, что можно рассчитать один раз в основной программе и передавать шейдеру как параметр.
- Никогда не рассчитывайте в пиксельном шейдере то, что можно рассчитать один раз в вершинном и передавать как параметр. Учтите, что вершинные шейдеры, как правило, вызываются гораздо реже.
- Максимально оптимизируйте шейдеры вручную. Не надейтесь на то, что компилятор шейдеров сделает это за вас.

7. Проблемы совместимости в GLSL

Несмотря на то, что GLSL поддержан стандартом OpenGL 2.0, существуют проблемы совместимости. Одной из главных проблем совместимости являются различия в реализации GLSL в драйверах корпораций NVidia и ATI – ведущих производителей видеокарт любительского класса. Так, к примеру, реализация ATI требует строгого приведения типов и не допускается директивы include, подключающей к шейдеру заголовочный файл. При этом интересным моментом является то, что в данном аспекте корпорация ATI **права -** её реализация этого момента соответствует стандарту GLSL. Зато корпорация NVidia реализовала некоторый функционал, выходящий за рамки стандарта,

очевидно для большей идейной совместимости со своим детищем – CG. Это приводит к путанице среди программистов.

Эту особенность нельзя недооценивать. Несмотря на простоту обнаружения такой проблемы совместимости, она часто встречается в реально существующих проектах. Как показывает практика, часто даже известные авторы книг пренебрегают тестированием и их примеры содержает шейдеры, которые работают только на определенном подмножестве видеокарт.

Резюмируя, недооценка описанной проблемы ни к чему хорошему привести не может.

Другой проблемой совместимости является то, что программа чаще всего должна выполняться на множестве компьютеров с разными конфигурациями. В таком случае, большую роль играет то, какое конкретное число шейдерных инструкций низкого уровня допускает каждая конкретная видеокарта. Вполне возможен случай, когда вы напишите шейдер настолько сложный, что на какой-то из видеокарт не хватит шейдерных инструкций низкого уровня для компиляции вашего шейдера.

Обе главные проблемы совместимости эффективно можно обнаружить лишь одним путем – всесторонним тестированием на различном оборудовании и тщательным изучением аппаратных особенностей и возможностей линейки видеокарт ведущих фирм-производителей.

Помимо этих двух проблем, не стоит забывать третьей – шейдеры поддерживаются не всеми даже относительно современными видеокартами и совсем не поддерживаются старыми.

8. Заключение

Шейдеры предоставляют программисту мощное средство для реализации сложных трехмерных эффектов, postprocessing-эффектов и увеличения общего быстродействия. Хотя качественное использование шейдеров – сложная задача, требующая наличия у программиста множества дополнительных знаний, преимущества использования шейдеров заметно превышают недостатки.