Shaders

Шейдер (**Shader**) — программа для процессора графической карты (GPU), управляющая поведением шейдерной стадии графического конвейера и занимающаяся обработкой соответствующих входных данных. Шейдеры делятся на три типа:

- **Vertex shader** (вершинный шейдер) — программа для вершинной шейдерной стадии конвейера, обрабатывающая вершинные данные. Вершинный шейдер может описывать, например, трансформацию вершин из объектного пространства в пространство камеры. Вершинный шейдер выполняется отдельно для каждой вершины.
- **Geometry shader** (геометрический шейдер) — шейдерные программы, выполняющиеся в конвейерной стадии геометрического шейдера. Геометрический шейдер работает с вершинными данными, но выполняется сразу для элемента геометрии, например, для треугольника, то есть на вход подаются три вершины. Кроме этих трёх вершин, возможно использование вспомогательных вершин (т.н. adjacent vertices). Геометрические шейдеры способны создавать новую геометрию, могут использоваться для создания частиц, изменения детализации модели «на лету», создание силуэтов и т.д.
- **Fragment shader** (пиксельный шейдер) — программа для пиксельной шейдерной стадии конвейера. Пиксельный шейдер выполняется для каждого фрагмента растеризованной геометрии. Обычно пиксельный шейдер занимается закраской геометрического объекта — наложение текстур, освещение, и наложение разных текстурных эффектов, таких как отражение, преломление, туман, Bump Mapping и пр. Пиксельные шейдеры также используются для пост-эффектов.

С появлением программируемых шейдеров на GPU каждый тип шейдера выполнялся на специализированных процессорах, поддерживающих разные наборы инструкций. Позже шейдерная архитектура была унифицирована и деление на типы графических процессоров исчезло, вместе с тем стало возможно выполнение на GPU вычислений общего назначения (не связанных напрямую компьютерной графикой). Тем не менее деление шейдеров на типы сохранилось, так как их применение напрямую связанно с архитектурой графического конвеера видео карт, осуществляющего тесселяцию и растеризацию изображения.

Изначально шейдеры писались на ассемблеро-подобном языке, позже появились шейдерные языки высокого уровня, такие как: HLSL, GLSL и другие. Синтаксис шейдерных программ на этих языках очень близок к языку ANSI C, из которого для упрощения языка и повышения производительности исключены многие возможности. В дальнейшем речь пойдет о языке OpenGL Shading Language - **GLSL** (**G**raphics **L**ibrary **S**hader **L**anguage), что как не трудно догадаться из названия используется в OpenGL.

GLSL

GLSL - высокоуровневый язык используемый для шейдеров OGL. По синтаксису и в целом внешне идентичен C (не C++), основными отличиями является отсутствие указателей, рекурсии и динамического выделения памяти. Основные типы данных (специальные типы для текстур вроде **samplerCube** опущены):

		Вектора из 2, 3 и 4х эл-ов			Матрицы 2x2, 3x3 и 4x4		
	float	**vec2**	**vec3**	**vec4**	**mat2**	**mat3**	**mat4**
	double	**dvec2**	**dvec3**	**dvec4**	**dmat2**	**dmat3**	**dmat4**
	int	**ivec2**	**ivec3**	**ivec4**	**imat2**	**imat3**	**imat4**
	uint	**uvec2**	**uvec3**	**uvec4**	**umat2**	**umat3**	**umat4**
	bool	**bvec2**	**bvec3**	**bvec4**	**bmat2**	**bmat3**	**bmat4**

Объявление простых переменных идентично C, также можно инициализировать их прямо при объявлении. Единственное небольшие различие, что вместо приведения для указания типа вызывается "конструктор", в который надо передать соответствующее типу данных число переменных:

```
// Комментарии работают также как в C и C#
float a, b;
int   c = 2;
float d = 2;         // Неправильно
float e = (float)2; // Опять неправильно
float f = float(2); // Правильно
float g = 2.0;       // Тоже верно
bool  i = true;

ivec2 va = ivec2(1, 2);
ivec2 vb = ivec2(3, 4);
ivec3 vc = ivec3(c, vb);    // == ivec3(2, 3, 4)
ivec4 vd = ivec4(va, vb);   // == ivec4(1, 2, 3, 4)
```

Для доступа к компонентам векторов можно использовать селекторы или буквы x,y,z,w или s,t,p,q или r,g,b,a. Их назначение идентично, а их разнообразие обуславливается сугубо читаемостью кода, как не сложно догадаться из названий первые применяются обычно для координат и нормалей, вторые для текстурных координат а последние для цветов. Ниже небольшая демонстрация:

```
vec4  foo   = vec4(1.0, 2.0, 3.0, 4.0);
float posX  = foo.x;
float posY  = foo[1];
vec2  posXY = foo.xy;
float depth = foo.q;
```

Селекторы матриц могут состоять как из 1, так и из 2х элементов, например, m[0], или m[0][1]. В первом случае выбирается весь ряд, во втором - только один элемент из него.

Объявление и работа с структурами ничем не отличается:

```
struct sfoo {
    ivec3 dir;
    ivec3 normal;
};
sfoo  a;
sfoo  b = sfoo(ivec3(1, 2, 3), ivec3(4, 5, 6));
a.normal= ivec3(7, 8, 9);
```

Доступные в GLSL циклы и выражения практически идентичны с C:

```
if(bool условие)
   ...
else
   ...

for(инициализация; выражение; выражение)
...

while(bool условие)
   ...


do
   ...
while(bool условие)
```

Помимо идентичных с C директив связанных с циклами **continue** и **break** добавлена особая - **discard**. Последняя завершает работу шейдера для текущего пикселя и отменяет его дальнейшую обработку (передачу в буфер экрана или в буфер глубины). Очевидно, что использование discard допускается только в фрагментном шейдере.

Шейдеры, так же как и программы на C, делятся на подпрограммы (функции) в которых уже и содержится выполняемых код. Любой шейдер, независимо от типа, должен иметь функцию main() со следующим прототипом: **void main() { ... }** Конечно, можно объявлять собственные функции. Здесь опять же все идентично C, с небольшими нюансами: нельзя использовать рекурсивные функции, нельзя возвращать массивы, допускается перегрузка функций.

Квалификаторы переменных и параметров

Параметры функций могут иметь следующие квалификаторы:

- **in** - для входящих параметров
- **out** - выходные данные функции
- **inout** - параметры, могущие быть и входными и выходными

В случае, если квалификатор явно не задан, параметр считается входящим (те **in**)

Для глобальных переменных (объявленных не в теле функции) доступны следующие квалификаторы:

- **const** - объявление констант
- **attribute** - Глобальные переменные, которые могут изменятся для каждой вершины, и передаются в вершинные шейдеры. Может использоваться только в них. Для шейдера переменная - read-only.
- **uniform** - глобальная переменная, которая может изменятся для каждого полигона (не может быть между glBegin/glEnd), передаётся OpenGL в шейдеры. Может использоваться в обоих типах шейдеров, для шейдеров является read-only.
- **varying** - используются для передачи интерполированных данных между вершинным и фрагментным шейдером. Доступны для записи в вершинном шейдере, и read-only для фрагментного шейдера.

Как упоминалось прежде, есть два основных типа шейдеров: вершинный и фрагментный. Для вычисления цвета пикселя часто необходим доступ к интерполированным данным вершин. Например, вычисляя влияние освещения на пиксель, нам нужна информация о нормали этого пикселя. Однако в OpenGL нормали являются атрибутами вершин, следовательно доступны лишь вершинному шейдеру. Архитектура графического конвеера OpenGL устроенна таким образом, что после обработки всех вершин идет стадия экспорта данных вершин. После чего все обработанные вершины со всеми интерполируемыми значениями размещаются в кэше. Дальше растеризатор из них примитивы.

Сборка примитива - это работа с плоскостями отсечения, CULL_FACE, определение видимости примитива на экране, возможность отсечения в hierarchical-Z (то есть если растеризатор сможет сразу определить что примитив не виден, то отбросит его), и самое главное - определение пикселей под закраску (то есть выбираются какие пиксели будут закрашиваться этим примитивом). После того, как процессоры z-stencil исключают пиксели, не прошедшие z-stencil тесты, для оставшихся выполняется фрагментный шейдер.

Таким образом при вызове фрагментного шейдера вершины его полигона уже были обработаны, а поступающие значения интерполируются. Например, рассмотрим цвет пикселя является интерполяцией цветов вершин полигона из которых он состоит. Цвет пикселя это одна из встроенных переменных с квалификатором varying.

Встроенные основные функции:

function signature	description
genType *abs*(genType α)	returns the absolute value of α.i.e, -α if α < 0;
genType *sign*(genType α)	returns • -1 for α < 0 • 0 for α = 0 • 1 for α > 0
genType *floor*(genType α)	returns the nearest integer less than or equal to α
genType *ceil*(genType α)	returns the nearest integer greater than or equal to α
genType mod(genType α, float β) *genType mod*(genType α, genType β)	equivalent to α%βin Java
genType min(genType α, float β) *genType min*(genType α, genType β)	returns • α when α < β • &beta when β < α
genType max(genType α, float β) *genType max*(genType α, genType β)	returns • α when α > β • &beta when β > α
genType clamp(genType α, float β, float δ) *genType clamp*(genType α, genType β, genType δ)	returns • α when β < α < δ • β when α > β • δ when α > δ
genType mix(genType α, float β, float δ) *genType mix*(genType α, genType β, genType δ)	returns the linear blend of α and β. i.e. α + δ(β - α)
genType step(float limit, genType α)	returns • 0 when α < limit;

genType step(genType limit, genType α)	• 1 when α > limit; returns
genType smoothstep(float α0, float α1, genType β) *genType smoothstep*(genType α0, genType α1, genType β)	• 0 when β < α0 • 1 when β > α1; • smooth Hermite interpolation when α0 < β < α1

float *length*(genType α)	returns the length of a vector α
float *distance*(genType α genType β)	returns the distance between α and β
float *dot*(genType α, genType β)	returns the dot product of α and β
vec3 *cross*(genType α, genType β)	returns the cross product of α and β
genType *normalize*(genType α)	returns the nomalized vector of α. i.e. α with a length of 1.
vec4 *ftransform*()	only accessiable to vertex shader this function transforms the incoming vertex just as the OpenGL fixed-functionality *transform* (*ftransform*). This function can be used to assign a value to gl_Position when no vertex manipulation is intended for the vertex shader.
genType *normalize*(genType α)	returns the nomalized vector of α. i.e. α with a length of 1.
genType *faceforward*(genType N, genType I, genType Nref)	If *dot*(Nref, I) < 0 return N; otherwise return -N

genType *pow*(genType α,genType β)	returns α^{β}. Results are undefined if α < 0 or α = 0 and &beta <= 0.
genType *exp*(genType α)	returns the natural exponential of a, i.e., e^{α}
genType *log*(genType α)	returns the natural logarithm of α. If the value returned is β then α = e^{β}
genType *exp2*(genType α)	returns 2^{α}
genType *log2*(genType α)	returns the base 2 logarithm of α. If the value returned is β then α = 2^{β}
genType *sqrt*(genType α)	returns the square root of α i.e., √α. Returned value is undefined if α < 0.
genType *inversesqrt*(genType α)	returns the inverse of the square root of α i.e., 1/√α. Returned value is undefined if α <= 0.

function signature	description
genType *radians*(genType d)	converts degrees to radians
genType *degrees*(genType r)	converts radians to degrees
genType *sin*(genType r)	trigonometric sine function
genType *cos*(genType r)	trigonometric cosine function
genType *tan*(genType r)	trigonometric tangent function
genType *asin*(genType x)	trigonometric arc sine function. Returns a value between -π/2 to π/2. The returned value is undefined if |x| > 1.
genType *acos*(genType x)	trigonometric arc cosine function. Returns a value between 0 to π/2. The returned value is undefined if |x| > 1.
genType *atan*(genType x, genType y)	trigonometric arc tangent function of y/x. Returns a value between -π to π. The returned value is undefined if x and y are both 0.
genType *atan*(genType yx)	trigonometric arc tangent function of yx. Returns a value between -π/2 to π/2.

Загрузка шейдерной прогрммы

Традиционно OGL не использует предварительно скомпилированные шейдеры, а компилирует их при загрузке в процессе работы приложения. Это позволяет оптимизировать код шейдеров под конкретный видео процессор и платформу на которой запущенно приложение. Для удобства разместим код шейдера в текстовом файле, содержащимся в ресурсах приложения. Для этого в обозревателе решений надо нажать ПКМ по нужному проекту и в контекстном меню выбрать пункт "Добавить" -> "Создать Элемент". В появившемся окне слева выбираем "Установленные" -> "Элементы Visual C#" -> "Общий", а справа выбираем "Текстовый файл", внизу вводим название и расширение файла и жмем кнопку "Добавить". Если все сделано верно, в обозревателе решений появиться созданный нами объект. Выделим его и снова нажмем ПКМ а затем в контекстном меню выберем пункт "Свойства". Здесь необходимо установить значение "Действие при построении" выбрав "Внедренный ресурс". Создадим таким образом два файла "shader_name.vert" и "shader_name.frag" для вершинного и фрагментного шейдера соответствующе. Прочитать содержимого такого файла можно будет при помощи метода:

```
string vert_source = HelpUtils.GetTextFileFromRes("shader_name.vert");
```

Но прежде чем скомпилировать код шейдера необходимого его создать:

```
uint CreateShader(uint type)
```

- type - Тип создаваемого шейдера GL_VERTEX_SHADER или GL_FRAGMENT_SHADER
- return - 0 в случае если произошла ошибка или идентификатор созданного шейдера.

В созданные объекты шейдеров надо прежде всего загрузить сам код шейдера, для чего воспользуемся методом:

void ShaderSource (uint shader, string source)

- shader - идентификатор шейдера (полученный ранее при помощи CreateShader)
- source - текст с исходным кодом шейдера, считанного из файла или в нашем случае, загруженного из ресурсов.

Теперь загруженный шейдер можно скомпилировать при помощи:

void CompileShader (uint shader)

- shader - идентификатор шейдера (полученный ранее при помощи CreateShader)

Из загруженных ранее шейдеров нужно собрать шейдерную программу, для этого прежде всего создадим ее:

uint CreateProgram()

- return - 0 в случае если произошла ошибка или идентификатор созданной шейдерной программы.

Добавим в ранее созданную шейдерную программу скомпилированные вершинный и фрагментный шейдер:

void AttachShader (uint program, uint shader)

- program - идентификатор шейдерной программы (полученный ранее при помощи CreateProgram)
- shader - идентификатор шейдера (полученный ранее при помощи CreateShader)

Когда все шейдеры добавлены остается последний шаг необходимый для того, чтобы получить готовую программу - слинковать ее:

void LinkProgram (uint program)

- program - идентификатор шейдерной программы (полученный ранее при помощи CreateProgram)

Наша программа готова и можно установив объект программы, сделать его частью текущего рендера. Иными словами, информировать OGL что мы хотим

использовать данную шейдерную программу при обработке кадра. Вызвав данную:

void UseProgram (uint program)

- program - идентификатор шейдерной программы (полученный ранее при помощи CreateProgram) или NULL (0) если требуется отключить использование шейдера.

Теперь шейдерная программа не только загружена, но и выполняется в процессе тесселяции и растеризации изображения. Однако это справедливо лишь для простейших шейдерных программ. Ранее у описании языка GLSL уже упоминались квалификаторы uniform и attribute. Как не трудно понять из их значений, в случае их использования потребуется инициализировать такие переменные. Для этого прежде всего необходимо получить индентификаторы аттрибутов и форм в шейдерно программе, делать это разумнее всего единожды при создании шейдерной программы:

int GetAttribLocation (uint program, string name)

int GetUniformLocation (uint program, string name)

- program - идентификатор шейдерной программы (полученный ранее при помощи CreateProgram)
- name - имя переменной соответствующего вида
- return - идентификатор переменной

Для форм значения необходимо передать непосредственно, воспользовавшись, в зависимости от типа переменной, одной из перегрузок метода:

void Uniform1(int location, float v0)

void Uniform2(int location, float v0, float v1)

void Uniform3(int location, float v0, float v1, float v2)

void Uniform4(int location, float v0, float v1, float v2, float v3)

void UniformMatrix2 (int location, int count, bool transpose, float[] value)

void UniformMatrix3 (int location, int count, bool transpose, float[] value)

void UniformMatrix4 (int location, int count, bool transpose, float[] value)

Как очевидно из названия, последние 3 предназначены для матриц, первая для простых переменных, а оставшиеся для векторов.

- location - идентификатор переменной (полученный при помощи GetUniformLocation).

Для аттрибутов принцип передачи параметров похож на принцип рендинга VA\VB. Вначале информируем OGL, что будут использоваться соответствующие переменные в установленной шейдерной программы.

void EnableVertexAttribArray (uint index)

- index - идентификатор переменной (полученный при помощи GetAttribLocation).

Затем устанавливаем указатель или смещение для VB на соответсвующие данные при помощи метода:

void VertexAttribPointer (uint index, int size, bool normalized, int stride, [] pointer)

- index - идентификатор переменной (полученный при помощи GetAttribLocation).
- pointer - указатель на первый элемент соответствующих элементов в массиве данных (например, &vertices[0].vx, &vertices[0].nx, &vertices[0].r)
- size - размер векторов содержащихся в массиве. Иными словами количество элементов описывающих величину (например 3 если вершины задаются как {x,y,z} или 4 если задаются как {x,y,z,w}. Аналогично с цветом в зависимости от того как он задается {r,g,b или {a,r,g,b})
- stride - шаг, то есть размер между элементами данных. Обычно равен размеру всей структуры - sizeof(Vertex) или 0, если элементы в памяти расположены последовательно.

По окончанию отображения данных информируем о прекращении использования данных размещенных в оперативной памяти при помощи метода:

void DisableVertexAttribArray (uint index), параметр array аналогичен одноименному параметру в методе EnableVertexAttribArray () описанном выше.

Приложение: проверка ошибок компиляции

-

```
var errorhandler = new Action<string, object, object>((format, arg0, arg1) => {
   string errormessage = String.Format(format, arg0, arg1);
   Trace.WriteLine(errormessage);
   throw new Exception(errormessage);
   MessageBox.Show(errormessage, "SHADER CREATION ERROR", MessageBoxButtons.OK,
MessageBoxIcon.Error);
   Application.Exit();
});


var compile_shader = new Action<uint>(shader => {
    gl.CompileShader(shader);

    // Проверяем была ли компиляция выполнена успешно
    gl.GetShader(shader, OpenGL.GL_COMPILE_STATUS, parameters);
    if (parameters[0] != OpenGL.GL_TRUE)
    {
    // В случае если компиляция не удалась пытаемся добиться от OGL
    // что именно ему не понравилось. Для этого вначале получаем длину
    // сообщения, выделим память под него, а затем уже запрашиваем
    // скопировать туда само сообщение. В случае C# это выглядит
    // немного иначе, но суть таже.
    gl.GetShader(shader, OpenGL.GL_INFO_LOG_LENGTH, parameters);
    StringBuilder strbuilder = new StringBuilder(parameters[0]);
    gl.GetShaderInfoLog(shader, parameters[0], IntPtr.Zero, strbuilder);
    errorhandler("OpenGL Error: ошибка во время компиляции {1}.\n{0}",
                 strbuilder.ToString(), shader==vert_shader?"VERTEX_SHADER"
                 :shader==frag_shader?"FRAGMENT_SHADER":"??????????_SHADER");
    }
});

... ...

   gl.LinkProgram(shader_program);
   gl.GetProgram(shader_program, OpenGL.GL_LINK_STATUS, parameters);
   if (parameters[0] != OpenGL.GL_TRUE)
       errorhandler("OpenGL Error: ошибка линковки програмы шейдера", null,
null);
```

Приложение: пример шейдера

```
// Vertex Shader

#version 330

varying  vec3 pass_color;

void main(void) {
     gl_FragColor  = vec4(pass_color, 1.0);
}



// Fragment shader

#version 330

uniform uint curtick;
attribute vec3 color;
varying   vec3 pass_color;

const float scale = 2.14;

void main(void) {
     gl_Position *= scale;
     pass_color = color;
}
```