Для лабораторных работ рекомендуется использовать VisualStudio 2010 или более поздние версии.

Список сокращений:

ЛКМ – левая кнопка мыши.

ПКМ – правая кнопка мыши.

ВВЕДЕНИЕ В GLSL

ЧТО ТАКОЕ ШЕЙДЕРЫ

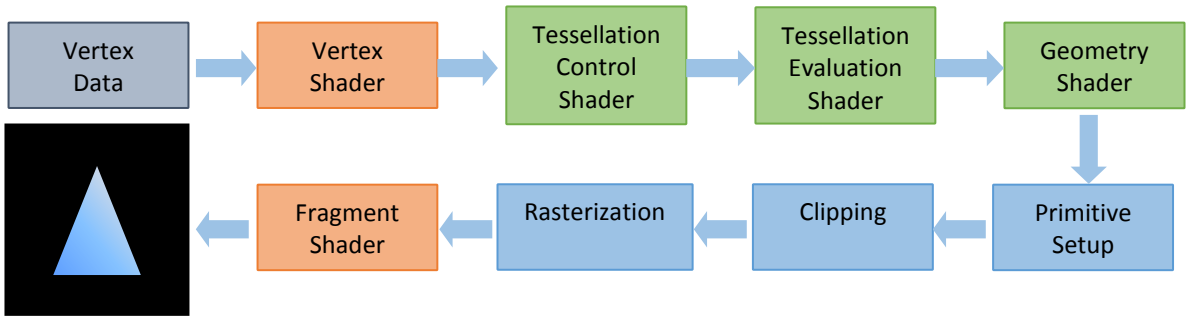
Шейдеры – это компонент шейдерной программы, мини-программа, которая выполняется в рамках отдельного этапа в общем конвейере OpenGL, выполняются непосредственно на GPU и действуют параллельно. Шейдерные программы предназначены для замены части архитектуры OpenGL, которую называют конвейером с фиксированной функциональностью. С версии OpenGL 3.1 фиксированная функциональность была удалена и шейдеры стали обязательными. Шейдер — специальная подпрограмма, выполняемая на GPU. Шейдеры для OpenGL пишутся на специализированном C-подобном языке — GLSL. Они компилируются самим OpenGL перед использованием.

Шейдерная программа объединяет набор шейдеров. В простейшем случае шейдерная программа состоит из двух шейдеров: вершинного и фрагментного.

Вершинный шейдер вызывается для каждой вершины. Его выходные данные интерполируются и поступают на вход фрагментного шейдера. Обычно, работа вершинного шейдера состоит в том, чтобы перевести координаты вершин из пространства сцены в пространство экрана и выполнить вспомогательные расчёты для фрагментного шейдера.

Фрагментный шейдер вызывается для каждого графического фрагмента (пикселя растеризованной геометрии, попадающего на экран). Выходом фрагментного шейдера, как правило, является цвет фрагмента, идущий в буфер цвета. На фрагментный шейдер обычно ложится основная часть расчёта освещения.

Виды шейдеров: вершинный, тесселяции, геометрический, фрагментный.



Графический конвейер 4.3

GLSL (OPENGL SHADING LANGUAGE)

Язык высокого уровня для программирования шейдеров. Номер версии GLSL соответствует версии OpenGL. GLSL был спроектирован для совместного использования с OpenGL. GLSL имеет встроенные возможности доступа к состоянию OpenGL.

Открытый межплатформенный стандарт. Нет других шейдерных языков, являющихся частью межплатформенного стандарта. GLSL может быть реализован разными производителями на произвольных платформах.

Компиляция исходного кода во время выполнения. Отсутствие дополнительных библиотек и программ. Все необходимое – язык шейдеров, компилятор и компоновщик – определены как часть OpenGL.

ТИПЫ ДАННЫХ GLSL.

ОБЪЯВЛЕНИЕ ПЕРЕМЕННЫХ

Переменные GLSL такие же, как в C++ : могут быть объявлены по необходимости и имеют ту же область видимости:

```
float f;
f = 3.0;
vec4 u, v;
for (int i = 0; i < 10; ++i)
    v = f * u + v;
```

void	для функций, которые не возвращают значение
bool	два значения true или false

СКАЛЯРНЫЕ ТИПЫ ДАННЫХ

int	знаковое целое
uint	беззнаковое целое
float	число с плавающей точкой
double	число с плавающей точкой двойной точности

ВЕКТОРНЫЕ ТИПЫ ДАННЫХ

Образование наименования типа: <скалярный тип данных> vec <количество компонент>. Например, bvec4.

Для доступа к компонентам вектора можно воспользоваться двумя способами:

- обращение по индексу;
- обращение к полям структуры (x, y, z, w или r, g, b, a или s, t, p, q)

vec(2|3|4)	2,3,4 компонентный вектор чисел с плавающей точкой.
dvec(2|3|4)	2,3,4 компонентный вектор чисел с плавающей точкой двойной точности
bvec(2|3|4)	2,3,4 компонентный вектор типа bool
ivec(2|3|4)	2,3,4 компонентный вектор целых чисел
uvec(2|3|4)	2,3,4 компонентный вектор беззнаковых целых

МАТРИЧНЫЕ ТИПЫ ДАННЫХ

При выполнении операций над этими типами данных они всегда рассматриваются как математические матрицы. В частности, при перемножения матрицы и вектора получаются правильные с математической точки зрения результаты. Матрица хранится по столбцам и может рассматриваться как массив столбцов.

mat(2|3|4) mat2x2 , mat3x3, mat4x4	2×2, 3×3, 4×4 матрица чисел с плавающей точкой
mat2x3, mat2x4	матрица чисел с плавающей точкой с 2 столбцами и 3, 4 строками
mat3x2, mat3x4	матрица чисел с плавающей точкой с 3 столбцами и 2, 4 строками
mat4x2, mat4x3	матрица чисел с плавающей точкой с 4 столбцами и 2, 3 строками
dmat(2|3|4) dmat2x2 , dmat3x3, dmat4x4	2×2, 3×3, 4×4 матрица чисел с плавающей точкой двойной точности
dmat2x3, dmat2x4	матрица чисел с плавающей точкой двойной точности с 2 столбцами и 3, 4 строками
dmat3x2, dmat3x4	матрица чисел с плавающей точкой двойной точности с 3 столбцами и 2, 4 строками
dmat4x2, dmat4x3	матрица чисел с плавающей точкой двойной точности с 4 столбцами и 2, 3 строками

ИНИЦИАЛИЗАТОРЫ И КОНСТРУКТОРЫ

При объявлении переменных их можно инициализировать начальными значениями, подобно языкам C/C++:

```
float f = 3.0;
bool b = false;
int i = 0;
```

При объявлении сложных типов данных используются конструкторы. Они же применяются для преобразования типов:

```
vec2 pos = vec2(1.0, 0.0);
vec4 color = vec4(pos, 0.0, 1.0);
vec3 color3 = vec3(color);
bool b = bool(1.0);
```

СТРУКТУРЫ

Структуры на языке шейдеров OpenGL похожи на структуры языка C/C++:

```
struct Light
{
    vec3 position;
    vec3 color;
}
...
Light pointLight;
```

Все прочие особенности работы со структурами такие же, как в C. Ключевые слова union, enum и class не используются, но зарезервированы для возможного применения в будущем.

МАССИВЫ

Массивы. В языке шейдеров OpenGL можно создавать массивы любых типов:

```
float values[10];
vec4 points[];
vec4 points[5];
```

Принципы работы с массивами те же, что и в языках C/C++.

ШАГ1

За основу берем приложение OpenGL из предыдущей темы. Будем рисовать треугольник с разноцветными вершинами.

Шейдеры – это два текстовых файла. Их нужно загрузить с диска и скомпилировать в шейдерную программу. Создадим два пустых текстовых файла «basic.vs» для вершинного шейдера и «basic.fs» - для фрагментного.

УПРАВЛЯЮЩАЯ ПРОГРАММА

Определение версий GLSL и OpenGL.

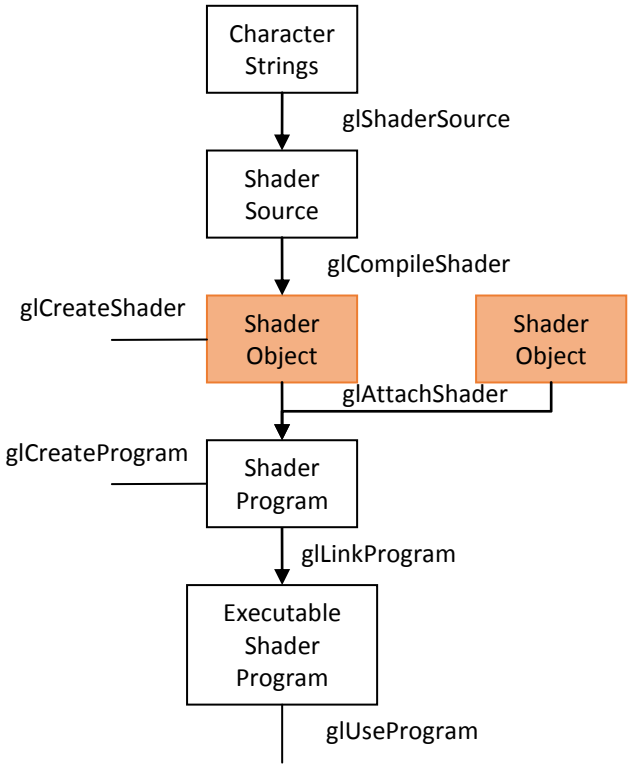
Чтобы определить версии GLSL и OpenGL, поддерживаемые текущим драйвером достаточно воспользоваться функцией glGetString.

```
string glVersion = GL.GetString(StringName.Version);
string glslVersion = GL.GetString(StringName.ShadingLanguageVersion);
```

Для хранения дескриптов шейдерной программы и шейдерных объектов, добавляем членами класса переменные:

```
int BasicProgramID;
int BasicVertexShader;
int BasicFragmentShader;
```

ЗАГРУЗКА ШЕЙДЕРОВ




```
void loadShader(String filename, ShaderType type, int program, out int address)
{

    address = GL.CreateShader(type);
    using (System.IO.StreamReader sr = new StreamReader(filename))
    {
        GL.ShaderSource(address, sr.ReadToEnd());
    }
    GL.CompileShader(address);
    GL.AttachShader(program, address);
    Console.WriteLine(GL.GetShaderInfoLog(address));
}
```

glCreateShader создаёт объект шейдера, её аргумент определяет тип шейдера, который может принимать значения из перечисления ShaderType: ComputeShader, FragmentShader, GeometryShader, GeometryShaderExt, TessControlShader, TessEvaluationShader, VertexShader. Функция возвращает значение, которое можно использовать для ссылки на объект вершинного шейдера (дескриптор). Если во время создания объекта шейдера возникнет ошибка, функция вернет 0.

glShaderSource загружает исходный код в созданный шейдерный объект. Эта функция может принимать массив строк, поэтому есть возможность компилировать код сразу из множества источников. Эта функция копирует исходный код во внутреннюю память программы, занятую исходным кодом, можно освободить.

Далее надо скомпилировать исходный шейдер, делается это вызовом функции glCompileShader() и передачей ей дескриптора шейдера, который требуется скомпилировать.

Перед тем, как шейдеры будут добавлены в конвейер OpenGL их нужно скомпоновать в шейдерную программу с помощью функции glAttachShader(). На этапе компоновкипроизводится стыковка входных переменных одного шейдера с выходными переменными другого, а также стыковка входных/выходных переменных шейдеров с соответствующими областями памяти в окружении OpenGL.

ИНИЦИАЛИЗАЦИЯ ШЕЙДЕРНОЙ ПРОГРАММЫ

```
private void InitShaders()
{
    // создание объекта программы
    BasicProgramID = GL.CreateProgram();
    loadShader("..\\..\\Shaders\\basic.vert", ShaderType.VertexShader, BasicProgramID,
               out BasicVertexShader);
    loadShader("..\\..\\Shaders\\basic.frag", ShaderType.FragmentShader, BasicProgramID,
                out BasicFragmentShader);
    //Компановка программы
    GL.LinkProgram(BasicProgramID);

    // Проверить успех компановки
    int status = 0;
    GL.GetProgram(BasicProgramID, GetProgramParameterName.LinkStatus, out status);

    Console.WriteLine(GL.GetProgramInfoLog(BasicProgramID));
}
```

Если программа стала не нужна, её можно удалить из память OpenGL вызовом функции glDeleteProgram(). Она делает дескриптор программы недействительным и освобождает память, занятую программой. Если в момент удаления программа используется, то она будет удалена только тогда, когда перестанет использоваться.

ВЕРШИННЫЙ ШЕЙДЕР.

Вершинный шейдер вызывается один раз для каждой вершины. Чтобы передать вершинному шейдеру что-то для обработки, необходим некоторый механизм передачи исходных данных (для каждой вершины) в шейдер. Обычно такие исходные данные включают: координаты вершины, вектор нормали и координаты текстуры. В версиях OpenGL до 3.0 для передачи каждого элемента информации о вершине имелись отдельные «каналы» - различные функции, такие как glVertex, glNormal, glTexCoord. Шейдер мог обращаться к значениям, переданным таким способом, посредством внутренних переменных, таких как gl_Vertex и gl_Normal. Но в версии OpenGL 3.0 эта функциональность была объявлена устаревшей и позднее удалена из библиотеки. Теперь информация о вершинах должна передаваться через универсальные вершинные атрибуты, обычно в комплексе с вершинными буферными объектами (vertex buffer object). Основная программа связывает буферный объект с входным атрибутом и определяет, как шагать по данным. Программист сейчас должен определить произвольный набор атрибутов для каждой вершины.

В вершинном шейдере определение входных атрибутов выполняется с использованием квалификатора in.

```
#version 430
/* Входные атрибуты для вершинного шейдера. Ключевое слово in */
layout (location = 0) in vec3 VertexPosition;
layout (location = 1) in vec3 VertexColor;

/*Выходная переменная. Ключевое слово out*/
out vec3 Color;

void main()
```

```
{
	Color = VertexColor;
	/*Встроенная выходная переменная*/
	gl_Position = vec4(VertexPosition, 1.0);
}
```

В нашем вершинном шейдере определена одна выходная переменная Color, значение которой будет передаваться во фрагментный шейдер. Также есть встроенная выходная переменная gl_Position, куда передается позиция вершины.

ФРАГМЕНТНЫЙ ШЕЙДЕР

```
in vec3 Color;
/*Будет содержать значение, полученное в результате интерполяции цветов трех вершин треугольника.*/
out vec4 FragColor;

void main()
{
	FragColor = vec4(Color, 1.0);
}
```

Во фрагментном шейдере есть одна входная переменная Color,она связана с соответствующей выходной переменной в вершинном шейдере и будет содержать значение, полученное в результате интерполяции цветов трех вершин треугольника. Просто дополняем копию цвета ещё одним компонентом и передаём в выходную переменную FragColor (gl_FragColor в предыдущих версиях).

По аналогии с входными переменными в вершинном шейдере этой переменной также должен быть назначен соответствующий индекс. Эта переменная должна быть связана с фоновым буфером цвета, по умолчанию имеющим нулевой индекс. Если выходная переменная одна, то компоновщик автоматически присвоит нулевой индекс. Но можно присвоить индекс явным образом:

```
layout (location = 0) out vec4 FragColor;
```

Благодаря такому подходу можно определить во фрагментном шейдере множество выходных переменных.

ДРУГОЙ СПОСОБ НАЗНАЧЕНИЯ ИНДЕКСОВ АТРИБУТАМ.

Можно присвоить индексы атрибутам в управляющей программы с помощью вызовов функций glBindAttribLocation() для входных атрибутов и glBindFragDataLocation() для выходных атрибутов.

```
glBindAttribLocation(programHandle, 0, “VertexPosition”);
glBindAttribLocation(programHandle, 1, “VertexColor”);
glBindFragDataLocation(programHandle, 1, “FragColor”);
```

НАСТРОЙКА БУФЕРНЫХ ОБЪЕКТОВ В УПРАВЛЯЮЩЕЙ ПРОГРАММЕ.

Сначала создайте член класса int vaoHandle для хранения дескриптора объекта массива вершин.

Дальнейший код можно поместить в функцию InitShaders(). Сначала создаем массивы в которых будут содержаться параметры вершин:

```
float[] positionData = { -0.8f, -0.8f, 0.0f, 0.8f, -0.8f, 0.0f, 0.0f, 0.8f, 0.0f };
float[] colorData = { 1.0f, 0.0f, 0.0f, 0.0f, 1.0f, 0.0f, 0.0f, 0.0f, 1.0f };
```

glGenBuffers создаёт два буферных объекта. Дескрипторы сохраняются в массиве для последующего использования.

```
int[] vboHandlers = new int[2];
GL.GenBuffers(2, vboHandlers);
```

Каждый буферный объект связывается с точкой привязки GL_ARRAY_BUFFER (BufferTarget.ArrayBuffer) вызовом glBindBuffer. Все возможные точки привязки содержатся в перечислении BufferTarget.

```
GL.BindBuffer(BufferTarget.ArrayBuffer, vboHandlers[0]);
```

Для заполнения массивов данными вызывается функция glBufferData. glBufferData выделяет память под данные. Третий параметр подсказывает библиотеке OpenGL как будут использоваться данные, чтобы она могла решить как лучше организовать управление внутренним буфером. Имя параметра составлено из двух частей, первая может быть: STATIC, DYNAMIC или STREAM, а вторая может быть DRAW, READ или COPY.

STATIC данные будут записаны один раз, и использованы многократно, используется для данных, которые задаются один раз при инициализации.

DYNAMIC данные будут использованы много раз, прежде чем поменяются.

STREAM данные будут меняться каждый фрейм или одни данные будут отрисованы всего несколько раз.

_DRAW данные изменяются управляющей программой и используются как источник данных для отрисовки.

_READ данные будут читаться из памяти OpenGL по запросу, следует использовать для буферных объектов пикселей и других буферов, которые будут использоваться для получения информации из OpenGL.

_COPY данные используются для отрисовки и для чтения из OpenGL, приложение будет использовать OpenGL для генерации данных, которые будут помещены в буфер, который затем будет использоваться в качестве источника для последующих операций рисования.

```
GL.BufferData(BufferTarget.ArrayBuffer,
              (IntPtr)(sizeof(float) * positionData.Length),
              positionData, BufferUsageHint.StaticDraw);
GL.BindBuffer(BufferTarget.ArrayBuffer, vboHandlers[1]);
GL.BufferData(BufferTarget.ArrayBuffer,
              (IntPtr)(sizeof(float) * colorData.Length),
              colorData, BufferUsageHint.StaticDraw);
```

После настройки буферных объектов их нужно объединить в объект массива вершин (vertex array object, VAO). VAO содержит информацию о связях между данными в буферах и входными атрибутами вершин. glGenVertexArray() создаёт объект VAO и возвращает его дескриптор.

```
vaoHandle = GL.GenVertexArray();
GL.BindVertexArray(vaoHandle);
```

Активация атрибутов вершин с индексами 1 и 0:

```
GL.EnableVertexAttribArray(0);
GL.EnableVertexAttribArray(1);
```

Снова связываем буферный объект с целевой точкой привязки, а потом устанавливается связь между буферными объектами и индексами атрибутов. Первый аргумент – это индекс атрибута, второй – число компонент на каждую вершину (от 1 до 4), четвёртый – логическое значение, определяющее необходимость нормализации данных (необходимость отображения целочисленных значений со знаком в диапазон [-1,1] и целочисленных значений без знака в диапазон [0,1]). Пятый аргумент – это шаг, смещение в байтах между соседними атрибутами. Последний аргумент – смещение в байтах первого атрибута в буфере относительно начала этого буфера, на тот случай если в буфере перед первым элементом записаны дополнительные данные. Настройка объекта VBO – однократная операция.

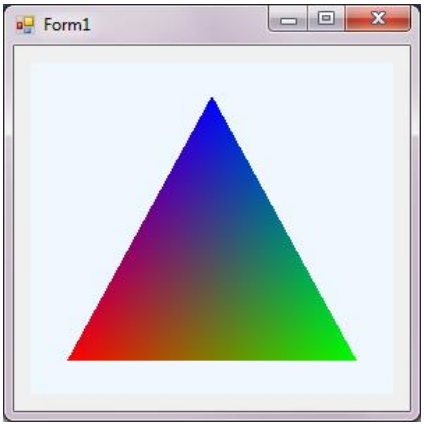
```
GL.BindBuffer(BufferTarget.ArrayBuffer, vboHandlers[0]);
GL.VertexAttribPointer(0, 3, VertexAttribPointerType.Float, false, 0, 0);
GL.BindBuffer(BufferTarget.ArrayBuffer, vboHandlers[1]);
GL.VertexAttribPointer(1, 3, VertexAttribPointerType.Float, false, 0, 0);
```

В функцию Draw добавьте следующие инструкции:

```
GL.UseProgram(BasicProgramID);
GL.DrawArrays(PrimitiveType.Triangles, 0, 3);
openGlControl.SwapBuffers();
GL.UseProgram(0);
```

glUseProgram() делает активной шейдерную программу BasicProgramID, всё, что будет отрисовано до вызова следующей glUseProgram() будет отрисовано с помощью этой шейдерной программы. glDrawBuffers() запускает отрисовку буферов, выполняя обход всех активных атрибутов и передавая данные по конвейеру в вершинный шейдер. Первый аргумент определяет режим отображения, во втором указывается первый активный индекс, в третьем – число вершин для отображения.

Запускаем, получаем треугольник:



ССЫЛКИ:

1. Вольф Д. OpenGL 4. Язык шейдеров. Книга рецептов / пер. с англ. А.Н.Киселева. – М.: ДМК Пресс, 2015. – 368 с.