Core OpenGL - Setup

Notes for a Msc Course in Computer Graphics

University of Minho
António Ramires

Introduction

OpenGL setup includes building the pipeline, feeding it, and in some situations retrieving the results.

This short tutorial covers the initial part of the OpenGL setup, for an application to be able to build a simple pipeline with vertex and a fragment shader, and send all the relevant data to the graphics pipeline.

Perhaps the most common usage of OpenGL is to render graphics. The example we will use in here is therefore directed to rendering purposes. The input will be the data required to draw a cube, the output will be a rendered cube. The shaders are very minimalist on purpose.

Although simple, this example will show all the most common aspects of OpenGL+GLSL usage.

Building the pipeline

First some terminology. The graphics pipeline is a sequence of stages, where some of these stages are programmable. There are many possible configurations for the pipeline, since not all the programmable stages are mandatory, only the first programmable stage is required, acting as the input of the graphics pipeline. The remaining programmable stages serve specific purposes, hence their usage is not required in all situations.

A *shader* is a (small) program that will be executed in the GPU, in a programmable stage. A *program* can be seen as a sequence of *shaders*.

Currently there are five types of shader programs: vertex, two for tessellation, geometry, and fragment. Figure 1 depicts a simplified diagram of the pipeline. The blue boxes represent the programmable stages, and the white boxes are for those stages (still) using fixed function.

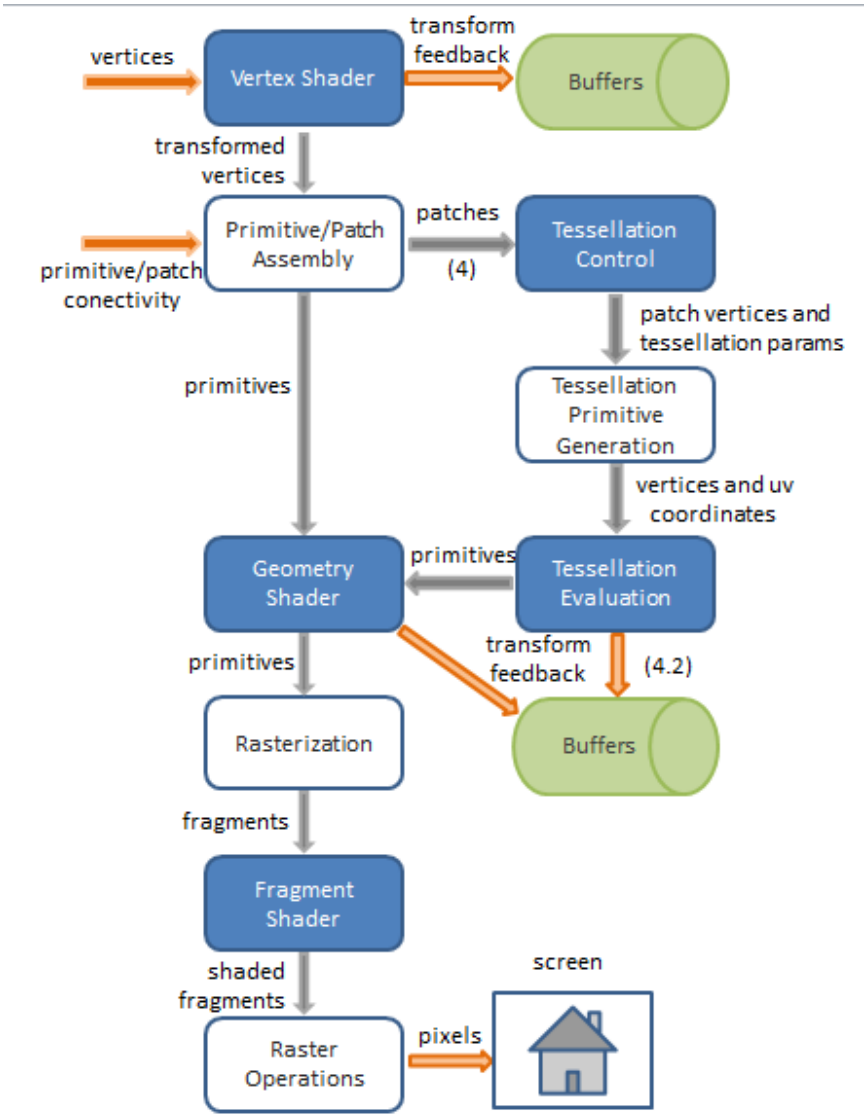

Figure 1 - Simplified pipeline diagram

This short tutorial only covers the setup so we're not going to talk about the shaders themselves. Each programmable stage can have a shader program, that must be compiled separately. The shaders are then attached to a program, which in turn is linked.

Our pipeline is going be very simple, having only a vertex shader (mandatory) and a fragment shader. We are going to assume that the source code for the shader programs is stored in two separate files. Figure 2 shows a diagram of the required steps.

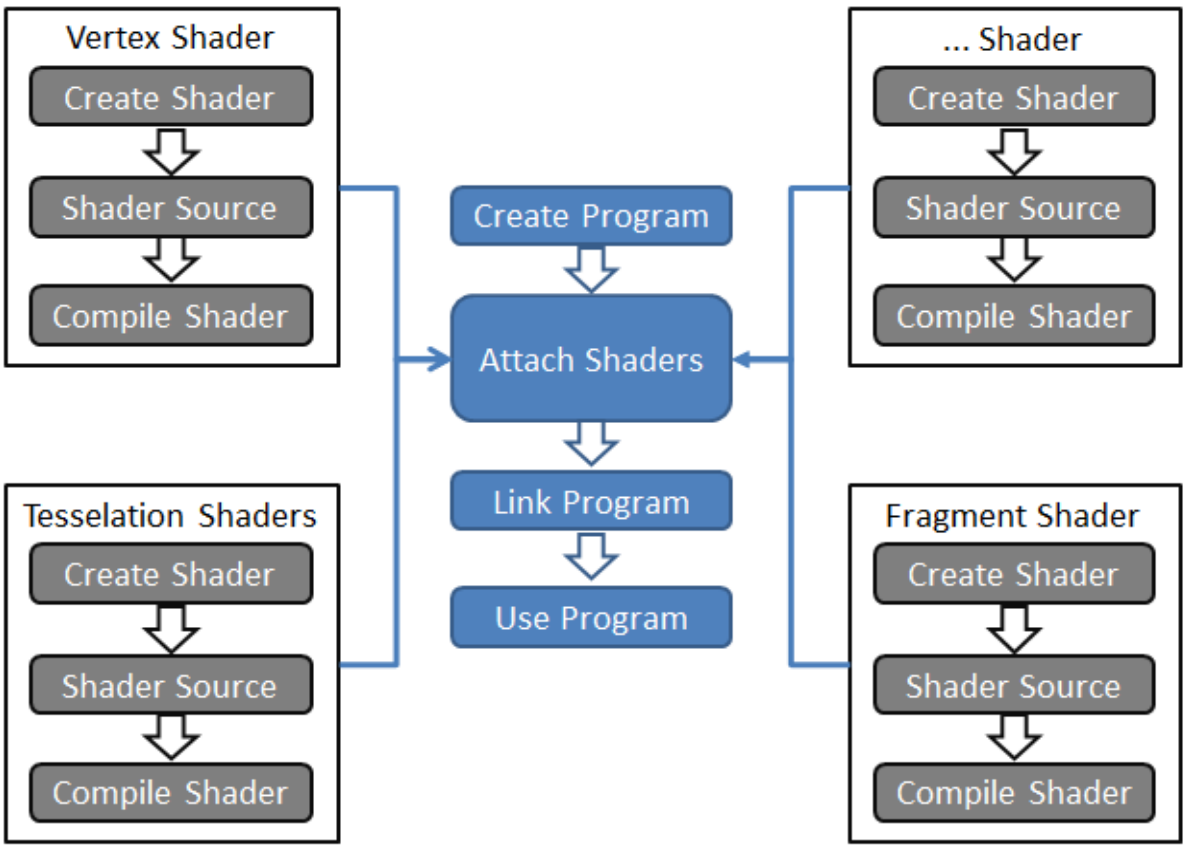

Figure 2 - Building a program

The following snippet of code shows how to do it in practice, creating a vertex shader, and a program, ending up with a linked program (as long as the shader is correctly written) :

```
GLuint f,v;

v = glCreateShader(GL_VERTEX_SHADER);

vs = textFileRead(vertexFileName);
const char * vv = vs;
glShaderSource(v, 1, &vv,NULL);
free(vs)

glCompileShader(v);

GLunit p;

p = glCreateProgram();
```

```
glAttachShader(p,v);
glAttachShader(p,f);
glLinkProgram(p);
```

Feeding the pipeline

To use core OpenGL, the graphics pipeline must be fed with data, which will be processed by shaders. The outcome of this processing can be an image, or just data with no graphical representation.

To draw a cube we need to specify its vertices, and how they are connected to define faces. Furthermore, vertices have attributes, besides position, namely normals and texture coordinates.

To define these attributes, we initially place the data in an array. In here we are going to consider an array for each attribute, although these can also be interleaved in a single array.

```
// Data for a set of triangles
float position[ ] = {-1.0f, 0.0f, -5.0f, 1.0f,
            1.0f, 0.0f, -5.0f, 1.0f,
            0.0f, 2.0f, -5.0f, 1.0f, …};

float textureCoord[ ] = { … };

float normal[ ] = { … };
```

Considering the index *i* for all arrays we get the set of attributes for vertex *i*.

Defining an array of indices allows us to connect these vertices arbitrarily to create primitives. So we can add the following array with indices:

```
unsigned integer index[ ] = {0, 1, 2, …};
```

For each input attribute, we need to get its handle on the pipeline. In OpenGL these handles are called “locations”. So assuming that we’ve created a program named *p*, successfully linked it, and that this program contains an input attribute named *pos*, then we can write:

```
posLoc = glGetAttribLocation(p,"pos");
```

The next step is to create a set of buffers with this data. OpenGL allows us to define an array of buffers using a Vertex Array Object, or VAO.

First we create a VAO object and bind it:

```
GLuint vao;
glGenVertexArrays(1, &vao);
glBindVertexArray(vao);
```

For each individual buffer, we can then proceed to do as follows: first we create each buffer, bind it, fill it with data, and finally associate it with an input attribute of the graphics pipeline.

```
GLuint buffer;
glGenBuffers(1, &buffer);

// bind buffer for positions and copy data into buffer
glBindBuffer(GL_ARRAY_BUFFER, buffer);
glBufferData(GL_ARRAY_BUFFER, sizeof(position),
                      position, GL_STATIC_DRAW);
glEnableVertexAttribArray(posLoc );
glVertexAttribPointer(posLoc , 4, GL_FLOAT, 0, 0, 0);
```

For the index array, a similar approach is used, the main difference being in the type of the buffer:

```
GLuint buffer;
glGenBuffers(1, &buffer);

// bind buffer for positions and copy data into buffer
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, buffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(index),
                      index, GL_STATIC_DRAW);
```

As described, attributes are local variables for a vertex, meaning their values may change per vertex. GLSL also allows the definition of global variables, called “uniforms”. These are variables that are constant, at least for each draw call.

As for attributes, it is required to get a handle for uniforms. So lets assume we have a uniform variable named *myMatrix*. To set its value, we must first get its location, as shown below:

```
float mat[16];
loc = glGetUniformLocation(p, “myMatrix”);
glProgramUniformMatrix4fv(p, loc, sizeof(float)*16, false, mat);
```

All the above steps could be part of the setup bit of the application. In the rendering function, we will ask OpenGL to draw the above defined geometry. To achieve this we can use the following, for VAOs containing an index buffer:

```
glUseProgram(p);
glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, index.size(),
                      GL_UNSIGNED_INT, NULL);
```

or, considering VAOs without indices:

```
glUseProgram(p);
glBindVertexArray(vao);
glDrawArrays(GL_TRIANGLES, 0, count);
```