Assignment 2: Shape and Shader

CS321 Spring 2018

Due Date: 2018.3.15.

Follow the instructions carefully. If you encounter any problems in the setup, please do not hesitate to reach out to TA.

Introduction

In OpenGL everything is in 3D space, but the screen and window are a 2D, so a large part of OpenGL's work is about transforming all 3D coordinates to 2D pixels that fit on your screen. The process of transforming 3D coordinates to 2D coordinates is managed by the graphics pipeline of OpenGL. The graphics pipeline can be divided into two large parts: the first transforms your 3D coordinates into 2D coordinates and the second part transforms the 2D coordinates into actual colored pixels. In this lab, we'll briefly discuss how we can create some fancy pixels on the screen.

Shader

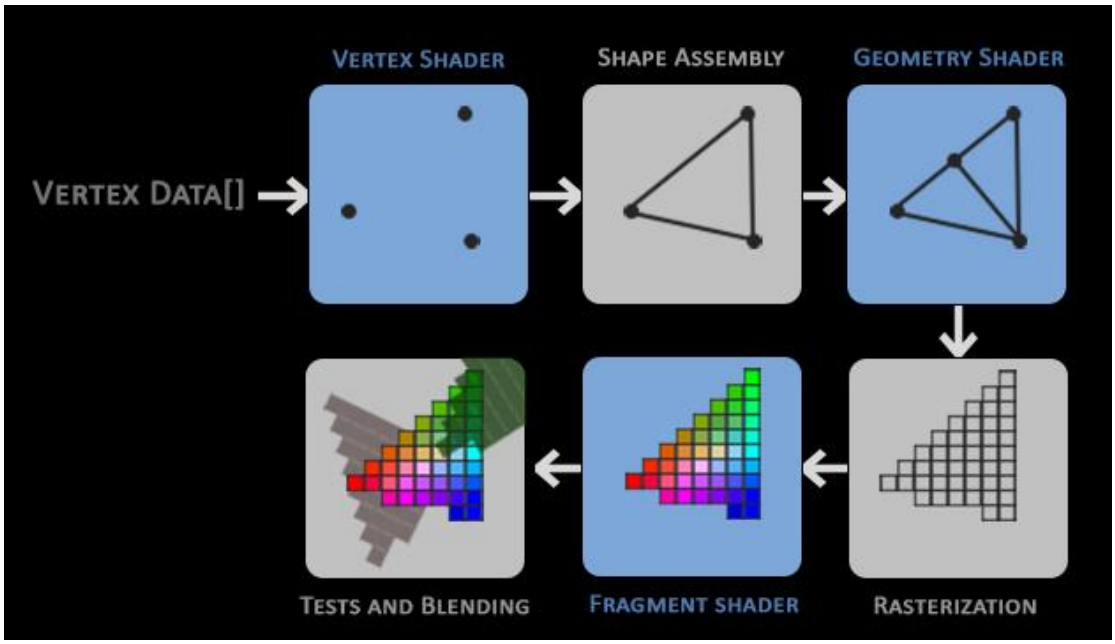
The graphics pipeline takes a set of 3D coordinates as input and transforms these to colored 2D pixels on your screen. The graphics pipeline can be divided into several steps where each step requires the output of the previous step as its input. All these steps are highly specialized (they have one specific function) and can easily be executed in parallel. Because of their parallel nature most graphics cards of today have thousands of small processing cores to quickly process your data within the graphics pipeline by running small programs on the GPU for each step of the pipeline. These small programs are called *shaders*.

Some of these shaders are configurable by the developer which allows us to write our own shaders to replace the existing default shaders. This gives us much more fine-grained control over specific parts of the pipeline and because they run on the GPU, they can also save us valuable CPU time. Shaders are written in the OpenGL Shading Language (GLSL).

The figure below is an abstract representation of all the stages of the graphics pipeline. As you can see the graphics pipeline contains many sections that each handle one specific part of converting your vertex data to a fully rendered pixel.

As input to the graphics pipeline we pass in a list of three 3D coordinates that should form a triangle in an array here called Vertex Data; this vertex data is a collection of vertices. A vertex is basically a collection of data per 3D coordinate. This vertex's data is represented using vertex attributes that can contain any data we'd like but for simplicity's sake let's assume that each vertex consists of just a 3D position and some

color value.



The graphics pipeline is quite a complex process and contains many configurable parts. However, for almost all the cases we only need to work with the vertex and fragment shader. The geometry shader is optional and usually left to its default shader.

In Modern OpenGL we are required to define at least a vertex and fragment shader of our own (there are no default vertex/fragment shaders on the GPU). For this reason, it is often quite difficult to start learning Modern OpenGL since a great deal of knowledge is required before being able to render your first triangle.

VBO (Vertex Buffer Object)

With the vertex data defined we'd like to send it as input to the first process of the graphics pipeline: the vertex shader. This is done by creating memory on the GPU where we store the vertex data, configure how OpenGL should interpret the memory and specify how to send the data to the graphics card. The vertex shader then processes as much vertices as we tell it to from its memory.

We manage this memory via so called vertex buffer objects (VBO) that can store a large number of vertices in the GPU's memory. The advantage of using those buffer objects is that we can send large batches of data all at once to the graphics card without having to send data a vertex a time. Sending data to the graphics card from the CPU is relatively slow, so wherever we can we try to send as much data as possible at once. Once the data is in the graphics card's memory the vertex shader has almost instant access to the vertices making it extremely fast.

A vertex buffer object is our first occurrence of an OpenGL object as we've discussed in

the OpenGL tutorial. Just like any object in OpenGL this buffer has a unique ID corresponding to that buffer, so we can generate one with a buffer ID using the `glGenBuffers` function:

```
unsigned int VBO;
glGenBuffers(1, &VBO);
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

Do this once your window is created (after the OpenGL Context creation) and before any other OpenGL call.

Then we need three 3D points in order to make a triangle

```
float vertices[] = {
    -0.5f, -0.5f, 0.0f,
     0.5f, -0.5f, 0.0f,
     0.0f,  0.5f, 0.0f
};
```

The next step is to give this triangle to OpenGL. We do this by creating a buffer:

```
glBufferData(GL_ARRAY_BUFFER,    sizeof(vertices),    vertices,
GL_STATIC_DRAW);
```

Vertex Shader

The vertex shader is one of the shaders that are programmable. Modern OpenGL requires that we at least set up a vertex and fragment shader if we want to do some rendering, so we will briefly introduce shaders and configure two very simple shaders for drawing our first triangle.

The first thing we need to do is write the vertex shader in the shader language GLSL (OpenGL Shading Language) and then compile this shader so we can use it in our application. Below you'll find the source code of a very basic vertex shader in GLSL:

```
#version 330 core
layout (location = 0) in vec3 aPos;

void main()
{
    gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

Fragment Shader

The fragment shader is the second and final shader we're going to create for rendering a triangle. The fragment shader is all about calculating the color output of your pixels. To keep things simple the fragment shader will always output an orangish color.

```
#version 330 core
out vec4 FragColor;

void main()
{
    FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
}
```

Compile a shader

We wrote the source code for the vertex shader (stored in a C string), but in order for OpenGL to use the shader it has to dynamically compile it at run-time from its source code.

The first thing we need to do is create a shader object, again referenced by an ID. So, we store the vertex shader as an unsigned int and create the shader with `glCreateShader`:

```
unsigned int vertexShader;
vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
glCompileShader(vertexShader);
```

Shader Program

A shader program object is the final linked version of multiple shaders combined. To use the recently compiled shaders we have to link them to a shader program object and then activate this shader program when rendering objects. The activated shader program's shaders will be used when we issue render calls.

When linking the shaders into a program it links the outputs of each shader to the inputs of the next shader. This is also where you'll get linking errors if your outputs and inputs do not match.

Creating a program object is easy:

```
unsigned int shaderProgram;
shaderProgram = glCreateProgram();
glAttachShader(shaderProgram, vertexShader);
glAttachShader(shaderProgram, fragmentShader);
glLinkProgram(shaderProgram);
```

Right now, we sent the input vertex data to the GPU and instructed the GPU how it should process the vertex data within a vertex and fragment shader. We're almost there, but not quite yet. OpenGL does not yet know how it should interpret the vertex data in memory and how it should connect the vertex data to the vertex shader's attributes. We'll be nice and tell OpenGL how to do that.

Link vertex attributes

The vertex shader allows us to specify any input we want in the form of vertex attributes and while this allows for great flexibility, it does mean we have to manually specify what part of our input data goes to which vertex attribute in the vertex shader. This means we have to specify how OpenGL should interpret the vertex data before rendering.

With this knowledge we can tell OpenGL how it should interpret the vertex data (per vertex attribute) using `glVertexAttribPointer`:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 *
sizeof(float), (void*)0);
glEnableVertexAttribArray(0);
```

We need to repeat this process every time we want to draw an object. It may not look like that much but imagine if we have over 5 vertex attributes and perhaps 100s of different objects (which is not uncommon). Binding the appropriate buffer objects and configuring all vertex attributes for each of those objects quickly becomes a cumbersome process. What if there was some way we could store all these state configurations into an object and simply bind this object to restore its state?

VAO (Vertex Array Object)

A vertex array object (also known as VAO) can be bound just like a vertex buffer object and any subsequent vertex attribute calls from that point on will be stored inside the VAO. This has the advantage that when configuring vertex attribute pointers, you only have to make those calls once and whenever we want to draw the object, we can just bind the corresponding VAO. This makes switching between different vertex data and attribute configurations as easy as binding a different VAO. All the state we just set is stored inside the VAO.

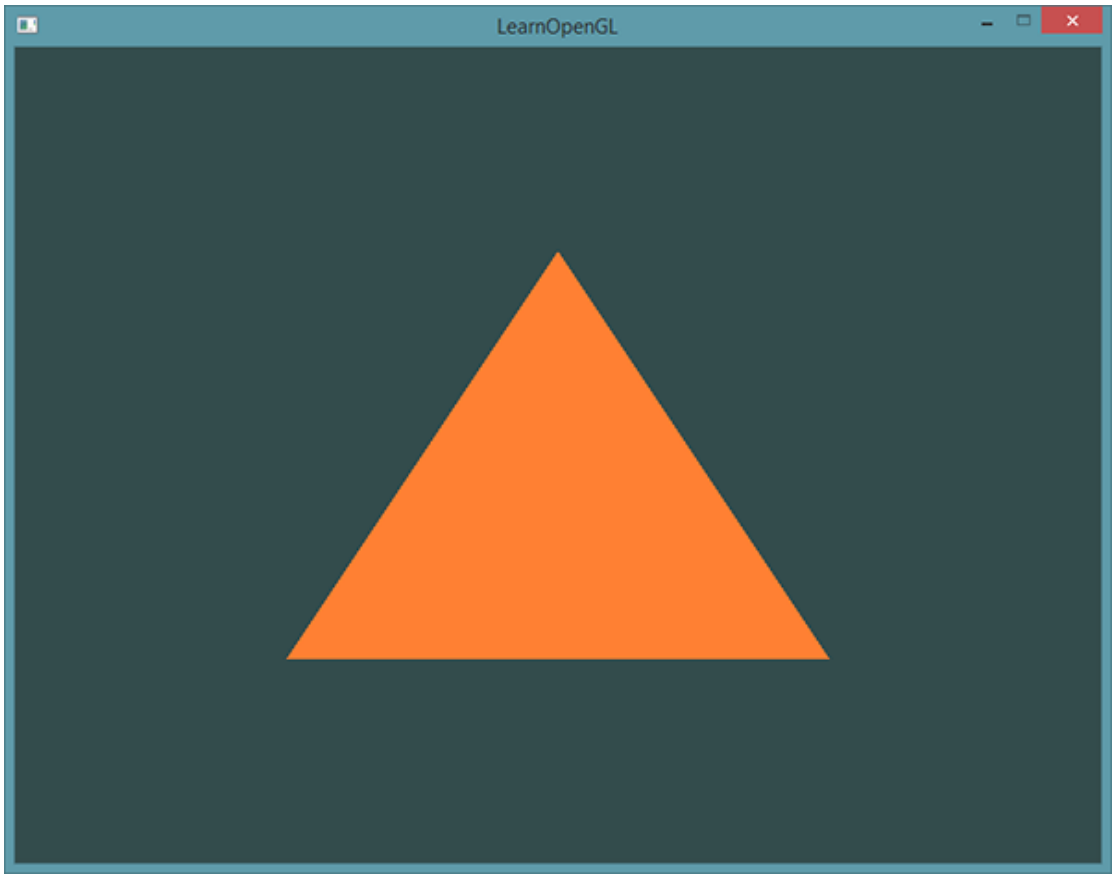
The process to generate a VAO looks simliar to that of a VBO:

```
unsigned int VAO;
glGenVertexArrays(1, &VAO);
```

To use a VAO all you need to do is bind the VAO using `glBindVertexArray`. From that point on we should bind/configure the corresponding VBO(s) and attribute pointer(s) and then unbind the VAO for later use. As soon as we want to draw an object, we simply bind the VAO with the preferred settings before drawing the object and that is it.

Result

Now try to compile the code and work your way backwards if any errors popped up. As soon as your application compiles, you should see the following result:



Exercise

1. Finish example.
2. Try to draw 2 triangles next to each other using glDrawArrays by adding more vertices to your data
3. Create two shader programs where the second program uses a different fragment shader that outputs some colors; draw both triangles again where one outputs some colors
4. Draw a cube with color.