Lab - 1
Introduction to Shaders
September 7, 2022

1 Introduction

In this lab we will see how to render a simple triangle. We will be learning how to write shaders, compile and link them. Further, it will be demonstrated what is VBO and VAO and how to use them. ALso render a simple primitive like triangle using it. The code to draw a triangle will be provided to you.

2 Graphics Pipeline

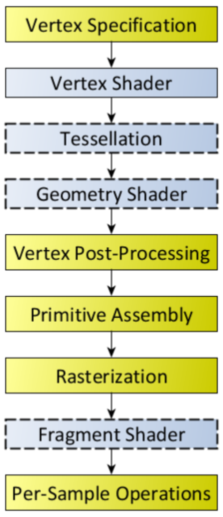

Figure 1: Graphics Pipeline

The OpenGL rendering pipeline is initiated when you perform a rendering operation. Rendering operations require the presence of a properly-defined vertex array object and a linked Program Object or Program Pipeline Object which provides the shaders for the programmable pipeline stages.

Once initiated, the pipeline operates in the following order:

- Vertex Processing:

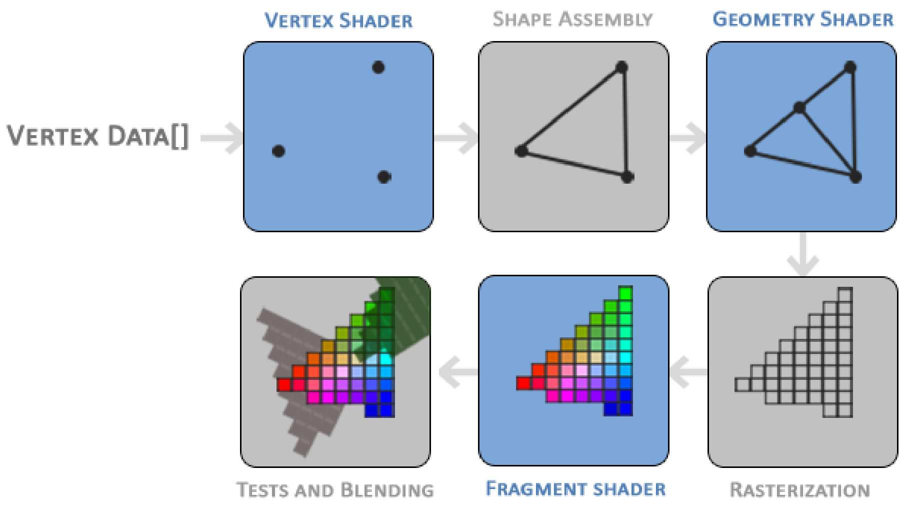

Figure 2: Graphics Pipeline Visualization

Each vertex retrieved from the vertex arrays (as defined by the VAO) is acted upon by a **Vertex Shader**. Each vertex in the stream is processed in turn into an output vertex.

Optional primitive **tessellation** stages.

- Vertex Post-Processing, the outputs of the last stage are adjusted or shipped to different locations.

  Transform Feedback happens here.

  Primitive Assembly

  Primitive Clipping, the perspective divide, and the **viewport transform** to window space.

- Scan conversion and primitive parameter interpolation, which generates a number of Fragments.

- A Fragment Shader processes each fragment. Each fragment generates a number of outputs.

- Per-Sample-Processing, including but not limited to

  Scissor Test

  Stencil Test

Depth Test

Blending

Logical Operation

Write Mask

3 Shaders

There are two primary shaders we will be covering in the duration of the course. The **vertex** shaders and the **fragment** shaders. Both these shaders carry out different tasks in the graphics pipeline. Atleast one vertex and fragement shader needs to be set up for the task of rendering.

3.1 Vertex Shader

The vertex shader is responsible for deciding the spatial location where rendering takes place. It does this with the help of vertex attributes namely, the position data such as a 3D vector containing the coordinates of a point. We declare the input vertex attributes with the in keyword. For eg. to declare an input vertex attr. for position can be done by -

```
in vec3 aPos; // declares a vec3 input vertex attrib.
```

The input value is then assigned to the predefined gl Position variable which is vec4 (4D vector). This is variable is used as the output of the vertex shader. The complete shader looks like this-

```
in vec3 aPos;
void main(){
gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);
}
```

3.2 Fragment Shader

The fragment shader is reponsible for the appearance and colors of the the rendered object (triangle in our case). We can fix the output color of the fragments with the following snippet.

```
out vec4 FragColor;
void main(){
```

```
FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);
// RGBA value for orange color
}
```

In the above code if you notice the declaration of the variable FragColor has a keyword out before it. This specifies that is it a value that will be returned from the fragment shader.

4 Shader Program

The shader program is the linked version of multiple shaders combined. Before the shaders are **linked** they are needed to be **compiled**. To use a shader it has to be dynamically compiled at run time from the source. To do this we create a shader object which is referenced by an ID. `glCreateShader` creates and returns the ID of a shader. This ID can be stored by a variable and later used for reference.

```
unsigned int vertexshader;
vertexshader = glCreateShader(GL_VERTEX_SHADER);
```

Next, we attach the shader source to this ID using `glShaderSource` and compile the shader with `glCompileShader`. After that you can cerate shader program by `glCreateShader` and attach them with the main function with `glAttachShader`. Finally complete it with `glLinkProgram`.

The program object can then be used post the call to `glUseProgram(program)` which activates it as the active program.

4.1 Linking Vertex Attribute

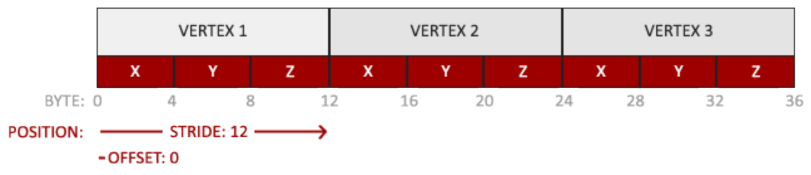

Figure 3: Position data stored as 32-bit(4byte) floating point for x,y,z values.

Once linking is done you would want to process data in the shaders which requires some form of communication. For vertex attribute the user has to specify in what form the data will be supplied to the vertex shader. This helps Opengl interpret vertex

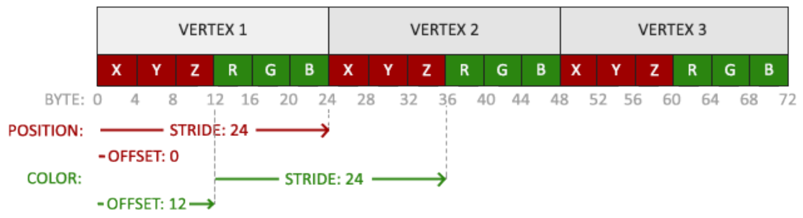
VERTEX 1
VERTEX 2
VERTEX 3
X Y Z R G B X Y Z R G B X Y Z R G B
BYTE: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72
POSITION:
STRIDE: 24
OFFSET: 0
COLOR:
STRIDE: 24
OFFSET: 12

Figure 4: Position data stored as 32-bit(4byte) floating point for x,y,z with R,G,B values.

data before rendering. To pass information to shader we first need to create **Vertex Buffer Object** and copy the data for rendering to it. To use a buffer object you have to bind to it by `glBindBuffer` and the transfer data to it by `glBufferData`, we will look at it more during the lab session. Once the data has been moved to buffers we tell about how this data is interpreted via `glVertexArrtibPointer` and is enabled by `glEnableVertexAttribArray`. The `glVertexArrtibPointer` takes in the following params-

1. Position - location of vertex attribute to configure specified by layout in shader.
2. size of attribute (vec3/vec4)
3. datatype e.g. `GL_FlOAT`
4. specification for normalization (`GL_TRUE` if data needs to be normalized else `GL_FALSE`)
5. stride (3 * sizeof(float) for float data type)
6. offset

Core OpenGL requires that we use a **Vertex Array Object** so it knows what to do with our vertex inputs. If we fail to bind a VAO, OpenGL will most likely refuse to draw anything. Thus requiring use to create by

```
int VAO; glGenVertexArrays(num vertex array object names, VAO)
```

and bind to them by calling `glBindVertexArray(VAO)`.

A vertex array object stores the following:

- Calls to `glEnableVertexAttribArray` or `glDisableVertexAttribArray`.
- Vertex attribute configurations via `glVertexAttribPointer`.
- Vertex buffer objects associated with vertex attributes by calls to `glVertexAttribPointer`.

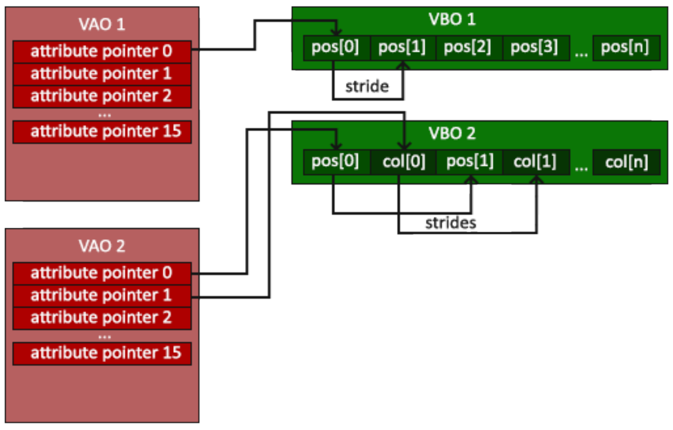

Figure 5: Binding VAOs and VBOs.

5 Deliverables

This task will be graded. Your task is to extend the provided code(main.cpp) to draw a rectangle using 2 triangles and extened it further to a cuboid.

(Hint: Cuboid will have 12 triangles)

Upload the zip file of code and output. Name the zip file as Lab01_<name_ roll no>.zip

6 References

- https://www.opengl.org/documentation/
- https://www.khronos.org/opengl/wiki/Rendering Pipeline Overview
- https://www.khronos.org/registry/OpenGL-Refpages
- https://www.glfw.org/documentation.html
- https://www.khronos.org/opengl/wiki/Framebuffer