CSE 333/533: Computer Graphics

Lab 1: Introduction to OpenGL Shaders

Instructor: Ojaswa Sharma , TAs: Vishwesh Vhavle , Aadit Kant Jha

Due date: 23:59, 18 August 2023

Introduction

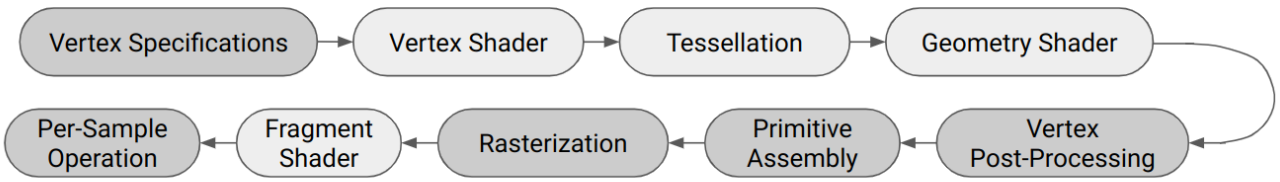
In OpenGL, shaders are a fundamental component of the graphics rendering pipeline that allow us to customize and manipulate various stages of the rendering process. Shaders are small programs written in a specialized shading language that are executed directly on the GPU and provide control over how the graphics hardware processes vertices, calculates lighting, colors, and other visual effects.

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 fragment shader needs to be set up for the task of rendering.

Goal of today's lab will be for you to learn about shaders and the graphics pipeline in general by a writing some shader scripts yourself and using it to render triangles to form a cuboid. Further, it will be demonstrated what is VBO and VAO and how to use them.

Graphics Pipeline

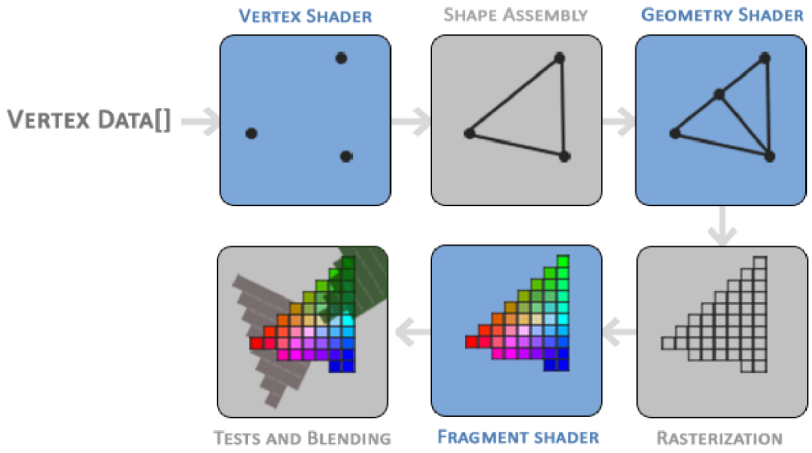
The OpenGL rendering pipeline is initiated when you perform a rendering operation. Rendering operations require the presence of a properly-defined vertex array object and a linked Program Object or Program Pipeline Object which provides the shaders for the programmable pipeline stages.



Graphics Pipeline

Once initiated, the pipeline operates in the following order:

1. **Vertex Processing:**
   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



Graphics Pipeline Visualization

2. **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.

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

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

5. **Per-Sample-Processing:** Including but not limited to
   - Scissor Test
   - Stencil Test
   - Depth Test
   - Blending
   - Logical Operation
   - Write Mask

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.

Example: To declare an input vertex attr. for position can be done by:
`in vec3 vVertex; // 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 vVertex;
uniform mat4 vModel;
uniform mat4 vView;
uniform mat4 vProjection;
uniform vec3 vColor;
out vec3 fColor;

void main() {
	gl_Position = vProjection * vView * vModel * vec4(vVertex, 1.0);
	fColor = vColor;
}
```

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.

```
in vec3 fColor;
out vec4 outColor;
void main(void) {
	outColor = vec4(fColor, 1.0);
}
```

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

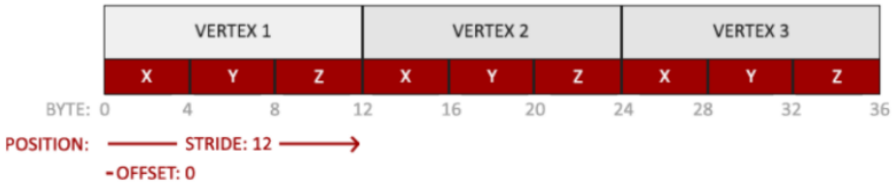
Shader Program

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.

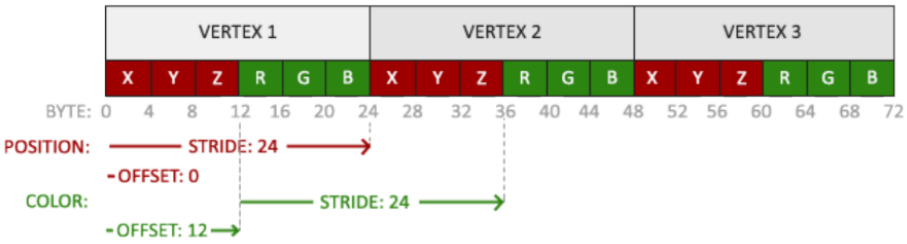
```
unsigned int vertexshader;
vertexshader = glCreateShader(GL_VERTEX_SHADER);
```

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.

Linking Vertex Attribute



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



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

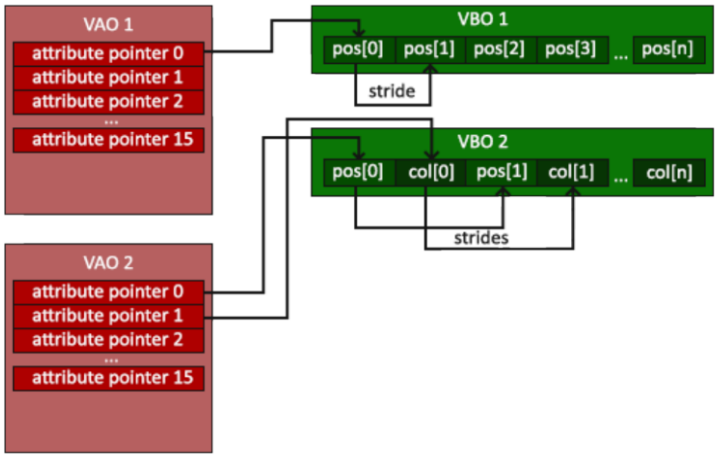
1. Position - location of vertex attribute to configure specified by layout in shader.
2. size of attribute (vec3/vec4)
3. datatype e.g. `GL_FLOAT`
4. specification for normalization (`GL_TRUE` if data needs to be normalized else `GL_FALSE`)
5. stride (3 * sizeof(float) for float data type)
6. Offset

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`.



Binding VAOs and VBOs

Exercise

You must edit the code to complete the cuboid. You must define all the vertices of the cuboid. You must define the order in which the vertices must be used to form all the triangles required to complete the cuboid. Your output must look like this:

Deliverables

Upload the zip file of **code** and **image of output**.

Name the zip file as lab01_<name>_<roll number>.zip
Example: lab01_vishwesh_2020156.zip

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

Note: Your code should be written by you and be easy to read. You are NOT permitted to use any code that is not written by you. (Any code provided by the instructor/TA can be used with proper credits within your program). Theory questions need to be answered by you and not copied from other sources. Please refer to IIIT-Delhi's Policy on Academic Integrity here.