



KOSS Selection Task

Dhanvith Nayak
23IM10010



OpenGL Shaders: General Usage and Render Pipelines



My Plan

- Build some stuff in OpenGL
- Learn the inner workings on how shaders are actually executed in the GPU
- If time permits, dig deeper into shaders and the more complicated things in OpenGL
- Make a presentation showing my process of understanding shaders and OpenGL




- First, I decided to dabble into setting up OpenGL projects and getting basic things to work in it
- So naturally the first thing to do was to create a blank window using GL to work with it further
- Seems simple enough



NOT



- Compiling and setting up GLFW and GLAD was not much trouble as I had worked before with Make and a little bit of CMake
- GLFW is an open-source library that makes creating OpenGL contexts and taking user input very simple
- GLAD is an online loader generator that will provide us with pointers to all the OpenGL functions in our graphics driver

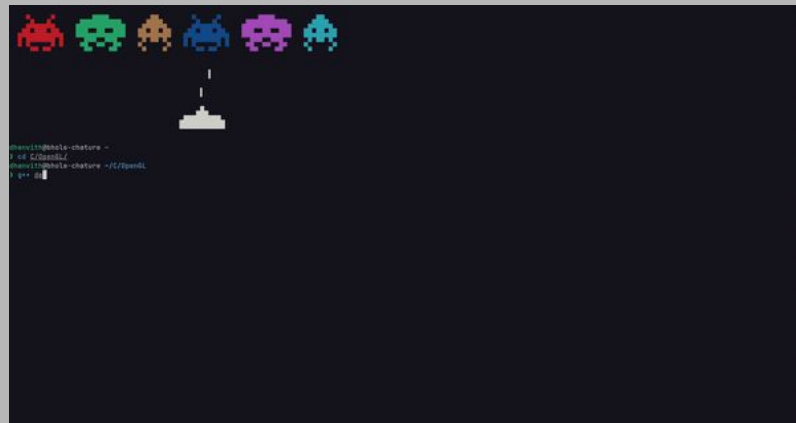
- 
- In the demo program, we first initialize GLFW and declare the version of OpenGL that we are using (in our case, v4.6)
 - We then create a window and deal with the case in which the window could not be created
 - We then retrieve all the function pointers using GLAD
 - Keep the window running until told to stop


We have a blank window!!


—



- This seemed simple enough but it took a little while to set up everything to compile without errors
- On compilation, I was bombarded with a tsunami of undefined reference errors. I guessed that this might be because I haven't linked the required dynamic libraries during compilation
- I linked the libraries during compilation but the error still persisted. I was feeling stuck for a while
- Then, I got the clue to use clang instead of gcc/g++, and finally, this did solve the issue of compilation and library linking



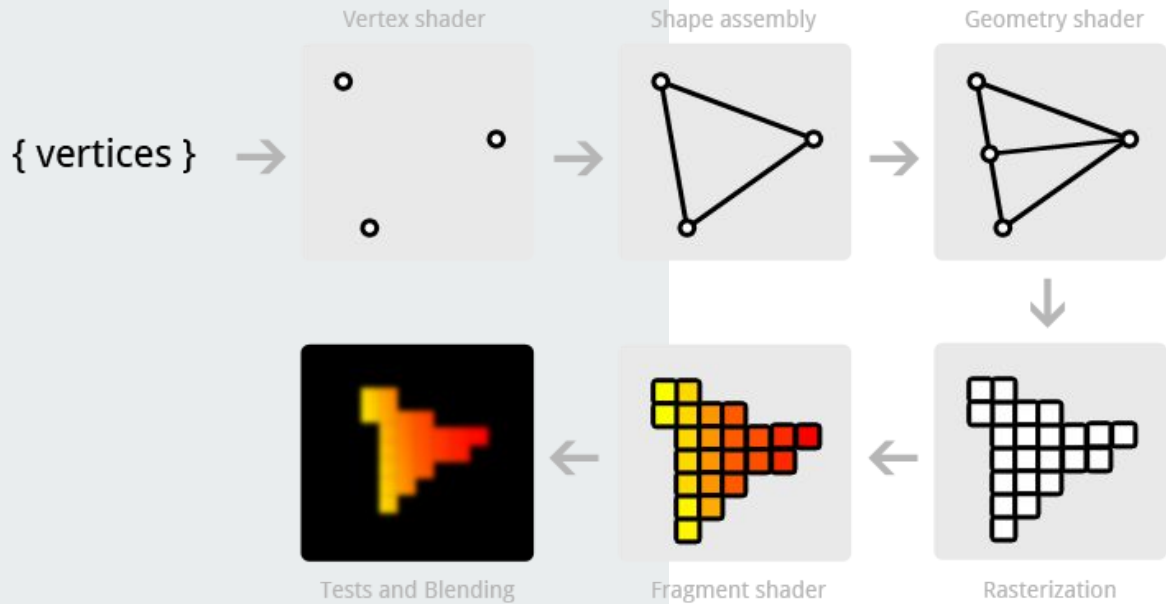
- 
- Once I had a blank window working, I went ahead to draw my first triangle in OpenGL
 - Drawing a triangle in core OpenGL took far more effort than expected since we are expected to know quite a bit about how OpenGL works behind the hood to actually use it
 - Even though this is for sure not beginner friendly, this approach from OpenGL gives a lot more control to a graphics programmer over how the rendering process is executed
 - Now let's look at some crucial concepts to draw objects in OpenGL



The OpenGL Render Pipeline

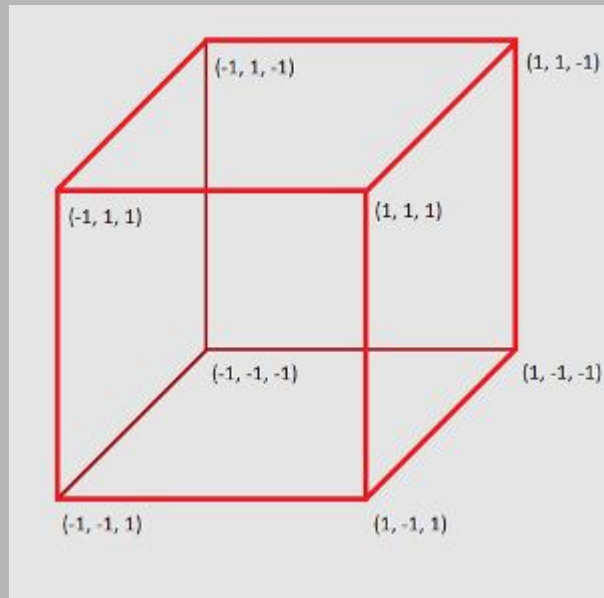
The complete process to turn a 3D object into its 2D representation

- The OpenGL graphics pipeline consists of the entire process of converting 3D coordinates into coloured pixels on a 2D screen
- Each step in this process has a specific task and most of it can be run parallelly



The Render Pipeline Stages

- We first start with a set of **vertices** that are made up of data elements called **vertex attributes**
- These attributes could be in any form, but for the sake of simplicity we can also show them directly as 3D coordinate
- Vertices define the boundaries of something called a **primitive**

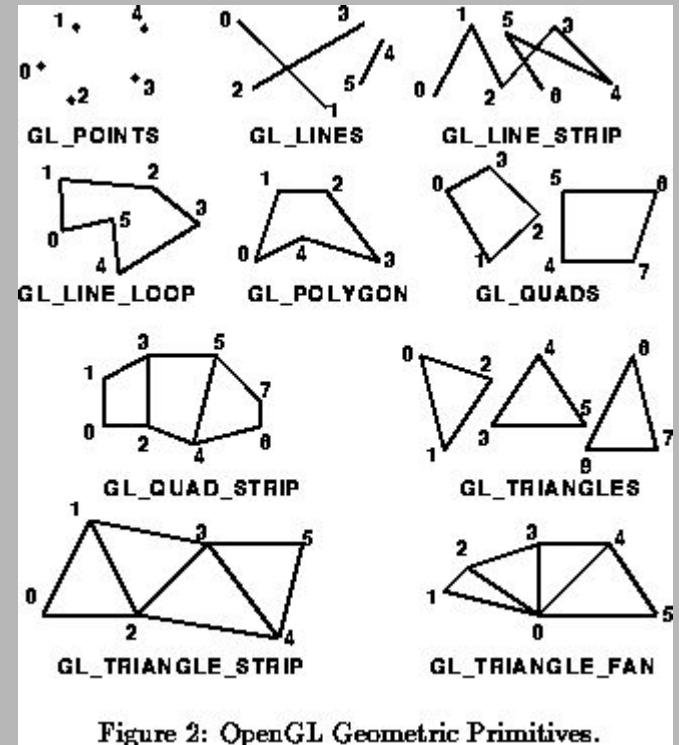




- OpenGL only draws those vertices with 3D coordinates in the range -1.0 to 1.0 in all three axes
- This range is called the **Normalized Device Coordinates** range
- Any vertex with position outside this range is truncated by OpenGL

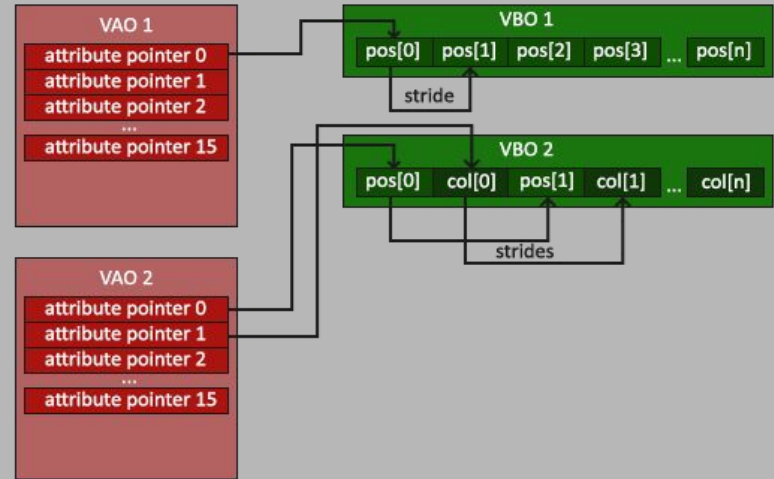
Primitives


- Primitives are basic shapes that are drawn in OpenGL
- There are different primitive shapes such as GL_POINTS, GL_TRIANGLES, etc.
- The actual process of determining how the vertices form a primitive is handled at a later stage in the render pipeline



VBOs and VAOs

- VBO (Vertex Buffer Objects) and VAO (Vertex Array Objects) are two OpenGL objects that work with vertex data in the GPU's memory
- VAOs hold information about what data each vertex holds, while VBOs store the actual vertex itself



- 
- A Vertex Buffer Object (VBO) is an OpenGL **Object** that has the capability to store a large amount of vertex data in the GPU memory
 - Repeatedly sending vertex data from the CPU to the GPU is an expensive process
 - VBOs help in this regard by allowing a large amount of data to be sent at once
 - A Vertex Array Object (VAO) is another object that stores vertex attribute pointer states
 - When we define a vertex attribute pointer, a VAO stores it
 - Whenever we want to draw that specific object, we bind to that VAO instead of repeatedly defining the attribute pointers



1. The Vertex Shader

- Each vertex first goes through the vertex shader for initial processing of its attributes
- Here we define how to interpret the attribute data into a set of 3D coordinates within the normalized device coordinates
- Vertex shaders MUST be implemented manually as there is no default implementation
- The output of the vertex shader can optionally go through further processes such as tessellation



2. Primitive Assembly

- In this process, the collection of vertices from the vertex shader and other prior processes is compiled into a set of primitives
- The type of primitive that the vertices be transformed into is defined early on by ourselves
- Here also there are optional steps such as Transform Feedback
- The render pipeline can also be stopped here and the data from the vertex shader and primitive assembly be stored in buffer objects as is done in transform feedback

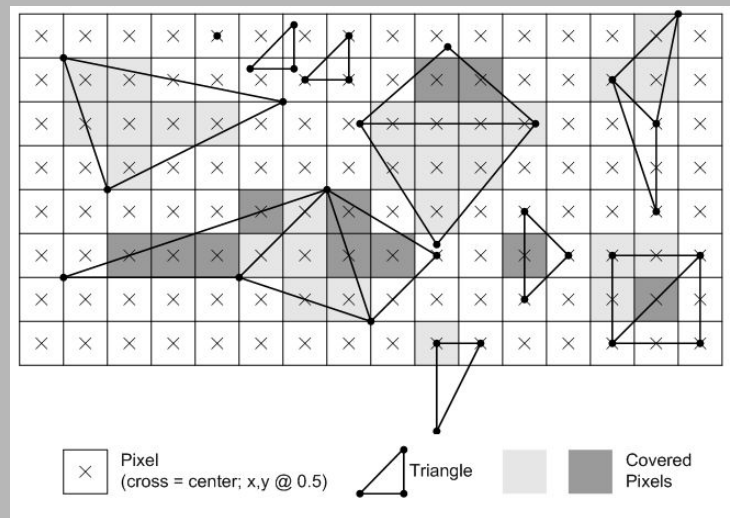


3. Geometry Shader

- This optional process takes in each primitive and gives out any number of primitives based on the input
- This is an optional step that can be implemented by the programmer in case he needs something specific
- Geometry shaders can do anything with the input, including modifying the vertex data and converting the primitive into a different type altogether

4. Rasterization

- The primitives that are obtained from the previous stages are used in this stage
- Primitives are converted into discrete elements called **fragments**
- A fragment is basically all the data to render a specific pixel
- It can contain the position where it should be rendered and other data obtained from previous processes





5. Fragment Shader

- Each fragment from the rasterization stage is processed by the fragment shader
- The data of each fragment is processed in the fragment shader to generate colour, depth and stencil values for each fragment (or pixel)
- A fragment shader is technically optional but if not defined, the colour values will be left undefined
- We define the vertex and fragment shader ourselves in nearly every case



6. Tests and Blending

- The output from the vertex shader goes through a set of tests
- If the fragment fails a test, it is usually not updated in the output buffer
- Some of the tests are the Pixel Ownership Test, Stencil Test and the Depth Test
- If all the specified tests are passed by the fragment, each fragment goes through a blending process
- This changes the colour of the fragment based on the colour of the previous version of the fragment to smooth out the colour change if any
- The fragment is finally written to the framebuffer to be displayed in the next frame

Element Buffer Objects (EBOs)

- Useful in drawing more complicated objects in which there are a lot of shared vertices
- Assume we want to draw a rectangle, instead of mentioning vertices 6 times, we can define only the unique 4 vertices
- Then, we can define the order in which the primitives should be drawn from the vertices
- EBOs store the index that gives OpenGL information on what vertices to draw

