Shaders and the Rendering Pipeline

What is the Rendering Pipeline?

- The Rendering Pipeline is a series of stages that take place in order to render an image to the screen.
- Four stages are programmable via “Shaders”.
- Shaders are pieces of code written in GLSL (OpenGL Shading Language), or HLSL (High-Level Shading Language) if you’re using Direct3D.
- GLSL is based on C.

The Rendering Pipeline Stages

- 1. Vertex Specification
- 2. Vertex Shader (programmable)
- 3. Tessellation (programmable)
- 4. Geometry Shader (programmable)
- 5. Vertex Post-Processing
- 6. Primitive Assembly
- 7. Rasterization
- 8. Fragment Shader (programmable)
- 9. Per-Sample Operations

Vertex Specification

- A vertex (plural: vertices) is a point in space, usually defined with x, y and z co-ordinates.
- A primitive is a simple shape defined using one or more vertices.
- Usually we use triangles, but we can also use points, lines and quads.
- Vertex Specification: Setting up the data of the vertices for the primitives we want to render.
- Done in the application itself.

Vertex Specification

- Uses VAOs (Vertex Array Objects) and VBOs (Vertex Buffer Objects).
- VAO defines WHAT data a vertex has (position, colour, texture, normals, etc).
- VBO defines the data itself.
- Attribute Pointers define where and how shaders can access vertex data.

Vertex Specification: Creating VAO/VBO

- 1. Generate a VAO ID.
- 2. Bind the VAO with that ID.
- 3. Generate a VBO ID.
- 4. Bind the VBO with that ID (now you’re working on the chosen VBO attached to the chosen VAO).
- 5. Attach the vertex data to that VBO.
- 6. Define the Attribute Pointer formatting
- 7. Enable the Attribute Pointer
- 8. Unbind the VAO and VBO, ready for the next object to be bound.

Vertex Specification: Initiating Draw

- 1. Activate Shader Program you want to use.
- 2. Bind VAO of object you want to draw.
- 3. Call glDrawArrays, which initiates the rest of the pipeline.

Vertex Shader

- Handles vertices individually.
- NOT optional.
- Must store something in gl_Position as it is used by later stages.
- Can specify additional outputs that can be picked up and used by user-defined shaders later in pipeline.
- Inputs consist of the vertex data itself.

Vertex Shader: Simple Example

```
1  #version 330
2
3  layout (location = 0) in vec3 pos;
4
5  void main()
6  {
7      gl_Position = vec4(pos, 1.0);
8  }
```

Tessellation

- Allows you to divide up data in to smaller primitives.
- Relatively new shader type, appeared in OpenGL 4.0.
- Can be used to add higher levels of detail dynamically.
- Won’t be used in this course.

Geometry Shader

- Vertex Shader handles vertices, Geometry Shader handles primitives (groups of vertices).
- Takes primitives then “emits” their vertices to create the given primitive, or even new primitives.
- Can alter data given to it to modify given primitives, or even create new ones.
- Can even alter the primitive type (points, lines, triangles, etc).
- Will use it once briefly in this course.

Vertex Post-Processing

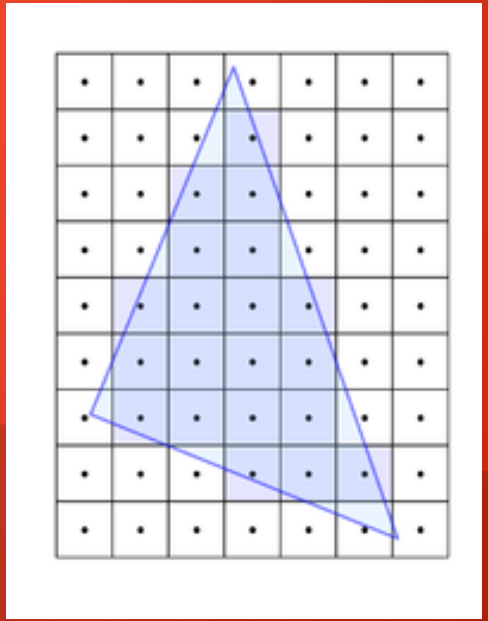
- Transform Feedback (if enabled):
  - Result of Vertex and Geometry stages saved to buffers for later use.
  - We won’t be using this though…
- Clipping:
  - Primitives that won’t be visible are removed (don’t want to draw things we can’t see!).
  - Positions converted from “clip-space” to “window space” (more on this later).

Primitive Assembly

- Vertices are converted in to a series of primitives.
- So if rendering triangles… 6 vertices would become 2 triangles (3 vertices each).
- Face culling.
- Face culling is the removal of primitives that can’t be seen, or are facing “away” from the viewer. We don’t want to draw something if we can’t see it!

Rasterization

- Converts primitives in to “Fragments”.
- Fragments are pieces of data for each pixel, obtained from the rasterization process.
- Fragment data will be interpolated based on its position relative to each vertex.



Fragment Shader

- Handles data for each fragment.
- Is optional but it’s rare to not use it. Exceptions are cases where only depth or stencil data is required (more on depth data later).
- Most important output is the colour of the pixel that the fragment covers.
- Simplest OpenGL programs usually have a Vertex Shader and a Fragment Shader.

Fragment Shader: Simple Example

```
1  #version 330
2
3  out vec4 colour;
4
5  void main()
6  {
7      colour = vec4(1.0, 0.0, 0.0, 1.0);
8  }
```

Per-Sample Operations

- Series of tests run to see if the fragment should be drawn.
- Most important test: Depth test. Determines if something is in front of the point being drawn.
- Colour Blending: Using defined operations, fragment colours are “blended” together with overlapping fragments. Usually used to handle transparent objects.
- Fragment data written to currently bound Framebuffer (usually the default buffer, more on this later).
- Lastly, in the application code the user usually defines a buffer swap here, putting the newly updated Framebuffer to the front.
- The pipeline is complete!

On the Origin of Shaders...

- Shaders Programs are a group of shaders (Vertex, Tessellation, Geometry, Fragment…) associated with one another.
- They are created in OpenGL via a series of functions.

Creating a Shader Program

- 1. Create empty program.
- 2. Create empty shaders.
- 3. Attach shader source code to shaders.
- 4. Compile shaders.
- 5. Attach shaders to program.
- 6. Link program (creates executables from shaders and links them together).
- 7. Validate program (optional but highly advised because debugging shaders is a pain).

Using a Shader Program

- When you create a shader, an ID is given (like with VAOs and VBOs).
- Simply call glUseProgram(shaderID)
- All draw calls from then on will use that shader, glUseProgram is used on a new shaderID, or on ‘0’ (meaning ‘no shader’).

Summary

- Rendering Pipeline consists of several stages.
- Four stages are programmable via shaders (Vertex, Tessellation, Geometry, Fragment).
- Vertex Shader is mandatory.
- Vertices: User-defined points in space.
- Primitives: Groups of vertices that make a simple shape (usually a triangle).
- Fragments: Per-pixel data created from primitives.
- Vertex Array Object (VAO): WHAT data a vertex has.
- Vertex Buffer Object (VBO): The vertex data itself.
- Shader programs are created with at least a Vertex Shader and then activated before use.

See you next video!