**OPENGL**

SHADERS & THE RENDERING PIPELINE

* The rendering pipeline is a series of stages that take place in order to render an image to the screen.
* Four of these stages are programmable via “shaders”:
  + Vertex
  + Fragment
  + Geometry
  + Tessellation
* Shaders are pieces of code written in GLSL (Open GL Shading Language) -> based on C.

Rendering pipeline

1. Vertex specification
2. Vertex shader
3. Tessellation
4. Geometry shader
5. Vertex post-processing
6. Primitive assembly
7. Rasterization
8. Fragment shader
9. Per-sample operations

Vertex specification

*Vertex –* A point in space, usually defined with *x, y & z* cords.

*Primitive –* A simple shape defined using one or more vertices (triangles, points, lines, etc.)

* Sets up the data of the vertices for the primitives we want to render
* Uses VAOs (Vertex Array Objects) and VBOs (Vertex Buffer Objects)
* VAO defines *what* data a vertex has (position, colour, texture, etc.)
* VBO defines the data itself
* Attribute pointers define where and how shaders can access vertex data
* VAO & VBO are stored in the GPUs RAM

How:

1. Generate a VAO ID
2. Bind the VAO with that ID
3. Generate a VBO ID
4. Bind the VBO with that ID
5. Attach the vertex data to that VBO
6. Define attribute pointer formatting
7. Enable attribute pointer so shader can access it
8. Unbind VAO & VBO ready for the next object to be bound

Initiating draw:

1. Activate shader program 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
* Must store something in *gl\_Position* at is it used by later stages
* Inputs consist of the vertex data itself

Example:

#version 330 // specify GLSL version to use

Layout (location = 0) in vec3 pos; // in = input, vec3 = type, pos = var name

Void main() {

gl\_Position = vec4(pos, 1.0); // gl\_Position requires vec4

// could also write:

// gl\_Position = vec4(pos.x, pos.y, pos.z, 1.0);

}

Tessellation

* Allows you to divide up data into smaller primitives

Geometry shader

* Handles primitives (groups of vertices)
* Can alter data given to it to modify given primitives, or even create new ones

Vertex post processing

* Transform feedback (if enabled):
  + Result of vertex & geometry saved to buffers for later use
* Clipping:
  + Primitives that won’t be visible are removed (you don’t want to draw things that can’t be seen)

Primitive assembly

* Vertices are converted in to a series of primitives
  + If rendering triangles… 6 vertices would become 2 triangles
* Face culling
  + Removal of faces of primitives that can’t be seen; e.g. if facing a cube square on, the bottom, sides, top and back faces won’t be drawn

Rasterization

* Converts primitives in to *fragments*
  + Fragments are pieces of *data* for each pixel

Fragment shader

* Handles data for each fragment (think of a fragment like a pixel)
* Most important output is the colour of the pixel that the fragment covers
* Colour, texture, shadows, lighting…

Example:

#version 330

Out vec4 colour;

Void main() {

colour = vec4(1.0, 0.0, 0.0, 1.0); // rgba()

}

Per-sample operations

* Series of tests run to see if the fragment should be drawn
  + Most important part is the depth test:
    - Determines if something is in front of the point been drawn

Lastly – swap the back buffer with the front buffer.

The pipeline is now complete!

CREATING A SHADER PROGRAM

1. Create an empty program
2. Create empty shaders
3. Attach shader source code to shaders
4. Compile shaders
5. Attach shaders to program
6. Link program
7. Validate program

When you create a shader, an ID is given. Simple call *glUseProgram(shaderID)*. All draw calls from now on will use that shader.

CREATE A WINDOW

1.0f

-1.0f

1.0f

- 1.0f

1. Check if *glfw* initialized
   1. *If (!glfwInit()) { //error }*
2. Create a window
   1. *GLFWwindow\* window = glfwCreateWindow(1024, 768, “title”, NULL, NULL);*
   2. *if (!window) { // error }*
   3. *glfwMakeContextCurrent(window);*
3. Check if glew initialized
   1. *If (glewInit() != GL\_TRUE) { // error }*
4. Basic game loop
   1. *While (!glfwWindowShouldClose(window)) {*

*glfwPollEvents();*

*glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // rgba colour of window*

*glClear(GL\_COLOR\_BUFFER\_BIT);*

*glSwapBuffers(window);*

*}*

1. Call *glfwTerminate()* before program exits (including at error points)

DRAW TRIANGLE / CREATE SHADERS