**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() != GLEW\_OK) { // 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

*For simplicity, we will write everything in one file, and use global variable for easy readability. Creating a window will also not be included but referenced where needed.*

// globals

GLuint VAO, VBO, shader;

// shaders

Static const char\* vertexShader =

“#version 330\n”

“layout (location = 0) in vec3 pos;\n”

“void main() {\n”

“gl\_Position = vec4(pos, 1.0);\n”

“}”;

Static const char\* fragmentShader =

“#version 330\n”

“out vec4 color;\n”

“void main() {\n”

“color = vec4(1.0f, 0.0f, 1.0f, 0.8f);\n” // rgba()

“}”

void createTriangle() {

float vertices[9] = {

-1.0f, -1.0f, 0.0f, // look at window diagram above

1.0f, -1.0f, 0.0f,

0.0f, 1.0f, 0.0f

};

// remember, VAO and VBO are global vars

// VAO

glGenVertexArrays(1, &VAO); // arg1: no. of arrays, arg2: id storage

glBindVertexArray(VAO); // pass the id created from function above

// VBO

glGenBuffers(1, &VBO);

glBindBuffer(GL\_ARRAY\_BUFFER, VBO);

glBufferData(

GL\_ARRAY\_BUFFER, 9 \* sizeof(GL\_FLOAT), vertices, GL\_STATIC\_DRAW

);

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 0, 0); // define array

glEnableVertexAttribArray(0);

// unbind VAO and VBO

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

glBindVertexArray(0);

}

// check errors along the way using glGetShaderiv and other functions

Void addShader(GLint program, const char\* shaderCode, GLenum type) {

GLuint theShader = glCreateShader(type);

Const GLchar\* theCode[1];

theCode[0] = shaderCode;

GLint codelength[1];

codelength[0] = strlen(shaderCode);

glShaderSource(theShader, 1, theCode, codeLength);

glCompileShader(theShader);

glAttachShader(program, theShader);

}

Void compileShaders() {

shader = glCreateProgram(); // global var

if (!shader) { // error }

addShader(shader, vertexShader, GL\_VERTEX\_SHADER);

addShader(shader, fragmentShader, GL\_FRAGMENT\_SHADER);

// used for errors

GLint result = 0;

GLchar elog[1024] = {0};

// create GPU executables

glLinkProgram(shader);

glGetProgramiv(shader, GL\_LINK\_STATUS, &result); // checking for errors

if (!result) {

glGetProgramInfoLog(shader, sizeof(elog), NULL, elog);

std::cout << “Error linking: “ << elog << std::endl;

return;

}

glValidateProgram(shader);

glGetProgramiv(shader, GL\_VALIDATE\_STATUS, &result);

if (!result) {

glGetProgramInfoLog(shader, sizeof(elog), NULL, elog);

std::cout << “Error validating: “ << elog << std::endl;

return;

}

}

// in our main function, after creating the window but above game loop…

createTriangle();

compileShaders();

// game loop now becomes…

While (!glfwWindowShouldClose(window)) {

glfwPollEvents();

glClearColor(1.0f, 1.0f, 0.0f, 1.0f);

glClear(GL\_COLOR\_BUFFER\_BIT);

// draw!

glUseProgram(shader);

glBindVertexArray(VAO);

glDrawArrays(GL\_TRIANGLES, 0, 3);

// clear shader

glBindVertexArray(0);

glUseProgram(0);

glfwSwapBuffers(window);

}

UNIFORM VARIABLES

* Type of variable in a shader
* Uniforms are values global to the shader that aren’t associated with a particular vertex

#version 330

uniform mat4 model; // constant

void main() {

gl\_Position = model \* vec4(pos, 1.0);

}

* Each uniform has a location ID in the shader
* Need to find the location so we can bind a value to it:
  + *Int location = glGetUniformLocation(shaderID, “uniformVarName”);*
* Now, we can bind a value to that location:
  + *glUniform(location, 3.5f);*

TRANSFORMING

You can have multiple transformations to a model, e.g. a translate followed by a rotate to make an object move left to right whilst rotating.

*Follows on from examples above*

Translation

// normal includes, plus:

#include <glm/glm.hpp> // opengl maths library

#include <glm/gtc/matrix\_transform.hpp>

#include <glm/gtc/type\_ptr.hpp>

// more global variables

GLuint uniformModel;

bool direction = true;

float triOffset = 0.0f;

float triMaxOffset = 0.7f;

float triIncrement = 0.0005f;

// alter vertex shader to be:

Static const char\* vertexShader =

“#version 330\n”

“layout (location = 0) in vec3 pos;\n”

“uniform mat4 model;”

“void main() {\n”

“gl\_Position = model \* vec4(0.4 \* pos.x, 0.4 \* pos.y, pos.z, 1.0);\n”

“}”;

// add to the last line of the compileShader function

uniformModel = glGetUniformLocation(shader, “model”); // global var

// in while loop, before clearing window

If (!direction) {

triOffset += triIncrement;

} else {

triOffset -= triIncrement;

}

If (abs(triOffset) >= triMaxOffset) {

direction = !direction;

}

// normal clear code

// draw

glm::mat4 model(1.0f);

model = glm::translate(model, glm::vec3(triOffset, 0.0f, 0.0f));

glUniformMatrix4fv(uniformModel, 1, GL\_FALSE, glm::value\_ptr(model));

// rest of code…

* This will result in a triangle, 0.f times in width and height the size of the window, moving from left to right…

Rotation

glm::rotate(model, angle, glm::vec3(x, y, z)); // angle (radians), vec3 (point to rotate around)

// Global var

const float toRadians = 3.14/180.0f; // multiple to convert degrees to radians

float curAngle = 0.0f;

// in while loop

curAngle += 0.0f;

if (curAngle >= 360) {

curAngle -= 360; // prevent number going out of bounds

}

// after glm::translate

model = glm::rotate(model, curAngle \* toRadians, glm::vec3(0.0f, 0.0f, 1.0f)); // z-axis

Scaling

glm::scale(model, glm::vec3(x, y, z));

Scaling should be the last thing to transform else it could cause some issues with the other transformations.

model = glm::scale(model, glm::vec3(0.4f, 0.4f, 1.0f)); // where 1.0f is unchanged

Dynamic values can be passed inside of the vector to give a scaling up/down effect.

// global vars

bool sizingUp = true;

float curSize = 0.4f;

float maxSize = 0.8f;

float minSize = 0.1f;

// in game loop

if (sizingUp) {

curSize += 0.0001f;

} else {

curSize -= 0.0001f;

}

if (curSize >= maxSize || curSize <= minSize) {

sizeDirection = !sizeDirection;

}

// after all transforms

model = glm::scale(model, glm::vec3(curSize, curSize, 1.0f)); // 2d so don’t scale in z

INTERPOLATION

Interpolation is a type of estimation, a method of constructing new data points within the range of a discrete set of known data points. You use interpolation when you have a limited amount of data trying to represent unlimited amount of data, e.g. a line where you have 2 known points (the beginning and end).

PROJECTIONS

In order to understand projections, you need to understand the different coordinate systems at play:

* Local space – the raw position of each vertex drawn relative to the origin. Multiply by model matrix to get…
* World space – position of vertex in the world itself if the camera is assumed to be positioned at the origin. Multiple by view matrix to get…
* View space – position of vertex in the world relative to the camera position and orientation. Multiply by projection matrix to get…
* Clip space – position of vertex in the world, relative to the camera position and orientation, as viewed in the area not to be “clipped” from the final output.
* Screen space – After clipping takes place, the final image is created and placed on the coordinate system of the window itself.

To create clip space, we define an area of what is not to be clipped with a projection matrix. The two commonly used types of projections:

1. Orthographic (most commonly used in 2d apps)
2. Perspective (most commonly used in 3d apps – gives an illusion of depth)

Add interpolation & projection

// Add global variable

GLuint IBO uniformProjection; // (IBO = index buffer object)

// change shaders to:

static const char\* vertexShader =

“#version 330\n”

“out vec4 vCol;\n”

“layout(location = 0) in vec3 pos;\n”

“uniform mat4 model;\n”

“uniform mat4 projection;\n”

“void main() {\n”

“gl\_Position = projection \* model \* vec4(pos, 1.0);\n”

“vCol = vec4(clamp(pos, 0.0f, 1.0f), 1.0f);\n”

“}”;

static const char\* fragmentShader =

“#version 330\n”

“in vec4 vCol;\n”

“out vec4 color;\n”

“void main() {\n“

“color = vcol;\n”

“}”

// inside createTriangle function

void createTriangle() {

// at the top

unsigned int indices[] = {

0, 3, 1,

1, 3, 2,

2, 3, 0,

0, 1, 2

};

// change vertices to:

float vertices[] = {

-1.0f, -1.0f, 0.0f,

0.0f, -1.0f, 1.0f,

1.0f, -1.0f, 0.0f,

0.0f, 1.0f, 0.0f

};

// after VAO

glGenBuffers(1, &IBO);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, IBO);

glBufferData(

GL\_ELEMENT\_ARRAY\_BUFFER,

sizeof(indices),

indices,

GL\_STATIC\_DRAW

);

}

// add to the last line of compileShaders()

uniformProjection = glGetUniformLocation(shader, “projection”);

// after calling compileShaders() inside of main

glm::mat4 projection = glm::perspective(45.0f, (GLfloat)width/(GLfloat)height, 0.1f, 100.0f);

// in while loop

// change glClear() to:

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

// after glUniformMatrix4fv call for uniformModel

glUniformMatrix4v(uniformProjection, 1, GL\_FALSE, glm::value\_ptr(projection));

glBindVertexArray(VAO);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, IBO);

glDrawElements(GL\_TRIANGLES, 12, GL\_UNSIGNED\_INT, 0);

// clear shader

glBindVertexArray(0);

glBindBuffer(GL\_ELEMENT\_ARRAY\_BUFFER, 0);

glUseProgram(0);

HOW TO SPLIT CODE UP

|  |  |  |
| --- | --- | --- |
| **Window** | **Mesh** | **Shader** |
| Window(GLint w, GLint h);  int initialize()  float getBufferWidth();  float getBufferHeight();  bool shouldClose();  void swapBuffers();  GLFWwindow\* win;  GLint w, h;  GLint bufWidth, bufHeight; | Mesh();  void createMesh(  GLfloat\* vertices,  unsigned int\* indices,  unsigned int numOfVert,  unsigned int numOfIndices  );  void renderMesh()  void clearMesh();  GLuint VAO, VBO, IBO;  GLsizei indexCount; | Shader();  void createFromFiles(  const char\* vertexPath,  const char\* fragPath  );  std::string readFile(  const char\* path  );  GLuint getProjectionLoc();  GLuint getModelLoc();  void useShader();  void clearShader();  GLuint  shaderID,  uniformProjection,  uniformModel;  void compileShader(  const char\* vertexCode,  const char\* fragCode  );  void addShader(  GLint program,  const char\* shaderCode,  GLenum type  ); |

**TEST**

PART 1

1. Describe each of the 9 steps of the rendering pipeline
2. Create a window
3. What is the structure of a simple vertex shader?
4. What is the structure of a simple fragment shader?

PART 2

1. Explain the methods:
   1. glGenBuffers
   2. glBindBuffer
   3. glGenVertexArrays
   4. glBindVertexArray
   5. glBindBufferData
   6. glVertexAttribPointer
   7. glEnableVertexAttribArray
2. Explain the methods:
   1. glCreateShader
   2. glShaderSource
   3. glCompileShader
   4. glAttachShader
3. Explain the methods:
   1. glCreateProgram
   2. glLinkProgram
   3. glGetProgramiv
   4. glGetProgramInfoLog
   5. glValidateProgram
   6. glGetUniformLocation
4. Explain the methods:
   1. glViewport
   2. glClearColor
   3. glClear
   4. glUniformMatrix4fv
   5. glUseProgram

PART 3

1. Draw a static 2D triangle on screen
2. Add some transforms (translate, rotate and scale) to the triangle above – with different combinations
3. Draw a 3D triangle on screen with transformations