Graphics pipeline

* Get vertices
* Use vertex shader on vertices
* Create shape with primitive hint (primitive assembly)
* Use geometry shader on shape
* Rasterize final shape
* Do clipping
* Use fragment shader on shape
* Do alpha test and blending

I understand the reason behind this order, but I don't understand why alpha tests and blending is done after fragment shader. Is this to make it easier to calculate colors when shapes have an alpha value?

OpenGL works with NDC ranging from -1 to 1 on all axises.

Fragments are data that OpenGL needs to calculate a pixels color.

Primitive assembly seems to interpret the vertices counterclockwise.

**Vertex input**

Create memory on GPU for vertex data. Tell gpu how to interpret vertex data. Specify how to send vertex data to GPU. Tell how vertex shader should interpret all the vertices.

The memory is managed with VBO, vertex buffer objects. This allows us to send large batches of vertices to the GPU allowing shaders to access the vertices fast.

VBO has type GL\_ARRAY\_BUFFER.

When copying user data to the VBO (do this with glBufferData), we have 3 different ways of storing it:

* GL\_STATIC\_DRAW: the data will most likely not change at all or very rarely.
* GL\_DYNAMIC\_DRAW: the data is likely to change a lot.
* GL\_STREAM\_DRAW: the data will change every time it is drawn.

**Vertex shader creation**

Shaders are written in GLSL

#version 330 core  
layout (location = 0) in vec3 aPos;  
  
void main()  
{  
 gl\_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);  
}

gl\_Position is an keyword that we can/must give a value.

We first take the version and specify that we are using core.

Layout location will be explained later.

We specify vertex attribute aPos and forward it. Normally positions of the vertices are not in NDC.

We create a shader with glCreateShader, it returns ID.

Shader compiling

Shaders are compiled during runtime (driver dependant?) So we need to write code to compile a shader.

We call glShaderSource to compile which for some reason the last parameter stays null. Second parameter is amount of source codes were passing (can be more than one?).

To check for errors during compilation, run glGetShaderiv which returns state of last compilation. If there was an error, run glGetShaderInfoLog to get error message.

**Fragment shader creation**

#version 330 core  
out vec4 FragColor;  
  
void main()  
{  
 FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);  
}

Fragment shaders only require one output value that we can specify with the out keyword (why not an keyword like gl\_Position here?).

**Shader program**

The shader program links outputs of shaders with other inputs of shaders. This is where linker errors can happen of in/out mismatches.

Call glCreateProgram to get programID.

Call glAttachShader to add shaders to the program. I think order is managed under the hood.

Call glLinkProgram to link the shaders of the program, after all shaders are attached.

glGetProgramiv and log is used to check link status and errors.

To use this new program, call glUseProgram with the ID.

Now delete the shaders with glDeleteShader because they are in the program now.

**Linking vertex buffer attributes**

OpenGL does not know how to interpret the VBO. So we specify it.

Vertex buffer data is stored as follows:

* The position data is stored as 32-bit (4 byte) floating point values.
* Each position is composed of 3 of those values.
* There is no space (or other values) between each set of 3 values. The values are tightly packed in the array.
* The first value in the data is at the beginning of the buffer.

We use glVertexAtrribPointer to specify how each vertex attribute should be interpreted.

glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);  
glEnableVertexAttribArray(0)

Parameter list:

* Because position is in layout 0, we use 0 here (not sure what and why this is, so will research it more in the future).
* The amount of components in vertex. 3 because it only has position (the VBO)
* Type of the components.
* If it has to be normalized (for colors I guess, we use coordinates).
* The size of the stride, can leave 0 to let OpenGL decide (I think), because VBD is tightly packed with no random data, we can also define it.
* Offset of where to start in buffer. You may be able to put more meshes in one VBO.

This is applies on the last glBindBuffer.

Now we can call glEnableVertexAttribArray with the location. I don't understand this and will research it more.

**Drawing**

This is how drawing looks:

// 0. copy our vertices array in a buffer for OpenGL to use  
glBindBuffer(GL\_ARRAY\_BUFFER, VBO);  
glBufferData(GL\_ARRAY\_BUFFER, sizeof(vertices), vertices, GL\_STATIC\_DRAW);  
// 1. then set the vertex attributes pointers  
glVertexAttribPointer(0, 3, GL\_FLOAT, GL\_FALSE, 3 \* sizeof(float), (void\*)0);  
glEnableVertexAttribArray(0);   
// 2. use our shader program when we want to render an object  
glUseProgram(shaderProgram);  
// 3. now draw the object   
someOpenGLFunctionThatDrawsOurTriangle();

**Vertex Array Object**

A VAO binds all vertex attribute calls (I don't know what vertex attributes are, will research).

By binding the VAO ,we can easily draw another object by just binding another VAO.

Because the vertexAttribPointer points to the latest bount VBO, it always points to the right VBO (wow that’s pretty clever).

Thus changing our draw call to just binding the program and VAO and then someDrawFunction.

**The OpenGL draw**

Bind the VAO and programs.

glDrawArrays take 3 parameters:

* Primitive type
* Starting index of vertex array
* Amount of vertices

**Element Buffer Objects**

To save on duplicate vertices, EBOs contain indexes to vertices.

The bind is the same as a VBO.

The VAO stores all the glBindBuffer calls. So the VBO and EBO will be bound when binding the VAO.