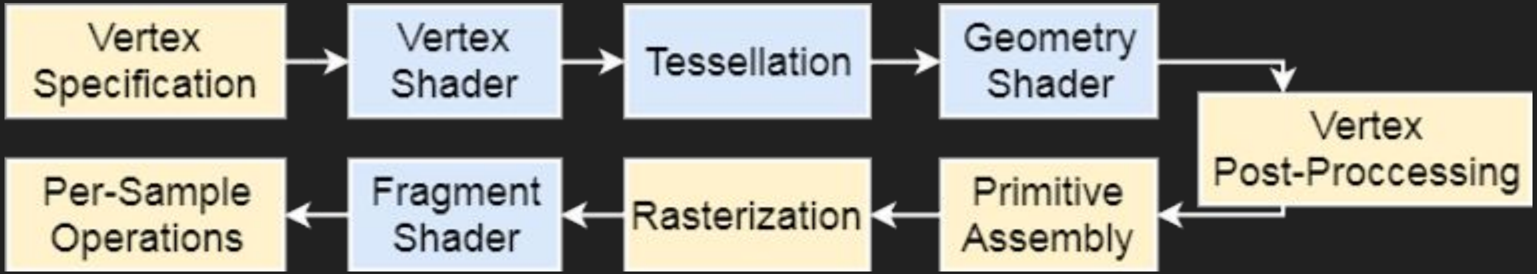
OpenGL shader & GLSL

2020 Computer Graphics

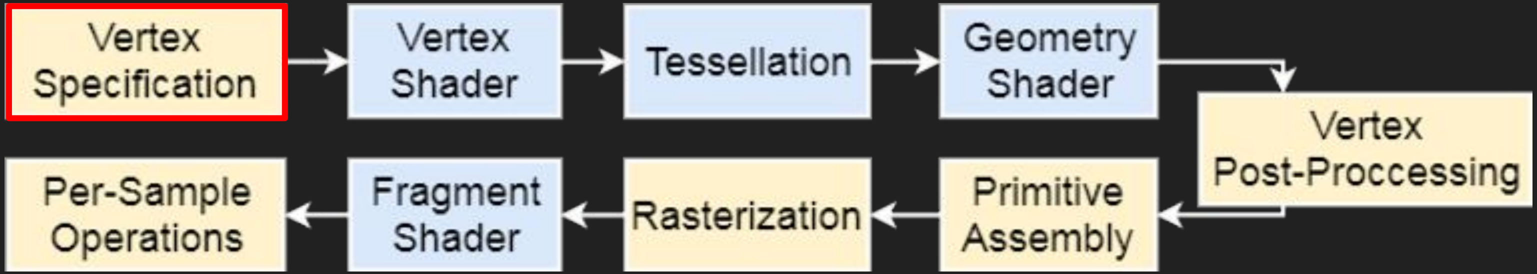
OpenGL pipeline

- Diagram of the Rendering Pipeline.
- The blue boxes are programmable shader stages.



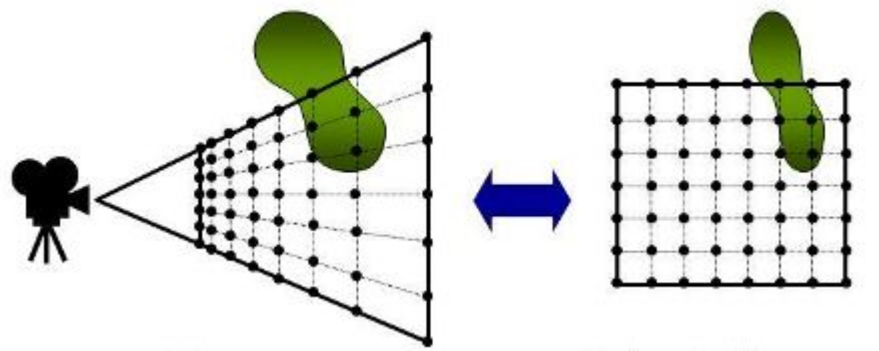
Vertex Specification

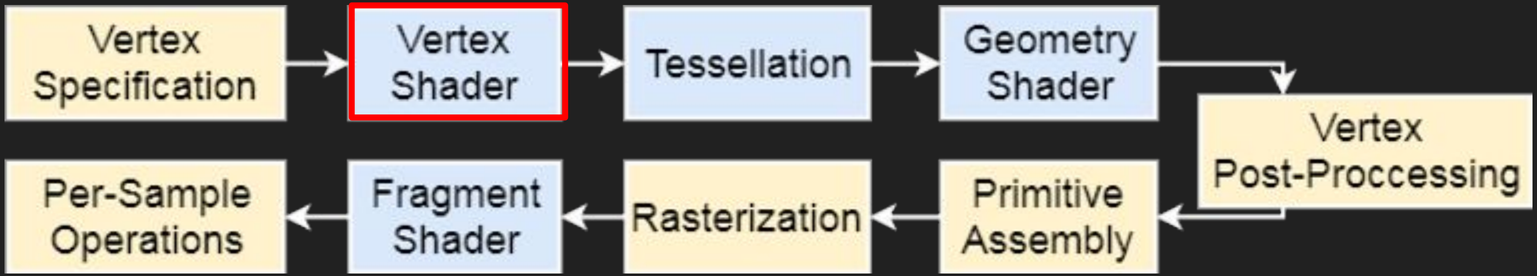
- Set up an ordered list of vertices and send to the pipeline.
- The vertices define the boundaries of a ***primitive***
- Vertex Array Objects
  - The data of each vertex
- Vertex Buffer Objects
  - The actual vertex data itself



Vertex Shader

- A vertex → another new vertex
- Transform to post-projection space
- Per-vertex lighting
- Not optional





Vertex Specification

Vertex Shader

Tessellation

Geometry Shader

Vertex Post-Proccessing

Primitive Assembly

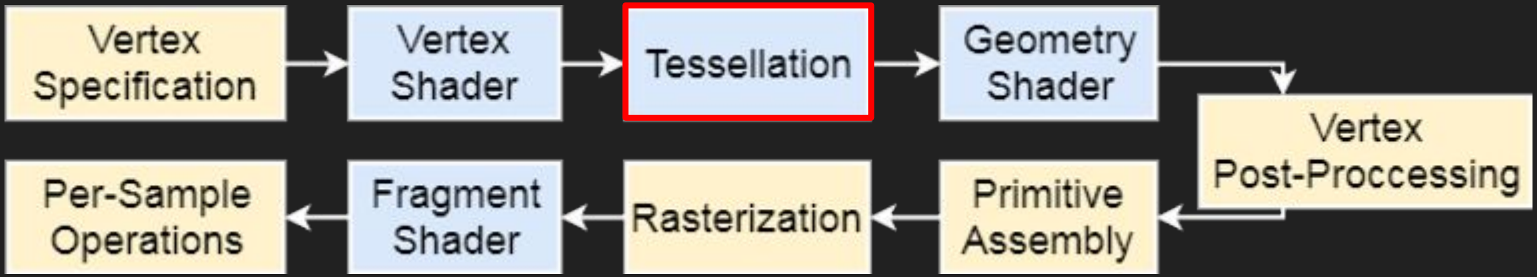
Rasterization

Fragment Shader

Per-Sample Operations

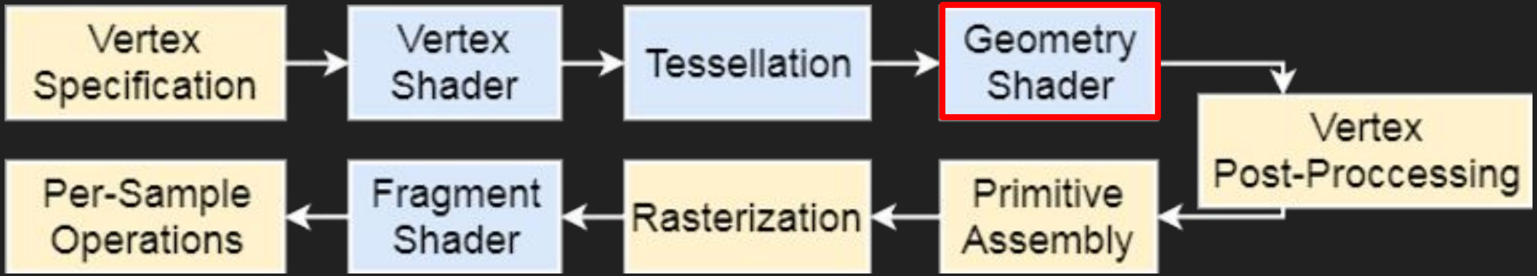
Tessellation

- “Patched” input data
- Divided into smaller primitives
    - With some new vertices
- Optional



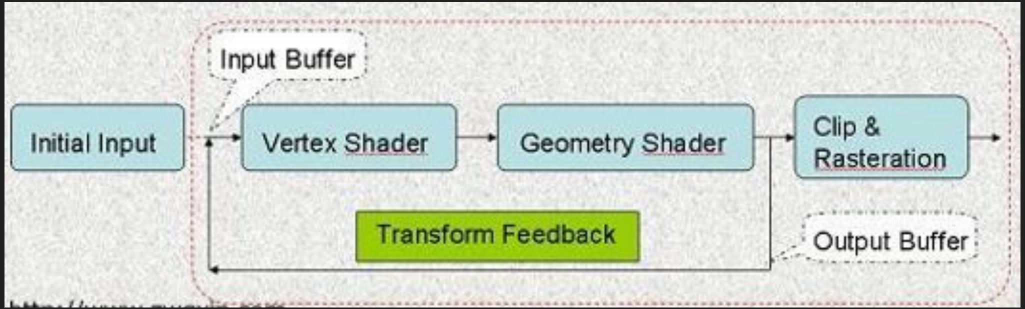
Geometry Shader

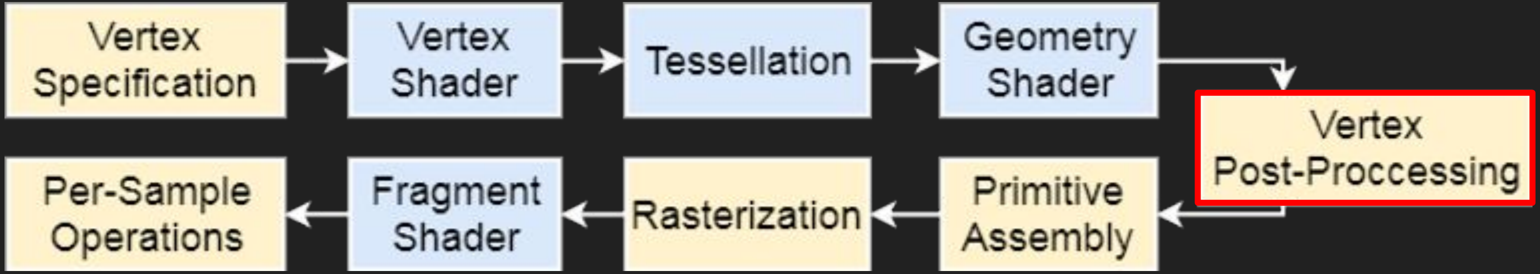
- An input primitive → zero or more output primitives
- Type of primitives
  - Subset of primitives in Primitive Assembly process
- Optional



Vertex Post-Processing

- **Transform Feedback**
  - Hold the data from previous stage for use later

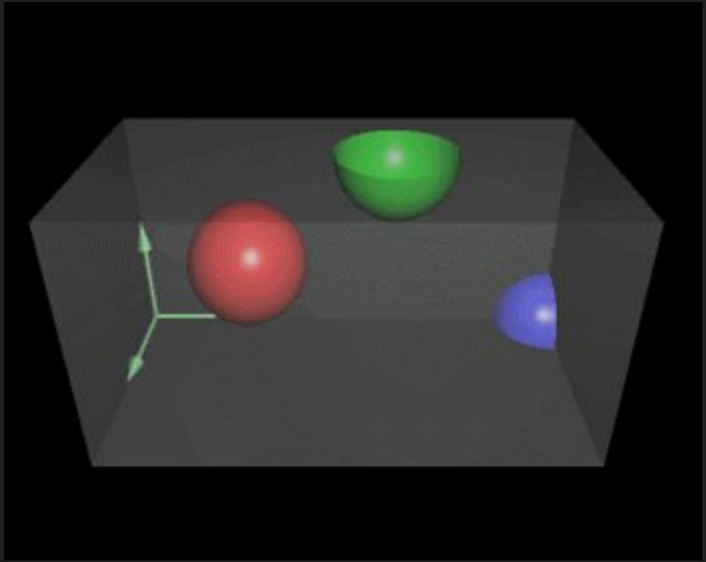


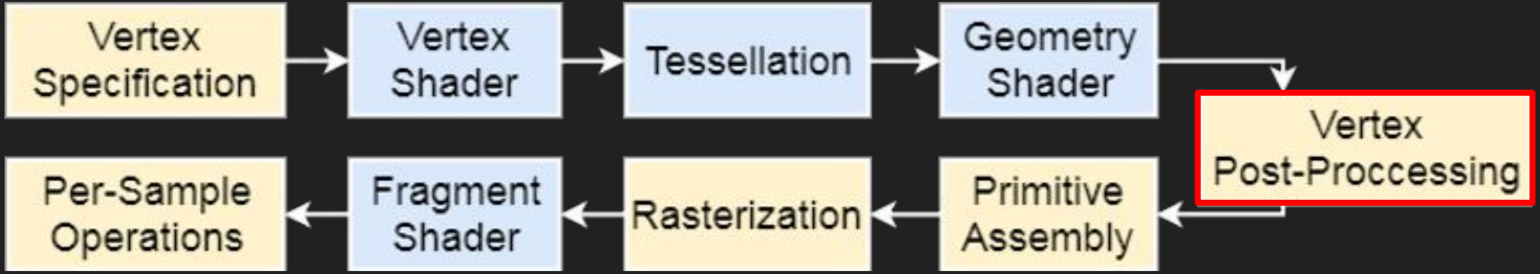


Vertex Post-Processing

- **Clipping**
  - Boundary of the viewing volume
  - User-defined clipping operations

    (the last Vertex Processing shader stage)





Vertex Specification

Vertex Shader

Tessellation

Geometry Shader

Vertex Post-Proccessing

Primitive Assembly

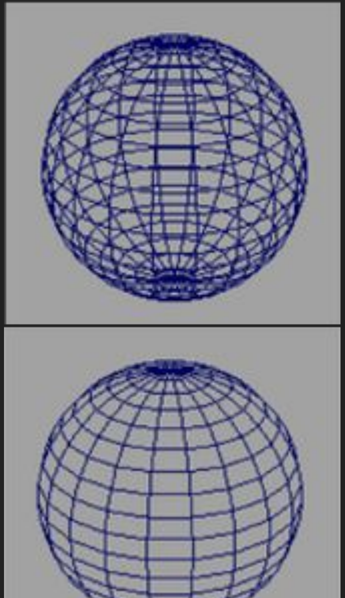
Rasterization

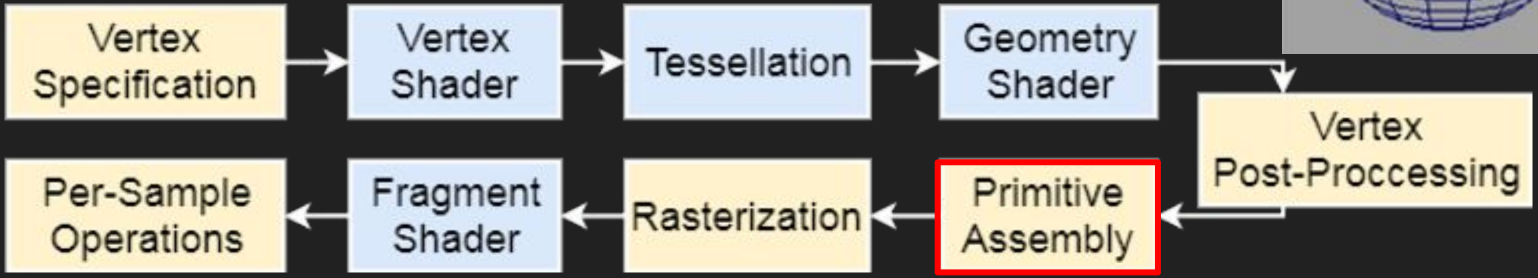
Fragment Shader

Per-Sample Operations

Primitive Assembly

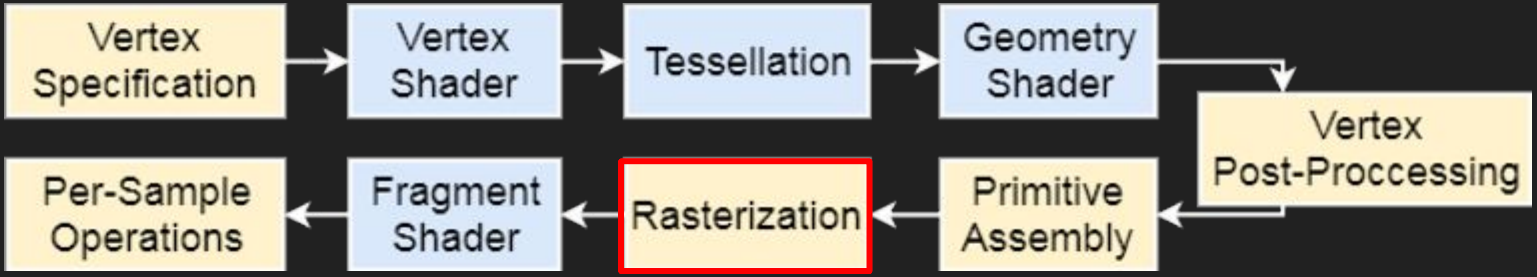
- Vertices → Primitives
- Output type: Simple primitives (lines, points, or triangles)
- Transform Feedback operations
- **(Back) Face Culling**
  - to avoid rendering triangles facing away from the viewer





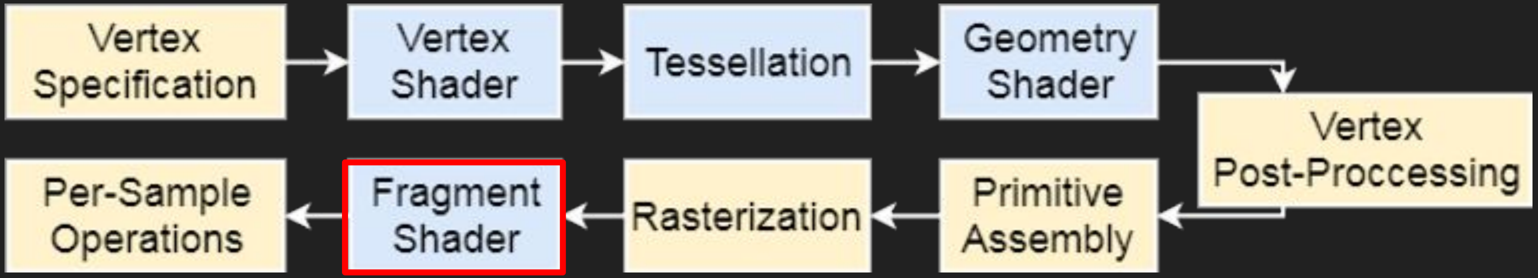
Rasterization

- Primitives → Fragments



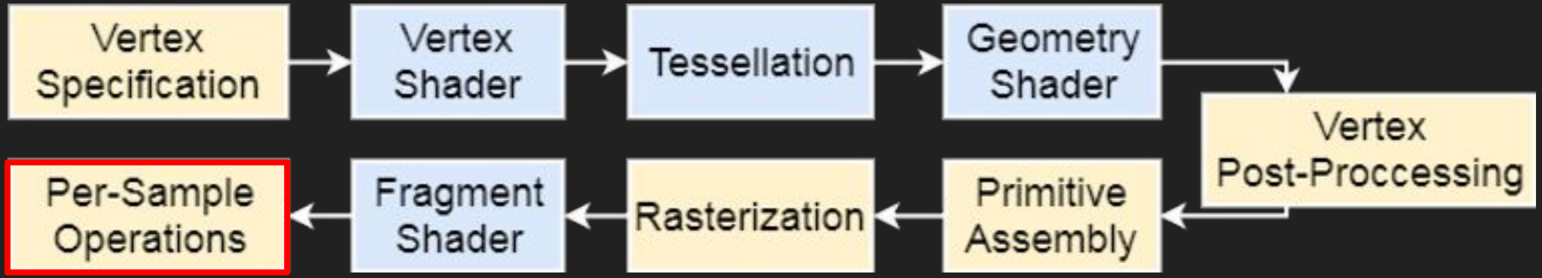
Fragment shader

- Output: color, depth value
- Optional – If there's no Fragment Shader?
  - The depth get their usual values
  - Colors are undefined!
  - Useful when focus on depth information



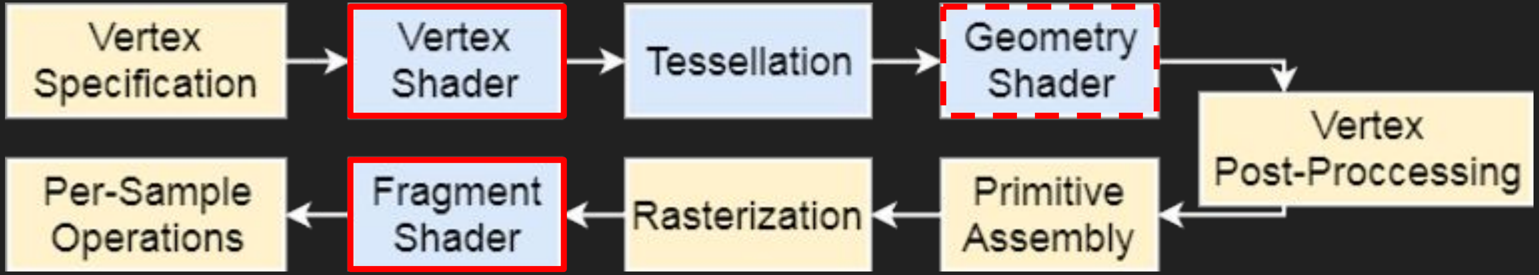
Per-sample operations

- **Depth Test**
  - Compare the depth value with the value in depth buffer



How to use shader

Rendering Pipeline



Outline

- Shader Programming in OpenGL
- Data Connection
  - VBO
  - VAO
  - Uniform
  - Texture
- GLSL Syntax
- Vertex Shader
- Fragment Shader

Shader Programming in OpenGL (shader.hpp)

- char *ReadShader(const char * shaderpath)
  - return a pointer to the shader source
- bool CreateShader(unsigned int &shaderID, unsigned int shaderType, const GLchar* shaderSource)
  - creates an empty shader and compile it
  - shaderType : GL_VERTEX_SHADER, GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER, GL_COMPUTE_SHADER
- bool CreateProgram(unsigned int &ProgramID, int n_args, arg1, ..., argn)
  - create a program and attach shaders to the program
  - n_args : number of shaders to attach

Shader Programming in OpenGL (shader.hpp)

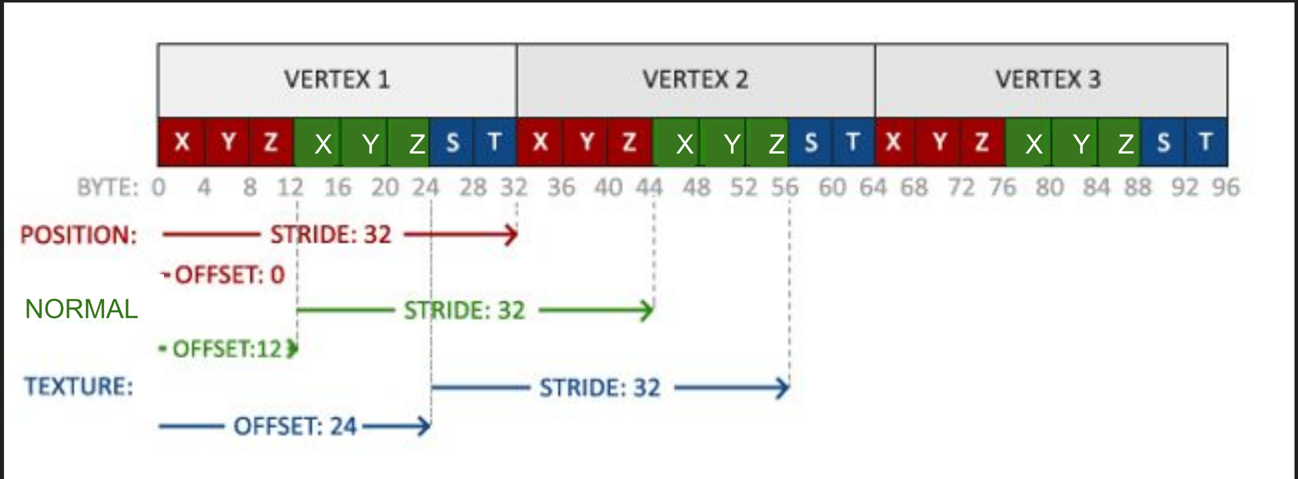
```
char* vertex_shader_resource = "...void main(){...}";      //shader source code
GLuint vert_id = glCreateShader(GL_VERTEX_SHADER);  //GL_FRAGMENT_SHADER
glShaderSource(vert_id, 1, &vertex_shader_resource, NULL);
glCompileShader(vert_id);
GLuint program_id = glCreateProgram();
glAttachShader(program_id, vert_id);
/* you can attach another shader (fragment shader) */
glLinkProgram(program_id);
glDetachShader(program_id, vert_id);
/* detach another shader (fragment shader)*/
```

Shader Programming in OpenGL (shader.hpp)

```
void display() {
  glUseProgram(program_id); //Phong, Dissolve, Ramp
  /* Shader program effect in this block */
  /* Pass parameters to shaders */
  glUseProgram(0);
  /* Pass 0 to stop the program*/
  glUseProgram(another_program_id);
  /* Another shader program effect */
  glUseProgram(0);
}
```

Data Connection - VBO

- VBO : Vertex Buffer Object


VERTEX 1
VERTEX 2
VERTEX 3
X Y Z X Y Z S T X Y Z X Y Z S T X Y Z X Y Z S T
BYTE: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72 76 80 84 88 92 96
POSITION:
STRIDE: 32
OFFSET: 0
NORMAL
STRIDE: 32
OFFSET:12
TEXTURE:
STRIDE: 32
OFFSET: 24

Implementation in OpenGL

```
struct VertexAttribute{ GLfloat position[3]; }; //normal, texcoord
//vector<glm:vec3> position;

VertexAttribute *vertices;

GLunit vboName;

glGenBuffers(1, &vboName);  //generate 1 buffer

glBindBuffer(GL_ARRAY_BUFFER, vboName);

glBufferData(GL_ARRAY_BUFFER, sizeof(VertexAttribute) * vertices_length,

vertices, GL_STATIC_DRAW);
```

Link to GLSL

```
glEnableVertexAttribArray(0);
glVertexAttribPointer(0,
3,
GL_FLOAT,
GL_FALSE,
sizeof(VertexAttribute),
(void*)(offsetof(VertexAttribute, position))
);
```

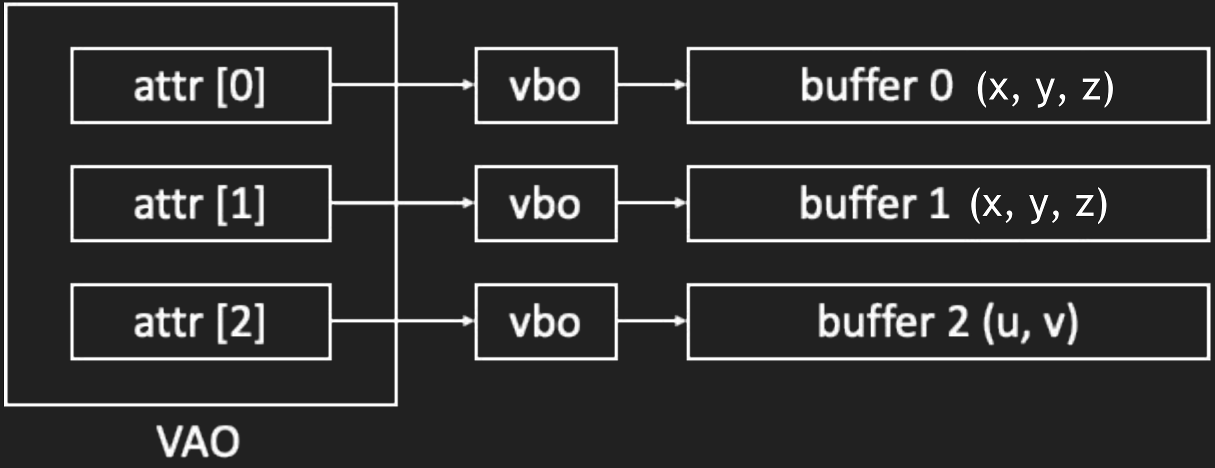
OpenGL

```
layout(location = 0) in vec3 pos;
```

GLSL (vertex shader)

Data Connection - VAO

- VAO : Vertex Array Object



```
GLuint vaoHandle;
GLunit vbo_ids[2];
void init(){
    glGenVertexArrays(1,&vaoHandle);
    glBindVertexArray(vaoHandle);
    glGenBuffers(2, vbo_ids);

    glBindBuffer(GL_ARRAY_BUFFER, vbo_ids[0]);
    glBufferData( /* ... */ );
    glEnableAttribArray(0);
    glVertexAttribPointer(0, /* ... */ );
    glBindBuffer(GL_ARRAY_BUFFER, 0);

    glBindBuffer(GL_ARRAY_BUFFER, vbo_ids[1]);
    glBufferData( /* ... */);
    glEnableAttribArray(1);
    glVertexAttribPointer(1, /* ... */ );
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

```
void display(){
    glUseProgram(program);
    glBindVertexArray(vaoHandle);
    /* draw objects with the VAO */
    glDrawArrays(GL_TRIANGLES,0,3);
    glBindVertexArray(0);
    glUseProgram(0);
    }
```

Data Connection - Uniform

- Uniform
  - act as parameters that the user can pass to the program
  - do not change in shader
- ~~Attribute~~ (deprecated)
  - alias to in
- ~~Varying~~ (deprecated)
  - alias to out

Data Connection - Uniform

```
GLfloat pmtx[16];    //getP(), getV()
glGetFloatv(GL_PROJECTION_MATRIX, pmtx);
GLint pmatLoc = glGetUniformLocation(program, "Projection");

glUseProgram(program);
glUniformMatrix4fv(pmatLoc, 1, GL_FALSE, pmtx);
glUseProgram(0);
```

OpenGL

```
uniform mat4 Projection;
```

GLSL(vertex shader)

Data Connection - Texture

```
GLint texLoc = glGetUniformLocation(program, "Texture");
glUseProgram(program);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texObj);
glUniform1i(texLoc, 0);
/* draw objects */
glBindTexture(GL_TEXTURE_2D, 0);
glUseProgram(0);
```

OpenGL

```
layout(binding = 0) uniform sampler2D Texture;
in vec2 texcoord;
out vec4 outColor;
void main() { outColor = texture2D(Texture, texcoord); }
```

GLSL (fragment shader)

GLSL Syntax

- Basic Variable Types
  - vec2, vec3, vec4, ...
  - mat2, mat3, mat4, ...
  - float, int, bool, ...
  - sampler2D, ...

- Basic Functions
  - max, min, sin, cos, pow, log, ...
  - dot, normalize, reflect, ...
  - transpose, inverse, ...

Vertex Shader

- must have gl_Position

```
/* Example of vertex shader */
#version 330
layout(location = 5) in vec4 in_Pos;
layout(location = 6) in vec4 in_Norm;
uniform mat4 MV;
uniform mat4 P;
out vec3 normal;
void main() {
    gl_Position = P * MV * in_Pos;
    normal = vec3(
      /* normal after modelview transform
    */
    );
}
```

Fragment Shader

- must have a out vec4 for color buffer

```
/* Example of fragment shader */
#version 330
in vec3 normal;
out vec4 outColor;
void main() {
    if(abs(normal.z) < 0.3) {
      outColor = vec4(1.0);
    }
    else {
      outColor = vec4(1.0, vec2(0.0), 1.0);
    }
}
```

Reference

- https://en.wikipedia.org/wiki/OpenGL
- https://www.opengl.org/wiki/Rendering_Pipeline_Overview
- http://www.cs.cmu.edu/afs/cs/academic/class/15462-s15/www/lec_slides/lec02.pdf