Introduction to GLSL

GLSL

- OpenGL Shading Language is a high-level language based on C programming language.
- Added since OpenGL 2.0 version.
- More flexible and efficient on rendering.

Rendering pipeline in OpenGL

Fixed function pipeline

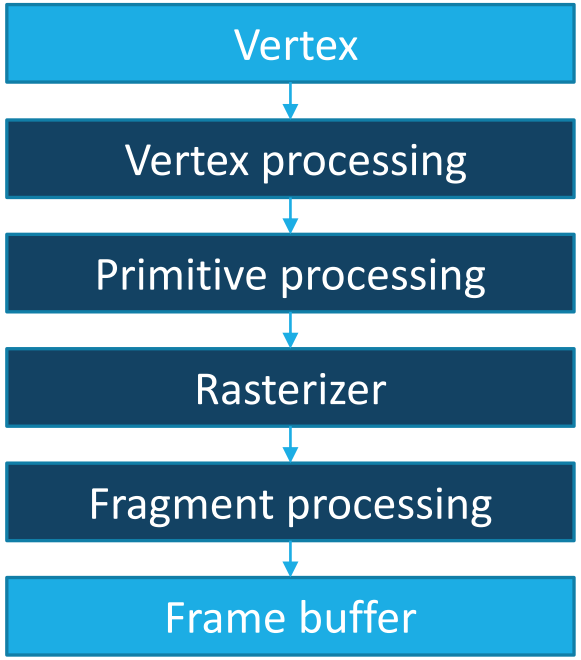
- You can't change each function and the order of execution.
- Deprecated since OpenGL 3.0
- Easy to learn, but less flexible.

Programmable pipeline

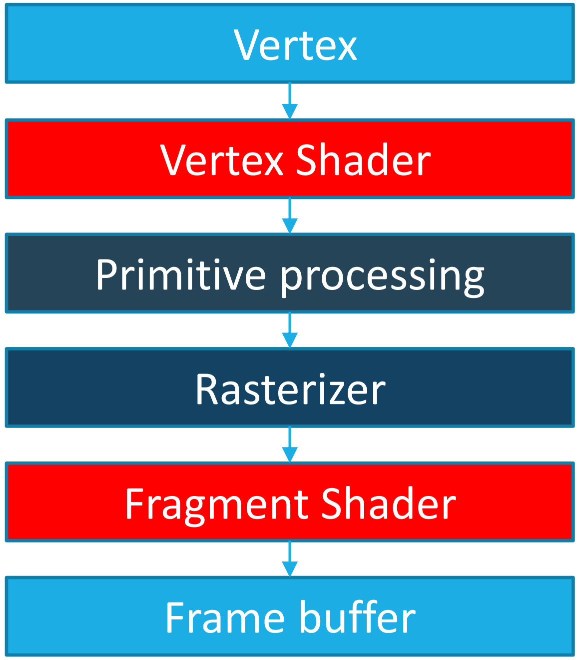
- Use shader programs, which was written in GLSL.
- More difficult to learn, but can achieve more advanced effects.

Rendering pipeline in OpenGL

Fixed function pipeline



Programmable pipeline



Shader

- A program designed by users.
- Run in GPU pipeline.

Vertex Shader

- **Input:** Single vertex
- **Output:** Single vertex

Fragment Shader

- **Input:** One pixel
- **Output:** One or no pixel

Shader Initialize

```
void shaderInit() {
    // Create shader program
    GLuint vert = createShader("Shaders/example.vert", "vertex");
    GLuint frag = createShader("Shaders/example.frag", "fragment");
    program = createProgram(vert, frag);

    // Generate buffers
    glGenBuffers(1, &vboName);
    glBindBuffer(GL_ARRAY_BUFFER, vboName);

    // Copy vertex data to the buffer object
    VertexAttribute *vertices;
    vertices = drawTriangle();
    glBufferData(GL_ARRAY_BUFFER, sizeof(VertexAttribute) * verticeNumber, vertices, GL_STATIC_DRAW);
    glEnableVertexAttribArray(0);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(VertexAttribute), (void*)(offsetof(VertexAttribute, position)));
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}
```

Shader setting

```
void shaderInit() {
    // Create shader program
    GLuint vert = createShader("Shaders/example.vert", "vertex");
    GLuint frag = createShader("Shaders/example.frag", "fragment");
    program = createProgram(vert, frag);
```

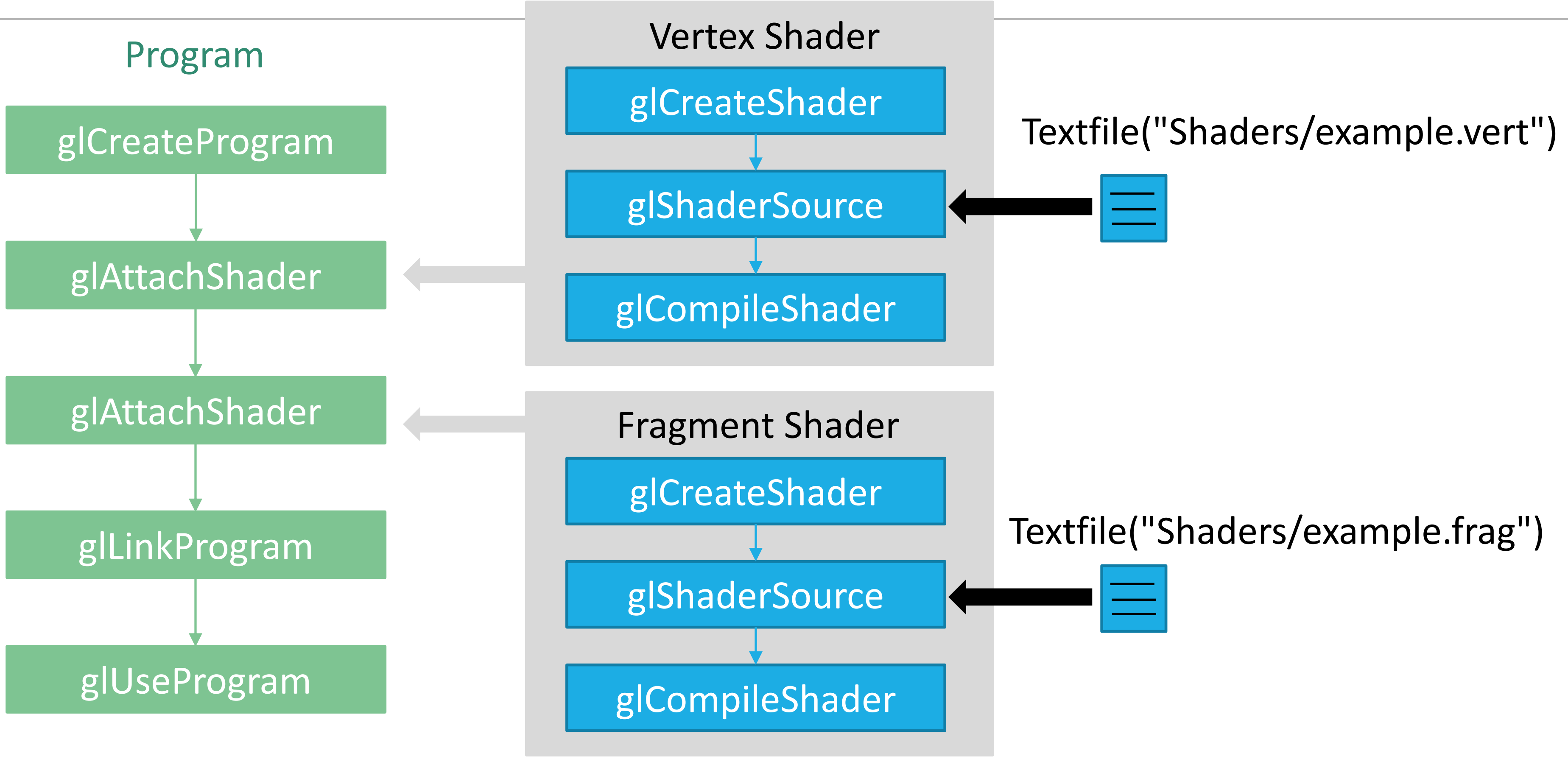
- In the function : createShader() (defined in shader.h)
  - GLuint **glCreateShader** (GLenum shaderType);
    - Specifies the type of shader to be created and creates an empty shader object.
    - shaderType : GL_COMPUTE_SHADER, GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER, GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER
  - void **glShaderSource** (GLuint shader, GLsizei count, const GLchar **string, const GLint *length);
    - Sets the source code in shader to the source code in the array of strings specified by string.
    - Ex : string = & textFileRead("Shaders/example.vert")
  - void **glCompileShader**(GLuint shader);
    - Compile the shader.

Shader setting

```
void shaderInit() {
    // Create shader program
    GLuint vert = createShader("Shaders/example.vert", "vertex");
    GLuint frag = createShader("Shaders/example.frag", "fragment");
    program = createProgram(vert, frag);
```

- In the function : createProgram() (defined in shader.h)
  - GLuint **glCreateProgram**(void);
    - creates a program object.
  - void **glAttachShader** (GLuint program, GLuint shader);
    - Attach the shader object to the program object.
  - void **glLinkProgram** (GLuint program);
    - Link this program
  - void **glDetachShader** (GLuint program, GLuint shader);
    - Detaches the shader object from the program object.

Shader setting



```
GLuint vboName;
```

```
glGenBuffers(1, &vboName);
glBindBuffer(GL_ARRAY_BUFFER, vboName);
```

Vertex Buffer Objects (VBO)

- Since the vertex shader access only one vertex at one time, we use Vertex Buffer Objects to make the execution be faster. The advantage of using these buffered objects is that we can send a large amount of vertex data from system memory to GPU memory at one time instead of sending it once per vertex.
- Step 1 : Use **glGenBuffers()** to generate vertex buffer objects
  - void **glGenBuffers** (GLsizei n, GLuint * buffers);

    n : Specifies the number of buffer object names to be generated.
    buffers : Specifies an array in which the generated buffer object names are stored.
- Step 2 : Use **glBindBuffer()** to bind the target buffer, which is GL_ARRAY_BUFFER here.
  - void **glBindBuffer** (GLenum target, GLuint buffer);

    target : GL_ARRAY_BUFFER、GL_TEXTURE_BUFFER、…….
    buffer : Specifies the name of a buffer object.

Vertex Buffer Objects (VBO)

```
class VertexAttribute {
public:
    GLfloat position[3];
    void setPosition(float x, float y, float z) {
        position[0] = x;
        position[1] = y;
        position[2] = z;
    };
};
```

- Step 3 : Set up vertices
- Step 4 : Use **glBufferData()** to copy the data into the target.
  - void **glBufferData** (GLenum target, GLsizeiptr size, const GLvoid * data, GLenum usage);

target : GL_ARRAY_BUFFER、GL_TEXTURE_BUFFER、.......

size : Specifies the size in bytes of the buffer object's new data store.

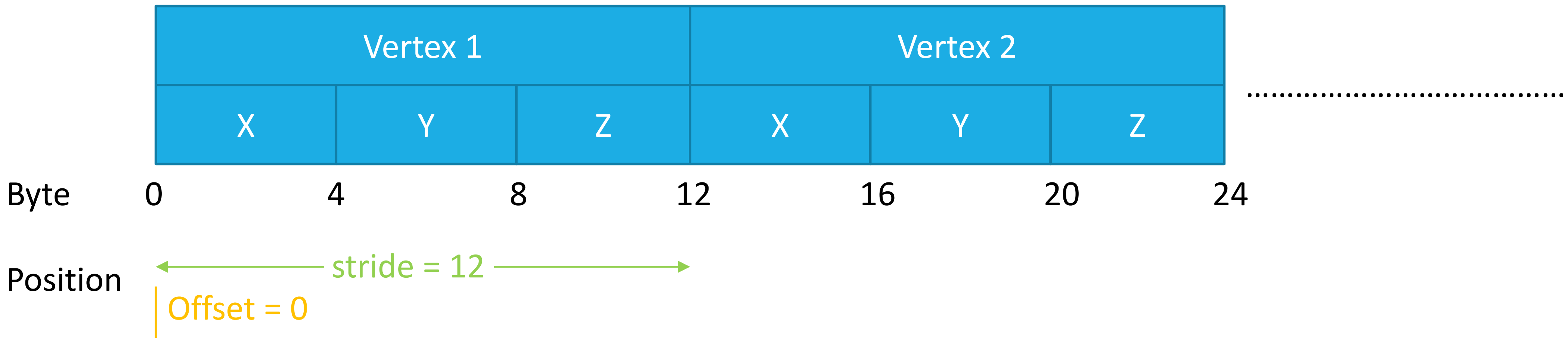
data : Specifies a pointer to data that will be copied into the data store for initialization, or NULL if no data is to be copied.

usage : Specifies the expected usage pattern of the data store. Ex: GL_STATIC_DRAW means the data store contents will be modified once and used at most a few times.

```
VertexAttribute *vertices;
vertices = drawTriangle();
glBufferData(GL_ARRAY_BUFFER, sizeof(VertexAttribute) * verticeNumber, vertices, GL_STATIC_DRAW);
```

Vertex Buffer Objects (VBO)

- Now the VBO looks like this



Vertex Attribute Pointer

- We can use **glVertexAttribPointer()** to link the vertex buffer with the vertex shader input.
- void **glVertexAttribPointer** (GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const GLvoid * pointer);
  - index : Specifies the index of the generic vertex attribute to be modified.
  - size : Specifies the number of components per generic vertex attribute.
  - type : Specifies the data type of each component in the array. Ex: GL_FLOAT
  - normalized : Specifies whether fixed-point data values should be normalized or not.
  - stride : Specifies the byte offset between consecutive generic vertex attributes.
  - pointer : Specifies a offset of the first component of the first generic vertex attribute in the array in the data store of the buffer currently bound to the GL_ARRAY_BUFFER target. The initial value is 0.

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

In vertex shader →

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

Enable the vertex attribute arrays

- Remember to use **glEnableVertexAttribArray()** to enable the vertex attribute arrays because default it is disabled.
- void **glEnableVertexAttribArray** (GLuint index);
  - index : the same as the index of **glVertexAttribPointer().**

```
glEnableVertexAttribArray(0);
```

Unbind the VBO

- Use **glBindBuffer()** with the buffer set to zero to unbind the target buffer.
  - Ex: glBindBuffer(GL_ARRAY_BUFFER, 0) means to unbind the VBO previously bound.

```
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

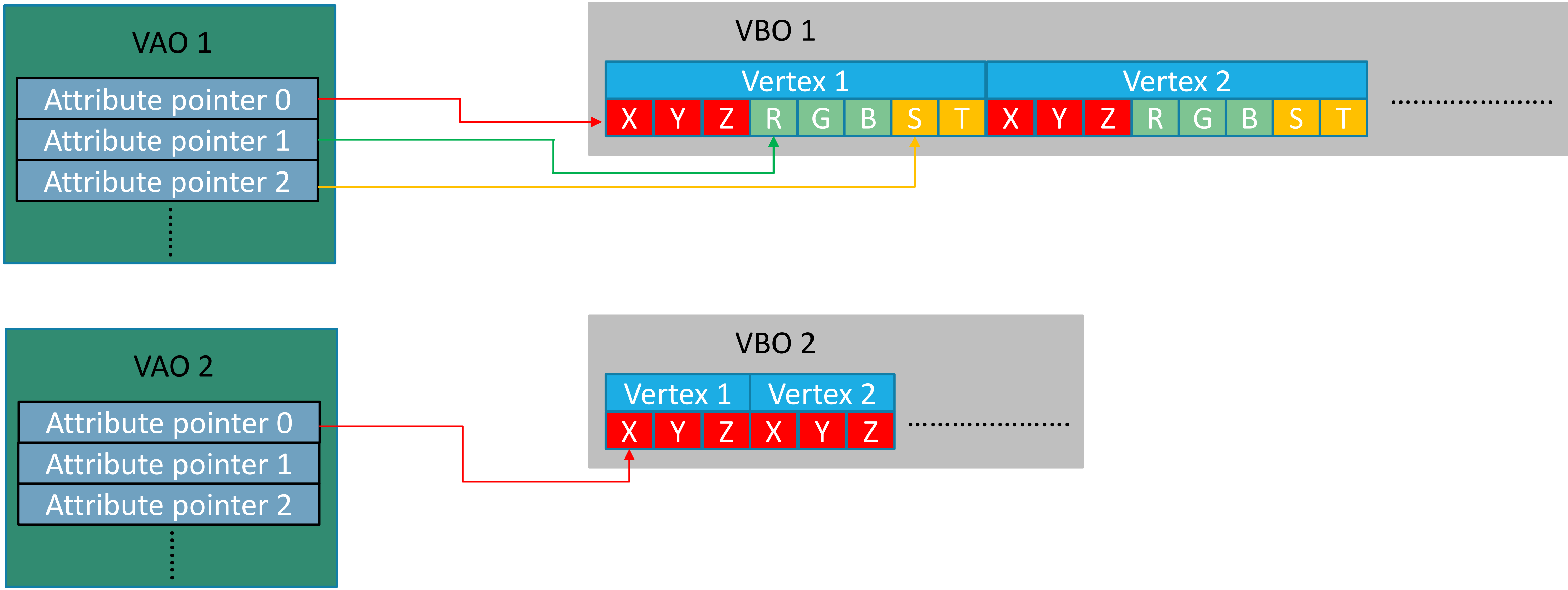
Supplement -- Vertex Array Object (VAO)

- If you want to render more than one objects, you have to repeat above steps (slides 10 ~15).

  → very troublesome

- Use VAO(Vertex Array Object) to handle this problem.
- First, you have to set up all the VAOs with its corresponding VBO, including all VertexAttributePointer. After that, every time you want to render a certain object, you just need to bind its VAO.

VAO


VAO 1
Attribute pointer 0
Attribute pointer 1
Attribute pointer 2
VBO 1
Vertex 1
Vertex 2
X Y Z R G B S T X Y Z R G B S T
VAO 2
Attribute pointer 0
Attribute pointer 1
Attribute pointer 2
VBO 2
Vertex 1
Vertex 2
X Y Z X Y Z

VAO setting (similar to VBO)

```
GLuint VAO;
glGenVertexArrays(1, &VAO);
glBindVertexArray(VAO);
```

- Step 1 : Use **glGenVertexArrays()** to generate vertex array objects
  - void **glGenVertexArrays** (GLsizei n, GLuint * arrays);

    n : Specifies the number of vertex array object names to be generated.

    arrays : Specifies an array in which the generated vertex array object names are stored.

- Step 2 : Use **glBindVertexArray()** to bind a vertex array object.
  - void **glBindVertexArray** (GLuint array)

    array : Specifies the name of the vertex array to bind.

VAO setting (similar to VBO)

- Step 3 : Setting up its corresponding VBO, for example :
  - glBindBuffer(GL_ARRAY_BUFFER, VBO);
  - glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
  - glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(GLfloat), (GLvoid*)0);
  - glEnableVertexAttribArray(0);
- Step 4 : Use **glBindVertexArray (0)** with the array's name set to zero to unbind the array object.
  - void **glBindVertexArray** (GLuint array)

    Ex: glBindVertexArray(0) means to unbind the VAO previously bound.

When Rendering

- Step 1 : Use **glBindVertexArray(VAO)** to bind the VAO you want.
- Step 2 : Use **glDrawArrays()** to render primitives from vertex array data.
  - void **glDrawArrays()** (GLenum mode, GLint first, GLsizei count);

    mode : Specifies what kind of primitives to render. Ex: GL_POINTS, GL_LINES, GL_TRIANGLE_STRIP……

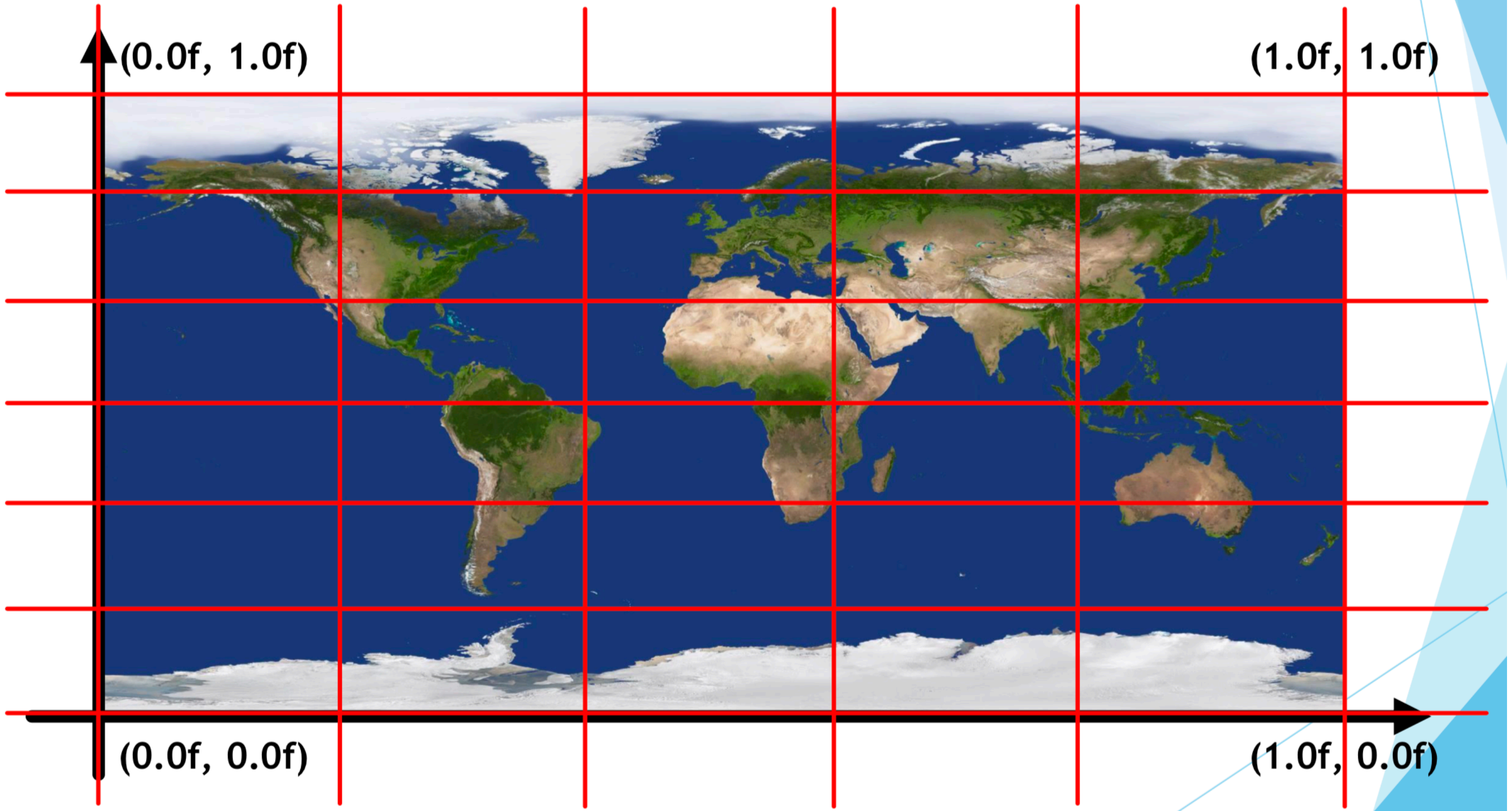
    first : Specifies the starting index in the enabled arrays.

    count : Specifies the number of indices to be rendered.
- Step 3 : Remember to unbind the VAO. (**glBindVertexArray(0)**)

*Every time you want to render another object, you just need to bind another VAO.

Texture

Texture coordinate



How to load and bind a texture

- Put “FreeImage.h” in folder “include”.
- Put “FreeImage.lib” in folder “lib”.
- Put “FreeImage.dll” in folder “dll”.

We will give you a visual studio project including these files and textures.

```
GLuint texture;

void LoadTexture(char* pFilename) {
    glEnable(GL_TEXTURE_2D);
    FIBITMAP* pImage = FreeImage_Load(FreeImage_GetFileType(pFilename, 0), pFilename);
    FIBITMAP *p32BitsImage = FreeImage_ConvertTo32Bits(pImage);
    int iWidth = FreeImage_GetWidth(p32BitsImage);
    int iHeight = FreeImage_GetHeight(p32BitsImage);
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, iWidth, iHeight, 0,
        GL_BGRA, GL_UNSIGNED_BYTE, (void*)FreeImage_GetBits(p32BitsImage));

    glGenerateMipmap(GL_TEXTURE_2D);

    glBindTexture(GL_TEXTURE_2D, 0);
    FreeImage_Unload(p32BitsImage);
    FreeImage_Unload(pImage);
}
```

How to load and bind a texture

- void glEnable(Glenum cap);
  - Use GL_TEXTURE_2D to enable texture
- Use FreeImage library to load and free texture memory
- void glGenTextures(GLsizei n, GLuint * textures);
  - Takes as input how many textures we want to generate and stores them in a unsigned int array
- void glBindTexture(GLenum target, GLuint texture);
  - Bind a named texture to a texturing target
- void glTexImage2D(GLenum target, GLint level, GLint internalformat, GLsizei width, GLsizei height, GLint border, GLenum format, GLenum type, const GLvoid * data);
  - Generate a two-dimensional texture image

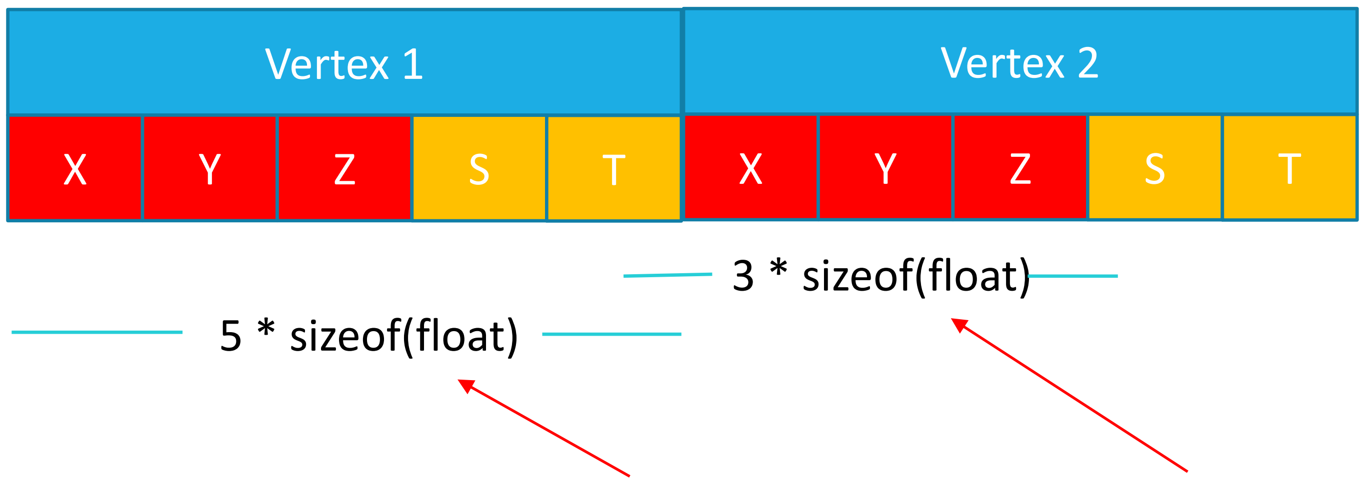
How to load and bind a texture

- void glTexParameteri(GLenum target, GLenum pname, GLint param);
- Texture wrapping
  - Texture coordinates usually range from (0,0) to (1,1) but if we specify coordinates outside this range, the default behavior of OpenGL is to repeat the texture images
  - glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
  - glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
- Texture filtering
  - Texture coordinates do not depend on resolution but can be any floating point value, thus OpenGL has to figure out which texture pixel to map the texture coordinate to
  - glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_Nearest);
  - glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

How to load and bind a texture

- void glTexEnvf(GLenum target, GLenum pname, GLfloat param);
  - Set a texture environment parameter
- void glGenerateMipmap(GLenum target);
  - Generate mipmaps for a specified texture object

Applying texture




```
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(VertexAttribute), (void*)(offsetof(VertexAttribute, texcoord)));
glEnableVertexAttribArray(1);
```

In vertex shader

- We need to let the vertex shader to accept the texture coordinates as a vertex attribute and then forward the coordinates to the fragment shader – using location
- Your output object name and type in vertex shader must be as same as the input object name and type in fragment shader

e.g. In vertex shader, our output object is called Texcoord

See next page

```
#version 430

layout(location = 0) in vec3 position;
layout(location = 1) in vec2 texcoord;

out vec2 Texcoord

void main() {
  gl_Position = vec4(position, 1.0);
  Texcoord = texcoord
}
```

```
glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, sizeof(VertexAttribute), (void*)(offsetof(VertexAttribute, texcoord)));
glEnableVertexAttribArray(1);
```

In fragment shader

- How do we pass the texture object to the fragment shader?
  - GLSL has a built-in data-type for texture objects called a sampler
  - We can then add a texture to the fragment shader by simply declaring a uniform sampler2D that we later assign our texture to

- Your output object name and type in vertex shader must be as same as the input object name and type in fragment shader

```
#version 430

in vec2 Texcoord

out vec4 FragColor

uniform sampler2D ourTexture;

void main() {
    FragColor = texture(ourTexture, Texcoord)
}
```

e.g. In fragment shader, our input object is called Texcoord

Assign texture to fragment shader

- GLint glGetUniformLocation(GLuint program, const GLchar *name);
  - return the location of a uniform variable
- void glActiveTexture(GLenum texture);
  - Select active texture unit
- void glUseProgram(GLuint program);
  - Installs a program object as part of current rendering state
- void glUniform1i(GLint location, GLint v0);
  - specify the value of a uniform variable for the current program object

Main.cpp

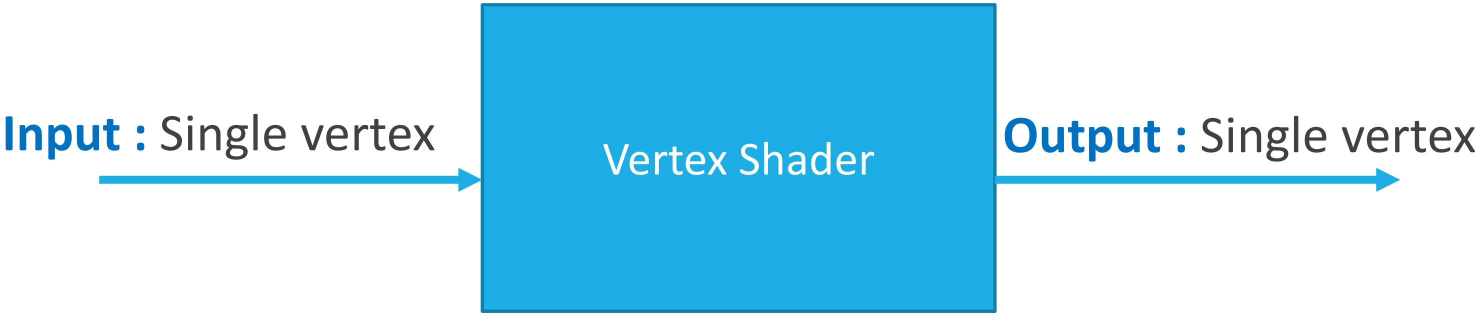
```
GLint texLoc = glGetUniformLocation(program, "ourTexture");
glUseProgram(program);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texture);
glUniform1i(texLoc, 0);
```

Shader.frag

```
#version 430

in vec2 Texcoord

out vec4 FragColor

uniform sampler2D ourTexture;

void main() {
    FragColor = texture(ourTexture, Texcoord)
}
```

GLSL Shader Coding example

example.vert



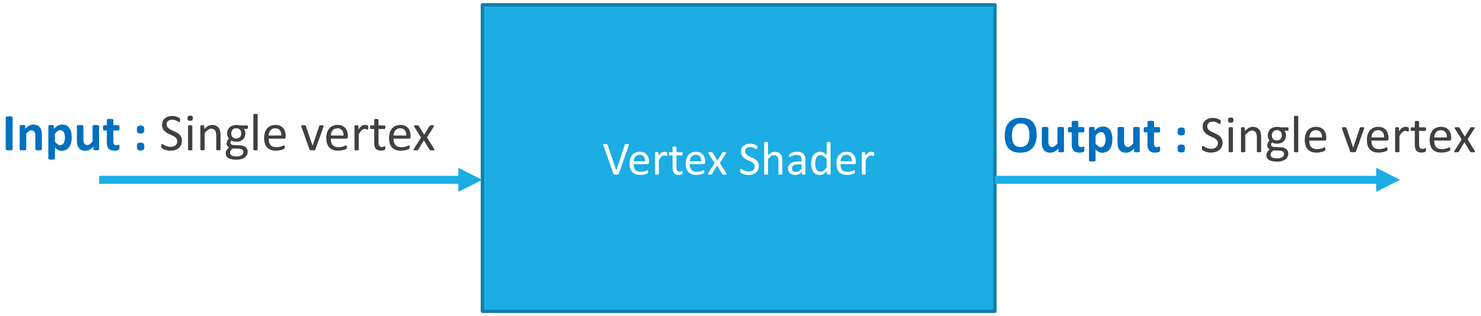
In vertex shader, first, send the vertex's position to the shader using location. Then transform the position to clip space by model view and projection matrix.

```
#version 430

layout(location = 0) in vec3 position;

uniform mat4 Projection;
uniform mat4 ModelView;

void main() {
  gl_Position = Projection * ModelView * vec4(position, 1.0);

}
```

- **layout:**
  Layout qualifiers affect where the storage for a variable comes from.

- **uniform:**
  A uniform is a global shader variable, they do not change from one shader invocation to the next within a particular rendering call.

example.vert

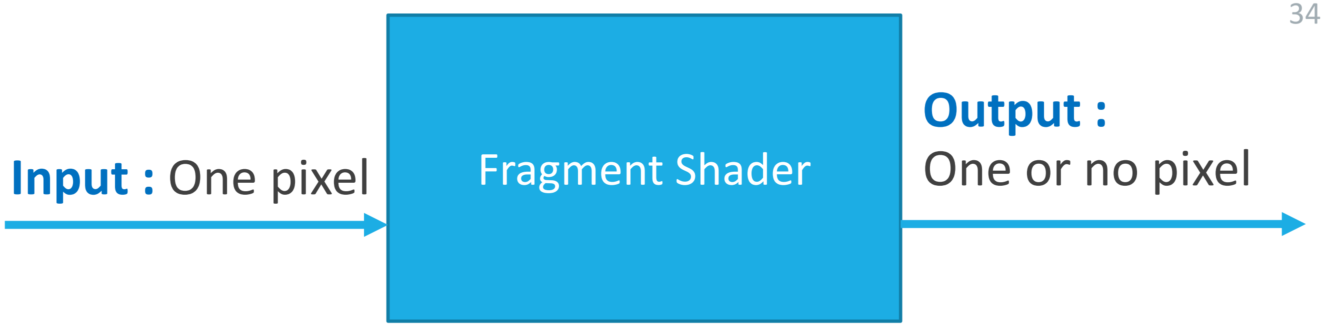


- **gl_Position:**
  Contain the clip space position of the current vertex.

/*HW2 HINT*/
Try to get the texture coordinate from the vertex. Then pass it to the fragment shader.

```
#version 430

layout(location = 0) in vec3 position;

uniform mat4 Projection;
uniform mat4 ModelView;

void main() {
  gl_Position = Projection * ModelView * vec4(position, 1.0);

}
```

example.frag



In fragment shader, we output the final color of the pixel.

```
#version 430
out vec4 frag_color;

void main() {
    frag_color = vec4(0.0, 1.0, 0.0, 1.0);
}
```

/*HW2 HINT*/
To output the texture, we may need the coordinates from the previous shader.
See the instruction in the texture chapter…

Reference

https://learnopengl.com/Advanced-OpenGL/

https://learnopengl.com/Getting-started/Textures

https://www.khronos.org/opengl/wiki/Built-in_Variable_(GLSL)