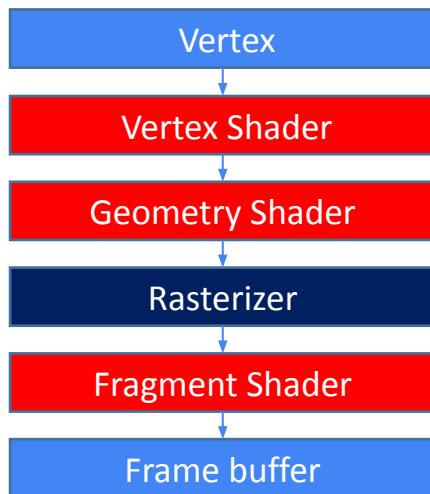


OpenGL shader & GLSL

HW2 Tutorial

OpenGL pipeline



Shader

A program designed by users.

Run in GPU pipeline.

Vertex Shader

Geometry Shader

Fragment Shader

•**Input:** Single vertex

•**Output:** Single vertex

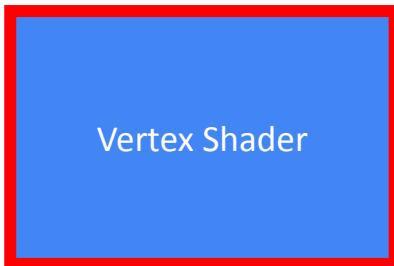
•**Input:** One primitive

•**Output:** Can be more than one primitive

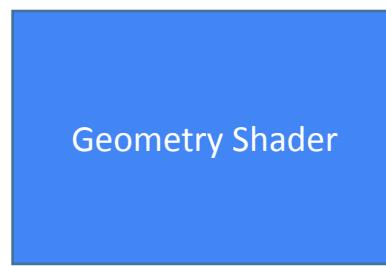
•**Input:** One pixel

•**Output:** One or no pixel

Shader



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



- Input:** One primitive
- Output:** Can be more than one primitive



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

Shader setting

In the function : createShader()

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**.

```
vertexShader = createShader(dirShader + "vertexShader.vert", "vert");
fragmentShader = createShader(dirShader + "fragmentShader.frag", "frag");
```

Shader setting

In the function : createProgram()

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.

```
shaderProgram = createProgram(vertexShader, fragmentShader);
```

Use program

```
void display() {  
    glUseProgram(program_id);  
    /* 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);  
}
```

program_id is the return GLuint from [glCreateShader](#)

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.

```
unsigned int VAO, VBO[3];  
glGenBuffers(3, VBO);  
glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
```

Vertex Buffer Objects (VBO)

Step 3 : Set up the data

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.

```
glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT) * (model.positions.size()), &(model.positions[0]), GL_STATIC_DRAW);
```

Implementation in OpenGL

```
struct VertexAttrib{ GLfloat position[3]; ... };
```

```
VertexAttrib *vertices;
```

```
GLunit vboName;
```

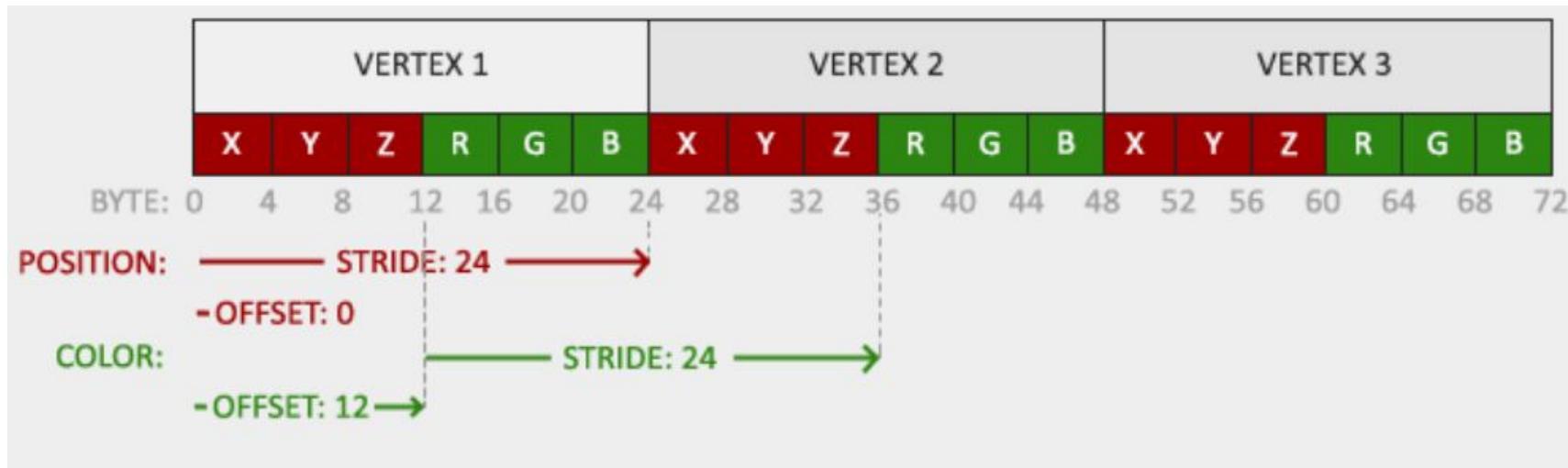
```
glGenBuffers(1, &vboName); //generate 1 buffer
```

```
glBindBuffer(GL_ARRAY_BUFFER, vboName);
```

```
glBufferData(GL_ARRAY_BUFFER, sizeof(VertexAttribute) *  
vertices_length,
```

```
vertices, GL_STATIC_DRAW);
```

Vertex Buffer Objects (VBO)



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(GL_FLOAT) * 3, 0);
```

Vertex Attribute Pointer

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

OpenGL

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

GLSL (vertex shader)

Unbind the VBO

Use **glBindBuffer()** with the buffer set to zero to unbind the target buffer.

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

Vertex Array Object (VAO)

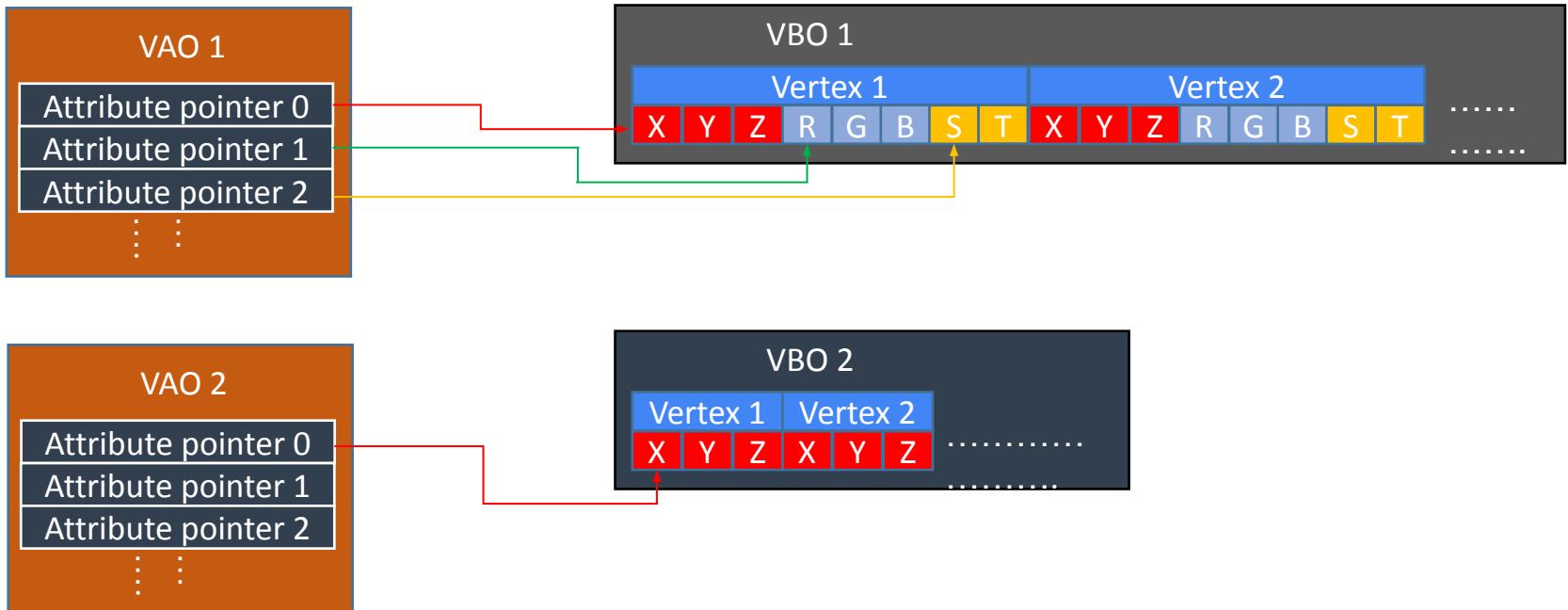
If you want to render more than one objects, you have to repeat above steps (slides 8 ~14).

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 VertexAttribPointer. After that, every time you want to render a certain object, you just need to **bind its VAO**.

Vertex Array Object (VAO)



Vertex Array Object (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.

```
unsigned int VAO, VBO[3];  
glGenVertexArrays(1, &VAO);  
glBindVertexArray(VAO);
```

Vertex Array Object (VAO)

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.

```
glBindVertexArray(groundVAO);
glDrawArrays(GL_TRIANGLES, 0, groundObject->positions.size());
glBindVertexArray(0);
```

Data Connection - Uniform

```
mat4 view = lookAt(vec3(0, 5, 5), vec3(0, 0.5, 0), vec3(0, 1, 0));  
GLint vLoc = glGetUniformLocation(program, "View");  
  
glUseProgram(program);  
glUniformMatrix4fv(vLoc, 1, GL_FALSE, value_ptr(view));  
glUseProgram(0);
```

OpenGL

```
uniform mat4 View;
```

GLSL (vertex 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 430  
  
layout(location = 0) in vec3 position;  
  
uniform mat4 Projection;  
uniform mat4 ModelView;  
  
out vec3 color; //to fragment shader  
  
void main() {  
    gl_Position = Projection * ModelView * vec4(position, 1.0);  
    color = vec3(1.0, 0.0, 0.0);  
}
```

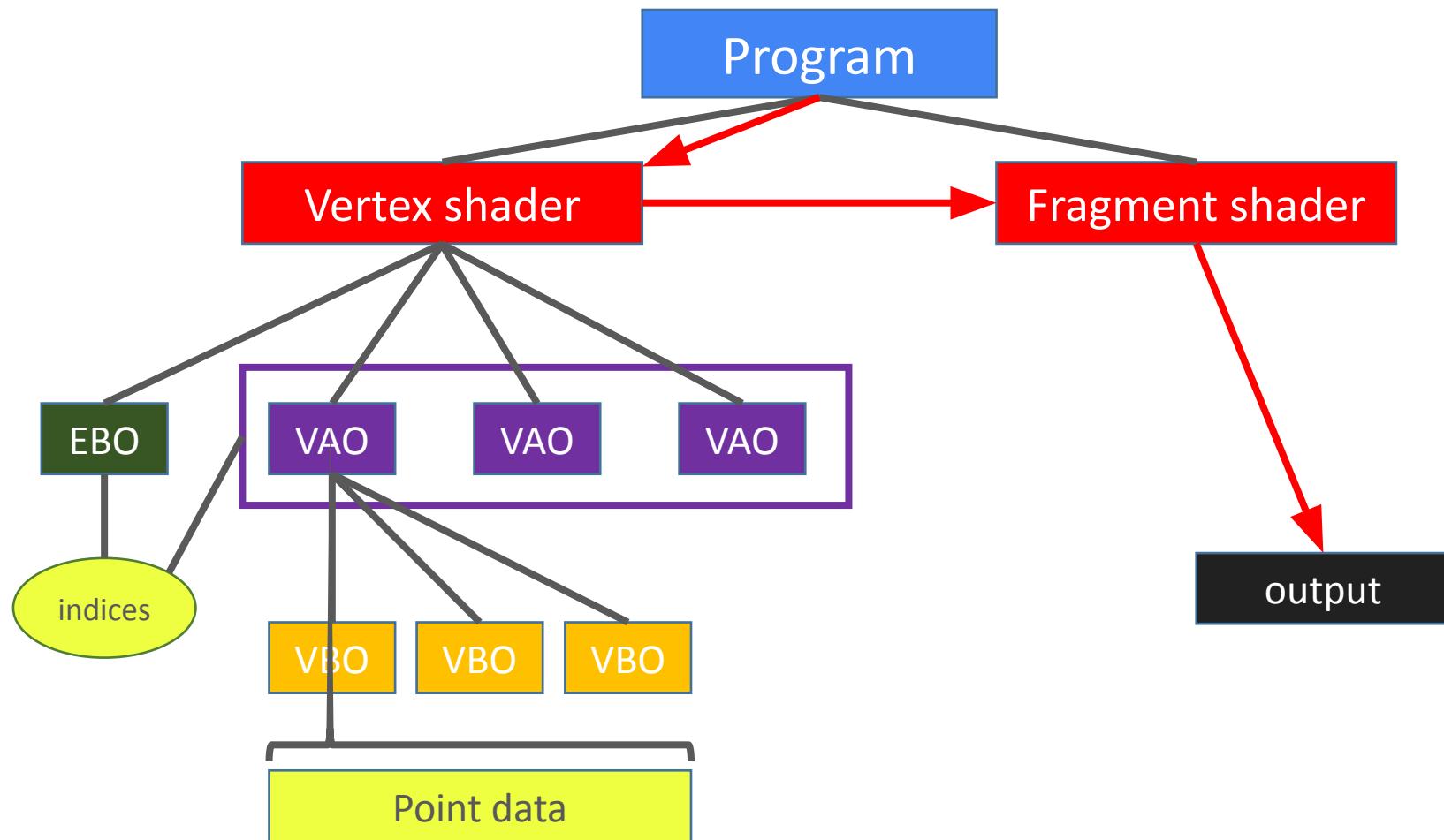
Fragment Shader

- must have a `out vec4` for color buffer

```
/* Example of fragment shader */
#version 430

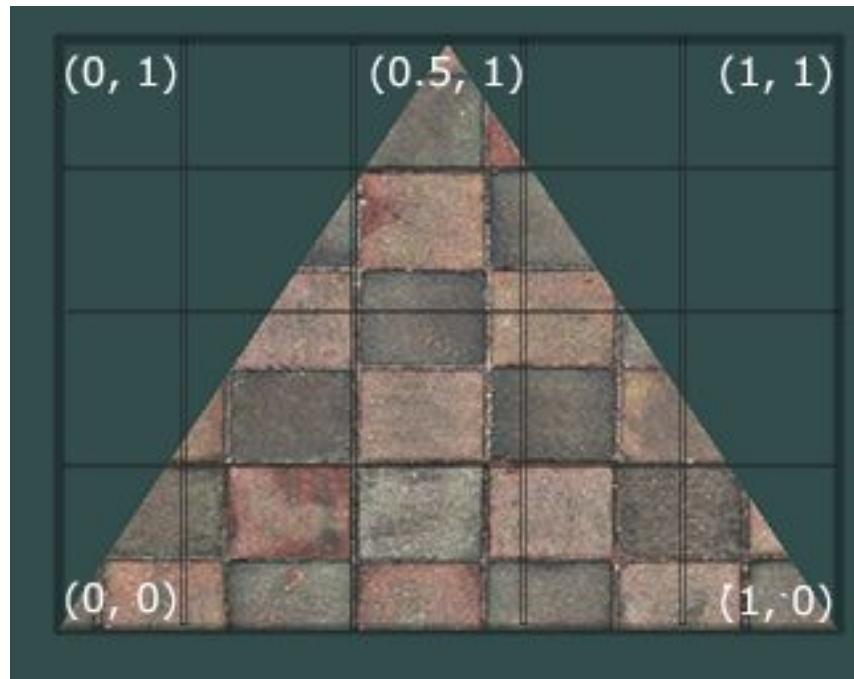
in vec3 color; //from vertex shader
out vec4 frag_color;

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



Texture in OpenGL

Texture coordinate



How to load and bind a texture

- void `glEnable`(GLenum cap);

 Use `GL_TEXTURE_2D` to enable texture

- 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 (Ex.`GL_TEXTURE_1D`, `GL_TEXTURE_2D`,
`GL_TEXTURE_3D`, `GL_TEXTURE_1D_ARRAY`)

```
unsigned int texture;
glEnable(GL_TEXTURE_2D);
glGenTextures(1, &texture);
glBindTexture(GL_TEXTURE_2D, texture);
```

How to load and bind a texture

- void `glTexParameter`i(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
 - `glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);`
 - `glTexParameter(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
 - `glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_Nearest);`
 - `glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);`

```
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameter(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

When Rendering

- void **glActiveTexture(GLenum textureUnit);**

selects which **texture unit** subsequent texture state calls will affect. You can use the **textureUnit** from **GL_TEXTURE0** to **GL_TEXTUREn**, $0 \leq n < \text{GL_MAX_TEXTURE_UNITS}$, and texture units are subsequent, you can use **GL_TEXTUREn** or **GL_TEXTURE0 + n**. (Ex. **GL_TEXTURE2 = GL_TEXTURE0 + 2**)

- 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

```
glActiveTexture(GL_TEXTURE0);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
```

Data Connection - Texture

```
glActiveTexture(GL_TEXTURE0);
 glGenTextures(1, &texture);
 glBindTexture(GL_TEXTURE_2D, texture);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER, GL_LINEAR);
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

LoadTexture() function

```
/* load texture image as data/
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
GL_RGB, GL_UNSIGNED_BYTE, data);
```

Different : No need to unbind texture object

```
fishTexture = loadTexture(dirTexture + "fish.jpg");
columnTexture = loadTexture(dirTexture + "column.jpg");
groundTexture = loadTexture(dirTexture + "ground.jpg");
```

```
glUseProgram(program);
glGetUniformLocation(program, "Texture");
glUniform1i(texLoc, 0);
/* draw objects */
glUseProgram(0);
```

OpenGL main loop

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

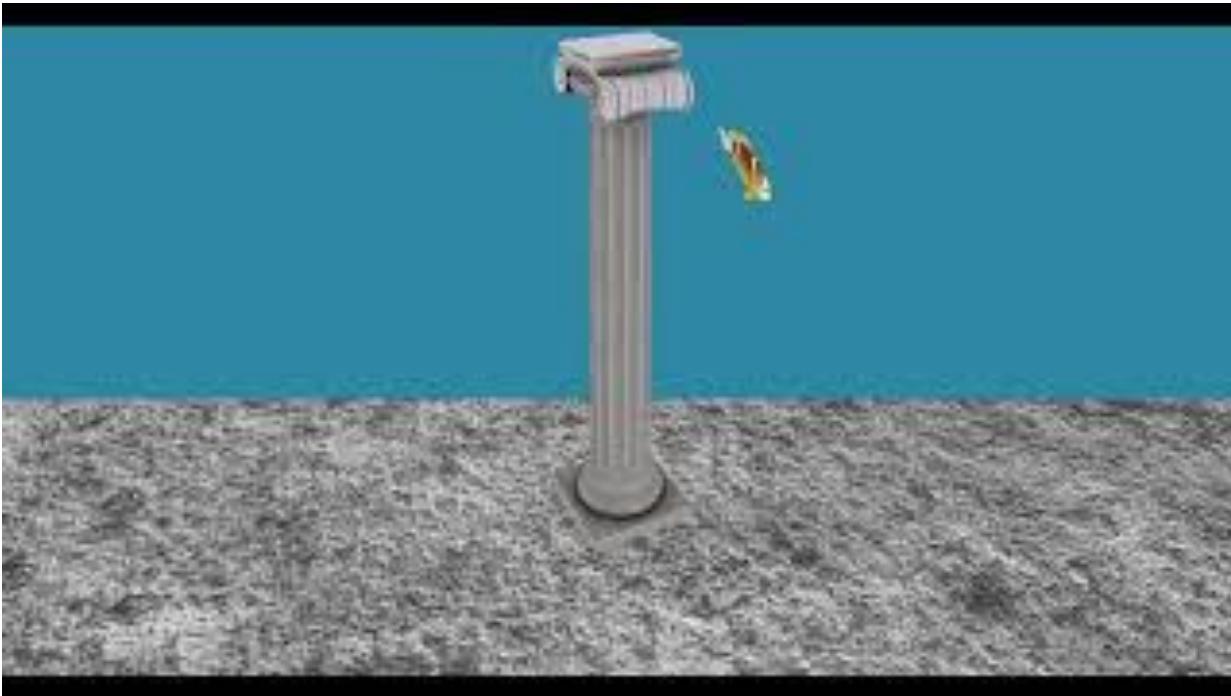
GLSL (fragment shader)

Homework 2 - Fish in Atlantis

The clownfish Coral returned to her beloved home in the ancient Atlantean quarter after a year of travel. She only found that the familiar, beautiful coral reef **replaced by a single stone column**. She discovered the city's Guardians had initiated a total "order" protocol, digitally modeling and replacing all living structures. Now a desperate texture plastered across the rotating column, and the residual shaders in the water were causing involuntary visual effects. It forces Coral's own **body to rapidly squeeze** and **her intensity to cyclically change**, as a constant reminder that their living reality had been tragically replaced by a sterile, simulated one.

Homework 2 - demo

[video link](#)



Homework 2 - spec

Goal:

Using GLSL to draw three model with its texture simultaneously

Using shaders to achieve some special effects

Spec:

Fish:

- scale the fish to a smaller size: (0.05, 0.05, 0.05)
- rotate it in x axis with -90 degrees
- then place it to (0, 0, 3)

Column:

- scale the column to a smaller size: (0.05, 0.05, 0.05)
- rotate it in x axis with -90 degrees
- then place it to (0, -4, 0)

Ground:

- scale the ground to a bigger size: (35, 1, 25)
- then place it to (0, -5, 8)

Homework 2 - spec

Rotate and revolve :

- The column need to self rotate at 10 degrees per frame around y axis
- The fish need to revolve the column at -20 degrees per frame, and swim up and down slowly. **You need to make sure the fish is still visible in the window.**

Keyboard function :

- press key 'R' to **rotate the Fish 360 degree around -x axis (can't be interrupted)**
- press key 'S' to start/stop **squeezing the Fish**
- press key 'B' to switch the color mode of the Fish between normal and **breathing light**

Red characters just let
you know why I
choose key R, S, B as
input(`-ω-`)

Homework 2 - spec

Squeezing:

- For the Fish coordinate (x, y, z) in vertex shader,
 $x += z * \sin(squeezeFactor) / 2;$
 $z += x * \sin(squeezeFactor) / 2;$
- When squeezing, squeezeFactor +90 degree/sec

Breathing:

- When the color mode of the Fish is **normal**
the color of the Fish should be the texture color
- When the color mode of the Fish is **breathing light**
the color of the Fish should be the texture color * (breathing color * intensity)

Breathing color:

- Breathing color is yellow
- Intensity is affected by $\sin(\text{time})$, scale and offset factor

Homework 2 - advanced design example

Stripes:

- Get the x, y coordinate of column, use the atan() calculate the angle in XY plane, and normalize it.
- Scale for frequency and convert it into into a 0.0 or 1.0 color switching factor.
- Use mix() to blend between black and white.

Gradient:

- When the color mode of the Fish is **normal**
the color of fish should be the texture color
- When the color mode of the Fish is **gradient light**
the color of fish should be the texture color * (gradient color * intensity)

Gradient color:

- Get the z coordinate of fish, normalize it into a 0.0 to 1.0 range factor for color blending.
(recommend to use scale and offset to control the contrast and range)
- Use the factor to one of the RGB channel.

Restrictions !!

- Your GLSL version should $\geq \#version 330$
- Deprecated shader syntaxes are not allowed, e.g. attribute, varying
- You are only allowed to use VBO and VAO when rendering model
- You are only allowed to pass uniform data to shader using `glUniform*` series function
- Using built-in uniform variables in shader is forbidden!
(That is, you **cannot** use `gl_ModelViewMatrix` or `gl_NormalMatrix` ...etc)
The only `gl_XXX` term should be in your shader code is `gl_Position`.

Change window name

```
/* TODO#0: Change window title to "HW2 - [your student id]"
 *           Ex. HW2 - 312550000
 */
// glfw window creation
GLFWwindow *window = glfwCreateWindow(SCR_WIDTH, SCR_HEIGHT, "HW2 - [your student id]", NULL, NULL);
if (window == NULL) {
```

Remember to include your student ID in glfwCreateWindow(), or you will receive a 3% penalty.

Homework 2 - score

1. createShader (5%)
2. createProgram (5%)
3. modelVAO (5%)
4. loadTexture (5%)
5. draw the Column with texture (20%)
6. draw the Fish with texture (20%)
7. vertex shader (5%)
8. fragment shader (5%)
9. keyboard function (R, S, B) (each 4% / total 12%)
10. report (8%)
11. advanced design: More creative, realistic change in vertex or fragment shader (10%)

Homework 2 - report

- Please specify your name and student ID in the report.
- Explain in detail how to use GLSL by taking screenshots.
(first create program ,second create VAO and VBO, third bind together.....etc.)
(You need to write additional explanation. **Don't just paste the code** with comment.)
- Describe the problems you met and how you solved them.
- Explain how to use your design of advanced part (optional).
- You need to involve a **youtube link** of your demo video in your report, and the video should demonstrate the criteria in previous page, including the window name like "HW2-112550000".

Homework 2 - submission

Deadline: **2025/11/19 23:59:59**

10% penalty for each week late

Final score = original score * $(1 - 0.1 \times \text{weeks late})$

Format

HW2_[studentID].zip e.g. [HW2_123456789.zip](#)

```
| - main.cpp  
| - shaders  (directory)  
|   | - fragmentShader.frag  
|   | - vertexShader.vert  
| - report.pdf
```

If your uploading format doesn't match our requirement,
there will be penalty to your score. (-5%)

If your uploading files can't run successfully,
there will be penalty to your score. (-10%)

Reference

<https://thebookofshaders.com/glossary/>

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

<https://learnopengl.com/Getting-started/Shaders>

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