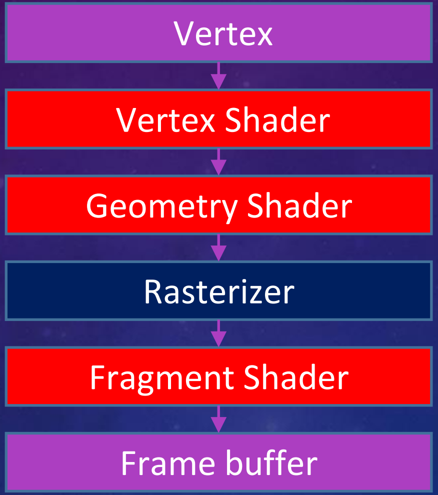
OPENGL SHADER & GLSL

HW2 TUTORIAL

OPENGL PIPELINE



SHADER

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

Vertex Shader

- **Input:** vertex, matrices
- **Output:** vertex

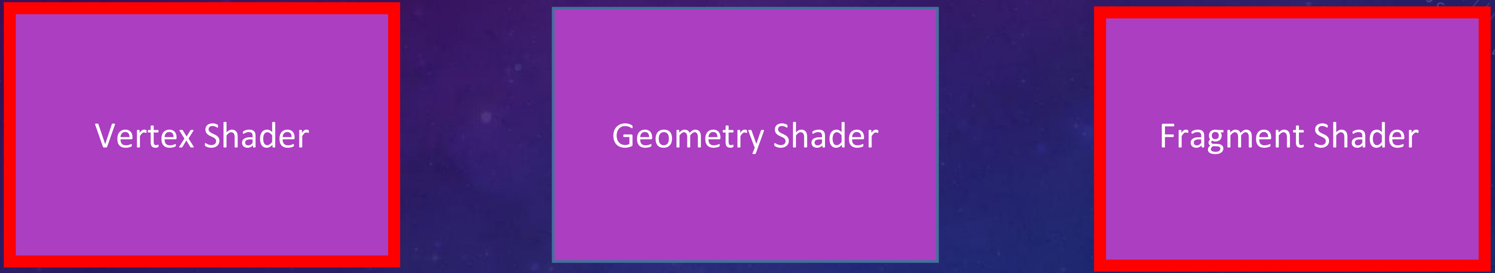
Geometry Shader

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

Fragment Shader

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

SHADER


Vertex Shader

Geometry Shader

Fragment Shader

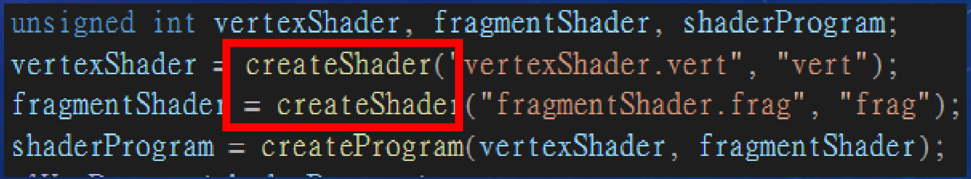
- **Input:** vertex, matrices
- **Output:** vertex

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

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

SHADER SETTING

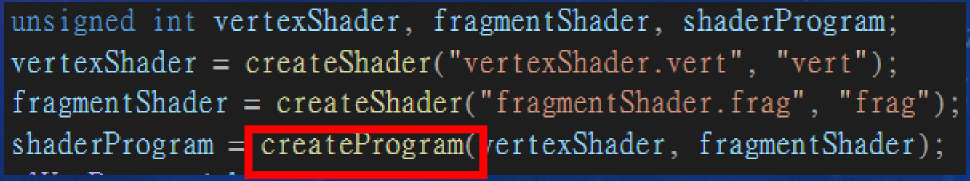
- In the function : createShader()
  - GLuint **glCreateShader** (GLenum shaderType);
    - Specifies the type of shader to be created and creates an empty shader object.
    - shaderType : GL_COMPUTE_SHADER, GL_VERTEX_SHADER, GL_TESS_CONTROL_SHADER, GL_TESS_EVALUATION_SHADER, GL_GEOMETRY_SHADER, GL_FRAGMENT_SHADER
  - 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.



```
unsigned int vertexShader, fragmentShader, shaderProgram;
vertexShader = createShader("vertexShader.vert", "vert");
fragmentShader = createShader("fragmentShader.frag", "frag");
shaderProgram = createProgram(vertexShader, fragmentShader);
```

SHADER SETTING

- 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.



```
unsigned int vertexShader, fragmentShader, shaderProgram;
vertexShader = createShader("vertexShader.vert", "vert");
fragmentShader = createShader("fragmentShader.frag", "frag");
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.

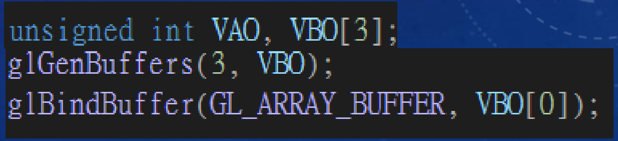
  buffers : Specifies an array in which the generated buffer object names are stored.

- 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.

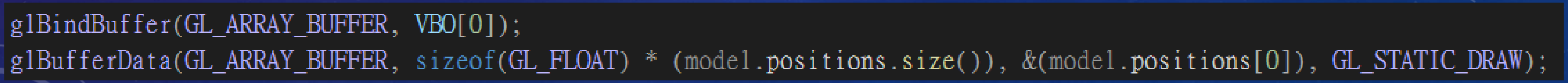
void **glBufferData** (GLenum target, GLsizeiptr size, const GLvoid * data, GLenum usage);

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.

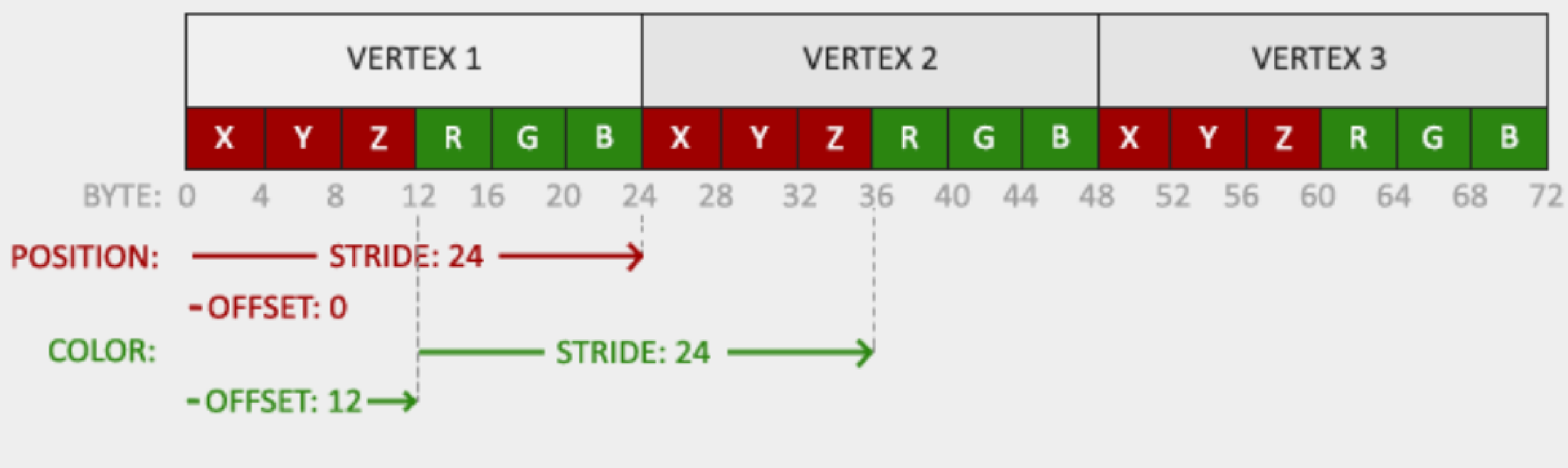


```
glBindBuffer(GL_ARRAY_BUFFER, VBO[0]);
glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT) * (model.positions.size()), &(model.positions[0]), GL_STATIC_DRAW);
```

IMPLEMENTATION IN OPENGL

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

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);
```

VERTEX BUFFER OBJECTS (VBO)



VERTEX 1						VERTEX 2						VERTEX 3					
X	Y	Z	R	G	B	X	Y	Z	R	G	B	X	Y	Z	R	G	B

BYTE: 0 4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64 68 72

POSITION: STRIDE: 24

-OFFSET: 0

COLOR: STRIDE: 24

-OFFSET: 12

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.

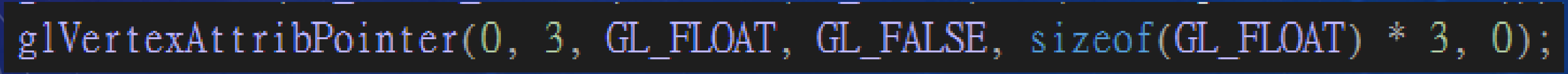
size : Specifies the number of components per generic vertex attribute.

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),
(void*)(offsetof(VertexAttribute, position)));
```

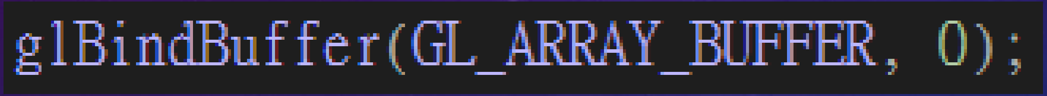
OpenGL

```
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)

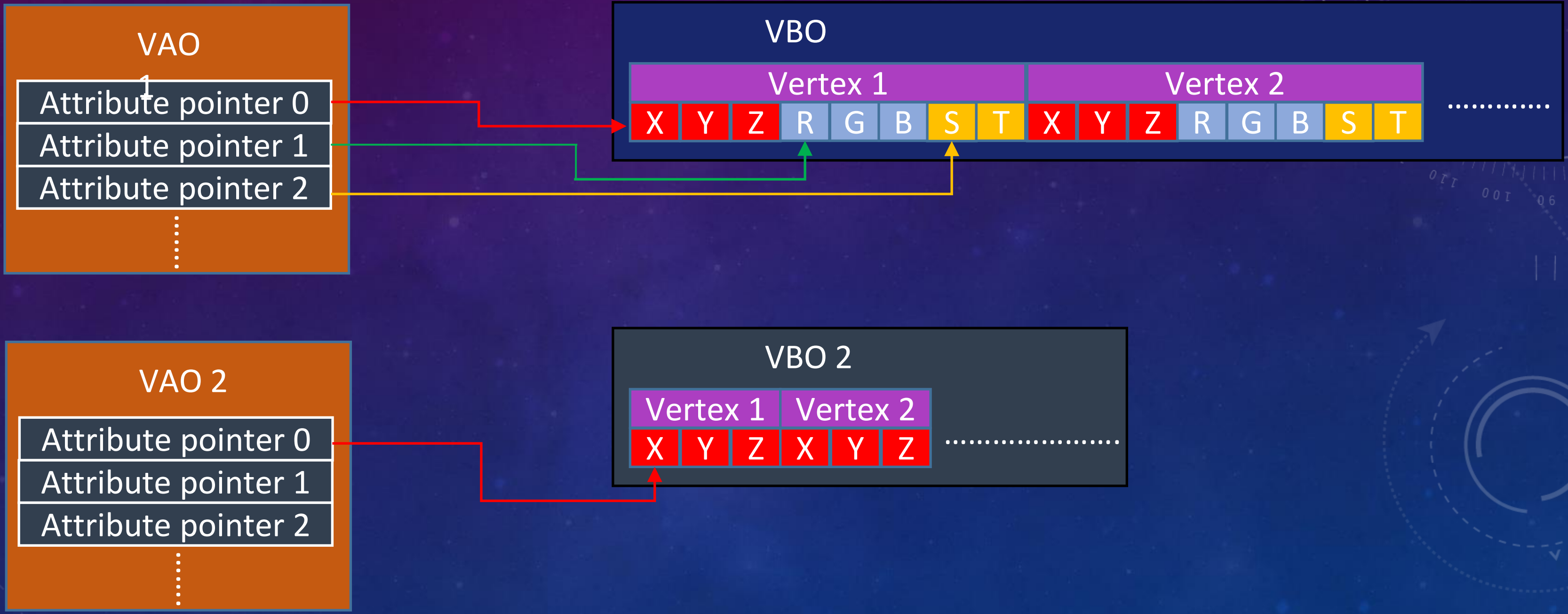
- If you want to render more than one objects, you have to repeat above steps (slides 8 ~14).

  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 VertexAttributePointer. 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);

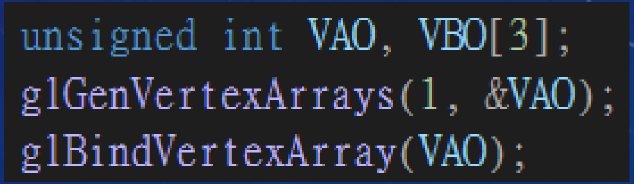
n : Specifies the number of vertex array object names to be generated.

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);
```

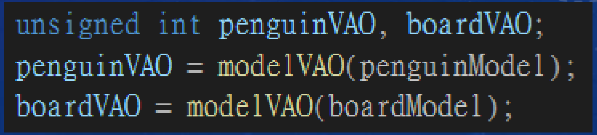
VERTEX ARRAY OBJECT (VAO)

- 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.



```
unsigned int penguinVAO, boardVAO;
penguinVAO = modelVAO(penguinModel);
boardVAO = modelVAO(boardModel);
```

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);

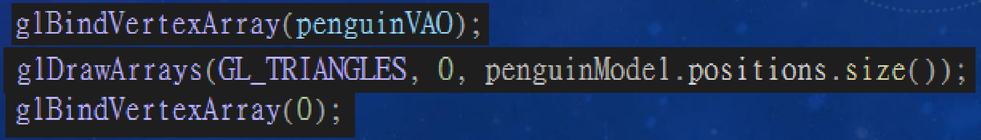
mode : Specifies what kind of primitives to render. Ex: GL_POINTS, GL_LINES, GL_TRIANGLE_STRIP……

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(penguinVAO);
glDrawArrays(GL_TRIANGLES, 0, penguinModel.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)

VERTEX SHADER

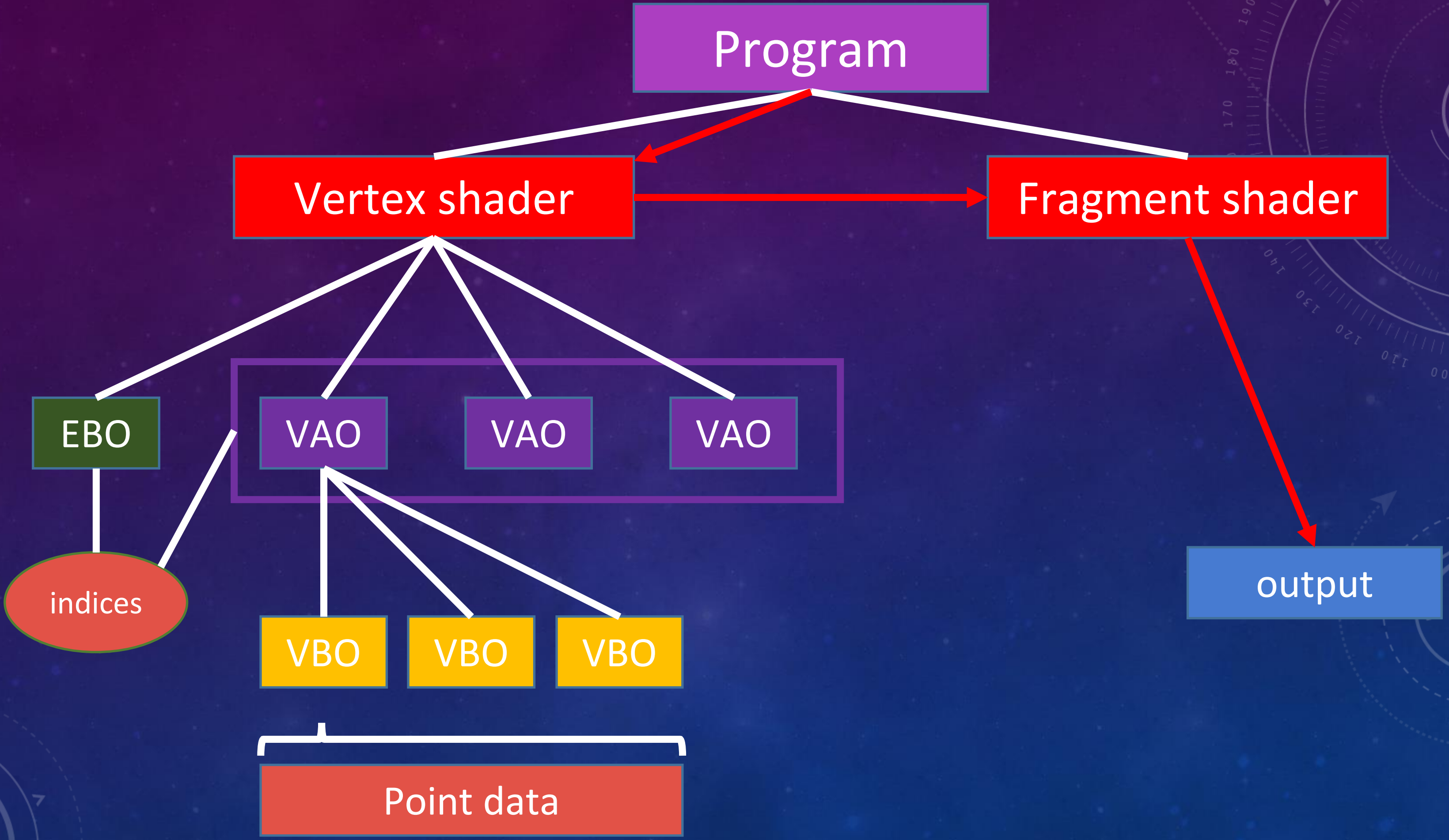
- must have gl_Position

```
/* Example of vertex shader */
#version 330

layout(location = 0) in vec3 position;

uniform mat4 Projection;
uniform mat4 View;

out vec3 color; //to fragment shader

void main() {
  gl_Position = Projection * View * vec4(position, 1.0);
  color = vec3(1.0, 0.0, 0.0);
}
```

FRAGMENT SHADER

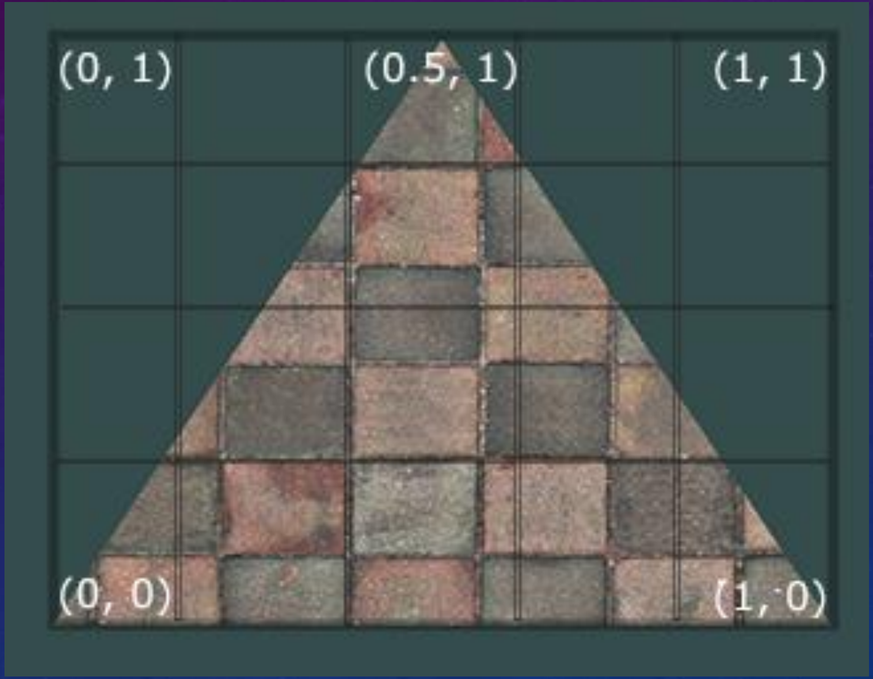
- must have a out vec4 for color buffer

```
/* Example of fragment shader */
#version 330

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

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


Program
Vertex shader
Fragment shader
EBO
VAO
VAO
VAO
indices
VBO
VBO
VBO
output
Point data

TEXTURE IN OPENGL

TEXTURE COORDINATE



HOW TO LOAD AND BIND A TEXTURE

- void glEnable(Glenum cap);

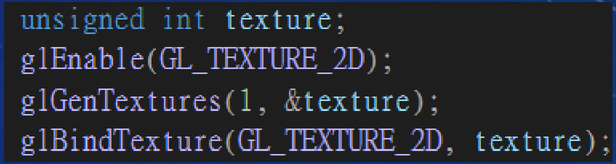
  Use GL_TEXTURE_2D to enable texture

- 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);

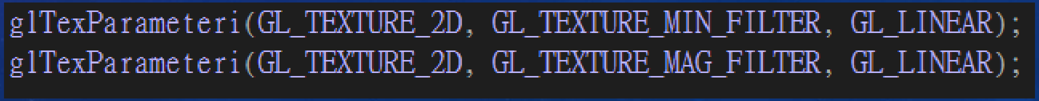
  Bind a named texture to a texturing target (Ex.GL_TEXTURE_1D, GL_TEXTURE_2D, GL_TEXTURE_3D, GL_TEXTURE_1D_ARRAY)



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

HOW TO LOAD AND BIND A TEXTURE

- void 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);



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

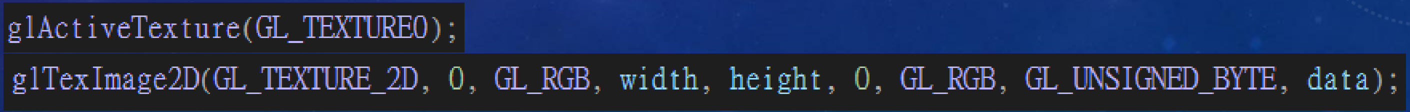
WHEN RENDERING

- void glActiveTexture(GLenum textureUnit);

  selects which texture unit subsequent texture state calls will affect. You can using the textureUnit from GL_TEXTURE0 to GL_TEXTUREn , 0 <= n < 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



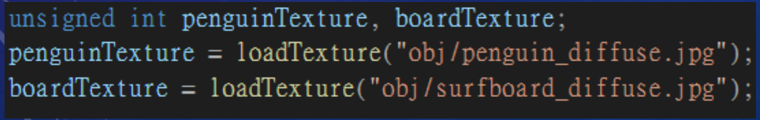
```
glActiveTexture(GL_TEXTURE0);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
```

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);


/* load texture image as data*/

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

LoadTexture() function

Different : No need to unbind texture object



```
unsigned int penguinTexture, boardTexture;
penguinTexture = loadTexture("obj/penguin_diffuse.jpg");
boardTexture = loadTexture("obj/surfboard_diffuse.jpg");
```

```
glUseProgram(program);

glGetUniformLocation(program, "Texture");

glUniform1i(texLoc, 0);

/* draw objects */

glUseProgram(0);
```

OpenGL main loop

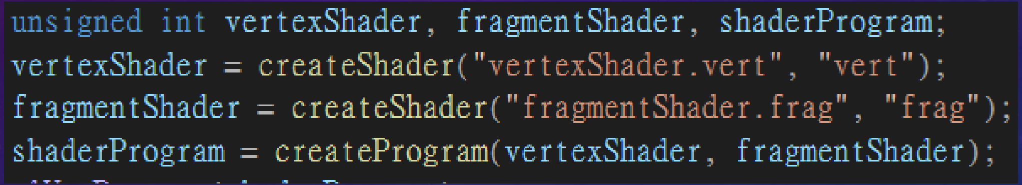
```
uniform sampler2D Texture;

in vec2 texcoord;

out vec4 outColor;

void main() { outColor = texture2D(Texture, texcoord); }
```

GLSL (fragment shader)

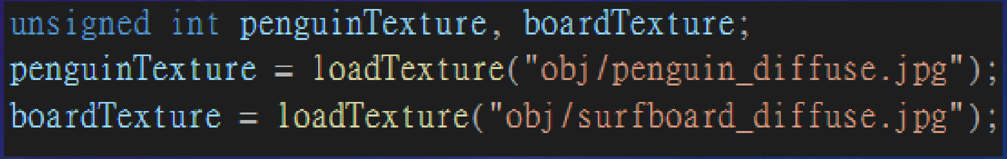
FUNCTIONAL PROGRAMMING(OPTIONAL)

- You are encouraged to complete TODO#1~3 by finishing the functions createShader, createProgram, loadTexture, and modelVAO, but it's not mandatory.

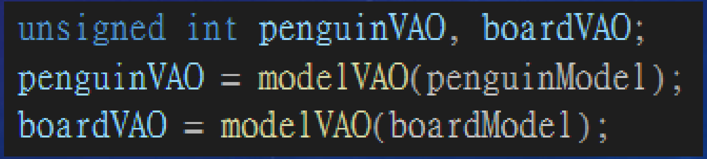
- In other words, you can complete TODO#1~3 without using these four functions.



```
unsigned int vertexShader, fragmentShader, shaderProgram;
vertexShader = createShader("vertexShader.vert", "vert");
fragmentShader = createShader("fragmentShader.frag", "frag");
shaderProgram = createProgram(vertexShader, fragmentShader);
```

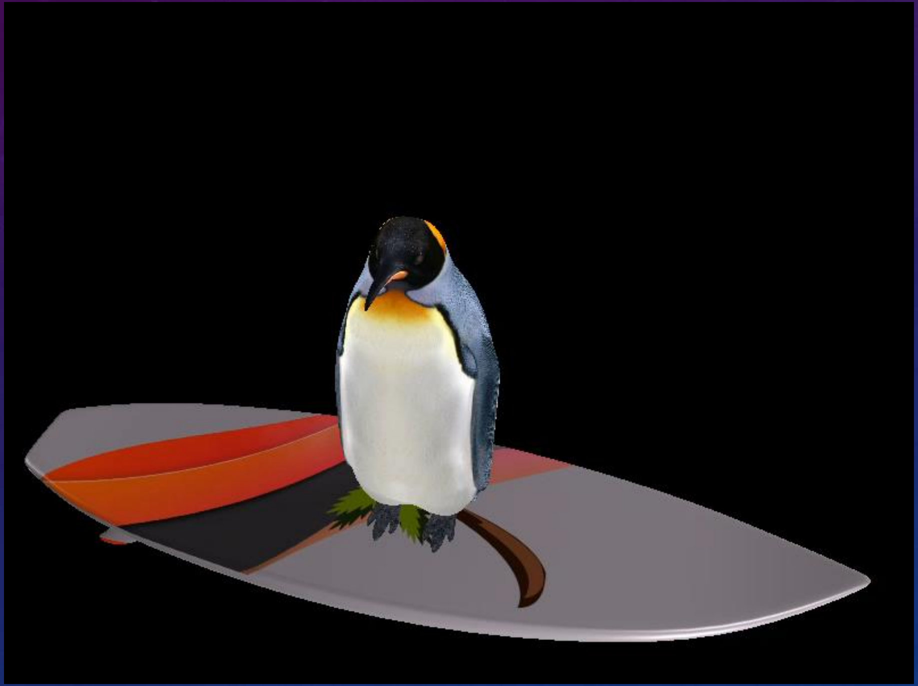


```
unsigned int penguinTexture, boardTexture;
penguinTexture = loadTexture("obj/penguin_diffuse.jpg");
boardTexture = loadTexture("obj/surfboard_diffuse.jpg");
```



```
unsigned int penguinVAO, boardVAO;
penguinVAO = modelVAO(penguinModel);
boardVAO = modelVAO(boardModel);
```

HOMEWORK 2 - PENGUIN SURFING



HOMEWORK 2 - SPEC

- Goal :
  - Using GLSL to draw two model with its texture simultaneously
  - Calulating the texture coordinate of sphere
- spec:
  - Build the model matrix in the following order.
  - Board :
    - position (0, -0.5, 0);
    - rotate -90 degree around X axis
    - scale (0.03, 0.03, 0.03)
  - Penguin :
    - position (0,0,0);
    - rotate -90 degree around X axis
    - scale (0.025, 0.025, 0.025)
  - keyboard function :
    - press key ‘s’ to start/stop penguin squeezing.
    - press key ‘g’ to enable/disable grayscale on the model.

HOMEWORK 2 - SPEC

- Penguin Surfing:
  - The board will rotate between +20 ~ -20 degrees around Y axis. Rotate speed: 20 degrees/sec
  - The board and penguin will move between 0 and 2 on the Z axis. Move speed: 1/sec
- Squeezing:
  - For vertex(x,y,z),

```
y += z * sin(squeezeFactor) / 2;
z += y * sin(squeezeFactor) / 2;
```

- When squeezing, squeezeFactor +90 degree/sec

RESTRICTIONS !!

- Change your window name to “HW2- <yourstudentID>”(TODO#0)
- Your GLSL version should = #version 330
- Deprecated shader syntaxes are not allowed, e.g. attribute, varying
- You are only allowed to use VBO and/or 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.

HOMEWORK 2 - SCORE

1. createShader, createProgram (5%)
2. Setup VAO, Setup VBO of vertex positions, and texcoords (10%)
3. draw the penguin with texture (10%)
4. draw the board with texture (10%)
5. penguin surfing (5%)
6. vertex shader (10%)
7. fragment shader (10%)
8. keyboard function (10%)
9. report (20%)
10. creativity (10%)

CREATIVITY

- New features can only be implemented by adding new key events.
- You can get 5 points or more if you implement additional vertex-dependent special effects (e.g. special vertex-dependent deformation).
- The better you do, the higher score you get! (max 10 points)

HOMEWORK 2 (REPORT)

- Your file name must be in the following format.
- HW2_report_<yourstudentID>.pdf
- 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.

HOMEWORK 2 (繳交規則)

1. DeadLine: 2023/ 11 / 27 23: 59:59

1. Penalty of 10% of the value of the assignment per late week.

If you submit your homework late, the score will be discounted.

Final score = original score * 0.9 for less than a week late

Final score = original score * 0.8 for less than two week late

and so on…

UPLOAD FORMAT

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

1. Please hand in the whole project file and report (.pdf) as HW2_<yourstudentID>.zip to e3 platform.

   e.g. HW2_0716XXX.zip

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)