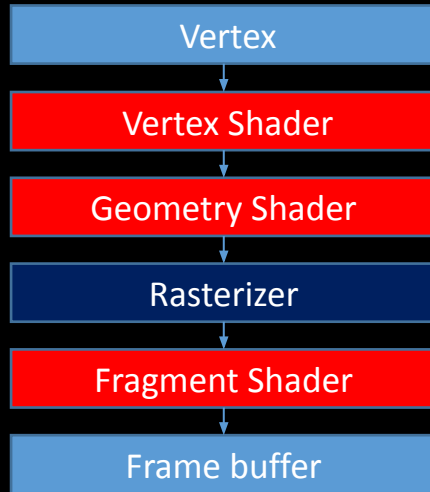


OpenGL shader & GLSL

HW2 Tutorial

OpenGL pipeline



Shader

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

Vertex Shader

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

Geometry Shader

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

Fragment Shader

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

Shader

A light blue rectangular box with a red border, containing the text "Vertex Shader".

Vertex Shader

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

A light blue rectangular box with a black border, containing the text "Geometry Shader".

Geometry Shader

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

A light blue rectangular box with a red border, containing the text "Fragment Shader".

Fragment Shader

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

Shader setting

- 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`.

```
void shaderInit() {  
    GLuint vert = createShader("Shaders/vertexShader.vert", "vertex");  
    GLuint frag = createShader("Shaders/fragmentShader.frag", "fragment");  
    program = createProgram(vert, frag);  
}
```

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.

```
void shaderInit() {  
    GLuint vert = createShader("Shaders/vertexShader.vert", "vertex");  
    GLuint frag = createShader("Shaders/fragmentShader.frag", "fragment");  
    program = createProgram(vert, frag);  
}
```

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.

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


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.

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

Implementation in OpenGL

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

```
VertexAttribute *vertices;
```

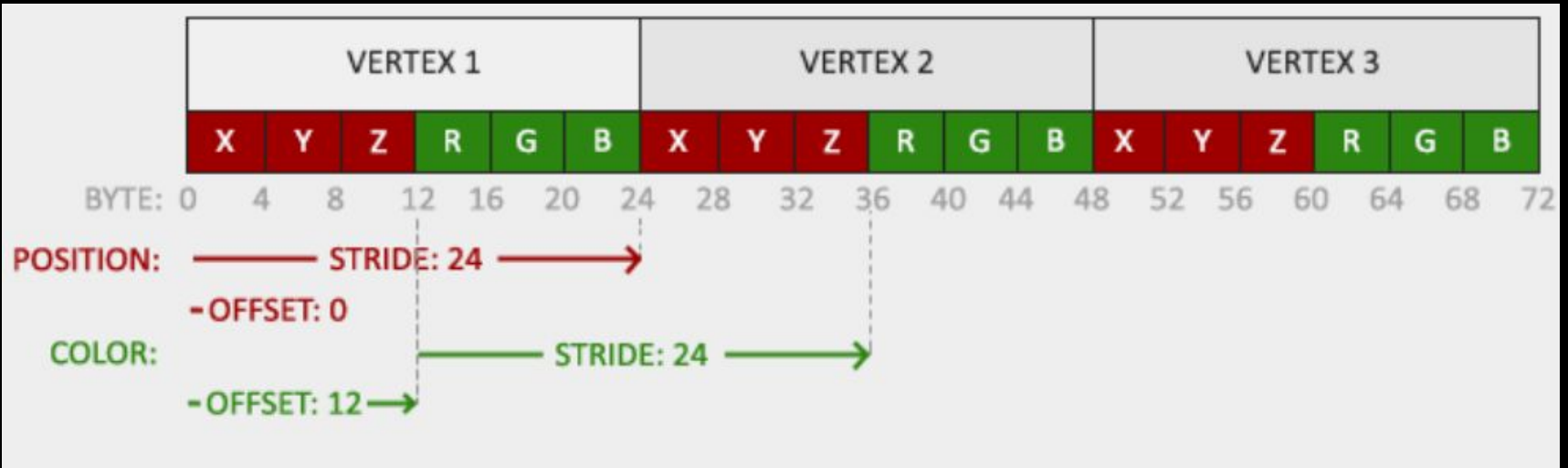
```
GLuint vboName;
```

```
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.

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

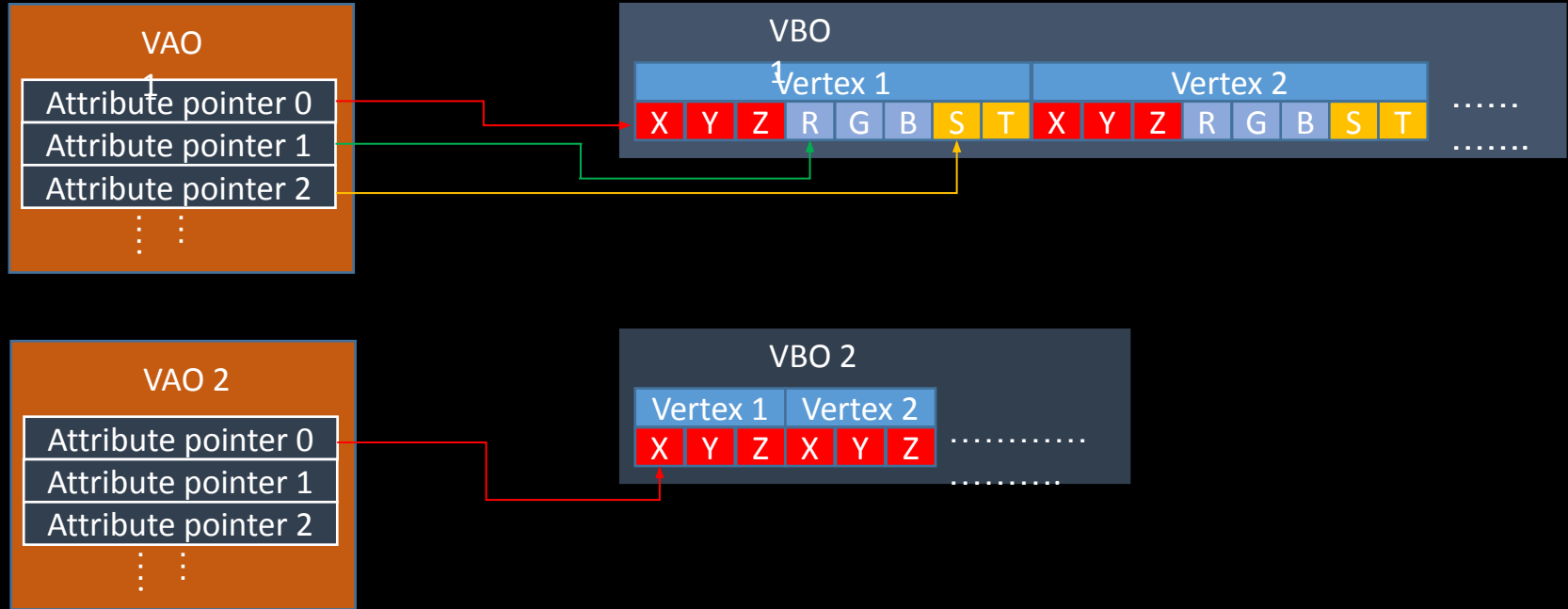
- Use `glBindBuffer()` with the buffer set to zero to unbind the target buffer.

```
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.

```
GLuint VAO;  
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.

Data Connection - Uniform

```
GLfloat pmtx[16];  
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)

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);
```

```
}
```

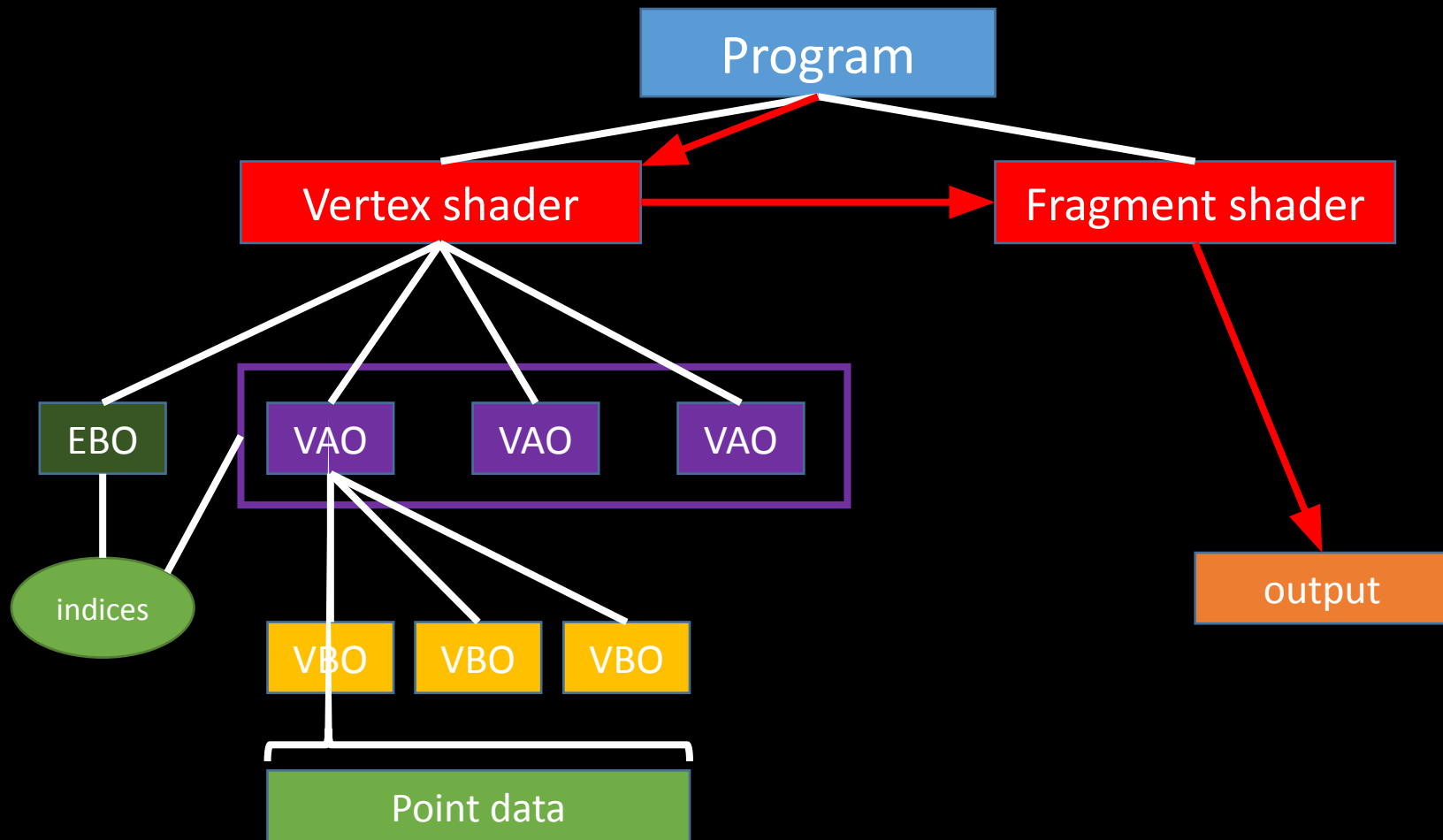
Fragment Shader

- must have a out vec4 for color buffer

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

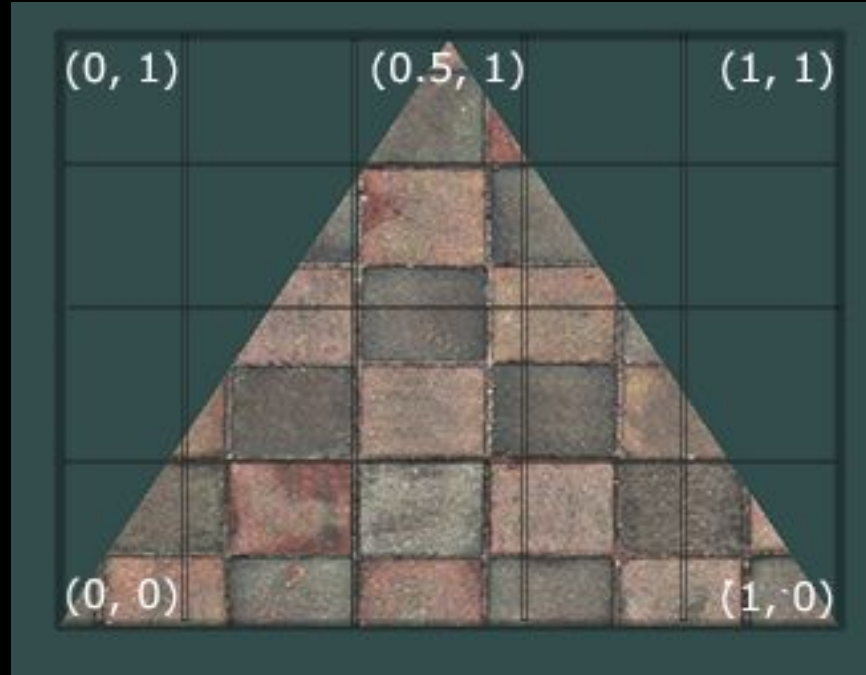
```
in vec3 color; //from vertex shader  
out vec4 frag_color;
```

```
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
- Use FreeImage library to **load** and **free** texture memory
- `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 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`)
- `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
- `void glUniform1i(GLint location, GLint v0);`
Pass Texture to shader sampler variable. **v0** is the number of texture. (Ex. The **v0** of `GL_TEXTURE1` is **1**)

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);`

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

```
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); }
```

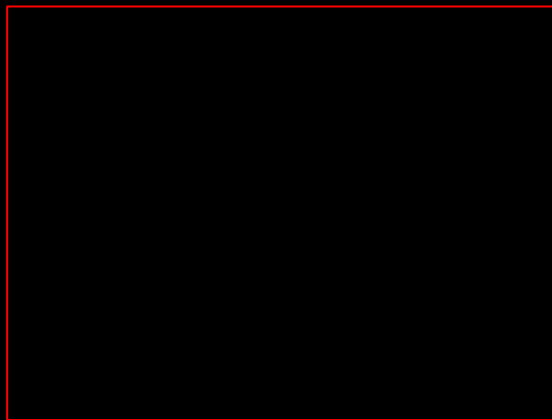
GLSL (fragment shader)

Homework 2 - Pokemon



Homework 2 - spec

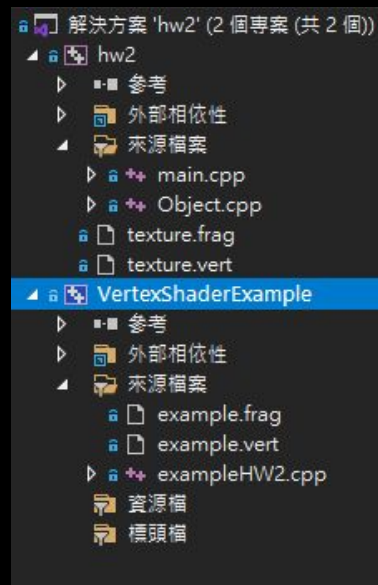
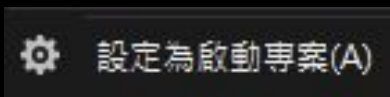
- Goal :
 - Using GLSL to draw two model with its texture simultaneously
 - Calculating the texture coordinate of sphere
- spec:
 - Pikachu :
 - position (0,0,0);
 - scale 5 times
 - rotate 25 degree around Y axis
 - Pokeball :
 - position (3,0,-3);
 - rotate 45 degree around Y axis
 - keyboard function :
 - press key 's' to start/stop all models rotation around y axis. (1 degree / per frame)



Homework 2 - tool

- Using **Shader.h** to create shader simply
- Some parameters will be used (defined in **Object.h**) for Pikachu model
 - Position : model->positions
 - Texcoords : model->texcoords
- Class **Vertex** & **VertexAttribute** (defined in **Vertex.h**) may help you to buffer data
- Feeling free to modify these header file above if you need.
(Of course you can finish this homework only using these header file without modification.)
- The simple shader sample code is contained in “**VertexShaderExample1**” project.
 1. right click this project
 2. select “Set as Startup Project”

```
vector<float> positions;  
vector<float> texcoords;
```



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/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 pokeball with texture (20%)
4. draw the Pikachu with texture (20%)
5. vertex shader (10%)
6. fragment shader (10%)
7. keyboard function (5%)
8. report (20%)

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.

Homework 2 (繳交規則)

1. DeadLine: 2020/ 12 / 8 23: 59:59
2. Penalty of 10% of the value of the assignment per late week.

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

submit between (12/9 - 12/15) : Your final score * 0.9

submit between (12/16 - 12/22) : Your final score * 0.8

submit after 12/23 : Your final score * 0.7

Upload Format

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

2. Please hand in the whole **project file** and **report** (.pdf) as **HW1_<yourstudentID>.zip** to e3 platform.

e.g. HW2_0716XXX.zip

#you **must check that you can run project** directly after unzipping the submission or you will get **5% penalty**

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))

