Implementation: Simple Drawing

Introduction to Computer Graphics

Yu-Ting Wu

Library

Library

- **GLEW: The OpenGL Extension Wrangler Library (link)**
  - A cross-platform open-source C/C++ extension loading library
  - Provide efficient run-time mechanisms for determining which OpenGL extensions are supported on the target platform

- **GLM: OpenGL Mathematics (link)**
  - A **header-only** C++ mathematics library for graphics software based on the **OpenGL Shading Language (GLSL) specifications**

Program Overview

Goals

- Draw a point
- Draw a circle (ellipse)
- Draw a triangle

Draw a Single Point

```
int main(int argc, char** argv)
{
    // Setting window properties.
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_DEPTH);
    glutInitWindowSize(640, 360);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("OpenGL Renderer");

    // Initialize GLEW.
    // Must be done after glut is initialized!
    GLenum res = glewInit();
    if (res != GLEW_OK) {
        std::cerr << "GLEW initialization error: "
                  << glewGetErrorString(res) << std::endl;
        return 1;
    }

    // Initialization.
    SetupRenderState();
    SetupScene();

    // Register callback functions.
    glutDisplayFunc(RenderSceneCB);
```

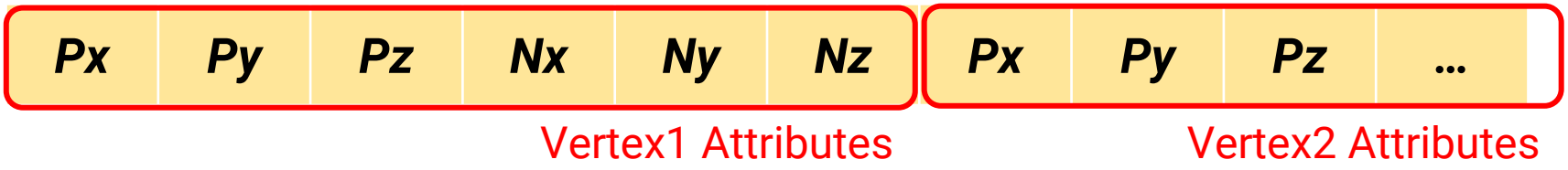
```
// OpenGL and FreeGlut headers.
#include <glew.h>
#include <freeglut.h>
```

Draw a Single Point (cont.)

```
// Global variables.
GLuint vbo;

void SetupScene()
{
    // Draw a single point.
    float VertexPosition[3] = {0.0f, 0.0f, 0.0f};
    // Generate the vertex buffer.
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(VertexPosition), VertexPosition, GL_STATIC_DRAW);
}

void RenderSceneCB()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Render a point on screen.
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glDrawArrays(GL_POINTS, 0, 1);
    glDisableVertexAttribArray(0);

    glutSwapBuffers();
}
```

Vertex Buffer

- A buffer storing the **vertex attribute data**
- Possible vertex attributes include (but are not limited to)
  - Vertex position
  - Vertex normal
  - Texture coordinate
  - Tangent
- Will be passed to GPU for rendering



Vertex Buffer

- **Generate a buffer**
  - *void **glGenBuffers**(GLsizei n, GLuint * buffers);*
- **Upload data into the buffer**
  - *void **glBindBuffer**(GLenum target, GLuint buffer);* [Link]
  - *void **glBufferData**(* [Link]

    GLenum target, GLsizeiptr size,

    *const void * data, GLenum usage);*

```
// Generate the vertex buffer.
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(VertexPosition), VertexPosition, GL_STATIC_DRAW);
```

Vertex Buffer (cont.)

- **Render with the vertex buffer**
  - *void **glEnableVertexAttribArray**(GLuint index);*
  - *void **glVertexAttribPointer**(*
    GLuint index ,
    GLint size ,
    GLenum type ,
    GLboolean normalized ,
    GLsizei stride ,
    *const void * pointer*
    );

The index of the attribute
E.g., 0 for position, 1 for normal, etc.

Number of components of the attribute

Type of the attribute component

The byte offset to the same attribute of the next vertex

The byte offset of the first component

Vertex Buffer (cont.)

- *void **glDrawArrays**(*
  GLenum mode,
  GLint first,
  GLsizei count
  );

The type of the primitive
E.g., GL_POINTS, GL_LINE_LOOP, GL_TRIANGLES, etc.

The start index

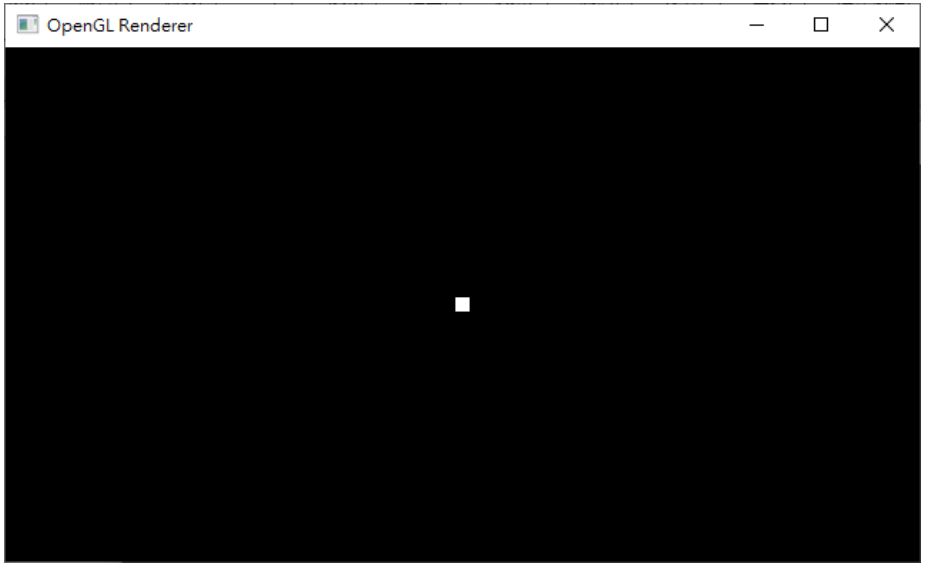
The number of **indices** to be rendered

- *void **glDisableVertexAttribArray**(GLuint index);*

```
// Render a point on screen.
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glDrawArrays(GL_POINTS, 0, 1);
glDisableVertexAttribArray(0);
```

Change the Point Size

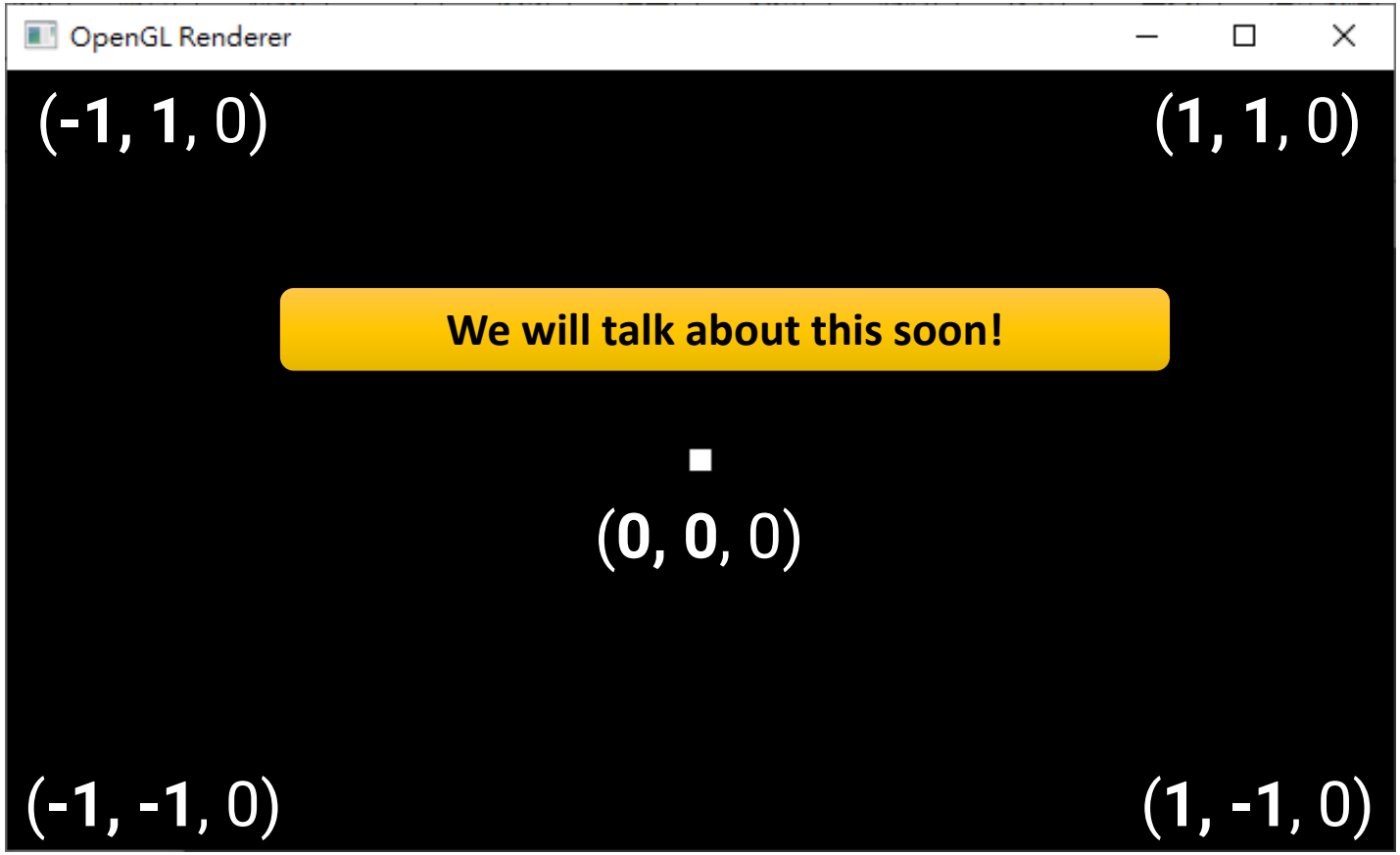
- *void **glPointSize**(GLfloat size)*




```
void SetupRenderState()
{
    // Default.
    glPointSize(1);
}
```

```
void SetupRenderState()
{
    glPointSize(10);
}
```

Insight: Coordinate



What about the *z* coordinate? You can find the point will only be visible if its *z* value is within ***[-1, 1]***

Avoid Deprecated APIs

- Although it seems convenient, do **NOT** use

```
glBegin(GL_POINTS/GL_LINES/GL_TRIANGLES);
        glVertex3f(…);
        glVertex3f(…);
        glVertex3f(…);
glEnd();
```

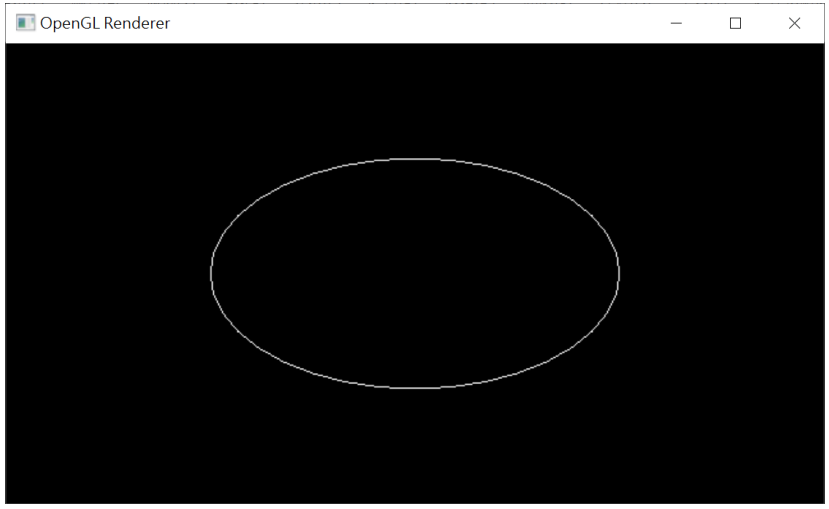
- These APIs have been deprecated since OpenGL 3.2 due to the performance issue

Draw a Circle (Ellipse)

```cpp
// C++ STL headers.
#include <iostream>
#include <vector>
#define _USE_MATH_DEFINES
#include <math.h>

// Global variables.
GLuint vbo;
const int numCircleSamples = 36;

void SetupScene()
{
    // Draw a circle.
    float VertexPosition[numCircleSamples * 3];
    const float thetaOffset = 2.0f * M_PI / (float)numCircleSamples;
    float startTheta = 0.0f;
    float r = 0.5f;
    for (int i = 0; i < numCircleSamples; ++i) {
        float theta = startTheta + i * thetaOffset;
        VertexPosition[3 * i + 0] = r * std::cos(theta);    // x.
        VertexPosition[3 * i + 1] = r * std::sin(theta);    // y.
        VertexPosition[3 * i + 2] = 0.0f;                   // z.
    }

    // Generate the vertex buffer.
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(VertexPosition), VertexPosition, GL_STATIC_DRAW);
}
```

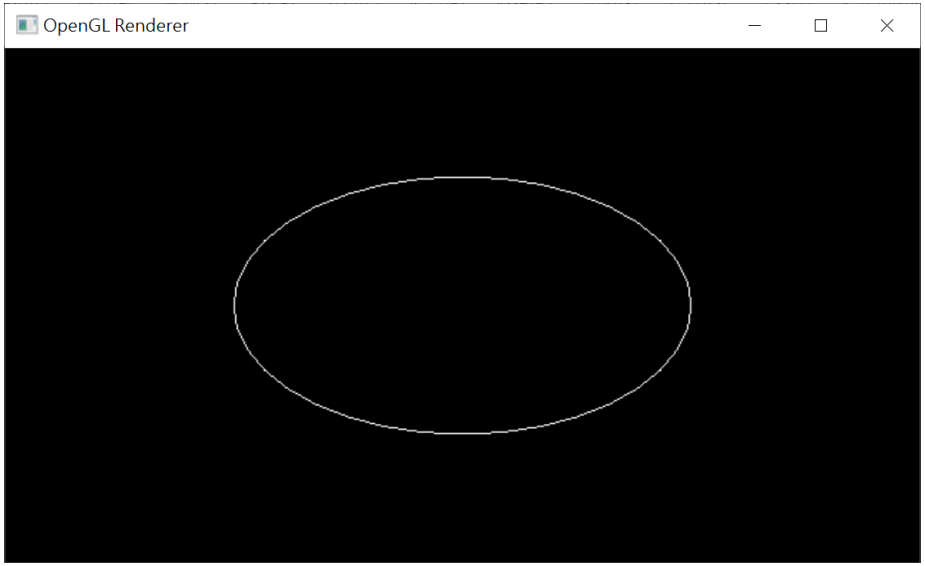
Draw a Circle (Ellipse)

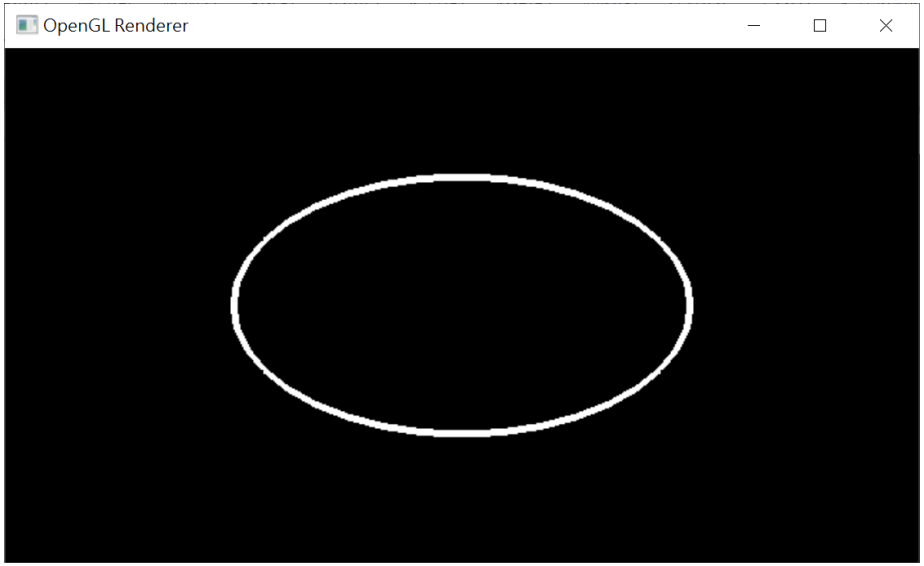
```
void RenderSceneCB()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Render a point on screen.
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glDrawArrays(GL_LINE_LOOP, 0, numCircleSamples);
    glDisableVertexAttribArray(0);

    glutSwapBuffers();
}
```



Change the Line Width

- *void **glLineWidth**(GLfloat width)*






```
void SetupRenderState()
{
    glLineWidth(5);
}
```

The GLM Library

- In computer graphics, we need a data structure to store and manipulate **multi-dimensional data**, such as position, normal, texture coordinate, and color
- The GLM library provides an elegant way to process multi-dimensional data
  - Support **operator overloading**
  - Match the syntax of OpenGL shading language (GLSL)
  - Support **alias** of components
    - For position or normal, we used to use *(x, y, z, w)*
    - For texture coordinate, we used to use *(u, v, s, t)*
    - For color, we used to use *(r, g, b, a)*

The GLM Library Examples

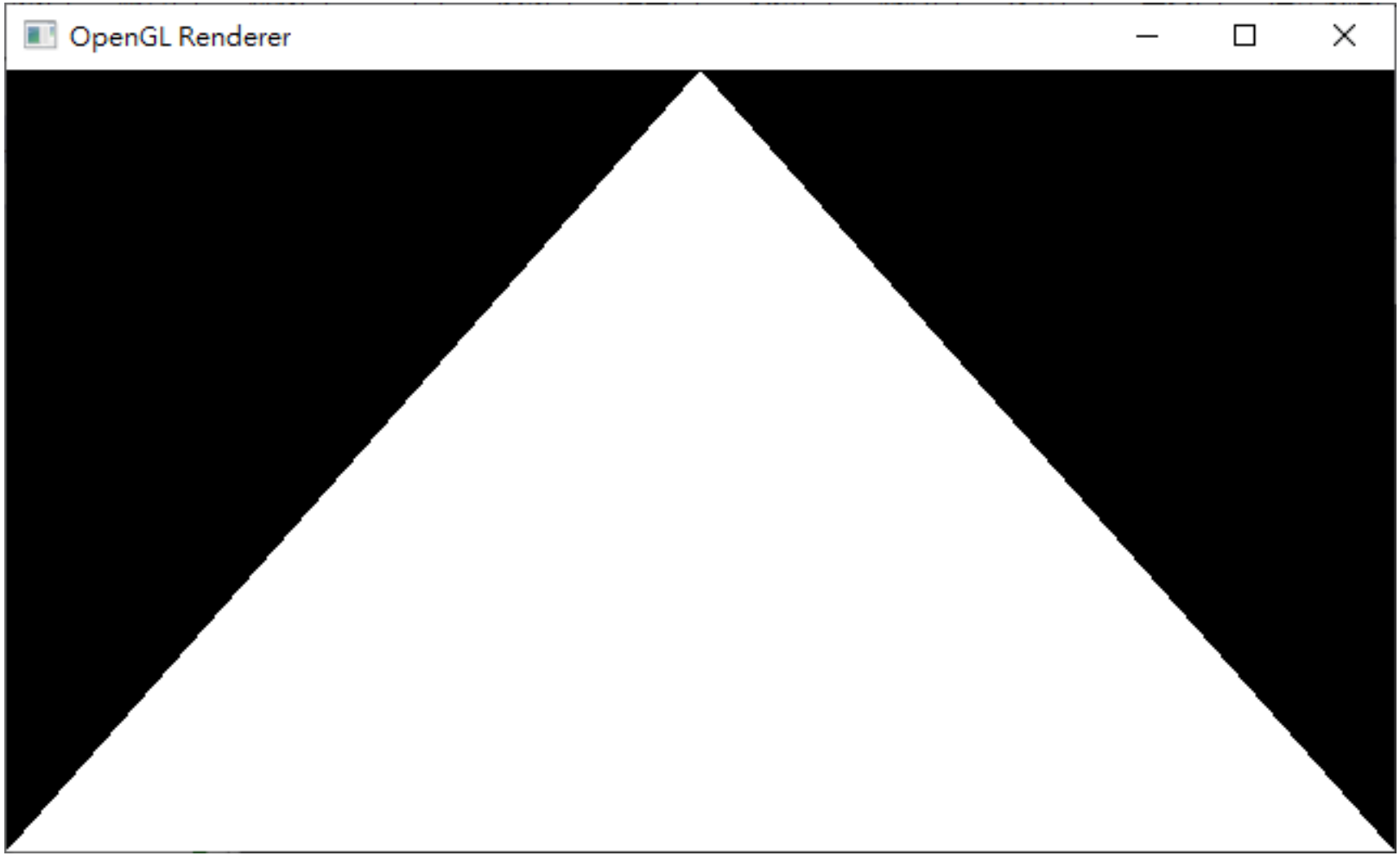
- The most common data types are three/four-dimensional vectors and four-by-four matrices
- Example: compute the average direction of three vectors

```
glm::vec3 dir1 = glm::vec3(1.0f, 0.0f, 0.0f);
glm::vec3 dir2 = glm::vec3(0.0f, 1.0f, 0.0f);
glm::vec3 dir3 = glm::vec3(0.0f, 0.0f, 1.0f);
glm::vec3 avgDir = (dir1 + dir2 + dir3) / 3.0f;
```

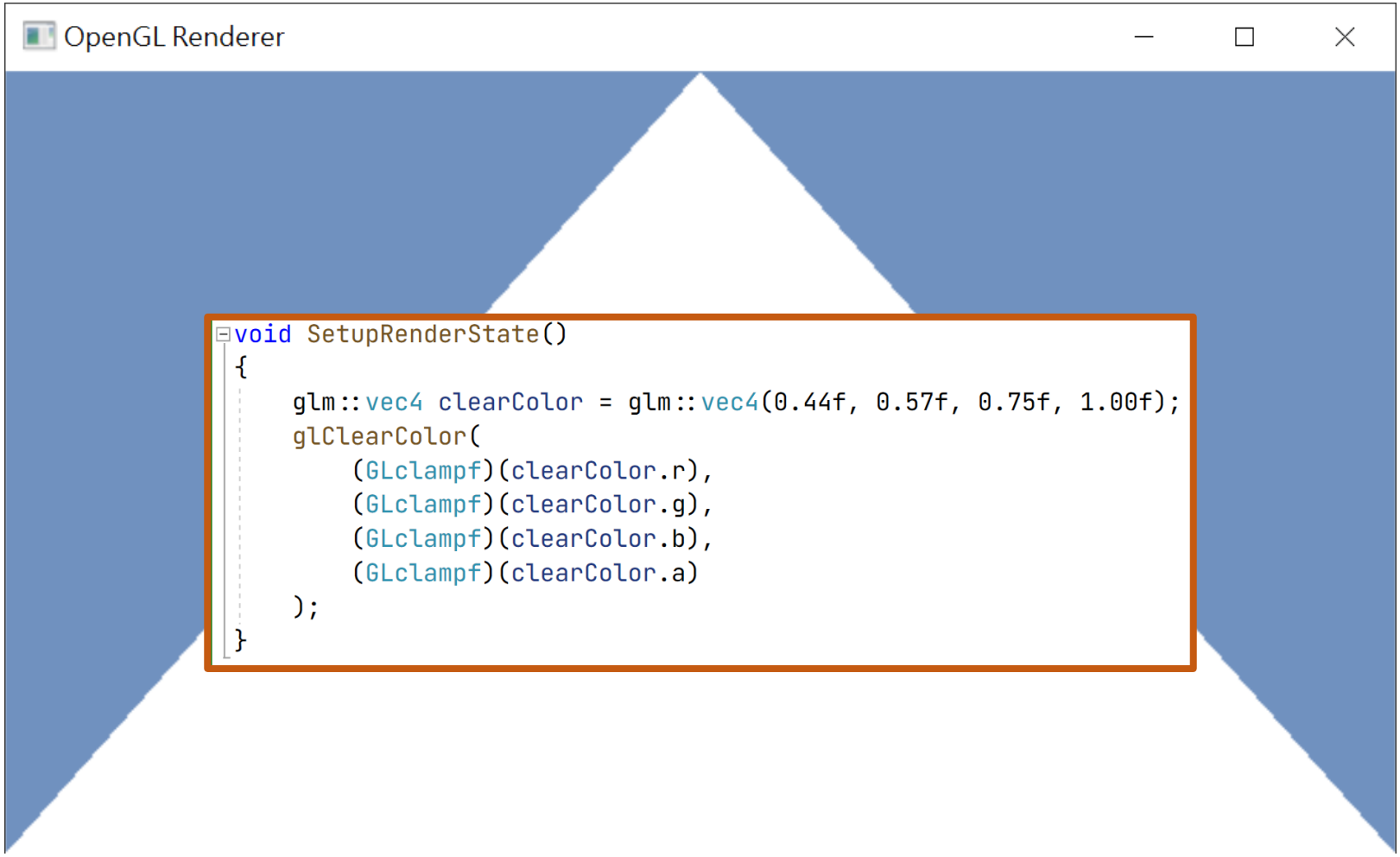
Draw a Triangle

```
void SetupScene()
{
    // Draw a triangle.
    glm::vec3 VertexPosition[3];
    VertexPosition[0] = glm::vec3(-1.0f, -1.0f, 0.0f);
    VertexPosition[1] = glm::vec3( 0.0f,  1.0f, 0.0f);
    VertexPosition[2] = glm::vec3( 1.0f, -1.0f, 0.0f);

    // Generate the vertex buffer.
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(VertexPosition), VertexPosition, GL_STATIC_DRAW);
}
void RenderSceneCB()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Render a point on screen.
    glEnableVertexAttribArray(0);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    glDisableVertexAttribArray(0);

    glutSwapBuffers();
}
```

Draw a Triangle (cont.)

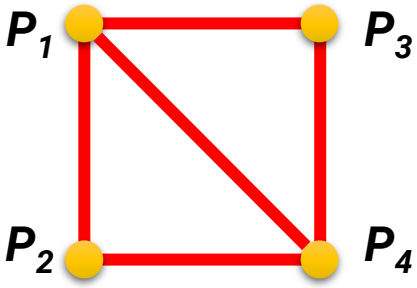


GLM Vector for Representing Color



Index Buffer

- When drawing multiple triangles, lots of the vertices are reused
- We can use an index buffer to identify the vertex defined in the vertex buffer
- Example: a quad with 2 triangles

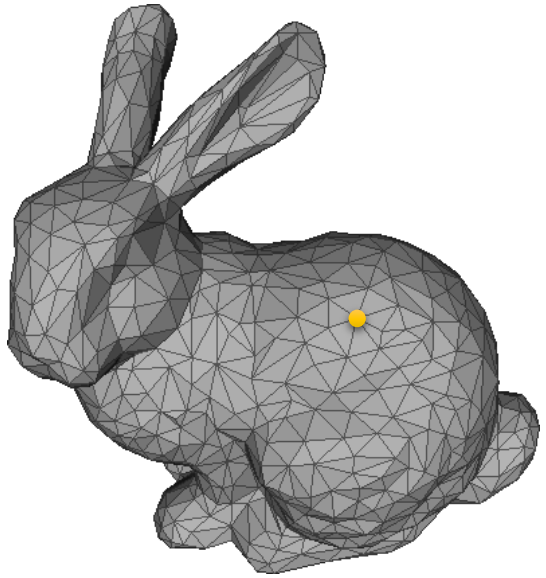


Vertex Buffer

P_1 x	P_1 y	P_1 z	P_2 x	P_2 y	P_2 z	P_3 x	P_3 y	P_3 z	P_4 x	P_4 y	P_4 z

Index Buffer

1	2	4	1	4	3



Index Buffer

- **Generate a buffer and upload data**
  - Use the same functions as we create the vertex buffer, but with different parameters

```
// Draw a quad with indexed triangles.
glm::vec3 vertexPosition[4];
vertexPosition[0] = glm::vec3(-0.8f,  0.8f, 0.0f);
vertexPosition[1] = glm::vec3(-0.8f, -0.8f, 0.0f);
vertexPosition[2] = glm::vec3( 0.8f,  0.8f, 0.0f);
vertexPosition[3] = glm::vec3( 0.8f, -0.8f, 0.0f);
// Generate the vertex buffer.
glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glBufferData(GL_ARRAY_BUFFER, sizeof(vertexPosition), vertexPosition, GL_STATIC_DRAW);

unsigned int vertexIndices[6] = { 0, 1, 3, 0, 3, 2 };
// Generate the index buffer.
glGenBuffers(1, &ibo);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(vertexIndices), vertexIndices, GL_STATIC_DRAW);
```

Index Buffer (cont.)

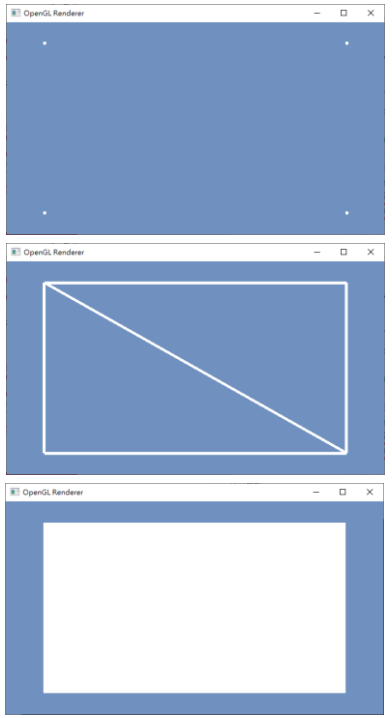
- **Render with the vertex buffer and index buffer**

```
// Render a quad on screen.
glEnableVertexAttribArray(0);
glBindBuffer(GL_ARRAY_BUFFER, vbo);
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, 0);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ibo);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
glDisableVertexAttribArray(0);
```

Change Polygon Render Mode

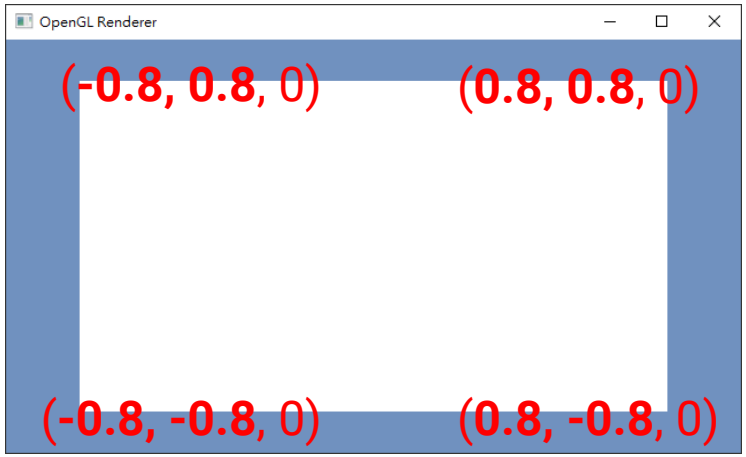
- OpenGL provides API for changing polygon render mode
  - *void **glPolygonMode**(GLenum face, GLenum mode);*

```
void ProcessSpecialKeysCB(int key, int x, int y)
{
    // Handle special (functional) keyboard inputs such as F1, spacebar, page up, etc.
    switch (key) {
    case GLUT_KEY_F1:
        // Render with point mode.
        glPointSize(5);
        glPolygonMode(GL_FRONT_AND_BACK, GL_POINT);
        break;
    case GLUT_KEY_F2:
        // Render with line mode.
        glLineWidth(5);
        glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
        break;
    case GLUT_KEY_F3:
        // Render with fill mode.
        glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
        break;
    default:
        break;
    }
}
```



Where is the Camera and Projection?

- The typical flow of bringing a 3D point to the 2D screen involves the **camera projection**
- For now, we specify neither the camera nor the projection, so you can consider that we set the **“projected”** positions of the vertices directly
- In the next implementation slides, we will go through the full transformation



A rectangle?
Why not a square?

Any Questions?