Section 3 Create a Triangle

The Graphics Pipeline

The graphics pipeline is the sequence of stages that a 3D scene goes through to be rendered on a display. The typical graphics pipeline consists of the following stages:

1. **Application**
2. **Vertex Shader**
3. **Primitive Assembly**
4. **Rasterization**
5. **Fragment Shader**
6. **Depth Test**
7. **Blending**
8. **Output Merger**

Let's go through each stage in detail:

1. Application

The application stage is where the 3D scene is defined. This includes creating the geometry (vertices, edges, and faces), setting up the camera and lighting, and defining the materials and textures. The application stage runs on the CPU and prepares the data to be sent to the graphics processing unit (GPU).

2. Vertex Shader

The vertex shader is the first programmable stage of the graphics pipeline, and it runs on the GPU. Its primary responsibilities are:

- Transforming the vertex positions from model space to screen space
- Calculating the lighting effects (diffuse, specular, ambient) for each vertex
- Mapping the texture coordinates for each vertex

The vertex shader takes the vertex data (position, normal, texture coordinates, etc.) as input and outputs the transformed vertex data, which is then passed to the next stage.

3. Primitive Assembly

In this stage, the transformed vertices from the vertex shader are used to create primitives, such as triangles, lines, or points. These primitives are the basic building blocks of the 3D scene.

4. Rasterization

The rasterization stage converts the 3D primitives into a 2D grid of pixels (fragments) that can be processed by the fragment shader. This process involves determining which pixels are covered by each primitive and calculating their barycentric coordinates, which are used to interpolate values across the primitive.

5. Fragment Shader

The fragment shader is the second programmable stage of the graphics pipeline, and it runs on the GPU. Its primary responsibilities are:

- Calculating the final color of each pixel (fragment) based on factors such as texture mapping, lighting, and material properties
- Performing any additional per-pixel operations, such as effects or post-processing

The fragment shader takes the interpolated values from the rasterization stage as input and outputs the final color for each pixel.

6. Depth Test

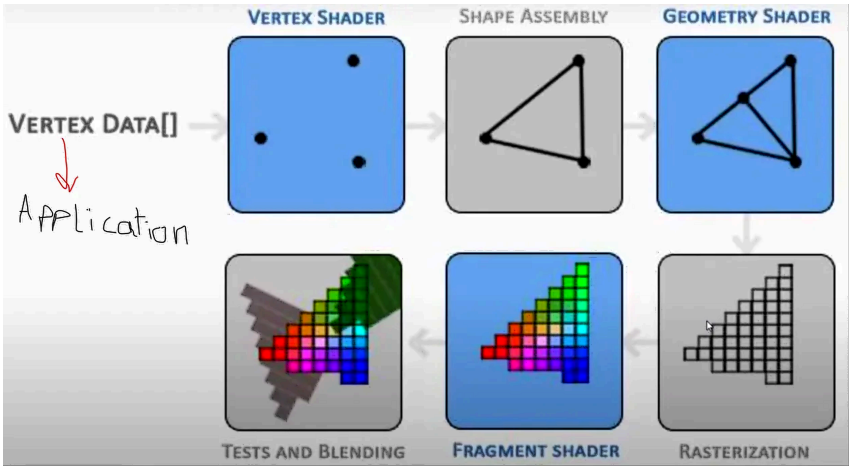
The depth test stage compares the depth (z-coordinate) of the current fragment with the depth of the fragments already stored in the depth buffer. If the current fragment is closer to the camera, it is accepted; otherwise, it is discarded.

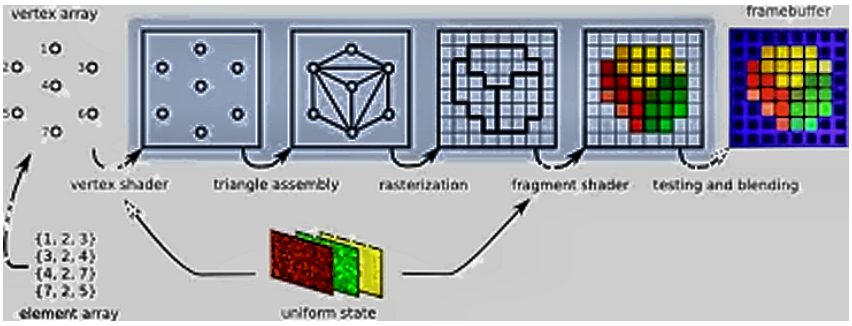
7. Blending

The blending stage combines the color of the current fragment with the color already stored in the color buffer, based on the fragment's alpha (transparency) value. This allows for the rendering of transparent or semi-transparent objects.

8. Output Merger

The final stage of the graphics pipeline is the output merger, which writes the processed fragments to the framebuffer, which is then displayed on the screen.





Summary:

The graphics pipeline is the step-by-step process that a 3D scene goes through to be displayed on your screen. Here's a simplified explanation of each step:

1. **Application**: This is where you, the programmer, create the 3D objects, positions, colors, textures, and other details that make up your scene.
2. **Vertex Shader**: The vertex shader takes the information about each point (vertex) in your 3D objects and calculates how that point should be positioned and lit on the screen.
3. **Primitive Assembly**: The transformed vertex information from the vertex shader is used to create the basic shapes (primitives) that make up your 3D objects, like triangles.
4. **Rasterization**: The 3D primitives are converted into a 2D grid of pixels that can be displayed on your screen.
5. **Fragment Shader**: This shader calculates the final color of each pixel based on factors like textures, lighting, and material properties.
6. **Depth Test**: The depth test compares the distance of each pixel (fragment) from the camera and decides which ones are visible and which are hidden behind other objects.
7. **Blending**: If there are any semi-transparent or blended objects, this stage combines their colors with the colors already in the image.

8. **Output Merger**: The final, processed pixels are sent to the framebuffer, which is what gets displayed on your screen.

Code Implementation

Now that you understand the graphics pipeline, let's focus on drawing a triangle and viewing it using OpenGL. Since OpenGL doesn't provide default vertex and fragment shaders, we need to write our own. However, for now, we'll concentrate on how to use these shaders rather than how to write them.

Step 1: Define our Libraries and Vertex & Fragment shaders

```
#include<iostream>
#include<glad/glad.h>
#include<GlFW/glfw3.h>
using namespace std;
// vertex shader
const char* vertexShaderSource = "#version 330 core\n"
"layout(location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"}\n";
// fragment shader
const char* fragmentShaderSource = "#version 330
core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
"}\n";
```

Step 2 : Prepare our Window

```cpp
#include<iostream>
#include<glad/glad.h>
#include<GlFW/glfw3.h>
using namespace std;
// Vertex Shader source code
const char* vertexShaderSource = "#version 330 core\n"
"layout(location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"}\n";
// Fragment Shader source code
const char* fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
"}\n";
int main()
{
    // Initialize GLFW
    glfwInit();

    // Tell GLFW what version of OpenGL we are using
    // In this case we are using OpenGL 3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    // Tell GLFW we are using the CORE profile
    // So that means we only have the modern functions
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    // Create a GLFWwindow object of 800 by 800 pixels, naming it "Triangle"
    GLFWwindow* window = glfwCreateWindow(800, 800, "Triangle", NULL, NULL);
    // Error check if the window fails to create
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    // Introduce the window into the current context
    glfwMakeContextCurrent(window);

    //Load GLAD so it configures OpenGL
    gladLoadGL();
    // Specify the viewport of OpenGL in the Window
    // In this case the viewport goes from x = 0, y = 0, to x = 800, y = 800
    glViewport(0, 0, 800, 800);


}
```

Step 3 Create our Triangle vertices

We need to specify the coordinates of our triangle. To do this, we will use `GLfloat vertices[]`, which defines an array of floating-point numbers.

> ⛔ Note:
> The origin point (0,0) of the vertice is at the center of the window. Be careful not to confuse this with the origin point of the window, which may be different.
> The x, y, and z axes are normalized, meaning the x-axis ranges from -1 on the left to +1 on the right, with y and z following the same scale.

in our example since we want a 2D Triangle we will set z-axis to 0.0f

```cpp
#include<iostream>
#include<glad/glad.h>
#include<GlFW/glfw3.h>
using namespace std;
// Vertex Shader source code
const char* vertexShaderSource = "#version 330 core\n"
"layout(location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"}\n";
// Fragment Shader source code
const char* fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
"}\n";
int main()
{
    // Initialize GLFW
    glfwInit();

    // Tell GLFW what version of OpenGL we are using
    // In this case we are using OpenGL 3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    // Tell GLFW we are using the CORE profile
    // So that means we only have the modern functions
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    // Create a GLFWwindow object of 800 by 800 pixels, naming it "Triangle"
    GLFWwindow* window = glfwCreateWindow(800, 800, "Triangle", NULL, NULL);
    // Error check if the window fails to create
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    // Introduce the window into the current context
    glfwMakeContextCurrent(window);

    //Load GLAD so it configures OpenGL
    gladLoadGL();
    // Specify the viewport of OpenGL in the Window
    // In this case the viewport goes from x = 0, y = 0, to x = 800, y = 800
    glViewport(0, 0, 800, 800);

    // HERE IS STEP 3
    //////////////////////////////////////////////////////////////////////////
    // Vertices coordinates
    GLfloat vertices[] =
    {
        -0.5f, -0.5f , 0.0f, // Lower left corner x, y, z
        0.5f, -0.5f , 0.0f, // Lower right corner x, y, z
        0.0f, 0.5f , 0.0f // Upper corner x, y, z
    };
    //////////////////////////////////////////////////////////////////////////

}
```

Step 4: Vertex & Fragments Shaders

Now that we defined our Vertices we need to define the Shaders and feed the source code for the Vertex and Fragment Shaders to them.

```cpp
#include<iostream>
#include<glad/glad.h>
#include<GlFW/glfw3.h>
using namespace std;
// Vertex Shader source code
const char* vertexShaderSource = "#version 330 core\n"
"layout(location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"}\n";
// Fragment Shader source code
const char* fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
"}\n";
int main()
{
    // Initialize GLFW
    glfwInit();

    // Tell GLFW what version of OpenGL we are using
    // In this case we are using OpenGL 3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    // Tell GLFW we are using the CORE profile
    // So that means we only have the modern functions
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    // Create a GLFWwindow object of 800 by 800 pixels, naming it "Triangle"
    GLFWwindow* window = glfwCreateWindow(800, 800, "Triangle", NULL, NULL);
    // Error check if the window fails to create
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    // Introduce the window into the current context
    glfwMakeContextCurrent(window);

    //Load GLAD so it configures OpenGL
    gladLoadGL();
    // Specify the viewport of OpenGL in the Window
    // In this case the viewport goes from x = 0, y = 0, to x = 800, y = 800
    glViewport(0, 0, 800, 800);

    // HERE IS STEP 4
    //////////////////////////////////////////////////////////////////////////
  // Create Vertex Shader Object and get its reference
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    // Attach Vertex Shader source to the Vertex Shader Object
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(vertexShader);
    // Create Fragment Shader Object and get its reference
    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    // Attach Fragment Shader source to the Fragment Shader Object
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(fragmentShader);

    //////////////////////////////////////////////////////////////////////////
    // HERE IS STEP 3
    //////////////////////////////////////////////////////////////////////////
    // Vertices coordinates
    GLfloat vertices[] =
    {
        -0.5f, -0.5f , 0.0f, // Lower left corner x, y, z
        0.5f, -0.5f , 0.0f, // Lower right corner x, y, z
        0.0f, 0.5f , 0.0f // Upper corner x, y, z
    };
    //////////////////////////////////////////////////////////////////////////
}
```

Line 1: `GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);`

- `GLuint`: This is a typedef for an unsigned integer. In OpenGL, it is commonly used to represent object names, such as shader or buffer object identifiers.
- `vertexShader`: This variable will hold the identifier (name) of the shader object.
- `glCreateShader(GL_VERTEX_SHADER)`: This function creates an empty shader object and returns its name. The parameter `GL_VERTEX_SHADER` specifies that the shader object to be created is a vertex shader. There are other types of shaders, such as `GL_FRAGMENT_SHADER`, but this code specifically creates a vertex shader.

Line 2: `glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);`

- `glShaderSource`: This function sets the source code in a shader object.
- `vertexShader`: The first argument is the shader object identifier returned by `glCreateShader`.
- `1`: The second argument specifies the number of strings in the array of source code. In this case, it is 1, meaning there is a single string.
- `&vertexShaderSource`: The third argument is a pointer to an array of pointers to the source code strings. Here, `&vertexShaderSource` points to a single string containing the shader source code.
- `NULL`: The fourth argument is an array of string lengths, or NULL if the strings are null-terminated. Since `vertexShaderSource` is a null-terminated string, NULL is passed here.

Line 3: `glCompileShader(vertexShader);`

The function call `glCompileShader(vertexShader);` is an essential step in the shader creation process in OpenGL. Here's what it does and why it's important:

Explanation:

- `glCompileShader`: This function compiles the source code that has been attached to the shader object.
- `vertexShader`: This is the identifier of the shader object that you want to compile. It was previously created using `glCreateShader` and had its source code set with `glShaderSource`.

Now we repeat the same steps for the Fragment shader.

Step 5: define the Shader Program

What is a Shader Program?

A shader program in OpenGL is a collection of shaders that are linked together to be used for rendering. Shaders are small programs written in GLSL (OpenGL Shading Language) that run on the GPU and are responsible for various stages of the graphics pipeline, such as vertex processing and fragment (pixel) processing.

```
#include<iostream>
#include<glad/glad.h>
#include<GLFW/glfw3.h>
using namespace std;
// Vertex Shader source code
const char* vertexShaderSource = "#version 330 core\n"
"layout(location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"}\n";
// Fragment Shader source code
const char* fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
"}\n";
int main()
{
    // Initialize GLFW
    glfwInit();

    // Tell GLFW what version of OpenGL we are using
    // In this case we are using OpenGL 3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    // Tell GLFW we are using the CORE profile
    // So that means we only have the modern functions
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    // Create a GLFWwindow object of 800 by 800 pixels, naming it "Triangle"
    GLFWwindow* window = glfwCreateWindow(800, 800, "Triangle", NULL, NULL);
    // Error check if the window fails to create
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    // Introduce the window into the current context
    glfwMakeContextCurrent(window);

    //Load GLAD so it configures OpenGL
    gladLoadGL();
    // Specify the viewport of OpenGL in the Window
    // In this case the viewport goes from x = 0, y = 0, to x = 800, y = 800
    glViewport(0, 0, 800, 800);

    // HERE IS STEP 4
    //**************************************************************************
  // Create Vertex Shader Object and get its reference
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    // Attach Vertex Shader source to the Vertex Shader Object
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(vertexShader);
    // Create Fragment Shader Object and get its reference
    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    // Attach Fragment Shader source to the Fragment Shader Object
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(fragmentShader);

    //**************************************************************************
    // HERE IS STEP 5
    // Create Shader Program Object and get its reference
    GLuint shaderProgram = glCreateProgram();
    // Attach the Vertex and Fragment Shaders to the Shader Program
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    // Wrap-up/Link all the shaders together into the Shader Program
    glLinkProgram(shaderProgram);

    // Delete the now useless Vertex and Fragment Shader objects
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
    //**************************************************************************
    //
    // HERE IS STEP 3
    //**************************************************************************
    // Vertices coordinates for the triangle
    GLfloat vertices[] =
    {
        -0.5f, -0.5f , 0.0f, // Lower left corner x, y, z
        0.5f, -0.5f , 0.0f, // Lower right corner x, y, z
        0.0f, 0.5f , 0.0f // Upper corner x, y, z
    };
    //**************************************************************************


}
```

Code Breakdown:

1. `GLuint shaderProgram = glCreateProgram();`

- `GLuint`: This is a typedef for an unsigned integer, often used in OpenGL to represent object identifiers.
- `shaderProgram`: This variable will hold the identifier of the shader program.
- `glCreateProgram()`: This function creates an empty shader program object and returns its identifier. This program will eventually contain the compiled and linked shaders.

Step 6: Create reference containers for the Vertex Array Object and the Vertex Buffer Object

Vertex Array Object (VAO) and Vertex Buffer Object (VBO)

Vertex Array Object (VAO)

A VAO is an OpenGL object that stores all of the state needed to supply vertex data. It records the format of vertex attributes and the buffer objects (like VBOs) that provide the actual vertex data. By storing these state settings, a VAO makes it easy to switch between different sets of vertex data and formats without having to re-specify the settings each time.

Vertex Buffer Object (VBO)

A VBO is an OpenGL object that stores vertex data in GPU memory. This can include positions, normals , colors, texture coordinates, and other vertex attributes. By storing the data in the GPU, VBOs allow for efficient rendering as they minimize the amount of data transferred between the CPU and GPU.

Code

```cpp
#include<iostream>
#include<glad/glad.h>
#include<GlFW/glfw3.h>
using namespace std;
// Vertex Shader source code
const char* vertexShaderSource = "#version 330 core\n"
"layout(location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"}\n";
// Fragment Shader source code
const char* fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
"}\n";
int main()
{
    // Initialize GLFW
    glfwInit();

    // Tell GLFW what version of OpenGL we are using
    // In this case we are using OpenGL 3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    // Tell GLFW we are using the CORE profile
    // So that means we only have the modern functions
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    // Create a GLFWwindow object of 800 by 800 pixels, naming it "Triangle"
    GLFWwindow* window = glfwCreateWindow(800, 800, "Triangle", NULL, NULL);
    // Error check if the window fails to create
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    // Introduce the window into the current context
    glfwMakeContextCurrent(window);

    //Load GLAD so it configures OpenGL
    gladLoadGL();
    // Specify the viewport of OpenGL in the Window
    // In this case the viewport goes from x = 0, y = 0, to x = 800, y = 800
    glViewport(0, 0, 800, 800);

    // HERE IS STEP 4
    //**************************************************************************
   // Create Vertex Shader Object and get its reference
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    // Attach Vertex Shader source to the Vertex Shader Object
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(vertexShader);
    // Create Fragment Shader Object and get its reference
    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    // Attach Fragment Shader source to the Fragment Shader Object
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(fragmentShader);

    //**************************************************************************
    // HERE IS STEP 5
    // Create Shader Program Object and get its reference
    GLuint shaderProgram = glCreateProgram();
    // Attach the Vertex and Fragment Shaders to the Shader Program
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    // Wrap-up/Link all the shaders together into the Shader Program
    glLinkProgram(shaderProgram);

    // Delete the now useless Vertex and Fragment Shader objects
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
    //**************************************************************************
    // HERE IS STEP 3
    //**************************************************************************
    // Vertices coordinates for the triangle
    GLfloat vertices[] =
    {
        -0.5f, -0.5f , 0.0f, // Lower left corner x, y, z
        0.5f, -0.5f , 0.0f, // Lower right corner x, y, z
        0.0f, 0.5f , 0.0f // Upper corner x, y, z
    };
    //**************************************************************************
    // HERE IS STEP 6
    //**************************************************************************
    // Create reference containers for the Vartex Array Object and the Vertex Buffer Object
    GLuint VAO, VBO;

    // Generate the VAO and VBO with only 1 object each
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);

    // Make the VAO the current Vertex Array Object by binding it
    glBindVertexArray(VAO);

    // Bind the VBO specifying it's a GL_ARRAY_BUFFER
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // Introduce the vertices into the VBO
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    // Configure the Vertex Attribute so that OpenGL knows how to read the VBO
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
    // Enable the Vertex Attribute so that OpenGL knows to use it
    glEnableVertexAttribArray(0);

    // Bind both the VBO and VAO to 0 so that we don't accidentally modify the VAO and VBO we created
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
    //**************************************************************************
```

Step-by-Step Explanation

1. **Create Reference Containers**:

```
GLuint VAO, VBO;
```

   - `VAO` and `VBO` are variables that will store the identifiers for the Vertex Array Object and the Vertex Buffer Object, respectively.

2. **Generate the VAO and VBO**:

```
glGenVertexArrays(1, &VAO);
glGenBuffers(1, &VBO);
```

   - `glGenVertexArrays(1, &VAO)` generates one VAO and stores its identifier in `VAO`.
   - `glGenBuffers(1, &VBO)` generates one VBO and stores its identifier in `VBO`.

   ⛔ Make sure that u generate the VAO before the VBO the order is very important!!

3. **Bind the VAO**:

```
glBindVertexArray(VAO);
```

   - This makes the VAO the current vertex array object. Any subsequent vertex attribute calls will be stored in this VAO.

4. **Bind the VBO**:

```
glBindBuffer(GL_ARRAY_BUFFER, VBO);
```

   - This binds the VBO to the `GL_ARRAY_BUFFER` target. This means that subsequent calls to functions that modify buffer objects will affect this VBO.

5. **Fill the VBO with Data**:

```
glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);
```

   - `glBufferData` creates a new data store for the VBO, using the data specified in the `vertices` array. `GL_STATIC_DRAW` indicates that the data will not change frequently.

6. **Configure Vertex Attributes**:

```
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
```

   - This function specifies how OpenGL should interpret the vertex data.

- `0`: The index of the vertex attribute.
- `3`: The number of components per vertex attribute (x, y, z).
- `GL_FLOAT`: The type of each component.
- `GL_FALSE`: Whether fixed-point data values should be normalized.
- `3 * sizeof(float)`: The stride, or byte offset between consecutive vertex attributes.
- `(void*)0`: The offset of the first component.

7. **Enable the Vertex Attribute**:

```
glEnableVertexAttribArray(0);
```

   - This enables the vertex attribute at index 0 so that it can be used by the vertex shader.

8. **Unbind the VBO and VAO**:

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

   - These calls unbind the currently bound VBO and VAO to avoid accidental modifications.

Full Code

```cpp
#include<iostream>
#include<glad/glad.h>
#include<GLFW/glfw3.h>
using namespace std;
// Vertex Shader source code
const char* vertexShaderSource = "#version 330 core\n"
"layout(location = 0) in vec3 aPos;\n"
"void main()\n"
"{\n"
"   gl_Position = vec4(aPos.x, aPos.y, aPos.z, 1.0);\n"
"}\n";
// Fragment Shader source code
const char* fragmentShaderSource = "#version 330 core\n"
"out vec4 FragColor;\n"
"void main()\n"
"{\n"
"   FragColor = vec4(1.0f, 0.5f, 0.2f, 1.0f);\n"
"}\n";
int main()
{
    // Initialize GLFW
    glfwInit();

    // Tell GLFW what version of OpenGL we are using
    // In this case we are using OpenGL 3.3
    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 3);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
    // Tell GLFW we are using the CORE profile
    // So that means we only have the modern functions
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
    // Create a GLFWwindow object of 800 by 800 pixels, naming it "Triangle"
    GLFWwindow* window = glfwCreateWindow(800, 800, "Triangle", NULL, NULL);
    // Error check if the window fails to create
    if (window == NULL)
    {
        std::cout << "Failed to create GLFW window" << std::endl;
        glfwTerminate();
        return -1;
    }
    // Introduce the window into the current context
    glfwMakeContextCurrent(window);

    //Load GLAD so it configures OpenGL
    gladLoadGL();
    // Specify the viewport of OpenGL in the Window
    // In this case the viewport goes from x = 0, y = 0, to x = 800, y = 800
    glViewport(0, 0, 800, 800);

    // HERE IS STEP 4
    //*****************************************************************************
  // Create Vertex Shader Object and get its reference
    GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
    // Attach Vertex Shader source to the Vertex Shader Object
    glShaderSource(vertexShader, 1, &vertexShaderSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(vertexShader);
    // Create Fragment Shader Object and get its reference
    GLuint fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
    // Attach Fragment Shader source to the Fragment Shader Object
    glShaderSource(fragmentShader, 1, &fragmentShaderSource, NULL);
    // Compile the Vertex Shader into machine code
    glCompileShader(fragmentShader);

    //*****************************************************************************
    // HERE IS STEP 5
    // Create Shader Program Object and get its reference
    GLuint shaderProgram = glCreateProgram();
    // Attach the Vertex and Fragment Shaders to the Shader Program
    glAttachShader(shaderProgram, vertexShader);
    glAttachShader(shaderProgram, fragmentShader);
    // Wrap-up/Link all the shaders together into the Shader Program
    glLinkProgram(shaderProgram);

    // Delete the now useless Vertex and Fragment Shader objects
    glDeleteShader(vertexShader);
    glDeleteShader(fragmentShader);
    //*****************************************************************************
    //
    // HERE IS STEP 3
    //*****************************************************************************
    // Vertices coordinates for the triangle
    GLfloat vertices[] =
    {
        -0.5f, -0.5f , 0.0f, // Lower left corner x, y, z
        0.5f, -0.5f , 0.0f, // Lower right corner x, y, z
        0.0f, 0.5f , 0.0f // Upper corner x, y, z
    };
    //*****************************************************************************
    // HERE IS STEP 6
    //*****************************************************************************
    // Create reference containers for the Vartex Array Object and the Vertex Buffer Object
    GLuint VAO, VBO;

    // Generate the VAO and VBO with only 1 object each
    glGenVertexArrays(1, &VAO);
    glGenBuffers(1, &VBO);

    // Make the VAO the current Vertex Array Object by binding it
    glBindVertexArray(VAO);

    // Bind the VBO specifying it's a GL_ARRAY_BUFFER
    glBindBuffer(GL_ARRAY_BUFFER, VBO);
    // Introduce the vertices into the VBO
    glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW);

    // Configure the Vertex Attribute so that OpenGL knows how to read the VBO
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 3 * sizeof(float), (void*)0);
    // Enable the Vertex Attribute so that OpenGL knows to use it
    glEnableVertexAttribArray(0);

    // Bind both the VBO and VAO to 0 so that we don't accidentally modify the VAO and VBO we created
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    glBindVertexArray(0);
    //*****************************************************************************
    // Main while loop
    while (!glfwWindowShouldClose(window))
    {
        // Specify the color of the background
        glClearColor(0.07f, 0.13f, 0.17f, 1.0f);
        // Clean the back buffer and assign the new color to it
        glClear(GL_COLOR_BUFFER_BIT);
        // Tell OpenGL which Shader Program we want to use
        glUseProgram(shaderProgram);
        // Bind the VAO so OpenGL knows to use it
        glBindVertexArray(VAO);
        // Draw the triangle using the GL_TRIANGLES primitive
        glDrawArrays(GL_TRIANGLES, 0, 3);
        // Swap the back buffer with the front buffer
        glfwSwapBuffers(window);
        // Take care of all GLFW events
        glfwPollEvents();
    }



    // Delete all the objects we've created
    glDeleteVertexArrays(1, &VAO);
    glDeleteBuffers(1, &VBO);
    glDeleteProgram(shaderProgram);
    // Delete window before ending the program
    glfwDestroyWindow(window);
    // Terminate GLFW before ending the program
    glfwTerminate();
    return 0;
}
```

Output

Assignment

Draw a Square