Jeremy Taylor

14336420

ICS 139W

Rendering a Rotating Cube with OpenGL 4.5 and C++ in Visual Studio 2017

Initial Setup:

1. The IDE being used in the following instructions is Visual Studio 2017. Windows 7 SP1, or a later version of Windows, is required to install Visual Studio 2017. You can download and install Visual Studio Community 2017 from Microsoft at https://www.visualstudio.com/downloads/. Be sure to install the core C++ components, at a minimum (Visual Studio C++ core features, VC++ 2017 toolset, and C++ profiling toolset).
2. Check that your graphics card can support OpenGL 4.5. Nvidia lists OpenGL 4.5 compatible cards at https://developer.nvidia.com/opengl-driver. AMD may not list supported OpenGL versions for their GPUs, so you may need to do additional searching or use a tool such as http://realtech-vr.com/admin/glview.
3. Ensure your graphics drivers are up to date. Drivers for AMD GPUs can be found at http://support.amd.com/en-us/download. Similarly, Nvidia's drivers can be found at http://www.nvidia.com/Download/index.aspx.
4. Download the libraries that are being used in this project. GLFW 3.2 can be downloaded at http://www.glfw.org/download.html (use the 32-bit Windows binaries download button). GLM can be downloaded at https://glm.g-truc.net/0.9.8/index.html by clicking the "Downloads" button on the left-hand side of the page. Glad can be downloaded at http://glad.dav1d.de/ (select a core profile and gl version 4.5).
    a) GLFW is being used as an alternative to the Windows API for creating an application window. It is much more convenient to work with GLFW as it can handle OpenGL context creation for us.

b) GLM is a standard mathematics library used in OpenGL applications. It is more convenient, for the sake of this manual, to use an established mathematics library instead of writing our own.

c) On most platforms, OpenGL versions later than 1.1 cannot be accessed without an OpenGL Loading Library. Glad provides a simple web tool to download its loader library, which makes it a convenient choice for this project.

Creating the Project:

1. Start Visual Studio 2017 and create a new empty project by using the Ctrl-Shift-N shortcut and selecting Windows Desktop Wizard (the containing directory and project name do need to be anything specific).
   a) After clicking "OK" on the New Project window, select "Console Application (.exe)" as the application type, uncheck the "Precompiled Header" box, and check the "Empty Project" box. Click "Ok" to create the project.
   b) The project's configuration should be set to 32-bit (x86) Debug. The drop-down menus for these items are in the toolbar underneath Visual Studio's topmost toolbar.
2. Use the Ctrl-Shift-A shortcut to add a new C++ source file (located under Installed>Visual C++) called main.cpp to the project.
3. In main.cpp, add the following code, which will define the entry-point of the program:

```
int main()
{
    return 0;
}
```

4. Compile the program by pressing the F7 key, which will also create the Debug output directory for the project.
5. In the Solution Explorer, right click the project item (which will have the same name from step 1) and click the "Open Folder in File Explorer" list item. Unzip the downloaded GLFW, glad, and GLM folders to this directory.

a) You may want to rename the GLFW folder to “glfw” and the GLM folder to “glm” for simplicity (the original folders may have longer names that include the version numbers).

6. Once again, right click the project item, but select the “Properties” list item this time. Under Configuration Properties>C/C++>General, find “Additional Include Directories.” Click in this box and add “$(ProjectDir)\<your-GLFW-folder>\include;$(ProjectDir)\<your-GLM-folder>\;$(ProjectDir)\glad\include;” to the box.
7. While still in the Property Pages window, navigate into Configuration Properties>Linker>General. Find the “Additional Library Directories” box and add “$(ProjectDir)\glfw-3.2\lib-vc2015;” to the box. In the “Additional Dependencies” box under Linker>Input, add “glfw3.lib” and then press “OK” to save the changes.
8. Add the glfw3.dll file you downloaded (it’s located in glfw-3.2\lib-vc2015) to the Debug output directory for your project. This directory is located at <Solution Directory>/Debug/.
9. Using the Alt-Shift-A shortcut, add glad’s source code to the project by navigating to glad/src/ and double-clicking the glad.c file.

Writing the Program:

1. Add a new C++ header file to the project called “GlfwFunctions.h.” In it, add the following code:

```
#pragma once

#include <glad/glad.h>
#include <GLFW/glfw3.h>

#undef CreateWindow // Get rid of WinAPI macro definition

namespace glfw
{
    // Creates a GLFWWindow that has been set up for an OpenGL 4.5 Core context
    GLFWwindow* CreateWindow( unsigned width, unsigned height, const char* title );

    // Function that will be called when an internal error occurs in GLFW
    void HandleError( int errorCode, const char* errorDescription );

    // Function that will be called by GLFW when the program receives keyboard input
    void HandleKeypress( GLFWwindow* window, int keyCode, int scanCode, int action, int
       modKeys );
```

```
    // Function that will be called by GLFW when the window is resized
    void HandleWindowResize( GLFWwindow* window, int newWidth, int newHeight );
}
```

a) To help keep the program slightly more organized, separate namespaces and header files will be used for different aspects of the program. This file will deal with functionality related to GLFW.

2. Add a new C++ source file called “GlfwFunctions.cpp” to the project. In it, add the following code:

```
#include "GlfwFunctions.h"


#include "Utility.h"

#include <iostream>

GLFWwindow * glfw::CreateWindow( unsigned width, unsigned height, const char * title )
{
    // This hint must be included to let GLFW know that the hints are being set for an
       OPENGL context
    // and not the OPENGL ES API (or NO_API).
    glfwWindowHint( GLFW_CLIENT_API, GLFW_OPENGL_API );

    // Request a context with version 4.5, with deprecated functionality removed.
    glfwWindowHint( GLFW_CONTEXT_VERSION_MAJOR, 4 );
    glfwWindowHint( GLFW_CONTEXT_VERSION_MINOR, 5 );
    glfwWindowHint( GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE );
    glfwWindowHint( GLFW_OPENGL_FORWARD_COMPAT, GL_FALSE );
    return glfwCreateWindow( width, height, title, nullptr, nullptr );
}

void glfw::HandleError( int errorCode, const char * errorDescription )
{
    std::cerr << "An error occurred in GLFW (" << errorCode << ": " << errorDescription
       << ")" << std::endl;
}

void glfw::HandleKeypress( GLFWwindow * window, int keyCode, int scanCode, int action,
int modKeys )
{
    switch ( keyCode )
    {
        case GLFW_KEY_ESCAPE:
        {
            if ( action == GLFW_PRESS )
            {
                // The window will close when the user presses escape
                glfwSetWindowShouldClose( window, GLFW_TRUE );
            }
        }
    }
```

```
}

void glfw::HandleWindowResize( GLFWwindow * window, int newWidth, int newHeight )
{
    // Resize OpenGL's viewport rectangle so the rasterized output image fits within the
window
    glViewport( 0, 0, newWidth, newHeight );

    util::Camera* camera = static_cast< util::Camera* >( glfwGetWindowUserPointer(
       window ) );
    camera->projectionMatrix = glm::perspective( glm::radians( 45.0f ), static_cast<
       float >( newWidth ) / newHeight, 0.1f, 100.0f );

}
```

a) The CreateWindow function will be used to create the application's window. The glfwWindowHint(…) functions are used to tell GLFW how to optionally configure the created OpenGL context.

b) HandleError will be used as the callback function for whenever an internal error in GLFW occurs. We will use this function to log and ignore the errors.

c) We will use HandleKeypress to respond to key presses. The only notable key press will be the escape key, which we will use to exit the program. By using a switch statement or a series of if-else statements, you can respond to any additional keyboard input that you desire.

d) The area of the window which OpenGL renders to does not automatically update when the window resizes. In the HandleWindowResize function we use glViewPort(…) to resize the viewport rectangle.

3. Add another C++ header file to the project and call it "GlFunctions.h." In it, add the following function declarations and struct definitions:

```
#pragma once

#include <glad/glad.h>

namespace gl
{
    struct Vec3f
    {
        float x, y, z;
    };
    struct Vertex
    {
        Vec3f position; // Position of the vertex in world space
```

```
        Vec3f color;
    };

    struct VertexBufferFormat
    {
        GLuint bindingIndex;    // specifies binding point of the VBO
        GLintptr offset;        // offset into the source data
        GLuint stride;          // number of bytes per vertex
    };

    struct AttributeFormat
    {
        GLuint size;            // number of components per source vector
        GLenum type;            // data type of source data
        GLboolean normalized;   // is the data a normalized integer
        GLuint relativeOffset;  // the offset into the vbo where this attribute starts
    };

    struct VertexArrayAttribute
    {
        AttributeFormat format;
        GLuint vertexBindingIndex;  // the vbo binding point that should be used for the
vao
        GLuint attributeIndex;      // the attribute location in the shader that this
attribute matches with
    };

    // Queries OpenGL for the name of the GPU and driver version
    const GLubyte* GetGraphicsDeviceName();

    // Queries OpenGL for the version of OpenGL being used
    const GLubyte* GetOpenGlVersion();


    // Creates a Vertex Buffer Object (VBO), using the vertices parameter as the source
data
    GLuint CreateVertexBuffer( const Vertex* vertices, unsigned numVertices );

    // Creates an Element Buffer Object (EBO), using the indices parameter as the source
data
    GLuint CreateElementBuffer( const unsigned* indices, unsigned numIndices );

    // Creates an "empty" Vertex Array Object (VAO)
    GLuint CreateVertexArrayObject();

    // Binds the given VBO to the VAO and defines where OpenGL should read from the data
    void BindVertexBufferToVertexArray( GLuint vertexArray, GLuint vertexBuffer,
VertexBufferFormat bufferFormat );

    // Binds the given EBO to the VAO, which will cause the indices supplied by the EBO
to be used
    // in indexed draw calls.
    void BindElementBufferToVertexArray( GLuint vertexArray, GLuint elementBuffer );

    // Specifies the organization of a vertex array attribute of the given VAO.
    void SetVertexArrayAttribute( GLuint vertexArray, VertexArrayAttribute attribute );

    // Creates and compiles a shader from the source code in the given string arrays
    GLuint CreateShader( GLenum type, const GLchar** source );

    // Creates a Shader Program object by linking together a vertex and fragment shader
```

```
    GLuint CreateBasicShaderProgram( GLuint vertShader, GLuint fragShader );

    // Sets the value for the shader uniform variable with the given name in the
shaderProgram.
    void SetProgramUniformMat4f( GLuint shaderProgram, const char* name, const GLfloat*
value );
}
```

a) These functions will help simplify creating the OpenGL objects required to render the cube. They abstract away the underlying OpenGL code but are designed in a way that, hopefully, makes it easier to grasp the concepts involved.

4. Add a C++ source file called “GlFunctions.cpp” to the project. Add to it the definitions of the functions declared in GlFunctions.h above:

```
#include "GlFunctions.h"

const GLubyte * gl::GetGraphicsDeviceName()
{
    return glGetString( GL_RENDERER );
}

const GLubyte * gl::GetOpenGlVersion()
{
    return glGetString( GL_VERSION );
}

GLuint gl::CreateVertexBuffer( const Vertex * vertices, unsigned numVertices )
{
    GLuint buffer;

    // Allocates and returns the name of a new buffer object.
    glCreateBuffers( 1, &buffer );

    // Copies the data (vertices) of size numVertices * sizeof(Vertex) into the buffer
object.
    glNamedBufferData( buffer, numVertices * sizeof( Vertex ), vertices,
GL_STATIC_DRAW );
    return buffer;
}

GLuint gl::CreateElementBuffer( const unsigned * indices, unsigned numIndices )
{
    GLuint buffer;
    glCreateBuffers( 1, &buffer );

    glNamedBufferData( buffer, numIndices * sizeof( unsigned ), indices,
GL_STATIC_DRAW );
    return buffer;
}

GLuint gl::CreateVertexArrayObject()
{
```

```
    GLuint vertexArray;

    // Allocates and returns the name of a new buffer object.
    glCreateVertexArrays( 1, &vertexArray );

    return vertexArray;
}

void gl::BindVertexBufferToVertexArray( GLuint vertexArray, GLuint vertexBuffer,
VertexBufferFormat bufferFormat )
{
    // Associates the vertex buffer object with the vertex array object by binding it to
the given binding point.
    // The layout of the VBO's data as it pertains to the VAO is also defined via the
offset and stride parameters.
    glVertexArrayVertexBuffer( vertexArray, bufferFormat.bindingIndex, vertexBuffer,
bufferFormat.offset, bufferFormat.stride );
}

void gl::BindElementBufferToVertexArray( GLuint vertexArray, GLuint elementBuffer )
{
    // Associates the given element (indices) buffer with the VAO so that DrawElements
    // calls will use elementBuffer to determine the vertex processing order.
    glVertexArrayElementBuffer( vertexArray, elementBuffer );
}

void gl::SetVertexArrayAttribute( GLuint vertexArray, VertexArrayAttribute attribute )
{
    // Specifies the location index of the attribute and defines the number
    // of components per vector, the data type, whether the data is a normalized integer,
    // and where the attribute's data begins in the source VBO (relativeOffset).
    glVertexArrayAttribFormat( vertexArray, attribute.attributeIndex,
attribute.format.size, attribute.format.type,
        attribute.format.normalized, attribute.format.relativeOffset );

    // Specifies which VBO supplies the source data for the VAO by using the index of
the vertex buffer binding point.
    glVertexArrayAttribBinding( vertexArray, attribute.attributeIndex,
attribute.vertexBindingIndex );

    // Enables the attribute array at the given index so that subsequent draw calls will
access and use
    // those values.
    glEnableVertexArrayAttrib( vertexArray, attribute.attributeIndex );
}

GLuint gl::CreateShader( GLenum type, const GLchar ** source )
{
    GLuint shader = glCreateShader( type );

    // Copies the source data for the shader into the shader object.
    glShaderSource( shader, 1, source, nullptr );

    // Compiles the shader - this will fail if the shader uses invalid syntax
    // or has other errors.
    glCompileShader( shader );
    return shader;
}

GLuint gl::CreateBasicShaderProgram( GLuint vertShader, GLuint fragShader )
{
```

```
    GLuint shaderProgram = glCreateProgram();

    // Attach shader objects to the program for linkage - it is recommended to only
    // use one shader object per shader stage.
    glAttachShader( shaderProgram, vertShader );
    glAttachShader( shaderProgram, fragShader );

    // Link the different shader objects into one program - there must be
    // a vertex and fragment shader at a minimum.
    glLinkProgram( shaderProgram );
    glDetachShader( shaderProgram, vertShader ); // Detaches the shader so the memory
can be freed later
    glDetachShader( shaderProgram, fragShader );
    return shaderProgram;
}

void gl::SetProgramUniformMat4f( GLuint shaderProgram, const char * name, const GLfloat*
value )
{
    GLint location = glGetUniformLocation( shaderProgram, name ); // get the location of
the uniform in the shader
    glProgramUniformMatrix4fv( shaderProgram, location, 1, GL_FALSE, value ); // upload
the data to the shader
}
```

a) You can read the comments in the functions to get some familiarity with what they do. The structure of an OpenGL program will be discussed later.

5. Add a final C++ header file to the project called “Utility.h,” and add the following struct definition to it:

```
#pragma once


#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/type_ptr.hpp>

namespace util
{
    // Used to transform the cube from world to view space and
    // to apply a perspective transformation to the resulting
    // vertex position.
    struct Camera
    {
        glm::mat4 viewMatrix;
        glm::mat4 projectionMatrix;
    };
}
```

6. Back in main.cpp, add the following include statements to the top of the file:

```
#include "GlfwFunctions.h"

#include "GlFunctions.h"
#include "Utility.h"
```

```
#include <iostream>
```

7. In main.cpp, add the following code above the “return 0;” statement:

```
glfwSetErrorCallback( glfw::HandleError );

if ( glfwInit() != GLFW_TRUE )
{
    return 1;
}
```

   a) GLFW must be initialized before OpenGL so that we can get a window to associate with an OpenGL context. We also set the error callback for GLFW in case GLFW cannot make an OpenGL context.

      i. If you get error code 65543 when running the program, it is likely that your GPU does not support OpenGL 4.5.

8. Underneath the previous code, add the following code that will set callbacks for keyboard input and initializes OpenGL:

```
    const unsigned windowWidth = 640, windowHeight = 360; // 16:9 resolution used

    GLFWwindow* window = glfw::CreateWindow( windowWidth, windowHeight, "Rotating Cube - OpenGL 4.5 Core" );
    if ( window == nullptr )
    {
        glfwTerminate();
        return 1;
    }

    glfwSetKeyCallback( window, glfw::HandleKeypress );
    glfwSetWindowSizeCallback( window, glfw::HandleWindowResize );

    // Set the application's camera as the user pointer (so we can change it on window resize events)
    util::Camera camera;
    glfwSetWindowUserPointer( window, &camera );

    // Use glad library to load OpenGL functions - requires passing the process address from GLFW to the loader.
    // You could go without a loader like glad, but then on some systems (Windows) you could only use OpenGL 1.2.
    glfwMakeContextCurrent( window );
    gladLoadGLLoader( ( GLADloadproc ) glfwGetProcAddress );

    glfwSwapInterval( 1 ); // Enables vertical sync (the default framebuffer will be swapped once per screen refresh)

    // Display the graphics device and version of the created OpenGL context for logging purposes
    // and to ensure OpenGL 4.5+ is supported.
```

```
    std::cout << "Graphics Device: " << gl::GetGraphicsDeviceName() << "\nOpenGL version:
" << gl::GetOpenGlVersion() << std::endl;
```

a) GLFW uses callback functions to handle events. In order to modify objects that exist outside the scope of these functions (without using a global variable), you can pass a "user" pointer to GLFW.

9. Next, add the following code, which will declare an array of cube vertices and an array of indices:

```
    // Define the data used for the cube and set up the Vertex Buffer Object (VBO),
Vertex Array Object (VAO) and Element/Indices buffer

    // for the cube.
    static gl::Vertex cubeVertices[] =
    {
        // Note - we can draw the sides, top, and bottom by suppling vertices
        // of a triangle out of the edges of the front and back faces.

        // front face positions and colors
        -1.0f,  -1.0f,  1.0f,       0.8f, 0.f, 0.f, // front is red
        1.0f,   -1.0f,  1.0f,       0.8f, 0.f, 0.f,
        1.0f,   1.0f,   1.0f,       0.8f, 0.f, 0.f,
        -1.0f,  1.0f,   1.0f,       0.8f, 0.f, 0.f,
        // back face positions and colors
        -1.0f,  -1.0f,  -1.0f,      0.f, 0.f, 0.8f, // back is blue
        1.0f,   -1.0f,  -1.0f,      0.f, 0.f, 0.8f,
        1.0f,   1.0f,   -1.0f,      0.f, 0.f, 0.8f,
        -1.0f,  1.0f,   -1.0f,      0.f, 0.f, 0.8f,
    };

    static GLuint cubeIndices[] =
    {
        // front
        0, 1, 2, 2, 3, 0,
        // back
        7, 6, 5, 5, 4, 7,
        // bottom
        4, 0, 3, 3, 7, 4,
        // top
        1, 5, 6, 6, 2, 1,
        // left
        4, 5, 1, 1, 0, 4,
        // right
        3, 2, 6, 6, 7, 3
    };

    GLuint vertexBuffer = gl::CreateVertexBuffer( cubeVertices,
ARRAYSIZE( cubeVertices ) );
    GLuint elementBuffer = gl::CreateElementBuffer( cubeIndices,

ARRAYSIZE( cubeIndices ) );
```

a) The minimum objects required to pass primitive data to an OpenGL shader program are a vertex buffer object and a vertex array object. However, it is often more efficient to also specify an indices buffer because we can reduce the number of overlapping vertices that are processed.

10. Add the following code, which will create a vertex array object and configure it for our cube data:

```
    GLuint vertexArrayObject = gl::CreateVertexArrayObject();


    gl::VertexBufferFormat bufferFormat;
    // This number must be used for matching the buffer data source (vertex buffer
    // and its binding point) to the VAO Attribute. The Attribute tells OpenGL how to
    // interpret the data from the VBO at the given binding point.
    bufferFormat.bindingIndex = 0;

    bufferFormat.offset = ( GLintptr ) nullptr; // The shader will start reading from
the start of the source data
    bufferFormat.stride = sizeof( gl::Vertex ); // The shader will consider every
sizeof(gl::Vertex) bytes to be another Vertex

    gl::BindVertexBufferToVertexArray( vertexArrayObject, vertexBuffer, bufferFormat );

    gl::VertexArrayAttribute attrib;
    attrib.attributeIndex = 0; // Use attribute index 0 of the VAO for position
    attrib.vertexBindingIndex = 0; // The source data is coming from VBO at binding
point 0 (this is a constant in this example)

    attrib.format.normalized = GL_FALSE;
    attrib.format.type = GL_FLOAT;

    // size tells the vertex shader how many components each vector contains (1-4
components for x,y,z,w) in the source data.
    // If this number is more than the shader uses, the extra is ignored. If this number
is less, then the extra components in the
    // shader are initialized from the default (0,0,0,1) vector.
    attrib.format.size = 3;

    // relativeOffset tells the vertex shader where the data for the attribute begins in
the current source vertex.
    // This is an offset per vertex rather than an offset into the source data as a
whole.
    attrib.format.relativeOffset = offsetof( gl::Vertex, gl::Vertex::position );

    gl::SetVertexArrayAttribute( vertexArrayObject, attrib );

    attrib.format.relativeOffset = offsetof( gl::Vertex, gl::Vertex::color );
    attrib.attributeIndex = 1; // Use attribute index 1 of the VAO for color

    gl::SetVertexArrayAttribute( vertexArrayObject, attrib );

    gl::BindElementBufferToVertexArray( vertexArrayObject, elementBuffer );
```

a) To summarize the above code, we're creating a vertex array object in order to specify how the vertex stage of our shader program should read from our vertex data.

b) Our cube data consists of 6 floats for each of the 8 vertices, where the first 3 floats are the position and the last 3 floats are the vertex color. We use the offsetof macro to calculate the relative offset of the position and color values in our vertex buffer.

11. The next section of code to add will create the shader program that we will use to render the cube:

```
    // Create shaders - for simplicty, we aren't going to read them from files

    const GLchar* vertexShaderSource =
        "#version 450 core\n"

        "layout( location = 0 ) in vec3 position;\n"
        "layout( location = 1 ) in vec3 color;\n"

        "layout( location = 0 ) out vec3 outColor;\n"

        // uniforms - must pass these in from the application
        "uniform mat4 projection;\n"
        "uniform mat4 view;\n"
        "uniform mat4 model;\n"

        "void main()\n"
        "{\n"
        "    outColor = color;\n" // pass the color value through directly to fragment
shader
        "    gl_Position = projection * view * model * vec4( position, 1.0 );\n" //
screenspace position
        "}\n";

    const GLchar* fragShaderSource =
        "#version 450 core\n"

        "layout( location = 0 ) in vec3 color;\n"

        "out vec4 outColor;\n"

        "void main()\n"
        "{"
        "    outColor = vec4( color, 1.0 );\n" // directly use the vertex color as the
fragment color
        "}\n";

    GLuint vertShader = gl::CreateShader( GL_VERTEX_SHADER, &vertexShaderSource );
    GLuint fragShader = gl::CreateShader( GL_FRAGMENT_SHADER, &fragShaderSource );
    GLuint shaderProgram = gl::CreateBasicShaderProgram( vertShader, fragShader );
    glDeleteShader( vertShader ); // free the shader object and its associated source
buffer
    glDeleteShader( fragShader );
```

a) Because we're only using one vertex and one fragment shader, it is easier to put the shader source code in the C++ code itself instead of reading it from a file. In a larger program with multiple large shaders, it would be cleaner to read your shaders from files during initialization.

b) The vertex shader calculates the clip space coordinates of the vertices and assigns them to gl_Position, which is the default keyword for the output value of the vertex shader. The perspective matrix passed to the shader program will determine the view frustum used to transform and cull vertices.

c) The fragment shader simply outputs the color value that it receives from the vertex shader stage. This output from the fragment shader is used as the color of the shaded fragment, which is a region of the screen that is at least one pixel in size.

12. The remaining code will set up the camera's view and projection matrices. It will also define the structure of the rendering loop:

```
    // Use the identity matrix as the model (to world) matrix

    glm::mat4 modelMatrix( 1.f );

    // Set up the camera
    camera.viewMatrix = glm::lookAt(
        glm::vec3( 0.f, 0.f, 5.f ),     // camera position
        glm::vec3( 0.f, 0.f, 0.f ),     // point the camera looks at
        glm::vec3( 0.f, 1.f, 0.f )      // up vector
    );

    camera.projectionMatrix = glm::perspective( glm::radians( 45.0f ), static_cast<
float >( windowWidth ) / windowHeight, 0.1f, 100.0f );

    // Pass the values of the model, view, and projection matrices to the shader
    gl::SetProgramUniformMat4f( shaderProgram, "model", glm::value_ptr( modelMatrix ) );
    gl::SetProgramUniformMat4f( shaderProgram, "view",
glm::value_ptr( camera.viewMatrix ) );
    gl::SetProgramUniformMat4f( shaderProgram, "projection",
glm::value_ptr( camera.projectionMatrix ) );

    // The color buffer will be cleared to 50% gray with glClear commands
    glClearColor( 0.5f, 0.5f, 0.5f, 1.0f );

    // Use our shader program that we created
    glUseProgram( shaderProgram );

    // Tell OpenGL to cull polygon faces that face away from the camera (with GL_BACK).
```

```
    // By default, faces with CCW winding order are considered front faces.
    glEnable( GL_CULL_FACE );
    glCullFace( GL_BACK );
    while ( !glfwWindowShouldClose( window ) )
    {
        glfwPollEvents();
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

        // Set the projection uniform every frame in case the window is resized
        gl::SetProgramUniformMat4f( shaderProgram, "projection",
glm::value_ptr( camera.projectionMatrix ) );

        // Rotate the cube around the Y-axis
        modelMatrix = glm::rotate( modelMatrix, glm::radians(1.0f), glm::vec3( 0.f, 1.f,
0.f ) );
        gl::SetProgramUniformMat4f( shaderProgram, "model",
glm::value_ptr( modelMatrix ) );
        // The VAO must be bound before any draw calls. We use glDrawElements
        // because we are doing an indexed draw with an element buffer.
        glBindVertexArray( vertexArrayObject );
        glDrawElements( GL_TRIANGLES, ARRAYSIZE( cubeIndices ), GL_UNSIGNED_INT,
nullptr );

        glfwSwapBuffers( window );
    }
```

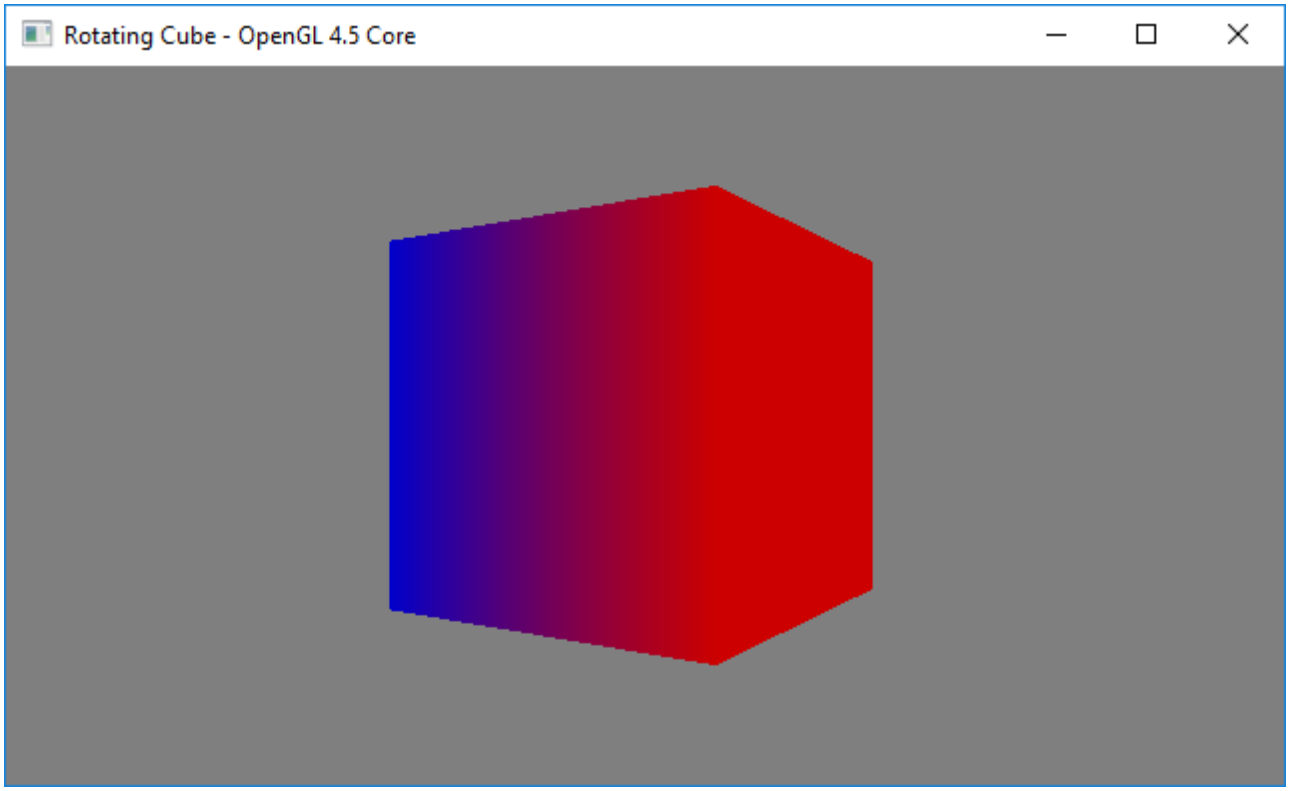
a) The model to world matrix is set to the identity matrix. This means that the cube's position in world space should be the same as its object space.

b) The camera's view matrix is created using the glm::lookAt function. The camera's position is set to (x=0, y=0, z=5).

c) In the rendering loop, we rotate the cube by applying a rotation transformation to its model matrix. To see this change in the rendered image, we have to pass the new value to the shader again.

Running the Program:

1. Compile the program by pressing the F7 key.
2. Run the program without debugging by going to Debug>Start Without Debugging in the topmost toolbar.
3. If the program has compiled and run successfully, you should see a red and blue rotating cube as seen below:



a) If you do not see a cube but the program successfully compiles and runs, you may be missing some of the application or shader code:

   i. Check the vertex shader to ensure that you are multiplying the projection, view, and model matrices in the correct order. This tutorial is using column-major matrices and vectors, so concatenation of transformations is done with the last transformation written as the first one when doing multiplication.

   ii. Check that you are correctly passing the projection, view, and model matrices to the shader program. You may have missed passing one or more values, or you may have mistyped the name of the uniform variable in either the shader or the call to SetProgramUniform4f

   iii. You may also have a typo in the code that configures the vertex array object and its attributes. Ensure that the attribute indices matches those specified in the shaders.

b) If you see a cube rendered, but its color is black, then you may have forgotten to assign to the color variable in one, or both, of the shaders. There may also be an issue with the values of the relative offsets for the position and color vertex array attributes.