Brett Stelly

October 19, 2018

**Computer Graphics**

This project makes use of OpenGL and therefore requires that the glfw3.h header file and GLM libraries are linked to the project. The glfw3 header file gives access to the OpenGL library and the GLM libraries are OpenGL mathematics libraries for doing calculations. These files can be found online.

**Cameras and Projections**

**CameraApplication class that derives from the Application class**

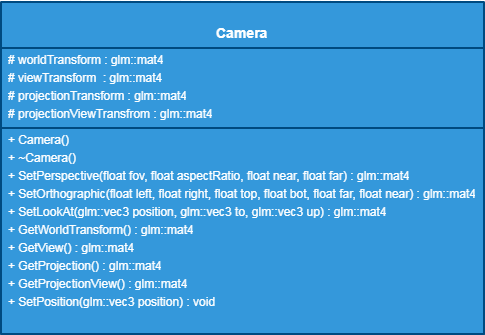
**Pre-conditions:** To create a class for the Camera application that derives from an application class, you will first need to make an application class with functions for startup, shutdown, update, draw, clearScreen, and run. The startup, shutdown, update, and draw functions should be made to be pure virtual functions, so that they can be overridden. This application class should be able to open a window. The GLM libraries need to be linked to the project.

Include the application header in the CameraApplication header. The CameraApplication class inherits from the Application class and should override functions for all the pure virtual functions in the application class. The CameraApplication class has access to all non-pure virtual functions, and so, I do not have to overwrite them. Instead, the CameraApplication class will use the code that was already written for those methods. This gives me the ability to create different applications, that derive from the Application class, with different functionalities if desired.

**Camera class used by CameraApplication class**

**Pre-conditions:** The CameraApplication class must be created and deriving from the Application class. You will also need to have linked the GLM libraries to the project.

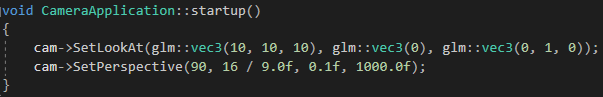
Now, create the Camera class itself. The camera class should resemble the following UML:



To allow the CameraApplication class to use the Camera class, include the Camera class’s header file in the CameraApplication header file. Add a public Camera in the CameraApplication header and assign it as a new Camera.



In the CameraApplication’s startup function I set the camera’s viewTransform member variable by calling setLookAt. Next, I set the camera’s projectionView member variable by calling SetPerspective, which will give the camera a perspective view.

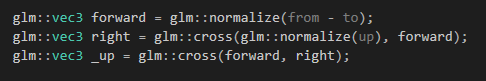


**SetLookAt implementation without use of glm**

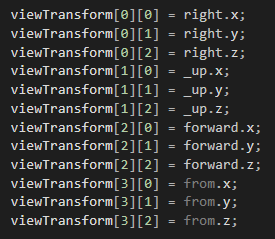
**Preconditions:** There is a function called “SetLookAt”. It should have a return type of glm::mat4 because this function returns a 4x4 matrix. The function takes in three glm::vec3s. The first vec3 is called “from”, and it is the position of the camera. The second vec3 is called “to” and it is the direction to look in. The last vec3 is called “up” and it is a normalized vector that points upwards.



Within the function, you will need to create a vec3 which we will call “forward”. Assign forward the normalized value of from minus to. Next, create a vec3 called “right”, and assign it the cross product of the up variable normalized and forward. Then create another vec3 called “\_up” and assign it the cross product of forward and right.



Now, we assign every index in the camera’s viewTransform as shown in the following picture:



Finally, return the viewTransform.

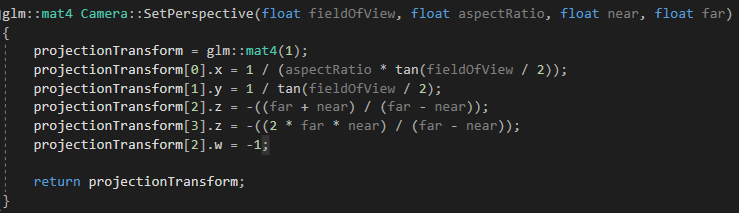
**Perspective Projection implementation without use of glm**

**Preconditions:** There is a function called “SetPerspective”. It should have a return type of glm::mat4, because this function returns a 4x4 matrix. The function takes in four floats for the field of view, aspect ratio, near clip plane, and far clip plane respectively.

The perspective projection implementation uses a field-of-view and aspect ratio to give the camera a 3-D view by distorting the space to fit the clip space.

To set up a camera with a perspective view, you will need to do calculations in the correct indices of the camera’s projectionTransform matrix. When finished with the calculations, return the projectionTransform.

The following picture shows the calculations to build the perspective view matrix:



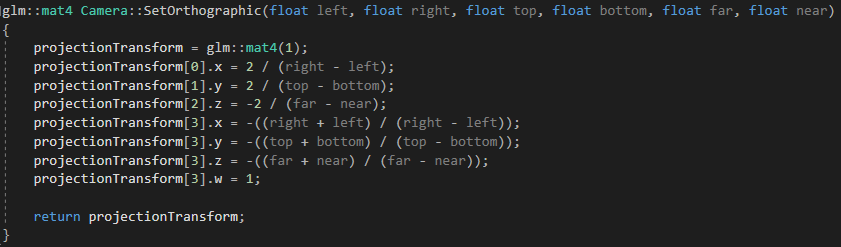
**Orthographic Projection implementation without use of glm**

**Preconditions:** There is a function called “SetOrthographic”. It should have a return type of glm::mat4, because this function returns a 4x4 matrix. The functions takes in six floats. The first four floats are for the left, right, top and bottom sides of the camera. The last two floats are for the far and near clip planes.

The orthographic projection implementation gives the camera a rectangular, tunnel vision projection that has no field of view.

To set up a camera with an orthographic view, you will need to do calculations in the correct indices of the camera’s projectionTransform matrix. When finished with the calculations, return the projectionTransform.

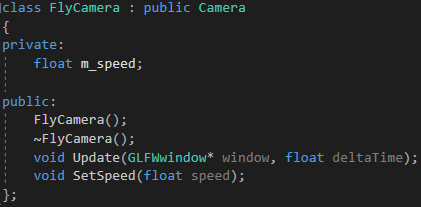
The following picture shows calculations for a way to build the orthographic projection matrix:



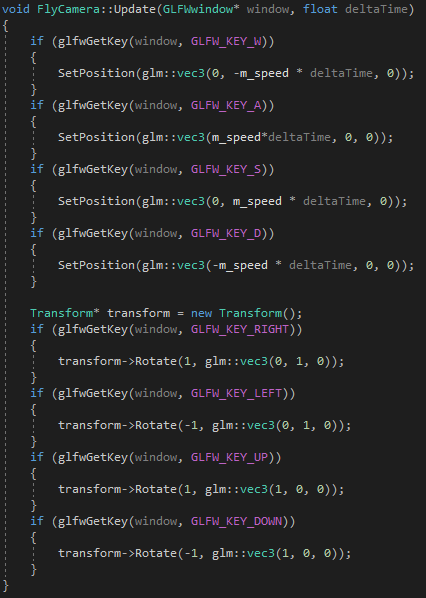
**“Fly-Camera” implementation using translation and rotation**

**Preconditions:** The camera class is set up and the GLM libraries and glfw3.h header file are linked to the project. For rotations, a Transform class should be created that takes in a float for radius and a glm::vec3 for the axis to rotate on, or you can use glm::rotate.

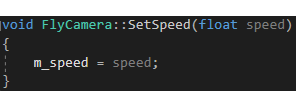
The fly camera is the camera that will be used to move around the scene. To implement a fly-camera, create a new class. This class will be called “FlyCamera”, and it will inherit from the Camera class. Inheriting from the camera class will allow the fly-camera to have the basic camera functions while allowing us to add functions specific to the fly-camera.

The fly-camera has a private member variable called “m\_speed”.

The FlyCamera class should have an Update function and a SetSpeed function, and both are public functions.

The Update function’s return type is void and it takes in two arguments. The first argument is a GLFWwindow pointer for the window. The GLFWwindow type comes from the glfw3.h file. The second argument is a float for delta time. This function is used to move the camera around the scene by checking for user input. By using glfwGetKey and passing in the window being rendered and a keycode as the arguments, we can create conditions to check for in order to move the fly-camera. If a condition is met, we can use the camera’s SetPosition function to translate the camera to a new location. We can also use a transform class or glm::rotate for rotating.

The SetSpeed function’s return type is also void and it takes in one argument. The argument is a float for the speed the fly-camera will move around the scene. When this function is called, it will simply set m\_speed to the value being passed in.

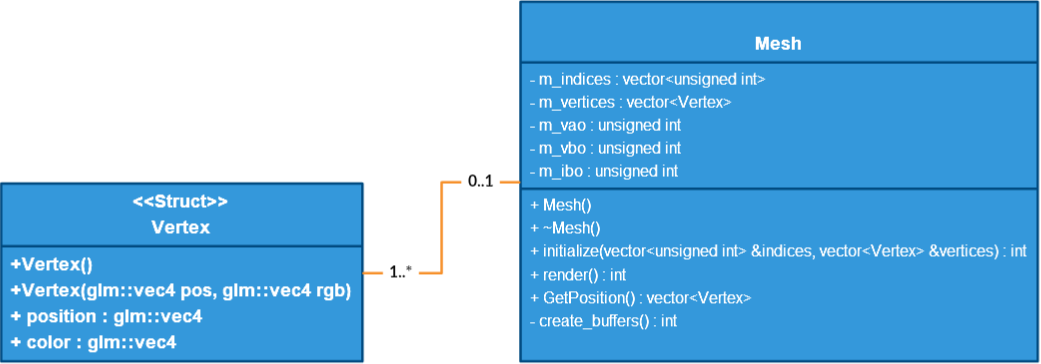


**Rendering Geometry**

**MeshRenderer class**

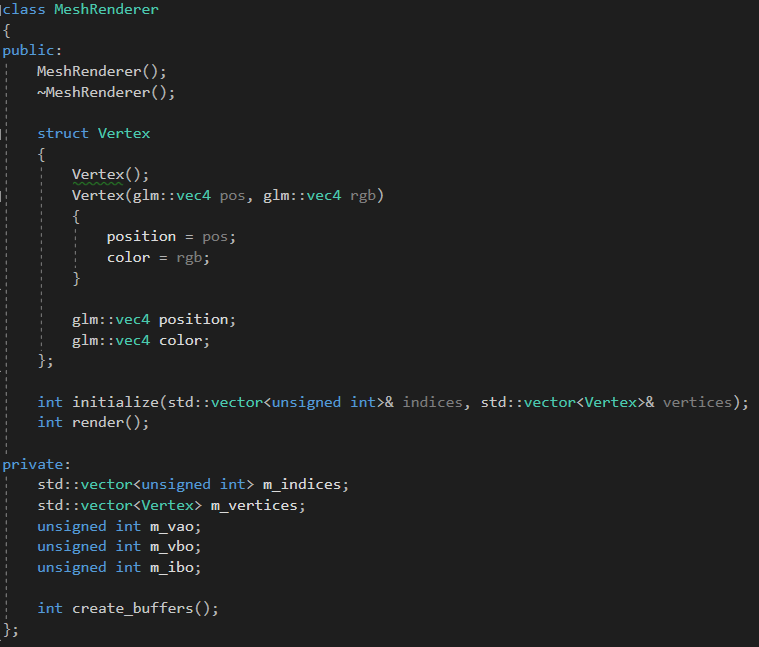
**Precondition:** You must have linked OpenGL loading libraries to your project. The loading library loads pointers to OpenGL functions, which will allow you to use those functions. You need to link the GLM libraries so that you can perform calculations. Include vector in the MeshRenderer header file. Also include

The MeshRenderer class will store all the geometry that is needed for rendering in the window. The MeshRenderer class should resemble the following UML diagram:

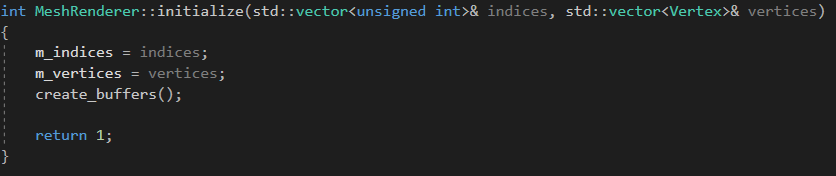


The MeshRenderer class consists of:

* A struct named “Vertex”. Vertex contains a glm::vec4 for position and a glm::vec4 for color. It has an overloaded constructor with both arguments being glm::vec4s for assigning the position and color on declaration.
* A vector of type unsigned integer called “m\_indices” for storing the order to connect the indices.
* A vector of type Vertex called “m\_vertices” for storing vertex information like position and color.
* An unsigned integer called “m\_vao” for the vertex attribute object.
* An unsigned integer called “m\_vbo” for the vertex buffer object.
* An unsigned integer called “m\_ibo” for the index buffer object.
* A function called “create\_buffers” with a return type being an integer for generating and binding vertex and index buffers.
* A function called initialize with a return type of integer that takes in a vector of type unsigned integer for the indices and a vector of type Vertex for the vertices, which passes them both by reference.
* A function called “render” with a return type of integer that is responsible for drawing the elements.



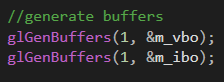
The initialize function assigns the first passed in argument to m\_indices and the second passed in argument to m\_vertices. Then it calls the function to create\_buffers to generate and bind the buffers for drawing. Lastly, 1 is returned if the function executed successfully.



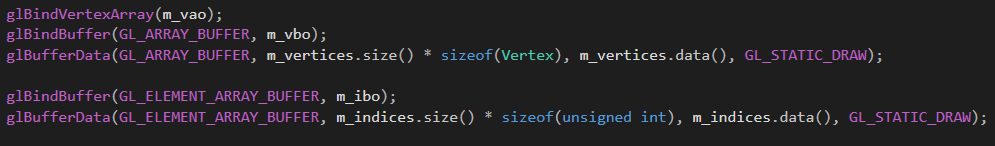
The create\_buffers function generates and binds the buffers that will be used for rendering. First, generate the vertex array by calling glGenVertexArrays and pass in m\_vao.



Next, generate the buffers by calling glGenBuffers twice, and then passing m\_vbo into the first and m\_ibo into the second.



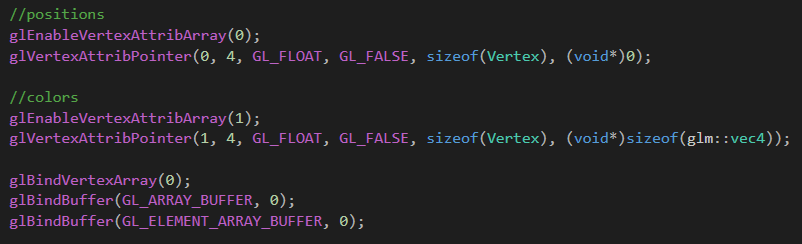
Now bind the vertex array object and vertex buffer object. Use the glBufferData function to buffer the data for the vertices. Next, bind the index buffer object, and then buffer the indices data with glBufferData.



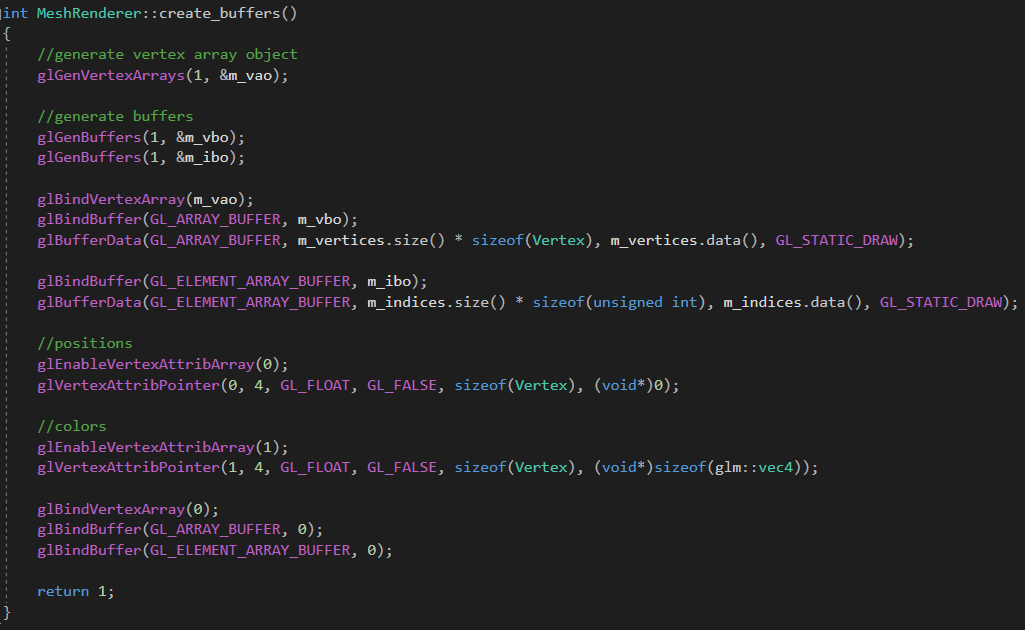
To use a shader, you must make a way to send data. First, use glEnableVertexAttribArray and put a zero as the argument since this is the first element we are enabling. Use glVertexAttribPointer to send the data. This data will be the position of the vertex.

Next, call glEnableVertexAttribArray again and this time pass in a 1 as the argument since this is the second element we are enabling. Use glVertexAttribPointer to send the color data.

Finally, call glBindVertexArray and pass in 0, call glBindBuffer and pass in GL\_ARRAY\_BUFFER, and call glBindBuffer and pass in GL\_ELEMENT\_ARRAY\_BUFFER.



When finished return 1.

**Complete code for create\_buffers:** 

The render function draws the geometry onto the screen.

- Bind the m\_vao by calling glBindVertex array and pass it in as an argument.

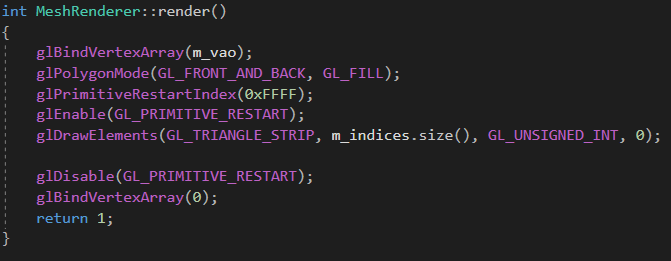
- Set the PolygonMode with glPolygonMode.

- Set the primitive restart by calling glPrimitiveRestartindex.

- Enable the primitive restart with glEnable.

- Set the drawing mode with glDrawElements. For now we will be drawing using triangle strips. Pass in the size of the indices vertex to set the amount of elements to draw.

- Once that’s done, disable GL\_PRIMITIVE\_RESTART, bind the vertex array to 0, and return 1.



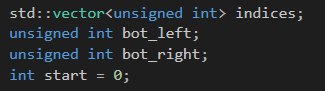
**Generating indices for geometry to be rendered using triangle strips**

**Preconditions:** Include std::vector in the project.

To generate indices for use with rendering geometry using triangle strips, we will create a function and call it genIndices. This function will have a return type that is a vector of type unsigned int. It takes in two integers as arguments. The first for the number of points and the second is for the number of meridians.

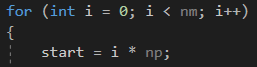


We begin by creating a few variables. You need a vector of type unsigned integer for the indices, a variable of type unsigned integer for the bottom left index, and a variable of type unsigned integer for the bottom right index. Lastly, you’ll need a variable of type integer that we will call “start”.

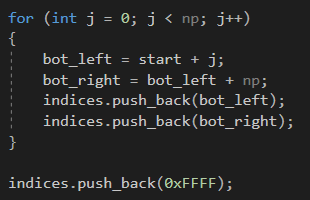


Now we can begin creating the indices. We will need to create a nested for loop. This for loop will order the indices so that OpenGL can correctly connect the points together to draw the shape.

The first for loop will declare a new variable of type int, ‘i’, and initialize it with zero. The condition will be that ‘i’ is less than the number of meridians, which was passed in. The variable ‘i’ will increment by one every iteration of the loop. Before the next for loop runs, we assign our start variable to be ‘i’ multiplied by the number of points.

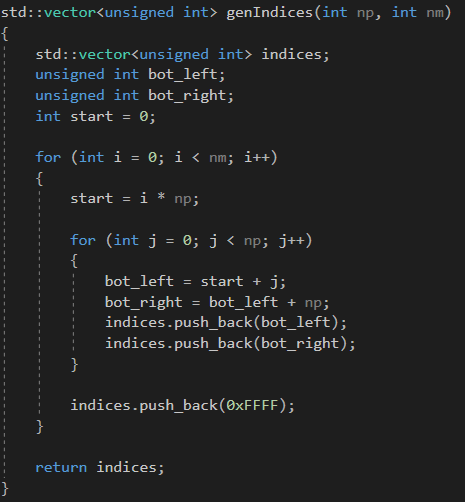


The second for loop will declare a new variable of type int, ‘j’, and initialize it with zero. The condition will be that j is less than the number of points. ‘j’ will increment by one every iteration of the loop. Within this loop the bottom left variable is assigned start + ‘j’. The bottom right variable is assigned the bottom left variable + number of points. The indices vector will then push back the bottom left variable and then it will push back the bottom right variable. Once the second for loop finishes executing, push back “0XFFFF”.



Once the for loops are finished executing, return the vector of indices.

**Completed Code:**

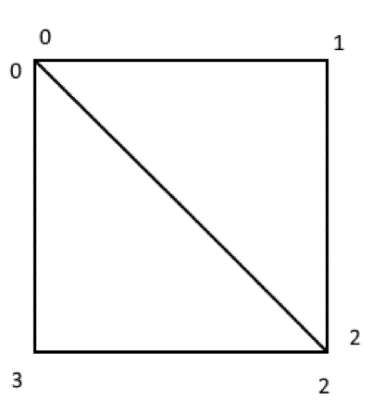


**Render a plane with pre-defined vertex information**

**Preconditions:** Include std::vector in the project. Also, the MeshRenderer class should have the Vertex struct created and the initialize and render functions should be defined. You will also need to link the GLM libraries to the project.

To render a plane with pre-defined vertex information, create a function with a void return type and call it “genPlane”.

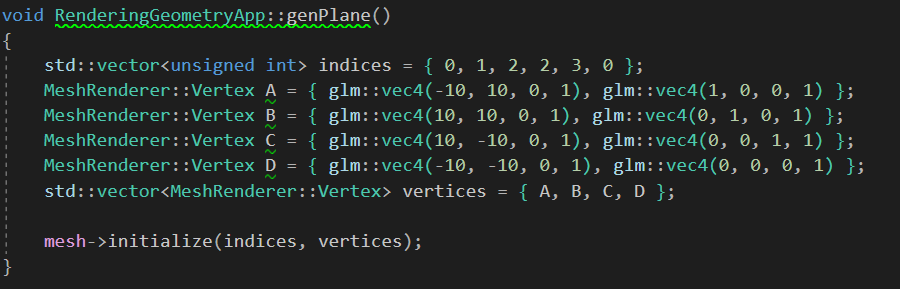
Within the function, create the indices for the plane by manually inputting them. A plane made of triangle strips would require two triangles. Therefore we need to list the indices in the correct order they will be drawn. Create a vector of type unsigned integer and initialize it with the numbers: 0, 1, 2, 2, 3, 0.

This picture shows how the triangle strips are connected by indices.

Next, create the vertices for the plane. A plane consists of four vertices, so create four different instances of the MeshRenderer class’s Vertex struct and initialize the arguments with a position and a color.

Create a new vector of type Vertex and call it “vertices”. Initialize this vector with the four Vertex variables that were just created.

Once the vertices are created, call the MeshRenderer class’s initialize function and put the indices and vertices variables as arguments to load in the geometry.



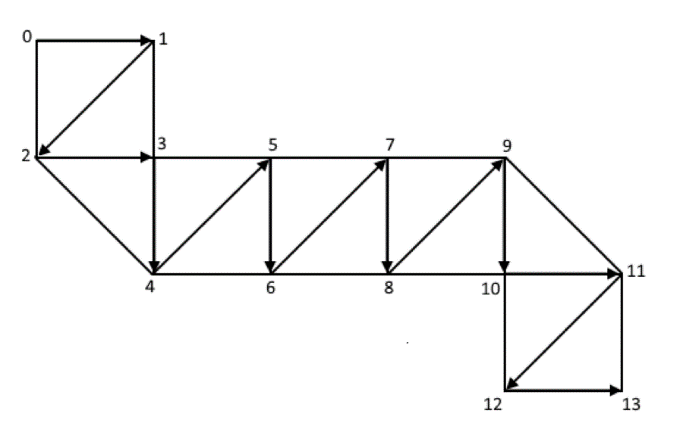
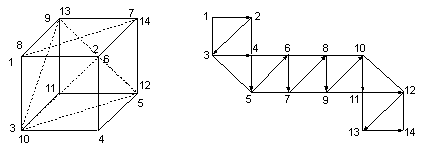
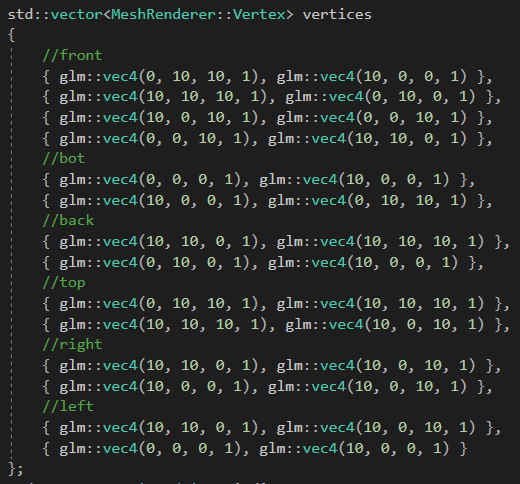
Calling the MeshRenderer class’s render function will draw the plane.

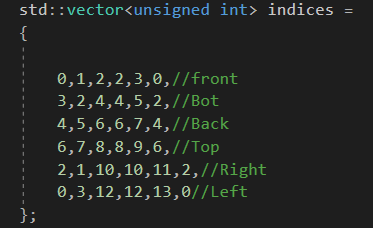


**Rendering a cube with predefined vertex information**

**Preconditions:** Include std::vector in the project. Also, the MeshRenderer class should have the Vertex struct created and the initialize and render functions should be defined. You will also need to link the GLM libraries.

To render a cube with predefined vertex information, create a function called “genCube”. This function creates the vertices and indices and passes the information into the MeshRenderer class’s initialize function.

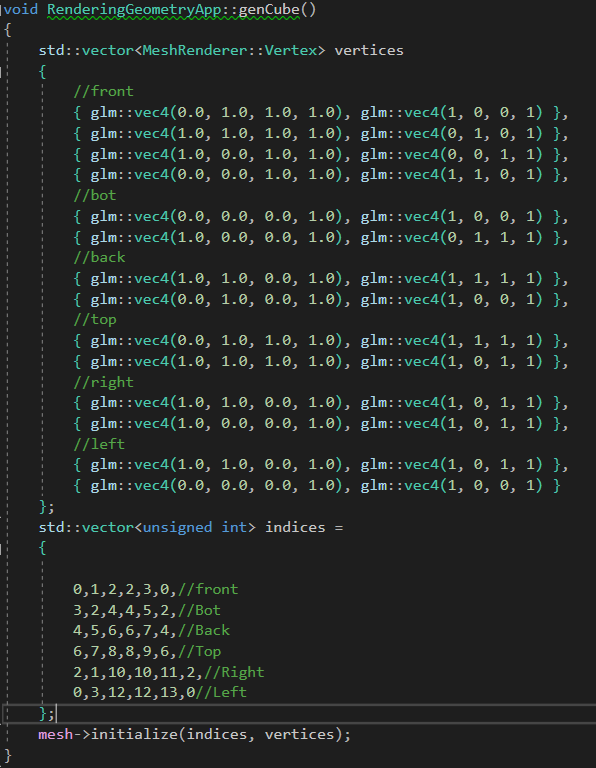
[](https://www.google.com/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&ved=2ahUKEwiHodzd6pHeAhWGv1MKHV2TCJ8QjRx6BAgBEAU&url=https://scs.senecac.on.ca/~chris.szalwinski/archives/gam666.063/content/basic.html&psig=AOvVaw1QQ4XQjFA5dV8GT1UiMspi&ust=1540015400837760) A cube is made up of six different planes so, first, I create the vertices. I created a vector of type Vertex and called it vertices. I initialize vertices on declaration. I will need to create fourteen different instances of Vertex. The first four vertices will be the plane’s front face. I will need to create two more vertices for every other side. I only need two vertices for the other sides, because the indices will connect the two vertices to two other vertices that already exist on the plane’s front face to create another side of the plane. 

Once I finish creating the vertices, I then create the indices. Since a cube is different from a sphere and only requires a set number of points, I can manually put the indices in order that they should be connected. To do this I create a vector of unsigned int and then initialize it with the order to connect the points.

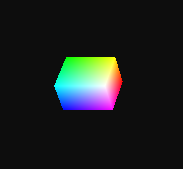
Call the MeshRenderer class’s initialize function and pass in the vertices and indices variables.

Calling the MeshRenderer class’s render function will render the cube.

**Finished Code:**



**Rendered Cube**



**Generating a half-circle given a number of points and a radius**

To generate a half-circle given a number of points and a radius, create a function called “genHalfCircle” and have it take in two integers as arguments. The first integer is for the number of points and the other is for the radius. The functions return type is a vector of type vec4.

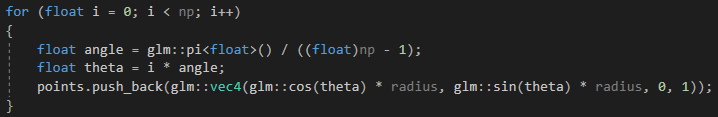


Within this function, declare a vector of type vec4 variable. This variable, which we will call “points” will store the points generated to create the half circle.



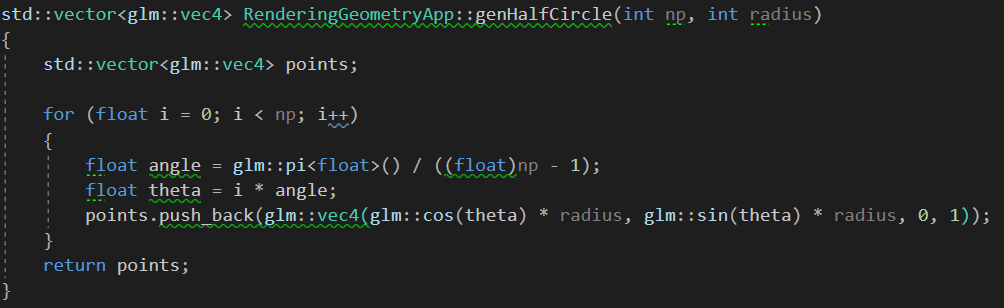
Next, create a for-loop. The for-loop initializes a variable of type float and assigns it zero. The condition is that the variable is less than the number of points passed into the function. The variable should increment by one each iteration of the loop.

Within the for-loop, declare a new float called angle, and assign it pi divided by the number of points minus one. Next, create another float called theta and assign it the for-loop’s variable multiplied by angle. Finally, push back a new ve4 onto the points vector. The vec4’s x value will be the cosine of theta multiplied by the radius passed in. The vec4’s y value will be the sine of theta multiplied by the radius. The last two values will be zero and one respectively.



After the for-loop runs, the function will return the points vector.

**Finished Code:**



**Generating a sphere given a half circle and number of meridians**

**Preconditions:** Link the GLM library to the project. The MeshRenderer class’s functions should be defined. Include std::vector in the project.

To generate a sphere when given a half circle and a number of meridians, create a function called “genSphere” with a return type being void. This function will take in three integers. The first integer is for the radius of the sphere. The second integer is for the number of points. The third integer is for the number of meridians.

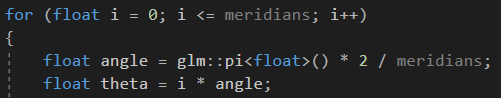


First, declare two vectors of type vec4. One vector will be named “points” and the other vector will be named “totalPoints”. The first variable “points” will be assigned the returned value of the function genHalfCircle. The arguments number of points and meridians arguments being passed into the the genSphere function will be passed into the genHalfCircle function also.



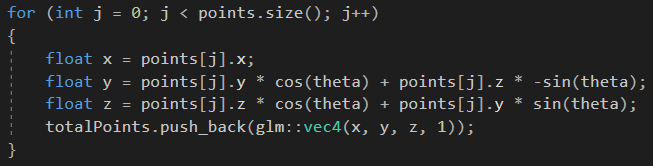
Next, create a for-loop. This for-loop declares a variable of type float called ‘i’ and initializes it with zero. The for loop’s condition is that ‘i’ is less than the meridians passed into the genSphere function. The ‘i’ variable will increment by one every iteration of the for-loop.

Within this for-loop, declare a variable of type float and call it angle. Assign angle with the result of pi multiplied by two divided by meridians. Now declare a variable of type float called theta. Theta is assigned ‘i’ multiplied by angle.



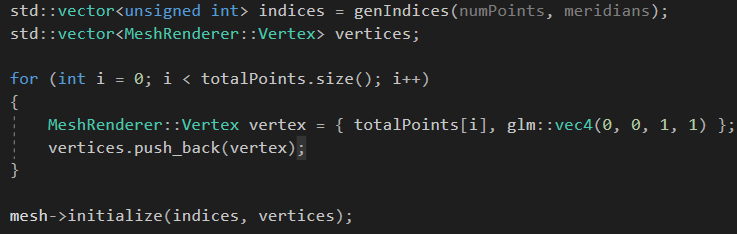
After declaring angle and theta, create another for loop within the current for loop. This new for loop declares a variable of type float called ‘j’ and initializes it with zero. This for-loop’s condition is that ‘j’ is less than the size of the points vector. Every iteration of the loop will increment ‘j’ by one.

This new for-loop is going to create all of the points that will make up the sphere by rotating the points of the half circle. To do this, declare three new variables of type float. The first variable, called ‘x’, will be assigned the x value of points[j]. The second variable, ‘y’, will be assigned the sum of points[j] y value multiplied by cosine theta and points[j] z value multiplied by negative sine theta. Once the point has been rotated, push back a new vec4 initialized with the arguments being the variables x, y, z, and the number 1, onto totalPoints.

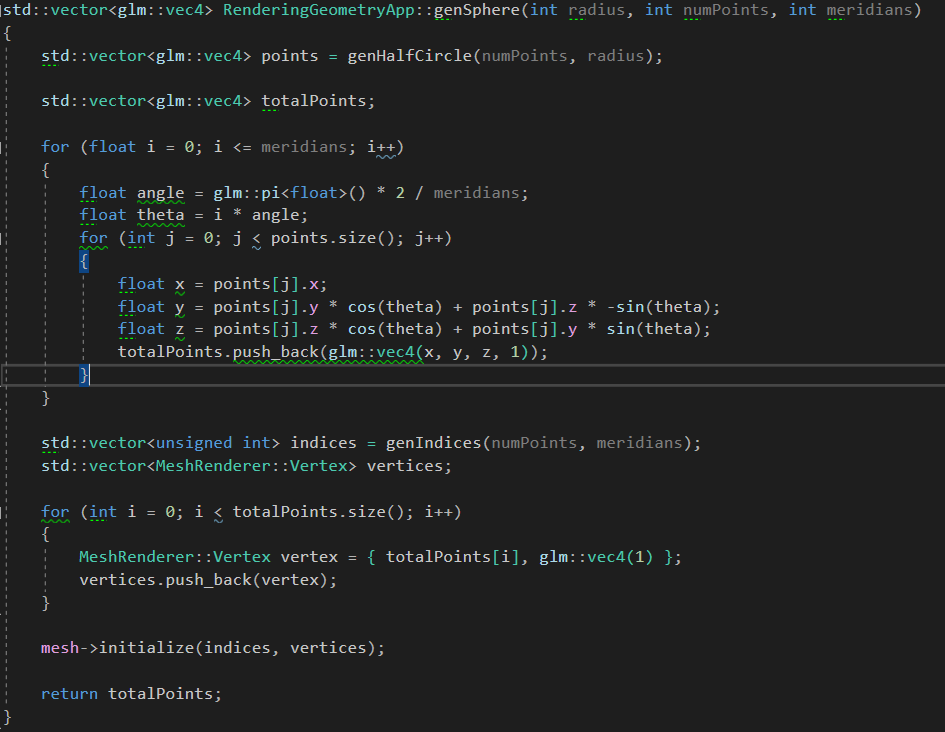


The totalPoints variable contains all of the points needed to generate a sphere.

Using the passed in number of points and number of meridians arguments call the genIndices function and assign it to a new variable of type vector of type unsigned int. Next, I create a vector of the MeshRenderer class’s Vertex struct and call it “vertices”. A for-loop is used to loop through all of the points in totalPoints and creates a new Vertex using the current index of totalPoints and a vec4 for color. After creating the new vertex, push back the vertex onto vertices. Once the for-loop is finished running, call the MeshRenderer class’s initialize function and pass in the indices and vertices variables as arguments. Calling the MeshRenderer class’s render function will draw a sphere using the loaded in geometry.



Finally, return totalPoints.

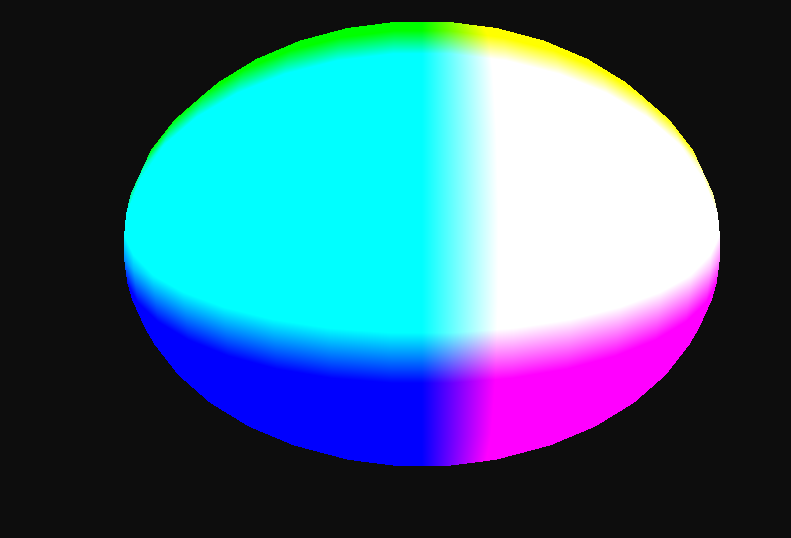
**Full Code:**

**Using triangle strips to render a sphere**

**Preconditions:** The MeshRenderer class’s render function is defined, the MeshRenderer class’s initialize function is defined, and the genSphere function is defined.

To render a sphere with triangles strips all you have to do is use the glDrawElements function which should be in the MeshRenderer class’s render function. The first argument of glDrawElements should be “GL\_TRIANGLE\_STRIP”. This will draw using triangles. Now all you need to do is call the genSphere function and the MeshRenderer’s draw function.



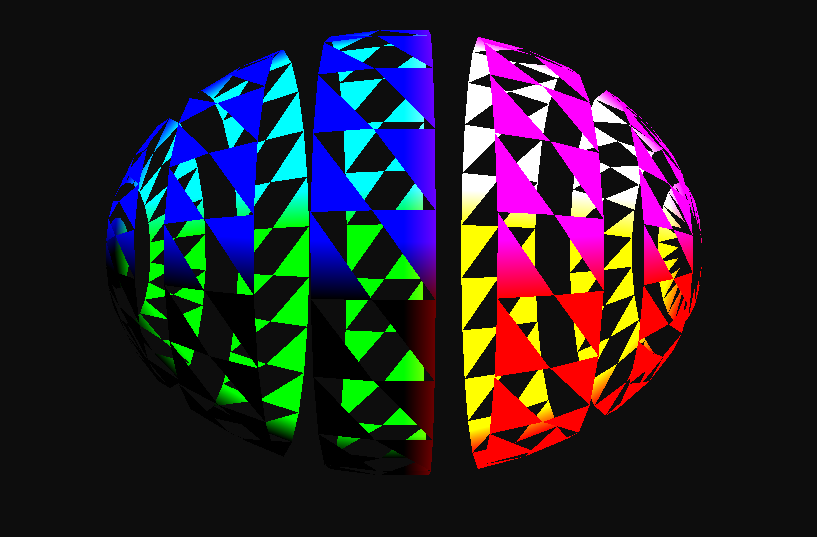


**Using Triangles to render a sphere**

**Preconditions:** The MeshRenderer class’s render function is defined.

To render a sphere with triangles all you have to do is use the glDrawElements function which should be in the MeshRenderer class’s render function. The first argument of glDrawElements should be “GL\_TRIANGLES”. This will draw using triangles. Now all you need to do is call the genSphere function and then the MeshRenderer class’s render function.

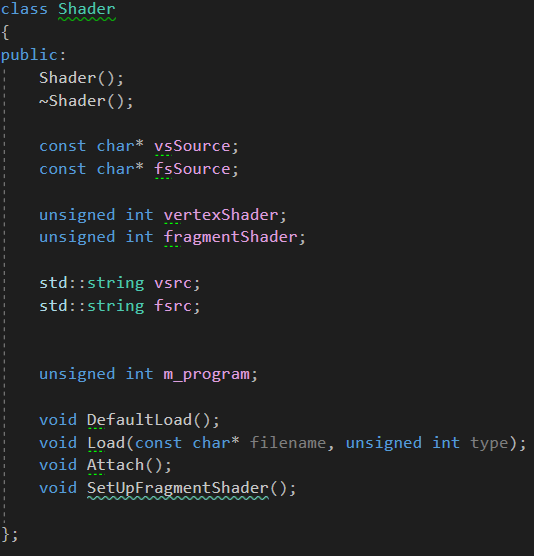




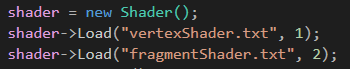
**Ability to load shaders from file using a Shader class object.**

This is a picture of my shader class. This class contains:

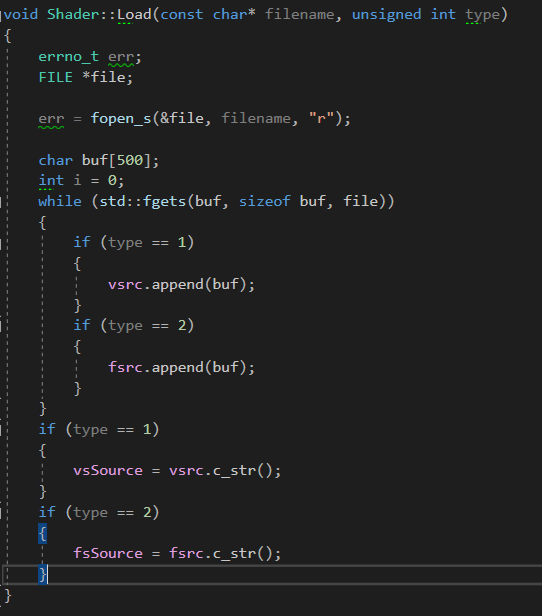
* Constant char pointers for the vertex and fragment shader file sources
* Unsigned integers for the vertex and fragment shaders
* Strings to store all of the text in the vertex and fragment shader file sources
* Unsigned integers for the shader program



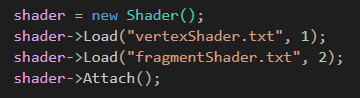
Load should be called twice. Once to load the vertex shader, and again to load the fragment shader. If a ‘1’ is passed in as a second argument, then a vertex shader will attempt to be loaded. If a ‘2’ is passed in as the second argument, then a fragment shader will attempt to be loaded.



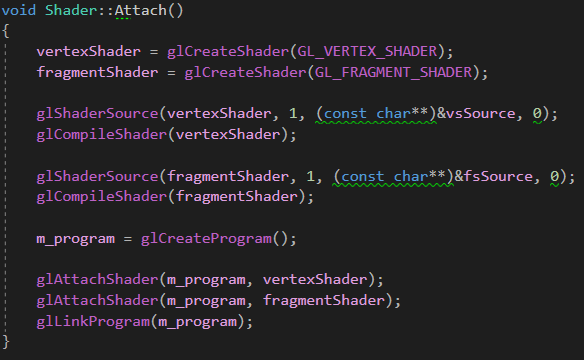
When Load is called, the file is read and stored in a variable. The shader class’s “vsrc” and “fsrc” variable store the file’s contents. Depending on the unsigned integer passed in, the shaders vsource or fsource member variables will be assigned the contents of the file.



Once both vertex and fragment shaders are loaded, I call the shaders Attach function to compile the shaders.



The Attach function compiles the information stored in the vsource and fsource member variables. A new program is created. This program is the shader program with the vsource and fsource shaders compiled. Once the program is created the shaders are attached to it and the program is linked to the whole application.



**Direct Lighting**

**Ambient lighting implementation**

The way to calculate ambient light is simple. We have to multiply the objects surface color by the color of the light.

To do this, we must first, in our program, create a handle for the lights color and send it to the fragment shader. First, in your fragment shader, declare a new uniform variable of type vec4. Now, in the rendering program, create a handle for the light’s color using glGetUniformLocation and pass in your shader’s program and the name of the uniform variable that was created earlier. Next in the program, create a variable of type vec4. This variable will be for the color of the light. We will use this variable to send information to the shader. Initialize this variable with a color for the light. Once the variable for the lights color is created and initialized, use glUniform4fv and pass in the handle created for the light’s color as the first argument, and then pass in the address of the first index of the light’s color variable. This function will send the information to the shader.

Now, going back to the fragment shader, we will use the uniform variable created in order to calculate the ambient light. In the main function of the fragment shader, create a vec4 variable and call it “ambient”. Assign ambient to lightColor. Finally, assign the fragColor to vColor \* ambient.

**Diffuse lighting implementation**

To calculate the diffuse light, you will need to create a uniform variable in the fragment shader. So, in the fragment shader, declare a uniform variable of type vec3 and call it “lightDirection”. In order to send information to this uniform variable, we need to create a handle and send information to the handle from the rendering program.

In the rendering program, create a variable of type int. This will be the handle for the light’s direction. Initialize the handle using glGetUniformLocation and pass in the shader’s program and the name of the uniform variable that was created in the fragment shader.

Next, create a variable of type vec3. This variable will be the light’s direction. Initialize this variable with a vec3.

Once the handle and the variable for the light’s direction are created, we must now send the information to the fragment shader. To do this, use the function glUniform3fv and pass in the handle as the first argument and the reference to the first index of the variable.

Now that the information is sent to the fragment shader, we can do the calculations to get the diffuse lighting. In the shader file, create a vec3 variable and call it “vertexNormal”. Initialize this variable by calling normalize and pass in the vPosition.xyz. Next, create a variable of type float and call it “lambert”. This variable will be the Lambertian term. Assign this variable by calling the dot function and pass in -lightDirection and vertexNormal. Finally to calculate the diffuse term, create a new variable of type vec4. This variable will be assigned vColor multiplied by lambert multiplied by lightColor.

To add the diffuse to the FragColor assign it the product of vColor times the sum of ambient plus diffuse.

**Specular Lighting Implementation**

To calculate the specular light, you will need to create two uniform variables in the fragment shader. In the fragment shader, declare two uniform variable of type vec3 and call them “lightPosition” and “cameraPosition”. To send information to the uniform variables, we need to create handles and send information to the handles from the rendering program.

In the rendering program, create two variables of type int. These will be the handles for the light’s position and camera’s position. Initialize the handles using glGetUniformLocation and pass in the shader’s program and the name of the uniform variable that was created in the fragment shader.

Next, create a variable of type vec3. This variable will store the light’s position. Initialize this variable with a vec3. You will also need to do the same for the camera’s position.

Once the handles and the variables for the light and camera positions are created, we must now send the information to the fragment shader. To do this, use the function glUniform3fv and pass in the handle as the first argument and the reference to the first index of the variable for both.

Now that the fragment shader contains the information for the light’s position, we can begin calculating the specular light. In the fragment shader, create a float for specular strength. Next create a vec3, call it “viewDir”, and assign it by calling normalize and passing in the cameraPosition minus vPosition.xyz. Then create a new vec3 variable called “reflectDir” and assign it by calling the function reflect and passing in the normalized light position and the normalized vPosition.xyz. Then create a float called “spec”. Initialize spec with the pow function. Pass in the max function with the arguments of being the dot product of viewDir and reflectDir, and 0. Pass in a power of 2 as the second argument of the pow function. The higher the power the sharper the specular light will be. Finally, create a variable of type vec4 and call it “specular”. Initialize specular with a vec4 with all four arguments being 1 multiplied by vColor multiplied by spec.

To add the effects of specular to FragColor, multiply vColor by specular. You can also add specular to ambient and diffuse and then multiplying to add their effects.

**Difference Between Phong and Blinn-Phong**

The difference between Blinn-Phong and Phong, are in how specular lighting is calculated. Instead of using a reflection vector like the Phong model, Blinn-Phong uses what is called a half-way vector which is a unit vector that is halfway between the view direction and the light direction.

The way Blinn-Phong is calculated allows it to be faster than the Phong model due to faster computation.