**Matt Bise**

**3D Game Engine Programming**

**Portfolio**

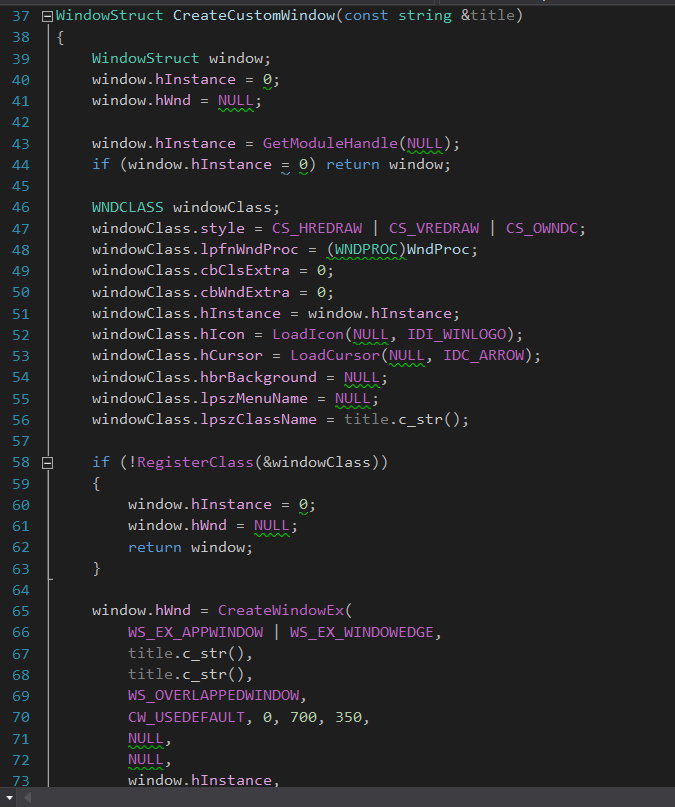
**Lab 01**

Lab 01 was a two part lab. For the first part, the objective was to get GLEW (OpenGL Extension Wrangler Library) up and running and create a basic OpenGL window using C++ and Visual Studio 2015. But before jumping straight into OpenGL calls, we used the Win32 API to create a plain window. We used the WNDCLASS struct to define our window and then, using the CreateWindowEx function, create a new window and retained a handle to it. We learned about the Win32 message passing system used to alert GUI applications of both user and system actions and implemented our own message loop and WNDPROC callback function to handle the messages. After this, we were able to use our window handle to get a device context which was then used to create an OpenGL context using the wglCreateContext function. The message loop was modified to set the size of the OpenGL viewport and swap out the buffers, since we used the double buffering technique. The final goal of the first part was using the glClearColor function to set the background color of our viewport.

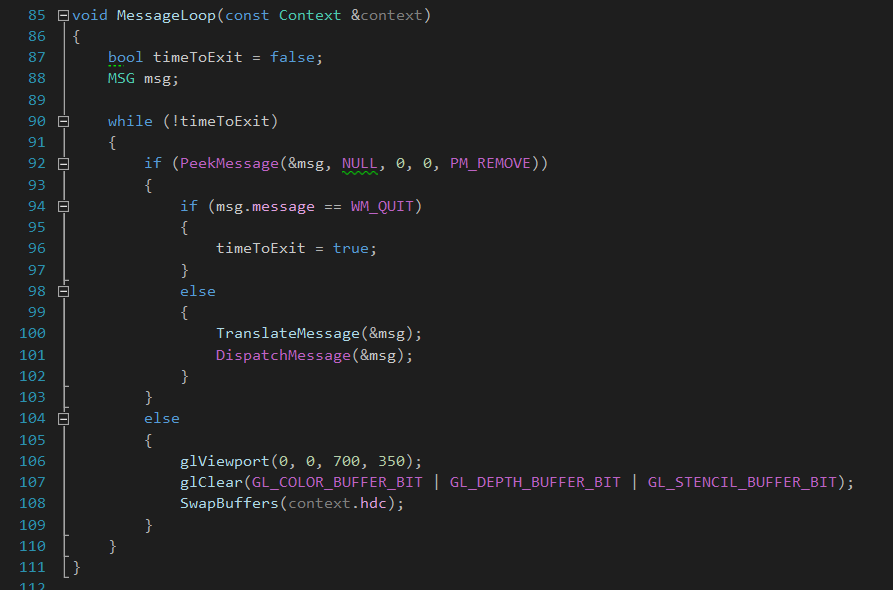
Part 2 of the first lab was focused more on working with files in C++ than working with OpenGL or graphics. The primary goal of this part was to learn about reading data from a text file in a specific format and using it to populate data structures within our application. We would be accomplishing this by creating a “dungeon” built out of characters displayed in a console. We used a struct to store data about each wall, such as the orientation (horizontal or vertical), the starting point, the length, and the character used to create it. We then created a text file using a format for defining one wall per line such that the line 0 0 0 X 20 would define a wall starting at (0, 0), drawn horizontally, using 20 X's. We created a function that utilized the ifstream class to read the file in line by line, while parsing each line and creating a struct represting the corresponding wall. We stored the walls in a vector which would later be written out to the console, wall by wall, using a draw wall function.

**Code Samples:**

Function for creating the Win32 window from part 1:



Message loop from part 1:



Function for initializing the OpenGL window from part 1:

Image 1:

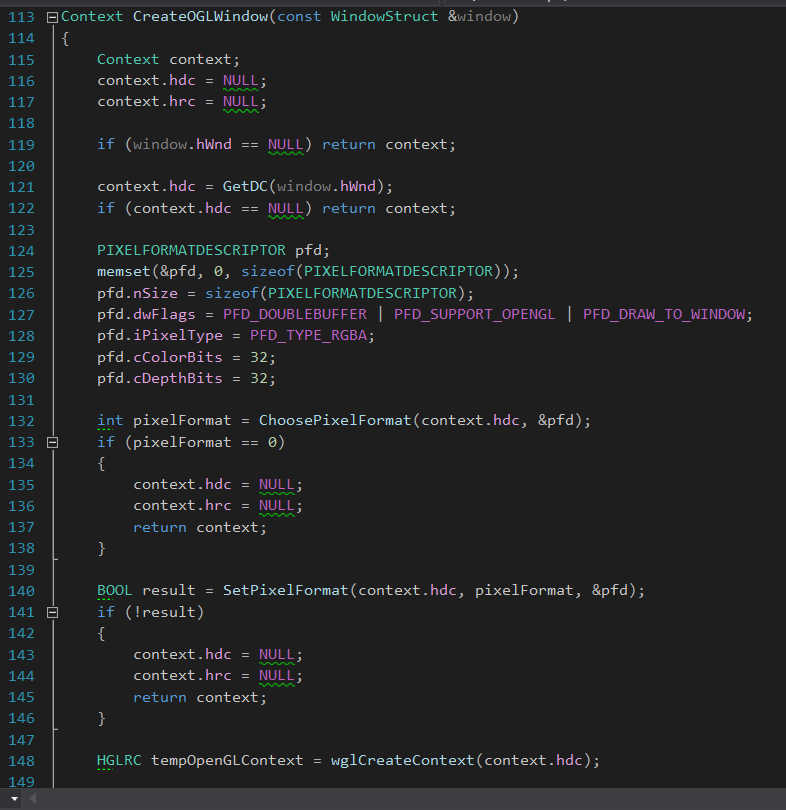
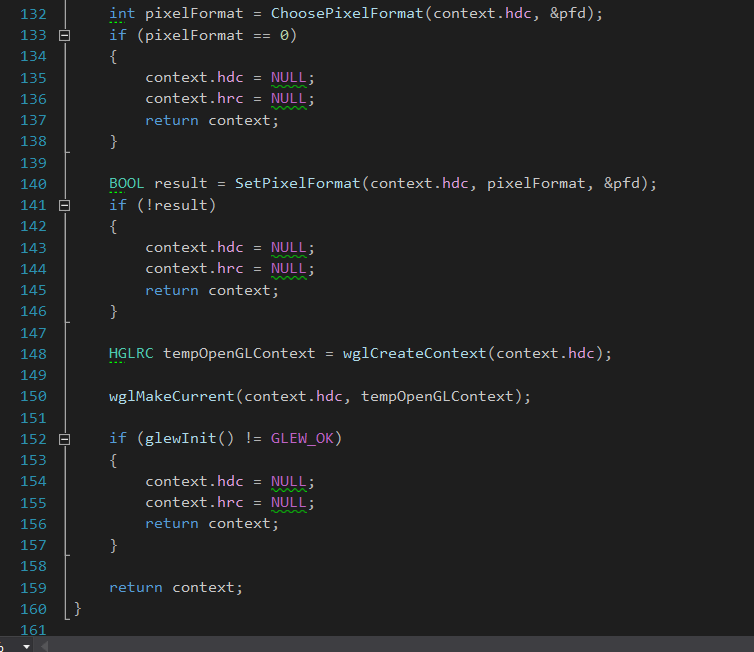
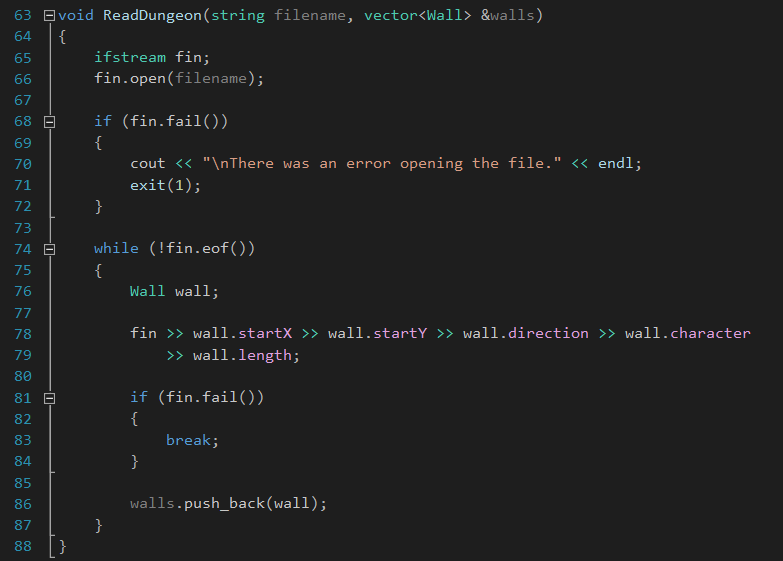


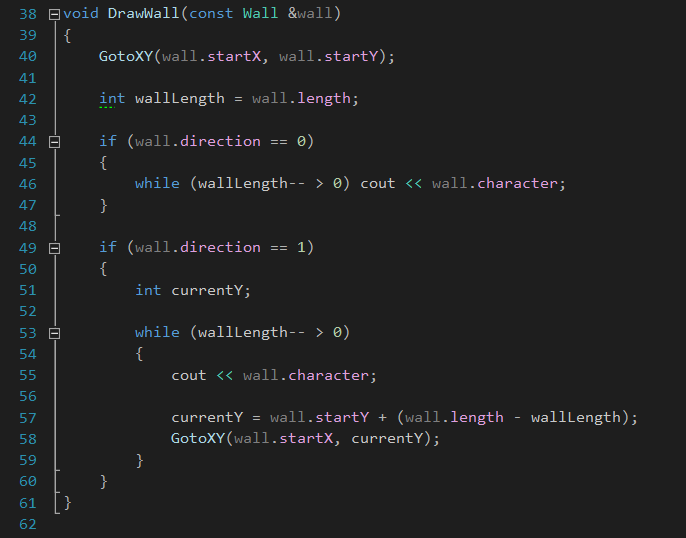
Image 2:





Function for reading the dungeon file from part 2:

Function for writing a wall to the console from part 2:

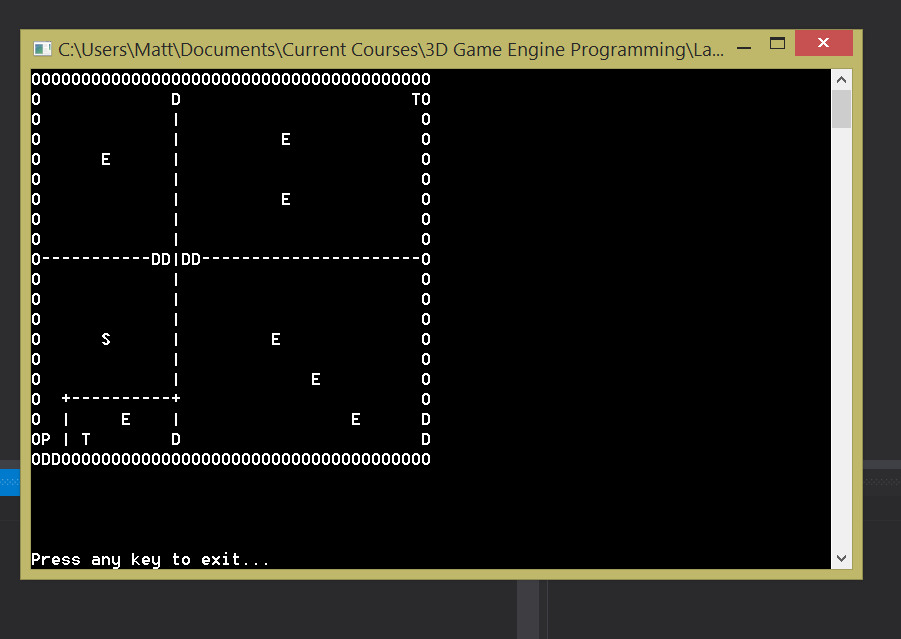


**Screenshots:**

Win32 window with OpenGL viewport and background color set from part 1:



“Dungeon” created from reading in the dungeon text file from part 2:



**Lab 02**

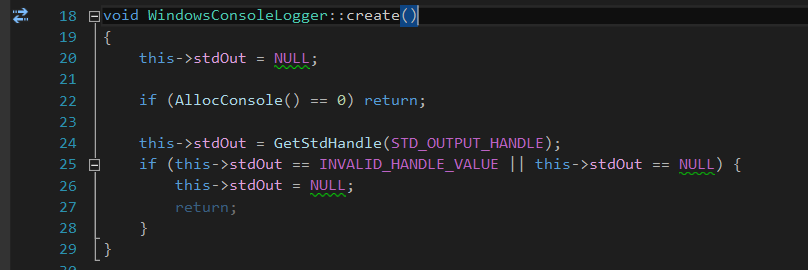
The primary objective of Lab 02 was to implement the basis for the console and presentation subsystems of the game engine. The console subsystem would be responsible for displaying a console window and writing out logging and debugging messages throughout the running time of the game. The presentation subsystem would contain the window class hierarchy and be responsible for creating the game window (Win32 window and OpenGL viewport), responding to Windows messages, and delivering calls to update logic and render graphics.

The logger was fairly simple: it created a new console window and contained overloaded log and debug methods, each of which simple printed out the passed value in the console window (the overloads accepted different types of data to be printed). The hierarchy consisted of a pure abstract Logger class and a WindowsConsoleLogger class that derived from it.

The window hierarchy was slightly more complex. It consisted of the MyGameWindow class which implemented the Win32OGLWindow class which implemented the Win32Window class which, finally, implemented the abstract GenericWindow class. GenericWindow defined the basic create, show, and listenForEvents functions and contained a logger. Its derived classes implemented these functions in their own ways. At the bottom of the hierarchy, MyGameWindow accepted the new window's attributes (title, width, height, background color, etc.) and used them to create a new window with an OpenGL viewport. It also defined a WndProc callback function which specifically checked for key presses. If the F1 key was pressed, “Help” was logged out to the console. If the ESCAPE key was pressed, the application exited. This lab culminated by modifying the class to gradually change the background color between two or more colors, depending on which key was pressed. I did this by maintaining a list of colors to rotate between and, depending on which key was pressed, added the corresponding colors to the list. The actual changing occurred in the update method, which ran once per frame (called from the message loop).

**Code Samples:**

Method to create the console window for logging:



Callback function to handle key presses:

Image 1:

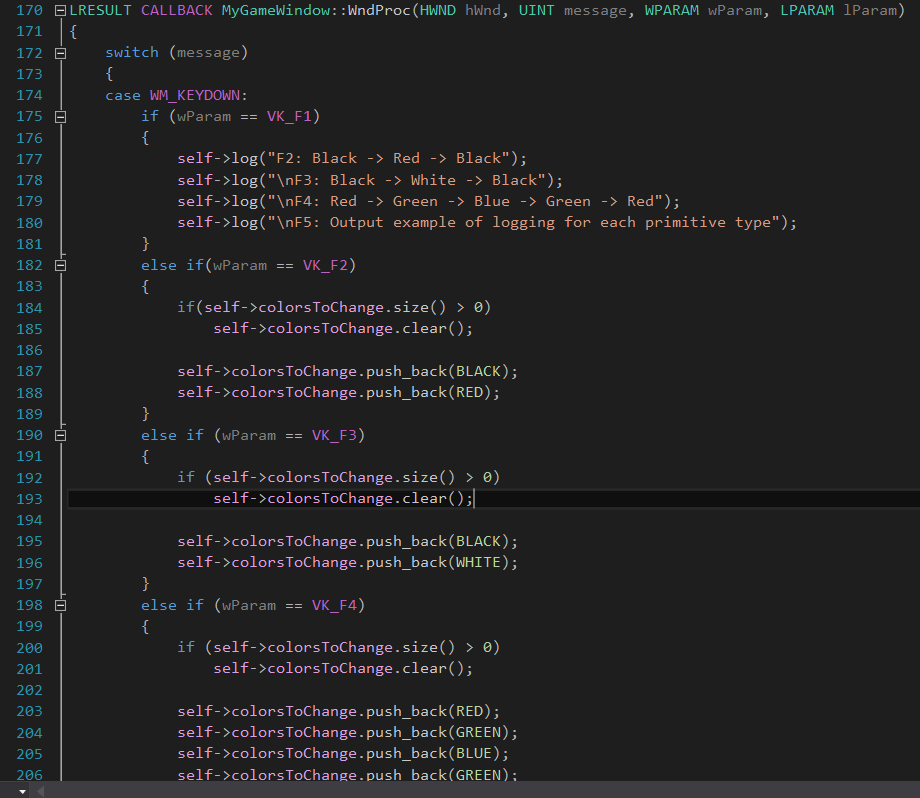


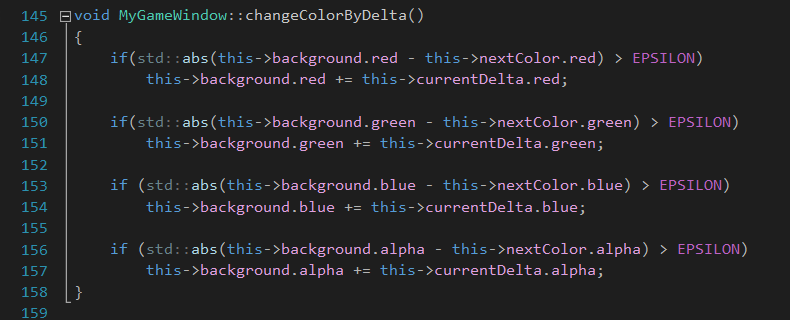
Image 2:

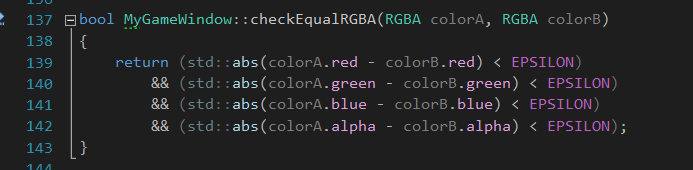


Functions used to change between background colors:

update:

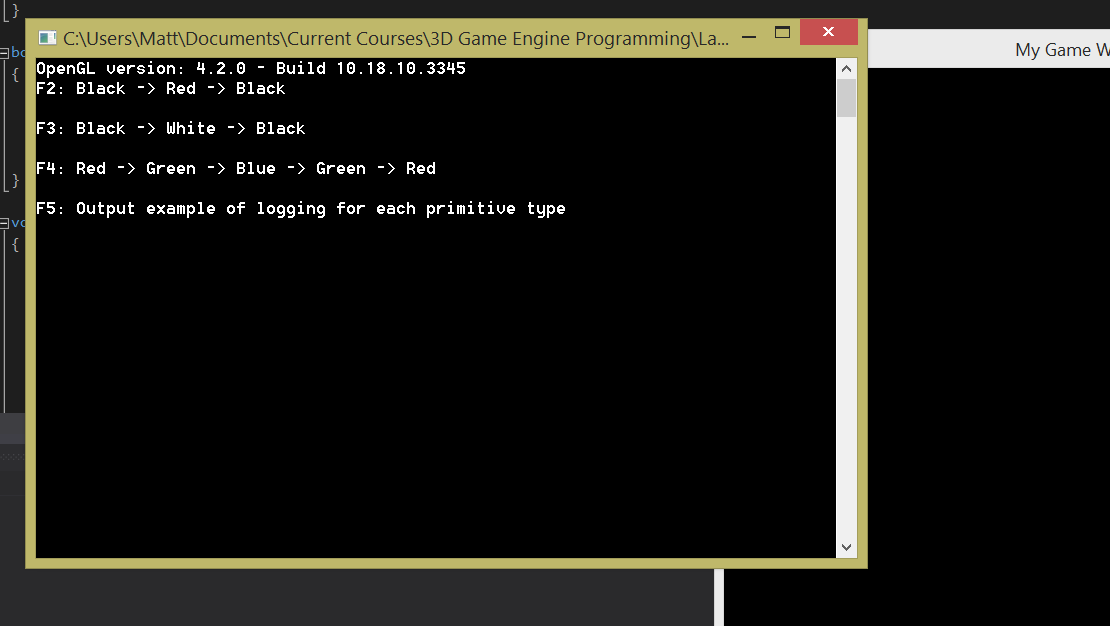
changeColorByDelta:

checkEqualRGBA:



**Screenshots:**

Console logger window with color change options:

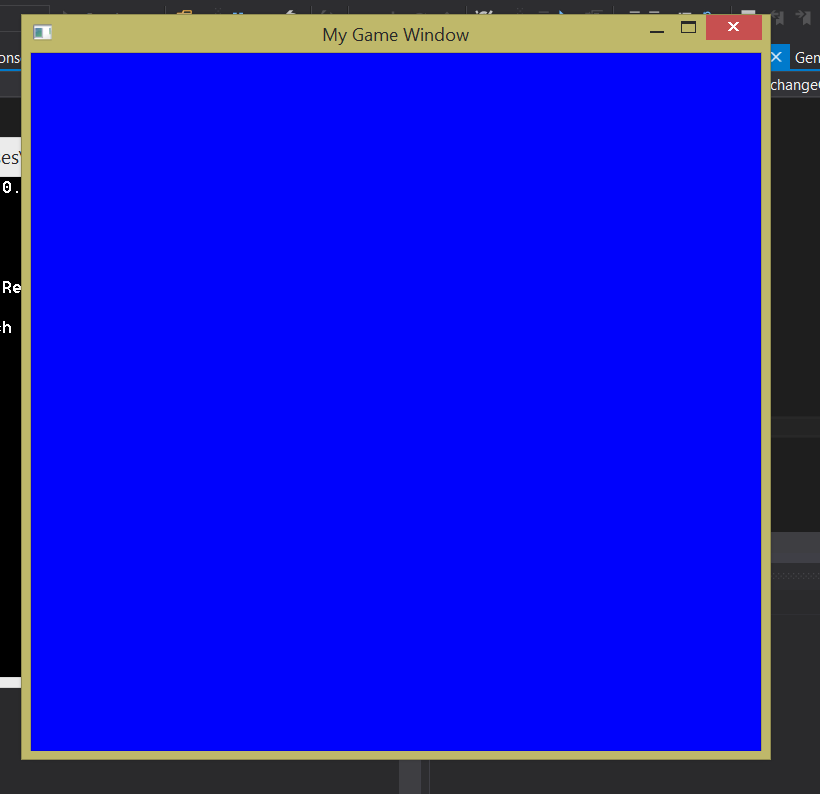


Main Win32 window with OpenGL viewport:

Color 1:



Color 2:



**Lab 03**

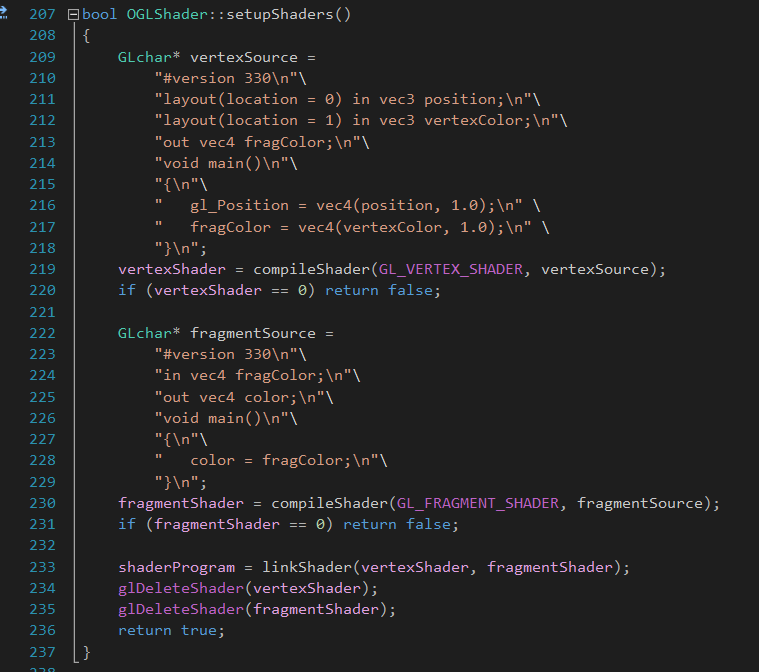
Lab 03 was where we really started using the features of OpenGL and rendering graphics. Its main goal was to provide insight into how rendering works with basic shaders in OpenGL. Initially, the application rendered two perpendicular lines representing the X and Y axis and two triangles, one in the first quadrant and one in the second. At this point, all the shading and rendering logic was contained in the OGLShader class, which inherited from the abstract Shader class, which in itself had a logger and the abstract methods create and renderObjects. The OGLShader class defined a struct which held data corresponding to a single vertex, that data being its X, Y, and Z coordinates and its RGBA color. There were also arrays of these structs which represented vertex data for various points, lines, and triangles. Throughout the lab, we proceeded to add more points, lines, and triangles to the scene using the methods contained within the OGLShader class. The setupShaders method defined the source code for the vertex and fragment shaders as pointers to char arrays. It then compiled the shaders, linked them and stored a handle to the resulting shader program, and deleted the shaders themselves (since we now had the program, the shaders were obsolete). The createPrimitives method first created the vertex array and set it as active, then created a vertex buffer object for each set of object data in the class and bound it to the target using the createVBOHandle method. After this, the corresponding render method could be called for each vertex buffer object in the class, which would set the active shader program, bind the corresponding VBO, and use glDrawArrays to render the object to the screen. As for the placement of the object creation and rendering functionality, the shader (and therefore the objects) were created in the create method of the GameWindow class (which corresponds to the MyGameWindow class from the previous lab). The objects were rendered in the update method of the same class, which was called from the runOneFrame method invoked in the message loop.

**Code Samples:**

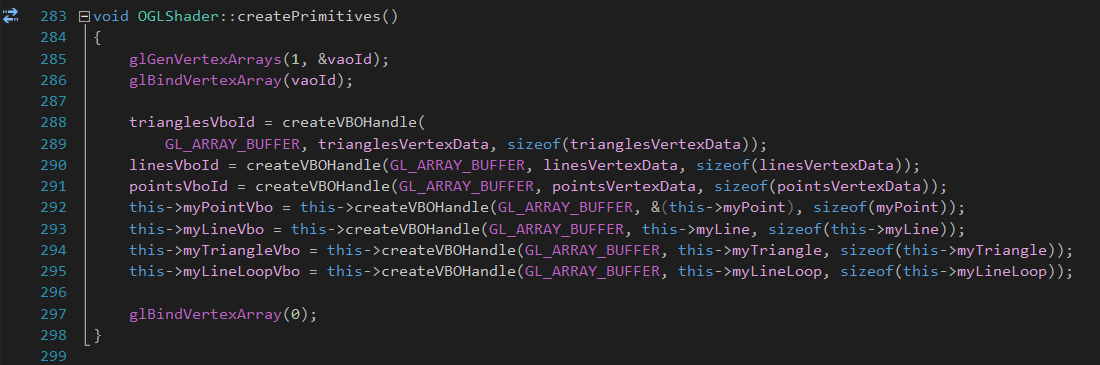
Struct used to define a vertex in the OGLShader class:



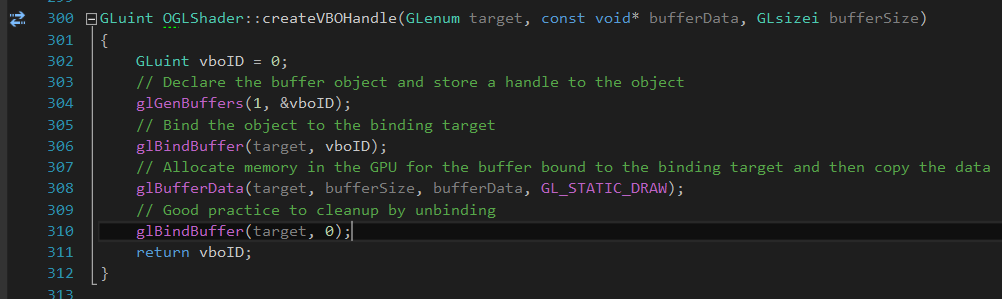
Source code for the vertex and fragment shaders in the setupShaders method:



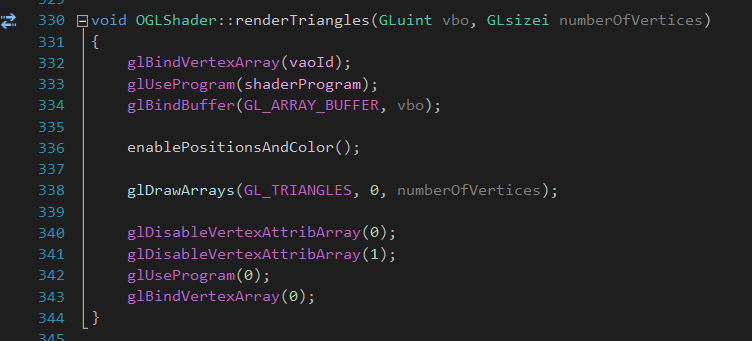
createPrimitives method:



createVBOHandle method:

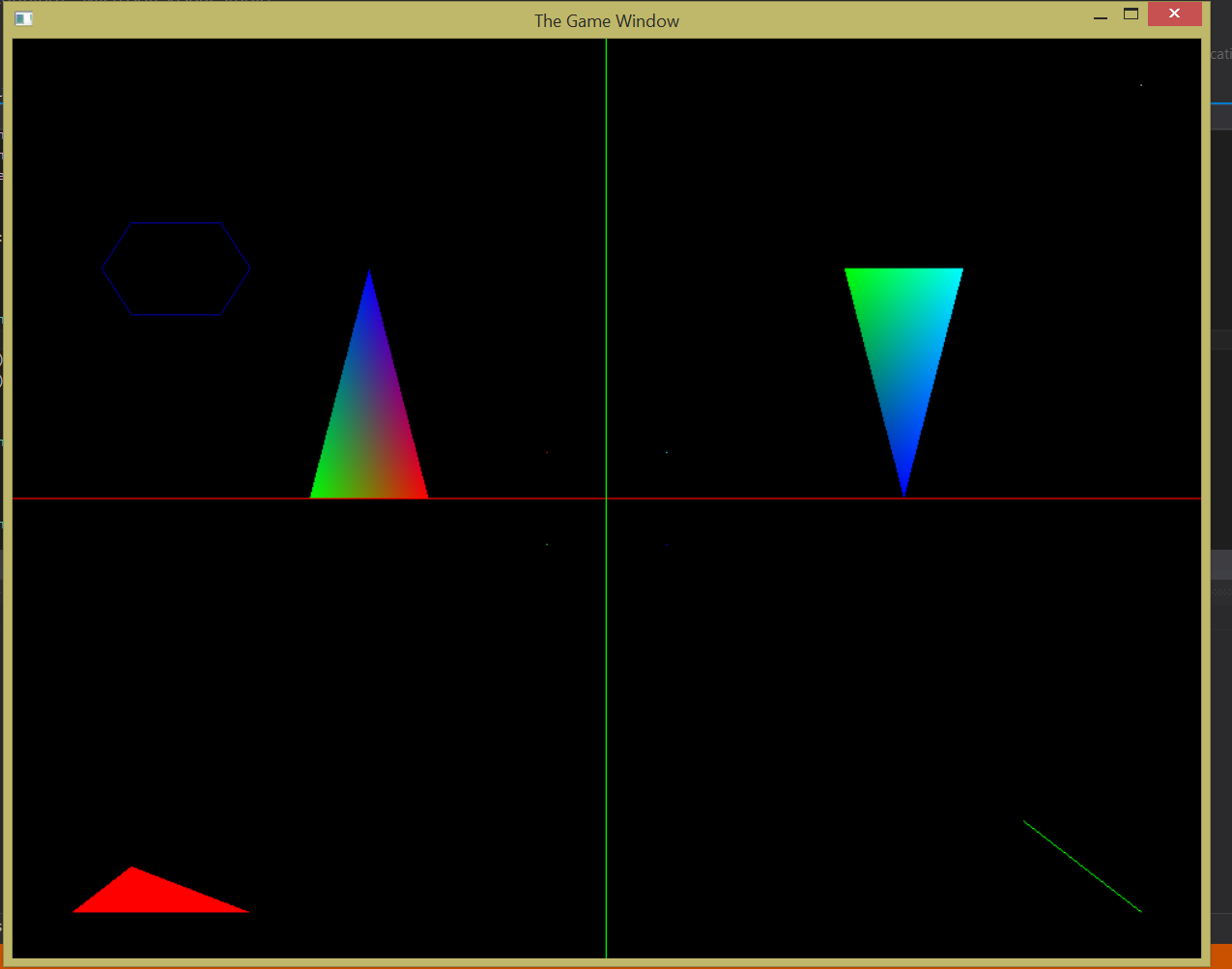


Example of a render method for an object in the OGLShader class:



**Screenshots:**

Application output:



**Lab 04**

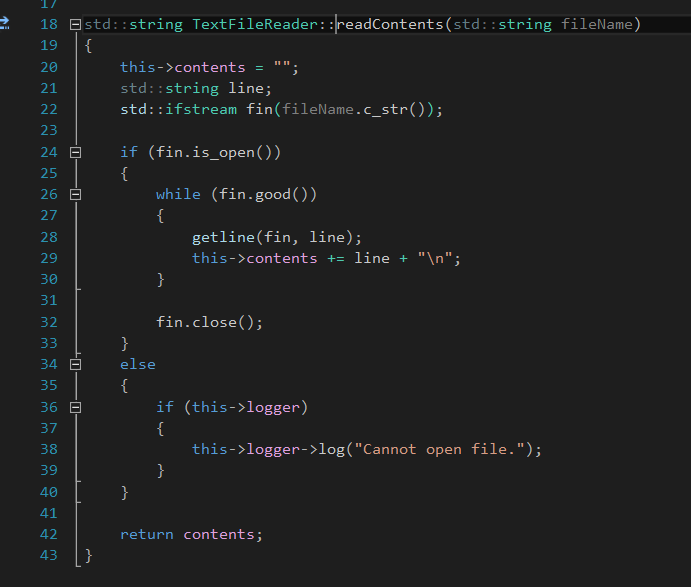
Lab 04 was, for the most part, an extension of Lab 03. The main objectives were to read the shader source code in from text files (versus writing it out as a string in the program), to refactor the bloated OGLShader class into a more structured and decoupled system, and to change the coordinate system to something more usable than the -1.0 – 1.0 normalized device coordinate system used by OpenGL. The vertex shader was stored in an external .glsl file which we read in using the TextFileReader class, which simply read the entire contents of the file into memory and returned it as a string.

The OGLShader class was split into the following classes: OGLVertexShader, OGLFragmentShader, OGLShaderCompiler, OGLShaderProgram, OGLRenderer, and OGLObject, all have which were derived from corresponding base classes. The OGLVertexShader and OGLFragment shader classes simply held the shader source code and defined a compile method. The resulting handle would then be passed to the link method of the OGLShaderProgram class, linking the shaders and obtaining a shader program handle which was stored in its super class, OGLBaseShader. The OGLObject class defined a VBOObject struct which stored all the information needed to create a new vertex buffer object. It contained a map which mapped a VBOObject to its corresponding name, and also contained a render method which accepted a handle to the shader program to use. The OGLRenderer class contained a shader program and a map which mapped an OGLObject to its name. It was also the classes which gave directives to setup the shaders, create the objects, and render all of the objects in the list.

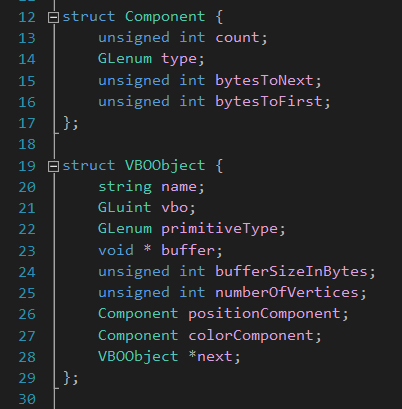
For our final objective in this lab, we had to change the coordinate boundaries of the viewport. This was done by dividing the position component in the vertex shader by the maximum value of the new coordinate system (e.g. for a system from -100 – 100, one would divide the position component by 100). This was done to simplify the task of mapping object coordinates to places on the screen, since it is easier to visualize spots between -100 and 100 than it is to visualize spots between -1.0 and 1.0;

**Code Samples:**

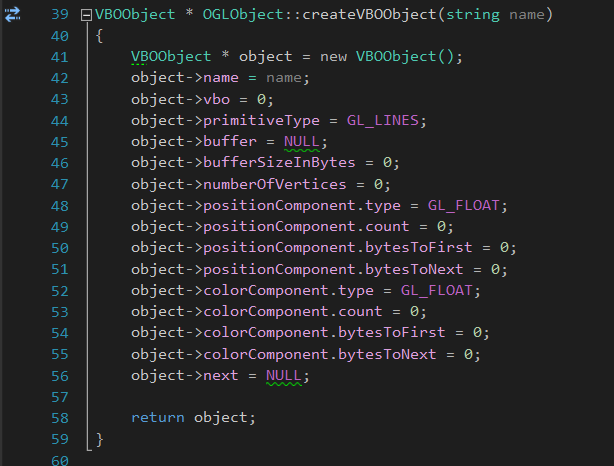
readContents method of the TextFileReader class

****

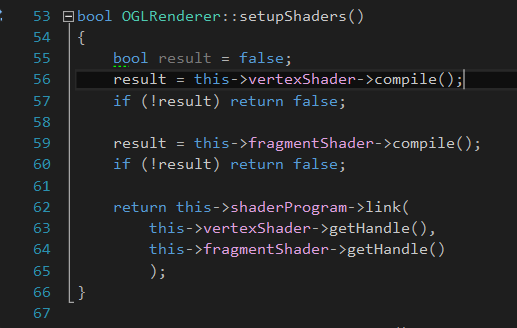
Structs in OGLObject class used to define a vertex buffer object:



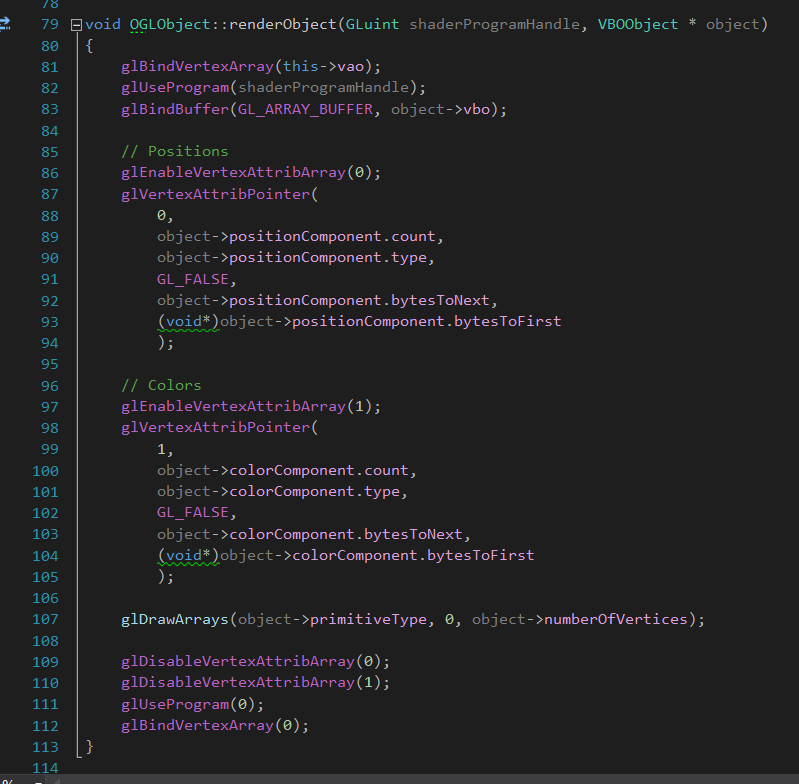
createVBOObject method of the OGLObject class:



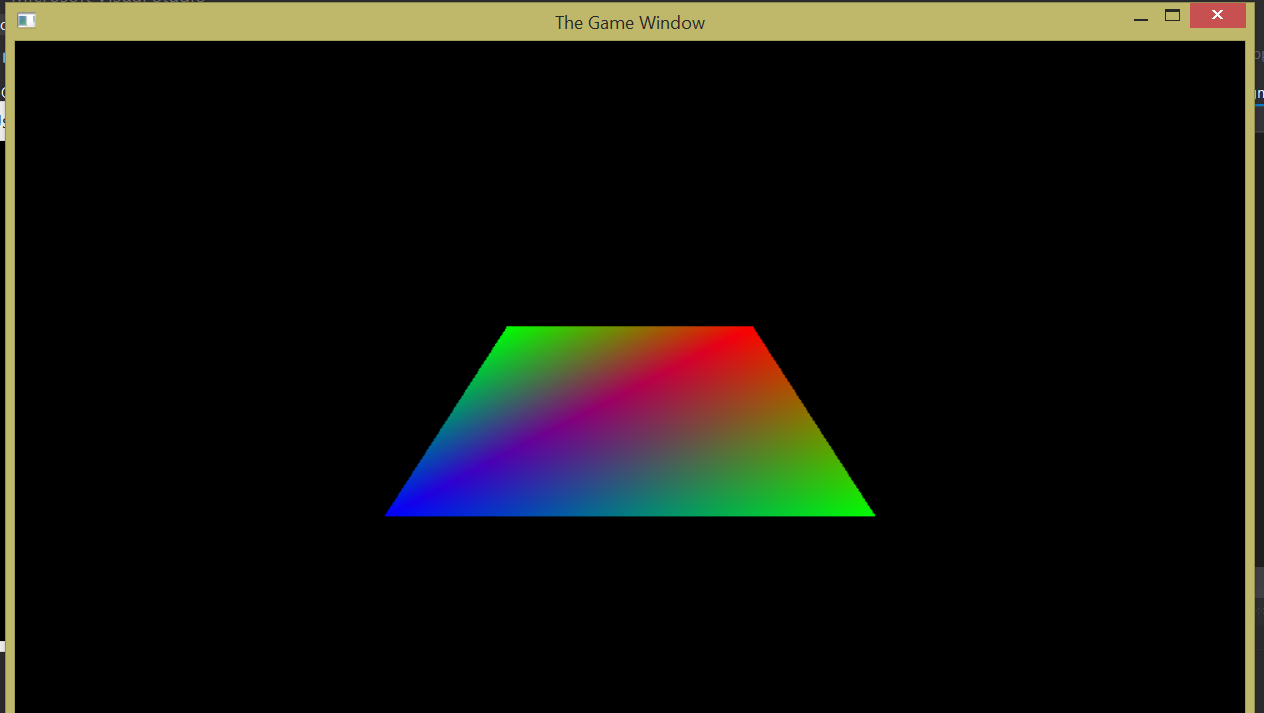
setupShaders method of the OGLRenderer class:



renderObject method of the OGLObject class:

****

**Screenshots:**

Application output:

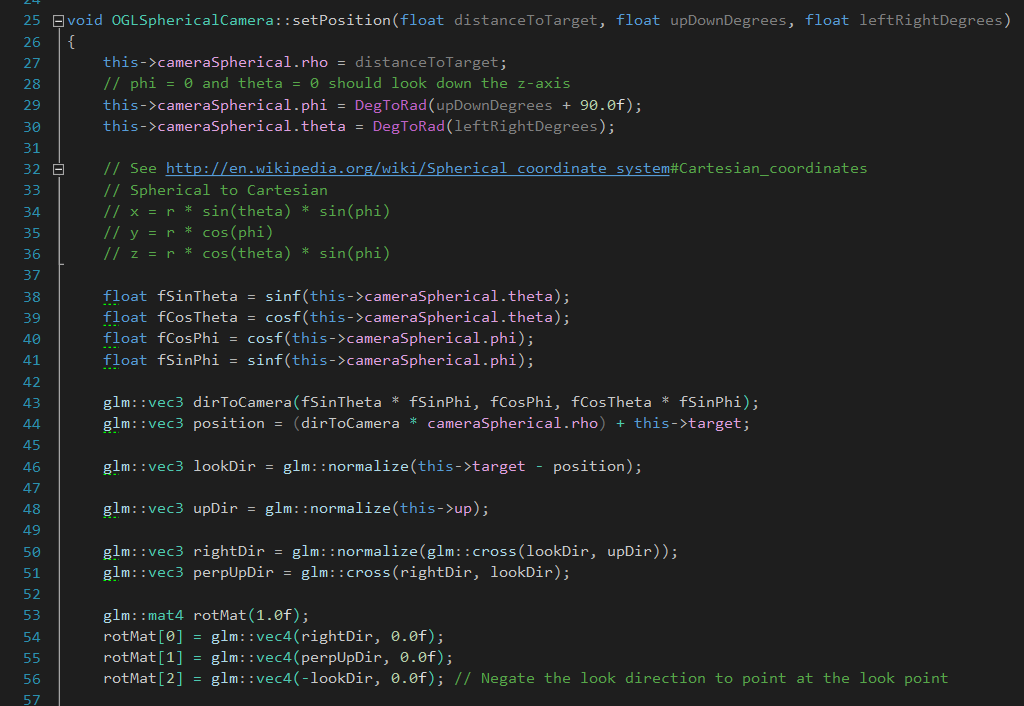
**Lab 05**

In Lab 05, we finally got a taste of 3D graphics programming with OpenGL. Our goals were to get our projects setup to use GLM (GL Mathematics), the Glutil library, and the Boost library, and to use them to gain experience generating 3D objects. The object model was very similar to the last lab, aside from the addition of the OGLSphericalCamera class (along with its base class Camera) and ObjectGenerator class. The spherical camera modeled a camera which could rotate about a fixed point and always faced that point. In its constructor, the up direction was defined as the Y axis and the target point was defined as the origin (0, 0, 0). The camera kept 4x4 matrix which contained its orientation and always contained its position in the world. The setPosition method would rotate the camera by a specified amount of up-down degrees and left-right degrees by converting its spherical coordinates to Cartesian coordinates. For this lab, the camera accepted inputs from the arrow keys (up, down, left, and right). The inputs were captured in the WndProc callback and the corresponding moveCamera\* method in the OGLRenderer class was called.

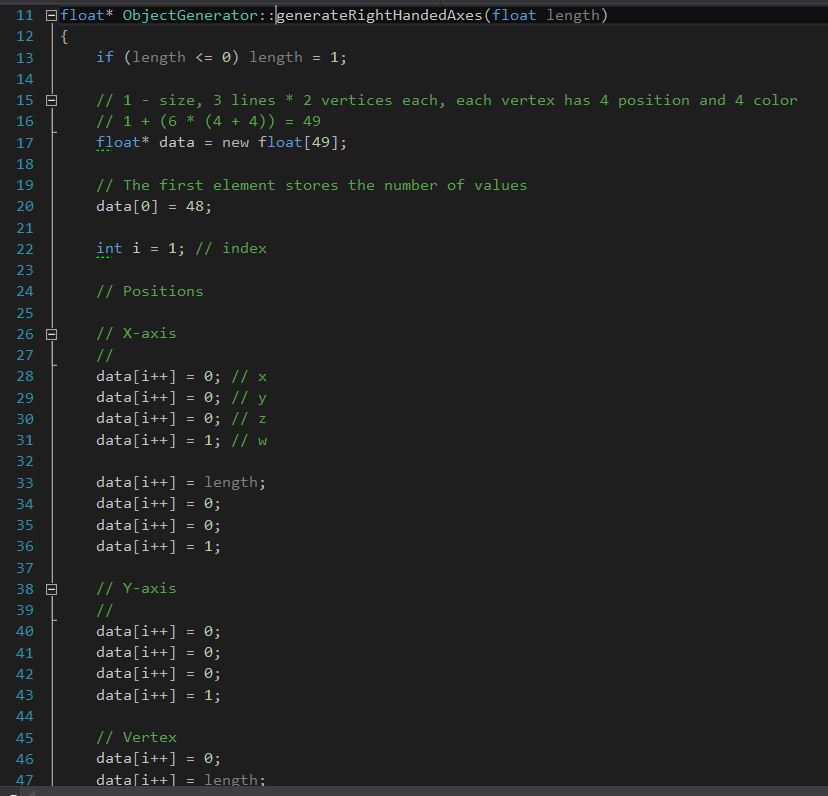
The ObjectGenerator class was a class full of only static methods than simply generated different kinds of 3D objects and returned a pointer to the array containing the object data. For this lab, the generator class could generate a right-handed axes system, a flat surface on the XZ plane, a pyramid, and a cuboid. The generator was utilized in the OGLRenderer class. The create method of this class called generator methods to create objects and, with the buffer of vertex data returned, created OGLObjects which were added to its map of objects. It contained two specialized method, generatePyramids and generateCuboids, which used a random number generator to create a certain number of objects with random positions and sizes and place them on the XZ place.

**Code Samples:**

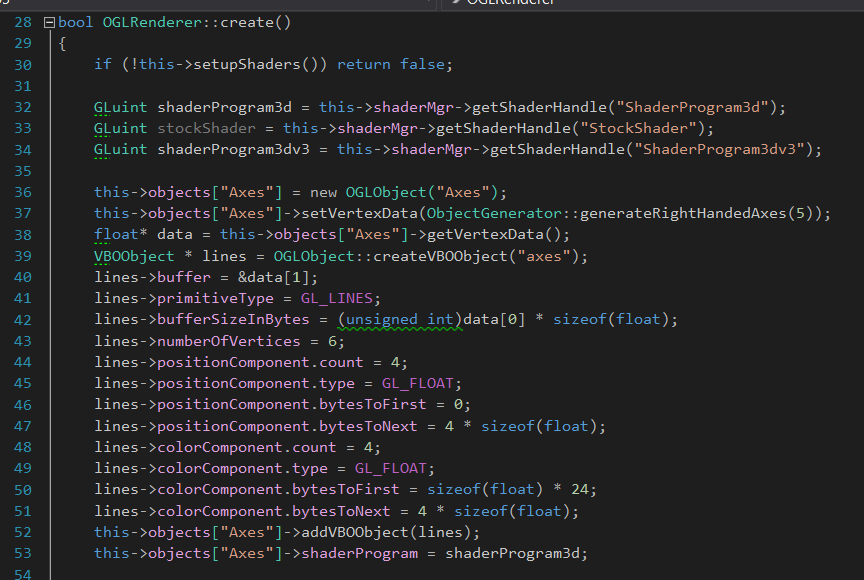
setPosition method of the OGLSphericalCamera class:



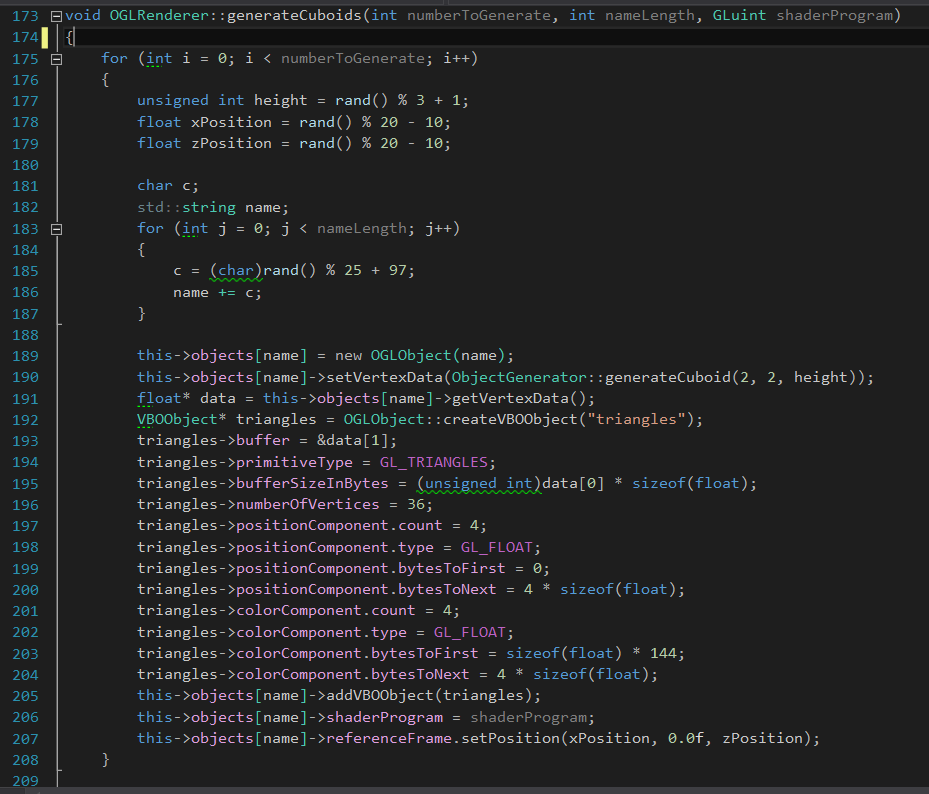
Example of a generate method from the ObjectGenerator class:



Part of the create method from the OGLRenderer class which creates the axes:

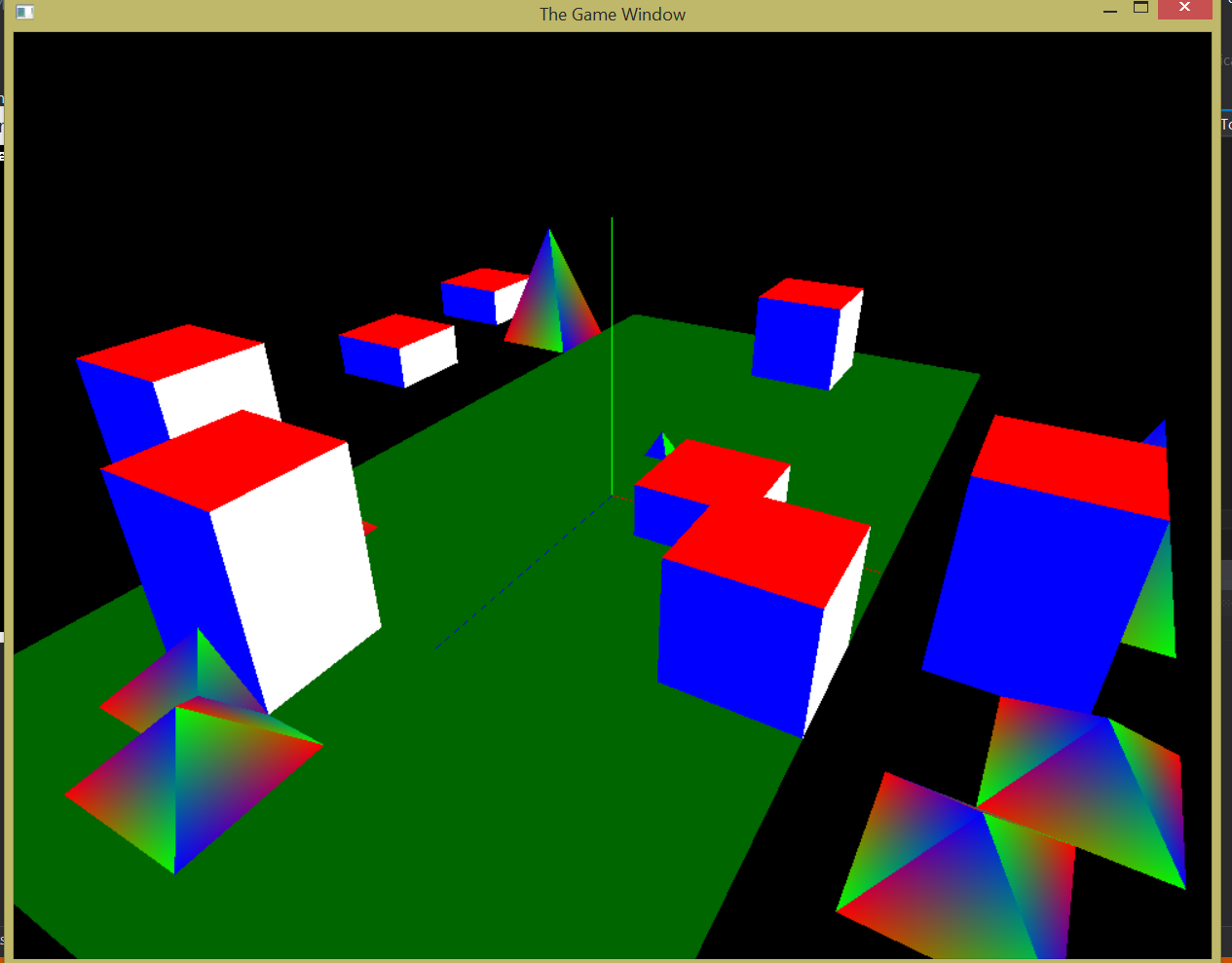


Example of the random generate methods from the OGLRenderer class:



**Screenshots:**

Application output:



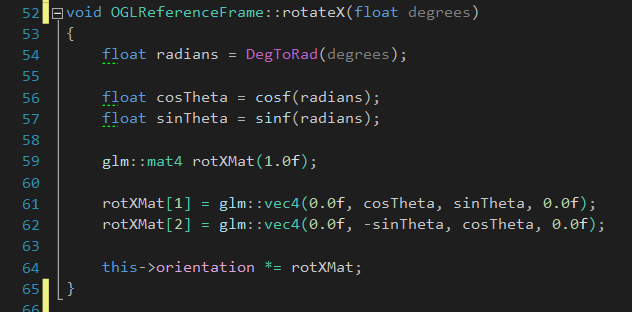
**Lab 06**

The main goals of Lab 06 were to learn about transforms (such as rotations and translations) in 3 dimensions and projection matrices, while further refactoring the code from the previous lab. There were multiple new classes added this lab, including GameEngine, GraphicsSystem, OGL3DObject, OGLReferenceFrame, StockObjectLoader and the pure abstract class Behavior. There were many more classes added, but these are the primary ones that were used for the objectives of this lab. The OGL3DObject class defines an OGLObject in 3D space and contains a 4x4 matrix representing the local to world projection matrix. Its base class, OGLObject, now contains an OGLReferenceFrame. The reference frame contains a 4x4 matrix that holds the orientation of the object in 3D space and also has methods to rotate the object about an axis and move the object in any direction. The object's reference frame is primarily used for animations, which are defined by implementing the Behavior class. One of the behaviors created in this lab is defined by the LoopyBehavior class. This behavior rotates the object around the Y axis at a random speed between 0.1 and 0.5 while it moves forward at a speed of 0.01. The transformations required for the behavior are invoked in the update function of the LoopyBehavior class.

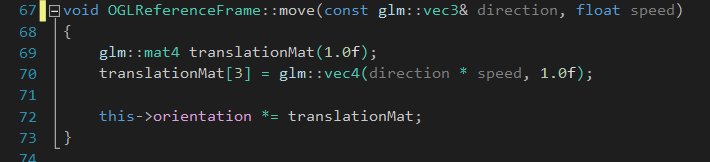
In this lab, objects are generated in the StockObjectLoader class by the familiar ObjectGenerator from the last lab. However, this time the objects are added to the GameObjectManager class upon creation, which is a member of the GameWorld class. In this lab, we see a relatively large refactoring of the graphics code and the object model really begins to take shape. GameEngine is the head of the object model and, in the WinMain method, the other objects are injected into its constructor via dependency injection in a hierarchical fashion. This makes for loose coupling and a game engine that is, overall, pretty simple and intuitive to manage.

**Code Samples:**

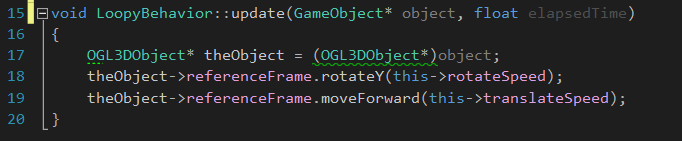
Example of a rotate method from the OGLReferenceFrame class:



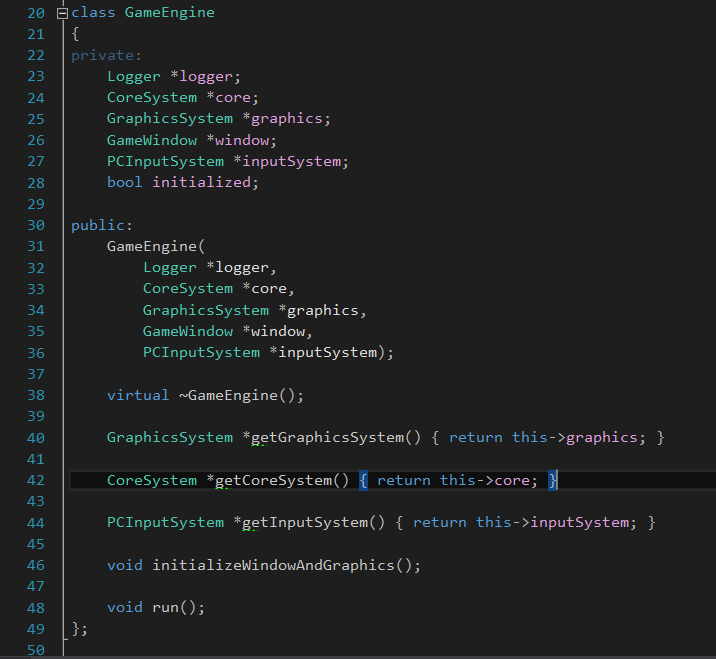
move method from the OGLReferenceFrameClass:



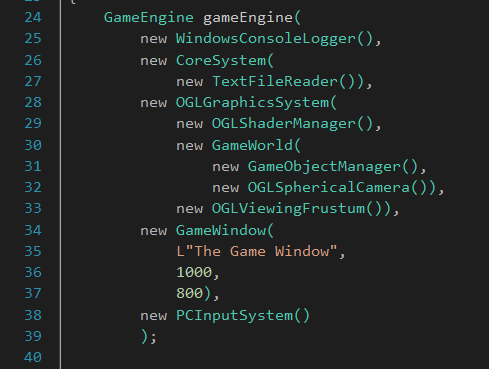
update method from the LoopyBehavior class:



GameEngine class, the root of the object model:



Dependency injection used to create objects in WinMain:



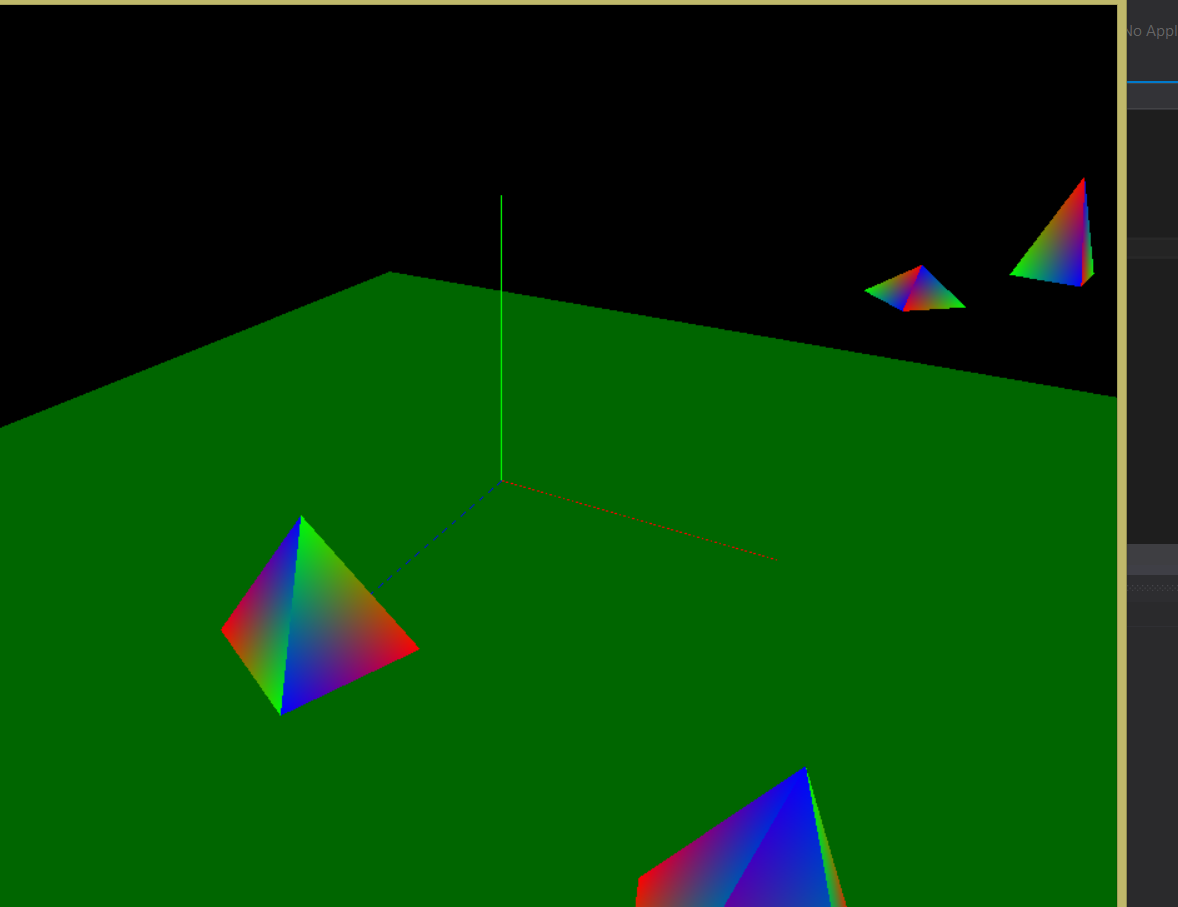
**Screenshots:**

Various stages of the loopy behavior at runtime:

Stage 1:



Stage 2:



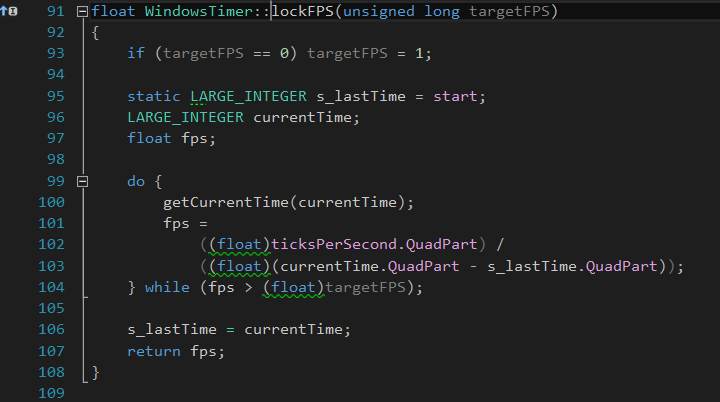
**Lab 07**

The objectives for Lab 07 were to gain experience using indexed arrays to store object data and to learn to define behaviors over time using a timer and basic state machines. Most of the classes in this lab are the same as they were for Lab 06, aside from a few changes relating to behaviors and object creation. There is now a WindowsTimer class that implements the ITimer interface (pure abstract class). It contains methods for locking in the number of frames per second and for general timing functionality (start, stop, get elapsed, etc.). In this lab, unlike in Lab 06, animations are based off of time instead of arbitrary “speed” values. The update method of IBehavior now accepts a pointer to a GameObject and the number of elapsed seconds as a float. One of the behaviors, for example, is called PatrolBehavior. In this animation, the object moves forward a set amount, rotates 180 degrees, and repeats. The amount the object moves before rotation as well as the speed (in units/second) is passed in via the constructor. Therefore, we were able to divide the speed by the elapsed seconds in the update method to get the amount at which the object should move during the current frame. We then used a state machine to define whether the object was moving or rotating. To determine when to switch states, all that was needed was a simple comparison between the current value moved/rotated and the maximum value to move/rotate.

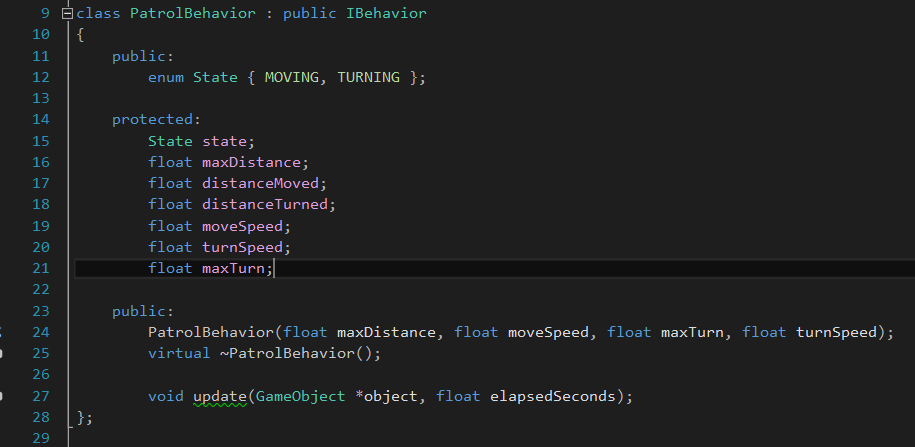
The other difference in this lab was the use of indexed arrays to create objects. Used in the generateBoxIndexedArray method of the ObjectGenerator class, indexed arrays allowed us to use the same vertices for multiple triangles in order to form a triangulated cube. Whereas we normally would have needed a total of 36 vertices to define all of the triangles needed for the cube, we were able to create one using this method with only eight vertices. After the cube was created and loaded into the object manager, its behavior was set in the setupGame method of the GameEngine class instead of in its own constructor.

**Code Samples:**

lockFPS method of the WindowsTimer class:



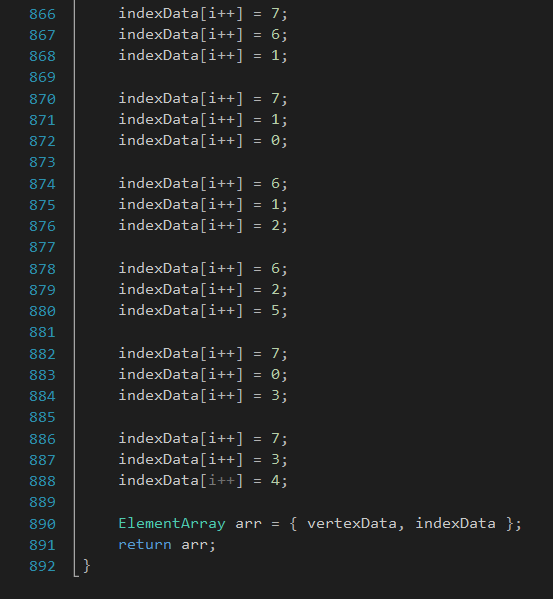
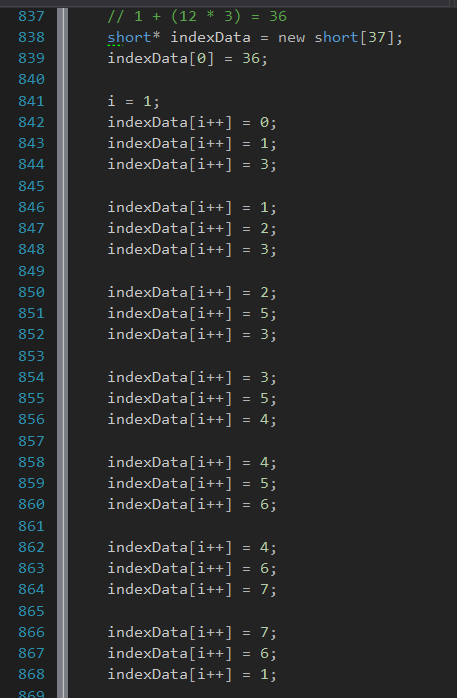
PatrolBehavior class with state enum and state tracking variables:



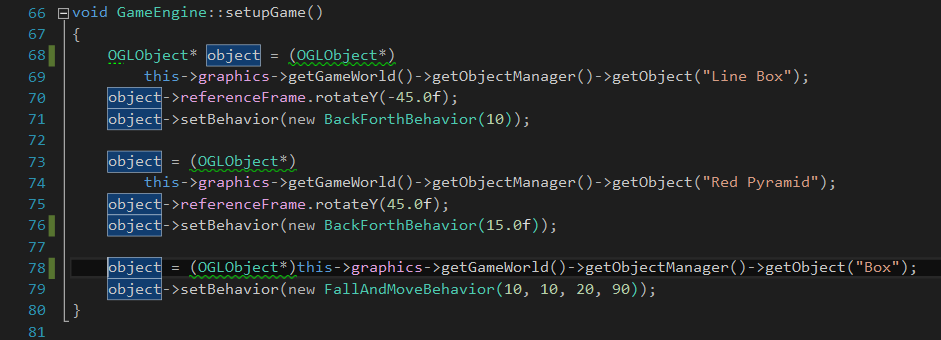
update method of the PatrolBehavior class:



Portion of the generateBoxIndexedArray method where the indexes are set:



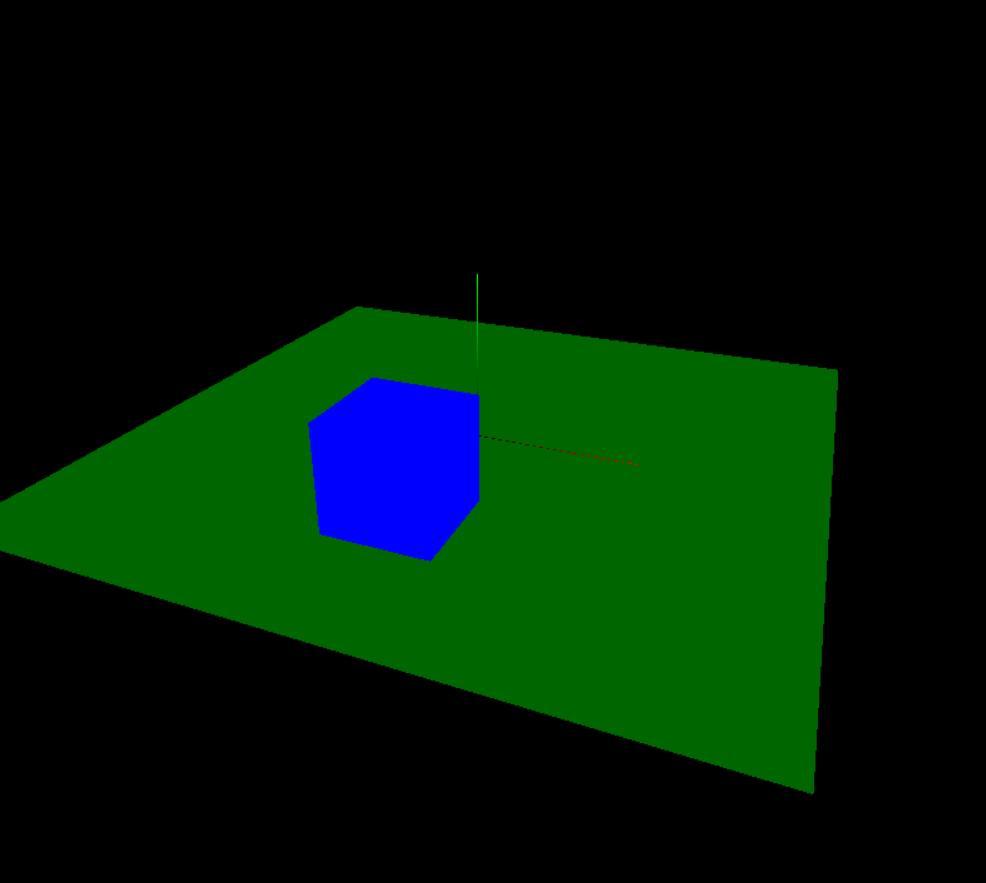
setupGame method of the GameEngine class:

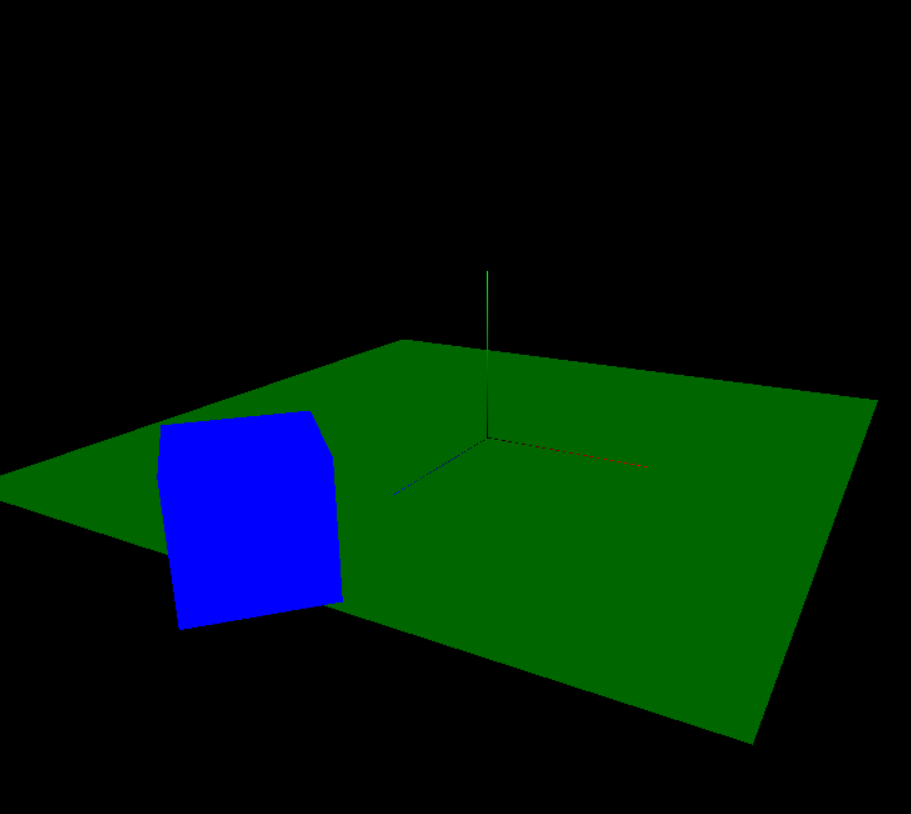


**Screenshots:**

Various stages of the patrol behavior at runtime:

Stage 1:

Stage 2:

****

**Lab 8**

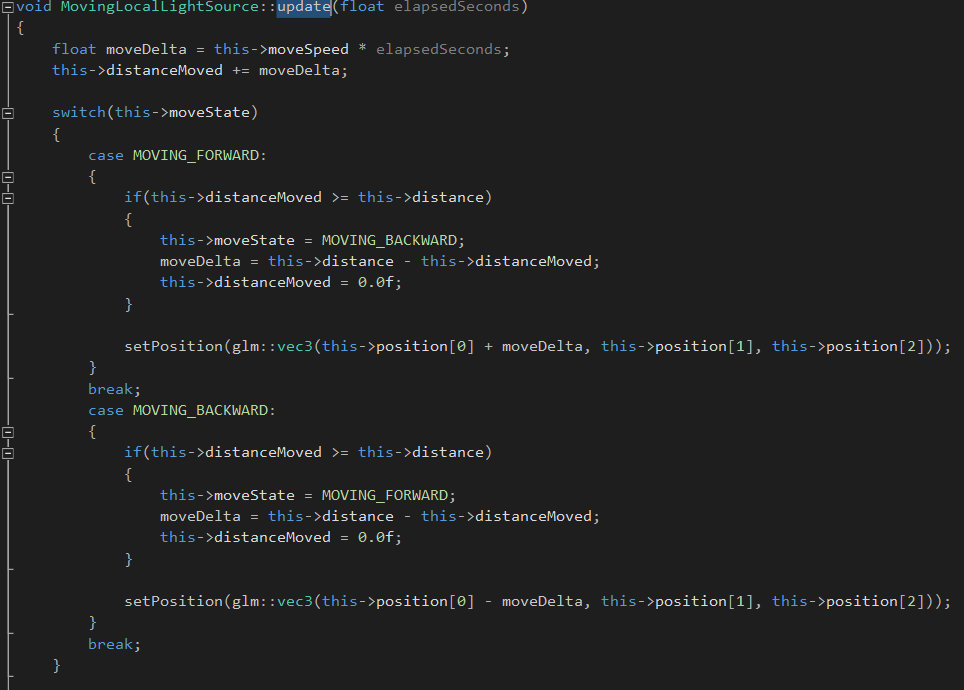
The goal of this lab was to begin creating a simple yet functional lighting model by building on our previous labs and projects. We began the lab with only a global light which lit the entire “world” and, by the end of the lab, we would have a fully controllable local light source for which we could dynamically adjust the brightness and position.

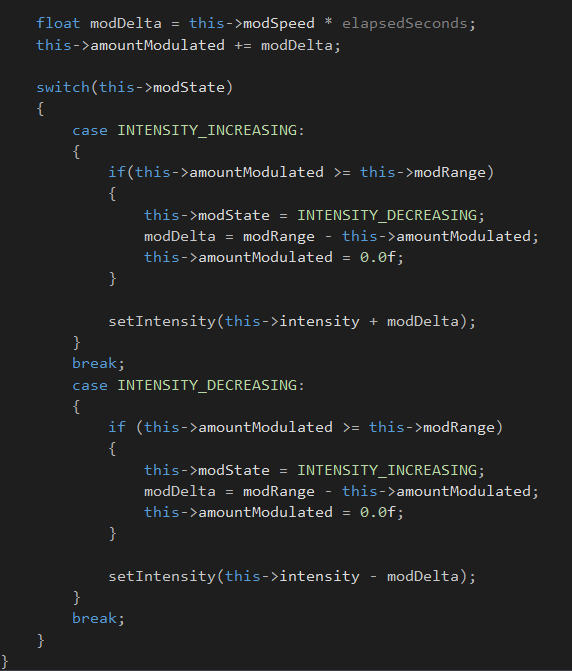
Over the course of this lab, we also refactored and added to our game engine. Previously, we only had a spherical camera which could be moved around a fixed point while always remaining a set distance away from the point. For the first step of this lab, we created a first person camera. This camera utilized state machines for the look state (controlled with the arrow keys) and the move state (controlled with 'W', 'A', 'S', and 'D'). It received its input from the WNDPROC callback function, which looked for the corresponding key presses. After this, we created a simple cuboid generator and two behaviors: one that would rotate an object about its Y axis and another that would mimic a four-point “patrol”. We then generated two cuboids and added them to the scene, with one having the rotate behavior and the other having the patrol behavior.

Finally, we got to the point where we actually worked with the lighting models. To start, we created a LocalLightSource class which would, of course, abstract a local light source. This class stored the position and intensity of a light source and contained an update() method which was called each time the rest of the game engine was updated. An instance of LocalLightSource was injected into the graphics system when the application was started. After this was complete, we created a moving, modulating local light source. The goal was to learn how to control the light source at runtime to provide some interesting effects. The light source would move back and forth by a predetermined distance while also modulating its intensity. At the moment, the shader only worked with one light source, so this movable light source replaced the simple local light source in the game engine.

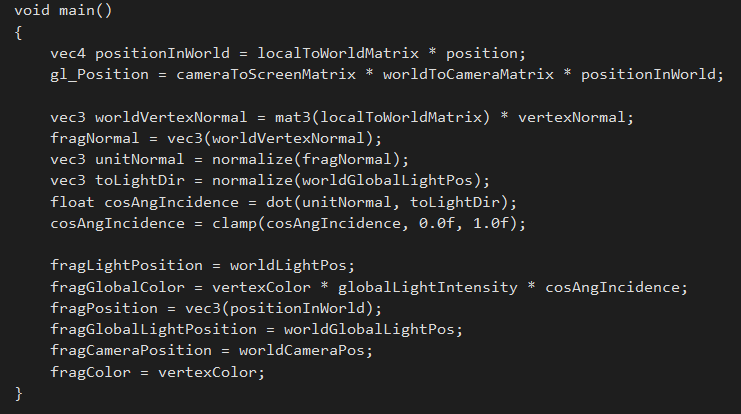
**Code Samples**

update() method of the MovingLocalLightSource class

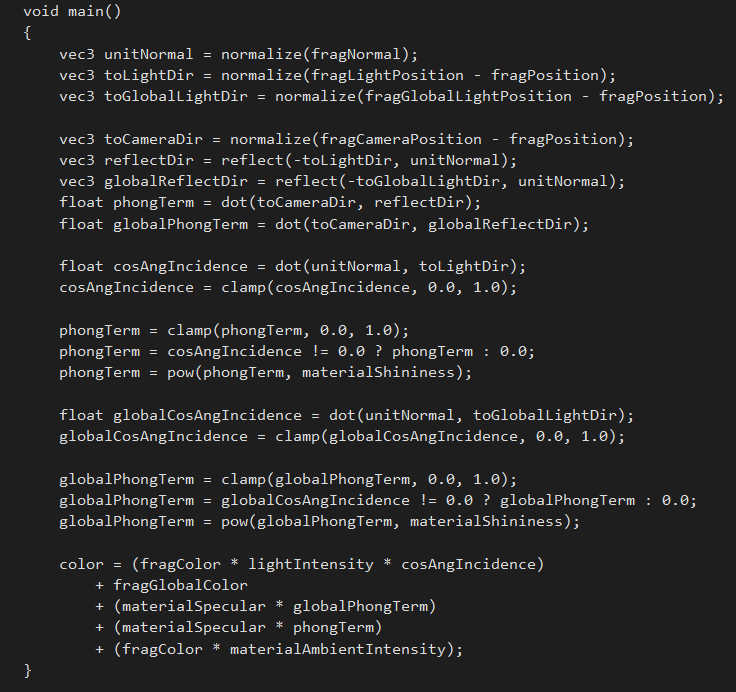




Vertex shader used for lighting



Fragment shader used for lighting



**Lab 9**

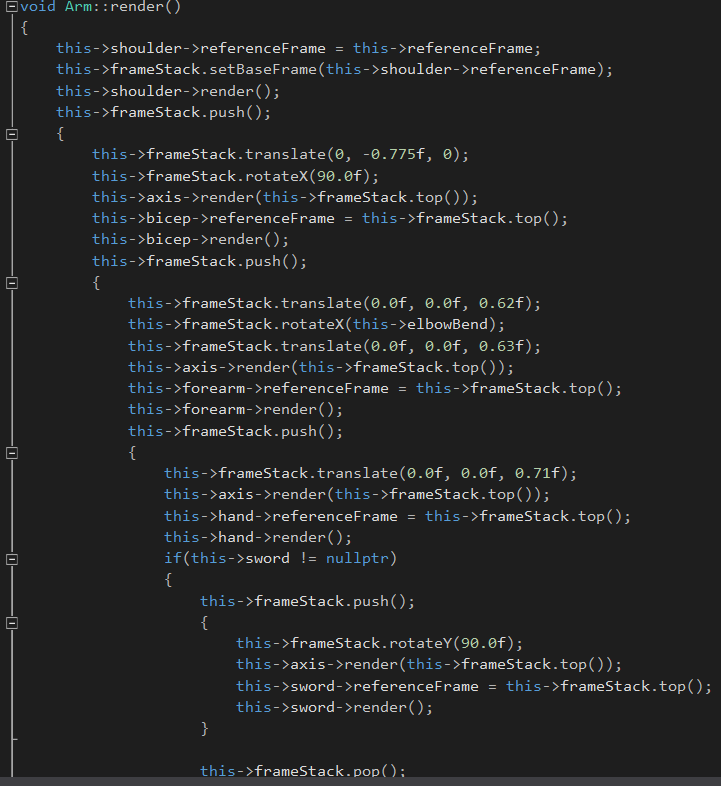
The objective of this lab was to begin creating more complex object models by using hierarchies to define our objects. Instead of just creating simple cubes, pyramids, etc., we created more complex objects by arranging these simple objects in a hierarchical structure. This was accomplished using a stack of reference frames; therefore, each object would use the reference frame of the object that preceded it. This allowed us to created more complex models and animations for those models in a logical way.

We began this lab by creating an Axis class which would be used to show the axes of each object in the hierarchy. This made it easier to see how the transformations of one object's reference frame would affected the other objects in the hierarchy. We then created an Arm class which contained four cuboids: one each for the shoulder, bicep, forearm, and hand. When the object was rendered, the shoulder cuboid's reference frame would be the root of the hierarchy. After that came the bicep, forearm, and hand. Once the arm was created, we had to orientate it. This showed how transforming the reference frame of the root object would transform and align the other objects in the hierarchy accordingly.

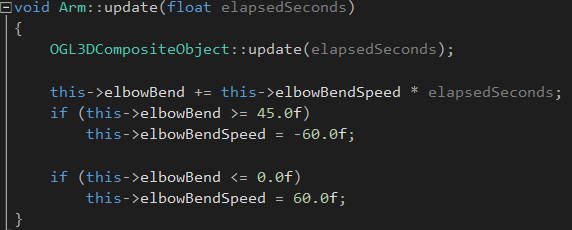
We then created Sword object using the same hierarchical techniques used for the arm to create the hilt, guard, and blade. The sword was added to the Arm and rendered in the reference frame of the hand. Upon completion of this, all of the objects were in their correct positions, but we still only had a straight, stationary arm. For the final two objectives of the lab, we bent the arm at the “elbow” by rotating the forearm's reference frame, then we added code in the Arm's update() method to make it “chop” by moving at the elbow. We learned that to do this, it was important to rotate the forearm at the elbow each time it was rendered before moving it down to its proper position.

**Code Samples**

The render() routine showing the hierarchical method used to render the Arm



The update() method used to calculate the degrees used to make the arm “chop”



Screenshot of the program running



**Lab 10**

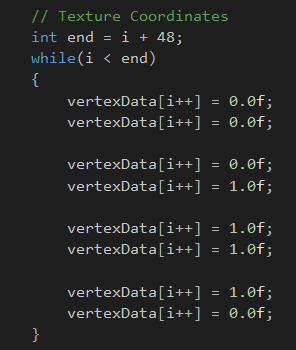
For this lab, our goal was to begin applying textures to simple objects using basic texture mapping patterns. This lab built on the lighting and hierarchical modeling ideas introduced in the previous two labs, allowing us to apply textures to simple objects (e.g. cuboids) while still building more interesting hierarchical models.

We began the lab by abstracting a flat plane object out into its own class. The class would define a plane with a specified width, height, and position. We then placed the plan parallel perpendicular to the X-Z plane in the center of the pre-constructed room. After this, we created a CustomTexture class which would be added to the plane. It defined a simple, square-shaped texture which had four quandrants forming a black and white checkerboard pattern. This simple texture was set as the plane's texture and mapped to the four corners of the plane. Upon successful completion of this step, we gave the custom texture a slightly more complex pattern, consisting of red and light blue, while also modifying the fragment shader to not render the light blue portions of the texture, thereby making those areas see through.

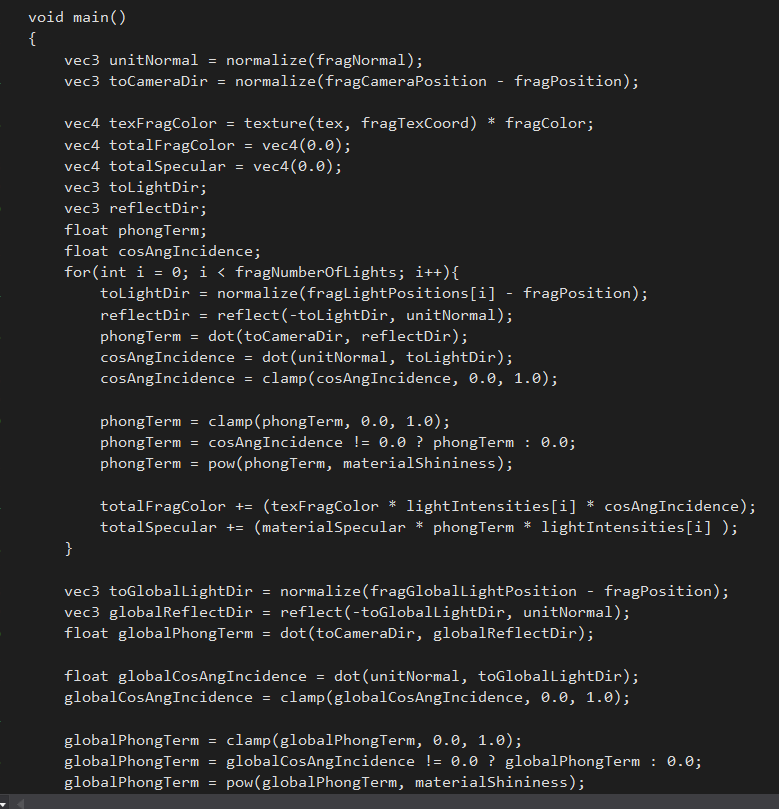
Things then got more interesting when we started loading bitmap images to use as textures. We used a simple bitmap image loader and found any image on the web which had dimensions that were a power of two, converted it to a bitmap, and loaded it in as a texture with the bitmap loader. It was then applied to the simple plane in the same way that the custom textures were applied. After this, we were ready to map textures on to more complex shapes than planes. We wrote a texture mapper for a cuboid, which simple took a square texture and mapped it onto each face of the cuboid as if it were a plane. Finally, we used our knowledge of texture mapping and hierarchical objects to build a treasure chest, complete with and open-and-close animation for the lid.

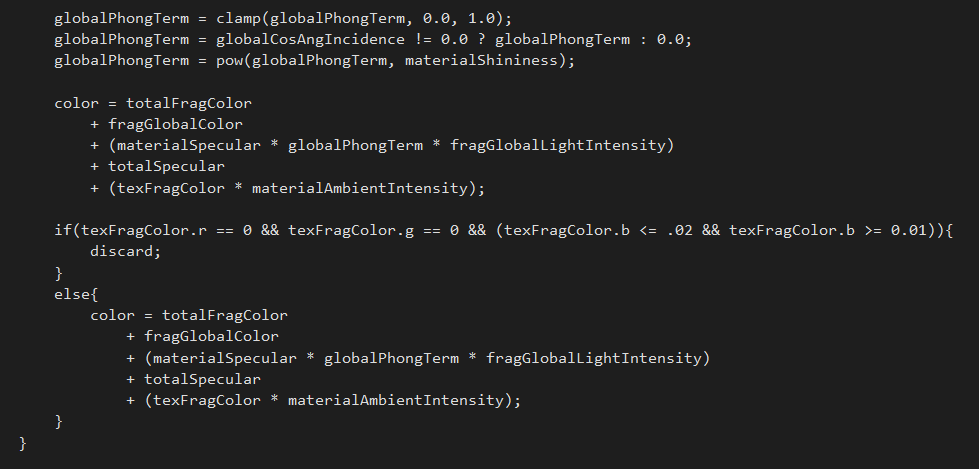
**Code Samples**

Mapping the texture onto the faces of a cuboid

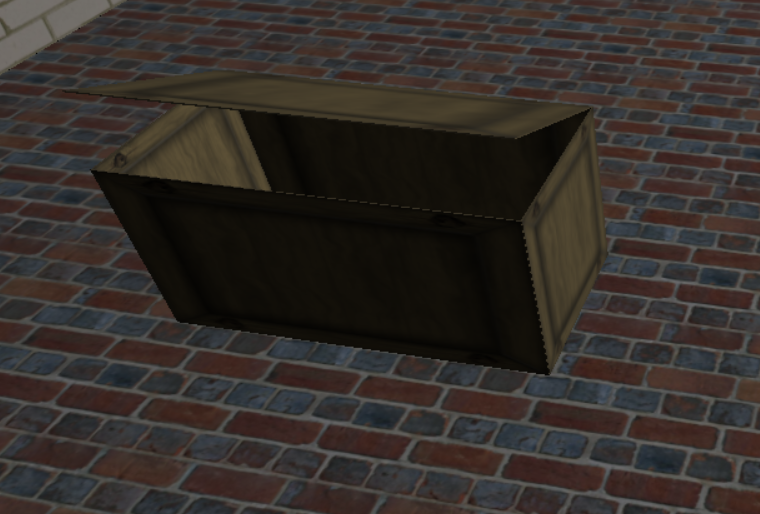


Fragment shader used for textures





Screenshot of the textured treasure chest



**Lab 11**

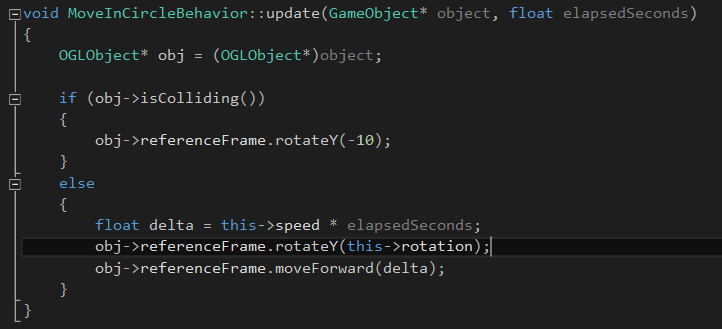
The goal of this lab was to provide an introduction to basic collision detection while also allowing us to work on creating an object that would be used in our game. A large portion of the object hierarchy was refactored and the ability to display text on the screen as a simple 2D texture was added. Collisions were determined using a bounding box on each object and a basic algorithm which checked each object in the scene to see if its bounding box entered the bounds of any other object in the scene. The object class also provided the option to display its bounding box as a line box around the object or hide the bounding box entirely.

We began the lab by creating a MyObject class which was a simple triangle plane and added the object to the scene. We then used our knowledge of texture mapping from the previous lab to read in a bitmap image as a 2D texture and map it onto the triangular plane at the point in which it was created. After this, we got the opportunity to create our own object. Since the objects that my team was using for our game were too complex to be created during class time, I opted to create a simple robot-type object using an object hierarchy consisting of cuboids. I added a behavior to my robot that allowed it continuously move around in a circle, with the speed of the movement being passed through the constructor.

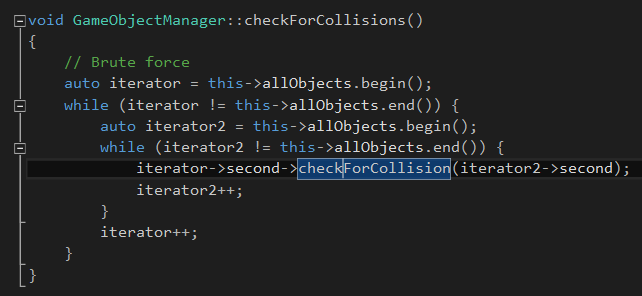
For the final step of the lab, we gave our custom object the ability to collide with other objects in the scene. We followed the examples given by the objects previously constructed in the scene to define a bounding box, add it to our object, and display it. Using this new ability, my robot would continue moving in a circle until it collided with another object in the scene, at which point it would rotate about its Y axis until the two objects stopped colliding. When any objects in the scene collided, a text display would pop up in the center of the screen displaying the names of all of the objects colliding.

**Code Samples**

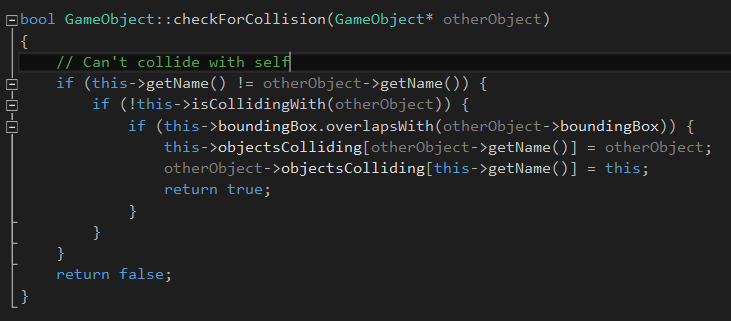
update() method of the robot's behavior



Method to check all objects for collisions using brute force approach



Method to check the defined object against another object for collision



Screenshot of the game running and three objects colliding



**Lab 12**

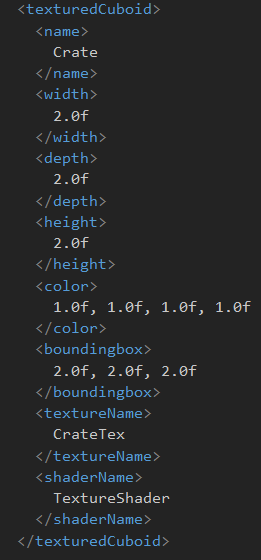
Lab 12 built off of the basic collision detection scheme introduced in the previous lab. It also provided another large refactoring of the game engine, now allowing objects, lights, and shaders to be defined in external XML files and read into the game. This lab also introduced the concept of gravity to the game and the assumption that all objects are initially moving, thereby creating the necessity for a fixedInPlace boolean attribute in the object class in order to define objects such as the ground, walls, etc.

We began by creating a flat surface with a brick texture for the ground and a cuboid with a wooden crate texture initially positioned in the air above the ground, both of which were defined in the GameAssets.data file. We gave the crate an initial acceleration of -32.2 m/s2 in the Y direction. This caused the crate to immediately begin falling; however, it fell straight through the ground because we had not implemented the proper collision detection yet. We then added resolveCollisions() methods to the game object class and the object manager class. This would cause an object to stop moving upon colliding with another object and prevented objects that were fixed in place from being checked for collisions, allowing the crate to fall and sit on the ground.

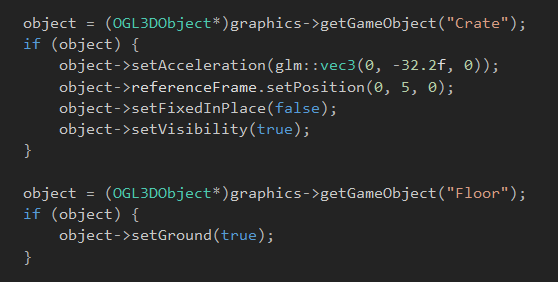
After this, we added velocity to the game object class so that when the object collided with another object, it could be “bounced back” out of the object's bounding box, thereby not colliding anymore. This caused the crate to continuously bounce when it hit the ground, which was obviously not desirable. To remedy this, we added the ability to define an object as being the ground and added a boolean variable to the game object class which could be used to specify whether or not that object was currently on the ground. If it was on the ground, then its acceleration did not get set. We ended the lab by defining a simple plane in the GameAssets.data file to be used as a back wall.

**Code Samples**

Example of an object defined in the GameAssets.data file



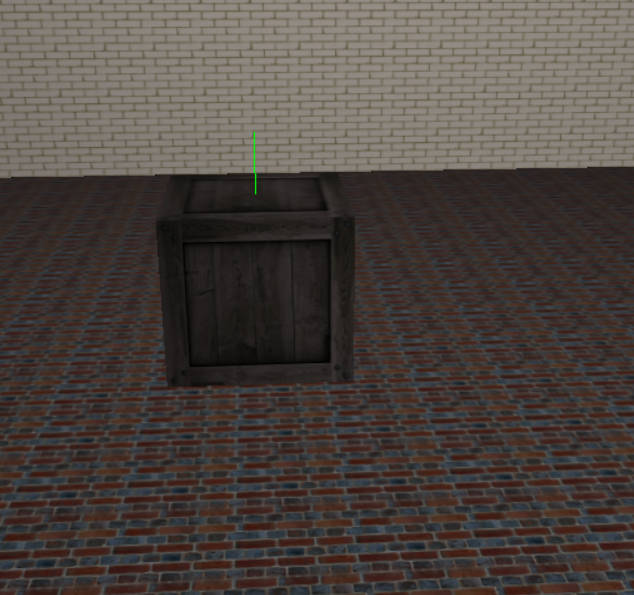
Setting up the Crate and Floor for the scene



Collision detection and resolution methods used in the GameObject class



Screenshot of the game running with the crate on the ground



**Project 1**

The goal of Project 1 was to begin the habit of reading in configuration information for the game engine from external files. This would continue on with the reading of shader source code from files and the reading of object data from files. However, for the first project, the primary objective was just to read in the game window's configuration data from a window.config file. This information included the window's title, start position, width and height, and background color. I chose to store my configuration data in to format <attribute\_name>: <attribute value> . This made for a file that was simple to write, read, and parse. Upon reading the file in, we had to use the data to create a Win32 window with an OpenGL viewport and display it.

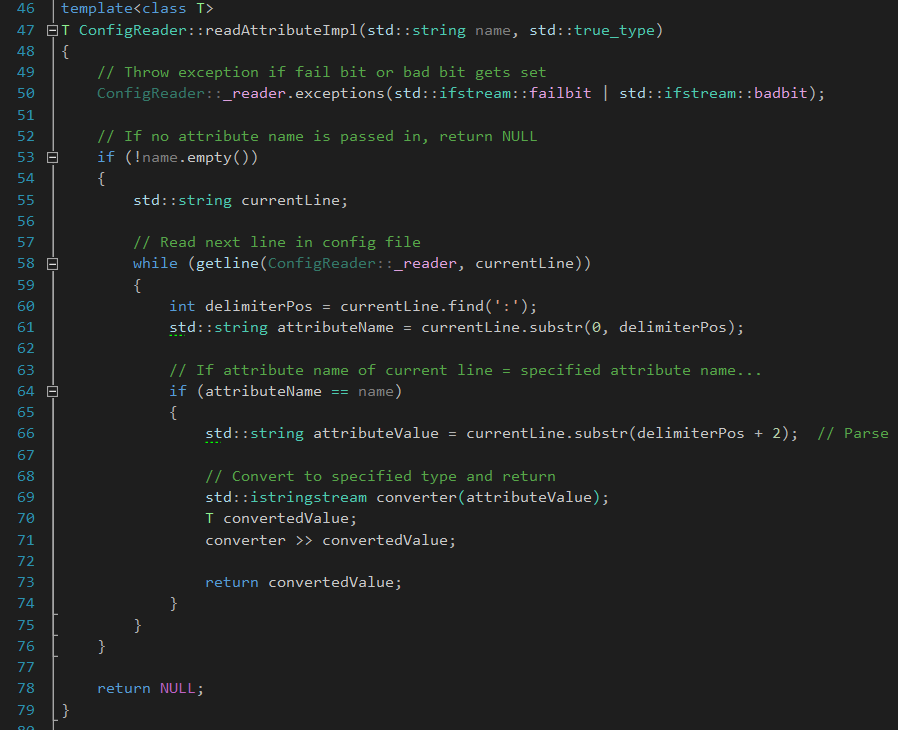
In order to read the configuration file, I needed to write a parser that would correspond to the file format that I chose to use. I created a ConfigReader class which contained the public methods openFile and readAttribute. The openFile method took a file name and tried to open the file with ifstream, returning true of false based on whether or not it was successful. The readAttribute method was a template method, with the template parameter being the attribute's type and the method parameter being the attribute's name to read. It called readAttributeImpl backing methods based on whether the type parameter was an integral type or a string, read the attribute with the specified name from the file and returned it as the specified type. I then created a WindowConfigReader class which inherited from the ConfigReader class. It had a readConfigFile method which used its base class's read method to read all the window-specific attributes in the window.config file. It then used the attributes to populate the WindowInfo struct that was passed to the method by reference. Upon reading the window information, the WindowConfig struct was passed into the constructor of the Win32OpenGLWindow class, which used the data to create the new window accordingly. After making sure the call to create succeeded, the window was shown and the listenForEvents loop was started.

**Code Samples:**

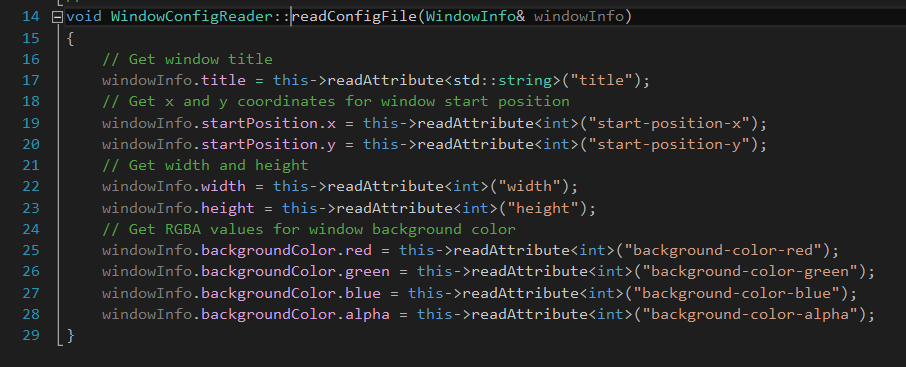
The window.config configuration file:



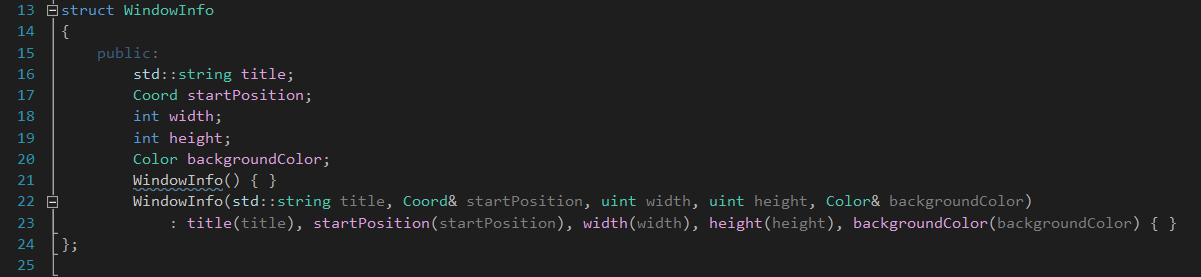
One of the of the readAttributeImp methods in ConfigReader:

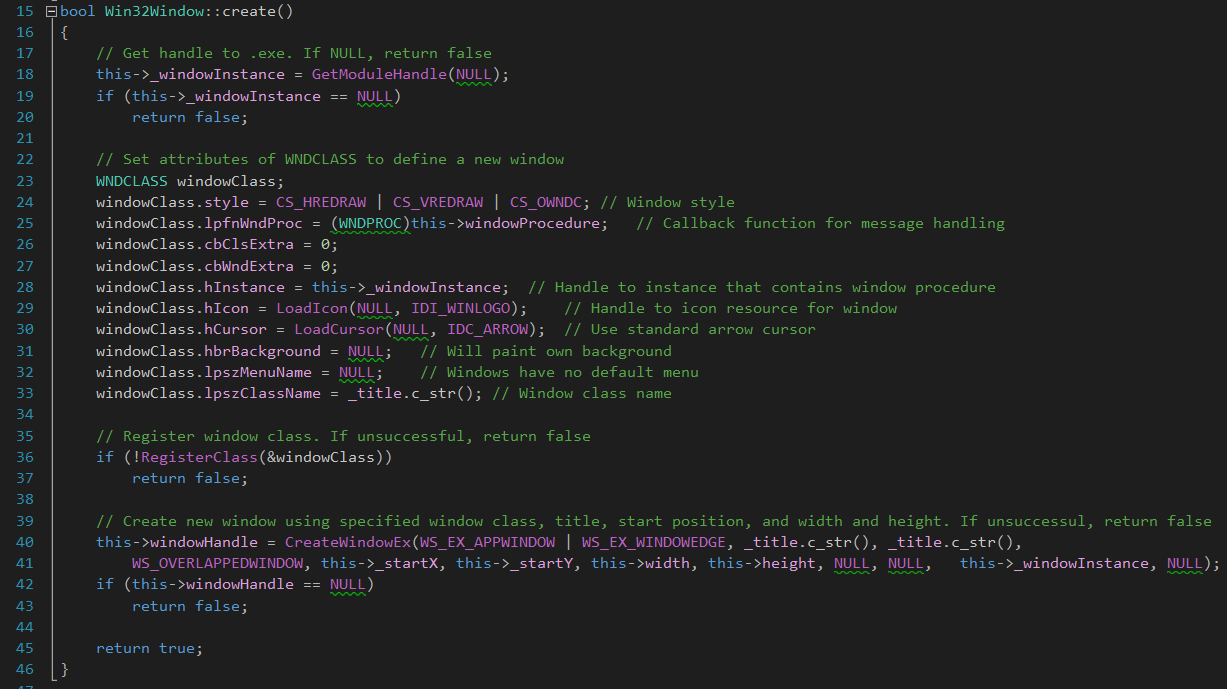


The readConfigFile method of the WindowConfigReader class:



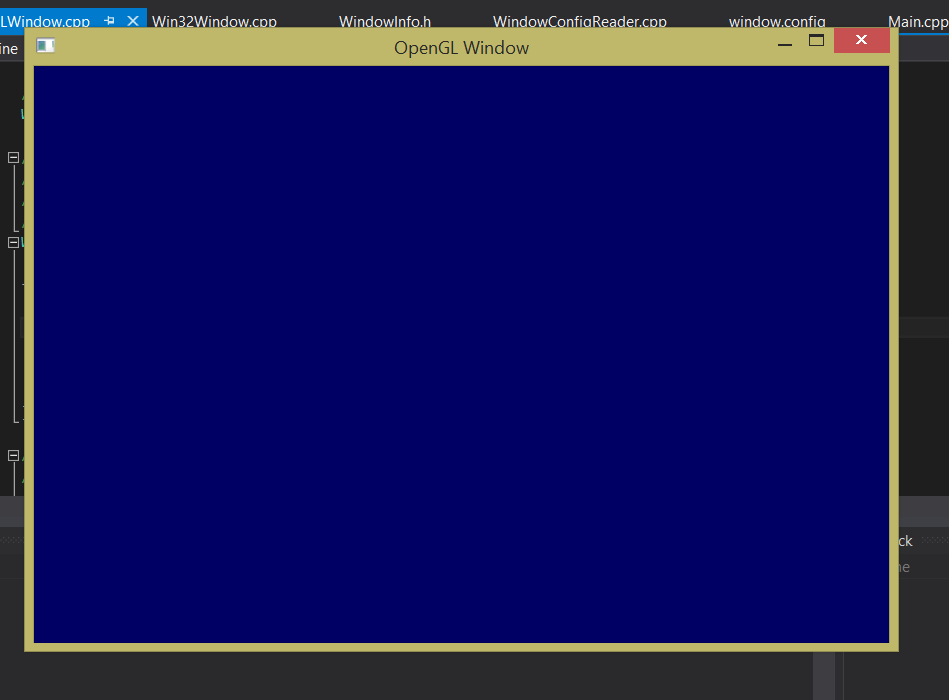
The struct used to store the window data:



The method which creates the Win32 window using the window info:

**Screenshots:**

The application after successful startup:



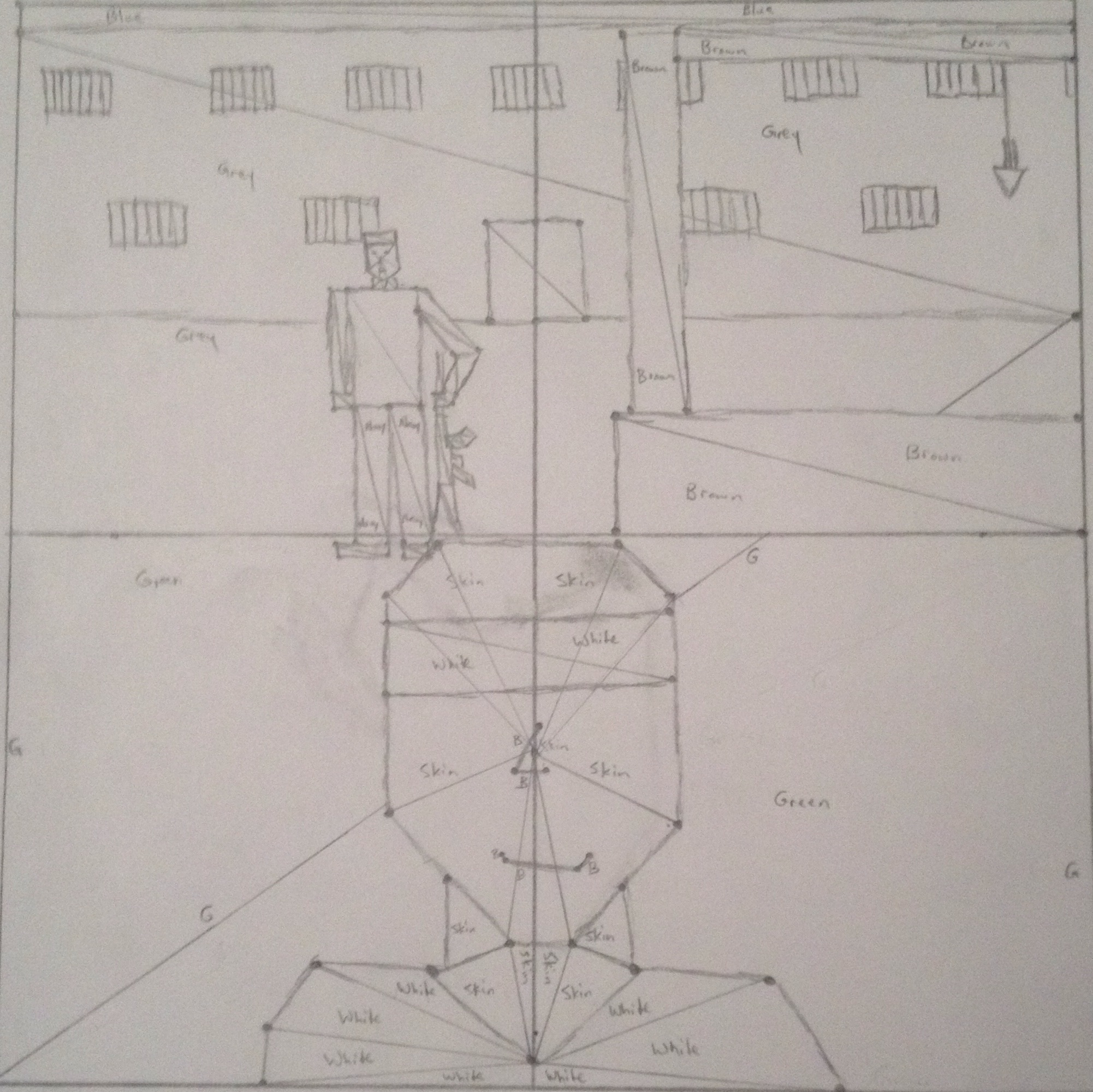
**Project 2**

Project 2 was a two part project. For the first part, we had to create a mock-up of a simple 2D scene where the X and Y values ranged from -1.0 to 1.0. I opted to sketch my scene out by hand. After finishing the outlining of all the objects in my scene, I began to triangulate them and write in the color of each triangle. Upon completion of that step, I had to design a file format in which I could store my scene data so that it could be parsed and used to recreate the scene with OpenGL. I chose a format in which I started each line with a letter representing the type stored in that line – P for point, L for line, and T for triangle. After the letter, I defined all the vertices for that type in the form X Y Z R G B X Y Z R G B... Comments had a '#' at the beginning of the line. I chose this format because it allowed me to specify entire objects together instead of specifying points together, lines together, and triangles together.

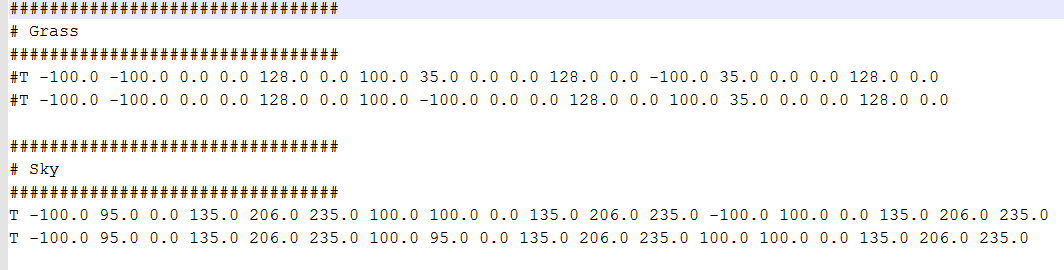
The second part of Project 2 was to create the scene from part one using OpenGL, while also making the background gradually shift between two colors. The codebase changed quite extensively from Project 1, since I had to add support for graphics rendering. I did so by using an object model very similar to the one used in Lab 04, the main differences being that I did away with the OGLShaderCompiler class and added the ShaderManager and SceneReader classes. The shader manager was used to keep track of the shader programs used in the application by mapping the OpenGLShaderPrograms to their corresponding names. The SceneReader class contained static methods used to open a file containing a scene, read the scene, and build the scene, storing the data in the OpenGLObject instance passed to the method by reference. I also utilized an OpenGLGameManager class which held pointers to the GameWindow and the OpenGLRenderer. The game manager was responsible for creating the window, setting the renderer, and starting the game loop.

**Code Samples:**

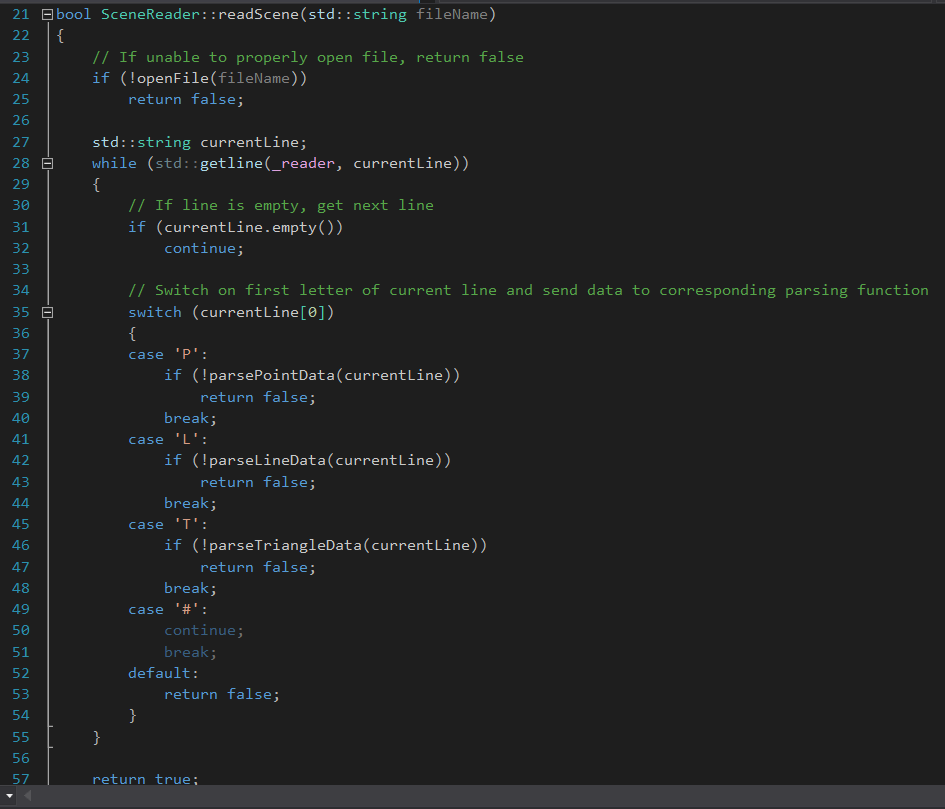
Sketched scene:



Sample of the scene file used to recreate my sketched scene:



readScene method from the SceneReader class:



parseTriangleData method from the SceneReader class:

Image 1:

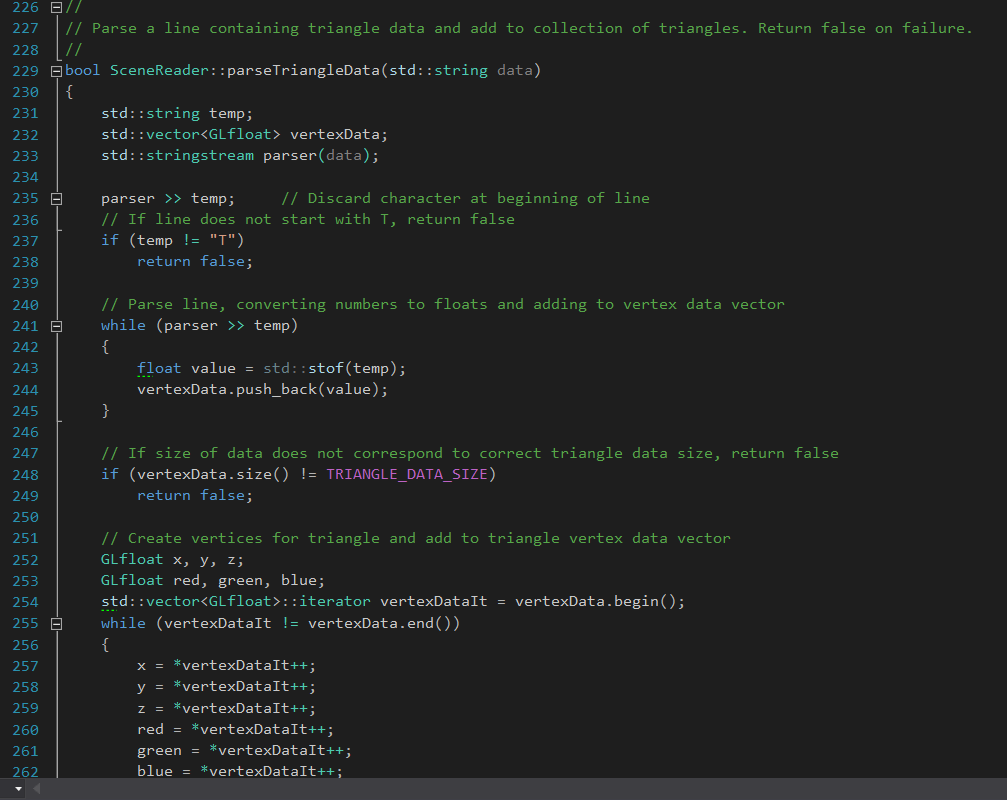
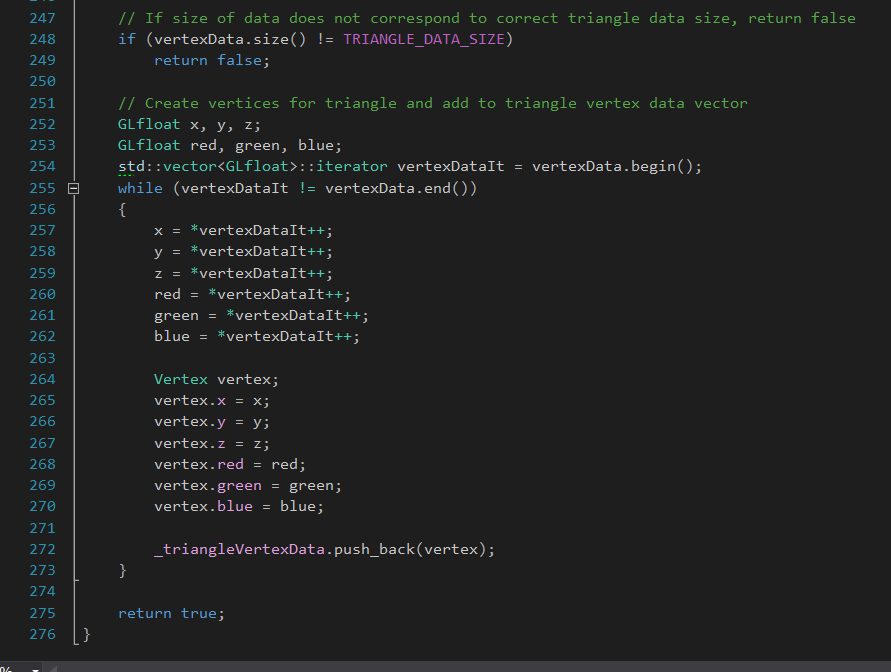
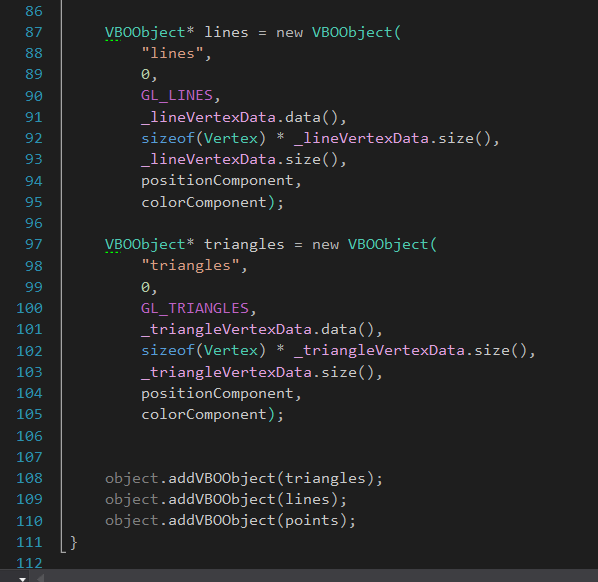


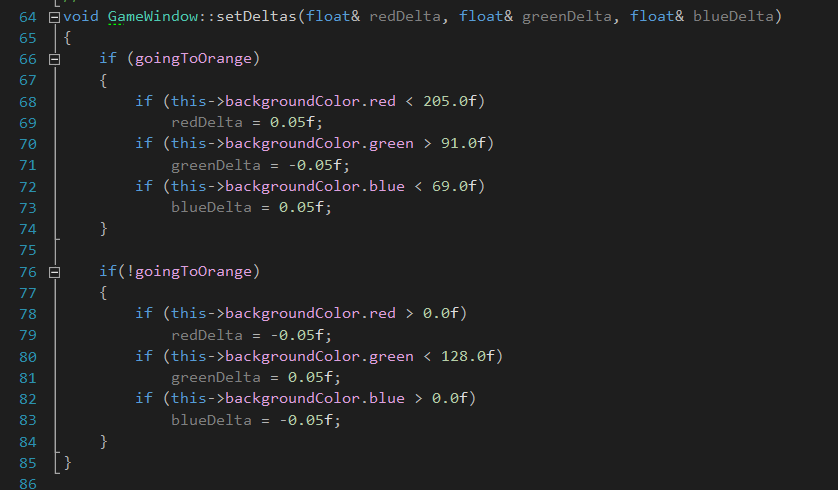
Image 2:



buildScene method from the SceneReader class:



Method used to set the deltas used to change the background color:



**Screenshots:**

Fully rendered scene:

**Project 3**

Project 3 was the first project in which we used 3D objects instead of a flat, 2D scene. The goal was to create a scene using at least three 3D objects which were more complex than the basic cube, pyramid, etc. while also giving these objects some sort of simple animations. I decided to make a guitar, a pick, and two music notes. The guitar and pick would start out laying on the floor and, when the program was run, they would rise into the air and rotate to an upright, “playing” position. The pick would then begin strumming up and down as the two music notes flew outward and away from the guitar in a semi-random floating motion. After a set number of seconds, the notes would return to their starting position and repeat the action.

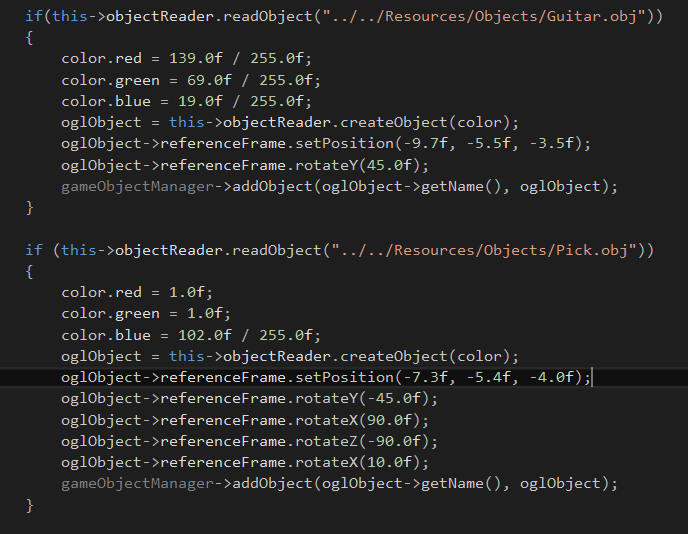
In order to make more complex models without having to map out all the vertices by hand, I learned the basics of the 3D modeling program called Blender. This allowed me to create my guitar, pick, and music note models with a higher level of detail than I would have been able to otherwise and allowed me to spend more time on the actual logic for the scene instead of writing models. I exported my Blender models in the OBJ file format, which created the need for me to write an OBJ file loader which would read my OBJ files in at the start of the program and create an object out of the data.

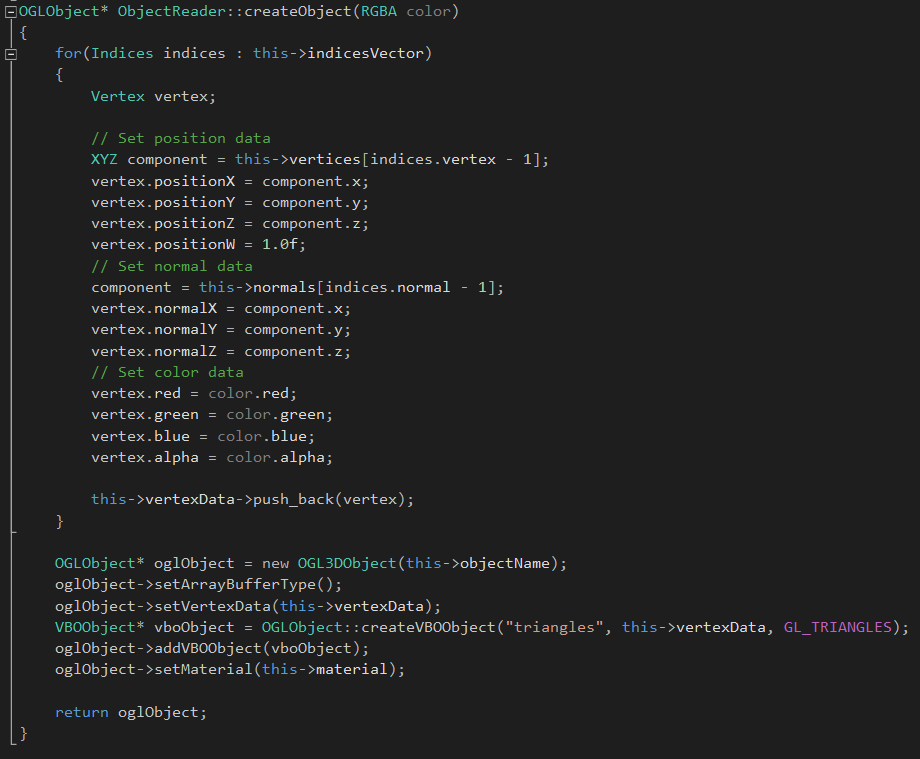
After completing the initial requirements for the project, I decided to add some more to it. We had just covered lighting in the class labs, so I decided to add that into my project. To do this, I created a simple cubical room that the objects would operate in and put a light in the ceiling. I also made use of Blender's ability to export material (MTL) files and the high level of cohesion between those and the OBJ files. I added the ability to read MTL files in to my OBJ file reader, which gave me the ability to add more characteristics to my objects which could only be recognized through the use of lighting.

In order to view the entirety of the scene, a spherical camera model was used. This camera was always fixed on the origin (0, 0, 0) which also happened to be the center of the room. The camera could be moved around the target point 360 degrees, but could not move up or down.

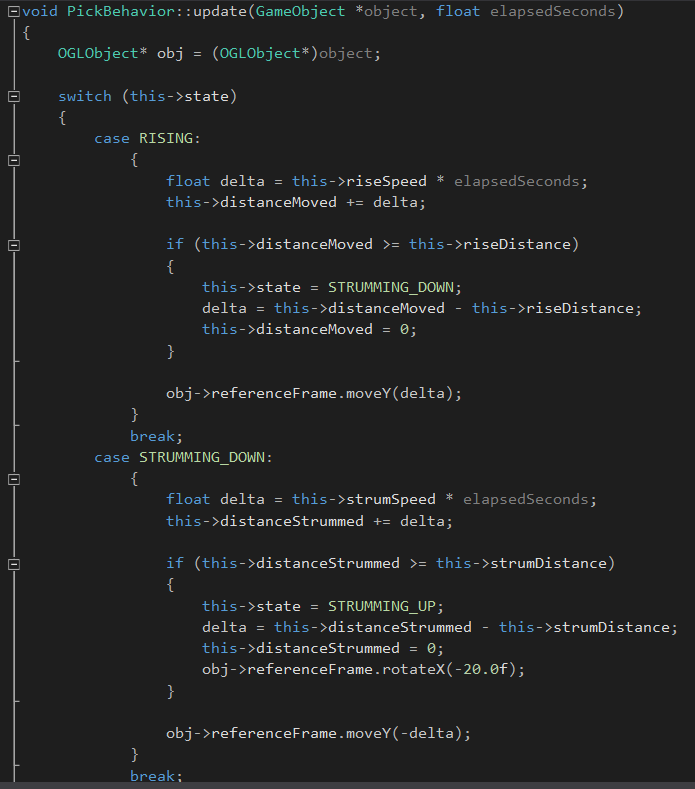
**Code Samples**

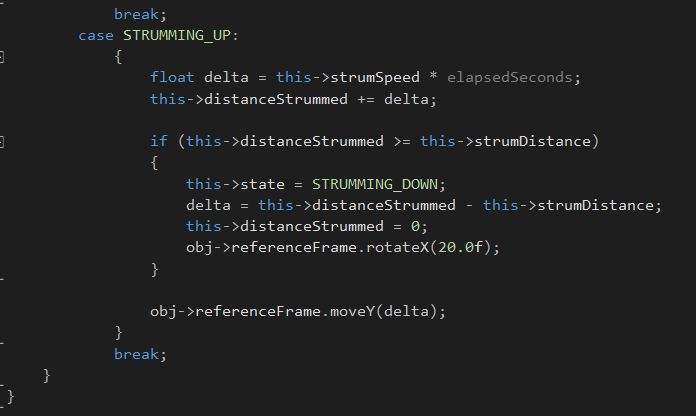
Loading the guitar and pick objects using the OBJ file reader



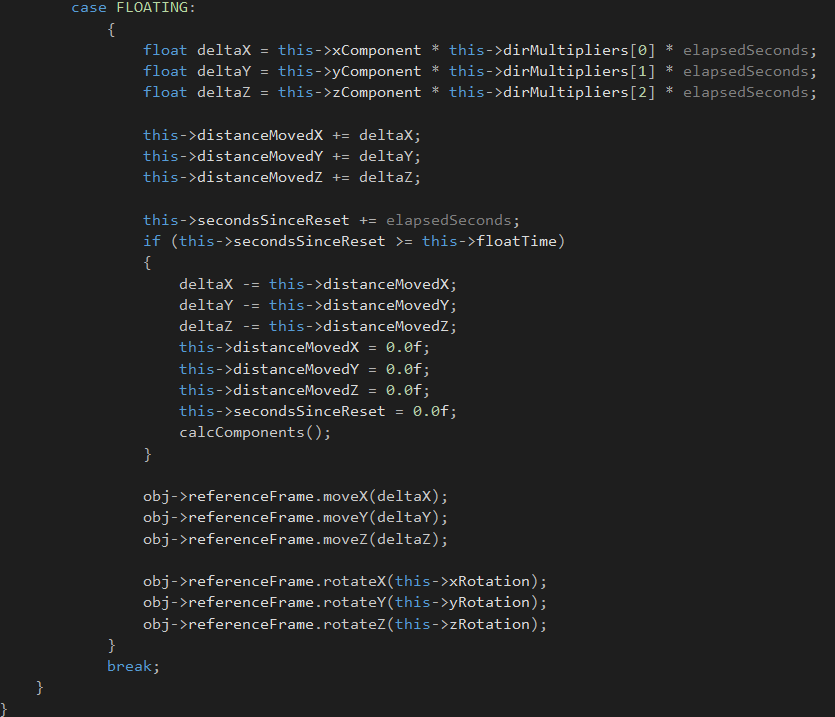
Method in the OBJ file reader to create an object using the data

update() method for the pick object's behaviors





update() method of the music note objects' behaviors



Screenshot of the program running and the music notes “floating”



**Project 4**

Project 4 was our final individual project for the semester. We used everything that we learned over the course of the semester for this project, aside from textures, collisions, and a small amount of physics. The goals of this project were two have a scene with two rooms with each room having at least one light source, some type of furniture, and at least on animated object created using the object hierarchy model that we had previously covered. The “player” would be represented using a first person camera. When the player got close to a light source, the light would turn off. Likewise, when the player entered or exited a room, all the lights in that room would either turn on or off, respectively.

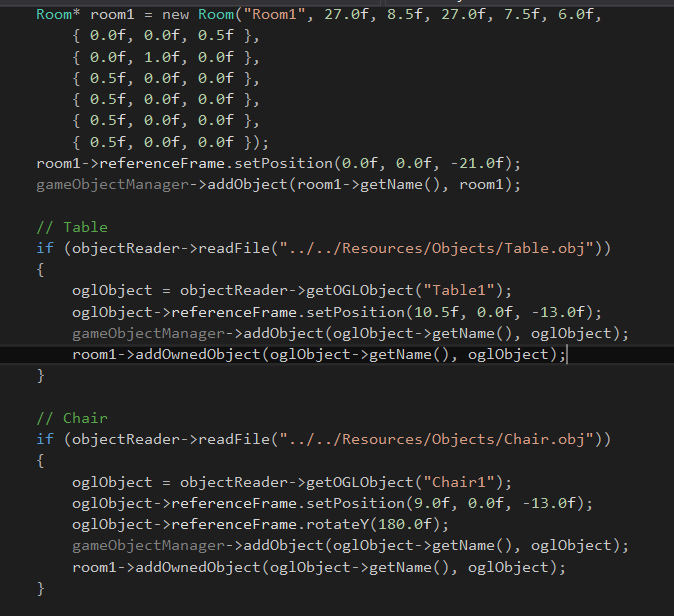
Like my last project, I used blender for all of my models (aside from the walls of the rooms). I created two rooms connected by a hallway. One room had a bed, desk, and chair for the furniture, a cat pacing in a square pattern as the animated object, and a tall lamp and a desk lamp for the light sources. The room opposite it had a desk, stereo system, and chair as furniture, person sitting in the chair strumming a guitar as the animated object, and a tall lamp and spinning ceiling fan as light sources. I opted to make the objects in a room children of that room, making it easier to position them using the room's reference frame instead of the using world coordinates.

One of my main challenges with this project was trying to determine when the player either left or entered the room. I abstracted all of the walls, the floor, and the ceiling out into their own Room class, so I had to come up with a solution that was not dependent on the location of each specific room, but could work for any room in any position that I decided to put it. I ended up determining the bounds of the room using some of the data passed to the class's constructor: position, which was the center of the room relative to the world's origin, width, and depth. Using this, I could calculate the bounds of the room in relation to the world's origin and, in turn, could use the camera's (“player's”) position to determine if the player was in the bounds of the room or not. I also had to keep track of whether or not the player was inside the room during the last frame to avoid the lights constantly turning on and off.

For this project, I tried to go with the “black box” approach as much as possible. I learned that the more I could abstract objects into logical hierarchies, the easier it was to create a complex scene with more objects without having to worry about the unique properties of each separate model. Once I got the room, hallway, animated objects, inanimate objects, and light sources properly abstracted, putting it all together proved to be relatively simple.

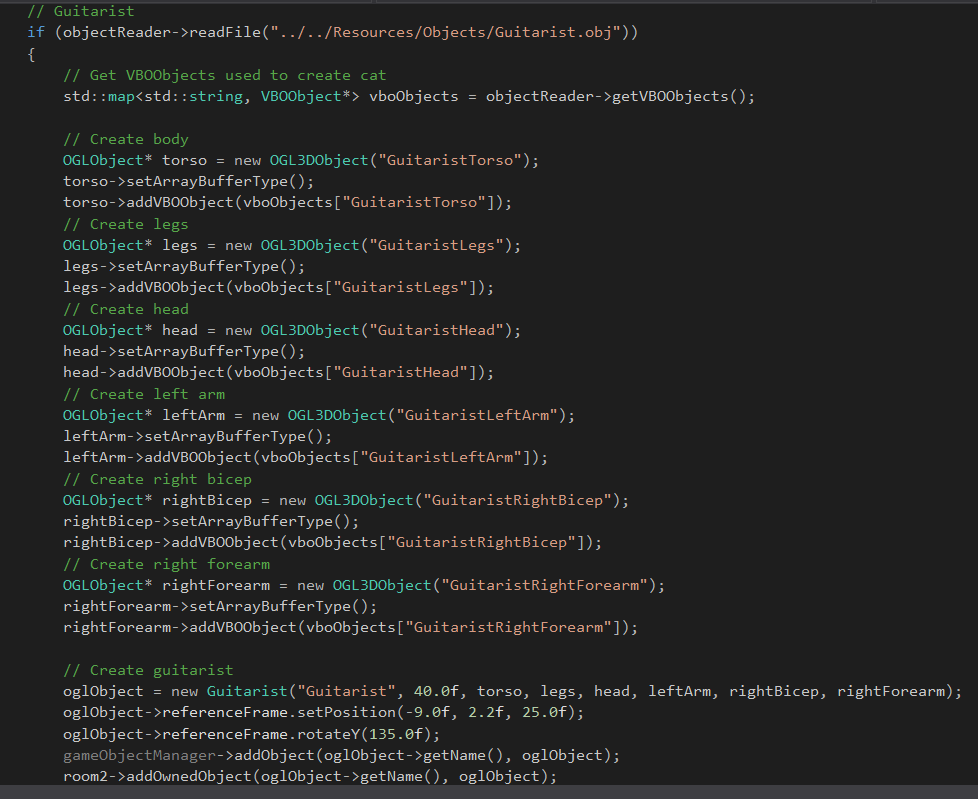
**Code Samples**

Creating room and adding objects to it

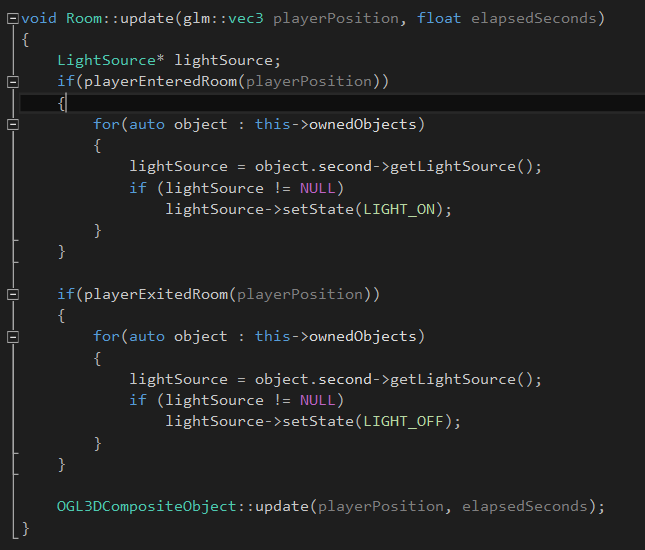




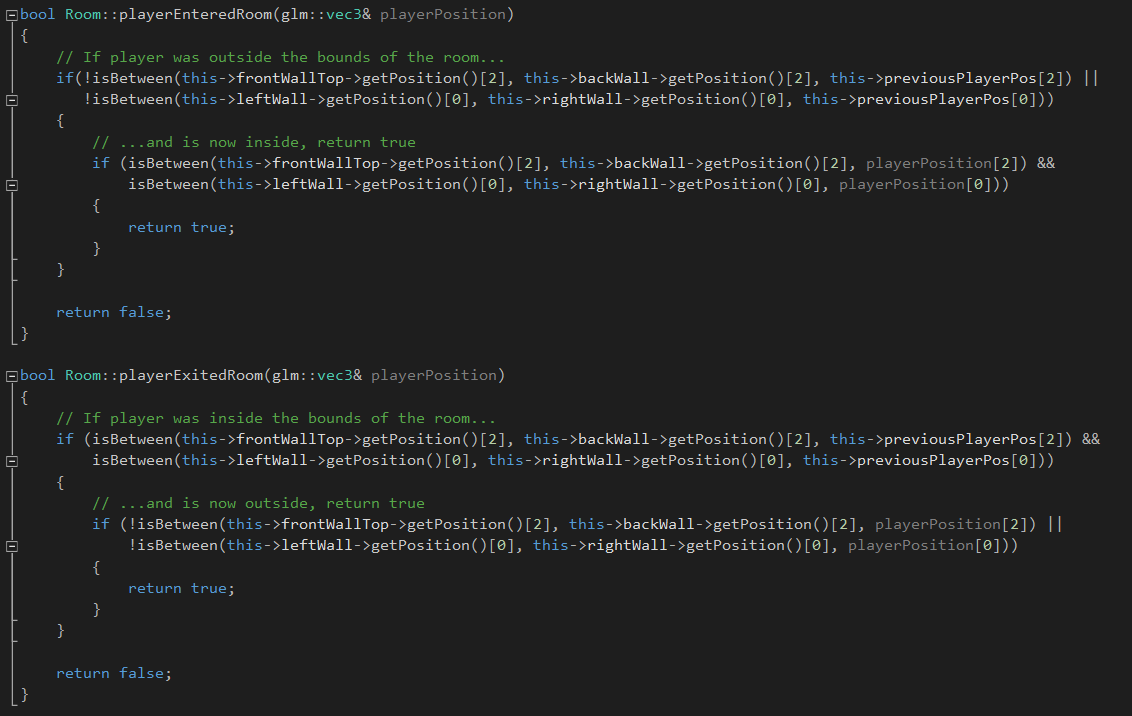
Creating the guitarist as a hierarchical object



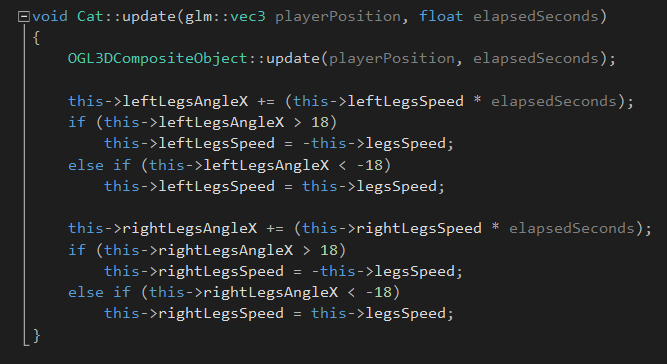
Update method for the room



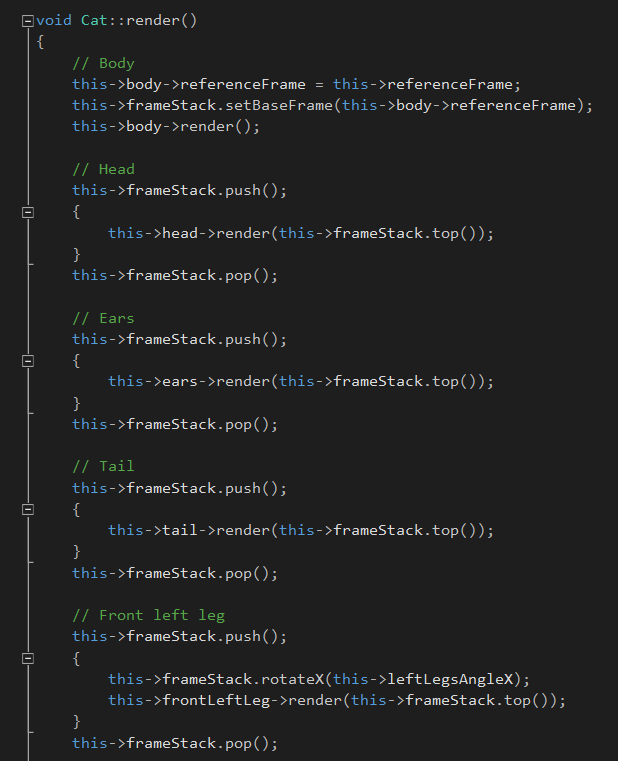
Methods to determine if player has entered/exited the room



Update method for the cat



Render method for the cat



Screenshots of the rooms with the program running

