COMP3811 Computer Graphics Coursework 2

The following report details the design decisions I made when producing my scene.

Implementation Overview

C++ Version

I decided to program using C++ version 17 due to a variety of factors. First and foremost, I have a considerable amount of prior experience with C++ and almost all of that is with version 17. Moreover, it offers numerous advantages over its predecessors. Notably, the Standard Template Library API has become increasingly more powerful which helps to cut down on development time. Finally, when viewed through a purely educational lens it makes sense to use the most up to date version of a language (C++ 20 is yet to see complete compiler support for all it's features).

Build System

Right from the start of the project I knew that I intended to utilise some third-party libraries and as such, I would regularly be making changes within the build system. Therefore, my choice of said system was very important. The build system I feel most comfortable with is that employed by Microsoft's Visual Studio. However, I wanted my scene to be able to be built and run on Linux so this was not an option. I have some experience with CMake but I felt it would have taken a considerable amount of time to write each 'lists' file, not to mention the steep learning curve.

I eventually decided to use a utility called Premake. Premake is not strictly a build system, but rather operates at a higher level: it can be used to generate build configuration files for a wide range of build systems. A set of Lua files are written that describe the configuration of a software project in a platform agnostic way. These files are then used by Premake, along with a requested target, to automatically generate build configuration files for a specific system / platform (e.g. Make or Visual Studio). Premake is a utility I've used before and found to be extremely intuitive. I even have previous experience with using Premake to configure and link to some of the exact Libraries I was intending to use - namely GLFW and ImGui. Naturally, it seemed like the ideal choice. I wrote Lua files for my project (these are the "premake5.lua" files) and then targeted VS2019 when on Windows, and Make when on Linux. I then used these systems to build the libraries and application, but never had to directly interact with eithers's configuration.

OpenGL Version

OpenGL is an API I am very familiar with, especially the modern retained-mode interface. I initially intended to use modern OpenGL for my project but quite quickly decided against it. The reason I changed my mind was because of the initial development time commitment modern OpenGL necessitates.To abstract away framebuffers, index buffers, vertex buffers, shaders, vertex attributes, and then write a renderer with Blinn-Phong shading, would have

required at least 30 hours of programming (in my estimations). Instead, I decided to exclusively use legacy, immediate-mode OpenGL. The only exceptions to this was to utilise, where supported, modern callback debugging, mipmapping, and anisotropic filtering.

Libraries

Whilst I feel somewhat comfortable using Qt, having been introduced to it in last year's UI module, I decided not to use it for this project. The reasons for this decision are manifold. I'm not particularly fond of the deferred pattern of UI rendering that Qt uses (despite its significant performance benefits compared to immediate), and I also dislike slots and signals. Also, Qt requires you to use it's own pre-build stage qmake system which makes adding libraries difficult. Furthermore, Qt does not provide an input polling interface which makes creating something like a smooth free camera (with WASD keyboard controls), as I wanted to do, impossible. However, these reasons notwithstanding, the paramount motive for not using Qt was that I had more experience with alternatives, namely GLFW and ImGui.

As well as GLFW and ImGui, I made use of a few other libraries. I have prior experience with all of these libraries so feel comfortable using them. In the list below I have detailed all the libraries I have used, given a short description of each, and commented on why I chose them:

- GLFW: A windowing library that provides an OpenGL context. It provides a simple way to create a window and OpenGL context, and also deal with events. Furthermore, it's cross platform which means I could use the same windowing library for both Windows and Linux.
- ImGui: A lightweight UI library. ImGui offers an immediate mode pattern of rendering which is very straightforward to use. It also has extensive code samples for integrating with OpenGL and GLFW.
- GLM: A maths library intended for use in OpenGL applications. Having a maths library was vital for working with matrices and vectors in my application. GLM is intuitive to use and extremely fast (making use of Streaming SIMD instruction set Extensions).
- Glad: An OpenGL loader. As previously mentioned, I made use of some OpenGL features that were not available in legacy OpenGL version 1. Therefore, to call the required functions, I had to load the corresponding pointers from the driver. I used Glad for this purpose as it is extremely easy to set up.
- stb_image: Image loading library. When creating OpenGL textures we cannot directly pass in image files. Rather, the images need to be first opened, decoded and loaded into memory. That memory address can then be passed to OpenGL. stb_image is the loader I chose to use as it supports a multitude of file types, and is incredibly easy to include (just one header file) and use.

Version Control

Version control was very important when creating my scene as I was constantly switching between two different machines, a linux and windows computer. I used Git for version control and the repo can be found at https://gitlab.com/ej2000/computer-graphics-cwk2.

Architecture

For the most part I have organised my code into classes. The main class that handles the execution of the program is the Application class. It's responsible for numerous things: at startup, it initialises the windowing system, Input class, Renderer, User Interface (ImGui), and scene, and also creates a window and OpenGL context; during runtime, it calculates the delta time between frames, clears the framebuffer, updates the scene, renders the UI and swaps the front and back buffers; at shutdown, it frees all necessary memory. The Scene class contains all the code responsible for drawing the scene.

Scene Description

My scene mainly depicts a forest and a pair of lumberjacks. It also features a table supporting a spinning top and map. The sections below explain how I designed and implemented all of the techniques requested in the indicative marking bands.

Camera System

Although not requested in the marking bands, I thought it would be immensely useful to implement a free camera. This gives the user an unconstrained view of the scene, and also proved to be a significant aid when constructing it. Having such a camera system was useful when locating mistranslated models, and when determining the specular components of materials.

The operation of the camera is as follows. The user can move the mouse to change the view direction of the camera. They can also press the WASD keys to move the camera towards and away from the view direction, and strafe from side to side. Additionally, to separate out the inputs intended for camera manipulation and those used for the UI, the camera can only be manipulated when the right mouse button is being pressed.

All the camera logic is contained within the PerspectiveCamera class. As the name suggests, a perspective projection is used. That was chosen over orthographic since it takes into account foreshortening, a crucial phenomena in rendering believable 3D graphics. The main pieces of state that the class makes available are the view and projection matrices. Every frame the application retrieves these two matrices and applies them to OpenGL's GL_MODELVIEW and GL_PROJECTION matrices respectively using the glLoadMatrixf function (see Code Snippet 1). Each time the matrices' get methods are called, the matrices are calculated using the private member variables that the PerspectiveCamera class maintains. Every frame the state of the camera is updated by invoking the update method, which in turn, updates these private member variables.

```
glMatrixMode(GL_MODELVIEW);
glLoadMatrixf(glm::value_ptr(m_camera.getViewMatrix()));

glMatrixMode(GL_PROJECTION);
glLoadMatrixf(glm::value_ptr(m_camera.getProjectionMatrix()));
```

Code Snippet 1

The projection matrix is relatively trivial to calculate. GLM offers a function called glm::perspective that takes in a vertical FOV, near clip, far clip and aspect ratio, and then returns a 4x4 matrix (very similar to gluPerspective). The FOV, near clip and far clip parameters are all set to hard coded values within the CameraProjection class declaration, being 45.0, 0.01 and 100.0 respectively. I experimented with using different values for the clipping distances, but eventually settled on the ones mentioned above - I found if the distance between them was too great, then the z-buffer became imprecise and z-fighting occurred. The aspect ratio is more complex as it needs to be consistent with the aspect ratio of the application window, otherwise the scene will look distorted. Therefore, every time the window is resized, the camera class needs to update its aspect ratio. This is done through a chain of callback functions. The main Application class registers a callback method which is invoked when GLFW detects a window resize event. This registration is done via the Window class and can be seen in Code Snippet 2. Whenever the Application's resize method is called the new window width and height is passed in. Then the scene's resize method, and eventually the camera's is called, updating the aspect ratio member variable.

```
glfwSetWindowSizeCallback(m_glfwWindow, [](GLFWwindow* window, int width, int
height)
{
    WindowCallbacks& windowCallbacks =
*static_cast<WindowCallbacks*>(glfwGetWindowUserPointer(window));
    if (windowCallbacks.windowResizeCallback)
        windowCallbacks.windowResizeCallback(width, height);
});
```

Code Snippet 2

The view matrix is a bit more difficult to calculate. The GLM::lookAt function is used to generate the matrix: it takes in a camera position, a position where the camera is looking, and a normalised up vector; it returns a 4x4 matrix. The normalised up vector is simply set as the y-direction, { 0, 1, 0 }. The position where the camera is looking is evaluated as the position of the camera summed with a view direction. This view direction is determined using two parameters, yaw and pitch. Yaw is the angle around the y-axis (left and right), and pitch is the angle around the x-axis (up and down). These two values map nicely to the 2 dimensional cursor space so are updated with how the mouse position changes each frame. To resolve a view vector from these two values the code in Code Snippet 3 is used. This formula is taken from a personal project I completed previously, but that was itself originally based on information from the online Learn OpenGL course (https://learnopengl.com/Getting-started/Camera). The final parameter to calculate is the camera position. The camera position is set to an initial value of { 0, 1, 3 } and then subsequently modified whenever the WASD keys (and right mouse button) are pressed.

Depending on the key being pressed, the position is either moved forward or backwards along the view direction, or forwards or backwards along the right direction - a vector that is perpendicular to the up and view directions, obtained using the cross product operator. See Code Snippet 4 for more details.

```
glm::vec3 direction;
direction.x = glm::cos(glm::radians(m_yaw)) *
glm::cos(glm::radians(m_pitch));
direction.y = glm::sin(glm::radians(m_pitch));
direction.z = glm::sin(glm::radians(m_yaw)) *
glm::cos(glm::radians(m_pitch));
m_viewDirection = glm::normalize(direction);
```

Code Snippet 3

```
if (Input::isKeyPressed(KeyCode::KEY_W))
{
    m_position += m_viewDirection * m_speed * timeStep;
}
else if (Input::isKeyPressed(KeyCode::KEY_S))
{
   m_position += -m_viewDirection * m_speed * timeStep;
}

if (Input::isKeyPressed(KeyCode::KEY_D))
{
   m_position += rightDirection * m_speed * timeStep;
}
else if (Input::isKeyPressed(KeyCode::KEY_A))
{
    m_position += -rightDirection * m_speed * timeStep;
}
```

Code Snippet 4

As discussed previously (and as can be seen in the code snippet above), the camera changes position and view direction depending on user input. This user input could be provided via an event system, but since the camera needed to be smooth and responsive, polling the inputs was a better decision. GLFW provides a simple interface to poll both key and mouse inputs. I created a static class called Input that abstracts this functionality away. It's operation can be seen in Code Snippet 4.

See the "User Interaction" section for details on the camera options the user is able to adjust (e.g. changing the camera movement speed).

Instancing

I used instancing throughout my scene, with every model being constructed from a set of three available meshes, a unit cube, a unit cylinder and a unit octahedron. Instancing was leveraged the most when creating the forest.

I started by writing a static Renderer class with methods that draw the three meshes mentioned above. For information on how these meshes were drawn, see the “Convex / Platonic Object Construction” section. Each of these methods take in a GLM 4x4 matrix - the transform matrix. The methods begin by pushing the current state of the GL_MODELVIEW matrix onto the stack so it can be restored later on. The transform matrix is then multiplied with the GL_MODELVIEW matrix using the glMultMatrixf function. This ensures that when the vertices of the mesh are submitted to OpenGL, they will be transformed by the supplied matrix. Using these methods I can translate, scale and rotate the unit meshes in any way I desire. Just before returning from the methods I pop the GL_MODELVIEW matrix stack. Along with the initial push, these two operations preserve the state of that matrix so it appears consistent to the caller. The methods also take in a material to apply to the objects. More details on materials can be found in the “Materials” section.

I then wrote a drawTree method in the Scene class which, as the name suggests, draws a tree model. In an identical manner to the Renderer draw methods, drawTree also takes in a matrix so the tree can be transformed in any way. The tree model consists of a trunk protruding straight from the ground, and a canopy set atop it. The trunk is constructed from a textured cylinder, and the canopy from a textured cube (see Code Snippet 5).

```
void Scene::drawTree(const glm::mat4& transform)
{
    // Apply the transform

    glMatrixMode(GL_MODELVIEW);
    glPushMatrix();

    glMultMatrixf(glm::value_ptr(transform));

    // Draw the trunk of the tree

    Renderer::CylinderTextureSpecification trunkCylinderTextureSpecification;
    trunkCylinderTextureSpecification.frontFace =
&m_treeTrunkCrossSectionTexture;
    trunkCylinderTextureSpecification.curvedFace = &m_barkTexture;
    trunkCylinderTextureSpecification.backFace =
&m_treeTrunkCrossSectionTexture;

    glm::mat4 trunkTranslation = glm::translate(glm::mat4(1.0f), { 0.0f,
1.4f, 0.0f });
    glm::mat4 trunkRotation = glm::rotate(glm::mat4(1.0f),
glm::radians(90.0f), { 1.0f, 0.0f, 0.0f });
    glm::mat4 trunkScale = glm::scale(glm::mat4(1.0f), { 1.0f, 1.0f, 3.0f });
```

```
    Renderer::drawCylinder(trunkTranslation * trunkRotation * trunkScale,
MaterialLibrary::getMaterial("WOOD"), trunkCylinderTextureSpecification);

    // Draw the tree canopy

    Renderer::CubeTextureSpecification
canopyCubeTextureSpecification(&m_treeCanopyTexture);

    glm::mat4 canopyTranslation = glm::translate(glm::mat4(1.0f), { 0.0f,
2.5f, 0.0f });
    glm::mat4 canopyScale = glm::scale(glm::mat4(1.0f), { 2.0f, 2.0f, 2.0f
});

    Renderer::drawCube(canopyTranslation * canopyScale,
MaterialLibrary::getMaterial("FAUNA"), canopyCubeTextureSpecification);

    // Restore the state of the MODELVIEW matrix
    glPopMatrix();
}
```

Code Snippet 5

From this point, creating a forest was straightforward. I planned out on a piece of paper where I wanted to place trees on the xz-plane, before populating an array in the Scene::drawForest method with these positions. Furthermore, to add some variety, I also specify a rotation angle for each of the trees. I then loop over each pair of location and angle, construct the equivalent translation and rotation matrices (around the y-axis), compute the product of said matrices to get the transformation matrix, and then call the drawTree method with that matrix. The rendered forest can be seen in Figure 1.

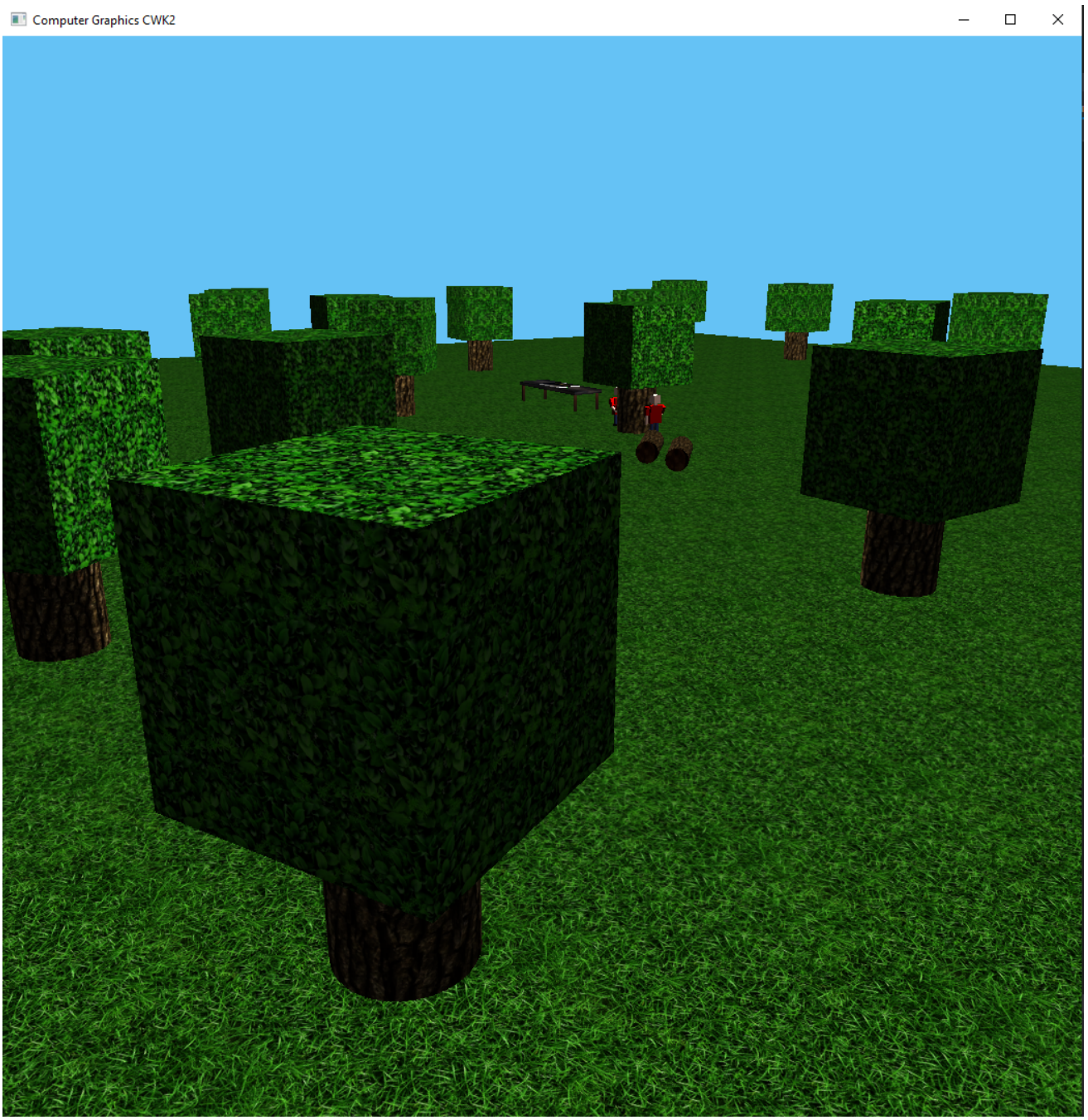


Figure 1

Materials

To encapsulate material properties I created a Material struct. It comprises all the data necessary for the Blinn-Phong reflection model, that is, RGBA ambient, diffuse, and specular components, and a shininess scalar.

To manage different material instances, I also created a static MaterialLibrary class. This class is essentially just a wrapper around a map of material objects. When the class is initialised, a number of materials are added to the map. They can then be retrieved

anywhere in the application using the getMaterial method. See Code Snippet 6 for examples of some of the materials that are added to the MaterialLibrary.

```
s_materials["GREEN_PLASTIC"] =
{
    {  0.05f,  0.175f,  0.05f, 1.0f },
    {   0.1f,  0.35f,    0.1f, 1.0f },
    {  0.45f,  0.55f,   0.45f, 1.0f },
    32.0f
};

s_materials["RED_CLOTH"] =
{
    { 0.25f, 0.0f, 0.0f, 1.0f },
    {  0.5f, 0.0f, 0.0f, 1.0f },
    {  0.3f, 0.0f, 0.0f, 1.0f },
    8.0f
};

s_materials["BLUE_CLOTH"] =
{
    { 0.059f, 0.081f, 0.161f, 1.0f },
    { 0.118f, 0.161f, 0.322f, 1.0f },
    { 0.083f, 0.113f, 0.255f, 1.0f },
    8.0f
};
```

Code Snippet 6

Within the Renderer class, there exists a method called loadMaterial. The Renderer draw methods take in a Material object, and before submitting any vertices, call loadMaterial with that Material object. loadMaterial uses the glMaterialf functions to specify the material properties of a passed in Material object to OpenGL. The subsequent geometry submitted to be drawn is then shaded using this material.

When choosing the properties for each material, I mostly followed the same process. I first researched to determine whether there were any pre-existing sets of values for the desired material. If this was the case, I would copy these values and tweek them where necessary. It was typical for me to increase the ambient component of any copied materials, because I found the original values to be much too dark. A website I used frequently for material values was http://www.it.hiof.no/~borres/j3d/explain/light/p-materials.html. If I couldn't find such values, then instead I would find values for a somewhat similar material, and modify them from there. Often, I would find materials that possessed appealing ratios between the three RGBA components, and a good shininess value, but the wrong colour. In such cases, I used a colour wheel to pick my target colour, and then tweaked the material values accordingly, making sure to maintain the original ratios and the shininess value. I also created some materials that were intended for use with textures. These materials were designed differently as they are applied to meshes where most of the colour comes from the texture. Due to this, these materials mostly have a neutral white colour. As mentioned previously, I found the free

camera to be of significant help when picking properties for the materials, particularly when deciding on the specular components.

By default, OpenGL lights have ambient and specular components. Therefore, I did not have to adjust them to be able to see those elements contribute to the shading.

Animation

There are two animated elements in my scene: the lumberjacks, and the spinning top. The lumberjacks will be covered in the “Hierarchical Model” section of this report, with the spinning top being discussed below.

The spinning top model is constructed from an octahedron, which forms the body, and a cylinder, which forms the handle. I chose to apply a green plastic material because its conspicuous specular component helped to visualise the rotation speed of the spinning top.

Before drawing the two meshes, a rotation matrix is applied to the GL_MODELVIEW matrix. Initially, the current state of the GL_MODELVIEW matrix is pushed onto the stack. Then, a rotation matrix is applied using the glMultMatrixf function. Afterwards, the octahedron and cylinder are drawn, their vertices being rotated in accordance with the rotation matrix. And finally, the original state of the GL_MODELVIEW matrix is restored by popping the matrix stack (see Code Snippet 7). The rotation matrix used is calculated using the glm::rotate function. The arguments to this function include an angle of rotation, and an axis to rotate about. As is characteristic of spinning tops, the rotation axis is simply the upwards y-axis so this is set to { 0, 1, 0 }. The angle of rotation is increased (or decreased if rotating in the opposite direction) per frame to produce the spinning animation. The quantity it is increased by is the product of a rotation speed, and a timestep. The rotation speed is a constant that persists over multiple frames - increasing it, increases the speed of rotation, and vice versa. The timestep is a value that changes every frame and is explained further in the following paragraph. However, the scheme described above possesses a fatal flaw. Namely, the interminable addition to (or subtraction from) the rotation angle will cause an overflow. To avoid this, I wrap the evaluation of the new rotation angle within a floating point modulus function. This modulus ensures that the value of the rotation angle is always between 0 and 360, so no overflow will occur. Due to the frequency of the sine and cosine functions being 360°, this modifying of the angle value does not cause any sudden jumps in the spin, but rather looks like a smooth transition.

```
// Draw spinning top

glm::mat4 spinningTopTranslation = glm::translate(glm::mat4(1.0f), { 1.0f,
0.9f, 0.4f });
// Applying the modulus operator to the rotation angle so an overflow does
not occur
// Angle goes smoothly from 0 to 360, and then immediately back to 0 again
m_spinningTopRotationAngle = glm::mod<float>(m_spinningTopRotationAngle +
(timeStep * m_spinningTopRotationSpeed), 360.0f);
```

```
glm::mat4 spinningTopRotation = glm::rotate(glm::mat4(1.0f),
glm::radians(m_spinningTopRotationAngle), { 0.0f, 1.0f, 0.0f });

glPushMatrix();

// Apply rotation and translation to the spinning top body and handle
glMultMatrixf(glm::value_ptr(spinningTopTranslation * spinningTopRotation));

// Draw the spinning top main body

glm::mat4 spinningTopBodyScale = glm::scale(glm::mat4(1.0f), { 0.05f, 0.1f,
0.05f });

Renderer::drawOctahedron(spinningTopBodyScale,
MaterialLibrary::getMaterial("GREEN_PLASTIC"));

// Draw the spinning top handle

glm::mat4 spinningTopHandleTranslation = glm::translate(glm::mat4(1.0f), {
0.0f, 0.06f, 0.0f });
glm::mat4 spinningTopHandleRotation = glm::rotate(glm::mat4(1.0f),
glm::radians(90.0f), { 1.0f, 0.0f, 0.0f });
glm::mat4 spinningTopHandleScale = glm::scale(glm::mat4(1.0f), { 0.01f,
0.01f, 0.05f });

Renderer::drawCylinder(spinningTopHandleTranslation *
spinningTopHandleRotation * spinningTopHandleScale,
MaterialLibrary::getMaterial("GREEN_PLASTIC"));

glPopMatrix();
```

Code Snippet 7

Timestep is the delta time between frames and is calculated in the main Application::run method every frame. As with all other aspects of animation within the scene, a timestep is provided to the spinning top code. By scaling animation transforms by the timestep each frame (in the spinning top instance, we are scaling a rotation angle that is then used in a call to glm::rotate), the speed of the animation is in now way tied to the computation speed of the hardware. This is a desirable trait, as designing animations that will run at different speeds depending on the hardware, would be incredibly inconvenient.

An image of the rendered spinning top can be seen in Figure 2.

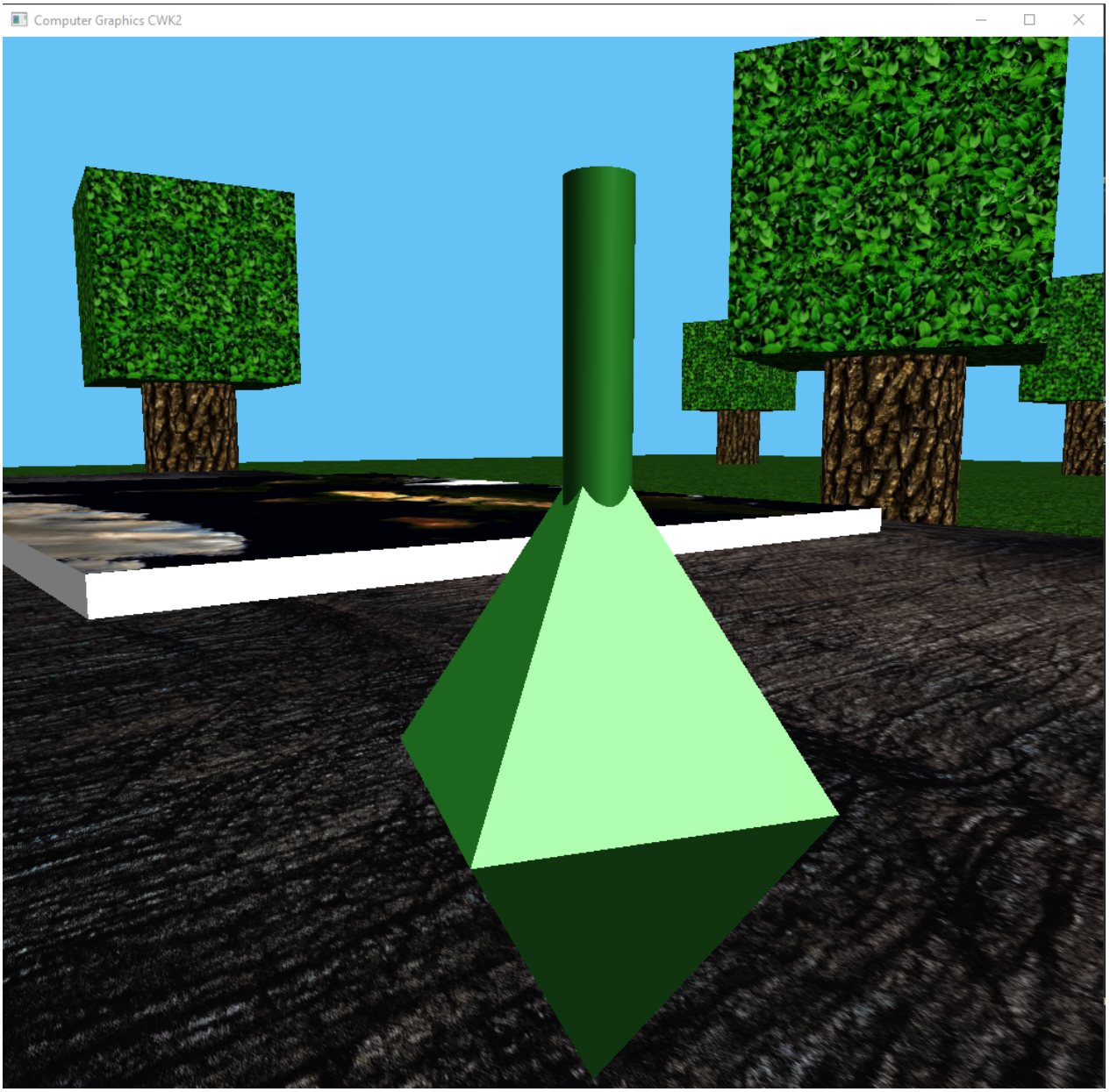


Figure 2

Convex / Platonic Object Construction

As mentioned previously, all the models in the scene are constructed from three meshes - a unit cube, unit cylinder and unit octahedron - and this drawing code can be found in the static Renderer class. The render primitive used for defining these meshes are triangles. I chose triangles because: I have created cube meshes out of triangles before so feel comfortable using them; graphics hardware is optimised for rendering triangles as fast as possible; the majority of modern rendering and modelling is done with triangles, so it makes sense to focus on learning how to construct meshes using that primitive. In all the triangles I drew, I adhered to the OpenGL standard of a counter clockwise winding order. Both for performance reasons, and to verify my winding order was correct, I enabled back face culling (see Code Snippet 8). This proved to be a prudent move as it made identifying erroneous vertex positions easy.

```
// Enable backface culling

glFrontFace(GL_CCW);
glCullFace(GL_BACK);
glEnable(GL_CULL_FACE);
```

Code Snippet 8

Creating the cube mesh was relatively trivial. I hard code the positions of the vertices for the 6 faces, with 2 triangles being used for each. The cube is centered on the origin and has a volume of 1 unit cubed. Working out the normals was very easy since they are just the canonical basis vectors (with some pointing in the negative directions).

Creating the octahedron mesh was a little more difficult because I had never modelled such a shape out of triangles before. I visualised the shape in my head and quickly hard coded the vertex positions and normals. The mesh is split into the 4 triangles that make up the top portion, and the 4 that make up the lower. The model is centered on the origin.

Creating the cylinder mesh was the most involved of the three. It requires drawing circles and a curved surface with perturbed normals. Since I'm using triangles, with straight edges, to represent an infinitely rounded shape, there is always going to be a degree of inaccuracy in my mesh. With this in mind, the drawCylinder methods provide an optional parameter to be specified - a level of detail number (LOD). Increasing this LOD means the cylinder is drawn with more triangles and hence is of higher quality, but this is at the cost of performance (and vice versa). The minimum LOD value for an accurate cylinder will change depending on the size of the cylinder, and proximity to the camera. Therefore, it makes sense to expose this parameter to the user when they invoke the draw method.

The cylinder is centered on the origin and is aligned along the z-axis, meaning the normal of the front circle of the cylinder points in the positive z-direction. In retrospect, I should have instead drawn the cylinder in the more traditional orientation, aligned along the y-axis. This would have reduced some of the rotations I had to perform when drawing some of my models.

The cylinder is drawn using two triangle fans for the circles, and a series of quad strips connecting collinear, corresponding points on those circles. A centerpoint for the front circle is specified as { 0, 0, 0.5 }. A loop then begins that iterates over a sequence of angles from 0° to 360°. The angle increases by a constant value of

$$\text{distanceBetweenCirclePoints} = \frac{360}{LOD}$$

each iteration. The current angle is called theta. At the start of each iteration, two positions are calculated. Polar coordinates are used to first obtain a point on the perimeter of the circle that also lies on the radius line that is rotated by theta degrees clockwise around the y-direction up vector. A second point on the perimeter of the circle is calculated in the same way, apart from the rotation angle is the sum of theta and the distanceBetweenCirclePoints constant. Along with the predefined centerpoint of the circle, this gives three vertices that can be used to define a whole portion of the cylinder. See Figure 3 for a diagram of these

points, and Code Snippet 9. Without modification, these three points are used to define a triangle segment in the front face circle fan of the cylinder. The z-component of each of these points is then minused by 1 to get the three points that define the corresponding triangle segment on the back facing circle fan. The final element is to draw the quad strip that connects the 4 points that lie on the perimeters of the 2 circles. The positions of the vertices for the two triangles that make up this quad are readily available, but the normals needed require further computation. The normals for the vertices that are located on the left of the quad (pointOnCircle1 and its corresponding point on the back circle face) are defined as the difference vector between pointOnCircle1 and frontCircleCenter. The normals for the right vertices (pointOnCircle2 and its corresponding point on the back circle face) are defined as the difference vector between pointOnCircle2 and frontCircleCenter. When the loop eventually stops, the two circle fans will be complete, along with all the quad strips.

This discrepancy between these two normals ensures that different colours will be calculated at the vertices. These colours will then be interpolated over the face of the triangles / quad strip (Gouraud shading). Doing this for every quad strip of the cylinder gives the illusion of a curved surface.

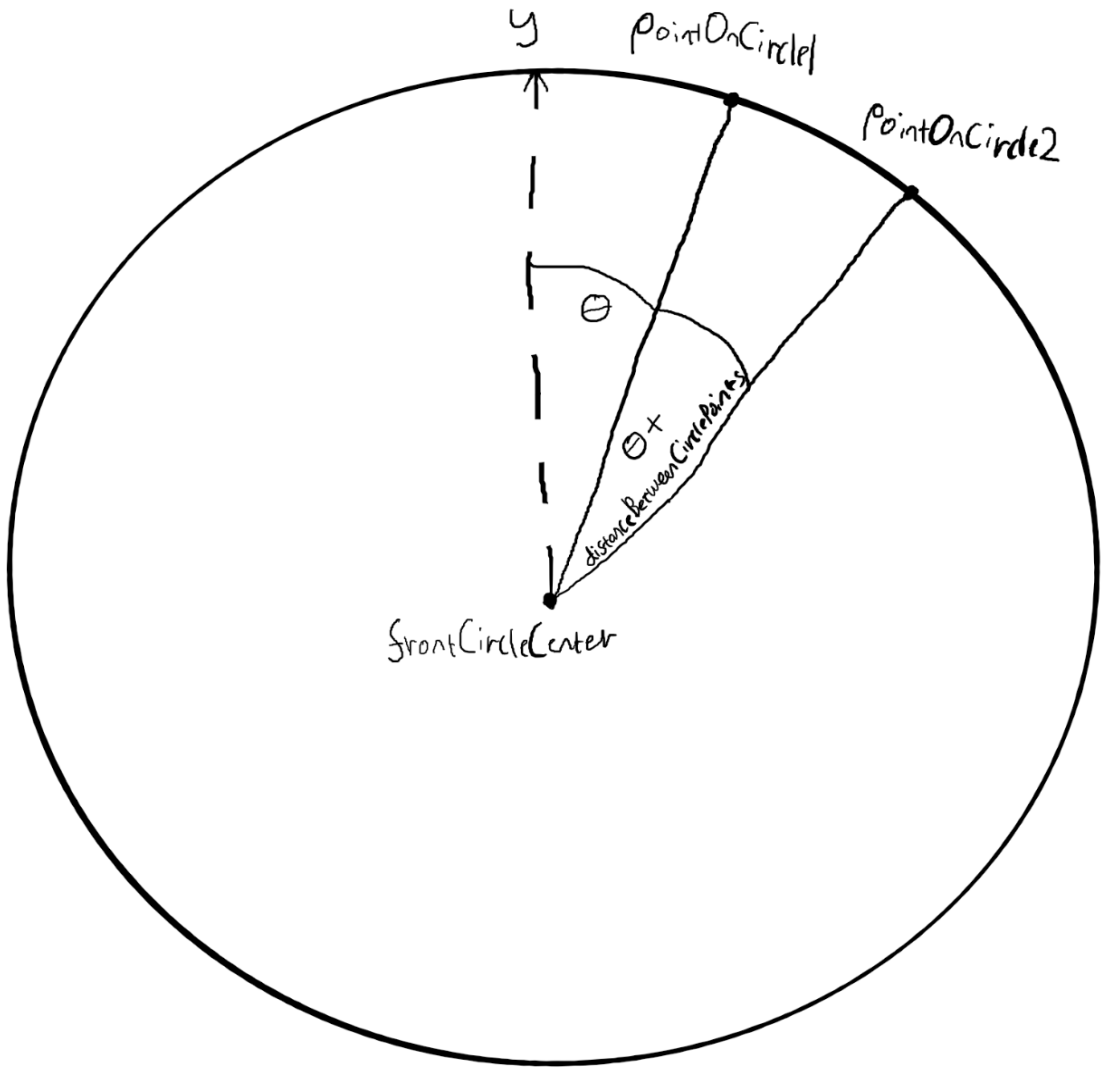


Figure 3

```
for (float theta = 0.0f; theta < 360.0f; theta +=
distanceBetweenCirclePoints)
{
    // Create the cylinder circles using a triangle fan

    glm::vec3 pointOnCircle1 = { 0.5f * glm::sin(glm::radians(theta)) +
frontCircleCenter.x, 0.5f * glm::cos(glm::radians(theta)) +
frontCircleCenter.y, frontCircleCenter.z };
    glm::vec3 pointOnCircle2 = { 0.5f * glm::sin(glm::radians(theta +
distanceBetweenCirclePoints)) + frontCircleCenter.x, 0.5f *
glm::cos(glm::radians(theta + distanceBetweenCirclePoints)) +
frontCircleCenter.y, frontCircleCenter.z };
```

Code Snippet 9

Texture Mapping

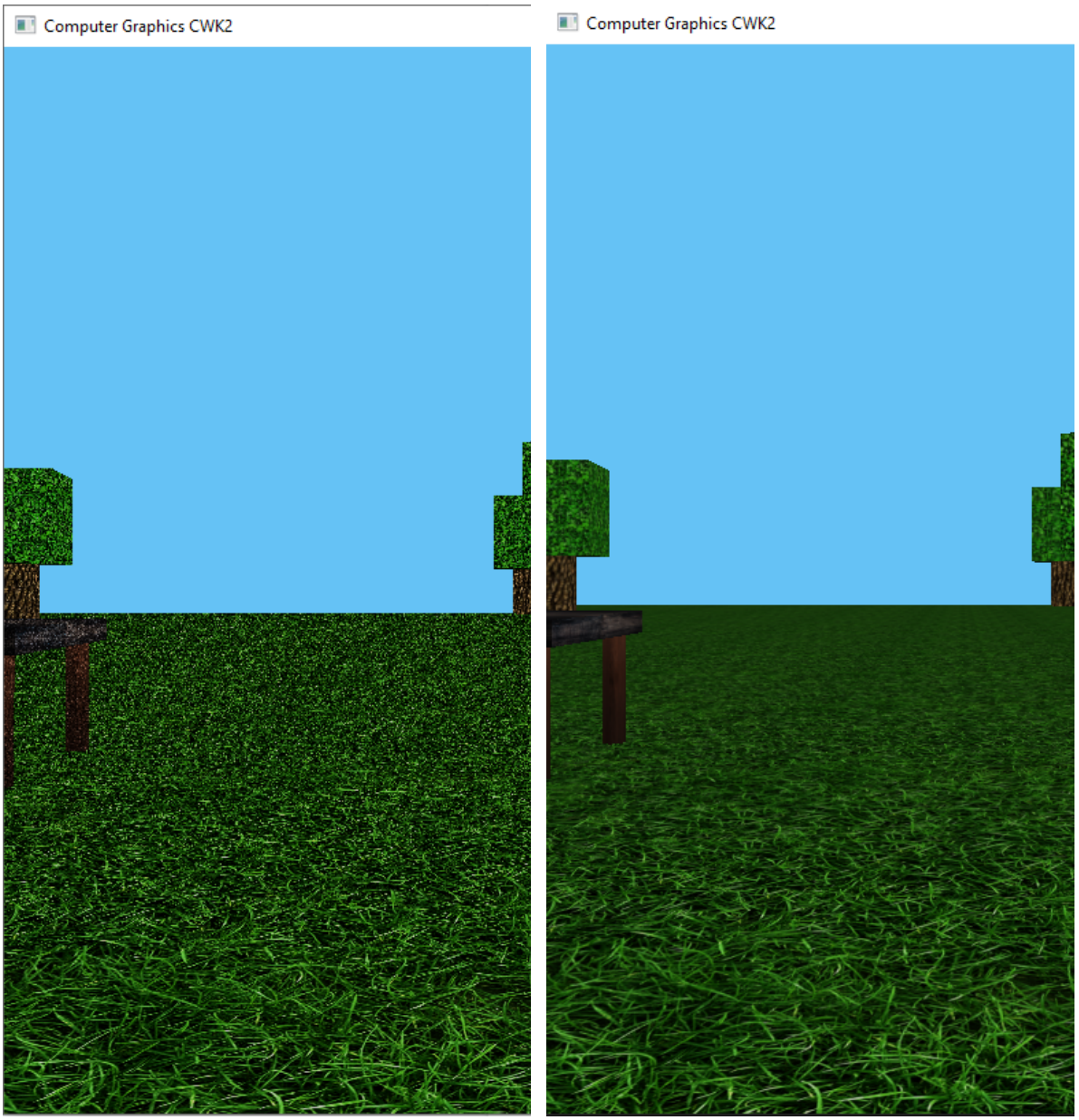
Texture mapping is a concept I have a good understanding of, having had previous experience with loading and using textures within OpenGL. To handle all operations related to the creation and binding of textures, I wrote a Texture class. The init method creates an OpenGL texture. It takes in a TextureSpecification struct object that specifies parameters used in the creation of the texture. Bundling function arguments into these 'specification' structs, is a design pattern I have used throughout my code. These parameters are, a file path to the image to be loaded, a wrapping mode, a minification filter, and a magnification filter. The init method begins by opening, decoding, and loading into CPU memory, the image identified by the supplied file path. As outlined before, the stb_image library is used for this purpose. If successfully loaded, a texture is generated from OpenGL, the image data is passed to the GPU, and the wrapping mode, minification filter, and magnification filter are all specified in accordance with their supplied values. At this point, if the OpenGL version supports it, mipmaps are generated for the texture and used in the minification filter (if LINEAR filtering is requested). Moreover, anisotropic filtering is also enabled if supported (See Code Snippet 10). The combination of both these filtering techniques greatly improves the clarity of textures when they are rendered far from the camera. See Figure 4 for a comparison between basic bilinear minification filtering, and anisotropic filtering with mipmapping. The final task performed by the init method is to free the CPU memory allocated to the loaded image; that data has been copied into VRAM so no longer needs to be resident in main memory. The Texture class also offers a bind method, which assigns the texture as the current GL_TEXTURE_2D to be used. The Texture class's destructor instructs OpenGL to delete the texture from the GPU.

```
// Use mipmaps if supported by the version of OpenGL. If not, just use
bilinear or nearest filtering.
if (Renderer::getOpenGLMajorVersion() > 2)
{
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
getGLMinFilter(specification.minFilter));
    glGenerateMipmap(GL_TEXTURE_2D);
```

```
}
else
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
specification.minFilter == Filter::LINEAR ? GL_LINEAR : GL_NEAREST);

// Use anisotropic filtering if OpenGL version greater than 4.5 (it became a
core feature in 4.6)
if (Renderer::getOpenGLMajorVersion() > 3 &&
(Renderer::getOpenGLMajorVersion() > 4 || Renderer::getOpenGLMinorVersion() >
5))
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY, 16);
```

Code Snippet 10



Bilinear Minification Filtering

Anisotropic Minification FIltering with Mipmapping

Figure 4

Creating the textures is only half the work though, other code is necessary to actually apply the textures to the meshes. The Renderer class has overloaded methods for drawing the Cube and Cylinder meshes with textures. These methods take in either an instance of the CubeTextureSpecification struct, or CylinderTextureSpecification struct. Both structs contain a Texture pointer for each of the faces of their corresponding mesh. Pointers are used because then a nullptr value can indicate faces that don't need to be textured (they were not used so the texture object didn't need to be copied - they only occupy 12 bytes of memory after all). This API enables different faces to have different textures, or some not to be textured at all. I utilised this flexibility when texturing the tree trunks and lumberjack heads within my scene. Specifying the texture coordinates for the cube was extremely easy, with each corner of each face mapping to one of the four bounds of the texture space (i.e. either { 0, 0 }, { 0, 1 }, { 1, 0 } or { 1, 1 }). Calculating the coordinates for the cylinder was more difficult. For the circle faces, the vertex positions on the xy-plane are mapped to the texture coordinates by applying a 0.5 offset. This offset was necessary to transform the values from being between [-0.5, 0.5], to [0, 1]. For the curved face, I visualised unwrapping it onto a quad, and the mapping became easy to see. The x-component of each vertex increases as theta increases, but the y-component is always either 0 or 1. As well as facilitating the specification of textures for each individual face, the methods also provide an optional tiling factor parameter. This tiling factor is multiplied with each of the texture coordinates, and has a value of 1 by default (thus not doing anything). Increasing the tiling factor will increase the texture coordinates. If this is combined with a repeating wrapping mode for a texture, it will be tiled multiple times over the mesh's face. This is a very simple way of adding more detail to a mesh.

Some notable examples of texture mapping in my scene are the following. To avoid an overly blurry look to the grass in my scene, I textured the ground mesh with a tiling factor of 25 and utilised a repeat wrapping mode. The downside of this approach is that when viewed from a distance, repeating patterns are easily discernible. Another example of texturing is in the design of the tree trunks. To give them a more authentic look, I textured the curved face and circular faces with two different textures: bark for the former, and a tree cross section for the latter (See Figure 5). I used the Mercator-projection image to create a (very zoomed out) map on the table. I imagined this map could be consulted by the lumberjacks to see which trees to target in the surrounding area. The images of Marc's face and Markus's face are used to add some personality to the lumberjack models.

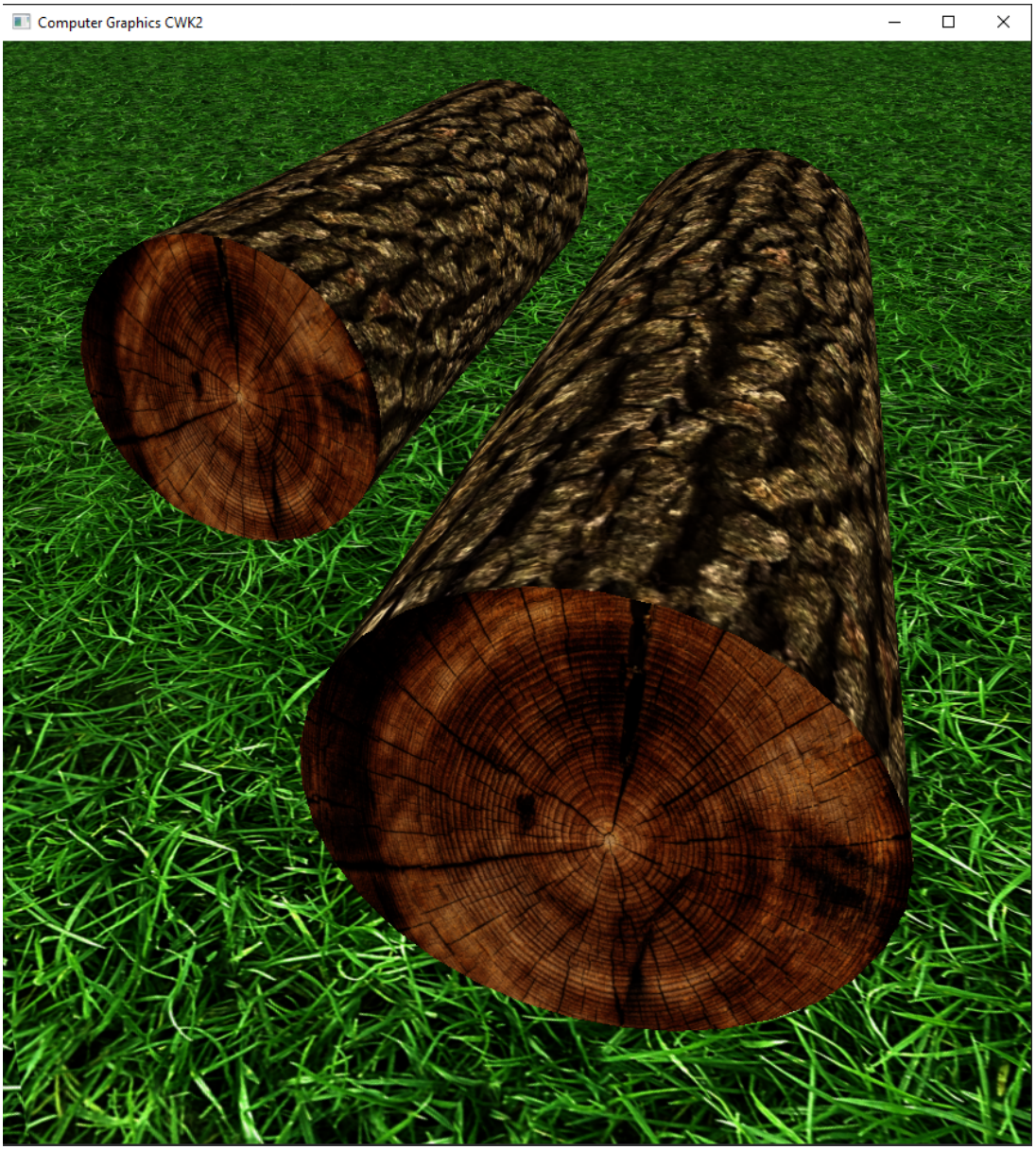


Figure 5

Hierarchical Model

I used hierarchical modelling to create the two lumberjacks in the scene. The lumberjacks move in a circular rotation around a tree, using their hatchets to chop it down (see Figure 6). All the code related to the lumberjacks is encapsulated within the Lumberjack class. Every frame, the update methods of the two lumberjacks are called, which draws them on the screen. The lumberjack model is solely created out of cube and cylinder meshes, with appropriate materials and textures being supplied. For example, a cube mesh and leather material are used to draw the lumberjack's boots, whilst a cylinder mesh and wood texture are used for the ax handle. As is inherent in hierarchical modelling, translation and rotation matrices are constantly being multiplied with the GL_MODELVIEW matrix, and popped and pushed from the stack, all for the purpose of moving into different coordinate systems when drawing the meshes.

Figure 6

The animation of the lumberjack is achieved by altering the various rotation angles every frame. However, this modification can not just be an increase or decrease from 0° to 360°, as was employed with the spinning top. The chopping motion requires a more complex behaviour. Two parts of the lumberjack are animated: the whole left arm, rooted at the upper arm; and the left lower arm and ax, rooted at the lower arm. The whole arm starts at the lumberjacks side, then smoothly raises to an angle of 55°, before smoothly lowering to be at the lumberjacks side once again. By starting and ending at the same spot, this behaviour can be looped indefinitely and will always look smooth. A mathematical function is used to generate the sequence of angles necessary to exhibit the described animation. It can be found in the Lumberjack::getRotationFunctionValue method and is the following:

$$y = -1 * (2x - 1)^2 + 1$$

As can be seen in Figure 7, this function expects x to be in the range [0, 1] and returns a value between [0, 1]. The lower left arm and ax have a different animation that demands its own function. The lower left arm and ax start inline with the upper arm limb. Then they are smoothly raised to an angle of 70°, before quickly lowering to an angle of ~50° and back up to 70° again. This more jerky motion models hitting the hatchet into the tree. Finally, the lower arm and ax are smoothly lowered back inline with the upper arm. The mathematical function that describes this behaviour is found in the Lumberjack::getJerkyRotationFunctionValue method and is the following:

$$y = (-1.0309 * (2x - 1)^2 + 1.0309) + min(\{0, (10 * (2x - 1)^2 - 0.3)\})$$

As can be seen in Figure 8, this function also expects x to be in the range [0, 1] and returns a value between [0, 1]. Both of these functions were created in a graphing calculator, through lots of trial and error. By multiplying the output of each function with the apex angle that needs to be reached, the current angle of rotation for both animations can be obtained. The final piece of the animation is to generate a number that goes from 0 to 1 smoothly, and then immediately back to 0 again. This is used as input to the mathematical functions above. Such a number is calculated per frame in a very similar manner to the spinning top code (see Code Snippet 11).

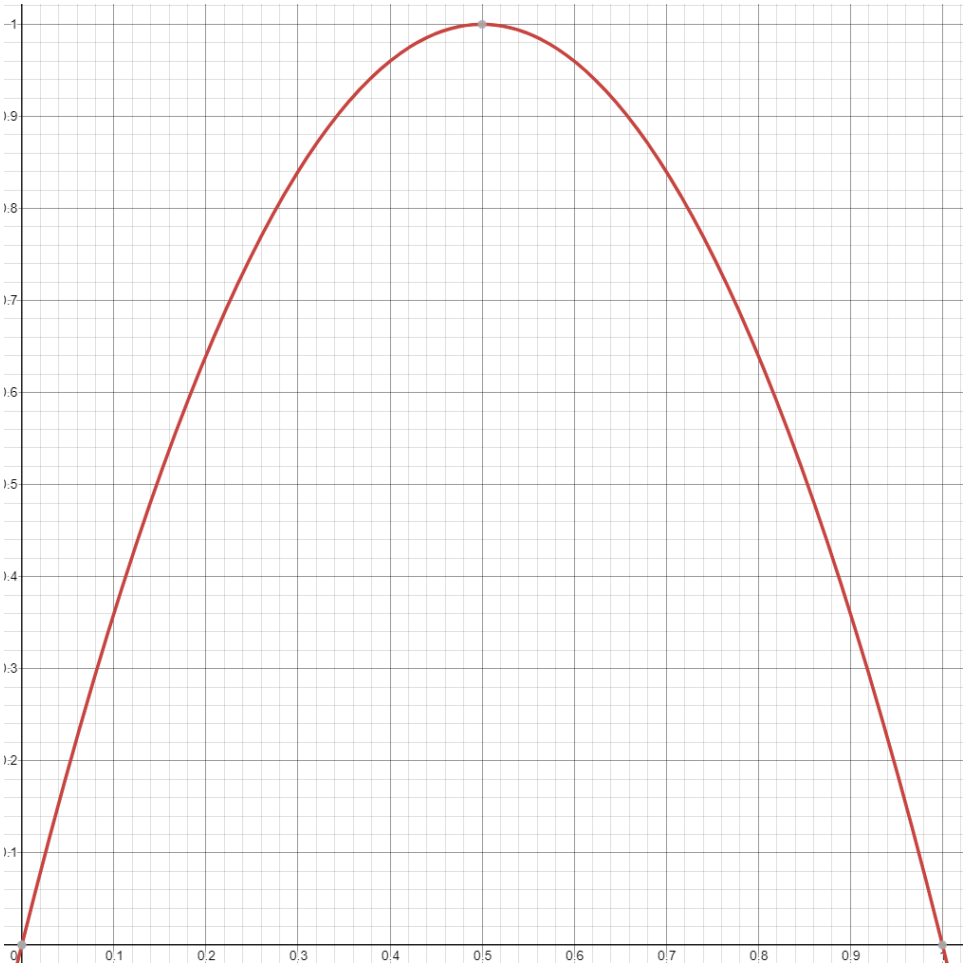


Figure 7

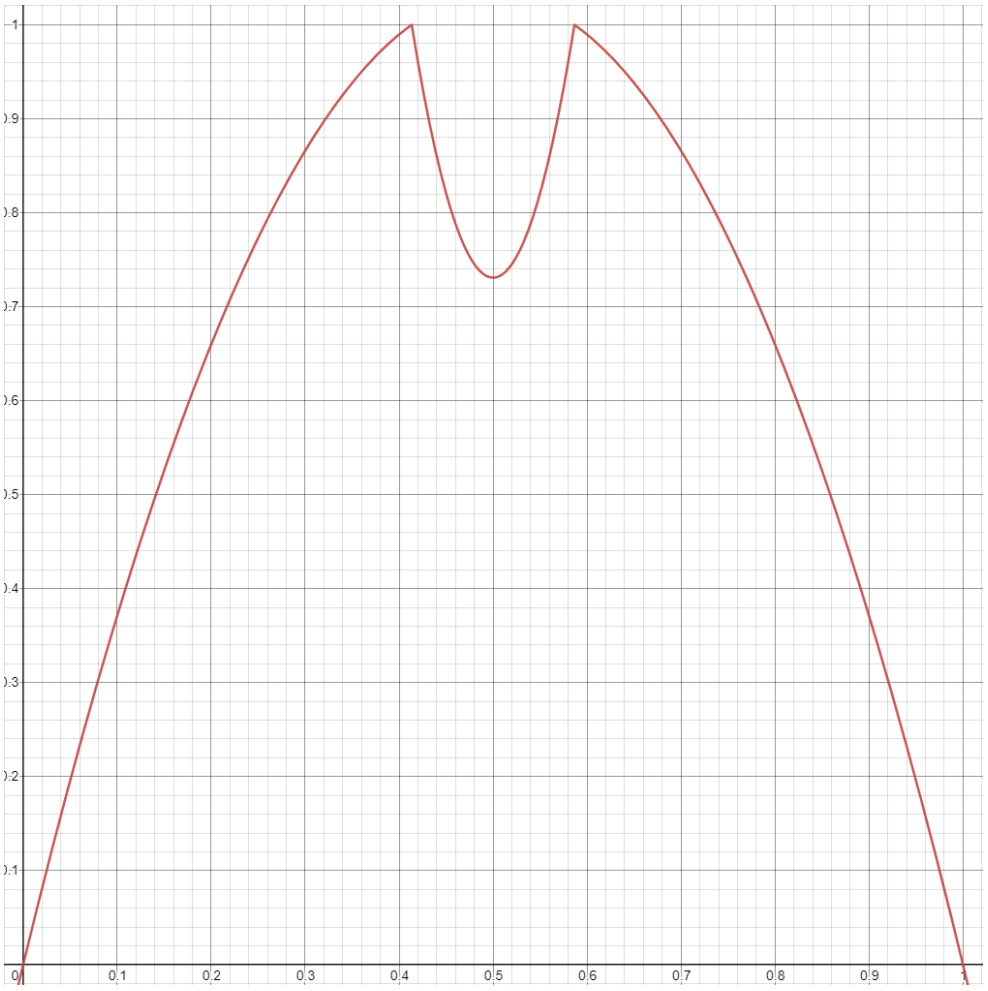


Figure 8

```
// m_currentRotationValue goes smoothly from 0 to 1, and then straight back
to 0 again
// Is used as input into animation curves that return values that can be used
to calculate the current rotation of limbs in the model
m_currentRotationValue = glm::mod<float>(m_currentRotationValue +
(m_animationSpeed * timeStep), 1.0f);
```

Code Snippet 11

From this point, the final piece of animation is to just make the two lumberjacks rotate around the centerpoint of the tree. This is done by calculating two sets of translation and rotation matrices. An angle that increases smoothly from 0° to 360°, and then immediately back to 0° again each frame, is used for one of the lumberjacks. The other lumberjack's angle is 180° out of phase with this one. The two translation points are then obtained using polar coordinates. The phase difference ensures the two lumberjacks are opposite one another. The rotation matrices also use the aforementioned angles, rotating the lumberjacks around the y-axis so they always face inwards to the tree. The two sets of matrices are multiplied together and used to update the transform of the two lumberjacks via the Lumberjack::setTransform method. These translations are calculated in the Scene::drawLumberjacksScene method (see Code Snippet 12).

```
// Draw the lumberjacks

// Calculate the position of the lumberjacks so they rotate around the center
point (the tree)

const float circleRadius = 0.84f;
m_lumberjacksRotationAngle = glm::mod<float>(m_lumberjacksRotationAngle +
(timeStep * m_lumberjacksRotationSpeed), 360.0f);
// Using polar coordinates to get positions on a circle around the tree
glm::vec3 lumberjack1Position = { circleRadius *
glm::sin(glm::radians(m_lumberjacksRotationAngle)), 0.0f, circleRadius *
glm::cos(glm::radians(m_lumberjacksRotationAngle)) };
glm::vec3 lumberjack2Position = -lumberjack1Position;
glm::mat4 lumberjack1Translation = glm::translate(glm::mat4(1.0f), {
lumberjack1Position });
glm::mat4 lumberjack2Translation = glm::translate(glm::mat4(1.0f), {
lumberjack2Position });

// Calculate the rotation of the lumberjacks so there always facing the
center of the circle

glm::mat4 lumberjack1Rotation = glm::rotate(glm::mat4(1.0f),
glm::radians(m_lumberjacksRotationAngle - 180.0f), { 0.0f, 1.0f, 0.0f });
glm::mat4 lumberjack2Rotation = glm::rotate(glm::mat4(1.0f),
glm::radians(m_lumberjacksRotationAngle), { 0.0f, 1.0f, 0.0f });

m_lumberjack1.setTransform(lumberjack1Translation * lumberjack1Rotation);
m_lumberjack1.update(timeStep);

m_lumberjack2.setTransform(lumberjack2Translation * lumberjack2Rotation);
m_lumberjack2.update(timeStep);
```

Code Snippet 12

User Interaction

There are an abundance of ways for the user to interact with the scene. They have options related to the camera, lights, spinning top, and lumberjacks.

As detailed in the “Camera System” section, the user can operate a free camera when holding the right mouse button. Using the UI they can change both the speed of the camera, and the sensitivity. Obviously changing the speed of the camera will make it move quicker or slower, whilst changing the sensitivity modifies the responsiveness of the mouse when manipulating the view direction. Moreover, the user can enable an option to lock the camera’s y-position. Doing this gives a more FPS style of camera.

Within the UI, the user can enable an option to visualise the lights in the scene. They can also edit properties of individual lights. A drop down list lets them choose which of the 3 lights in the scene they are currently editing. One modifiable property is the light's position, and another is the colour of its specular and diffuse component. See Figure 9 for an image of these being modified. To make the modification of individual lights easier, I chose to abstract away the details of each light into a Light class.

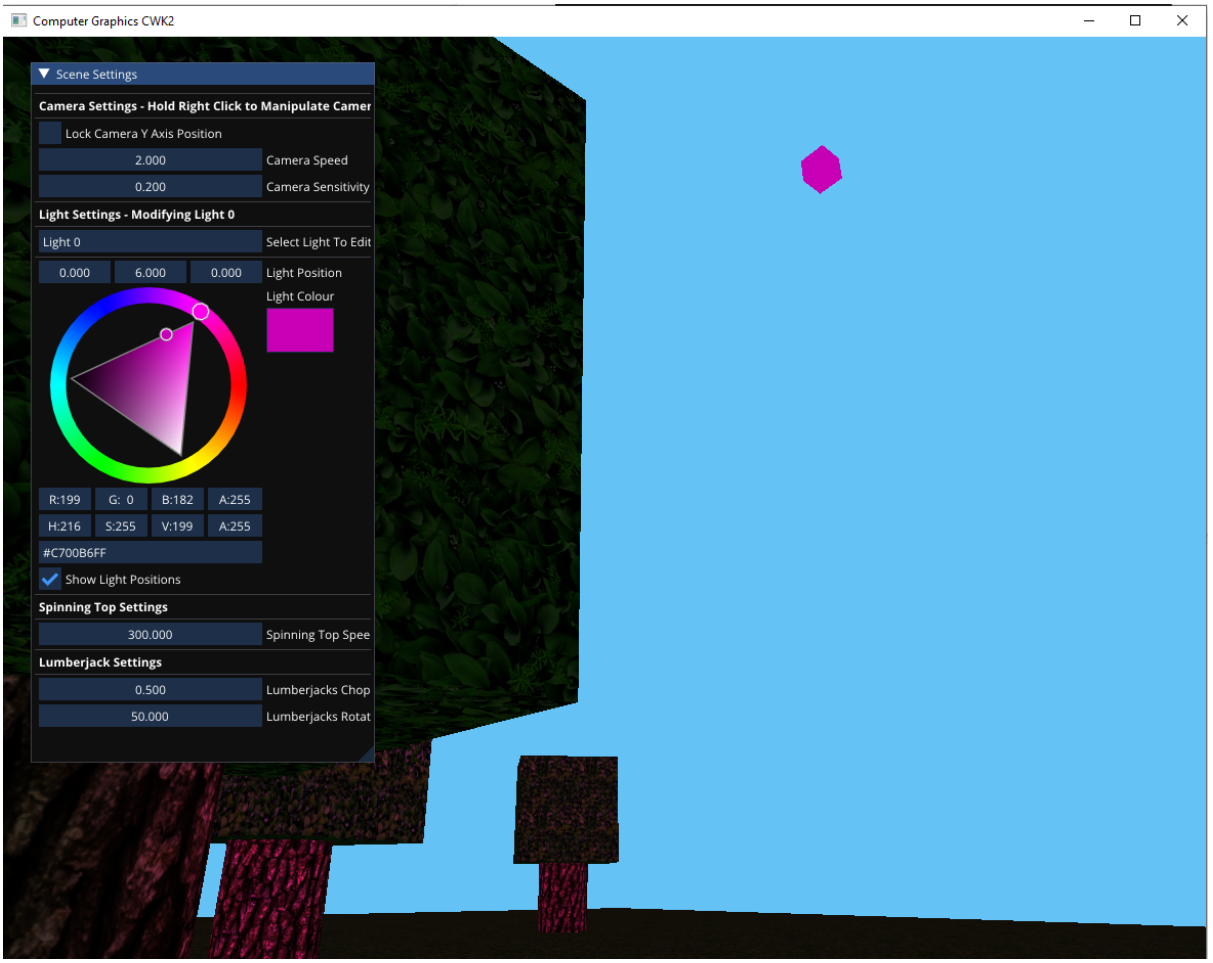


Figure 9

The only setting for the spinning top is to modify its rotation speed. Similarly, a user can also modify the lumberjacks' rotation speed around the tree, as well as the chopping speed.

See Figure 10 for an image of all the options for user intractability via the UI. As touched on before, I used ImGui to create my User Interface, and all the code to generate the aforementioned widgets can be found in the Scene::onUIRender method.

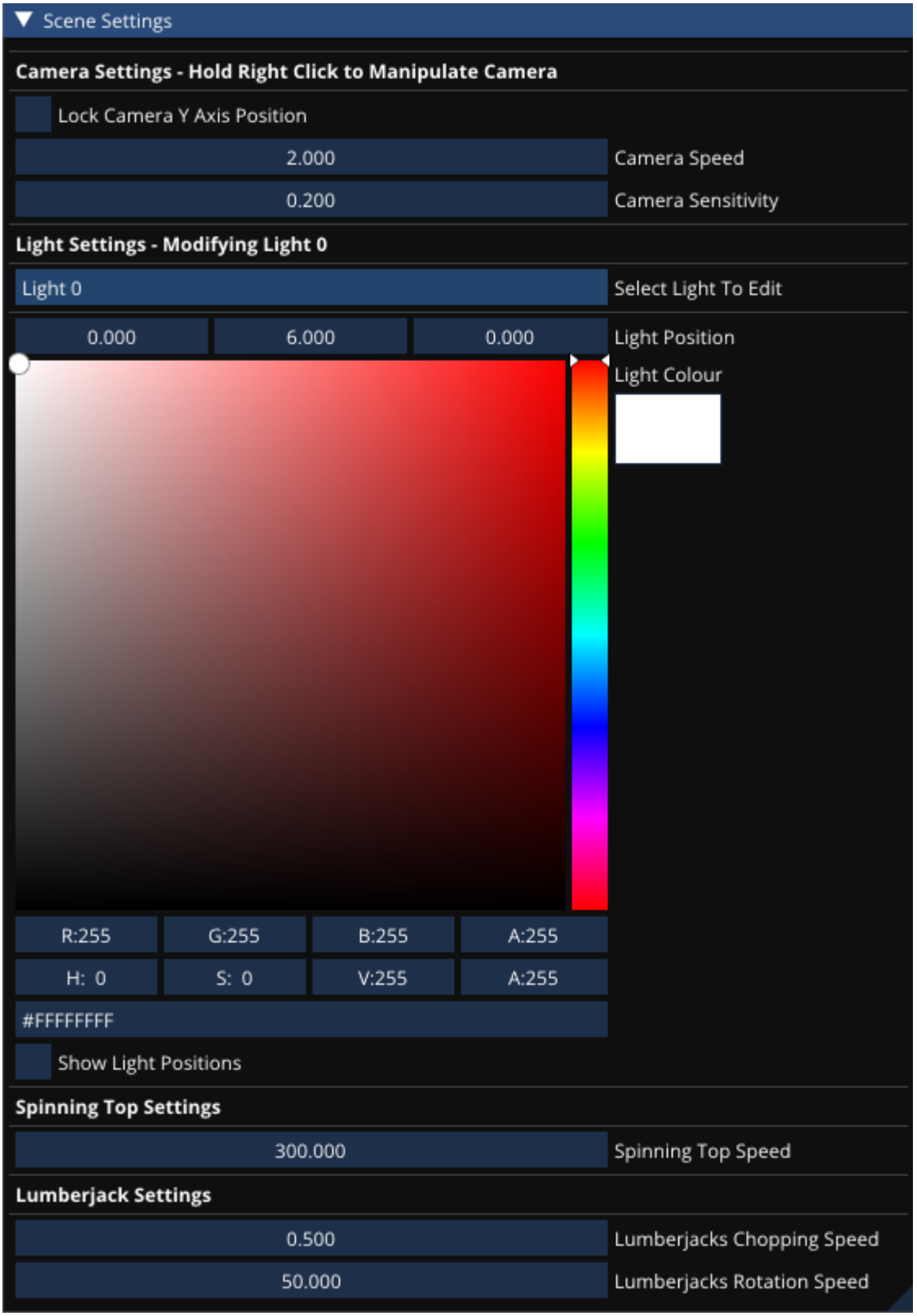

Scene Settings
Camera Settings - Hold Right Click to Manipulate Camera
Lock Camera Y Axis Position
2.000
Camera Speed
0.200
Camera Sensitivity
Light Settings - Modifying Light 0
Light 0
Select Light To Edit
0.000
6.000
0.000
Light Position
Light Colour
R:255
G:255
B:255
A:255
H: 0
S: 0
V:255
A:255
#FFFFFFFF
Show Light Positions
Spinning Top Settings
300.000
Spinning Top Speed
Lumberjack Settings
0.500
Lumberjacks Chopping Speed
50.000
Lumberjacks Rotation Speed

Figure 10