Computer Graphics: Coursework Report

psyak12 14302781

May 2020

Contents

1 Introduction

To summarise this graphics environment, there is a randomly generated mountainous terrain with randomly generated clouds, water, a sky with randomly generated stars, and randomly generated houses on grassy areas. My approach was to allow each aspect to be customised and therefore randomly generated which meant a lot of dynamically changing and adjusting properties. To my best ability and knowledge I have tried to keep everything built with performance in mind, and in general I wanted to create an interesting, low-poly design, scene with attention to detail.

2 Modelling

2.1 Generated Terrain

The terrain is made using a mesh of triangles in a row-column grid layout on the x-z axis. The rows and columns are set unit sizes apart, however the height is based on Perlin Noise. This noise allows random but smooth changes between heights close to each other allowing for natural-looking terrain generation. I implemented a class to deal with the generation of Perlin Noise values based on the different parameters such as the number of octaves, size of the noise map, and roughness of the values.

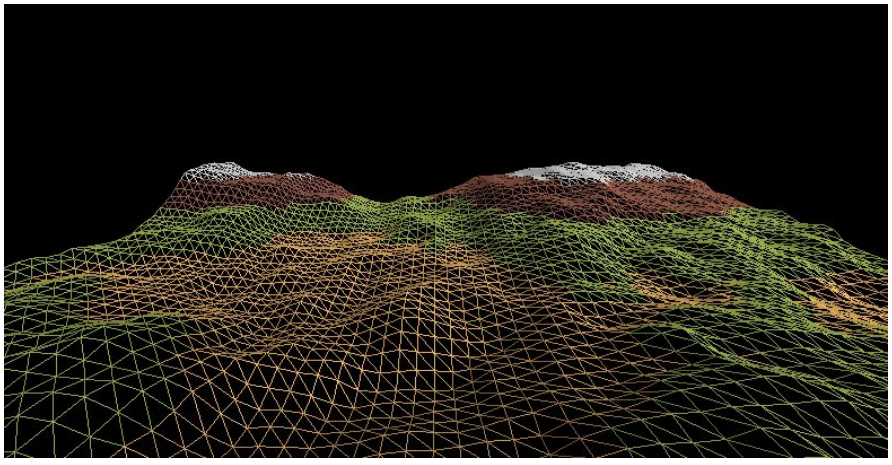

Figure 1: Terrain Mesh

My intention was to create this mesh without blending the colours, aiming for a low-poly design where each triangle can be seen as independent colours and shades of light. This meant that each triangle had to be constructed separately, duplicating each point for each triangle.

As seen in figure 2, water is not drawn everywhere. Water is only drawn if it is not covered by terrain to optimise performance.

The terrain was sectioned into different height levels. This was to determine whether a piece of land was to be underwater, grassland, brown earth, rocky mountains, or the icy snow-tops of these mountains. The height of each point therefore determines the colour of each point.

2.2 Generated Water

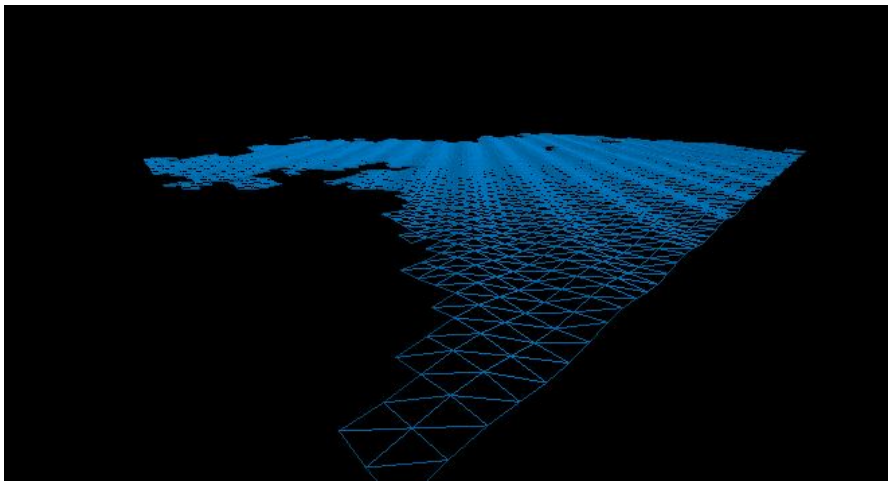

Figure 2: Water Mesh

The water is generated with a similar triangle mesh structure to that of the terrain. The heights this time are based on a sine wave, calculating the height using,

```
height = waveHeight * sin(row + col + wavePhase)
```

where the wavePhase is an incremented value with time5.1 to move the wave along in a diagonal direction (row + col) and waveHeight is the maximum height of each wave from the water level.

These heights are adjusted, but not recreated. The points are still initialised when the water is first created.

The water is placed at water level in the terrain, which is done by asking the terrain where it placed the water level relative to itself, and normalising that to the world coordinates.

```
float Terrain::WaterLevel()
{
    //grassLevel is a percentage, maxHeight and minHeight are
    //the bounds of the terrain in world space.
    return grassLevel * (maxHeight - minHeight) + minHeight;
}
```

2.3 House

A house is made from many elements that come together. Fundamentally, each house is made up of a roof, chimney, door, the foundation, and the walls, which are further made up of hexagonal prisms as the wooden logs.

2.3.1 Walls

As mentioned, each wall is made of hexagonal prisms acting as the wooden logs. The points of each hexagon are calculated using trigonometry and due to the increasing number of logs as more houses are placed, this resulted in a lot of sin() and cos() function calls and decremented performance. These points are now therefore pre-calculated for each house and read from an array whenever needed for each logs.

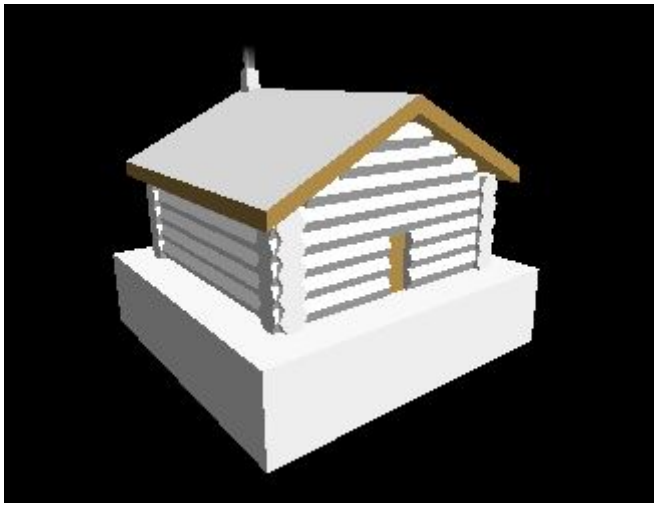

Figure 3: House Model without Textures

To draw a wall, each log is drawn after translating upwards by the diameter multiplied by the log spacing. This is done after calculating what the diameter of each log is based on the height of the wall and number of needed logs.

Once all logs are drawn for a wall, the next wall is drawn after translating to the next corner of the house and rotating 90 degrees. This is done for three walls. The front facing wall, and the last drawn wall, is more complicated due to the door.

2.3.2 Door and its Wall

The door is specified in terms of its width as a float, and height as the number of logs up it is, and the thickness of the wall since it is a 3D box not just a 2D plane. This is set to half the logs of the wall plus 1.

The wall for the door has to have a cutout for the door. This is done by drawing logs until the start of the door which is half of the length of a wall minus half of the door width. This is only done for the first n logs from the ground that would obstruct the door and logs above are drawn as normal. The logs on the other side of the door must also be drawn and these are drawn by translating to half of the length of the wall plus half of the door width and drawn with the same length as the logs on the original side.

2.3.3 Roof

The roof is created based on the roof height, side length of the house, and bevel width. These parameters are enough to create the correct slant and calculate the angle at which it slants. The latter is important to know when and where to start the logs in the triangle above the walls that is formed by the roof sides. These logs must also stop before reaching the other side of the roof, and so the length of each of these logs is the length of a wall take away 2 times the amount it is offset by.

```
void House::DrawTriangleRoofWall(float logDiameter, float sideLength,
float roofHeight, float logSpacing)
{
    //Angle that the roof makes with the x axis
    float angle = atan(roofHeight / (sideLength / 2));

    //How far we need to move to draw the next log up
    float offset = (logDiameter * logSpacing) / tan(angle);

    float cumualtiveOffset = 0.0f;

    //Start at the bottom, draw logs until top of roof is reached
```

```
        for (float y = 0; y < roofHeight * logSpacing; y += logDiameter * logSpacing)
        {
            glTranslatef(0.f, 0.f, -offset);
            cumualtiveOffset += offset;
            DrawLog(logDiameter / 2, sideLength - (cumualtiveOffset * 2), 6);
            glTranslatef(0.f, logDiameter * logSpacing, 0.0f);
        }
    }
```

2.3.4 Chimney

The chimney is a simple model made up of two boxes, a thicker one towards the bottom and a thinner one at the top, extending past the roof of the house, where the height of this chimney is specified by adding the wall and roof heights. This structure is translated to the back and middle of the house. The top part is the specified chimney radius and the bottom part is proportional to that by a multiplier.

2.3.5 Foundation

The foundation is simply a box, representing reinforced stones or cement on which the house lays. This is to level the terrain underneath the house. This means that the terrain needs to be big enough to cover the distance between the maximum of the 4 points the house lays on, and the minimum. These values are pre-calculated in the terrain when it generates the house positions and the scene reads from these values to pass through the height of the foundation into the constructor of the house.

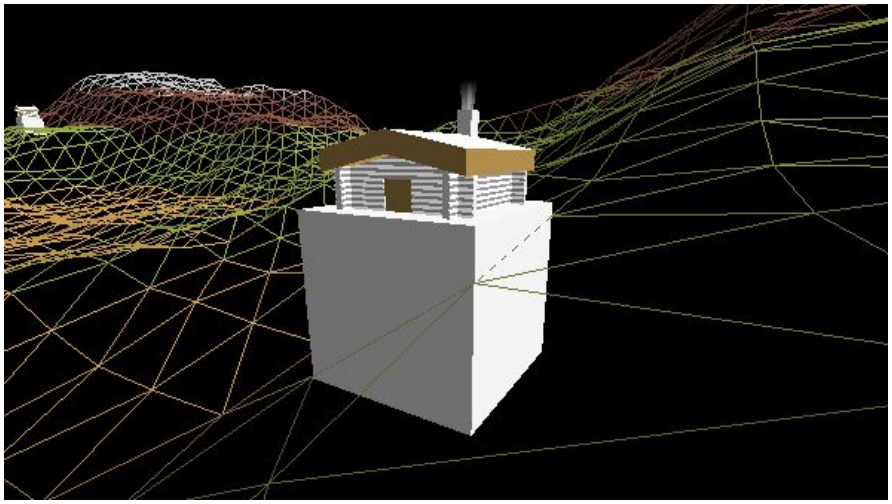

Figure 4: Foundation Ranging from Min to Max Point

Most parameters that control the final output of the house are fed through the constructor, giving the caller a lot of customisation on what the house will be like. These include, but are not limited to, the wall height, the roof height, the number of logs for the walls (diameter is then calculated to correctly size the logs), the density of the logs, the length of the house, the way the house is facing, the height of the foundation, how much the roof overhangs over the walls, and how thick the roof is.

2.4 Clouds

A cloud is made up of n cloud bubbles, specified by the constructor. Each cloud bubble has a maximum and minimum radius which the cloud then chooses randomly for each cloud bubble. This maximum and minimum radius is also passed to the cloud by the constructor.

Each cloud is drawn a random distance away from the radius, from 0 to half the terrain side length, in a random 360 degree direction.

One cloud bubble is always drawn at the centre position of the cloud. The rest of the cloud bubbles are randomly drawn away from the centre. Each bubble stores its own position, along with but not

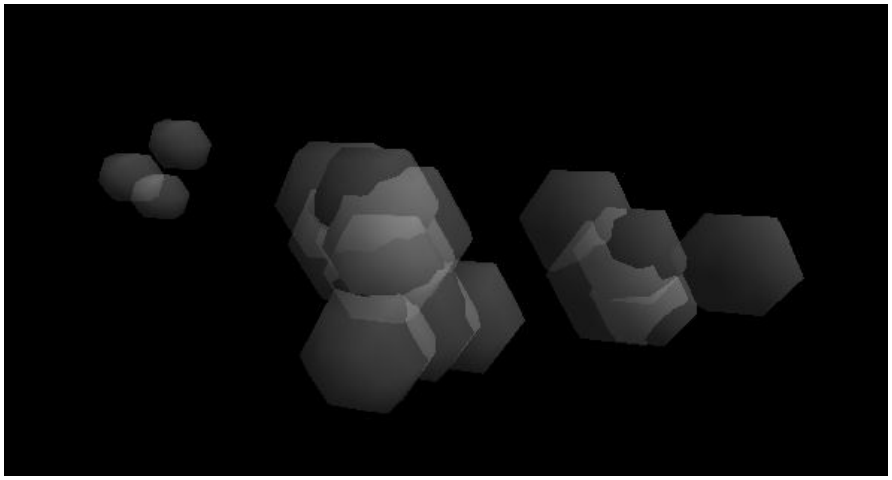

Figure 5: Randomly Generated Clouds

exclusive to, its radius and colour, so when a cloud is generated it creates and stores each bubble that it then draws in its Display() function. The random position is calculated by choosing random x, y, and z values between the max radius and the negative max radius.

Since clouds have transparency, they are added to the scene last so that they don't cover objects drawn before them as if they were opaque.

2.5 Sun, Moon, and Stars

The sun, moon, and stars, all rotate around the world, modelling a realistic sky scene. The sun and moon are both created using subdivision (so that they can be textured), and the stars are simple, bright white, 'glutSolidSphere()'s. All of these are drawn far from the scene to prevent parallax which would be an obvious giveaway that they are closer to the scene than what is realistic.

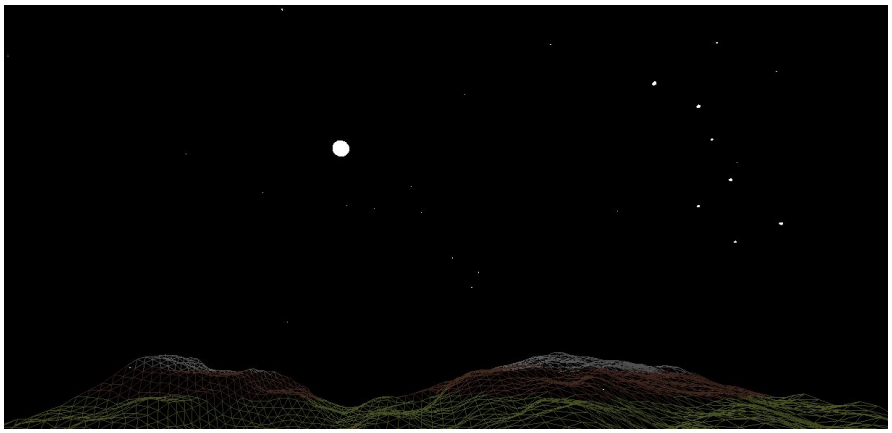

Figure 6: Example Sky with Moon and Stars

2.5.1 Sun and Moon

Both the sun and moon are subdivided only twice, which is enough to create the required spherical shape. When looking up the perceptive size of each I learned that they are actually about the same size in the sky due to the fact that the sun diameter is 400 bigger and the sun is 400 times further away from us. For this reason I have made the moon and sun the same radius and distance from the terrain.

Where they are in the sky is based on the sin() and cos() of the run-time. To make these appear in different places, there is a run-time offset to both, setting them apart from each other. The method

of using sin() and cos() to find the y and x coordinates must be used as opposed to rotating around the z axis and translating up due to GL_LIGHTS and the fact that their x and y coordinates must be specified.

2.5.2 Stars

The stars are all around the terrain at all times. They are not visible in the day time, which is explained further in the animation section5.2. The stars are small 'glutSolidSphere()'s and each is drawn by rotating about the x and y axis, followed by a translating in the new up direction by the radius which specifies how far away they should be.

Where each star is therefore depends on the initial x and y axis rotations which are randomised for all stars except a select few. These select few I specifically trial-and-errored into place of Ursa Major, the star constellation.

Figure 7: Ursa Major Star Constellation with Moon for context

The brightness of the stars is simply modelled with the radius of the 'glutSolidSphere()'. Brighter stars are set to be bigger than further away or dimmer stars. This is also taken into account when modelling Ursa Major, attempting the replicate the brightness of each star of the constellation correctly.

3 Lighting

3.1 Sun and Moon

The sun and moon both emit ambient, diffuse, and specular light. The sun emits purely white light so that nothing becomes tinted, however the moon emits dimmer, more gray, light. The ambience of both of these is much lower than the diffuse so that the changes between shades of the diffuse lighting are seen. Specular lighting is most intense for both of these light sources so that light reflects aggressively at the right angle of view.

Because we are not doing any work with shadows, when the moon and sun go under the map, their light still emits on faces that are not completely facing away from these light sources. Therefore, there if a check in the animation5.2 to see if the sun and moon go below the terrain at which point they are hidden to prevent this issue.

3.2 Normals

This section aims to inform an overview into the approach in using normals for models where they are non-trivial.

3.2.1 Terrain Normals

To calculate the normals of each triangle in the terrain mesh every frame would be heavy on performance. Therefore these are calculated and stored when the terrain is initialised, ready to be used in the Display function of the terrain. These are caluclated using the following:

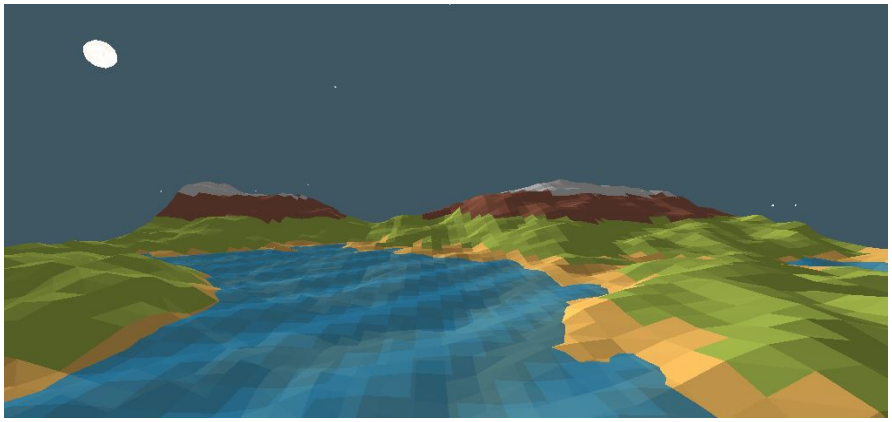

Figure 8: Example of Normals in Action

```
Vector3f* v = p2->subtractVector(p1);
Vector3f* w = p3->subtractVector(p1);

Vector3f* normal = new Vector3f(
  (v->y * w->z) - (v->z * w->y),
  (v->z * w->x) - (v->x * w->z),
  (v->x * w->y) - (v->y * w->x)
);

normal->normalise();
```

where p1, p2, and p3, are the three points of a triangle.

3.2.2 Water Normals

The normals of the water are calculated in the same way as above for the terrain, however this has to be done every time the water updates wave heights which lowers performance, however creates shaded waves which helps create a realistic looking wave.

3.2.3 House Normals

Since the roofs are generated at a random gradient, the normal also has to be calculated. However, this is easier because it is always only in two dimensions because the house is only orientated in 4 different ways. The normals are calculated by doing the inverse vector of the hypotenuse of the roof:

```
//Draw left side of roof
glNormal3f(-roofHeight, sideLength / 2, 0.f);
...
//Draw right side of roof
glNormal3f(roofHeight, sideLength / 2, 0.f);
```

For the left side of the roof, drawing a vector that goes the *height* of the roof in the left x direction, and goes the x length of half the roof (one side of the roof) *upwards*, therefore flipping the x and y, creates the normal. This is not a unit vector therefore this is ran with GL_NORMALIZE enabled.

3.3 Material Properties

Water has set material properties to react differently to the different types of light using the Phong lighting model. In this example, water is set to react with dark blue to ambient light of which there is only some from the sun and moon, it reacts with a lighter sea blue to diffuse lighting, and with a bright but faded out white that will blend with the blue to specular lighting. I have also set the

water to have a shininess level of 60 causing it to be more considered more shiny than default material properties.

4 Texture

4.1 Sun and Moon

The sun and moon are both textures using a real sun and moon texture image to start with. The resolution of these image were lowered and finally they were both pixelated to fit the design of the scene. The method to assign the texture coordinates is the same as of the demo on texturing a sphere, where each texture coordinate is calculated from the angle of each point mapped onto texture coordinate by restricting the angle to -0.5 and 0.5 by dividing the angle by 2PI, and adding 0.5, which maps the angle to a value between 0 and 1. This is done for each point of each triangle face. The same seam fix method is also used to fix the issue of where the texture wraps around. This simply maps values too far right of the texture map to the left of the texture map, so that they wrap around where a triangle is between both the right and left side of the texture map.

Figure 9: Sun and Moon Textured

4.2 House Logs

Each log is textured so that it resembles a shaved trunk of a tree.

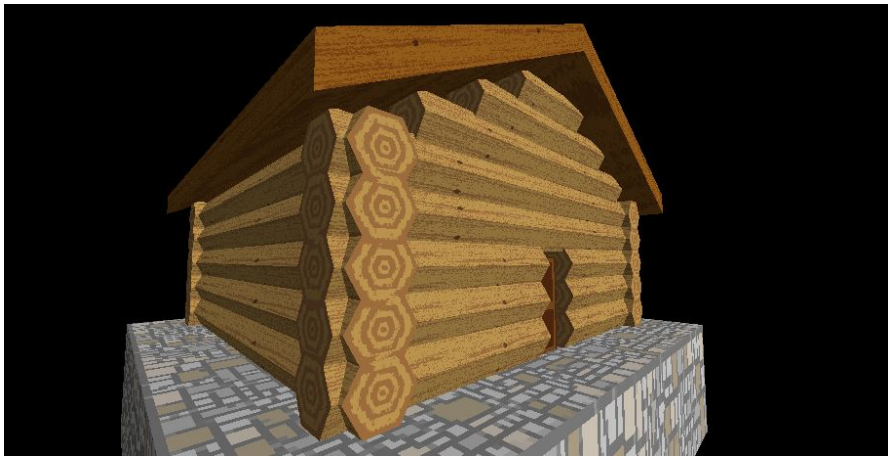

Figure 10: Wooden Log Texture

4.2.1 Cross Section

Each end of the log is a hexagon, textured as a cut cross-section of a tree. I created this texture, along with most others, on Paint. The texture coordinates for this hexagon are pre-calculated when the house is first initialised so that each coordinate does not have to be repetitively recalculated which slows performance. The texture coordinates are calculated by mapping the angle to value between 0 and 1:

```
t = cosA / 2 + 0.5;
s = sinA / 2 + 0.5;
```

Dividing by two restricts the value to -0.5 and 0.5, and adding 0.5 maps it to a value between 0 and 1 which is then used as the texture coordinate. These values are calculated once and stored in an array, used by the wooden logs whenever they are drawn.

4.2.2 Sides

The sides of each log are textured using another Paint-created image. As opposed to the cross section, the side texture coordinates are not pre-calculated because they are trivial. Each one of six wooden log sides are a rectangle, so each vertex of the rectangle can be set to the respective corner of the texture.

4.3 House Roof Textures

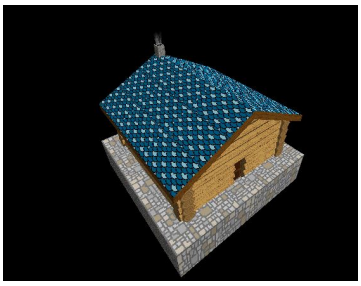

Figure 11: Roof Tiles Texture

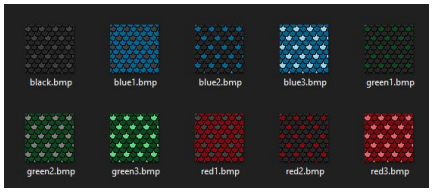

Figure 12: All Tile Textures

4.3.1 Tiles

I have created many variations of the tiles texture in Paint. 10 different roof tile textures. These are randomly picked when a house is first created and the texture coordinates are scaled such that the tile texture repeats itself along the length and height of the roof side but does not change ratio.

4.3.2 Bevel

The bevels are textured using the same texture as the logs, except that the glColor is set to a brown colour which blends with the texture, giving it a darker shaved wood look.

The front and back bevels of the roof are parallelograms. The left and right bevels are rectangles. This means that the texture coordinates are trivial and set to the respective corners, as skewing the shaved wood texture is not an issue.

4.4 House Foundation and Chimneys

4.4.1 Foundation

I wanted the house foundation to appear as stones and rocks that have been reinforced. For this I created another texture on Paint. In this case, just setting the texture coordinates to the corners gave rocks and stones that were way too big. Therefore, knowing that the textures are set to repeat, I set the texture coordinates out of the bounds of 0 and 1. By setting the right and top coordinates to a bigger value, say n, the texture will repeat so to fit n repetitions of the texture. In other words, the texture gets scaled by 1/n. Using this concept I scaled the top face evenly so that the texture is 2 times smaller, fitting 4 full textures onto the face.

Since the sides of the foundation will be of varied height to width ratios, this has to be taken into account when wanting to keep the stones texture proportionate. To achieve this, the width doesn't

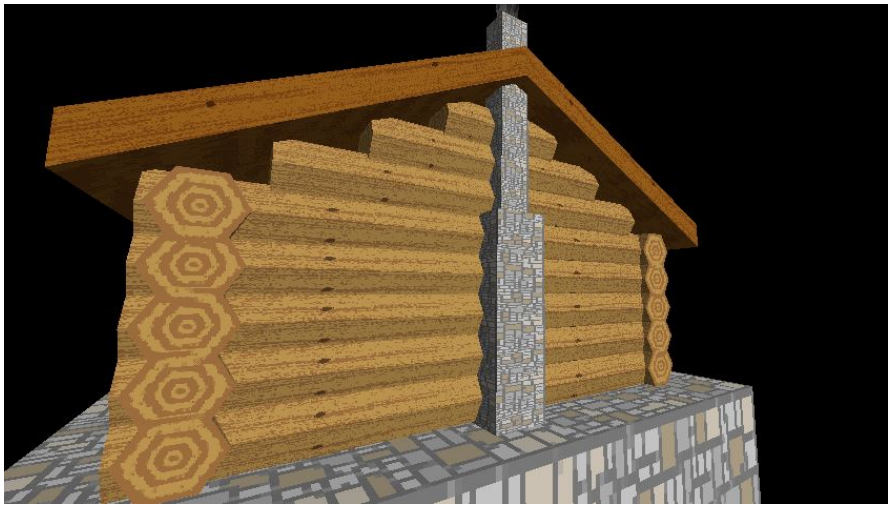

Figure 13: Foundation and Chimney Stone Textures

change so it is always set to the overall texture scale which is set to 2. However, for the height, the texture coordinates are decided based on the ratio of height to width:

```
heightTexScale = textureScale * (height / sideLength)
```

where the *textureScale* is how many times to repeat the texture, *height* is the height of the foundation and *sideLength* is the width of the foundation.

4.4.2 Chimney

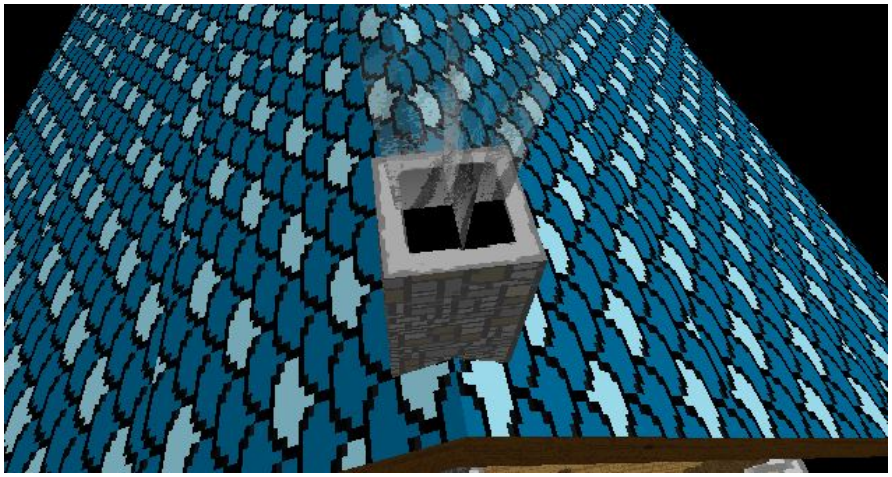

Figure 14: Top of Chimney Texture

The chimney texture is also made up of the stones and rocks, just as the foundation, with the exception of the top chimney face which is a black hole with the rock texture bordering the outside. For this the texture coordinates were set to each respective corner.

I intended to make the chimney look like it was made of smaller rocks and stones laid on top of each other. In this case the faces are much smaller, so the texture coordinates are scaled down by dividing the width by the desired size over 2, and dividing the height by the desired cobble size. This gives a texture proportion that I am happy, displaying the rocks as more flat horizontally.

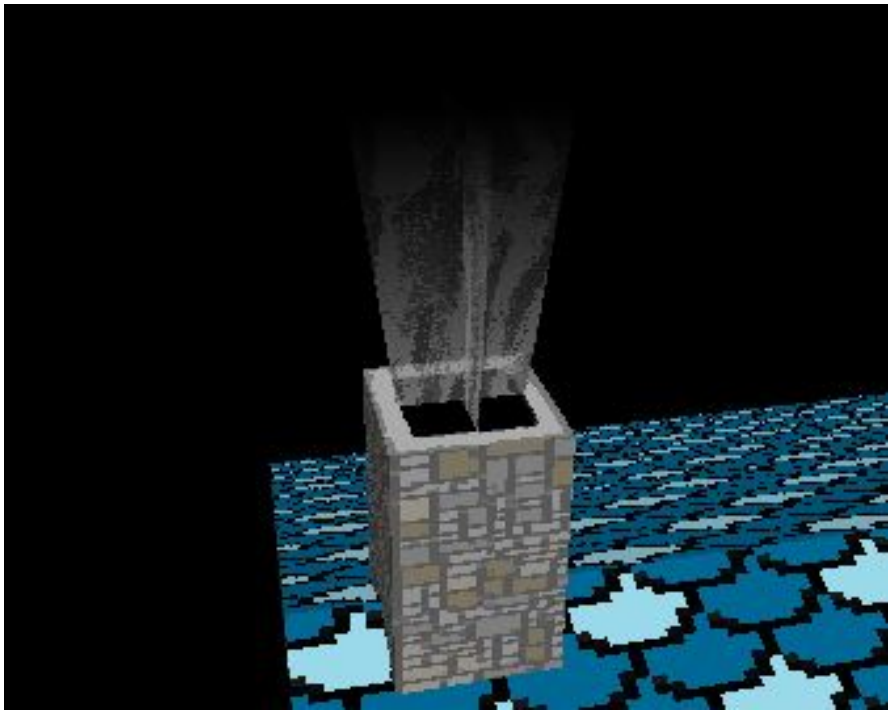

Figure 15: Smoke Animated Texture

4.5 Smoke Animated Texture

Smoke is implemented using 2 double sided 2D planes, one on the x-axis and another on the y-axis. This totals 4 faces, and creates the effect that it has 3D volume. The top vertices of these planes have a 0 alpha, and the bottom vertices have an alpha of 1. This creates a blend, effectively fading out the smoke as it is further away from the chimney.

The smoke texture was made using Paint and is made so that the bottom and top of the textures repeat - when the texture wraps around top to bottom it is continuous.

4.5.1 Animation

As the update function is called, a run-time counter is incremented with the running time. This running time is used to decrement the offset of the texture coordinates vertically. Moving the coordinates down creates the effect of smoke going up.

Adding to the upwards motion of the smoke, I also wanted to implement a wave into the horizontal movement of the smoke so that it looks more natural. To do this I used the run-time as a parameter into sin and cos to get a value between -1 and 1. The vertices on the right of each plane are incremented by 1 to the calculated value so that the width of the texture is used. The key is that the top vertices use the opposite function to the bottom vertices using the same run time parameter. As time goes on, the smoke looks like it is rising, waving naturally left to right.

5 Animation

5.1 Animated Water Waves

Each point in the triangle mesh for the water is set to a height depending on its row and column in the mesh, along with the wave phase it is in. This phase is incremented by the change in time, in proportion with the wave speed, in the update function and the height of each point recalculated using:

```
float Water::CurrentWaveHeight(int row, int col)
{
```

```
    //Points where the col + row add to the same number will have the same
   //height, forming diagonals.
    return sin(double(col) + double(row) + double(wavePhase)) * waveHeight;
}
```

The speed of the wave is controlled by the user6.1.2. This formula gives each point where the column and row add up to the same number, the same height wave height. This gives the effect of the wave moving in a diagonal direction.

5.2 Sun, Moon, and Stars in Day/Night Cycle

The sun, moon, and stars, are all animated to rotate around the terrain to imitate a day/night cycle. The way that the sun and moon are rotated around is different to the way that stars are rotated around. This is because the sun and moon are light sources and their x and y coordinates must be specifically calculated.

5.2.1 Background - Sky Colour

The background colour of the scene changes as day and night comes. To do this, I have defined the sky colour in the header file, separating the red, green, and blue channels. In the Update function, the actual colour is set as the same sine calculation that dictates how high the sun is in the sky so that they match up.

```
//Update colour of the sky
skyColour->red = MAX_RED * sunSin;
skyColour->green = MAX_GREEN * sunSin;
skyColour->blue = MAX_BLUE * sunSin;
glClearColor(skyColour->red, skyColour->green, skyColour->blue, 1.f);
```

5.2.2 Sun and Moon

Both the sun and moon are drawn into position only by translating. This is because a GL_LIGHT source x and y (and z) position has to be specified. The sun and moon therefore rotate by using the run time as a parameter into the cosine and sine functions multiplied by the radius for the x and y coordinate respectively.

In addition, there is a check for the sun and moon to test if it should be drawn. If the sun and moon are underneath the terrain, they won't be drawn. This also solves a lighting issue3.1.

```
//Check if sun should be visible
if (sunAngle < -20 && sunAngle > -160) {
  _showSun = false;
}
else {
  _showSun = true;
}

//Check if moon should be visible
if (moonAngle > 20 && moonAngle < 160) {
  _showMoon = false;
}
else {
  _showMoon = true;
}
```

5.2.3 Stars

The stars are rotated around the terrain by the z-axis which is changed is recalculated the update function. The stars actually reuse the angle that the moon is facing as it saves some processing power.

When the sun comes out and the colour of the sky stats to change, the stars slowly fade away by increasing the alpha that they are drawn with.

```
glColor4f(1.f, 1.f, 1.f, sin(_runtime +1.5) + 1);
```

The '+1' at the end is to offset the result of sin from values between -1 and 1 to values between 0 and 2, meaning that the stars will be visible for more than half of the day night cycle which is preferred as a styling choice.

5.3 Cloud Rotation and Morphing

Each cloud rotates around the centre of the terrain. This is done by preparing each cloud draw with a rotate around an angle. This angle is incremented in the Update function.

```
void Cloud::Update(const double& deltaTime)
{
    if (_flagAnimation)
    {
        //Incrememnt cloud angle
        rotateAngle += deltaTime / speed;

    ...
```

The speed is a variable controlled by the user6.1.3. This angle is used in two ways. Where the cloud is drawn in the terrain, but also for the counter-rotation on its own y-axis so that their orientation stays the same. This makes the clouds look slightly more dynamic and less like they are just spinning around the centre:

```
void Cloud::Display()
{
    ...
    //Rotate around center of terrain
    glRotatef(rotateAngle, 0.f, 1.f, 0.f);
    glTranslatef(position->x, position->y, position->z);

    for each (CloudBubble* bubble in bubbles)
    {
        glPushMatrix();
        //Rotate cloud back to prevent spinning around its own axis
        glRotatef(-rotateAngle, 0.f, 1.f, 0.f);
        DrawBubble(bubble);
        ...
```

If a user looks closely, they will notice that the clouds are also slowly morphing. Each bubble is set to bounce between its maximum and minimum radius, each bubble in its own phase. There is a per-bubble delta that gets added to the radius. This delta is negated when it reaches a border case, a max or min radius.

```
//For each cloud bubble, increment or decrement radius
for each (CloudBubble * bubble in bubbles)
{
    //If the bubble has gone past the max radius
    if (bubble->radius > maxBubbleRadius)
    {
        bubble->delta = abs(bubble->delta) * -1;
    }
    //If a bubble goes under the min radius
    if (bubble->radius < minBubbleRadius)
    {
        bubble->delta = abs(bubble->delta);
```

```
    }

    //Change bubble radius
    bubble->radius += (float)bubble->delta / (200 * speed);
}
```

Where 200 is an arbitrary value that works well in this configuration and speed is changed by the users interaction6.1.3.

5.4 Door

Figure 16: Door Opening/Closing Animation

When the user triggers the door animation6.2, the door smoothly rotates on its hinges to toggle being open and closed. The door is drawn from its hinges, therefore a glRotate on the y-axis is prepared before it is drawn. This rotate the door a specific angle before drawing it. This angle is changed by the Update function. There is logic in place to check if the door is supposed to be open or closed, and whether it is open or closed. Therefore, in the Update function, if the door is supposed to be open but the angle is less than what is set to be fully open, then the angle is incremented, rotating the door. Likewise, if the door is set to be closed but the angle is more than what the closed door angle is set to be, then the angle is decremented.

```
void House::Update(const double& deltaTime)
{
    if (!doorOpen && doorAngle > 0)
  {
        doorAngle -= 2;
  }
  else if (doorOpen && doorAngle < 90)
  {
        doorAngle += 2;
  }
  ...
```

The angle is incremented and decremented in values of two as this is the preferred door swing speed.

6 Interaction

6.1 Change Time Speed

In this environment, the user has the ability to change the speed of time. This changes a few factors; the day night cycle and therefore sun, moon, and star rotation speeds, the speed of the water waves, and the rotation and morphing speed of the clouds. This gives the user the feeling that time can be slowed and sped up.

The user can speed up and slow down time using keys 1 through 9, 1 being the fastest and 9 being the slowest. The 0 key can be used to stop time. However, each component of the scene reacts differently with different parameters to these keys each component that wants to react to time changes has the responsibility to react to these keys independently.

6.1.1 Day Night Cycle Speed

Time affects the day/night cycle by increasing the speed at which the sun, moon, and stars, rotate around the terrain5.2. This changes the day/night speed variable, as well as the animation flag which dictates whether the Update function will do anything. The day night speed variable is what the change in time is divided by before being added to the total run time, which is the variable that ties all animations in the day night cycle together.

6.1.2 Water Waves Speed

Water wave speed is dependent on the wave phase. The wave phase is updated using the change of time divided by the wave speed. This wave speed is what changes when the user changes the time speed. This change therefore changes how quickly the wave phase is progressed, slowing or speeding up the waves.

6.1.3 Cloud Morph and Rotation Speed

Changing the time speed changes the speed variable for the clouds. This speed variable is what changes the cloud angle at different rates. This is what affects how quickly they spin around the terrain and on their own y-axis to counter act their orientation rotation. The speed also affects the factor by which the cloud bubbles each update their radius. So the rate at which they morph is also affected by the user.

6.2 Locally Interacting with Houses

Given that the user is close enough to a house, they are able to open and close the door using the 'F' key. This toggles the boolean of the door that represents whether it should be open or not.

```
void House::HandleKey(unsigned char key, int state, int mx, int my)
{
  //Get the camera x, y, z
  float x, y, z;
  Scene::GetCamera()->GetEyePosition(x, y, z);

    //If camera is in range of the house
    if (InRange(x, y, z))
    {
        // Ignore key-release
        if (!state) return;
        // Handle key
        switch (key)
        {
        case 'F':
        case 'f':
            //Toggle door being open
            doorOpen = !doorOpen;
            break;
```

```
        }
    }
}
```

Toggling this boolean triggers the animation to behave in a different way5.4 to close or open the door.

To dictate whether a user is close enough, first the camera eye position is received from the scene, as shown above, and these coordinates are fed into the InRange(x, y, z) function.

```
bool House::InRange(float x, float y, float z)
{
    //Calculate the difference in x, y, and z coordinates
    float deltaX, deltaY, deltaZ;
    deltaX = abs(x - position->x);
    deltaY = abs(y - position->y);
    deltaZ = abs(z - position->z);

    //Magnitude of 3d vector, get distance
    float distanceFromCamera = sqrt(deltaX * deltaX + deltaY * deltaY + deltaZ * deltaZ);

    //Return whether the position is within the distance of two house lengths.
    return distanceFromCamera < sideLength*2;
}
```

6.3 Camera Movement

It was important for me to be able to move vertically in my scene. It is nice to be able to get closer to inspect each house for example, and necessary to trigger the local door opening and closing animations. For this, I adjusted the framework to include two new buttons, 'Q' for descent and 'E' for ascent. This builds on top of the framework-provided 'WASD' camera movement.

```
//If Q or q is pressed, subtract the up vector at the set speed
if (qKey)
    sub(eyePosition, up, speed);

//If E or e is pressed, add the up vector at the set speed
if (eKey)
    add(eyePosition, up, speed);
```

These keys are also added to the HandleKey function to change their state when their respective keys, upper or lower case, are pressed.

7 Conclusion

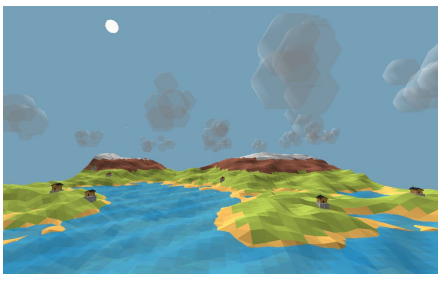

Figure 17: Example Scene in the Day Time

Figure 18: Example Scene in the Night Time

In the end, I am happy with the final result. Throughout the project I learned a lot about techniques

on how to calculate and draw using geometry. The process of thinking of an idea and using the ideas we have been taught along with maths and code to get our ideas on the screen was very satisfying and have made me appreciate today's graphics a lot more. I would say I am most proud of my terrain working together with the water and houses to create a dynamically adjusting environment based on input parameters.

7.1 Improvements

Optimisation is an aspect I would love to go back and further improve, for example, right now my water normals are being calculated per frame because the water changes. However, the water changes in a predictable way therefore these normals could be pre-calculated also and stored to improve the frames per second. Also, I would have like to change the resolution at which things are displayed based on how far away from the camera eye position they are. An example of this would be not drawing the walls of houses at a distance with logs but with a straight quad and only switching to more detail when the eye is close enough. Finally I would have liked to added more tesselation to the water so it reflected light in a more realistic and satisfying way. However, in all of this, I am still very proud of my final scene.