Natalie Hong (45309740)

COSC300 Graphics Report

Semester 1 - 2020

Contents

List of Images

1 Introduction

In this document, you will find my findings and attempts at creating a completed game. The game I had chosen to make was that of the supplied mega racer code.

1.1 Problem Statement

The problem I had was that of not knowing enough computer graphics knowledge to create a full game. Through the development of the mega racer game, I aimed to improve my understanding of computer graphics as well as create a fun, nice-looking and interactive game for users to play.

1.2 Methods, Techniques and Execution

To create the graphics required for this project, I used Python 3.5 and OpenGL to create objects and renderables within the computer graphics scene. I also used GitHub to allow for version control between each implemented feature that I added.

The features that were made were:

- 2 - Scaling the Terrain
- 3 - Setting up the Camera
- 4 - Orientating and Placing the Racer Model
- 5 - Texturing the Terrain
- 6 - Lighting from the Sun
- 7 - Improving Textures
- 8 - Adding Fog
- 9 - Adding Props
- 10 - Racer Headlights

In the following documentation, you will find screenshots and findings which include results and discussion around each feature as well as how I went about approaching each feature within the project.

2 Scaling the Terrain

Scaling terrain consisted of scaling the terrain according to the path image supplied.

```
# copy pixel 4 channels
    imagePixel = self.imageData[offset:offset+4];
    # Normalize the red channel from [0,255] to [0.0, 1.0]
    red = float(imagePixel[0]) / 255.0;

    xyPos = vec2(i, j) * self.xyScale + xyOffset;
# TODO 1.1: set the height
zPos = 0
```

Initially, the code supplied consisted of zPos = 0 and I was to change it. I thought long and hard about to change and looked at the hints supplied. In it, I was told to use self.heightScale and to somehow utilize it in order to get a consistent z position for each terrain created. I had a thought that multiplying the red value calculated would allow me to get a a consistent z

position. This was because red represented a value from 0 to 255 and multiplying it with the height scale would allow for different Z positions to be generated.

```
# TODO 1.1: set the height
zPos = self.heightScale * red
```

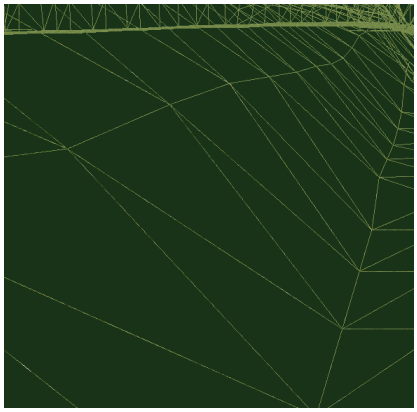


Figure 1: Scaled Terrain

This seemed to work as I was now able to see bumps and curves within the terrain which meant this feature was completed.

3 Setting up the Camera

The next feature I was to implement was that of setting up the camera so that it viewed the racer at all times and was scaled to the supplied camera offset. As well as this, I wanted the camera to be able to rotate according to the heading of the racer. After much experimentation I was able to come up with the following code:

```
x = g_racer.position[0]
y = g_racer.position[1]
z = g_racer.position[2]

....
g_viewTarget = [x, y, z]
```

Since I wanted my camera to look at the racer, I decided to make it so that the camera was always looking at where the racer was positioned. However, I did not want the camera to be at the exact same position as the racer as that would make it feel weird. To make sure the camera was positioned away from the racer, I used the assigned follow cam offsets. This code was very similar to Lab 3 of the course. The result can be seen in the above figure, please note that this screenshot was taken later into development of the game.

```
for i, positionPoint in enumerate(g_racer.position):
    g_viewPosition[i] = positionPoint + g_followCamOffset * -g_racer.heading[i]

g_viewPosition[2] += g_followCamLookOffset
```

In the code above, I was able to create a viewing position for the camera to be positioned at. Position points were each of the racer's coordinates and were then added to the follow cam offset multiplied by the negative of the racer's heading coordinates. Since -1 for the heading

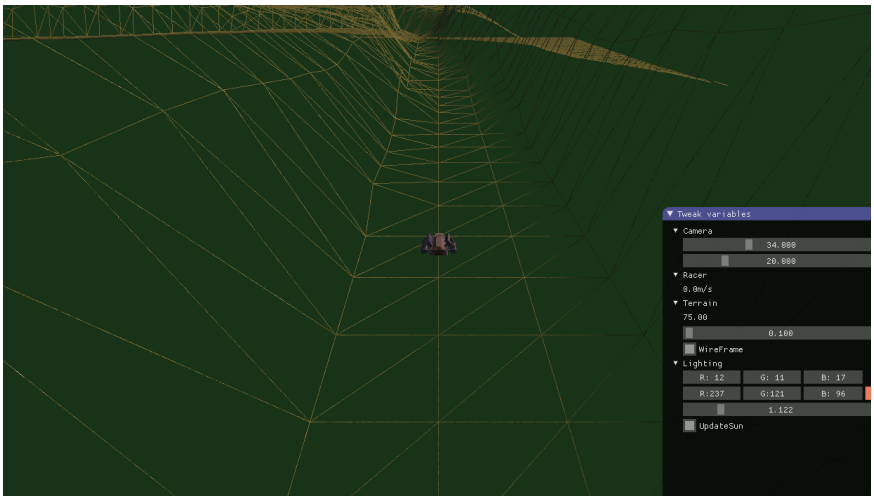


Figure 2: Looking at the Racer

coordinates corresponded to the right rather than the left, I made sure to make the values reverse using the negative version of the heading value.

Since z-axis had an added follow camera offset as it was specified in the task sheet, I made sure that the z axis needed to have the additional multiplier of the follow cam offset to it.

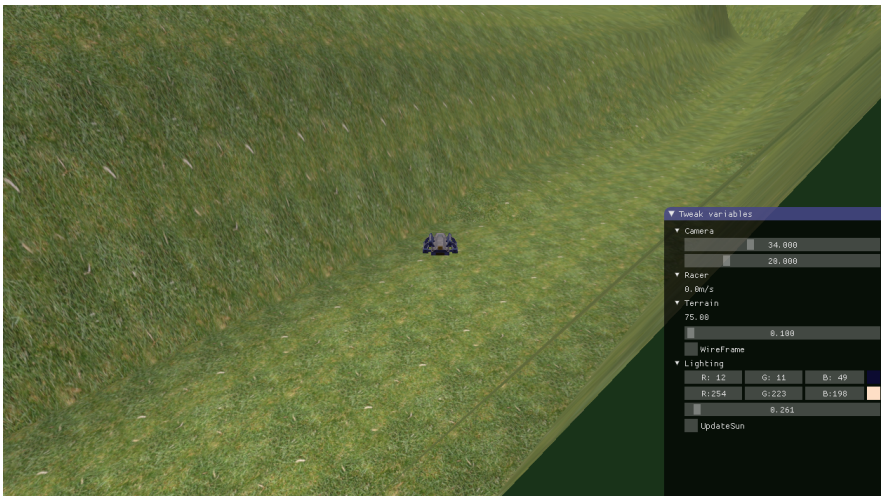


Figure 3: Camera Rotating with the Racer

The result of rotating the camera can be seen in the above figure, please note that this screenshot was taken later into development of the game.

4 Orientating and Placing the Racer Model

This task for the racer required me to rotate and orientate the model. This model loading was very similar to the of Lab 1. To place the model at the racer's location I did the following:

```
top = [0,0,1] # Define top in the Z-axis
modelInWorld = lu.make_mat4_from_zAxis(self.position, self.heading, top)
# Draw model in the specified world coordinates
renderingSystem.drawObjModel(self.model, modelInWorld, view)
```

First, I defined the top of the world in that of the z-axis. This was necessary as the task sheet recommended to do it and allowed me to use more functions. Using this top value, I made a 4x4 matrix using the racer's position, where it was heading and the top value defined earlier. Once I got this 4x4 matrix which represented coordinates to draw the model in, I then added the model to the coordinates and used the passed in view to place the Racer in the scene.

```
self.model = ObjModel(objModelName)
```

The last thing I did was load the model inside the racer's load function so the racer model could initialise on load.

Figure 4: The Loaded Model

Once loaded, I was able to rotate the model with the camera to allow for the racer model to always be facing forwards no matter which way the camera was adjusted to.

5 Texturing the Terrain

As you may have seen in previous figures, I was able to texture the terrain of the game. This was done by binding textures to a variable to create unique texturing for the grass and to allow for more

```
# Texture ids
 grassId = None

 # Texture unit allocations:
 TU_Grass = 0
```

First, I defined a grassId to use and bind a texture to. I also loaded the texture from the path (grass2.png) after I defined the texture. This was so that I could get the actual image to show.

```
    #TODO 1.4: Bind the grass texture to the right texture unit, hint:
        lu.bindTexture
    lu.bindTexture(self.TU_Grass, self.grassId)
    lu.setUniform(self.shader, "grassId", self.TU_Grass)
    ...
    self.grassId = ObjModel.loadTexture("data/grass2.png", basePath, True)
```

I then used similar code to that of the fifth and fourth lab (the ones about textures and lighting), to create a texture definition for the grass.

To do this, I translated the texture using the binded grass id to a texture and passed in the world space position and multiplied it with the xy texture scale. I did this to make sure that the terrain would be scaled to my game's world. I then was able to get a grass colour using this texture function which I then passed into compute the shading that the grass would be according to the light given from the sun.

I then was able to finally convert the reflected light into a fragment color variable which would serve as the level of light that the grassId texture was to give off.

```
void main()
{
    vec2 testvecw = v2f_worldSpacePosition.xy;

    vec3 materialColour = vec3(v2f_height/terrainHeightScale);

    // TODO 1.4: Compute the texture coordinates and sample the texture for
        the grass and use as material colour.
    vec3 grassColour = texture(grassId, testvecw * terrainTextureXyScale).xyz;
    vec3 reflectedLight = computeShading(grassColour, v2f_viewSpacePosition,
        v2f_viewSpaceNormal, viewSpaceLightPosition, sunLightColour);
fragmentColor = vec4(toSrgb(reflectedLight), 1.0);

}
```

The end result looked like the following:

Figure 5: The Textured Terrain

6 Lighting from the Sun

Using the shaders supplied from lab 4, I was able to create similar shaders for the sun. For part where I was supposed to add code, I noticed that the passed in variables were:

- vec3 materialColour
- vec3 viewSpacePosition
- vec3 viewSpaceNormal
- vec3 viewSpaceLightPos
- vec3 lightColour
- global variable of globalAmbientLight

These variables were very similar to the shaders in lab 4 so I used them as reference. Since there was no material alpha for the object like the previous labs, I decided to not add any extra variables to create this. The explanation for each step I took can be seen in the code block below.

```
vec3 computeShading(vec3 materialColour, vec3 viewSpacePosition, vec3
    viewSpaceNormal, vec3 viewSpaceLightPos, vec3 lightColour)
    {
        # TODO 1.5: Here's where code to compute shading would be placed most
            conveniently
        # Compute incoming light, stop complete blackness when the sun is hidden
        vec3 viewSpaceDirToLight =  normalize(viewSpaceLightPos - viewSpacePosition);
        float incomingIntensity = max(0.0, dot(viewSpaceNormal, viewSpaceDirToLight));
        # Get the yellow-ish tinge from the sun's incoming light
        vec3 incomingLight = incomingIntensity * lightColour;
        # Add directional light
        vec3 outgoingLight = (incomingLight + globalAmbientLight) * materialColour;
        return outgoingLight;
    }
```

This meant that the sun was able to update according to the sunAngle which was later used to calculate the values viewSpaceLightPosition, sunLightColour and globalAmbientLight which were passed into the shader code during the sun's update function. My game was able to look like the following:

Figure 6: Afternoon Sun Look

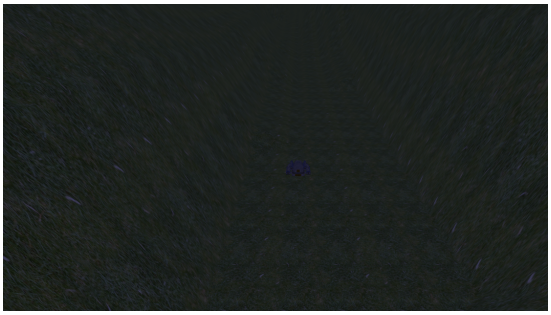


Figure 7: No Sun

7 Improving Terrain

Improving terrain according to the task sheet was that of adding different terrain textures to the game which coicided with that of higher rocks, steeper rocks and a road.

To start off, I defined some texture ids and texture variables to bind the new textures to alongside the existing grass texture one. I also loaded the texture from the path similar to that of the was I loaded the grass path.

```
# Texture ids
grassId = None
highRockId = None
steepRockId = None
roadId = None

# Texture unit allocaitons:
TU_Grass = 0
TU_High_Rock = 1
TU_Steep_Rock = 2
TU_Road = 3

...
#TODO 2.1: Improve Terrain Textures
self.highRockId = ObjModel.loadTexture("data/rock 2.png", basePath, True)
self.steepRockId = ObjModel.loadTexture("data/rock 5.png", basePath, True)
self.roadId = ObjModel.loadTexture("data/paving 5.png", basePath, True)
```

I went by the task sheet's hint of using the height variable in the shader to define the material colour at when the height was above 60 to be that of the high rock texture.

```
if (v2f_height > 60) {
    materialColour = texture(highRockId, testvecw * terrainTextureXyScale).xyz;
}
```

Calculating the steep rocks proved to be a bit of a challenge as I was not too sure on where to start. I searched up on the internet on how to find the angle between two vectors (that of the z-axis/flat ground and the worldSpaceNormal as specified in the spec). I was fortunate to come across a Stack Overflow question which was very related to what I wanted to do. I then found the normal of the z-axis and that of the world space and applied the dot product to the vectors. I then found the acos of the dot product to find the angle. The code looked like the following and was then used to determine steep slopes within the terrain:

```
    vec3 worldSpace_Normal = normalize(v2f_worldSpaceNormal); // Normal of world space
    vec3 zaxis_Normal = normalize(vec3(0, 0, 1)); // Normal of Z-axis
    float angle = acos(dot(worldSpace_Normal, zaxis_Normal));
    ...
    if (angle > 0.45) { // Angled
        materialColour = texture(steepRockId, testvecw * terrainTextureXyScale).xyz;
    }
```

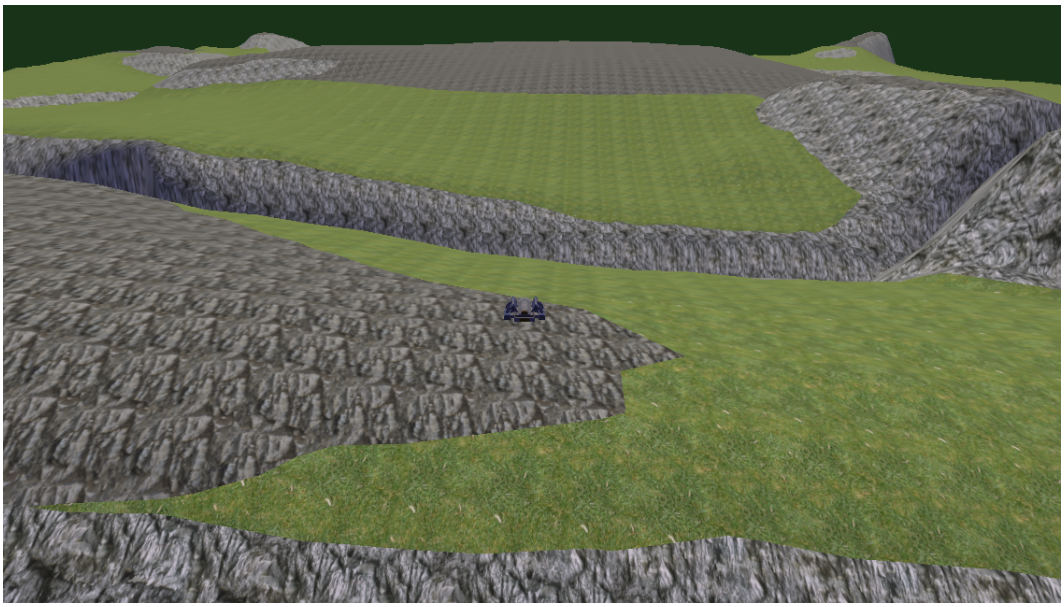


Figure 8: The Textured Terrain

With the high stones and the steep stones done, I was then to do the road. From what the task sheet said, I was to use the rgb pixels to determine where the road was to be placed. This turned out to be a little harder to do and I was unsure on how to correctly implement this, so I decided to approach it in my own way.

Similar to the way I did the steep stones, I checked with the height threshold but instead I used height less than rather than greater than.

```
if (v2f_height < 30) {
    materialColour = texture(roadId, testvecw * terrainTextureXyScale).xyz;
}
```

Figure 9: The Textured Terrain

This allowed me to get a semi-functional road, although it did vary in length at some points which was due to the height factor not being enough to calculate it. I felt like this approach was suitable though was the road was still visible within the track and looked nice.

It was suggested by a tutor that I should add noise to my terrain to add more variety and look to it. To do this, I used the code from lab 5 to implement my noise. The code looked like the block below with an added line after I calculated the material colour after all the if statements seen in the previous code blocks.

```
float random_gen(vec2 n) {
    return fract(sin(dot(n, vec2(12.9898, 4.1414))) * 43758.5453);
}

float noise_gen(vec2 p) {
    vec2 ip = floor(p);
    vec2 u = fract(p);
    u = u*u*(3.0-2.0*u);
    float res =
        mix(mix(random_gen(ip),random_gen(ip+vec2(1.0,0.0)),u.x),mix(random_gen(ip+vec2(0.0,1.0))
    return res*res;
}
...
materialColour += vec3(noise_gen(testvecw * terrainTextureXyScale));
```

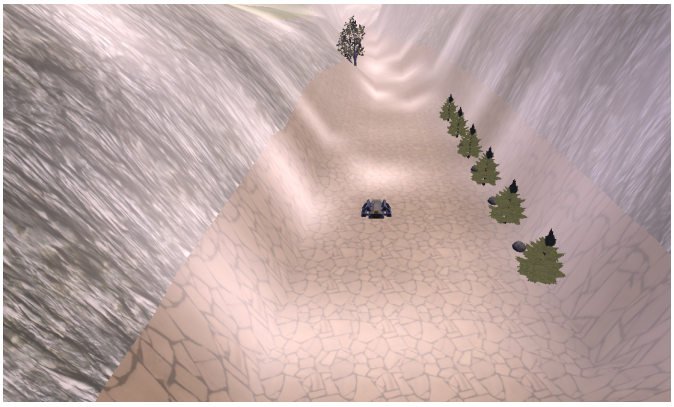


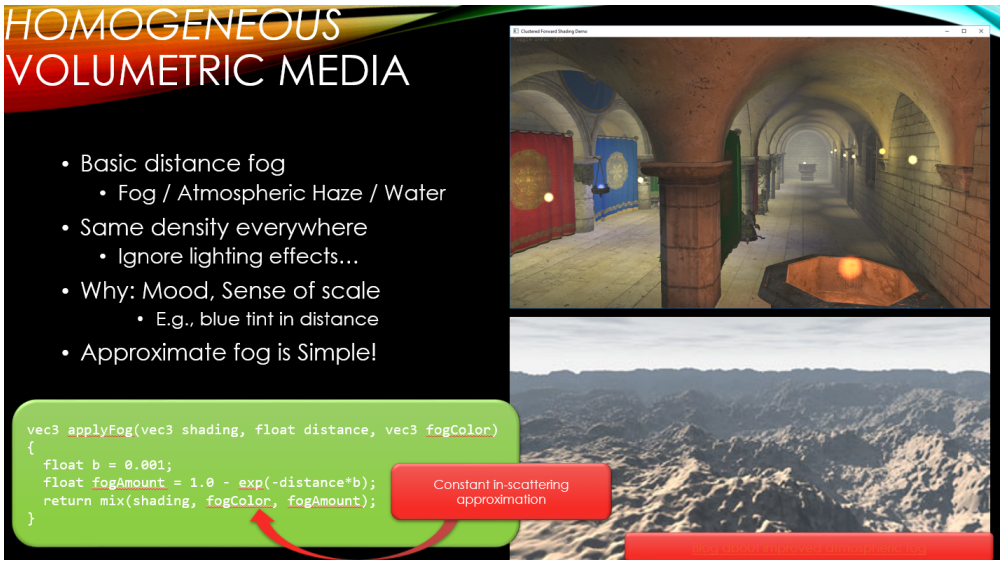
Figure 10: The Unscaled Noise

After implementing this, I realised that the noise was much too high, so I reduced it by multiplying the resulting float received from my noise gen function by 0.1. After doing this, I was able to get the following result.

Figure 11: The Scaled Noise

8 Adding Fog

When I was watching lectures for this course, I noticed that one of the lecture slides had a link to a helpful website that I could use.



When I went to the website, I was able to find some interesting information on how to implement fog.

```
vec3 applyFog( in vec3  rgb,      // original color of the pixel
               in float distance, // camera to point distance
               in vec3  rayDir,   // camera to point vector
               in vec3  sunDir )  // sun light direction
{
    float fogAmount = 1.0 - exp( -distance*b );
    float sunAmount = max( dot( rayDir, sunDir ), 0.0 );
    vec3  fogColor  = mix( vec3(0.5,0.6,0.7), // bluish
                           vec3(1.0,0.9,0.7), // yellowish
                           pow(sunAmount,8.0) );
    return mix( rgb, fogColor, fogAmount );
}
```

Figure 12: First Look

```
vec3 applyFog( in vec3  rgb,      // original color of the pixel
               in float distance ) // camera to point distance
{
    float fogAmount = 1.0 - exp( -distance*b );
    vec3  fogColor  = vec3(0.5,0.6,0.7);
    return mix( rgb, fogColor, fogAmount );
}
```

Figure 13: Final Decision

As seen in Figure 12, the code demonstrated was that of an apply fog function that was to be put in the megaracer file. I initially tried to implement this function but was unsure about the rayDir

and sunDir variables that were passed into it. I originally tried passing in viewSpacePosition and viewSpaceLightPosition which did not seem to do what I wanted as the sun did not end up shaded properly for some reason.

So instead I with the approach seen in Figure 13. Once I had impmented this, I realised that the fog was kept to one colour as the fogColor variable was kept constant and turned out to just be grey which did not give too much volume to it.

I tried adding sunLightColour to be the new fog colour but that seemed to make the fog colour only be independently influenced by the sun, which I did not want. So I decided to combine it with the global ambient light colour as per the specification task.

```
vec3 applyFog(in vec3 rgb, in float distance) {
    float b = 0.0045;
    float fogAmt = 1.0 - exp(-distance*b);
    vec3  fogColor = (sunLightColour + globalAmbientLight);
    return mix(rgb, fogColor, fogAmt);
}
```

My apply fog function now looked like this. Using the code that was from the lecture slides, I was able to add fog to my game's code which looked like the following:

```
// Applying fog
// reflectedLight - rgb pixel colour
// v2f_viewSpacePosition.z - Distance
fragmentColor = vec4(toSrgb(applyFog(reflectedLight,
       v2f_viewSpacePosition.z)), 1.0);
```

This however, resulted in my game looking like the following:

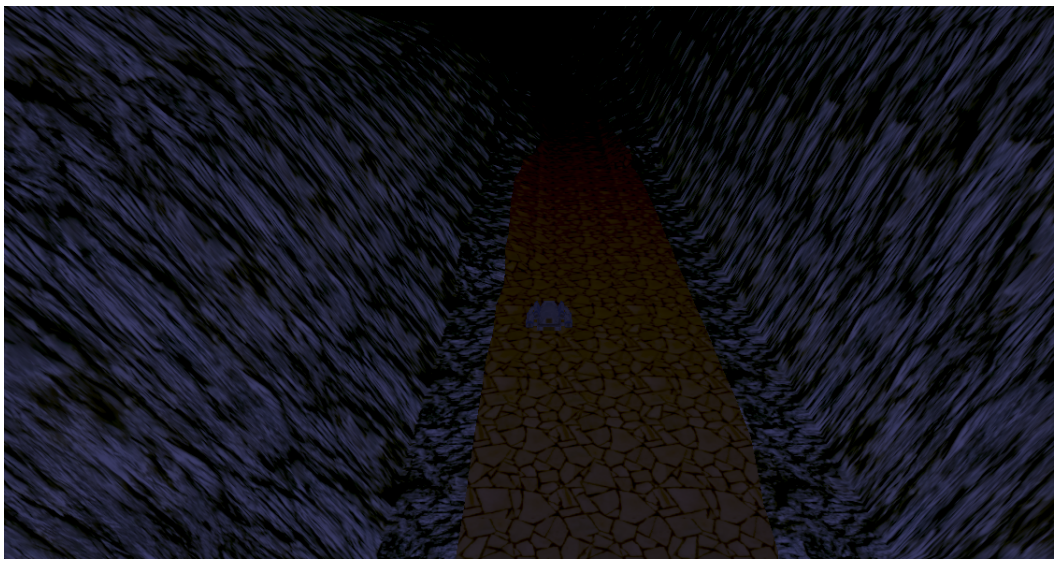


Figure 14: The Incorrectly Shaded Fog

I figured that this was happening as the distance in the applyFog function was making a negative version of the distance that was passed into it. To fix this, I simply had to change from this:

```
float fogAmt = 1.0 - exp(-distance*b);
```

To this:

```
float fogAmt = 1.0 - exp(distance*b);
```

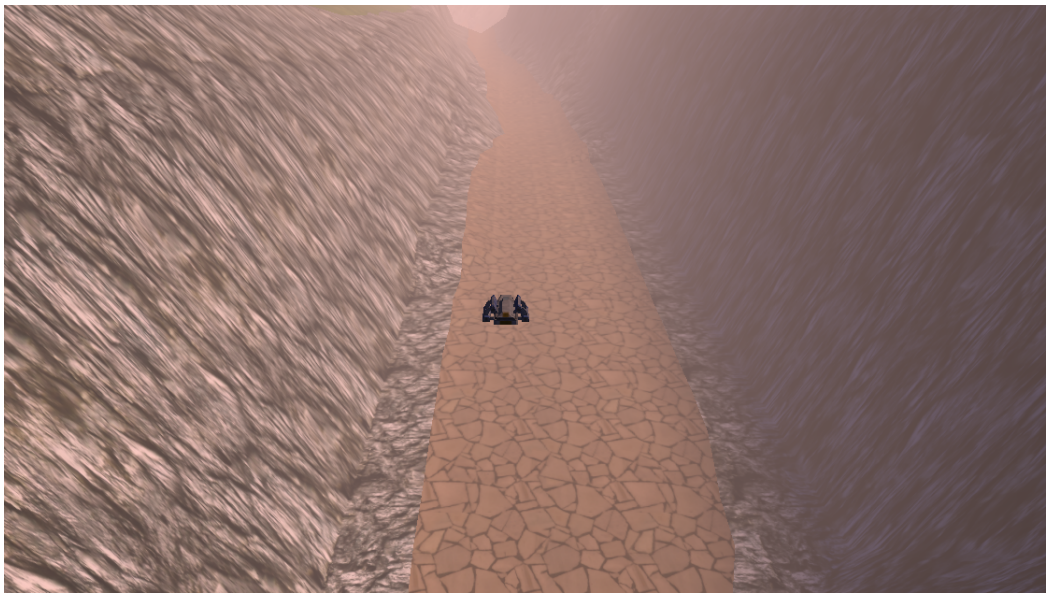


Figure 15: The Correctly Shaded Fog

The correctly shaded fog looked like that seen in Figure 15, this fog stayed constant throughout the sun cycle in the game and allowed for depth in the game to be seen more clearly.

9 Adding Props

Adding props was very similar to that of adding the racer as it involved loading model objects directly into the game and placing them but this time there was no need to worry about an update function as the props would only need to be loaded and rendered once within the game.

First, I created a Prop class and imported it into the mega racer file. This prop class was coded the following way and was very similar to the racer class without any update functions and didn't take in keybinds:

```
class Prop:
    position = vec3(0,0,0)
    heading = vec3(1,0,0)
    model = None

    def render(self, view, renderingSystem):
        top = [0,0,1]
        modelInWorld = lu.make_mat4_from_zAxis(self.position, self.heading, top)
        renderingSystem.drawObjModel(self.model, modelInWorld, view)

    def load(self, objModelName, position):
        # TODO 2.3: Load the prop
        self.position = position
        self.model = ObjModel(objModelName)
```

Once this class was made, I created a list that would hold all the props the game to be rendered. This prop list would hold a bunch of prop objects in it which would make it easier for me to place many props into populate my game.

```
    g_prop_list = []
```

Once I had this list, I then added a function into the render function to allow for me to see the racer's coordinates so that I could place the props into the map.

```
    print("[{},{},{}]".format(g_racer.position[0], g_racer.position[1],
        g_racer.position[2]-3))
```

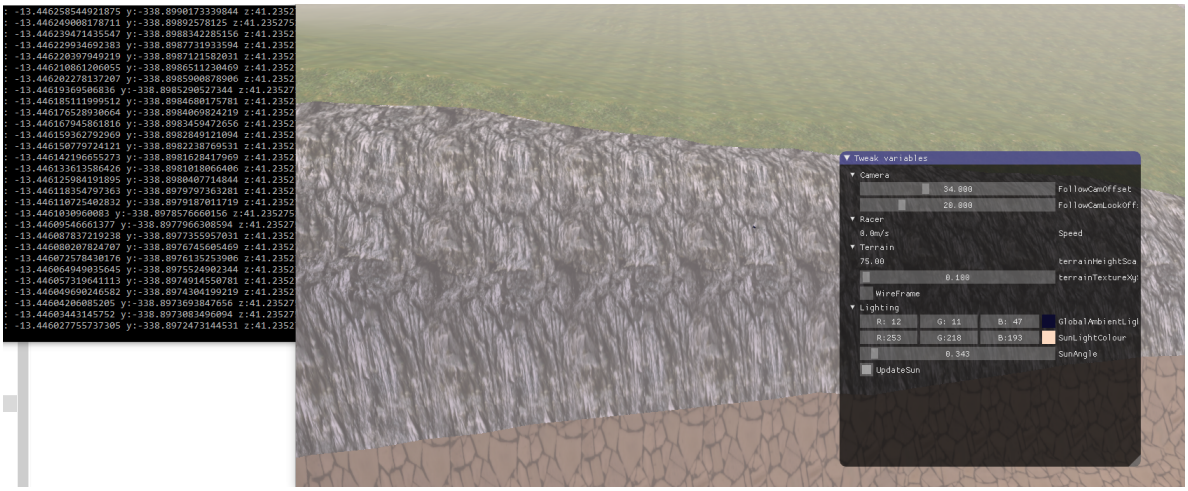


Figure 16: Getting the Coordinates from the Racer's Position

Using these coordinates, I was able to create a bunch of for loops which iterated through a list

of chosen coordinates and had a similar format to the following:

```
for pos in position_list:
    prop = Prop()
    prop.load("prop/texture/path", pos)
    g_prop_list.append(prop)
```

I also added this for loop into the renderFrame function in the rendering system class so that the props were able to be rendered in the game:

```
for prop in g_prop_list:
    prop.render(view, g_renderingSystem)
```

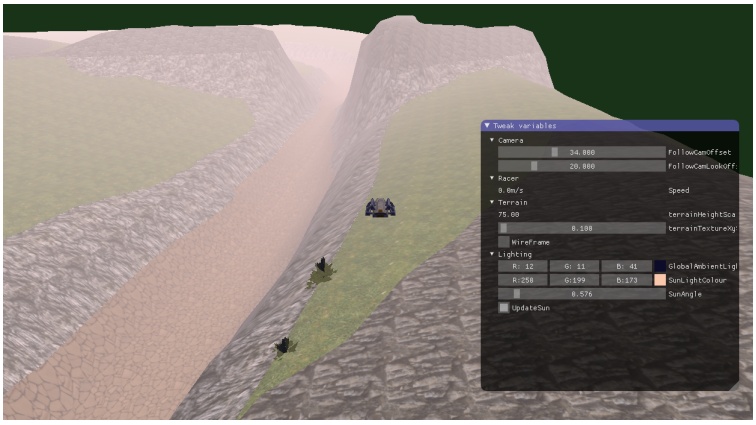


Figure 17: The Incorrectly Placed Props

This however, did not turn out to be what I wanted as the trees were not being placed direcly on the surface and looked unnatural. I figured this would be due to the racer flying on the terrain rather than touching it directly. After some trial and error I discovered that adjusting the racer's z-axis coordinate by 3 was enough to bring props back to touch the surface.

One thing that I was able to make with the props was this pattern of small trees and stones. I thought it looked nice and was easy to implement.

Figure 18: My Prop Design

10 Racer Headlights

Racer headlights were a little harder to implement than other features that I had attempted, this was due to the feature not improving upon an already existing one. I did however have an idea, this idea was to build upon the already existing sun shader I had implemented earlier and use it in a similar fashion to that of a global shader which revolved around the racer. I was hinted upon by a tutor to use the follwing code within my rendering system class.

```
lu.setUniform(shader, "viewSpaceHeadlightPosition", g_racer.position);
lu.setUniform(shader, "headlightColour", g_headlightColour);
lu.setUniform(shader, "viewSpaceHeadlightDirection", g_racer.heading);
```

This code was then to be used within the fragment shader to calculate the global lighting of the racer's headlight. I also changed the computeShading function to take in more parameters so that it was possible to calculate this headlight.

```
vec3 computeShading(vec3 materialColour, vec3 viewSpacePosition, vec3
    viewSpaceNormal, vec3 viewSpaceLightPos, vec3 lightColour, vec3
    viewSpaceHeadlightPos, vec3 headlightColour, vec3 viewSpaceHeadlightDirection)

# Changed from
vec3 computeShading(vec3 materialColour, vec3 viewSpacePosition, vec3
    viewSpaceNormal, vec3 viewSpaceLightPos, vec3 lightColour)
```

I then used the code supplied by the tutor to calculate the point direction and the headlight direction of the racer's headlight.

```
# Supplied by the tutor
vec3 pointDirection = normalize(viewSpacePosition - viewSpaceHeadlightPos);
viewSpaceHeadlightDirection.z = viewSpaceHeadlightDirection.z - 0.5;
vec3 headlightDirection = normalize(viewSpaceHeadlightDirection -
    viewSpaceHeadlightPos);
```

I tried using the same tactic that I used in the improving textures section where I got the normal of the two calculated vectors and found the dot product between them.

```
vec3 hd = normalize(headlightDirection);
vec3 pd = normalize(pointDirection);
float angle = dot(hd, pd);
```

I then created a threshold for the angle, if it was below 45 degres/close to the racer, make the colour brighter, if it wasn't, give the light a normal brightness.

```
if(angle < 0.45) {
    cutOffShade = vec3(5,5,5);
} else {
    cutOffShade = vec3(1.0,1.0,1.0);
}
```

I then multiplied the cutOffShade by the existing multipliers to add to the outgoingLight variable in hopes of creating a headlight of some sort.

```
vec3 outgoingLight = (incomingLight + globalAmbientLight) * cutOffShade *
    materialColour;
```

The result of this looked like the following:

After playing around with various combinagions in an attempt to get a spotlight falloff and more cone looking, I was unsuccessful in being able to implement a working spotlight. This was mainly due to the time limit I was given. I was however, able to implement the basics for headlights and am happy with the the end result of this feature regardless.

11 Conclusion

Overall, I was very happy with what I was able to implement within this game. I struggled a bit with the shaders and the GLSL parts in general. If I had more time I would have loved to implement nice particle effects and possibly more animations within the game.

Thank you for reading this report, I hope you enjoyed my documentation and my half semester journey and experience with computer graphics.