Name: Anna Hekkink
Student number: 45964345

COSC3000 Computer Graphics Project Report:

Mega Racer

AIMS:

The aim of this report is to document my implementation process, including being clear about when and why I have used code from the labs. I will also include screenshots of my Mega Racer computer game after completing each task.

My overall aim of this project was to implement the graphics of a simple racing game (which was already provided with game logic). In doing so, I aimed to make the game look more interesting and photorealistic. The techniques I used to achieve this included implementing cameras, texturing, Lambertian lighting and shading.

IMPLEMENTATION OF TASKS:

1. Tier 1 Tasks to Earn a Baseline Grade

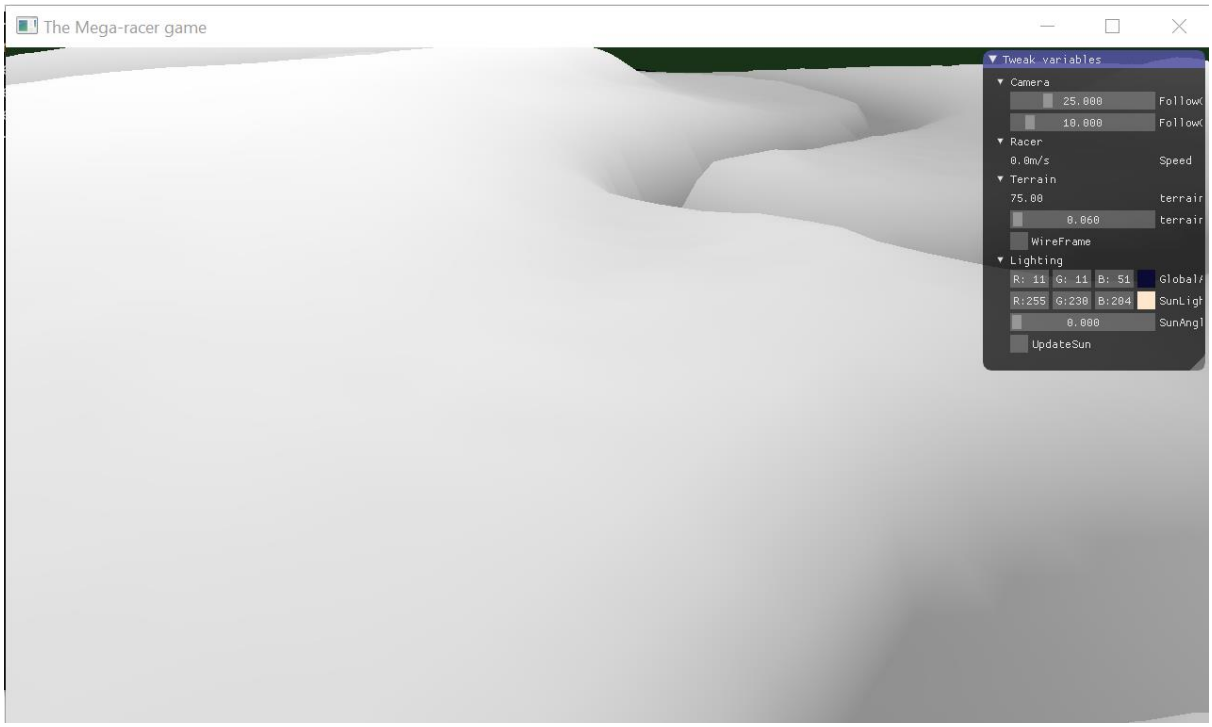
1.1. Scale the Terrain Grid

The first aim was to scale the terrain grid by calculating the z-coordinate for each vertex.

We are given imageData in the form RGBX (or RGBA) where R represents the height of the terrain, which is given as an integer in the range [0, 255]. We also use a scale heightScale = 75.0, meaning that the value R = 0 refers to the lowest point at 0m and R= 255 refers to the highest point in the game at 75m.

To calculate the z coordinate for each vertex, we first normalise the channel from [0, 255] to [0.0, 1.0] and name this value ‘red’. Then, we define the z-coordinate as follows:

zPos = red * self.heightScale



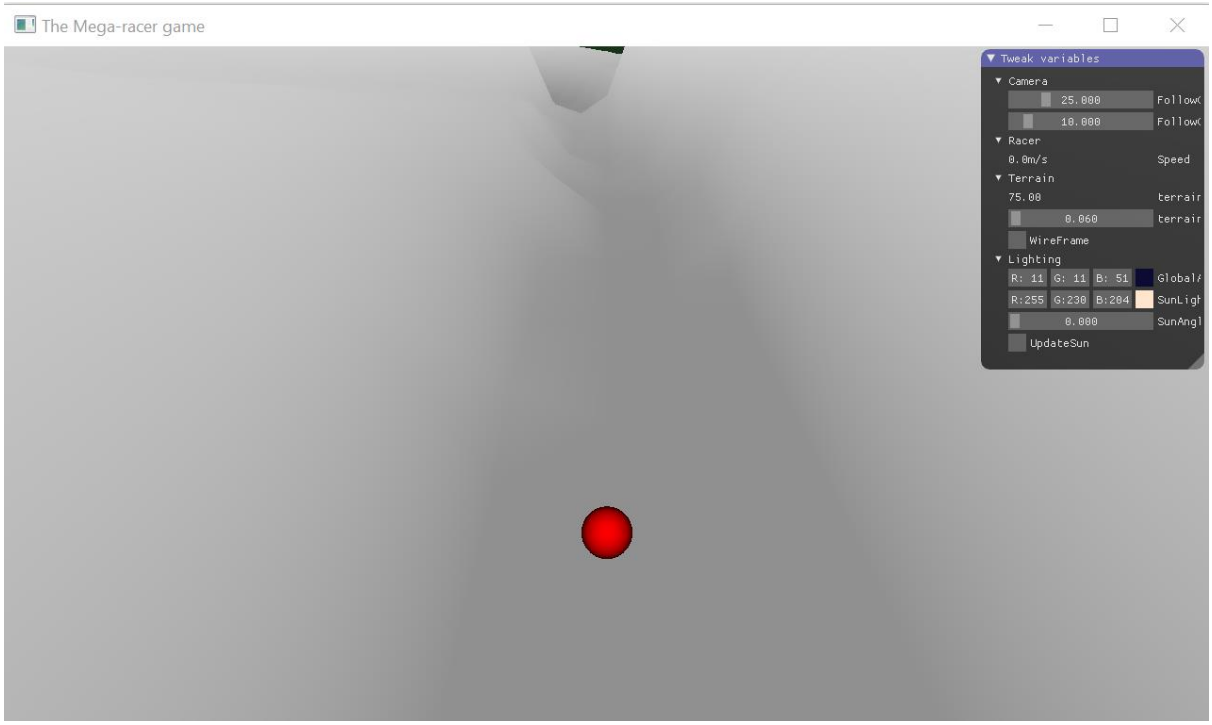
Task 1.1: Scale the Terrain Grid

1.2. Set up a camera to follow the racer

The subsequent task was to position the camera such that it follows the racer at an offset behind and above and looks at the racer. We can access the position coordinates of the racer using g_racer.position, and we can also access the direction that the racer is pointing by using g_racer.heading.

In order to set the camera behind the racer at an offset of magnitude g_followCamOffset, I first normalized the g_racer.heading vector to make it length 1, then multiplied it by the scalar value of g_followCamOffset. This new vector will then need to be subtracted from g_racer.position in order to place the camera behind the racer. Then, to position the camera above the racer as well, I simply added the g_followCamOffset to the z-coordinate of the position vector.

Finally, to make the look target of the camera sit just above the racer with an offset of g_followCamLookOffset, I set the view target to g_racer.position and added the offset to the z-coordinate.



Task 1.2: Set Up a Camera to Follow the Racer

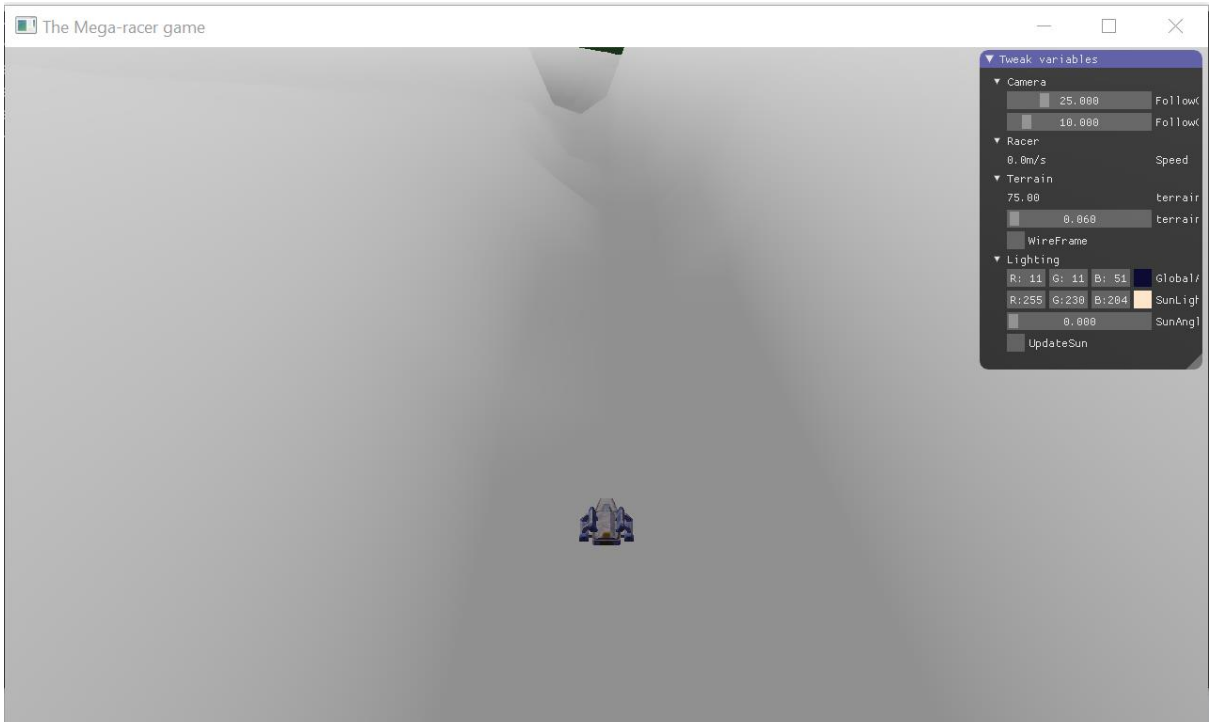
1.3. Place and orient a model for the racer

Following this, our next aim was to replace the red dot with an actual model for the racer. We wished to position this model into our world and scale, centre and rotate it such that it was consistent with our world space.

Firstly, I loaded the model for the racer from data/racer_02.obj. [I chose to use the low-poly model provided in order to speed up loading time]

Then, the task remained to render this model into the world and apply the necessary transformations. I followed the methodology discussed in lecture 2 to achieve this. I chose to use the z-axis as the model-space forward direction, so that I was able to utilise the make_mat4_from_zAxis function in lab_utils. I set the translation parameter to the position of the racer, stored in self.position. I then set the the 'z-axis' as the forwards direction of the racer, i.e. its heading. And then I set the 'y-axis' as the up direction in our world space, i.e. [0,0,1]. This function scales, centres and rotates our model to position it in our world space with consistent conventions.

Finally, I used the function renderingSystem.drawObjModel to render the model racer and draw it into the world using our transformations.



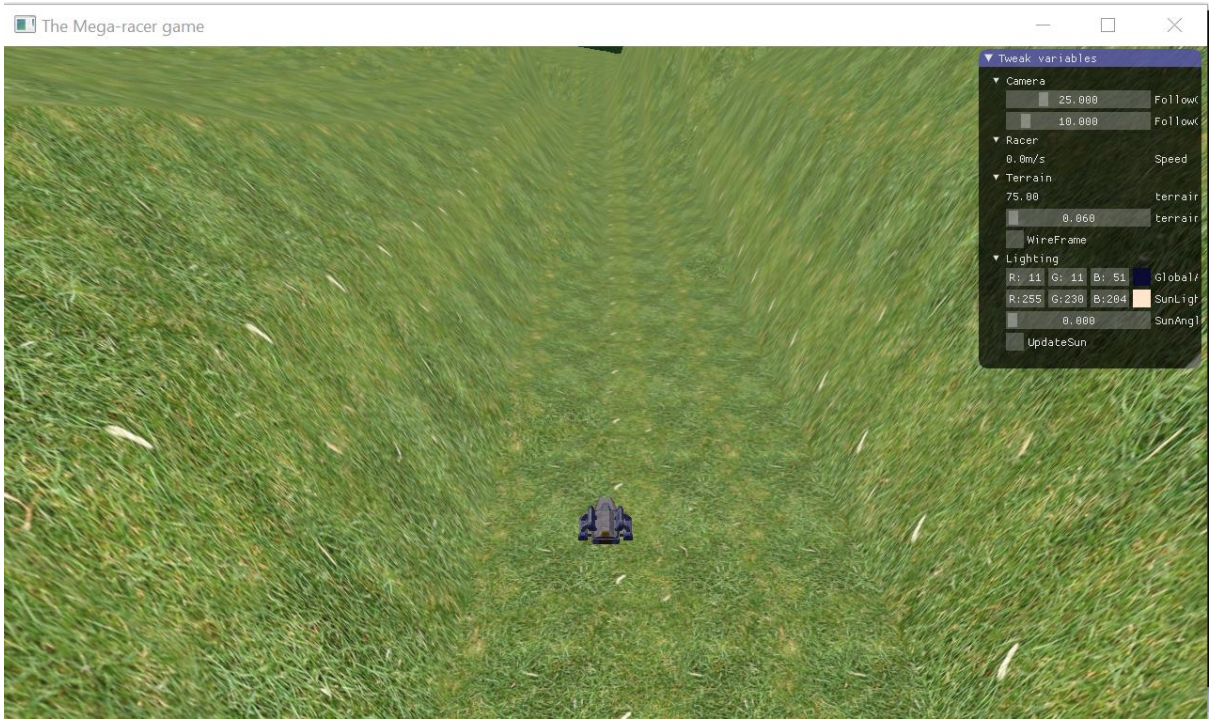
Task 1.3: Place and Orient a Model for the Racer

1.4. **Texture the Terrain**

The next task was to texture the terrain using the grass texture stored in ‘grass2.png’. To achieve this, I completed the following tasks:

1) Loaded the "data/grass2.png" texture using ObjModel.loadTexture and named it grassTexId.
2) Bound the texture to its texture unit using lu.bindTexture(self.TU_Grass, self.grassTexId).
3) Set the sampler uniforms, using lu.setUniform(self.shader, "grassTexId", self.TU_Grass), in order to allow the shader to access the texture.
4) Ensured the wrap was set to GL_REPEAT, since we wish to tile this texture over the whole domain.
5) Edited the fragment shader to make it sample the texture. Firstly, I defined the texture coordinates in terms of the world-space coordinates, using textureCoord = v2f_worldSpacePosition.xy. Then, I used texture() to retrieve the texels from my

GrassTexId texture and apply them to the texture coordinates scaled by the terrainTextureXyScale. I stored these in grassColour. And finally, I converted this 2D vector to a 3D vector using .xyz (as seen in lab 5), so that I had a 3D vector stored in grassColour, which I could use to redefine the materialColour variable. This is then passed to our computeShading() function, which will be used to compute the lighting of our scene. Eventually, this is outputted as our fragment colour.



Task 1.4: Texture the Terrain

1.5. **Lighting from the Sun**

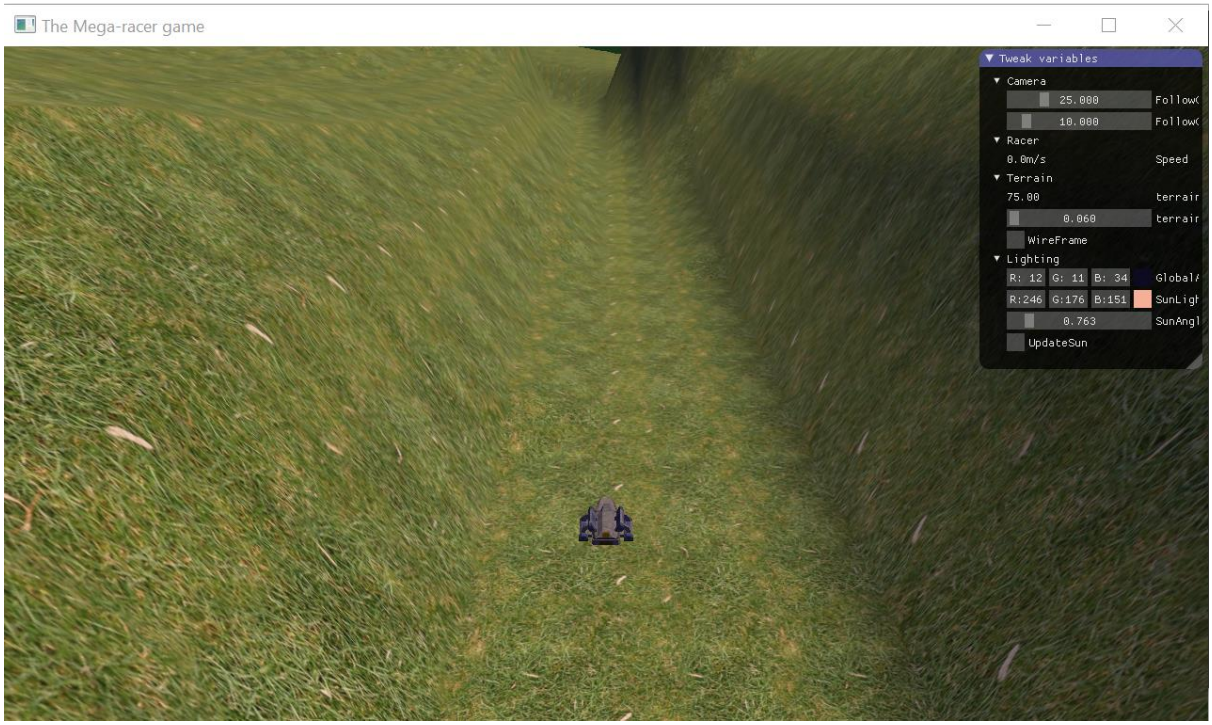
Then, our next aim was to implement a Lambertian shading model in all the shaders. To ensure both my shaders behaved in the same way, I wrote the code in a function called computeShading() and then called this function in the fragment shaders (which was already included in the block of code).

For this task, I used lab 4 as a template to work from. Essentially, we just need to calculate the amount of light arriving at the point which we're shading. I achieved this with the following steps:

1) I computed the normalized direction *from* the point we're shading *towards* the light source and stored this as vec3 viewSpaceDirToLight. I also normalized the surface normal, in case it wasn't already normalized.
2) I computed the incoming light intensity using the fact that it is proportional to the cosine of the angle between the surface normal and direction to the light. I utilised the dot product to achieve this and clamped the result at zero to prevent any weird looking artefacts.
3) I set the incoming light to the intensity multiplied by the light colour (which is inputted as sunLightColour inside the fragment shader).

4) I added a constant ambient lighting term to prevent things being pitch black when not facing the light.
5) I multiplied the material colour by this Lambertian lighting effect and outputted the result.

To improve my lighting model, I could have two colours for the ambient lighting and blend between them based on the orientation of the surface, e.g. surfaces that face downwards would get a green ambient light, and those facing upwards would get a blue tint from the sky.



Task 1.5: Lighting from the Sun

2. Tier 2 Tasks to Earn a Better Grade

2.1. Improve Terrain Textures

An aim of this task was to select different textures based on the terrain material. i.e. use a different texture for the road part of the terrain, in order to indicate to the player where they will be able to move faster. Another aim was to add variation to the non-road part of the terrain, by using height- and slope-based selection of texture. This is the part I attempted first, since it seemed like the simpler task, and I achieved it using the following steps:

1) I loaded a couple more textures in the same way as I did for the grass. Namely, I used ObjModel.loadTexture() and loaded the "data/rock2.png" file for my high texture, and the "data/rock5.png" for my steep texture. I named these and bound them to their corresponding texture units: TU_RockHigh and TU_RockSteep.
2) I created samplers in the fragment shader in the same way as the grass. Namely, I used vec3 highColour = texture(rockHighTexId, textureCoord * terrainTextureXyScale).xyz; and the same for the steepColour but using rockSteepTexId instead.
3) I determined if the terrain was steep, by using the fact that if the normal * [0,0,1] is almost equal to zero, then the normal is close to horizontal, which implies the terrain is close to vertical. Thus, I set the steepness as the absolute value of the dot product of those two normalised vectors. And I set a steepness threshold = 0.8, which roughly corresponds to having at least 35 deg steepness. If this threshold was satisfied, then I set the materialColour = steepColour.
4) I then determined if the terrain was highly elevated by normalising the height (dividing by the terrainHeightScale) and used a threshold value of 0.85. If the height was above this threshold, I set the materialColour = highColour.
5) Finally, I updated the material colour to blend the textures by mixing in the rock texture in varying amounts dependent on its height.

The next part of this task took a bit more work, mainly because the information for the type of terrain was not immediately available to the fragment shader. I had to create a new texture which contained the data itself (not colours), and then sampled from it using normalized texture coordinates. I could then use the data stored in the blue channel of this texture to determine if the terrain was road or not. To achieve this, I completed the following steps:

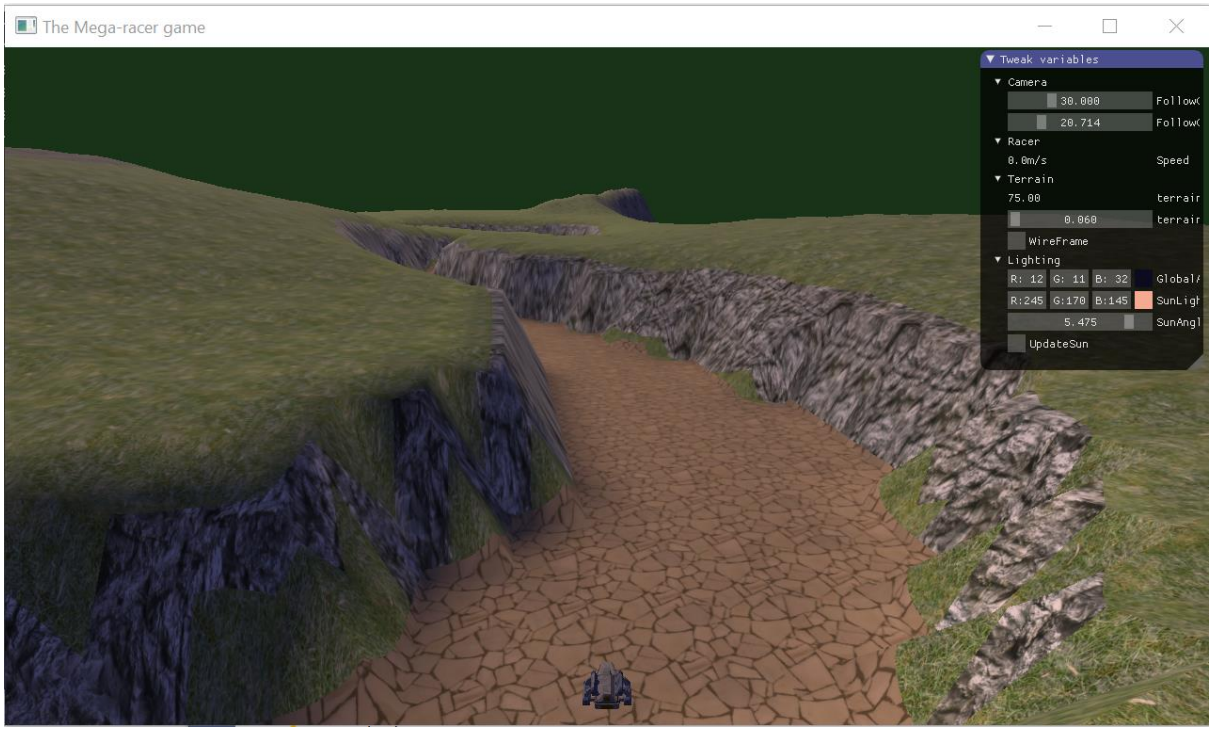
1) Loaded and bound the extra texture for the road, just as before for the grass and rocks. Then, I also created the texture containing the track data using ObjModel.loadTexture(), but setting srgb=False. And bound this to its texture unit: TU_DataMap.
2) Then, I computed the normalized the texture coordinates in the vertex shader and passed these to the fragment shader. This was done by using the x,y coordinates from the world space position, subtracting the xyOffset and then multiplying by the xyNormScale.
   i.e. v2f_normalisedTexCoord = ((v2f_worldSpacePosition).xy - xyOffset) * xyNormScale;
3) Now that I had the data map and normalized texture coordinates, I retrieved the data in the blue channel using:
   float blue = texture(terrainDataSampleTexId, v2f_normalisedTexCoord).z;
4) Next, I used an if statement to determine if the terrain was road or not. If the terrain point is road, it should have a value of 255 in its blue channel, so float blue should equal 1. To allow for some tolerance, I asked for blue > 0.95 in the if statement.

5) Finally, if this condition was satisfied, I sampled from the road texture using: vec3 roadColour = texture(roadTexId, textureCoord * terrainTextureXyScale).xyz; And then set the material colour = road colour.

Note that I placed this bit of code after we blending the previous texture together using mix(), this meant that the road colour would write over whatever the material colour was previously set as for that fragment. Hence, the road remained as its original texture colour, and was not blended with the rocks.

The material colour is then passed to the computeShading() function to compute the Lambertian lighting for the scene, just as before.

Task 2.1: Improve Terrain Textures



Task 2.1: Improve Terrain Textures

2.3. Add Props!

My next aim was to add some props to the non-road parts of the terrain. These were to be placed randomly within the specified areas of the map that they were allocated, and also rotated randomly to create some variation.

Firstly, I created a new class called 'prop' in the terrain.py file. The structure of this class was heavily based on the structure for the 'racer' class (but simplified since there was no need to update it). In my class, I stored information on the prop's position, the direction it was facing by default, and the random rotation that would be applied.

Inside my load() function I made the inputs as follows:
def load(self, objModelName, terrain, n, renderingSystem):
The only one different to the racer class, is the input n. This will be a random number generated in mega_racer.py to determine which position in the list of possible prop positions will be used when rendering this prop. In this load() function, I also created a random angle between 0 and 2*pi to be assigned as the random rotation of the object. To do this I needed to import random, and then used the function random.uniform(0.0, 2*math.pi).

Inside my render() function, I applied the appropriate transformations needed for the prop model to be placed into the world with the appropriate scaling, translation and (default) rotation. I used the make_mat4_from_zAxis function from lab utils to do this. Then I rotated the model around the z axis with the random rotation angle that was defined in the load() function, using lu.make_rotation_z(). And finally drew the prop in the world using renderingSystem.drawObjModel().

Then, back to my mega_racer.py file, I first imported the class 'Prop' from terrain.py. Then I initialised the vectors to store the props and their positions, each with a fixed length. I made

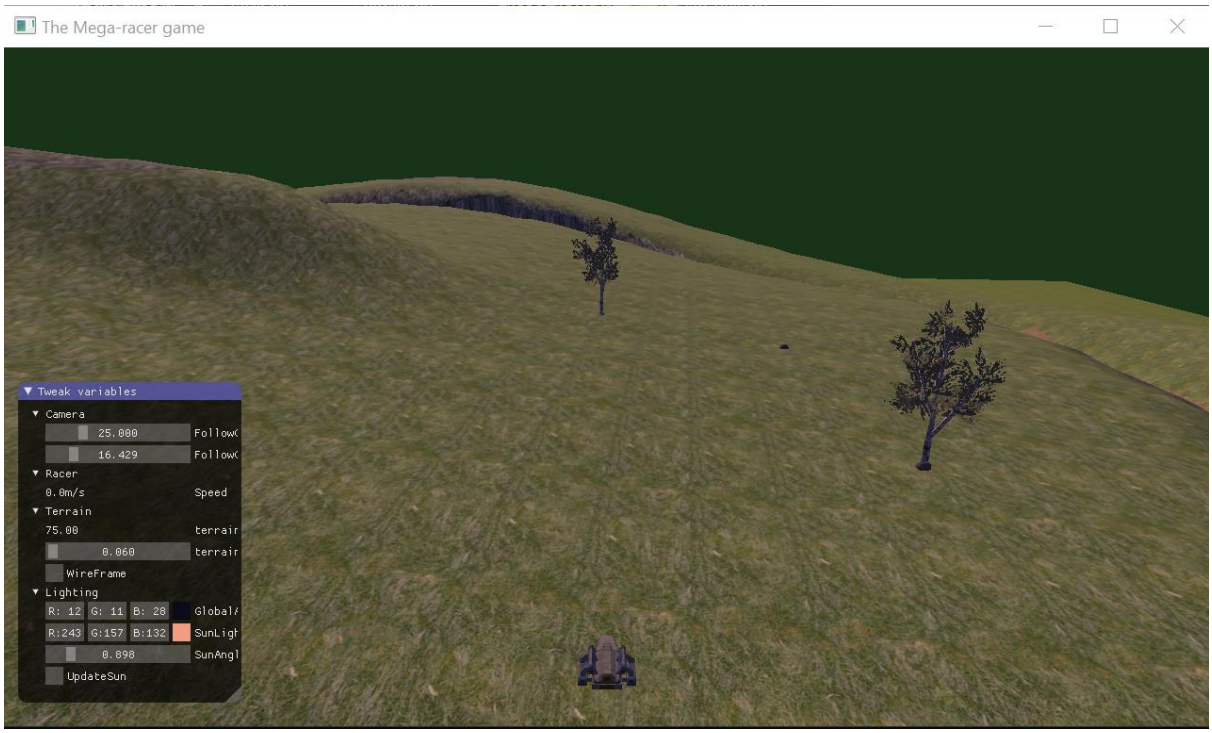
separate vectors for the trees and the rocks. E.g.
myTreeProps = [None]*10
myTreePropPositions = [None]*10

Next, in the main program, I load 10 trees and 10 rocks using a for loop for each. Inside my loops, I set prop = Prop(). Then I generate a random integer, n, between 0 and (the length of the list) - 1. Here, the list I am referring to is ‘g_terrain.treeLocations’ or ‘g_terrain.rockLocations’. Then load the model for the prop using:
prop.load("data/rocks/rock_01.obj", g_terrain, n, g_renderingSystem);
And then add this prop and its position to the corresponding storage vector.

Finally, in the renderFrame() function, I render all the props inside the myTreeProps vector, and all the props in the myRockProps vector using a for loop as follows:

```
for thisProp in myTreeProps:
    thisProp.render(view, g_renderingSystem)
```

Task 2.3: Props!



Task 2.3: Props!

2.4. Headlights

The aim of this task was to implement a global spotlight and make it work like a headlight. I managed to partially complete this task, however, in the end I could not configure it in a way that made it follow the actual racer and not my camera. I believe a model-to-world and a world-to-view transform is needed somewhere in my code, but I'm not sure where exactly.

Irrelevant of the fact that I did not manage to completely finish this task, I have still documented the process I went through here.

Firstly, I set up the parameters needed to define the spotlight by initialising them, setting up uniforms and defining them as global variables. The parameters included its position, direction, angle width, range, and colour. I also defined its offset as the amount it will be directed downwards, and I calculated a cut off value as cosine of the spotlight angle, which will be used to determine whether or not a fragment lies inside the spotlight cone.

Inside the update() function in mega_racer.py, I set the spotlight position to the g_racer.position, in an attempt to make it work like a headlight. I also set the direction as g_racer.heading – [0, 0, g_spotlightOffset], and normalized the vector. This was intended to

make the spotlight orientation the same as the racer's, and then point it downwards somewhat.

Next, I created a function in the common fragment shader which was similar to my computeShading() function. I named this function computeHeadlightShading() and inputted the material colour, fragment position and surface normal, and all of the needed spotlight parameters. Inside this function, I computed the following:

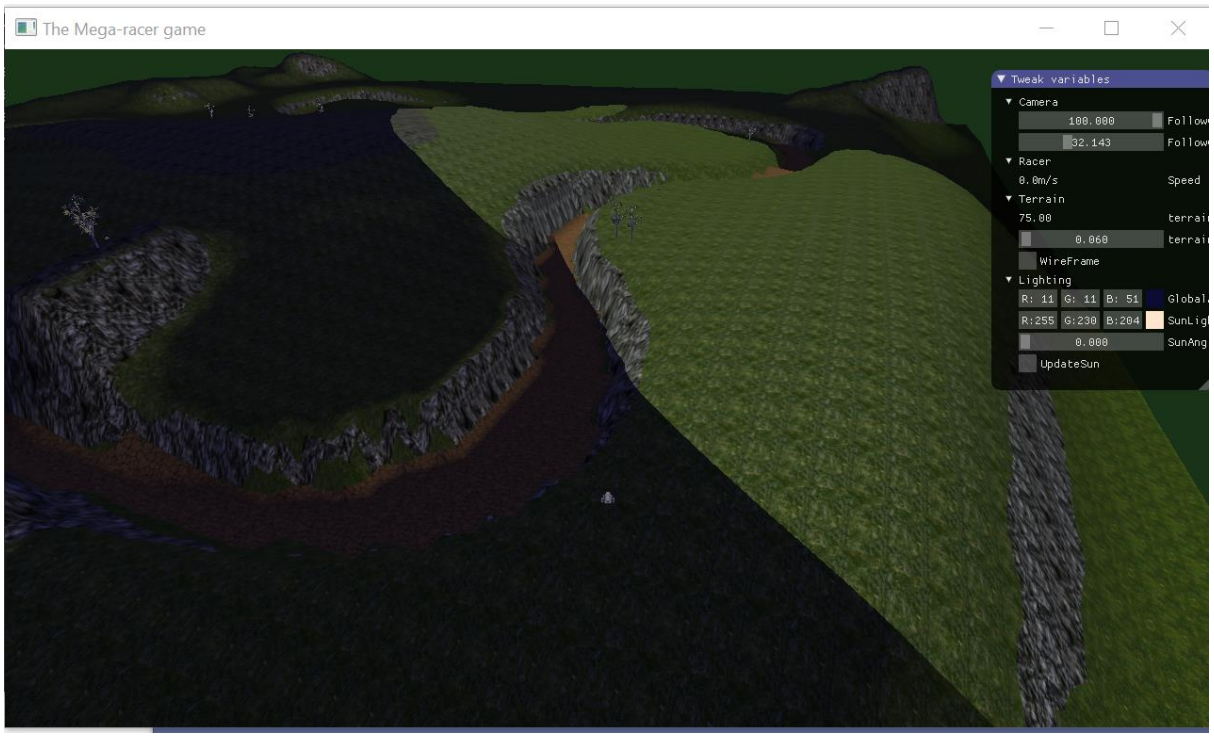
1) I calculated the vector pointing from the fragment to the headlight, and the length of this vector to determine the distance from the fragment to the headlight (which would be used to determine if the fragment was within range).
2) I calculated a value which could be used to determine whether or not the fragment was inside the cone of the spotlight. Namely, I calculated the dot product of the (negated) spotlight direction and the direction from the fragment to the spotlight, both of which were normalised, (and clamped the dot product at zero to avoid negatives). This value is proportional to the cosine of the angle between these two vectors. Hence if this value was less than our 'spotlightCutOff' parameter, the fragment lies inside the cone, and so the lighting calculations will be performed.
3) To calculate the lighting from the spotlight, I first normalized the surface normal. Then, I computed the incoming light intensity using that would be proportional to the cosine of the angle between the surface normal and direction to the headlight , and utilised the dot product to do this, again clamped at zero. Next, I multiplied the incoming intensity by the spotlight colour, and added global ambient light. I called this vector 'incomingLight'. Finally, I set the output colour as the incoming light multiplied by the material colour of the fragment.

This function was then called within the fragment shader in the terrain.py file, and the output was named headlight. Finally, I multiplied the reflected light by the headlight in the fragment colour calculation to add this lighting effect to the shader.

Unfortunately, when I run my program, the spotlight does not align with the position of the racer as I hoped it would, and I failed to figure out which transformations would fix this, and where they should be placed in my code.

If I had successfully managed to implement the headlight, I could have extended this task by creating a UI slider for the downward offset of the headlights, to allow this parameter to be changed in real time, as well as one for the spotlight angle and colour. In addition, I could have made the spotlight's intensity decrease with distance by using the inverse square law.

Task 2.4: Headlights



Task 2.4: Headlights