COSC3000 - COMPUTER GRAPHICS REPORT

Olympics Athletic Track

Tean-louise Cunningham (42637460)

June 5, 2020

Contents

List of Figures

Introduction

In continuance of the exploration of the Olympics from the visualisation project, a graphic simulation of running on an athletic track will be simulated. The main stadium of an Olympics is the most integral location as it hosts the opening and closing ceremonies, the lighting of the touch and the most popular events. The architecture and design choices of these stadiums ensure athletes can compete in optimal circumstances and thousands of spectators can experience these feats. This project will highlight the ingenuity of these stadiums by offering the opportunity to view its design in detail as a competitor on its athletic track. This will be achieved using the provided mega_racer files, demo and instructions as a base, and the Tokyo National Stadium, which is set to host the 2020 Olympics as inspiration.

The presentation of this project can be viewed here.

Mega Racer

Mega Racer is a basic game simulating a race car driving through a wilderness area. The basic game logic, movement and sun light has been provided in the code. Additionally, examples files for the object models, textures and map have been provided and were used for this implementation. There are two tiers of task; basic and advanced. All of the tier 1 tasks were completed successfully as well as the first three tasks from tier 2; improved terrain textures, and the addition of fog and props. These tasks provided a strong foundation for the proposed implementation of an Olympic Stadium in the next section.

Tier 1

These are the basic deliverables that highlight a basic understanding of computer graphic techniques. There are five tasks all of which were completed in the provided code and accompanied here with explanations, screenshots and the relevant code. The provided code produces the world space coordinate system however there are no objects yet in the world.

1.1 - Scale the terrain grid

The map of the world is a RGB image for simplicity. Fundamentally this image is simply a grid of pixels, each of which is an addressable colour sample (Lecture 1). The provided map and game logic relies on the value of the red pixel to indicate the height of the terrain, with 255 as the maximum height. The provided code normalises the R pixel value by dividing it by this maximum value to be a value between 0 and 1, and assigned as 'red'. By multiplying this variable with the provided uniform scaling variable, heightScale, all axis will be scaled to the appropriate units of the world space, in this case 75 metres, for a realistic appearance (Lecture 2). This operation is assigned to the zPos of the terrain when it loads in terrain.py to set the height, as well as uncommenting the last line from racer.py so the racer can follow the height.

```
# terrain.py
def load(...):
    ...
```

```
zPos = red * self.heightScale
...
```

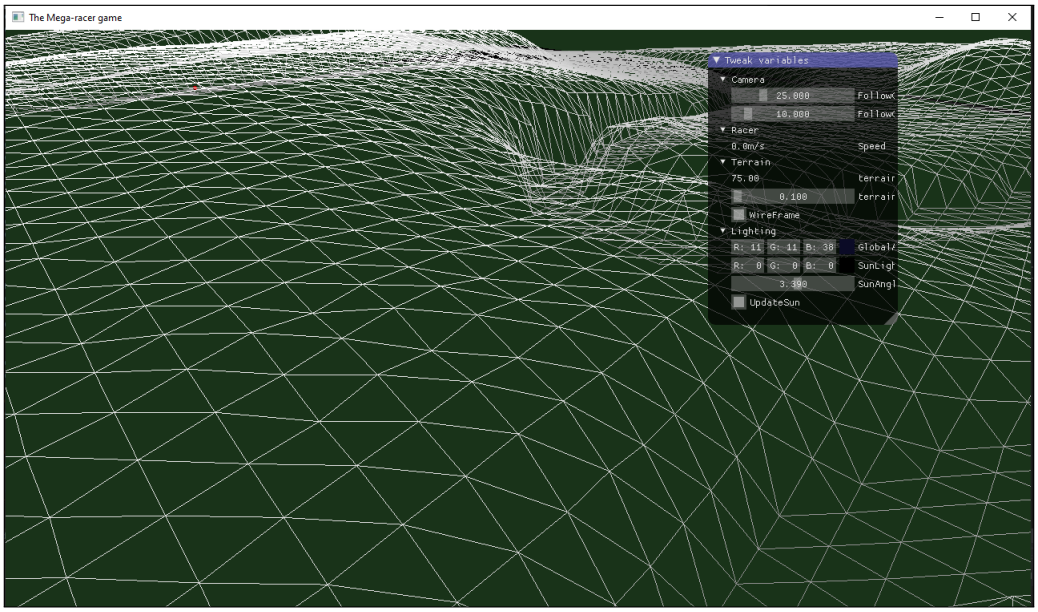

Figure 1: 1.1 - Scale the terrain grid

1.2 - Set up a camera to follow the racer

In this task the coordinate system of the view space will be determined, as the view space is simply the space as seen by the camera. This coordinate system is the result of transforming the world space so that the object is in front of a users view, through a combination of translations and rotations (Lecture 2). In the code, there is a variable clearly labelled worldToViewTransform which is assigned the output of lu.make_lookAt(). This function then calls lu.make_lookFrom() with a modified target parameter for smooth movement. A very similar process is followed in the example from lab2.

```
# Lab 2 (1) - Q5, Lab 2 (2)
worldToViewTransform = magic.make_lookAt(eyePos, [0,0,0], [0,1,0])
worldToViewTransform = magic.make_lookFrom(g_cameraPosition, cameraDirection, [0,1,0])

# mega_racer.py
def renderFrame(...):
    ...
    view.worldToViewTransform
        = lu.make_lookAt(g_viewPosition, g_viewTarget, g_viewUp)
-->
# lab_utils.py
def make_lookAt(eye,target,up):
    return make_lookFrom(eye, np.array(target[:3]) - np.array(eye[:3]), up)
-->
def make_lookFrom(eye,direction,up):
    .....
```

There are four key variables when creating a virtual camera; Up, Camera Position, View Direction and View Target. From the code, the meaning of each of the variables can be understood so they can be appropriately assigned so the world to view transformation is accurate. All of these variables are vectors with xyz coordinates and currently these variables only have default values.

- Up: In this case *g_viewUp* represents the rough up direction and, as commonly used, corresponds to the world up direction. This constrains the orientation of the view to avoid arbitrary roll, though it isn't perpendicular to the view direction (Lecture 3). This value does not need to be updated at any point after being globally declared.
- Camera Direction: The view direction of the camera is often in terms of another position or the view target i.e. where the camera is supposed to look. There is no specific variable relating to this concept, however it is calculated by subtracting the target (*g_viewTarget*) from the eye (*g_viewPosition*) variables in *make_lookFrom()*.

In order for the camera view to be relative to the movement of the racer these values need to be modified in update().

- View Target: The racer is the target and so the corresponding variable, *g_viewTarget*, needs to be updated to the position of the racer with each movement of the racer as this where the camera needs to be pointing. Additionally, the z coordinate requires the addition of *g_followCamLookOffset* to offset the look target above the racer.
- Camera Position: The position of the camera or the centre of projection is basically the the eye of the viewer, which in this case is the *g_viewPosition*.Since the desired view is that of above and behind the racer this variable needs to be updated to take into account corresponding position of the racer (view target), as well as the *g_followCamOffset* (behind and above) for all coordinates. This is achieved using vector addition to add the offset to the racer position.

Attempt 1 - Direction Missing: The result after running these changes performed the calculation of a view space coordinate system that now updated relative to the racer and was in a better position. However, the movement of the racer to the movement of the camera was the opposite and the rotation transformation was off. The camera position should have a relationship with both the view target and view direction, but the view direction has not been taken into consideration.

```
# mega_racer.py
def update(...):
    ...
    g_viewTarget = g_racer.position.copy()
    g_viewTarget[2] += g_followCamLookOffset

    g_viewPosition = g_racer.position + (g_followCamOffset)
    g_viewPosition[2] += g_followCamOffset
```

Attempt 2 - Correct to specification: On closer inspection of the Racer class, *heading* controls the direction of the racer. Since the camera needed to be behind the racer, the camera position (g_viewPosition) had to be multiplied by the inverse of the racer direction using vector multiplication. This produced the correct result in relation to the position of the racer with the offsets applied at the appropriate direction to the racer's movement.

```
# mega_racer.py
def update(...):
    ...
    g_viewTarget = g_racer.position.copy()
    g_viewTarget[2] += g_followCamLookOffset

    g_viewPosition = g_racer.position + (g_followCamOffset * -g_racer.heading)
    g_viewPosition[2] += g_followCamOffset
```

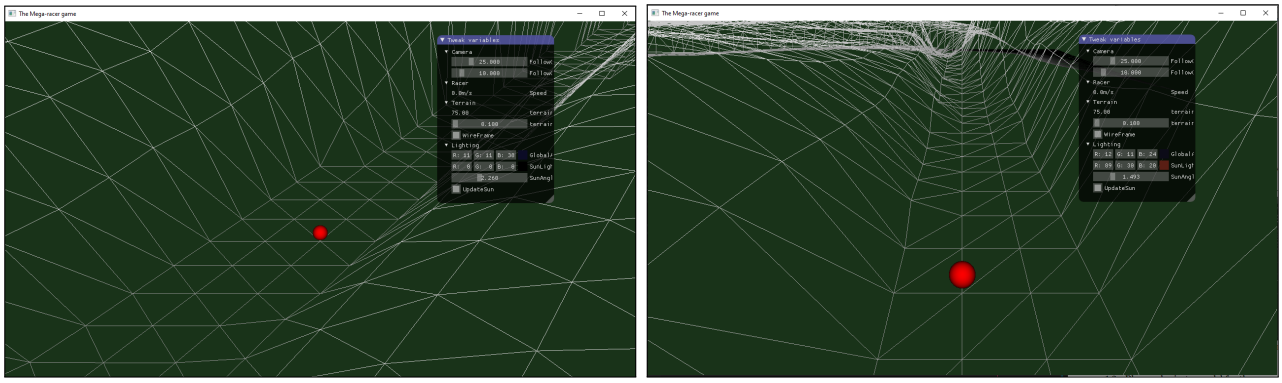

Figure 2: 1.2 - Camera follow racer - Attempts

Although the correct position was calculated, a more intimate view of the racer was desired. So instead the viewTarget was set to equal the racer's position, and the g_followCamLookOffset was added to the z coordinate of the view position. This placed the camera at the same distance above and behind, but instead the racer is in the centre of the camera view.

```
# mega_racer.py
def update(...):
    ...
    g_viewTarget = g_racer.position

    g_viewPosition = g_racer.position + (g_followCamOffset * -g_racer.heading)
    g_viewPosition[2] += g_followCamLookOffset
    ...
```

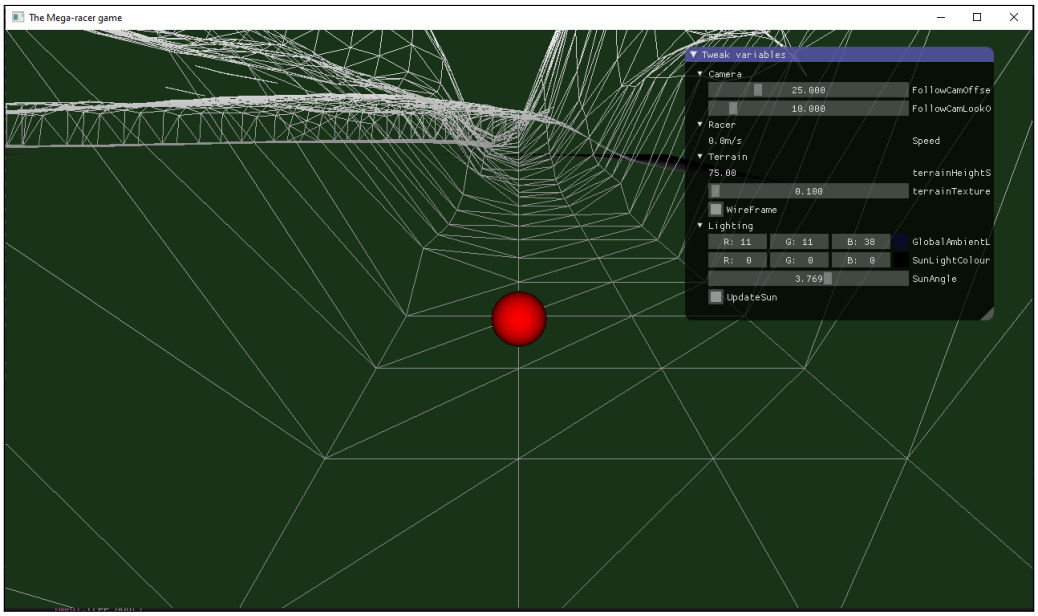

Figure 3: 1.2 - Camera follow racer - Final

1.3 - Place and orient a model for the racer

This section follows the process of an object to world space transformation. The object space is the local coordinate system to the object (racer), where the object is centred at the origin. By transforming to the world space the object will be placed and able to move in this scene appropriately. To complete this transformation there are two functions that need to be updated in racer.py; render() and load().

(1) Rendering the model: In *mega_racer.renderFrame()* the racer is being rendered in the world by calling *racer.render()*. This is where the model of the racer needs to be drawn.

```
# mega_racer.py
def renderFrame(...):
    ...
    g_racer.render(view, g_renderingSystem)
-->
#racer.py - Racer
def render(self, view, renderingSystem):
    .....
```

To render the racer model, it must be drawn and replace the sphere. To draw the object, as per the project notes, *renderingSystem.drawObjModel()* was used. The parameter, *renderingSystem*, is of RenderingSystem class which provides access to *drawObjModel()*.

```
# mega_racer.py - RenderingSystem
def drawObjModel(self, model, modelToWorldTransform, view):
    ...
```

This function has three parameters:

- model: The racer class has a *model* variable set to None.
- view: The the view is of ViewParams class and is set up to project and transform from view to clip space, and world to view space. It is passed in as an argument to *render*.
- worldToViewTransform: To transform the model from its object space to the world space a number of matrix operations need to be performed. The project notes mention *make_mat4_from_zAxis()* lab_utils.py (imported as lu in racer.py) as a useful function to create a transformation matrix. The functions transform the orientation (rotation) and position/movement (translation) of the model by aligning the z-axis exactly with v, the y-axis roughly with u and the the translation is p (Lecture 3).

  ```
  #lab_utils.py
  def make_mat4_from_zAxis(translation, zAxis, yAxis):
  ...
  ```

  The takes three parameters; translation, zAxis and yAxis. The zAxis represents forwards and the yAxis represents up. Therefore, for this world, the z and y axis will act conventionally to the same. This means that the z-axis is the same as the direction of the racer (*g_racer.heading*) and the y-axis is equal to up view of the racer (*g_viewUp*. This provides the easiest solution for rendering the racer. The translation parameter refers to the movement along a vector of model after all the relevant modifications to the coordinates, in this case that is the position of the racer.

With all of the parameters now provided and declared the object model can be rendered. It is important to note that with the above decision to set the z-axis as forwards all object models were set to the same. Also the handedness of openGL is right handed, as well as blender which ensures no issues with rotation. However, at this stage after running the code there was an error message relating to the model; it is of none type. So the next step is to create and load the racer model in load().

```
# racer.py - Racer
def render(self, view, renderingSystem):
    modelToWorldTransform =
        lu.make_mat4_from_zAxis(self.position, self.heading, [0.0, 0.0, 1.0])
    renderingSystem.drawObjModel(self.model, worldToViewTransform, view)
```

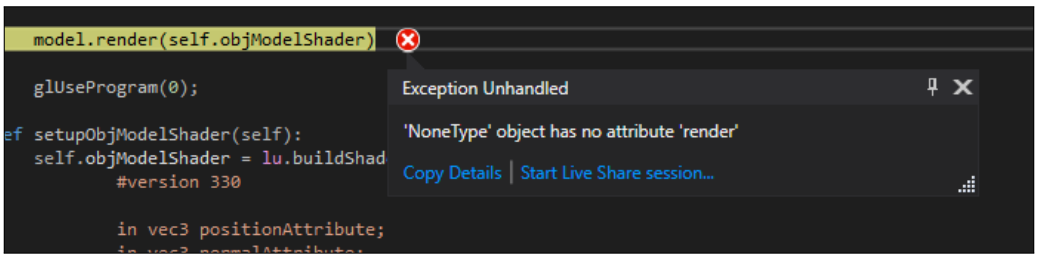

Figure 4: 1.3 -Error

(2) Loading the model: In mega_racer.py, *g_racer.load()* is called with three parameters; the object file for the racer (.obj), the terrain (Terrain()) and the rendering system (RenderingSystem()).

```
# mega_racer.py
g_racer.load("data/racer_02.obj", g_terrain, g_renderingSystem)
-->
# racer.py - Racer
def load(self, objModelName, terrain, renderingSystem):
        .....
```

The purpose of *load()* is to create and load the racer model. According to the project notes, the racer should be loaded as an instance of the ObjModel class, which corresponds to the use of *drawObjModel()* for rendering. To initialise an instance of ObjModel, as per the __init__(), the only requirement argument is the filename which is given as *objModelName*. The terrain and position variables are already assigned using the other parameters. Now the model for the racer has replaced the red dot and the movement of the racer is accurate.

```
# racer.py - Racer
def load(self, objModelName, terrain, renderingSystem):
        .....
    self.model = ObjModel(objModelName)
```

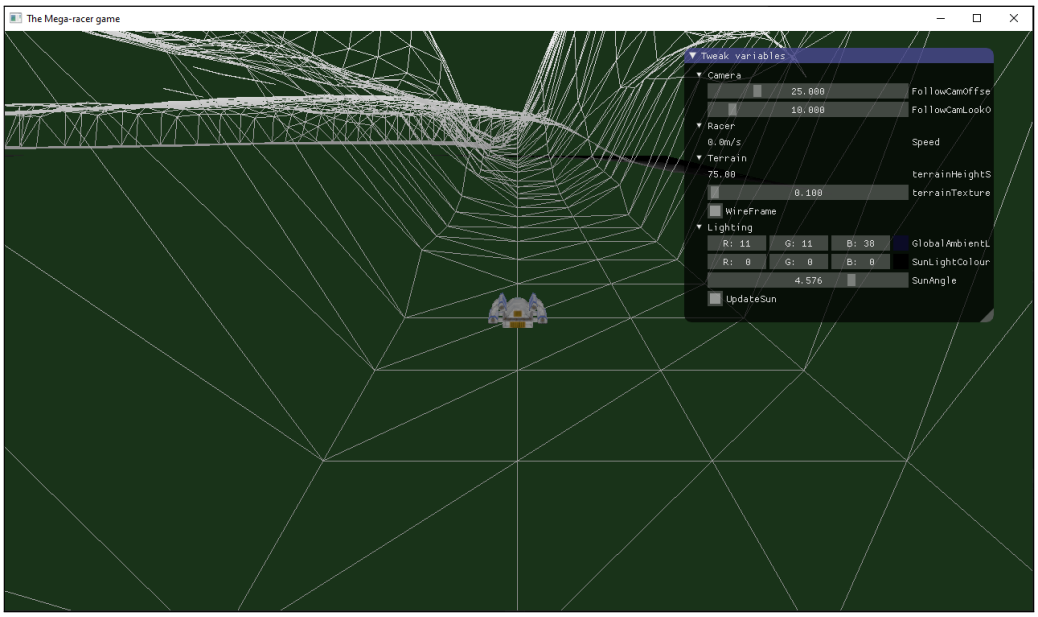

Figure 5: 1.3 - Final

1.4 - Texture the terrain

Textures are a cheap and simple way to add detail that could not otherwise be provided simply by geometric modelling. Texture mapping is simply 'gluing images to surfaces' (Lecture 6). In this task a grass texture will be added over the whole world using 2D texturing. To repeat the texture every metre tiling (texture repetition) will be used to continuously repeat the image to avoid loss of resolution. There are three steps to complete this section in terrain.py.

(1) render() - Bind the grass texture to the right texture unit.
In order to bind the texture, the hint is to is to use lu.bindTexture. This function, bindTexture(), takes two arguments texUnit and textureId. A similar function, with the same name, is used specifically in lab 5 question 12 (Lab Code, 2020). Additionally, setUniform() needs to be called to prepare the texture for the shader.

```
# Lab - lab_utils.py
def bindTexture(texUnit, textureId, textureType = GL_TEXTURE_2D):
    ...
    -->
# Lab 5 Q12 - lab_5_template_2.py
g_detailTexture = None
def renderFrame(...):
    .....
    lu.bindTexture(0, g_detailTexture)
    lu.setUniform(g_shaderProgram, "baseTexture", 0)
```

Following the same process, both *bindTexture()* and *setUniform* require the values of the texture unit allocation and the id of the texture. The *textUnit* for grass is already assigned as *TU_Grass* = 0 in terrain.py. The *textureId* was then declared within Terrain as *grassTexture* equal to None. Additionally, setUniform() requires a shader to be passed, which is already declared as *shader*, and a string name for the texture. In this case the same name as the texture id is used to minimise confusion. With these values declared, the two functions are called in the same way in *render()*.

```
# terrain.py
grassTexture = None

def render(...):
    ....
    lu.bindTexture(self.TU_Grass, self.grassTexture)
    lu.setUniform(self.shader, "grassTexture", self.TU_Grass)
```

(2) load(): Compute the texture coordinates and sample the texture for the grass and use as material colour.
As per the project notes the variable textureXyScale is to be used to scale the texture coordinates and is already set to a factor of 0.1 so the texture repeats every 10 metres. Also the world space coordinates should be used to sample the texture in the fragment shader. To determine how to set the sample, question 12 from lab 5 provided insight into how to set the texture as the colour in the fragment shader.

```
# Lab 5 - Q12
g_texCoordScale = 7.0
def renderFrame():
    ...
    lu.setUniform(g_shaderProgram, "detailTexture", 1)
    lu.setUniform(g_shaderProgram, "texCoordScale", g_texCoordScale)
    ...
```

```
def initResources():
    .....
    in vec2 v2f_textureCoord;
    uniform sampler2D detailTexture;
    uniform float texCoordScale;
    out vec4 fragmentColor;
    void main()
    {   ...
        vec3 detailColour = texture(detailTexture,
                            v2f_textureCoord * texCoordScale).xyz;
        ...
    }

-->
# terrain.py
textureXyScale = 0.1
def load():
    ...
    lu.setUniform(self.shader, "terrainTextureXyScale", self.textureXyScale);
    ...
    fragmentShader =
    ...
        uniform float terrainTextureXyScale;
        out vec4 fragmentColor;
        void main()
        {
            vec3 materialColour = vec3(v2f_height/terrainHeightScale);
            ...
        }
```

From this example it is evident that for the materialColour (i.e. detailColour) to reflect the texture it needs to be assigned the output of texture() with the following three arguments:

- detailTexture: The grass colour is similar to the detailColour so the detailTexture is equivalent to the grassTexture.
- v2f_textureCoord: In lab 5 the v2f_textureCoord is a 2 dimensional out vector from the vertex shader. It refers to the location in texture space. As per the project notes, since the vertex is regular, this same behaviour can be replicated by only selecting the x and y coordinates of the world space which are stored in v2f_worldSpacePosition.
- texCoordScale: The textCoordScale variable in lab 5 refers to *g_texCoordScale* in mega racer which is declared and uniformly set for the shader as texCoordScale. The same is done for textureXyScale in Terrain which as mentioned is nominated in the project notes as the scale for the texture coordinates.

Putting this all together, the texture coordinates were calculated and the texture of the grass was sampled to override the existing material colour.

```
# terrain.py - Terrain
def load(...):
    ...
    fragmentShader =
        ...
        uniform sampler2D grassTexture;
        void main()
        {
            vec3 grassColour = texture(grassTexture,
                                       v2f_worldSpacePosition.xy
                                           * terrainTextureXyScale).xyz;
```

```
        materialColour = grassColour;
        ...
    }
```

(3) load() - fragmentShader: Load texture and configure the sampler.
Also in lab 5, in initResources() after declaring the shader the image is opened and the texture is mapped (glTextParameter), then the texture is loaded using lu.loadTexture(). A similar function can be found in the ObjModel.py (instead of lab_utils.py) and performs the same actions. It also sets the texture to wrap repeatedly as desired.

```
# Lab 5
g_detailTexture = lu.loadTexture("data/details.jpg");
-->
def loadTexture(fileName):
    ....

# ObjModel.py - ObjModel
def loadTexture(self, fileName, basePath, srgb):
    ......
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

To load the texture for the terrain this function needs to be called with the relevant parameters. The *fileName* refers to the name of grass images and the srgb refers to whether it is a 2D image. Using this method the grass is generated with scalability, repetition and compression (Lecture 6).

```
# terrain.py - Terrain
def load(...):
    self.grassTexture = ObjModel.loadTexture("grass2.png", "data", True)
```

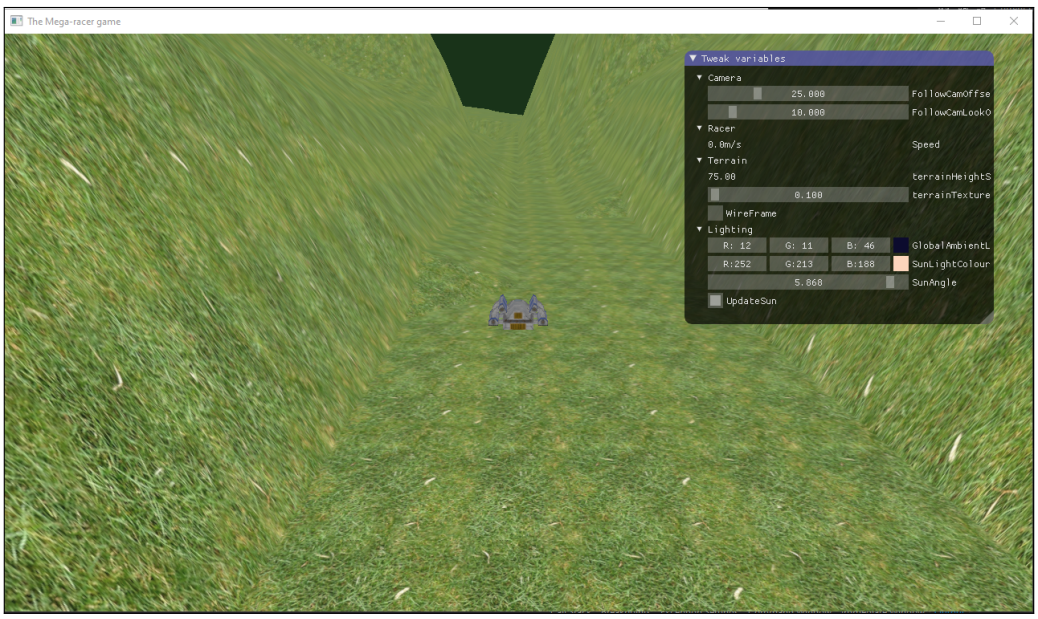

Figure 6: 1.4 - Final

1.5 Lighting from the sun

Lighting provides a more realistic touch and assists with showing the structure of surfaces. Since this is light from the sun, this is a directional light and it can be assumed that the light

rays are parallel since the light source is infinitely far away. A couple of things to note about this source of light:

- Position of sun = g_sunPosition: The sun is set to move around around the world.
- Colour of sunlight = g_sunLightColour = lightColour
- Sun is defined in world space

The project notes explain that the code for the light should be added to the computeShading() function in RenderingSystem class as part of the commonFragmentShaderCode variable. This function is called in setUpModelShader() where the viewSpaceNormal and viewSpacePosition variables are declared by the vertex shader, materialColour is declared as materialDiffuse in the fragment shader, and viewSpaceLightPosition and sunLightColour are carried over from the commonShaderCode variable.

```
#  mega_racer.py - RenderingSystem
def setupObjModelShader(...):
    v2f_viewSpaceNormal = normalize(modelToViewNormalTransform * normalAttribute);
    v2f_viewSpacePosition = (modelToViewTransform * vec4(positionAttribute, 1.0)).xyz;
    .....
    vec3 materialDiffuse = texture(diffuse_texture,
                                   v2f_texCoord).xyz
                                   * material_diffuse_color;
    vec3 reflectedLight = computeShading(materialDiffuse,
                                         v2f_viewSpacePosition, v2f_viewSpaceNormal, vi
                                         sunLightColour
                                       ) + material_emissive_color;
```

When computing the light, the following tips must be adhered to:

- All parameters om shading must be the same space -¿ set as view space (same as lab 4)
- The computeShading function needs to be called from each function -¿ this is already done.
- Need to make sure don't just have Lambertian term -¿ Fixed by multiplying by materialDiffuse term.
- Use clamping to check when light is behind surface.

To create the lighting effect required, all of the code followed that of fragmentShader.glsl in Lab 4. Question 1 was already provided as a basic shader. The view space is used for shading calculations. The computeShading() function is setup with five parameters, all of which are used to return a light value.

```
# mega_racer.py - RenderingSystem
commonFragmentShaderCode =
    ...
    vec3 computeShading(vec3 materialColour,
                        vec3 viewSpacePosition,
                        vec3 viewSpaceNormal,
                        vec3 viewSpaceLightPos,
                        vec3 lightColour)
    {
        return lightValue;
    }
```

(1) The direction towards the source of the light.
Starting from Lab 4, question 2, the first step was to compute the normalised direction towards the light from the shading point in view space. The light position is stored as viewSpaceLight-

Position and the current point is viewSpacePosition (provided by the vertex shader).

```
# Lab 4 - Q2 - FragmentShader
vec3 viewSpaceDirToLight = normalize(viewSpaceLightPosition - viewSpacePosition);

# mega_racer.py - RenderingSystem
commonFragmentShaderCode =
    ...
    vec3 computeShading(...)
    {
        vec3 viewSpaceDirToLight = normalize(viewSpaceLightPos- viewSpacePosition);
        return viewSpaceDirToLight;
    }
```

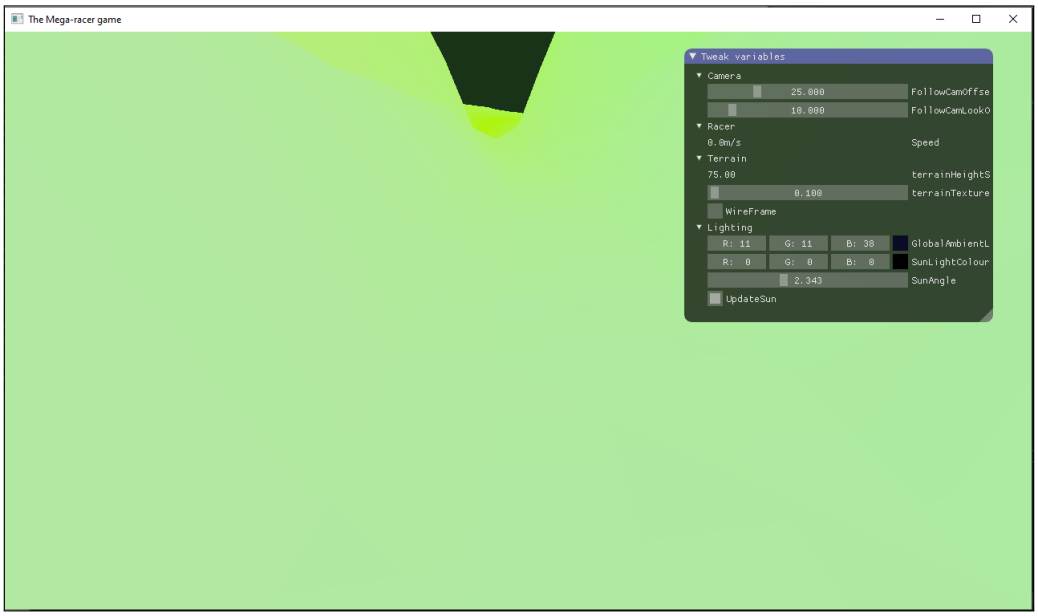

Figure 7: 1.5 - Step 1

(2) Compute the incoming light intensity. Using the now defined *viewSpaceDirToLight*, the incoming light intensity can be calculated. As mentioned, in mega_racer.py three variable of viewSpaceNormal is already provided by the vertex shader. This maintains the unit-length property of the normal.

```
# Lab 4 - Q2 - FragmentShader
vec3 viewSpaceNormal = normalize(v2f_viewSpaceNormal);
float incomingIntensity = max(0.0, dot(viewSpaceNormal, viewSpaceDirToLight));

# mega_racer.py - RenderingSystem
commonFragmentShaderCode =
    ...
    vec3 computeShading(...)
    {
        ...
        float incomingIntensity = max(0.0, dot(viewSpaceNormal, viewSpaceDirToLight));
        return incomingIntensity;
    }
```

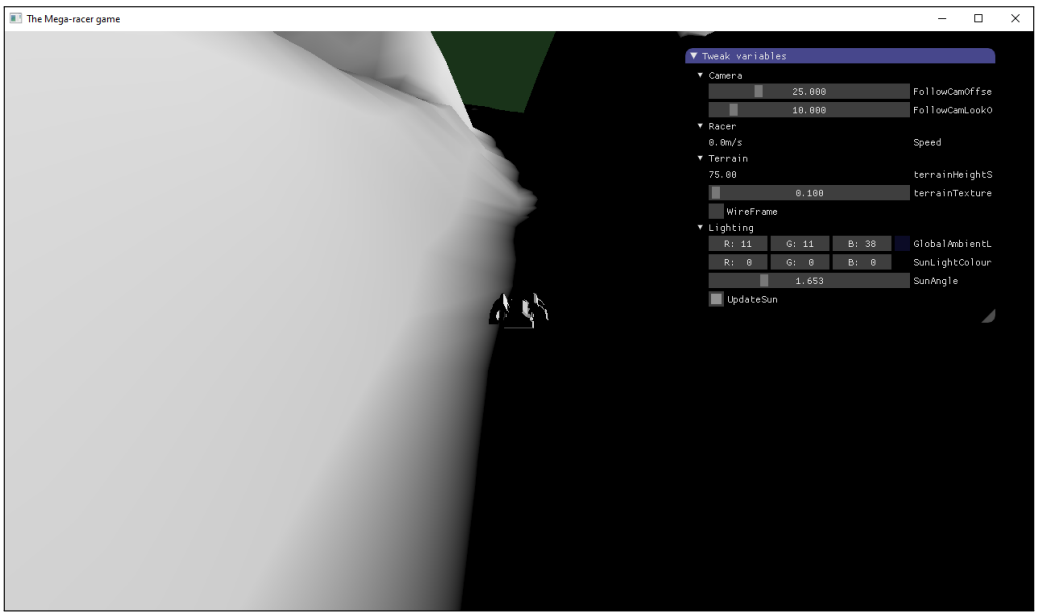

Figure 8: 1.5 - Step 2

(3) Modify the light that is emitted by the light source to have the correct colour and maximum intensity. The distribution of incoming light that is arriving at the surface needs to be handled so the correct colour and maximum intensity are shown.

```
# Lab 4 - Q2 - FragmentShader
vec3 incomingLight = incomingIntensity * lightColourAndIntensity;

# mega_racer.py - RenderingSystem
commonFragmentShaderCode =
    ...
    vec3 computeShading(...)
    {
        ...
        vec3 incomingLight = incomingIntensity * lightColour;
        return incomingLight;
    }
```

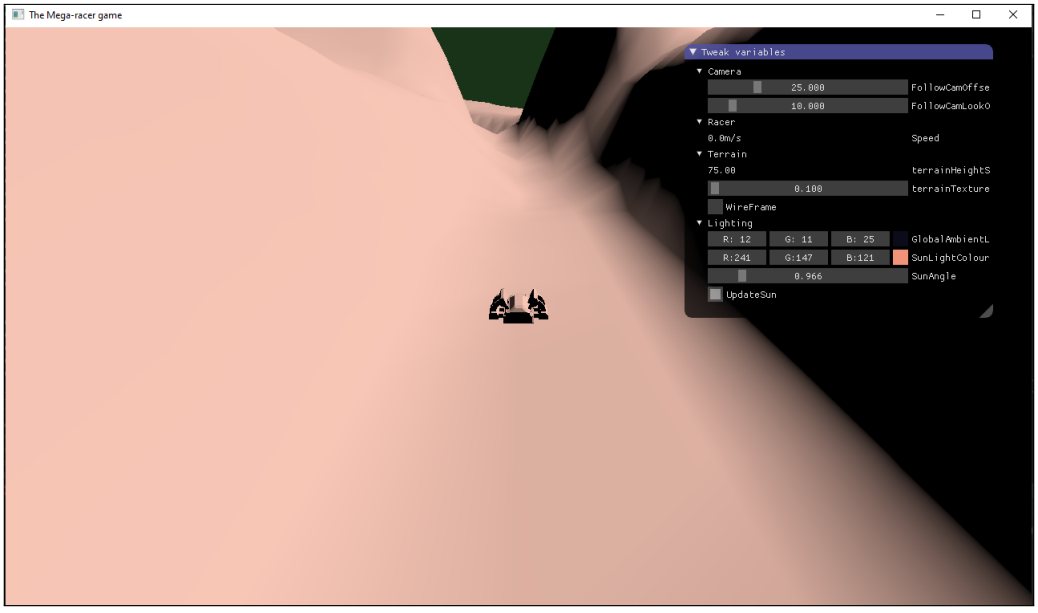

Figure 9: 1.5 - Step 3

(4) Diffuse Lambertian Reflection. The bidirectional reflection distribution (BRDF) is used widely to model how surfaces reflect light (Lecture 5). To diffuse the Lambertian reflection by multiplying the incoming light with a constant that represents the reflection of the material for the given spectrum. Following the same pattern as question 3 in lab 4, the argument passed as materialColour should provide this representation, as provided by the materialDiffuse variable passed in to computeShading() in setUpObModelShader().

```
# Lab 4 - FragmentShader
vec3 materialDiffuse = texture(diffuse_texture, v2f_texCoord).xyz * material_diffuse_color
vec3 outgoingLight = incomingLight * materialDiffuse;

# mega_racer.py - RenderingSystem
commonFragmentShaderCode =
    ...
    vec3 computeShading(...)
    {
        ...
        vec3 outgoingLight = incomingLight * materialColour;
        return outgoingLight;
    }
```

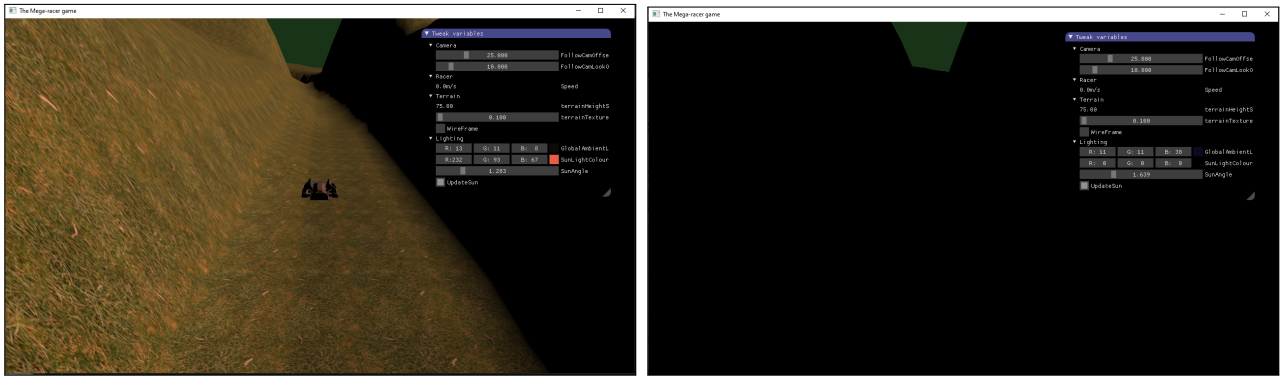

Figure 10: 1.5 - Step 4

(5) Take into account the indirect light. In order to handle indirect lighting, it is replaced by a constant that is known as ambient lighting. This step builds on step 3 and follows question 4 in lab 4. As mentioned in the notes, this is an approximation with a single colour value. The ambience is added to the incomingLight variable as there is the light comes from everywhere, and then multiplied by the BRDF as both are independent.

```
# Lab 4 - FragmentShader
vec3 outgoingLight = (incomingLight + ambientLightColourAndIntensity)
                        * materialDiffuse;

# mega_racer.py - RenderingSystem
commonFragmentShaderCode =
    ...
    vec3 computeShading(...)
    {
        ...
        vec3 outgoingLight = (incomingLight + globalAmbientLight)
                            * materialColour;
        return outgoingLight;
    }
```

This has produced the final results, light that doesn't cause pitch black and reflects according to the correct colour.

Figure 11: 1.5 - Step 5

Side notes: There are many other lighting models that could have been introduced; including a more advance ambient model or specular lighting.

Tier 2

The following deliverables were a higher difficulty and required more time, effort and research to complete. The tasks that were chosen were to improve terrain textures, add fog and add props. These tasks were all completed to a good standard, though improvements can always be made. Additionally, these were all tasks that could be carried forward to the Olympic Stadium model.

2.1 Improve terrain textures

Adding extra textures to the world not only offers visual appeal with more a more realistic environment but also provides functional ques to the user about the type of terrain they are driving on (rough or road). There are three additional textures to add in this section; high, steep and road. For each of the textures, the same process as the grassTexture was followed in terms of the following:

1. Declare a texUnit and a texID to introduce the texture.
2. Call bindTexture and setUniform to prepare it for the shader.
3. Assign the texture as the *materialColour* in the fragmentShader depending on height/slope logic.
4. Load the texture with the matching 2D image.

Both the highTexture and steepTexture are required to be mixed with the grassColour for a more realistic appearance. This function takes two vectors and weighting value between 0 and 1 (t), where the second vector argument is scalar multiplied by the t value and the first vector is multiplied by 1 minus the t value. Therefore more emphasis is placed on the second vector colour.

```
def mix(v0, v1, t):
    return v0 * (1.0 - t) + v1 * t
```

(1) Set the high texture The height texture should only be selected when the height in the world is above a certain threshold, as well as being mixed with the grass texture. The height

is given by the vertex shader, which is the z position of the positionIn or worldSpacePosition variable. The terrainHeightScale is equivalent to Terrain.heightScale which is 75.0 and used as a scale factor to calculate the z coordinates for each vertex.

```
# vertexShader:
v2f_height = positionIn.z
    -> v2f_worldSpacePosition = positionIn

# render
terrainHeightScale = self.heightScale
    -> heightScale = 75.0
```

The same steps as the grass texture are followed including declaring a unit, the texture variable, binding the texture, setting the uniform value and loading the texture. However, sampling the texture is slightly different as instead of overriding the materialColour with the highColour, the highColour is assigned in the same way at the given height and then the materialColour is assigned as the output of blending this highColour with the grassColour using mix().

```
    # terrain.py
    TU_High = 1
    highTexture = None

    def render():
        ....
        lu.bindTexture(self.TU_High, self.highTexture)
        lu.setUniform(self.shader, "highTexture", self.TU_High)

    def load():
        fragmentShader =
            ...
            uniform sampler2D highTexture;

            void main():
            {    ....
                OR if (v2f_height > 50) {
                if ((v2f_height/terrainHeightScale) > 0.9) {
                    vec3 highColour = texture(highTexture,
                                              v2f_worldSpacePosition.xy
                                              * terrainTextureXyScale).xyz;
                    materialColour = mix(materialColour,
                                              highColour,
                                              (v2f_height/terrainHeightScale));
            }

        self.highTexture = ObjModel.loadTexture("rock_2.png", "data", True)
```

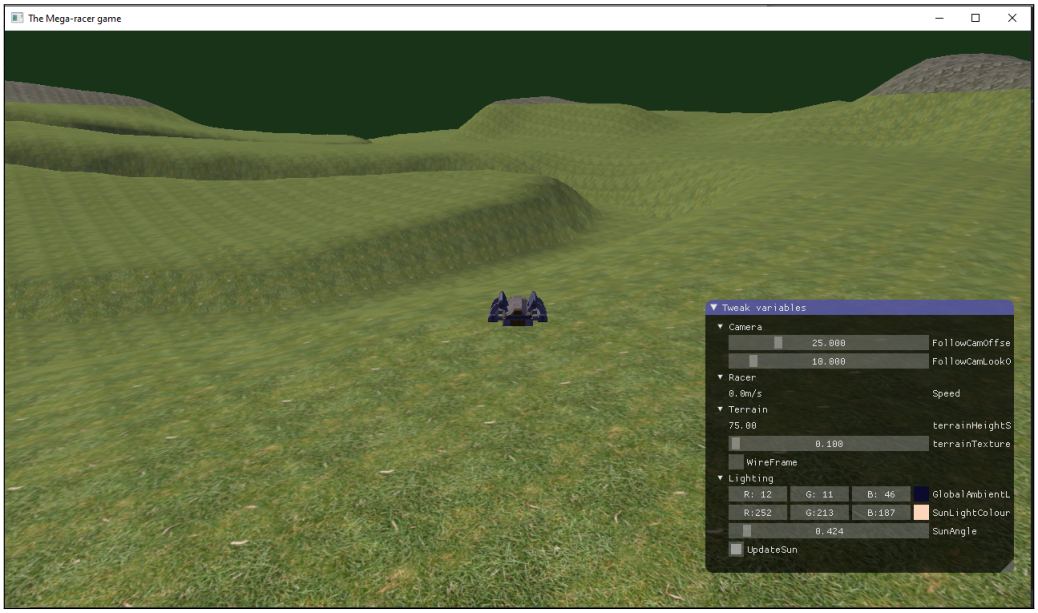

Figure 12: 2.1 - High Texture

(2) Set the steep texture The steep texture relies on only be loaded when the slope meets a certain threshold. To get the slope, first the coordinates of the normalised world space need to be available. The viewSpacePosition, viewSpaceNormal and worldSpacePosition had already been provided, and so the same pattern was followed to have access to the worldSpaceNormal.

```
vertexShader =
    out VertexData
    {
        vec3 v2f_viewSpacePosition;
        vec3 v2f_viewSpaceNormal;
        vec3 v2f_worldSpacePosition;
        vec3 v2f_worldSpaceNormal; // NEW
    }
    void main()
    {
        v2f_viewSpacePosition = (modelToViewTransform * vec4(positionIn, 1.0)).xyz;
        v2f_viewSpaceNormal = modelToViewNormalTransform * normalIn;
        v2f_worldSpacePosition = positionIn;
        v2f_worldSpaceNormal = normalIn; // NEW
    }
fragmentShader =
    in VertexData
    {
        vec3 v2f_viewSpacePosition;
        vec3 v2f_viewSpaceNormal;
        vec3 v2f_worldSpacePosition;
        vec3 v2f_worldSpaceNormal; // NEW
    }
```

Now that the normalised world space is available to use, the slope needs to be calculated and compared to the threshold to determine when to blend the textures. Again, the same steps are followed to map the texture, however the slope is calculated and then the logic of the threshold applied.

```
# terrain.py
TU_Steep = 2
steepTexture = None
```

```
def render():
    ....
    lu.bindTexture(self.TU_Steep, self.steepTexture)
    lu.setUniform(self.shader, "steepTexture", self.TU_Steep)

def load():
    fragmentShader = """
        ...
        uniform sampler2D steepTexture;

        void main()
        {....
            float slope = dot(v2f_worldSpaceNormal, vec3(v2f_worldSpaceNormal.x, 0.0,
            if (slope < 0.8) {
                vec3 steepColour = texture(steepTexture, v2f_worldSpacePosition.xy * t
                materialColour = mix(materialColour, steepColour, (v2f_height/terrainH
        }

    self.steepTexture = ObjModel.loadTexture("rock 5.png", "data", True)
```

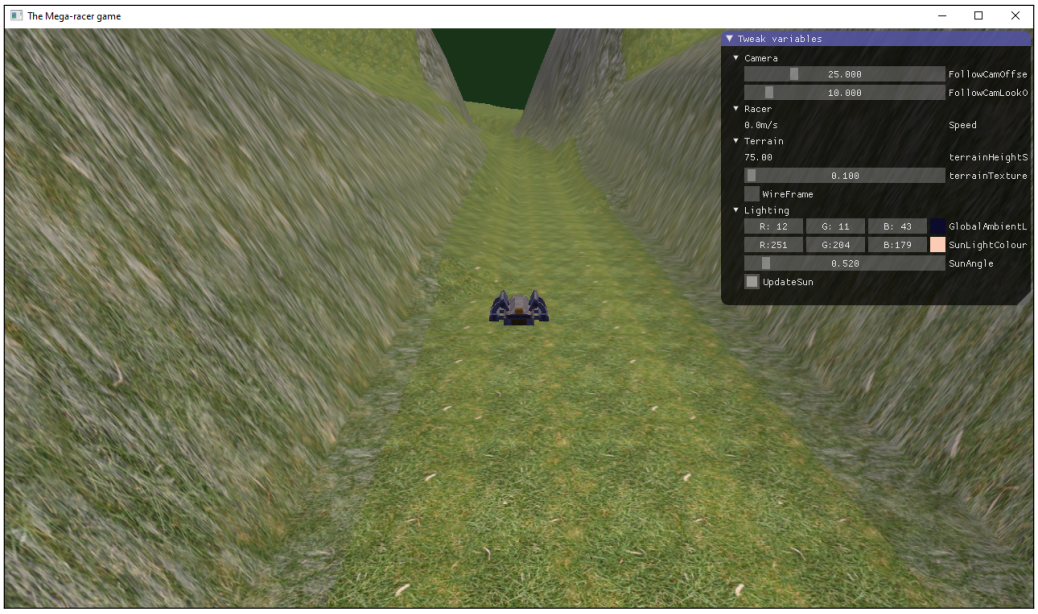

Figure 13: 2.1 - Steep Texture

(3) Add the road To load a texture for the road, since the game logic is not available to the fragment shader, a mapTexture needed to be created which could identify the *blueChannel* of the map for the shader. As previously mentioned, the blue pixel on the map is used to represent the road and subsequently the logic of the racer's speed. Since the track image is not a 2D file, when calling ObjModel.loadTexture the SRGB parameter is passed is False. As well as creating this map texture, a road texture also needs to be created, following the same steps as the grass texture. However, the logic in the fragmentShader was quite different and will be explained separately.

```
# terrain.py
TU_Road = 3
TU_Map = 4
roadTexture = None
mapTexture = None

def render():
    ....
    lu.bindTexture(self.TU_Road, self.roadTexture)
```

```
        lu.bindTexture(self.TU_Map, self.roadMap)
        lu.setUniform(self.shader, "roadTexture", self.TU_Road)
        lu.setUniform(self.shader, "mapTexture", self.TU_Map)

    def load():
        fragmentShader =
            ...
            uniform sampler2D roadTexture;
            uniform sampler2D mapTexture;

            void main():
            {   ....
                ????
            }
            }

        self.roadTexture = ObjModel.loadTexture("paving_5.png", "data", True)
        self.mapTexture = ObjModel.loadTexture("track_01_128.png", "data", False)
```

For the high and steep textures, the shader was relying simply on the height and slope, but now to for the shader to know what is considered 'road' the blueChannel needs to be calculated using the mapTexture. To be able to sample this texture the first step was to get the normalised texture coordinates. This was possible by following the similar steps to the worldSpaceNormal but with the xyNormScale and xyOffset. These were the last two declared variables in the vertex shader that hadn't been carried through to the fragment shader despite being declared as uniform in render() at the same time as the Texture XyScale and HeightScale. It was obvious that the xyNormScale variable would be used instead of terrainTextureXyScale, but the purpose of the offset was unclear at this time.

```
# render()
xyNormScale = 1.0 / (vec2(self.imageWidth, self.imageHeight) * self.xyScale);
lu.setUniform(self.shader, "xyNormScale", xyNormScale);
xyOffset = -(vec2(self.imageWidth, self.imageHeight) + vec2(1.0)) * self.xyScale / 2.0
lu.setUniform(self.shader, "xyOffset", xyOffset);
-->

# load()
vertexShader =
    out VertexData
    {
        vec2 v2f_xyNormScale;
        vec2 v2f_xyOffset;
    }
    void main()
    {
        v2f_xyNormScale = xyNormScale;
        v2f_xyOffset = xyOffset;
    }
fragmentShader =
    in VertexData
    {
        vec2 v2f_xyNormScale;
        vec2 v2f_xyOffset;
    }
```

When working on this section, it was often difficult to determine the problem, as unless the texture was declared, bound and loaded properly in the shader nothing would display. The below code is the first iteration that provided 'working' shading. The blueChannel variable

followed that of the other textures, though instead of multiplying the worldSpacePosition.xy by the terrainTextureXyScale it was multiplied by the xyNormScale variable and only the z coordinate needed to be stored. The threshold of 0.9 was chosen as this produced the most accurate result.

```
# terrain.py
def load():
    fragmentShader =
        void main():
        {    ....
        float blueChannel = texture(mapTexture, (v2f_worldSpacePosition.xy) * v2f_xyNo
        if (blueChannel >= 0.9) {
            vec3 roadColour = texture(roadTexture, v2f_worldSpacePosition.xy * terrainT
            materialColour =  mix(materialColour, roadColour, (v2f_height/terrainHeigh
            }
        }
```

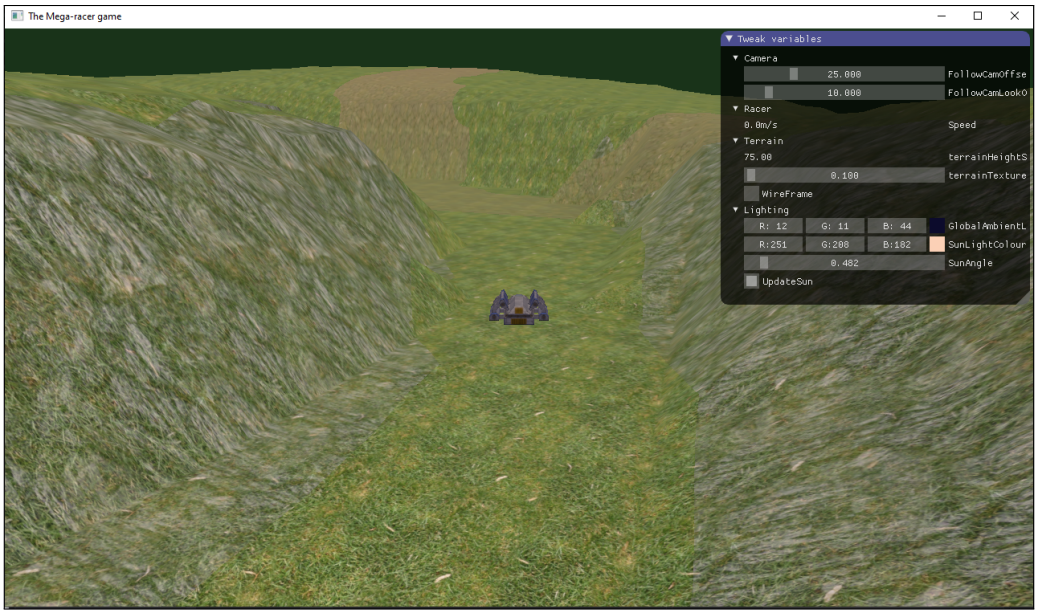

Figure 14: 2.1 - Road Texture (Mixed & No Offset)

There are clearly still some issues to resolve.

1. The grass texture was overtaking the blend of the pavement. This is because the roadColour was mixed with the grassColour as previously done. In the next iteration, the same as grassColour was followed such that only the roadColour would be declared as the materialColour when in the blueChannel.
2. The road was not in the position it needed to be and was instead on the hill. This was another simple fixing by subtracting the xyOffset variable from the worldSpacePosition. After discovering this fixed the coordinates of the road, the mixing of road and grass was trialled again but the same issue occurred.

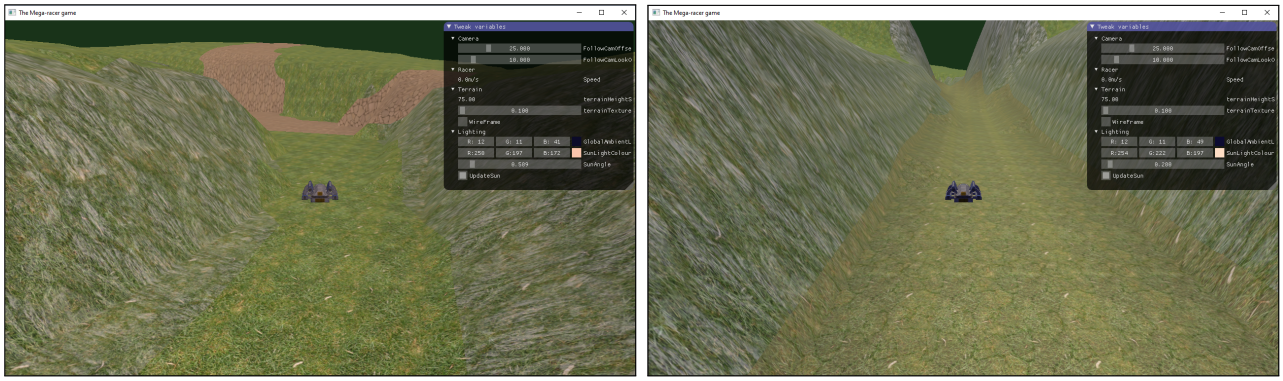

Figure 15: 2.1 - Road Texture (Not Mixed & No offset / Mixed & Offset)

After making these changes, the paving was successfully sampled and loading correctly onto the correct coordinates for the road according to the blue channel of the track.

```
# terrain.py
float blueChannel = texture(mapTexture, (v2f_worldSpacePosition.xy - v2f_xyOffset) * v
if (blueChannel >= 0.9) {
    vec3 roadColour = texture(roadTexture, v2f_worldSpacePosition.xy * terrainTextureX
        materialColour = roadColour;
}
```

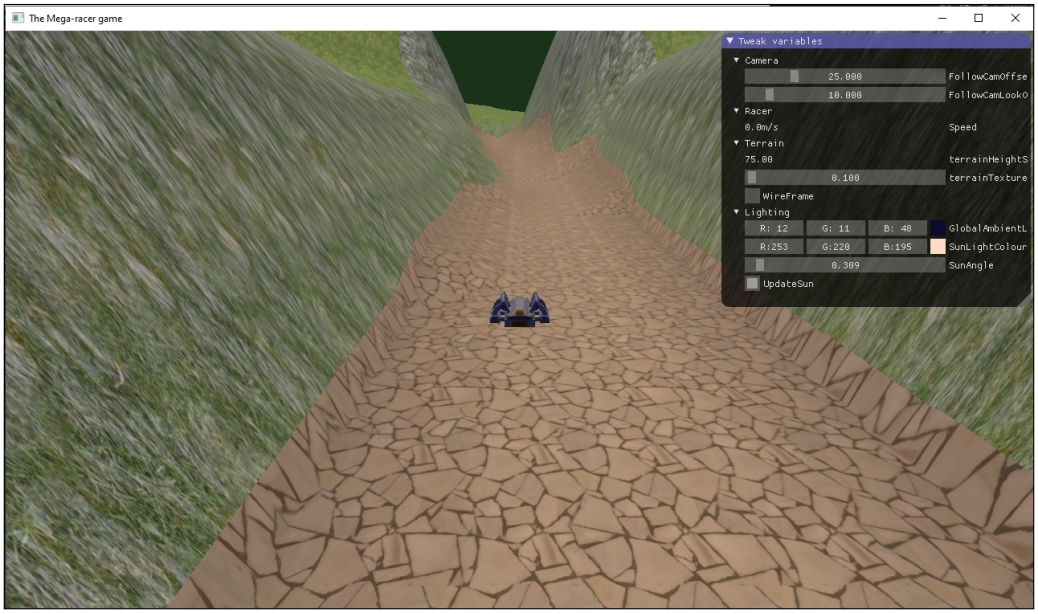

Figure 16: 2.1 - Road Texture

2.2 Add Fog

Adding fog to the world was an attempt to provide the opportunity to add realism, depth, scaling of objects, give distance cues and more lighting effects to the world. The provided tutorial in the project notes was followed for this step to implement basic and then height-based fog.

(1) Colour based fog) The first step of the tutorial was to implement colour to the fog, though it was referred to as scattering. This aimed to provide some context as to the strength of the sun by making the colour dependent on orientation. The b variable refers to the density and is set to 0.005 because ..... This function was declared in commonFragmentShaderCode

in mega_racer.py, then called as the output to fragmentColour in the fragmentShader. The relevant variables for the parameters are as follows:

- rgb = reflectedLight: This is currently the argument passed into toSrgb as for the 'color' parameter, so it will perform the same purpose as the 'rgb' parameter for applyFog().
- distance = -v2f_viewSpacePosition.z
  v2f_viewSpacePosition = (modelToViewTransform * vec4(positionAttribute, 1.0)).xyz;

```
# Tutorial
vec3 applyFog( in vec3  rgb, in float distance )
{
    float fogAmount = 1.0 - exp( -distance*b );
    vec3  fogColor  = vec3(0.5,0.6,0.7);
    return mix( rgb, fogColor, fogAmount );
}
-->
# mega_racer.py - Rendering system
commonFragmentShaderCode =
    ...
    vec3 applyFog(in vec3 rgb, in float distance)
    {
        float b = 0.005;
        float fogAmount = 1.0 - exp(-distance*b);
        vec3  fogColor  = vec3(0.5,0.6,0.7);
        return mix(rgb, fogColor, fogAmount);
    }
#terrain.py
def load(...):
    ...
    fragmentShader =
        ...
        void main()
        {
            ...
            fragmentColor = vec4(toSrgb(applyFog(reflectedLight,
                                                 -v2f_viewSpacePosition.z)),
                                 1.0);
        }
```

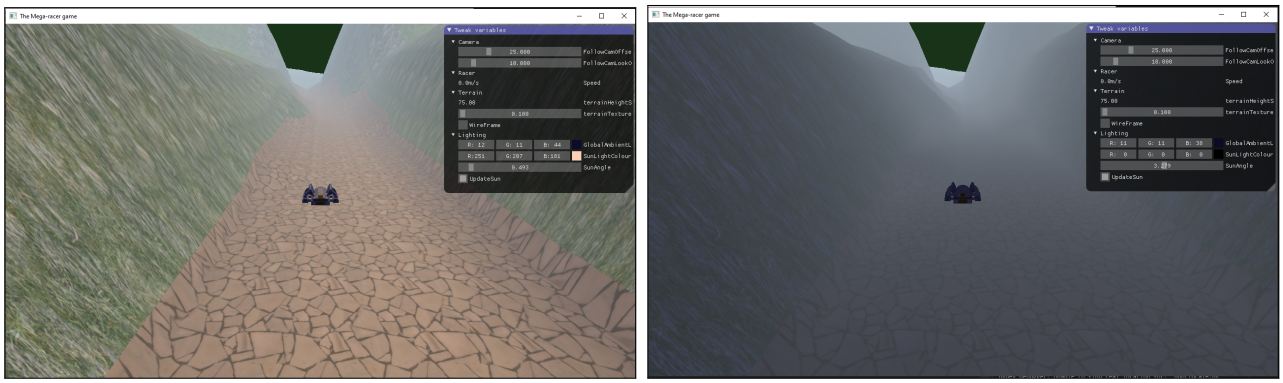

Figure 17: 2.2 - Basic

(2) Make colour sunLight and Ambient: As evident in the screenshots, this implementation of fogColour is not correct, especially when there is no direct sunlight. Following the advice of the project notes, a combination of the sunLightColour and ambientColour were trialled.

First, the these two values were added together.

```
vec3  fogColor  = (sunLightColour + globalAmbientLight)
```

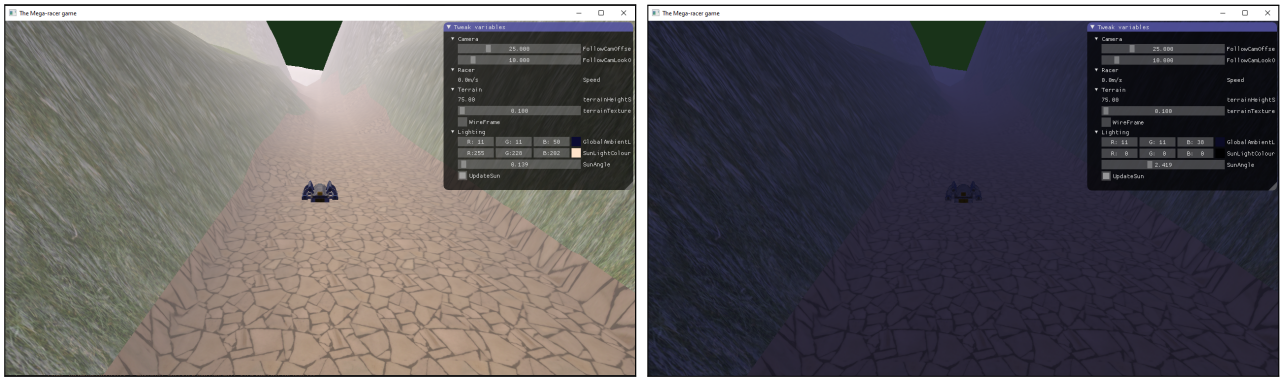

Figure 18: 2.2 - Fog Colour (Total)

Then an average of the two values was tested, which produced a less grey result and was the best combination at this stage.

```
vec3 fogColor = (sunLightColour + globalAmbientLight) / 2.0
```

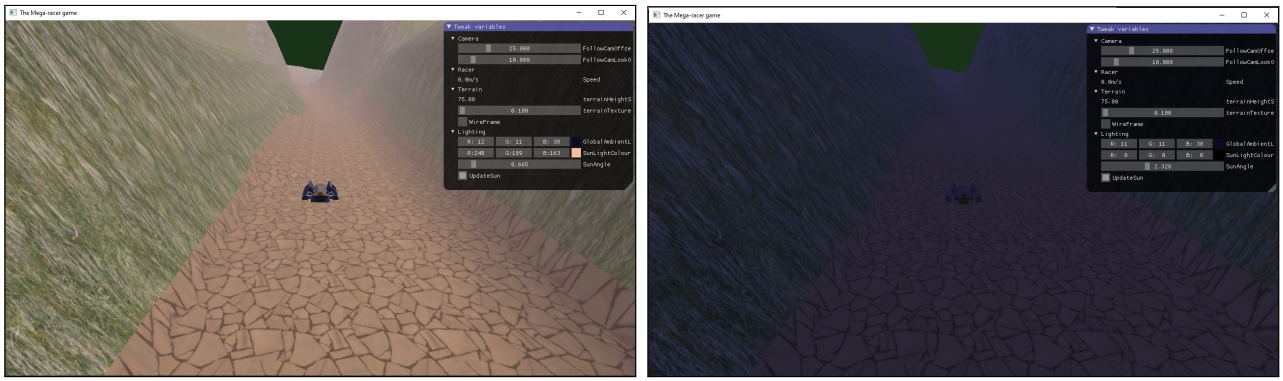

Figure 19: 2.2 - Fog Colour (Average)

(3) Height based To take the fog one step further, the height based fog implementation from the provided tutorial was implemented. Now as well as creating a fog appearance that doesn't have a constant colour, the fog will also not have a constant density. This essentially means the intensity of the fog colour changes with distance and height to create a more interactive fog appearance so areas further away have more fog intensity. The *applyFog()* function will be updated with this implementation and same as before, the call to the function also needs to be updated in fragmentShader. There are a few more variables needed as well:

- cameraPosition = vec3(worldToViewTransform[3][0],worldToViewTransform[3][1],worldToViewTrans
- cameraToPointVector = normalize(positionIn - cameraPosition);

The c variable is equal to a/b....

```
# Tutorial
vec3 applyFog( in vec3  rgb,      // original color of the pixel
               in float distance, // camera to point distance
               in vec3  rayOri,   // camera position
               in vec3  rayDir )  // camera to point vector
{
    float fogAmount = c * exp(-rayOri.y*b) * (1.0-exp( -distance*rayDir.y*b ))/ray
    vec3  fogColor  = vec3(0.5,0.6,0.7);
    return mix( rgb, fogColor, fogAmount );
}
-->
# mega_racer.py - Rendering system
```

```
    commonFragmentShaderCode =
        ...
        vec3 applyFog(in vec3 rgb, in float distance, in vec3  rayOri, in vec3 rayDir)
        {
            float b = 0.005;
            float c = 0.66;
            float fogAmount = c * exp(-rayOri.y*b) * (1.0-exp( -distance*rayDir.y*b ))
            vec3  fogColor  = (sunLightColour + globalAmbientLight) / 2.0;
            return mix( rgb, fogColor, fogAmount );
        }
# terrain.python
def load(...):
    ...
    vertexShader =
        ...
        uniform mat4 worldToViewTransform;

        out VertexData
        {
            ...
            vec3 cameraPosition;
            vec3 cameraToPointVector;
        }
        void main()
        {
            ...
            cameraPosition = vec3(worldToViewTransform[3][0],worldToViewTransform[3][1
            cameraToPointVector = normalize(positionIn - cameraPosition);
            ...
        }
    fragmentShader =
        in VertexData
        {
            ...
            vec3 cameraPosition;
            vec3 cameraToPointVector;
        }
        void main()
        {
            ...
            fragmentColor = vec4(toSrgb(applyFog(reflectedLight, -v2f_viewSpacePositio
        }
```

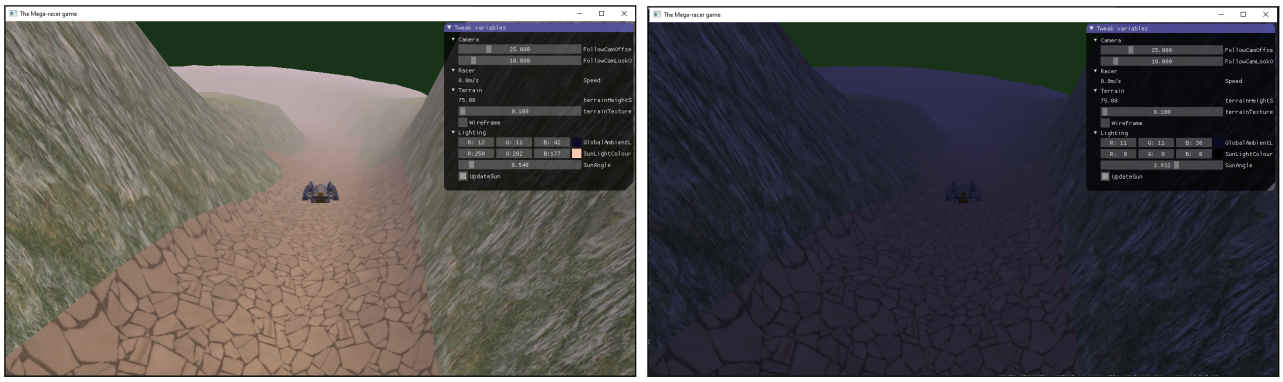

Figure 20: 2.2 - Height Fog

2.3 Props

Introducing other objects into the world, such as trees and rocks, creates a more realistic and fun environment. The logic related to generating the locations of trees and rocks have already been handed in terrain.py by load(). This is dependent on the value of the green pixel, the coordinate from the map is saved to the relevant prop list. Several object model files have also been provided in the data folder.

(1) Create the Prop class A new prop.py file was created and a Prop class declared. The props are an object just like the racer, so to get a foundation, the Racer class from racer.py was referenced heavily. To start with all of the same libraries were imported and all of the variables copied over. From here any variables that were not required were removed (velocity, speed, maxSpeedRoad, maxSpeedRough, terrain) and new variables added (rotation). Next the render() and load() functions from the Racer class were copied (update() is not required since once created the elements of the props don't change).

```
# prop.py
import libraries....

class Prop:
    position = vec3(0,0,0)
    heading = vec3(1,0,0)
    rotation = 0.0

    zOffset = 3.0
    angvel = 2.0

    model = None
```

The load() function stills needs to assign the variables of each prop instance as it did in the racer, however a few things of changed. First of all the terrain variable is no longer needed. Secondly, the position is randomised from a given list based on the prop type, as well as the rotation. This is achieved by importing the 'random' library and using choice() which randomly selects an entry from a list (https://pynative.com/python-random-choice/). Finally, the model of each unique prop should only be loaded once. The model for each prop type will be created in a prop manager class and then passed in to each instance of that prop type.

```
#prop.py - Prop
import random

def load(self, model, locations):
    self.position = random.choice(locations)
    self.rotation =random.choice(range(0,360))
    self.model = model
```

The render() function needed to follow the same steps from Racer.render(), as well as apply the random rotation of props in the world. The only difference is that when calling drawObjModel() the modelToWorldTransform parameter is multiplied by a matrix construction function (defined in lab_utils.py) based on the random rotation as the angle argument.

```
#prop.py - Prop
def render(self, view, renderingSystem):
    modelToWorldTransform = lu.make_mat4_from_zAxis(self.position, self.heading, [ 0.0, 0.
    rotationMatrix = lu.make_rotation_y(self.rotation)
    renderingSystem.drawObjModel(self.model, modelToWorldTransform * rotationMatrix, view)
```

(2) Create the PropManager class Now, that the Prop class has been created the next step was to create a class to manage all of the prop instances; PropManager. To start the required variables were declared. For each prop type there needed to be a maximum number of instances, a list to keep track of the instances, a single model.

```
# prop.py
class PropManager:
    treeMax = 50
    treeList = []
    treeModel = None

    rockMax = 20
    rockList = []
    rockModel = None
```

Now to create the instances of each prop type and save them in a global list (treeList or rockList). The process is the same for all prop types so a general function, loadPropList(), creates the instances for the prop type and fills the list. For each prop type, there are max number of instances (propMax). Each of these instances uses a single model of the prop type (propModel) with a position from the designated list for that prop type (propLocations). The instance is then appended to the relevant list for easier access (propList).

```
# prop.py - PropManager
def loadPropList(self, propModel, propMax, propLocations):
propList = []
i = 0
while i < propMax:
    prop = Prop()
    prop.load(propModel, propLocations)
    propList.append(prop)
    i += 1
    print(prop.position)
return propList
```

This function is called for each prop type, with their specific values, and assigned to the relevant prop type list in loadAllProps(), after first creating the unique model for each prop type in loadAllProps(). This means the model for each prop is created only once and all instances share the same model (i.e. textures, vector data, etc).

```
# prop.py - PropManager
def loadAllProps(self, terrain):
    # Load trees
    self.treeModel = ObjModel("data/trees/birch_01_d.obj")
    self.treeList = self.loadPropList(self.treeModel, self.treeMax, terrain.treeLocations)
    # Load rocks
    self.rockModel = ObjModel("data/rocks/rock_01.obj")
    self.rockList = self.loadPropList(self.rockModel, self.rockMax, terrain.rockLocations)
```

As well as loading the props, the props will need to have a function call that renders them as well in the same way as other object models in that each prop instance is rendered individually using render().

```
    # prop.py - PropManager
    def renderAllProps(self, view, renderingSystem):
        for prop in self.allProps:
            prop.render(view, renderingSystem)
```

(3) Place in the world With the PropManager defined the props can now be loaded in the world. The PropManager class needs to be imported and the variable created. Then, below were the g_racer and g_terrain are being assigned as instances of their corresponding class' and loaded, the same needs to be done for the props as g_props.Additionally, all of the props also needed to be rendered, again in the same area as the g_racer and g_terrain variables are rendered.

```
#mega_racer.py
from prop import PropManager
...
g_props = None
....
g_props = PropManager()
g_props.loadAllProps(g_terrain)
...
def renderFrame(...):
        ...
        g_props.renderAllProps(view, g_renderingSystem)
```

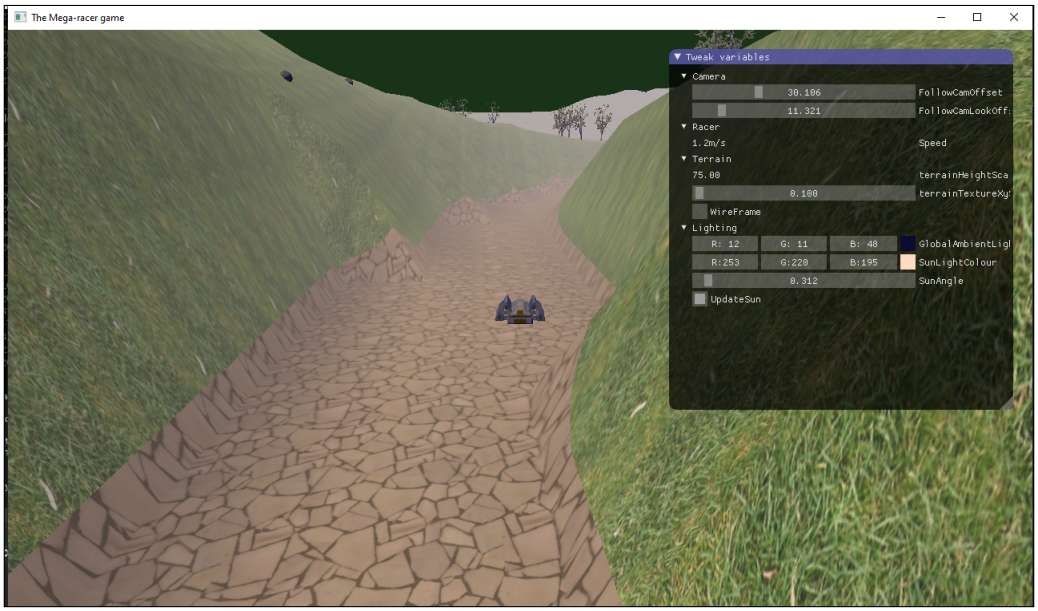

Figure 21: 2.3 - Props (trees and rocks)

Olympic Stadium

The most important area at an Olympics is the main Olympic Stadium where the athletic track is located. With the foundation of the mega racer implemented, this section was an opportunity to test my knowledge in a new environment, learn about graphic modelling software and try some advanced computer graphic techniques.

Modifications: The following are the changes, in order, that were made to the Mega Racer game in order to create an Olympic Stadium.

1. Car Model: Race Car --→ Athlete running
2. Props: Trees, Rocks --→ Cones, Rings, Podium
3. Terrain Textures: Grass, Paving, Rock Steep, Rock High --→ Grass, Synthetic track, Concrete, Seats

4. Track: Racing track in wilderness --→ Olympic Stadium with athletic track

Setup: To get started these steps were first taken:

1. Create a duplicate copy of mega_racer.py
2. Rename mega_racer.py to olympic_stadium.py
3. Change the window title to "The Olympic Stadium Tour"

```
window = glfw.create_window(g_startWidth, g_startHeight, "The Olympic Stadium Tour",
```

Blender: In the process of changing elements in the mega_racer world to reflect an Olympic stadium all of the object models (.obj and .mtl) needed to be either created or modified in blender. This was either because an appropriate free .obj file could not be found or was scaled incorrectly, and/or the .mtl file was missing or not correct. Some basics that needed to be learned to get started was scaling (tutorial) and rotating(tutorial).

Athlete

Instead of a race car driving around the track, I wanted to replace it with an athlete. A person is quite a complicated model to make from scratch so a premade .obj model of a basic person was sourced. The chosen model was this free basic mesh male model uploaded by Paul Chen on free3d.com. The model is in a basic standing position with no texturing. For a more realistic model for the stadium, the model needed to be scaled appropriately and posed in a running position. Importing the .obj file in blender and following this basic rigging tutorial on YouTube by PIXXO 3D, a meta-rig was set up for the model. This basically entails lining up a bone framework, like a skeleton, to the model. One of the errors that occurred was was moving one of the spine bones on accident so that it was detached disallowing the meta-rig from generating, however it was solved from this post through the blender forum.

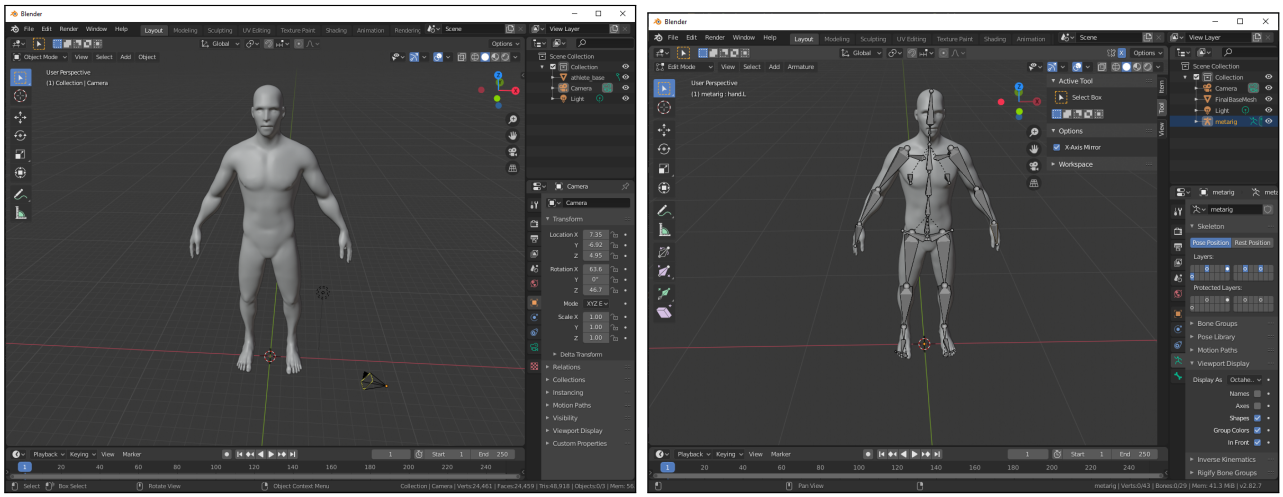

Figure 22: Racer - Blender - Base & Rig

Using this meta-rig, blender has a built-in feature that creates a rig that can be used to move the 'bones' around to place the model into poses. For the model, Ine leg needed to be lifted backward from the knee and point the foot down, and slightly lift the other foot and bend the knee slightly forward. The arms were moved in the same fashion with one arm back and the hand pointing down and the other arm forward with bend elbow and hand pointing up. Essentially, this put the model in a running pose.

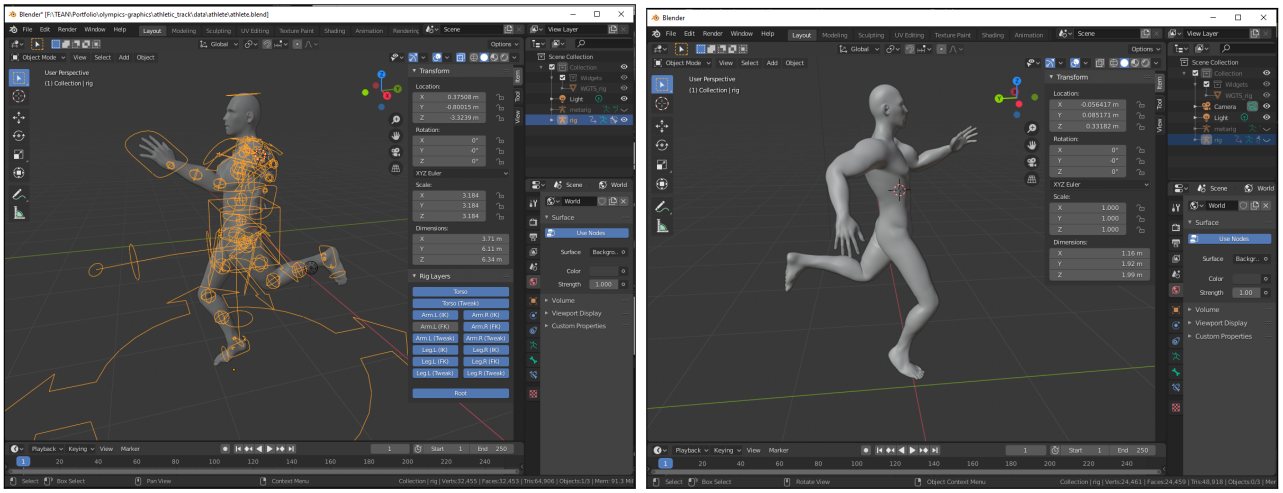

Figure 23: Racer - Blender - Posing

Now that the athlete was in the running position, some texturing needed to be added. Sourcing images from Google, some textures were found that were appropriate for an athlete including Lycra and breathable material. However, since the model was set as all one material this tutorial on YouTube from Jayanam, made aware that vertex groups would be needed to be used in order to separate them into different sections. In this step, I would have liked to have learn to extrapolate the areas to give them definition and shape the areas to look as they should i.e. make the feet look like shoes. For this basic model, after creating the groups they were able to be coloured separately by assigning the different materials (sourced image textures) to the groups as done by this tutorial from Weisbrod Imaging on YouTube.

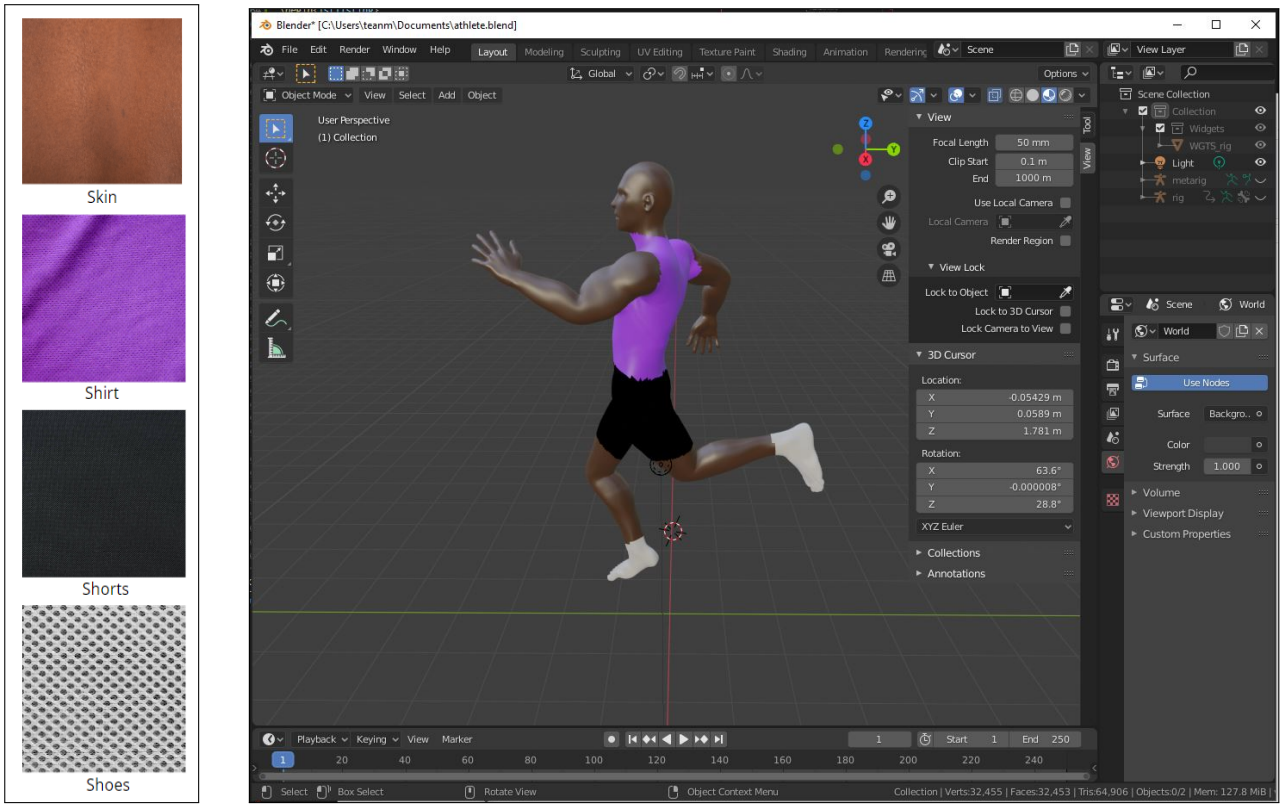

Figure 24: Racer - Blender - Colour

The athlete was coming together. However, I really didn't like the dodgy colouring attempt so I decided to try and add clothing for more realistic texturing. The following free clothing models were found on turbosquid.com:

- **Men Avatar Vest** by swatishr13: This model was imported with no colouring associated. The trims on the sleeve were set as white and the main colour used the same 'top' texture as in the previous colouring.
- **short pants** by sazandra: This file was able to be imported with all textures correctly. Since the model was in a running position, the pants were the wrong shape however the model used a mirror modify so the model was able to be split and just have half of the pants. This half of the pants were used as is and then duplicate was made with a mirror image of the local x and y coordinates that could be set at the right shape for the runner.
- **shoe** by Bruno Dalla: This model is of a single show so a duplicate mirrored on the local x was created. The associated texturing for this model didn't load correctly so the above 'shoe' texturing was used except for a trim around the bottom the same colour as the top.

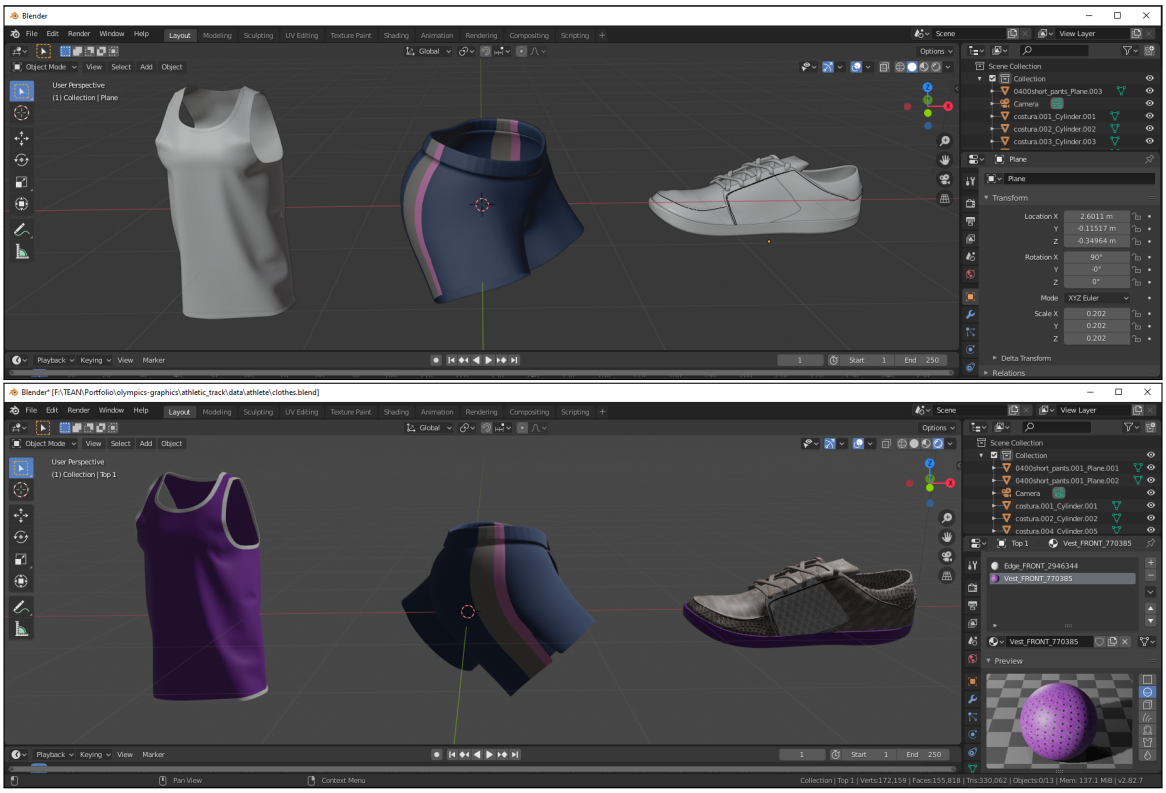

Figure 25: Racer - Blender - Clothes

The human model is now complete with clothes and posed. The next step would be to add skin texturing and more human features. This was a level of detail that was beyond my ability for this project. Perhaps another time.

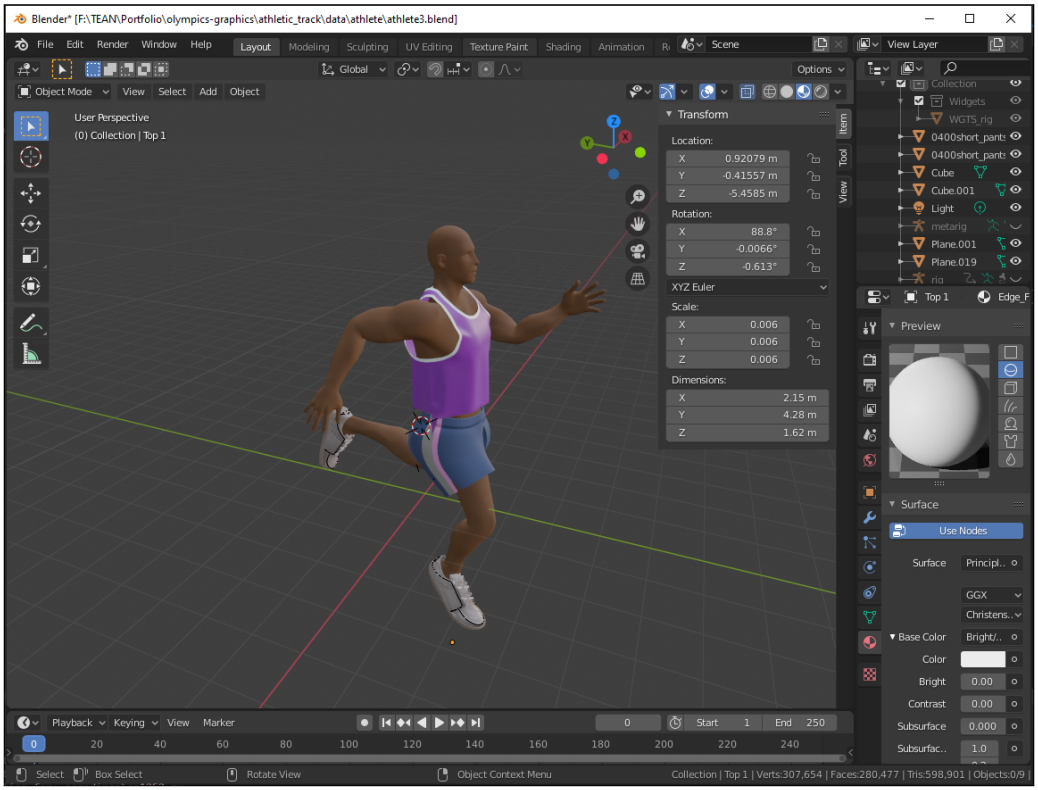

Figure 26: Racer - Blender - Final

Finally, the athlete was placed in the world to scale. The name of the athlete was kept as g_racer/racer throughout code as this is appropriate for the context but instead of the racer_01.model the new athlete.obj file imported.

```
#mega_racer.py
g_racer.load("data/athlete/athlete.obj", g_terrain, g_renderingSystem);
```

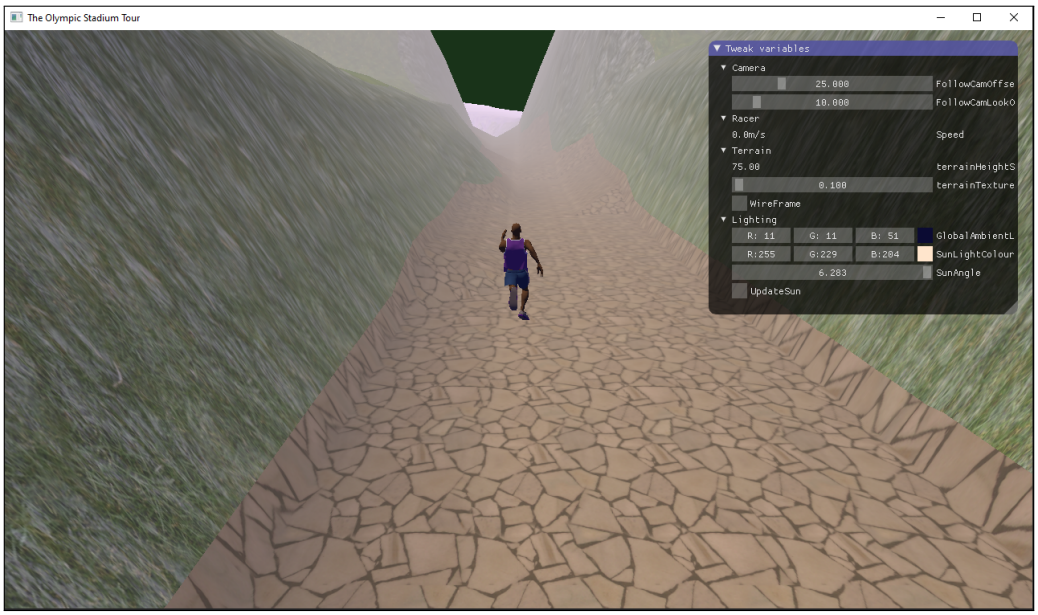

Figure 27: Athlete

Props

Since an Olympic Stadium is essentially a building, trees and rocks were note appropriate models to keep in the world. With some blender experience now obtained, the props for this implementation were made from scratch with the assistance of tutorials. Originally, these models were going to be sourced from online libraries such as turbosquid.com, free3d.com, cgtrader.com and opengameart.org, however after searching the models were not complete, inadequate or difficult to find. Making these models was an interesting experience as it gave me the experience of the quality and detail of texturing, shaping, scaling, shading and lighting of 3D graphic models. As well as as better understanding of how the .obj files were handled, manipulated and created, along with their accompanying .mtl file. The props that were created were Olympic Rings (to replace rocks) and safety cones (to replace trees).

Additionally, the code for this section was updated as originally more models were planned to be included. This code is very similar to mega racer though handles more types of props. Other props that were considered were a podium, hurdles, Olympic flame and crowd members.

```
#mega_racer.py
propTypes = [['cone', 50, g_terrain.treeLocations], ["rings", 2, g_terrain.rockLocatio
g_props.loadAllProps(propTypes)

#prop.py
class PropManager:
    propTypes = []
    allProps = []

    def loadProp(self, propType):
        propModel = ObjModel("data/{propName}/{propName}.obj".format(propName=propType
        i = 0
        while i < propType[1]:
            prop = Prop()
            prop.load(propModel, propType[2])
            i += 1
            self.allProps.append(prop)

    def loadAllProps(self, propTypes):
        self.propTypes = propTypes
        for prop in self.propTypes:
            self.loadProp(prop)

    def renderAllProps(self, view, renderingSystem):
        for prop in self.allProps:
            prop.render(view, renderingSystem)
```

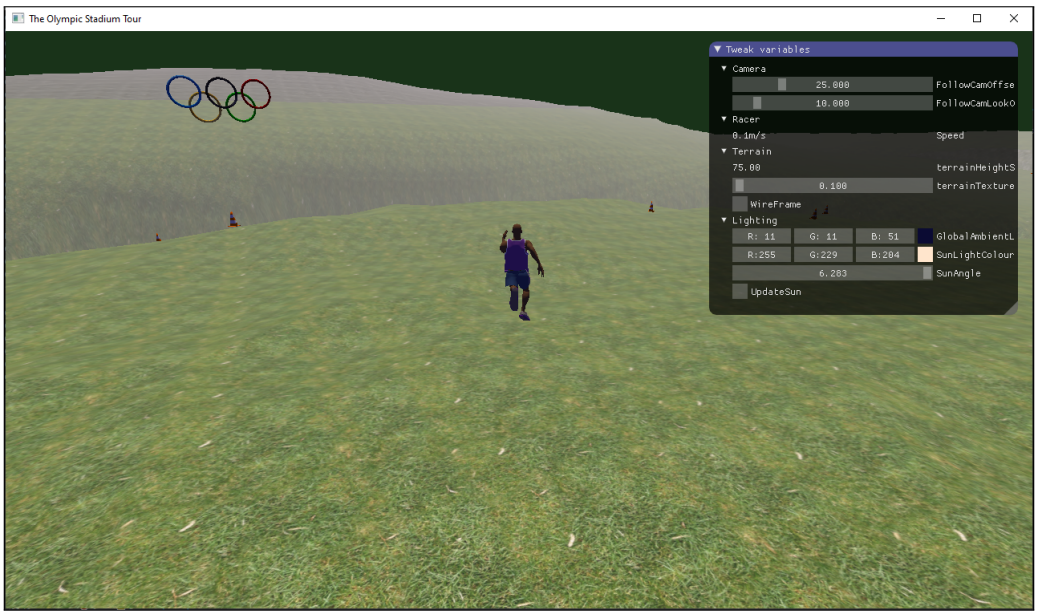

Figure 28: Athlete

Rings

It wouldn't be the Olympics without the iconic Olympic rings. Originally this model on cgtrader.com was sourced, however after trying to open it in blender not only were the set materials not working from the .mtl file but also the rings were connected and overlapping so I wasn't able to separate them out to colour them differently. They seemed simple enough to make myself so following this time-lapse video and sourcing my own textures for the rings to make a new model that could be used on the centre area of the stadium. Each ring is a torus that was reshaped slightly, coloured with the textures and then overlapped with the two bottom rings in front.

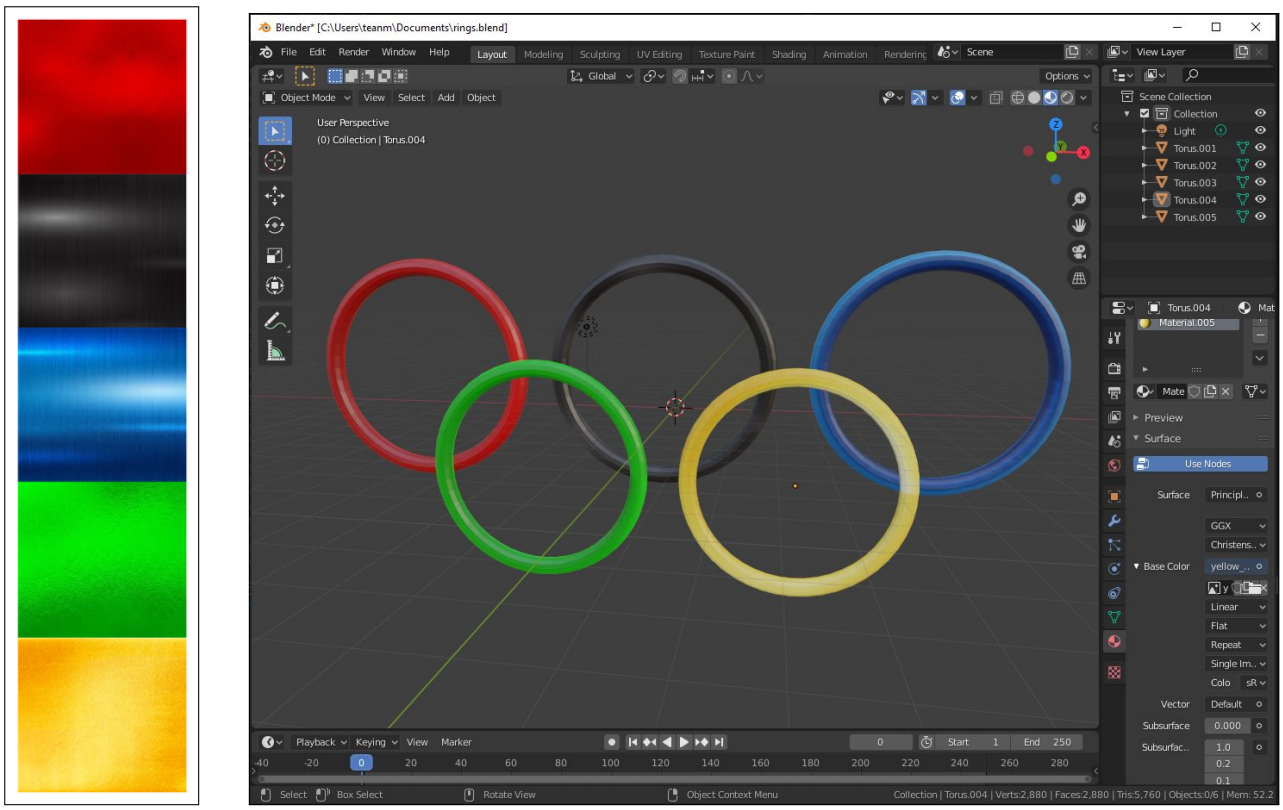

Figure 29: Props - Rings

Cones

To demonstrate props that would be repeated on a large scale, safety cones were placed around the inside of the middle area of the stadium. Similar to the Olympic rings, various models on cgtrader.com were found but the .mtl file was not correct for any of them, so I decided to create my own cones. Following this tutorial a realistic traffic safety cone was created. The cone started as a cylinder, the top was reshaped for the cone appearance, the lip on the top was created, a base added, the strips added using loop cuts and finally the colouring applied.

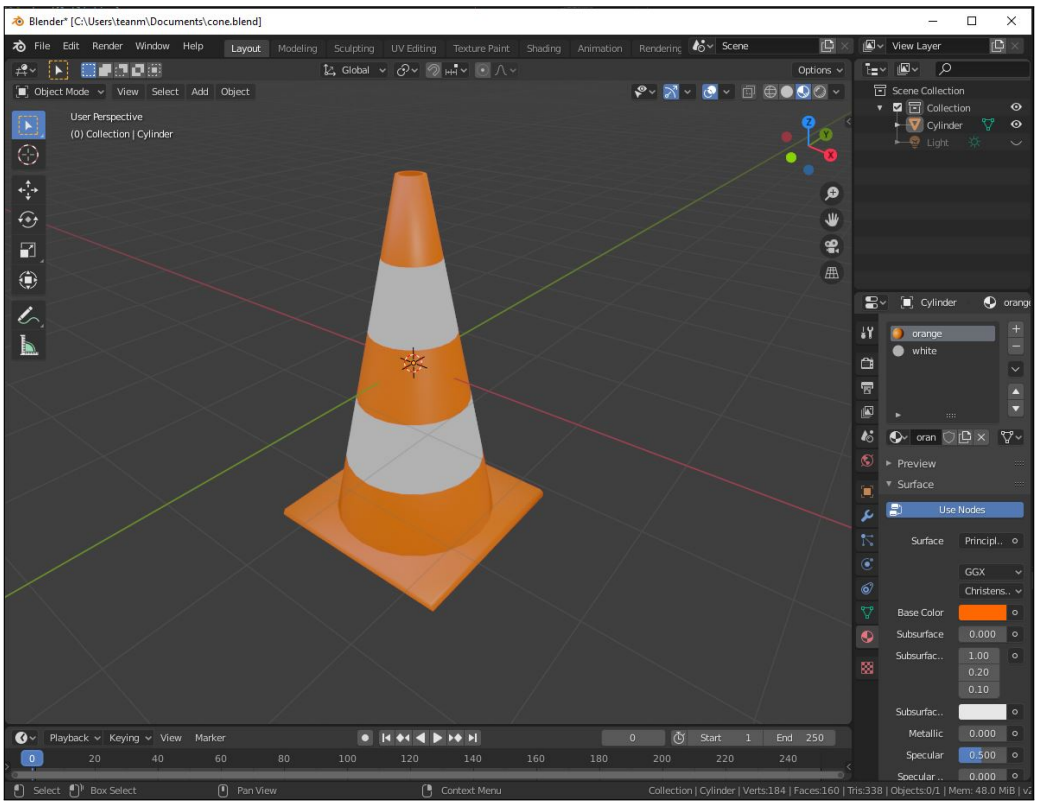

Figure 30: Props - Cone

Map

The current map does not resemble a stadium at all so I wanted to attempt to create my own map using Paint. The track needed to be to scale by using dimensions of athletic track and stadium. After trialling with a few image pixel sizes, I decided to optimise the RGB image quality by multiplying all measurements (in metres) by 2 pixels. Since Paint is very low quality using the original 128x128 caused too much interference with the 'smoothness' of the 'stadium' shape of the track, 1 to 1 scaling proved problematic with object models scaling and anything larger was too difficult to render. This was the happiest medium.

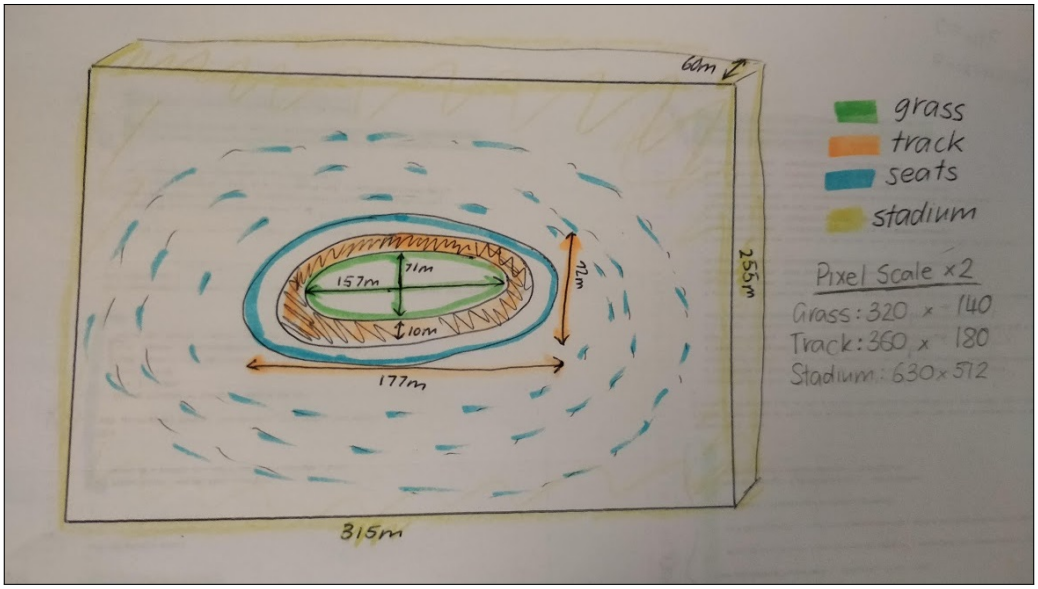

Figure 31: Track - Olympic Stadium - Dimensions

The track needed the 400m athletic track, grass in the entire of this track and then 6 levels of seating each 10m in height. With the basic layout and dimensional shape of the stadium set. The RGB colouring needed to be applied. Using the same setup, blue denoted the main channel (max speed), green the locations of object models and red the height.

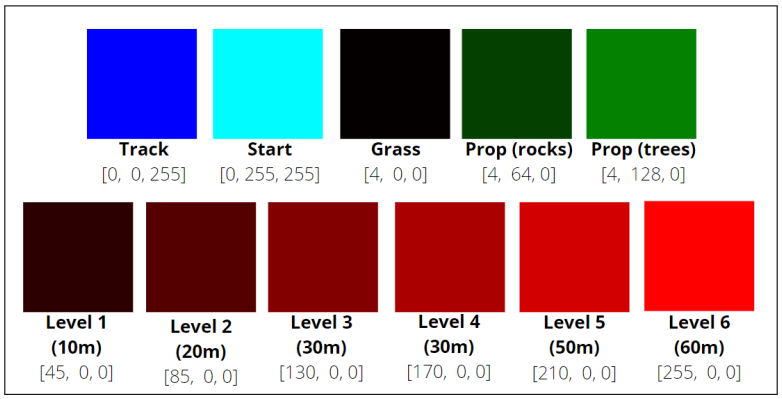

Figure 32: Track - RGB Colours

Combining all of these together following track was produced to represents a stadium. The smoothness is definitely not ideal, and the sloping was difficult to duplicate. Though with access to only Paint and the low quality this was the best I could do.

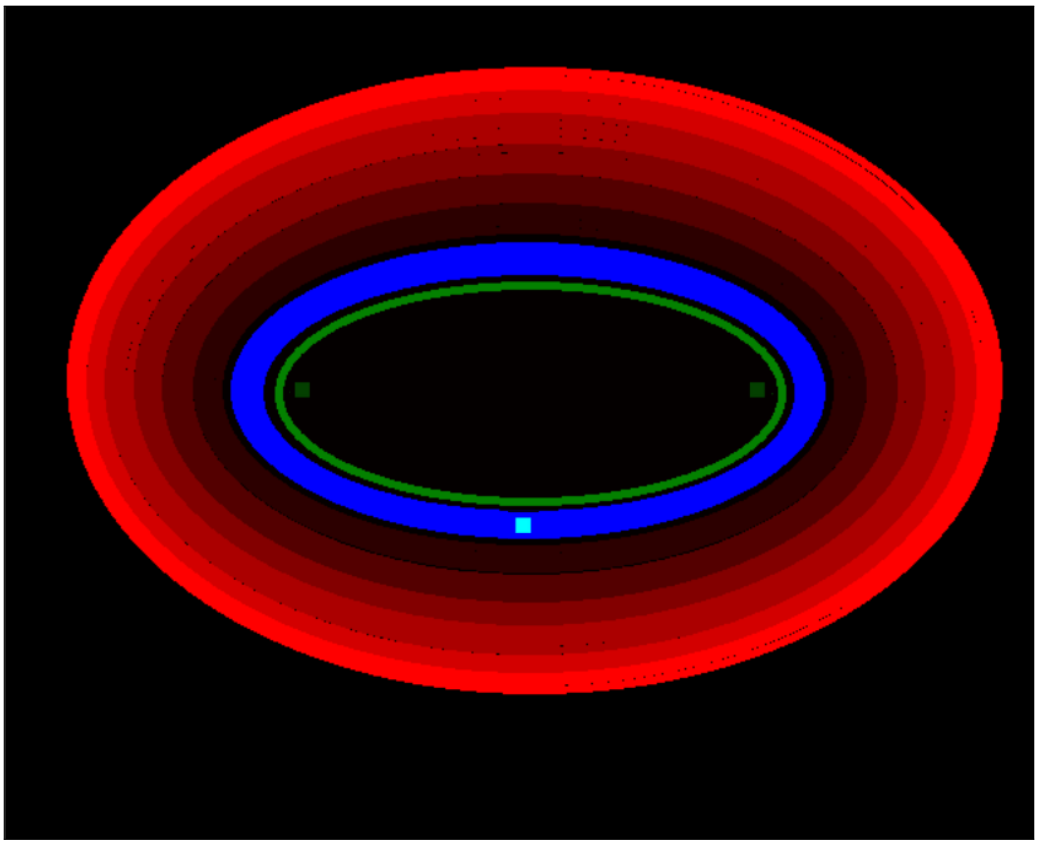

Figure 33: Track - Olympic Stadium - Final Track

```
#mega_racer.py
g_terrain.load("data/track.png", g_renderingSystem);

# terrain.py
class Terrain:
    xyScale = 2
    heightScale = 60.0
```

```
....
def load(...)
    ...
    self.mapTexture = ObjModel.loadTexture("track.png", "data", False)
```

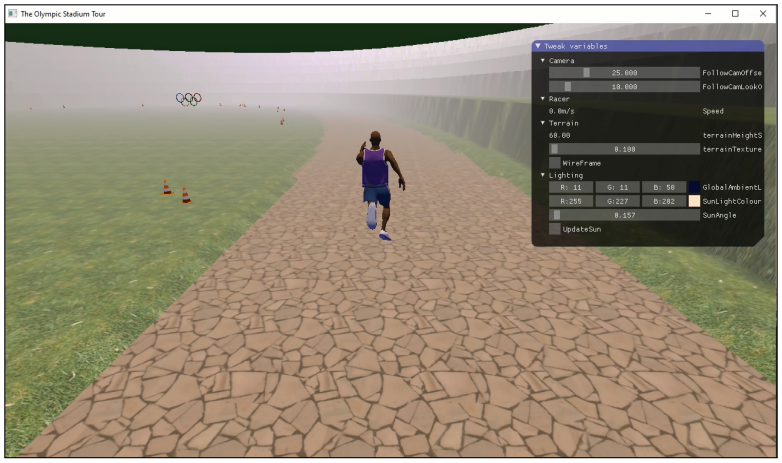

Figure 34: Track - RGB Colours

Terrain

Finally, the textures of the terrain needed to be updated to reflect an Olympic Stadium. There were four textures to be implemented, as per the colouring on the map.

1. Grass: This remained the same as mega_racer, however a different 2D image was selected for the grassTexture.
2. Wall: This replaces the highTexture from mega_racer. This texture is used to display a banner on the side wall, separating the athlete from the seating. This texture is only selected for the height of this wall, and the 2D image is a banner advertising the Tokyo 2020 Olympics.
3. Seats: This replaces the steepTexture from mega_racer. This texture is used for all areas above the height of the wall with a specific slope so that it only appears where there should be seating in the stands. This texture was the most disappointing and further adjustments to create more realistic seating would provide a more realistic appearance.
4. Track: This replaces the road as the texture where the racer/athlete is at max speed. Instead of pavement the texture is a 2D image of red synthetic athletic track with a white line. Using the measurements from the map, an almost 8-lane track is visualised.
5. Concrete: This is an additional texture that was added and replaces the grassTexture as the default texture. The 2D image is a simple concrete material that is used as the materialColour for all areas that do not meet the above logic.

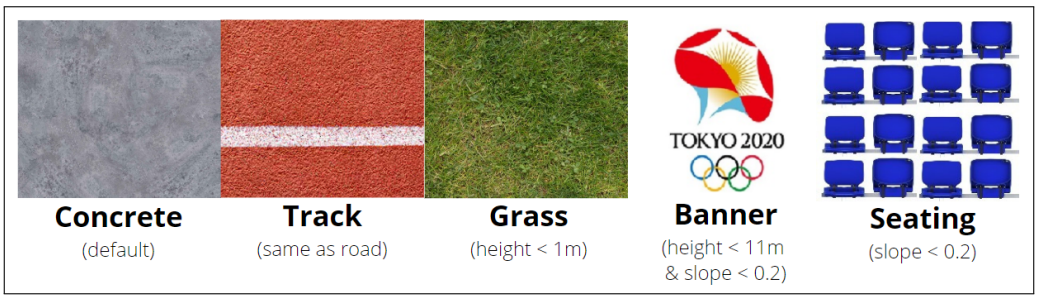

Figure 35: Terrain - Textures

```
#mega_racer.py
# terrain.py
class Terrain:
    ...
    textureXyScale = 0.25
    ...
    TU_Grass = 0
    TU_Wall = 1
    TU_Seats = 2
    TU_Track = 3
    TU_Concrete = 5
    ...
    wallTexture = None
    seatsTexture = None
    trackTexture = None
    concreteTexture = None
    ...
    def render(...):
        ...
        lu.bindTexture(self.TU_Wall, self.wallTexture)
        lu.setUniform(self.shader, "wallTexture", self.TU_Wall)
        lu.bindTexture(self.TU_Seats, self.seatsTexture)
        lu.setUniform(self.shader, "seatsTexture", self.TU_Seats)
        lu.bindTexture(self.TU_Track, self.trackTexture)
        lu.bindTexture(self.TU_Map, self.mapTexture)
        lu.setUniform(self.shader, "trackTexture", self.TU_Track)
        lu.setUniform(self.shader, "mapTexture", self.TU_Map)
        lu.bindTexture(self.TU_Concrete, self.concreteTexture)
        lu.setUniform(self.shader, "concreteTexture", self.TU_Concrete)
        ...
    def load(...):
        fragmentShader = """
            ...
            uniform sampler2D wallTexture;
            uniform sampler2D seatsTexture;
            uniform sampler2D trackTexture;
            uniform sampler2D mapTexture;
            uniform sampler2D concreteTexture;
            ...
            void main()
            void main()
            {
                vec3 materialColour = vec3(v2f_height/terrainHeightScale);
                // Default colour
                vec3 concreteColour = texture(concreteTexture, v2f_worldSpacePosition.
                materialColour = concreteColour;

                // 2.1
                float slope = dot(v2f_worldSpaceNormal, vec3(v2f_worldSpaceNormal.x, 0
```

```
        float blueChannel = texture(mapTexture, (v2f_worldSpacePosition.xy - v

        // Track Texture
        if (blueChannel >= 0.9) {
            vec3 trackColour = texture(trackTexture, v2f_worldSpacePosition.xy
            materialColour = trackColour;
        // Grass texture
        } else if (v2f_height < 1) {
            vec3 grassColour = texture(grassTexture, v2f_worldSpacePosition.xy
            materialColour = grassColour;
        // Wall/Banner texture
        } else if ((v2f_height < 11) && (slope < 0.2)) {
            vec3 wallColour = texture(wallTexture, v2f_worldSpacePosition.xy *
            materialColour = wallColour;
        // Seats
        } else if (slope < 0.2) {
            vec3 seatsColour = texture(seatsTexture, v2f_worldSpacePosition.xy
            materialColour = seatsColour;
        }
        vec3 reflectedLight = computeShading(materialColour, v2f_viewSpacePosi
        fragmentColor = vec4(toSrgb(applyFog(reflectedLight, -v2f_viewSpacePos
    }
...
self.wallTexture = ObjModel.loadTexture("banner.png", "data", True)
self.seatsTexture = ObjModel.loadTexture("seats.png", "data", True)
self.trackTexture = ObjModel.loadTexture("track.png", "data", True)
self.mapTexture = ObjModel.loadTexture("track_edit5.png", "data", False)
self.concreteTexture = ObjModel.loadTexture("concrete.jpg", "data", True)
```

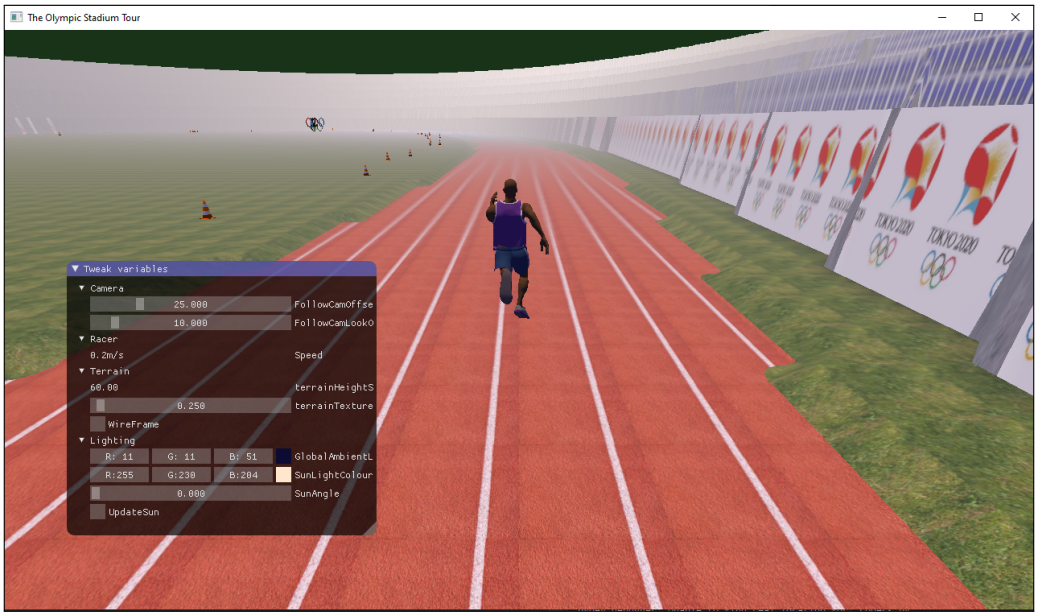

Figure 36: Terrain