COSC3000 - COMPUTER GRAPHICS REPORT

Olympics Athletic Track

Tean-louise Cunningham (42637460)

May 21, 2020

Contents

List of Figures

1 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.

2 Mega Racer - Tier 1

2.1 1.1 - Scale the terrain grid

In terrain.py load() change zPos from 0 to the below to set the height of the terrain.

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

The height of the terrain is indicated by the shading of red in the provided track image and heightScale refers to appropriate height to x and y measurements given.

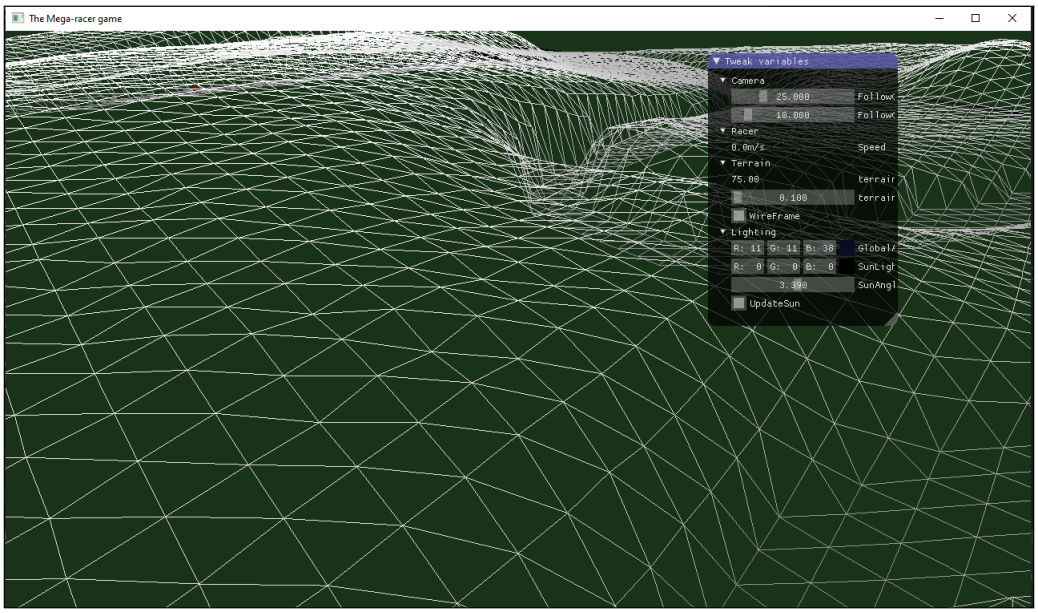

Figure 1: 1.1

Also, the last line from racer.py was uncommented so the racer can follow the height.

2.2 1.2 - Set up a camera to follow the racer

To start I looked at the code to understand what was happening and what each of the variables related to, particularly g_viewPosition, g_viewTarget and g_viewUp. I found they were being

called in mega_racer.py in *renderFrame()*:

```
view.worldToViewTransform = lu.make_lookAt(g_viewPosition,
                                           g_viewTarget,
                                           g_viewUp)
```

Then I headed to *make_lookAt()* to see the functionality of the function and parameters that the arguments were being used for. The function makes a transformation from world to view space inversely by calling another function *make_lookFrom()* with a modified target parameter for smooth movement.

```
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):
    .....
```

Also, the examples from lab2 use similar functions in the same manner:

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

Therefore, the variables have the following meanings:

- g_viewPosition = eye (location of camera, camera position)
- g_viewTarget = target (point to aim, camera direction=target-eye)
- g_viewUp = up (rough up direction)

With this understanding I could start to assign these variables appropriately to get the camera in the correct position. Firstly, it was clear that the 'target' needed to be updated to the position of the racer as this where the camera needs to be pointing.

```
g_viewTarget = g_racer.position
```

Secondly, the position of the camera needed to be updated to take into account corresponding position of the racer for all coordinates. According to .... the 'desiredPosition = target.position + offset'. Thirdly, all coordinates of the of the camera needed to include the follow offset to be behind and above the racer. Additionally, the z coordinates needed to take into account the look offset of the camera as this is how it is looking at it from above.

```
for i in range(0,3):
    g_viewPosition[i] = g_racer.position[i] + g_followCamOffset
g_viewPosition[1] += g_followCamLookOffset
```

After running these changes I could see that the camera was almost right, however it wasn't facing the correct direction; directly behind the racer.

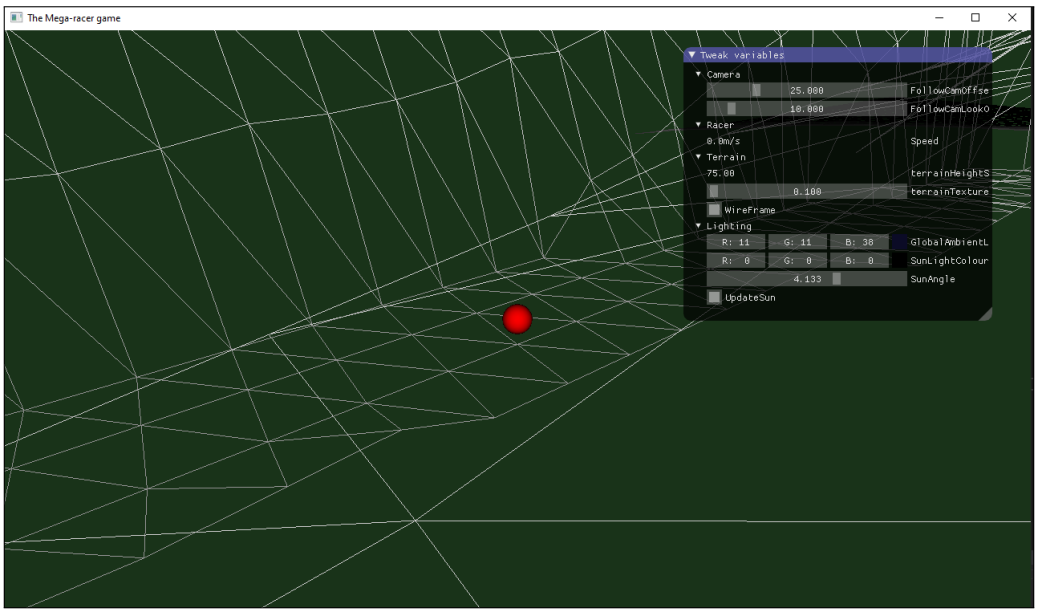

Figure 2: 1.2 - First Attempt

On closer inspection of the Racer class I identified a variable called heading which controlled the direction of the racer. Since the camera needed to be behind the racer, the view position had to be multiplied by the inverse of the racer direction.

```
for i in range(0,3):
    g_viewPosition[i] = g_racer.position[i]
                      + g_followCamOffset
                      * -g_racer.heading[i]
g_viewPosition[2] += g_followCamLookOffset
```

This produced the correct result! The camera was now the position of the racer with the offsets applied at the appropriate direction to the racer's movement.

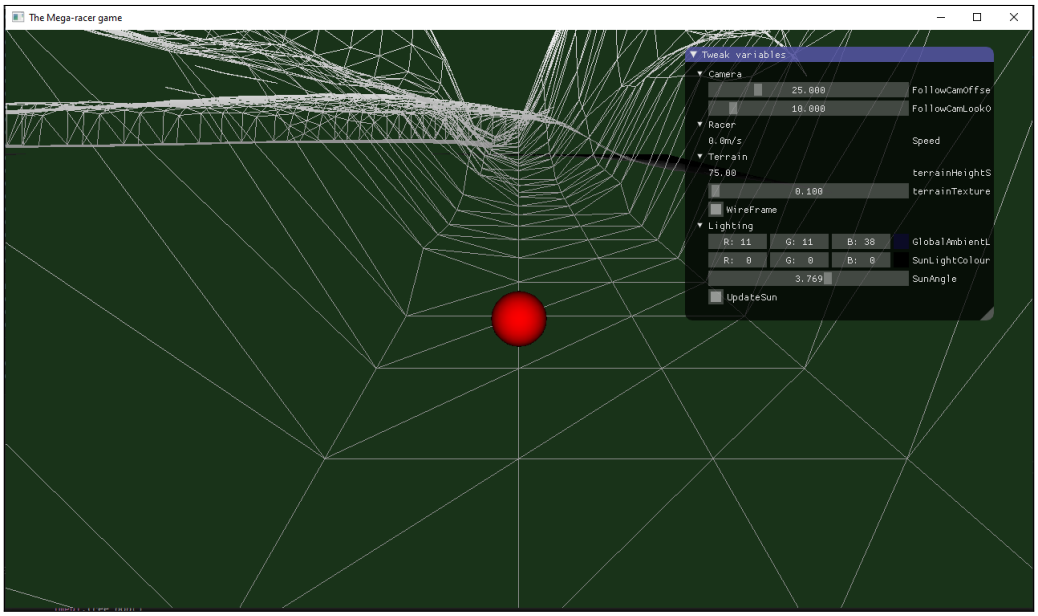

Figure 3: 1.2 - Final Attempt

2.3 1.3 - Place and orient a model for the racer

Looking at the comments in the code I found two places with TODO for this section in racer.py; render() and load(). Starting with rendering the model I look at mega_racer.py to see how this function was being called. The racer was being rendered in renderFrame().

```
def render(self, view, renderingSystem):
    .....
    -->
g_racer.render(view, g_renderingSystem)
```

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. The renderingSystem parameter is of RenderingSystem class. The task for render() is to draw the model and this class has the drawObjModel() function mentioned in the project notes that needs to be used. This function call is added to Racer.render(). In the Racer class there is model variable set to None, and view is passed in as an argument to render(). The only missing parameter is the modelToWorldTransform.

```
drawObjModel(self, model, modelToWorldTransform, view)
-->
renderingSystem.drawObjModel(self.model, ???, view)
```

The project notes mention make_mat4_from_zAxis() as a useful function for transforming the model to the world. This function is in the lab_utils file (imported as lu in racer.py). The function takes the parameters translation, zAxis and yAxis. The zAxis represents forwards and the yAxis represents up. Therefore, for this world, the z an y axis will act conventionally to the same. This means that the z-axis is the same as the direction of the racer (the heading) and the y-axis is equal to up view of the racer as determined by g_viewUp in mega_racer.py. The translation parameter refers to the position of model after all the relevant modifications to the coordinates, in this case that is the position of the racer.

```
make_mat4_from_zAxis(translation, zAxis, yAxis)
-->
modelToWorldTransform = lu.make_mat4_from_zAxis(self.position,
                                                self.heading,
                                                [0.0, 0.0, 1.0])
```

After running this 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().

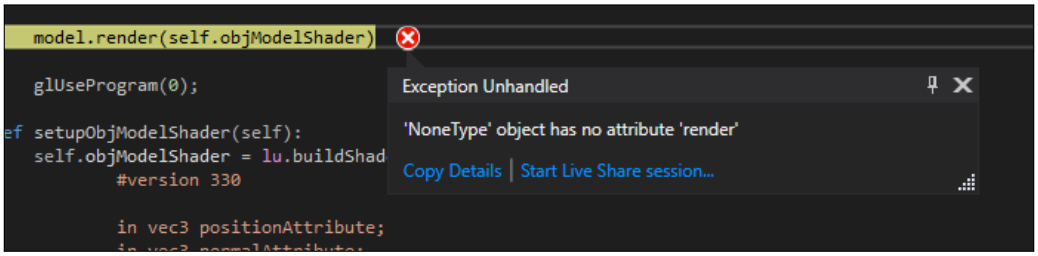

Figure 4: 1.3 -Error

Looking at mega_racer.py the load() method is called with three parameters; the object file for the racer, the terrain and the rendering system. The purpose of the load() function is to create and load the model.

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

The rendering system there seemed to be no relevant functions. In the project notes there was mention that the Racer class relied on the ObjModel class, and since drawObjModel() draws ObjModel's then the racer model needed to be an instance of this class. Looking at ObjModel__init__() the only requirement argument is the filename which is given.

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

Now the model for the racer has replaced the red dot and the movement relating to the arrows in accurate.

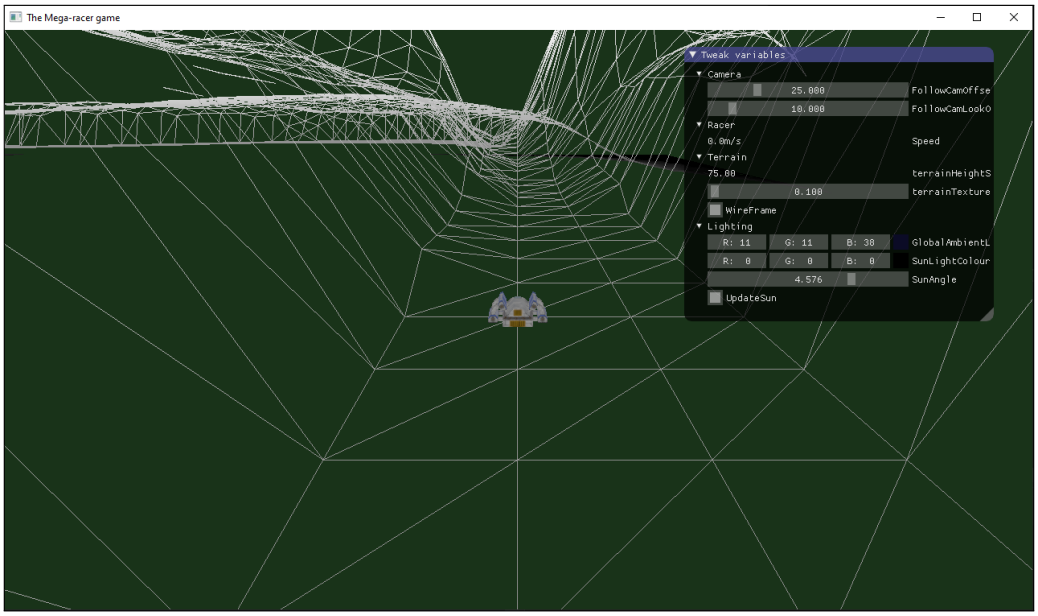

Figure 5: 1.3 - Final

2.4 1.4 - Texture the terrain

There are three steps to complete this section in terrain.py.

(1) render() - need to bind the grass texture to the right texture unit.
The hint is to is to use lu.bindTexture. This function, bindTexture(), takes two arguments texUnit and textureId. A similar function is used in lab_utils.py (Lab Code, 2020), also called bindTexture(). In order to use the bindTexture in render() for the project I looked at how it was called in lab 5 question 12 (Lab Code, 2020).

```
# lab_utils.py
def bindTexture(texUnit, textureId, textureType = GL_TEXTURE_2D)
    ...
 -->
# lab_5_template_2.py - Q12
g_detailTexture = None
def renderFrame(...):
```

```
.....
lu.bindTexture(0, g_detailTexture)
lu.setUniform(g_shaderProgram, "baseTexture", 0)
```

The textureId was first declared and then passed in and the texUnit set to 0. In terrain.py the texUnit is given as TU_Grass (also equal to 0) and the textureID needs to be declared. Additionally, setUniform() needs to be called to set the shader. Updating the variable names the same was added to terrain.py.

```
#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 I looked at Lab 5 question 12 which sets the colours of the texture in the fragment shader.

```
# Lab 5 - Q12
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;
        fragmentColor = vec4(detailColour, 1.0);
    }
```

From this example it is evident that the three arguments for texture() need to be assigned to the new materialColour (i.e. detailColour).

- detailTexture: The grass colour is similar to the detailColour so the detailTexture is equivalent to the grassTexture.
- texCoordScale: In lab 5 the v2f_textureCoord is a 2 dimensional out vector from the vertex shader. As per the project notes, since the vertex is regular, we can replicate this same behaviour by only selecting the x and y coordinates of the world space which are stored in v2f_worldSpacePosition.
- v2f_textureCoord: The textCoordScale in lab 5 is a new name for the given g_texCoordScale 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.

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

#Terrain
textureXyScale = 0.1
def render():
    lu.setUniform(self.shader, "terrainTextureXyScale", self.textureX
```

Putting this all together we get the following code to calculate the coordinates and sample the texture for the grass, overriding the existing material colour.

```
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
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.

```
self.grassTexture = ObjModel.loadTexture("grass2.png", "data", True)
```

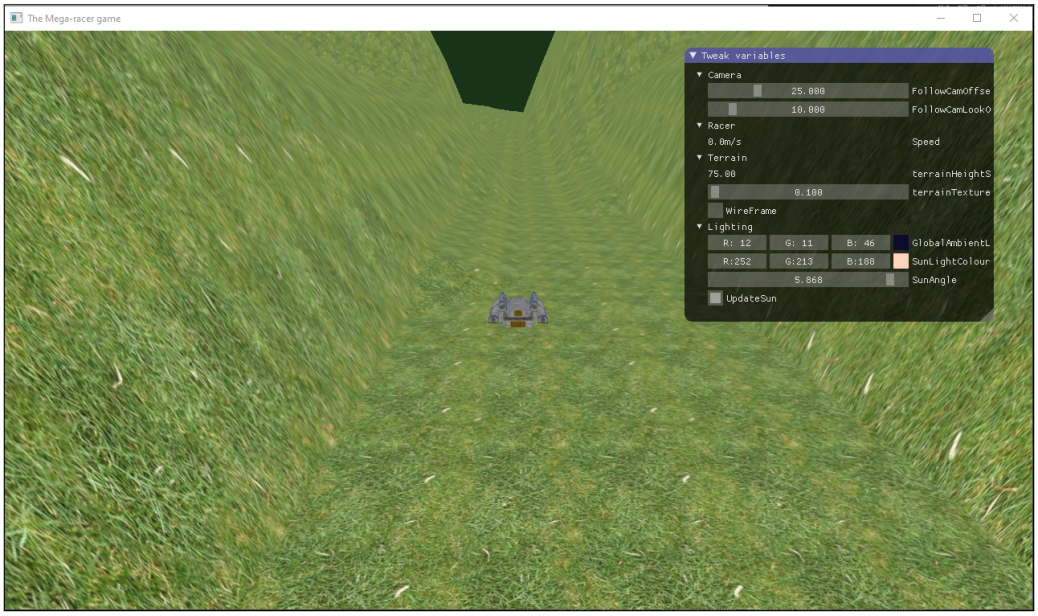

Figure 6: 1.4 - Final

2.5 1.5 Lighting from the sun

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.

```
# setupObjModelShader()
v2f_viewSpaceNormal = normalize(modelToViewNormalTransform * normalAttrib
v2f_viewSpacePosition = (modelToViewTransform * vec4(positionAttribute, 1
.....
vec3 materialDiffuse = texture(diffuse_texture, v2f_texCoord).xyz * mate
vec3 reflectedLight = computeShading(materialDiffuse,
                                     v2f_viewSpacePosition, v2f_viewSp
                                     sunLightColour
                                    ) + material_emissive_color;
```

A couple of things to keep in mind:

- position of sun = g_sunPosition: The sun is set to move around around the world.
- colour of sunlight = g_sunLightColour = lightColour
- all parameters om shading must be the same space -¿ set as view space (same as lab 4)
- sun is defined in world space
- 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.

```
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 is to compute the normalised direction towards the light from the shading point in view space. The light position is stored as viewSpaceLightPosition and the current point is viewSpacePosition (provided by the vertex shader).

```
# Lab 4 - Q2 - FragmentShader
vec3 viewSpaceDirToLight = normalize(viewSpaceLightPosition - viewSpaceP

# mega_racer.py
vec3 viewSpaceDirToLight = normalize(viewSpaceLightPos- viewSpacePosition
```

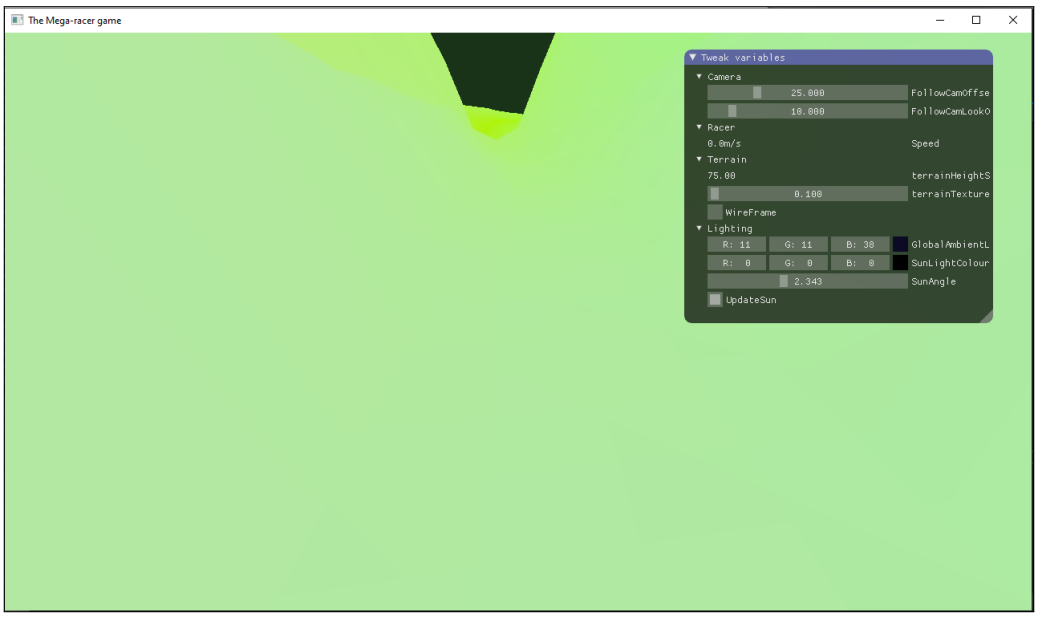

Figure 7: 1.5 - Step 1

(2) Compute the incoming light intensity. Use the calculated direction to calculate the incoming light intensity. 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, viewSpaceDirToLig

# mega_racer.py
```

```
float incomingIntensity = max(0.0, dot(viewSpaceNormal, viewSpaceDirToLig
```

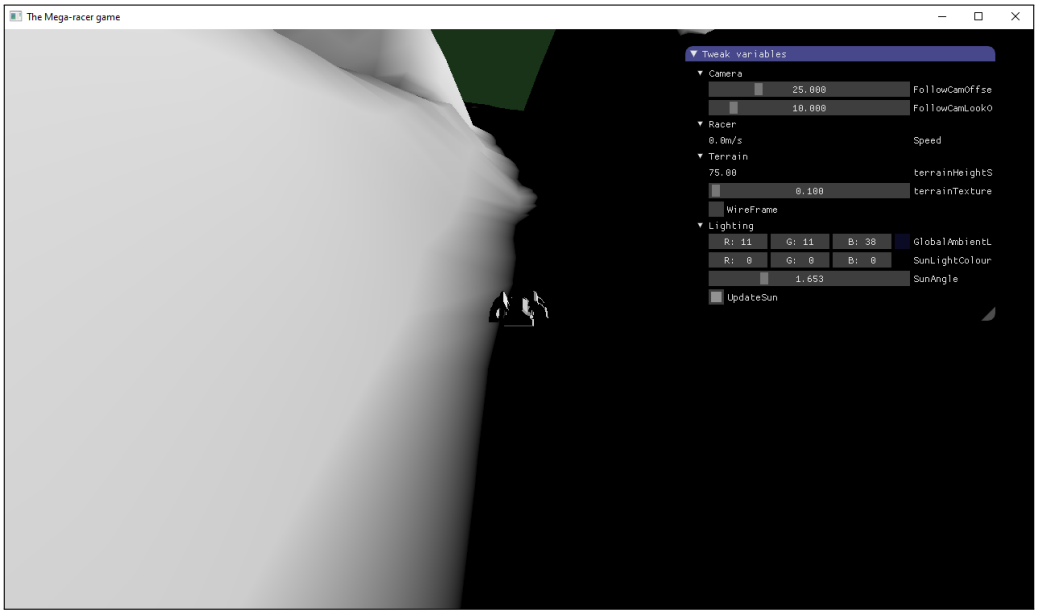

Figure 8: 1.5 - Step 2

(3) Modify the light that is emitted by the light source to have the correct colour and maximum intensity. This fixes the proportion if incoming light arriving at the surface so it is the correct colour and maximum intensity

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

# mega_racer.py
vec3 incomingLight = incomingIntensity * lightColour;
```

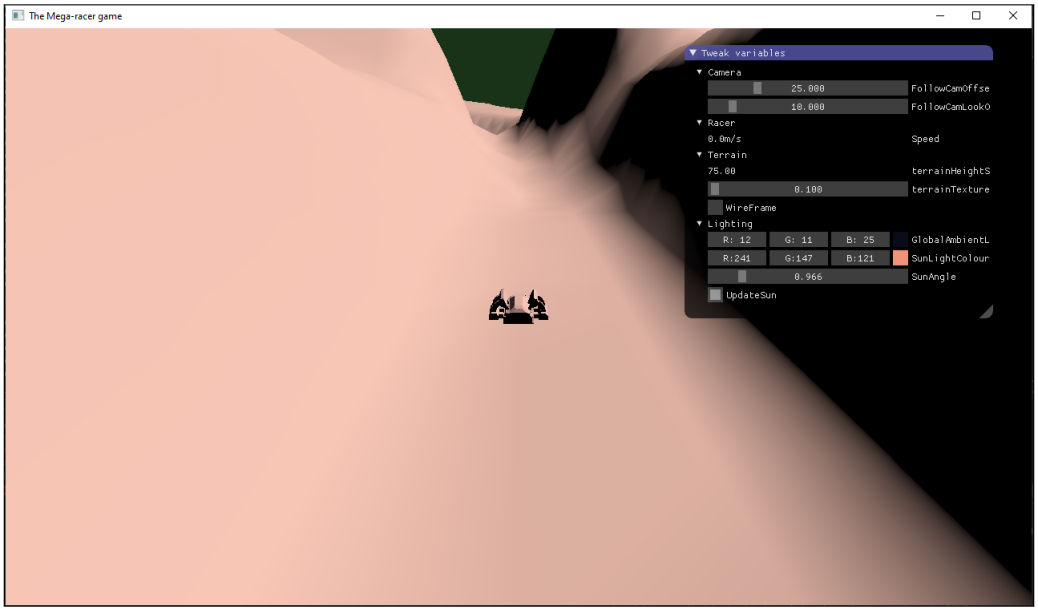

Figure 9: 1.5 - Step 3

(4) Diffuse Lambertian Reflection. Moving on to question 3 of lab 4, the next step is to diffuse the Lambertian reflection by multiplying the incoming light with a constant that represents the reflection of the material for the given spectrum. The argument passed as

materialColour should provide this representation, as provided by the materialDiffuse variable passed in to computeShading() in setUpObModelShader(). This variable follows the same pattern as that of lab 4.

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

# mega_racer.py
vec3 outgoingLight = incomingLight * materialColour;
```

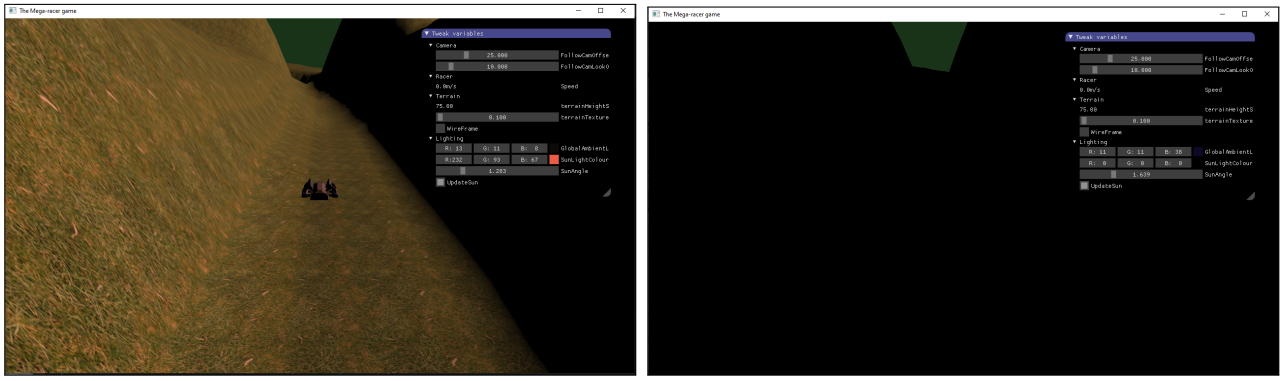

Figure 10: 1.5 - Step 5

4

(5) Take into account the indirect light. The next step is to handle the ambient light. 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
vec3 outgoingLight = (incomingLight + globalAmbientLight)
                     * materialColour;
```

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:

- As an aside, a slightly more advanced ambient model, which is sometimes used for outdoor scenes is to have two colours and blend between them based on the orientation of the surface, such that things facing straight up get a blue ambient light (from the sky) and down a green tint (to represent grass reflecting light up). We will leave that for now though!
- Specular lighting