COSC3000 - COMPUTER GRAPHICS REPORT

Olympics Athletic Track

Tean-louise Cunningham (42637460)

May 28, 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. Also, the last line from racer.py was uncommented so the racer can follow the height.

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

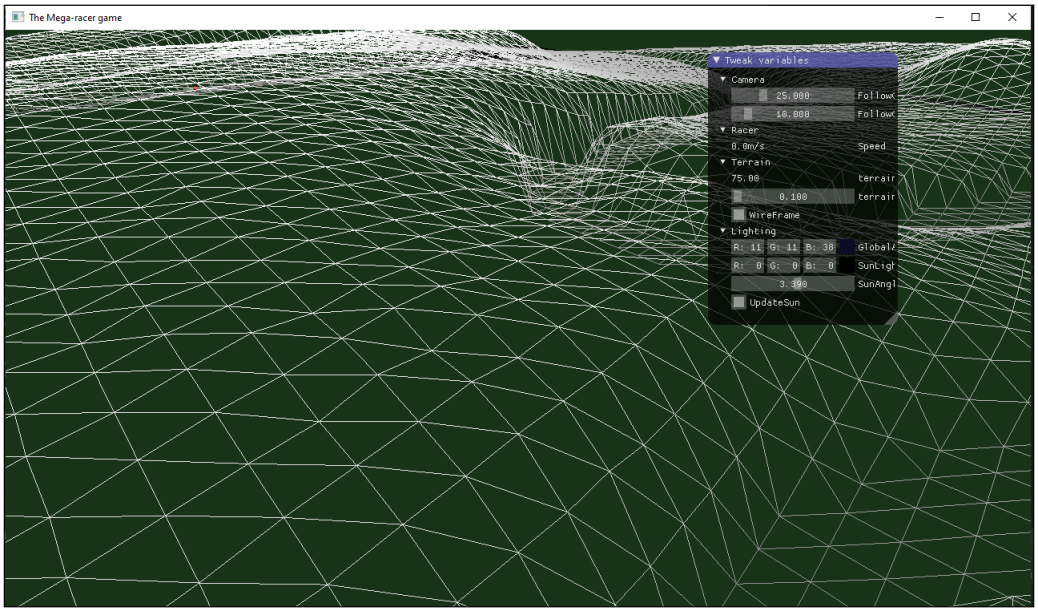

Figure 1: 1.1

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()*:

```
# mega_racer.py
def 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.

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

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.

```
# mega_racer.py
def update(...):
    ...
    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.

```
# mega_racer.py
def update(...):
    ...
    for i in range(0,3):
        g_viewPosition[i] = g_racer.position[i]
                          + g_followCamOffset
    g_viewPosition[2] += 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.

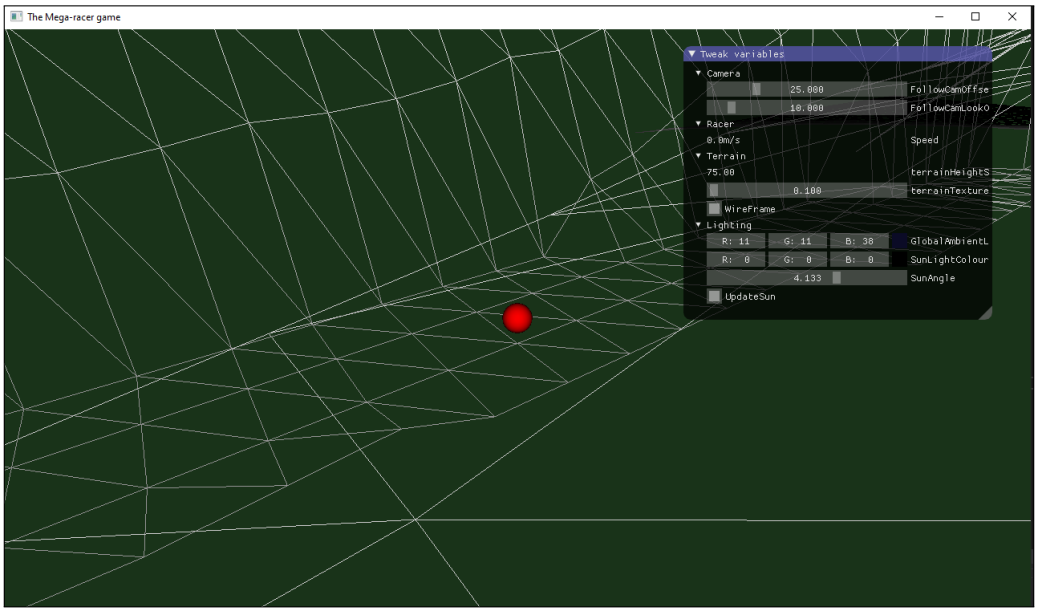

Figure 2: 1.2 - First Attempt

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.

```
# mega_racer.py
def update(...):
    ...
    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.

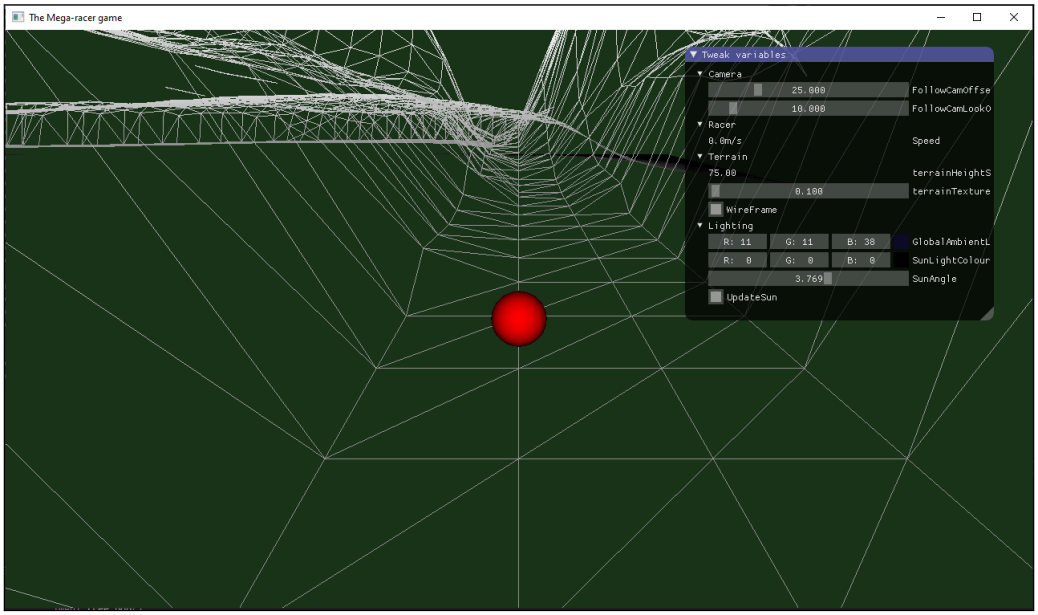

Figure 3: 1.2 - Final Attempt

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().

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

```
# mega_racer.py - RenderingSystem
def drawObjModel(self, model, modelToWorldTransform, view):
    ...
        —>
# racer.py - Racer
def render(self, view, renderingSystem):
    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.

```
#lab_utils.py
def make_mat4_from_zAxis(translation, zAxis, yAxis):
    ...
-->
# racer.py - Racer
def render(...):
    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().

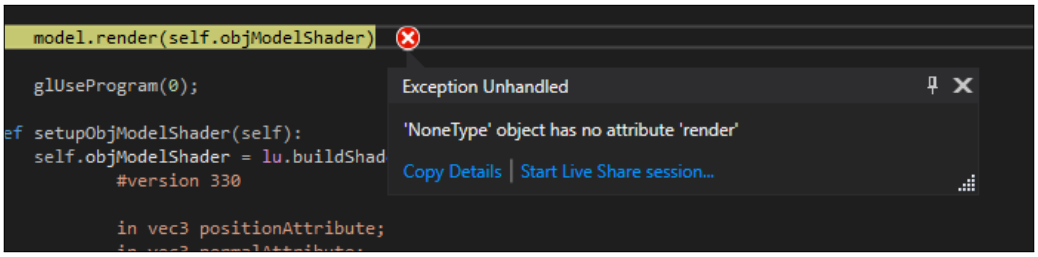

Figure 4: 1.3 -Error

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.

```
# mega_racer.py
g_racer.load("data/racer_02.obj", g_terrain, g_renderingSystem)
-->
# racer.py - Racer
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.

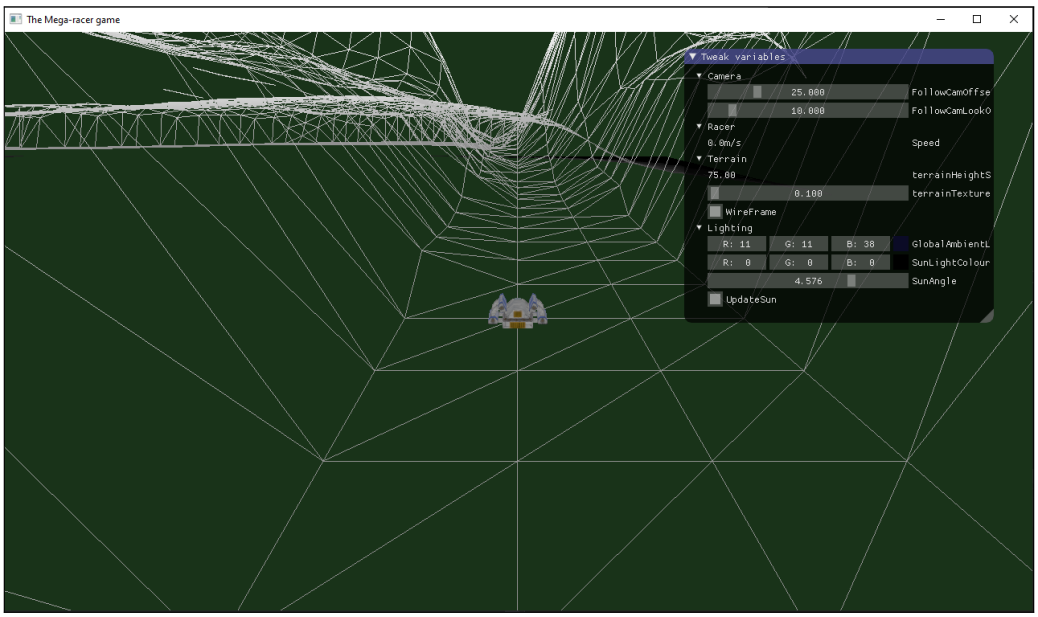

Figure 5: 1.3 - Final

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 - 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)
```

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.py
textureXyScale = 0.1
def render():
    lu.setUniform(self.shader,
                  "terrainTextureXyScale",
                  self.textureXyScale);
```

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.

```
# terrain.py - Terrain
def load(...):
    ...
    fragmentShader =
        ...
        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.

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

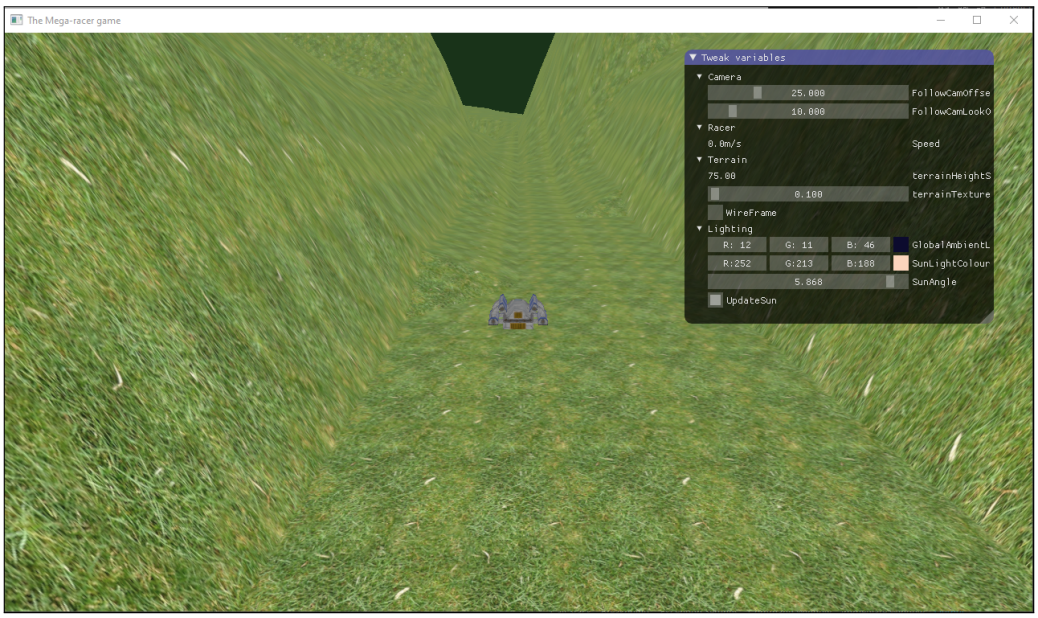

Figure 6: 1.4 - Final

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.

```
#  mega_racer.py - RenderingSystem
def setupObjModelShader(...):
    v2f_viewSpaceNormal = normalize(modelToViewNormalTransform * normalAttrib
    v2f_viewSpacePosition = (modelToViewTransform * vec4(positionAttribute, 1
    .....
    vec3 materialDiffuse = texture(diffuse_texture,
                                   v2f_texCoord).xyz
                                   * material_diffuse_color;
    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 mate-

rialDiffuse 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 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 - viewSpacePositi

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

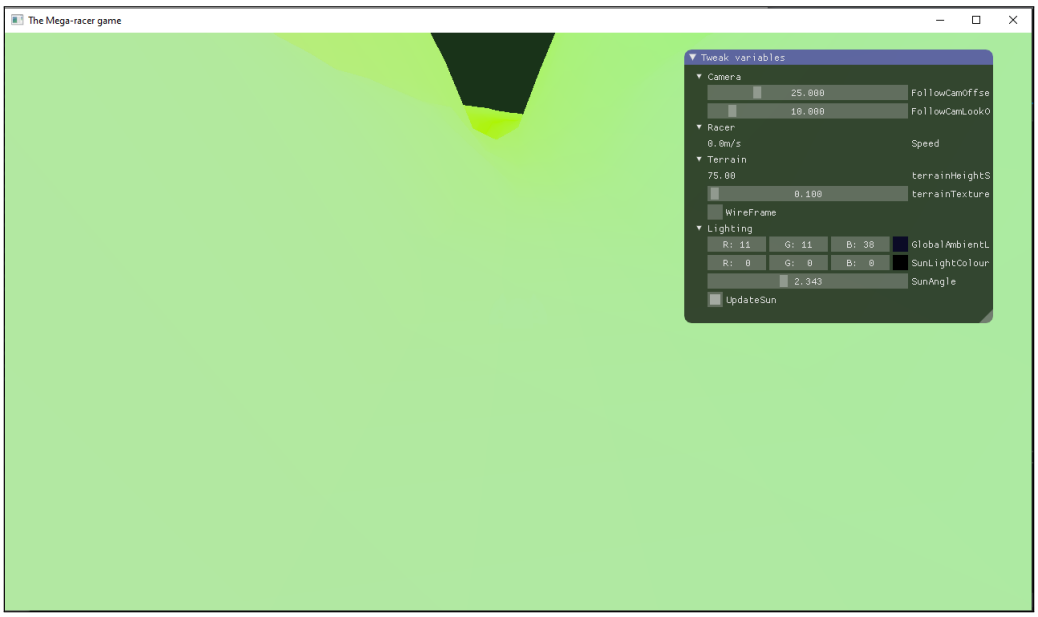

Figure 7: 1.5 - Step 1

(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, viewSpaceDirToLight))

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

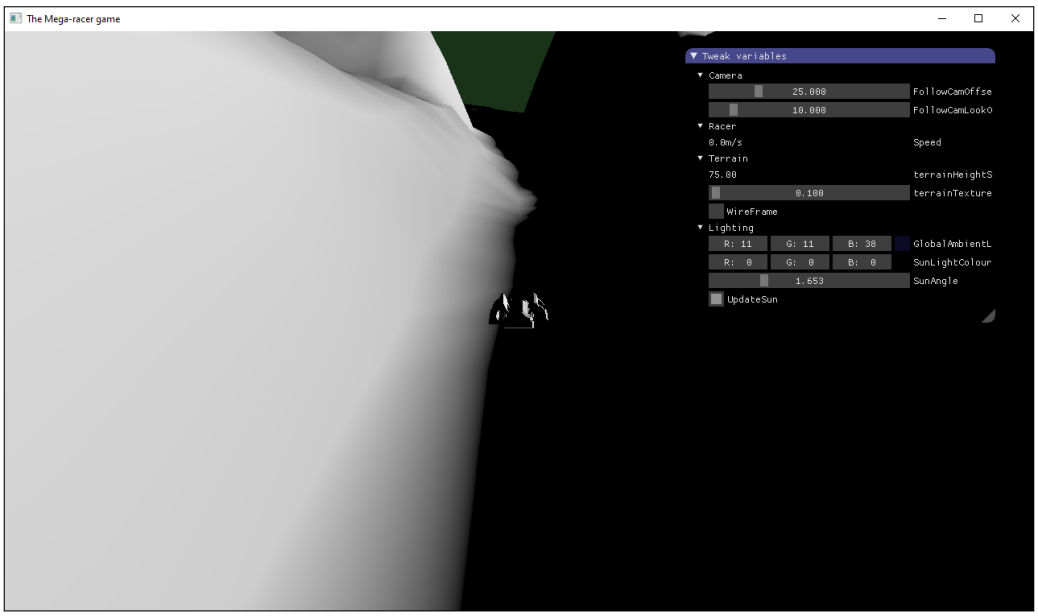

Figure 8: 1.5 - Step 2

(3) Modify the light that is emitted by the light source to have the correct colour and maximum intensity. 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 - RenderingSystem
commonFragmentShaderCode =
    ...
    vec3 computeShading(...)
    {
        ...
        vec3 incomingLight = incomingIntensity * lightColour;
        return incomingLight;
    }
```

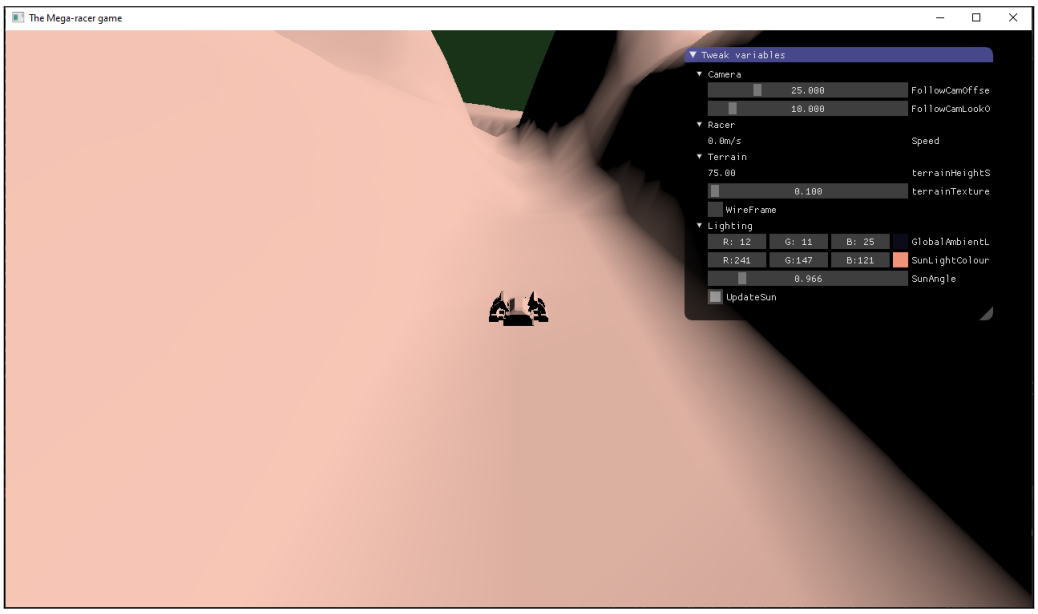

Figure 9: 1.5 - Step 3

(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 * material
vec3 outgoingLight = incomingLight * materialDiffuse;

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

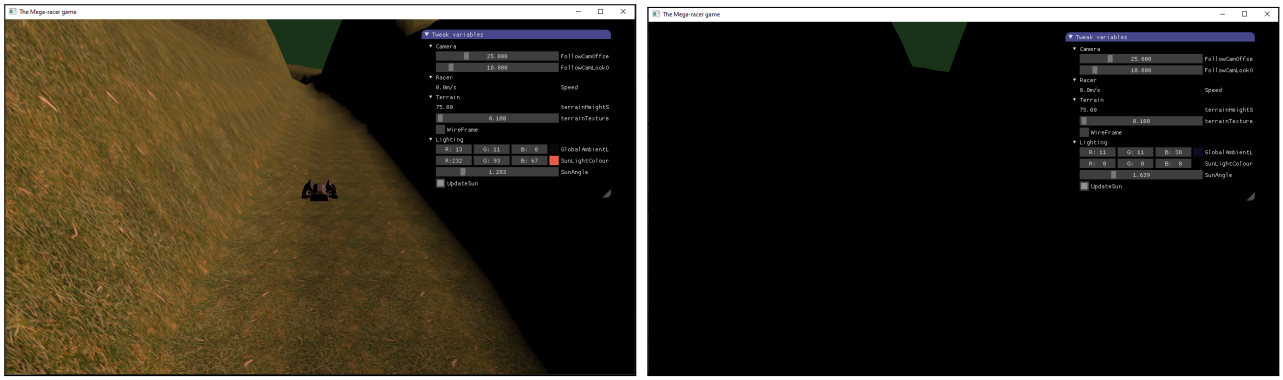

Figure 10: 1.5 - Step 5

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 - 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:

- 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

2.6 2.1 Improve terrain textures

(1) Set the high texture To set the high texture need to use the height and mix 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 texture should only appear at set heights and be blended with the grassColour.

```
    # 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():
            {
                ????
            }

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

..... About mix function ....

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


def load():
    fragmentShader =
        ...
        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));
    }
```

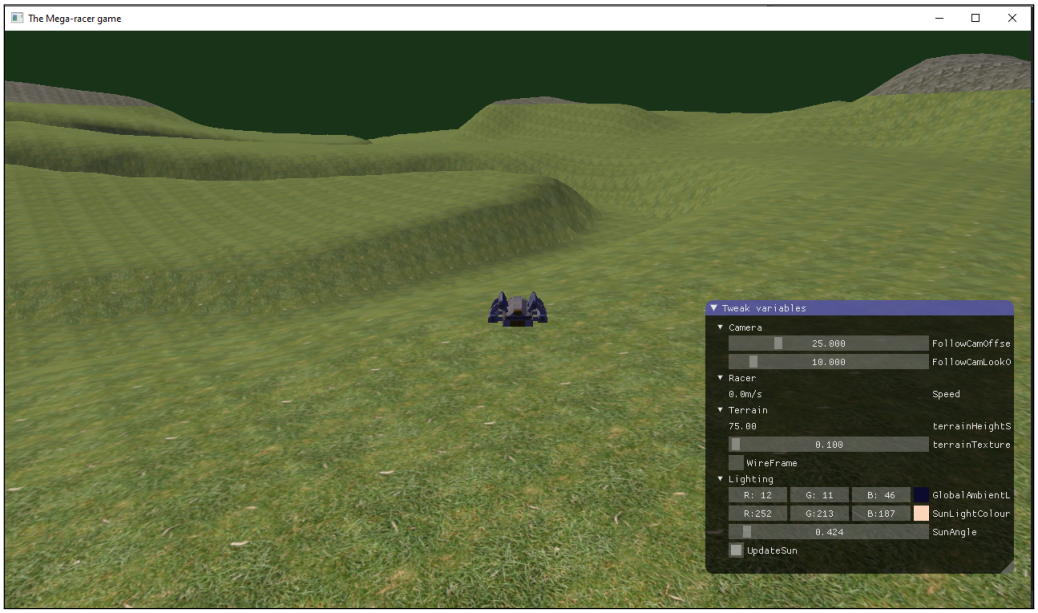

Figure 12: 2.1 - High Texture

(2) Set the steep texture First thing is to get the coordinates of the normalised world space. The viewSpacePosition, viewSpaceNormal and worldSpacePosition had already been calculated so I followed the same pattern to get 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,
        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.

```
# 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_worldSpaceNo
            if (slope < 0.8) {
                vec3 steepColour = texture(steepTexture, v2f_worldSpaceP
                materialColour = mix(materialColour, steepColour, (v2f_he
        }

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

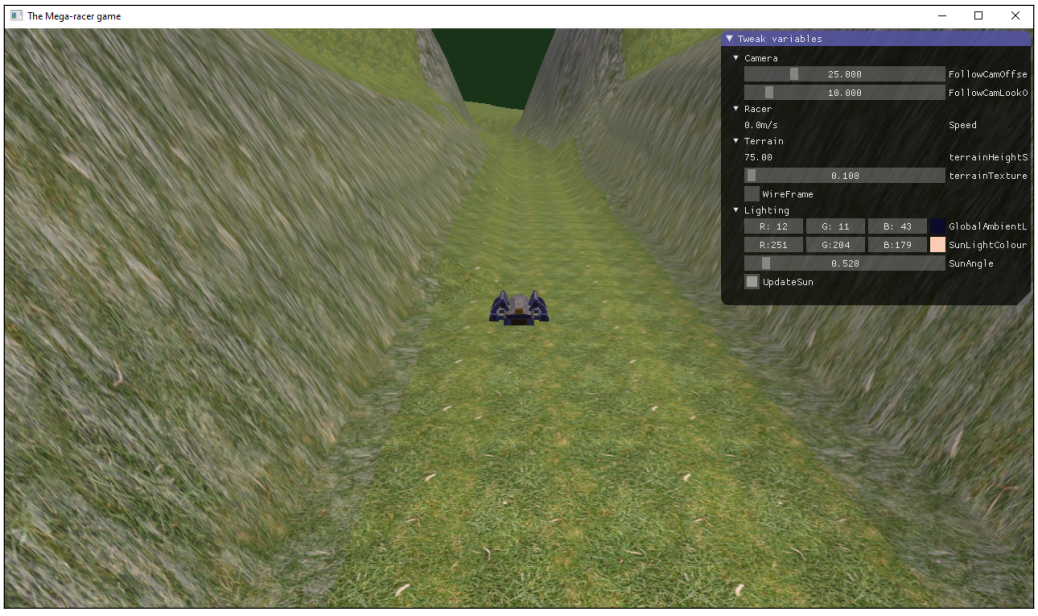

Figure 13: 2.1 - Steep Texture

(3) Add the road The same steps for the other textures were then applied to roadTexture and mapTexture, except for the sampling in the main() of fragmentShader as this would be a little different. Also, when using ObjModel.loadTexture since the track image is not a SRGB file, the argument for the srgb parameter is False.

```
# 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", Fa
```

Since the shader doesn't have access to the type of terrain (previously relying on height and slope), the blueChannel needs to be able to be accessed so the shader knows what is considered 'road'. 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. I knew I needed to use the xyNormScale variable instead of terrainTextureXyScale, but I wasn't sure of the purpose of offset yet. 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.

```
# 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.xyS
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) *
        if (blueChannel >= 0.9) {
            vec3 roadColour = texture(roadTexture, v2f_worldSpacePosition.xy
            materialColour =  mix(materialColour, roadColour, (v2f_height/ter
            }
        }
```

When I finally did get the shader working to a point that it would display it was clear that there were still a couple of issues to resolve.

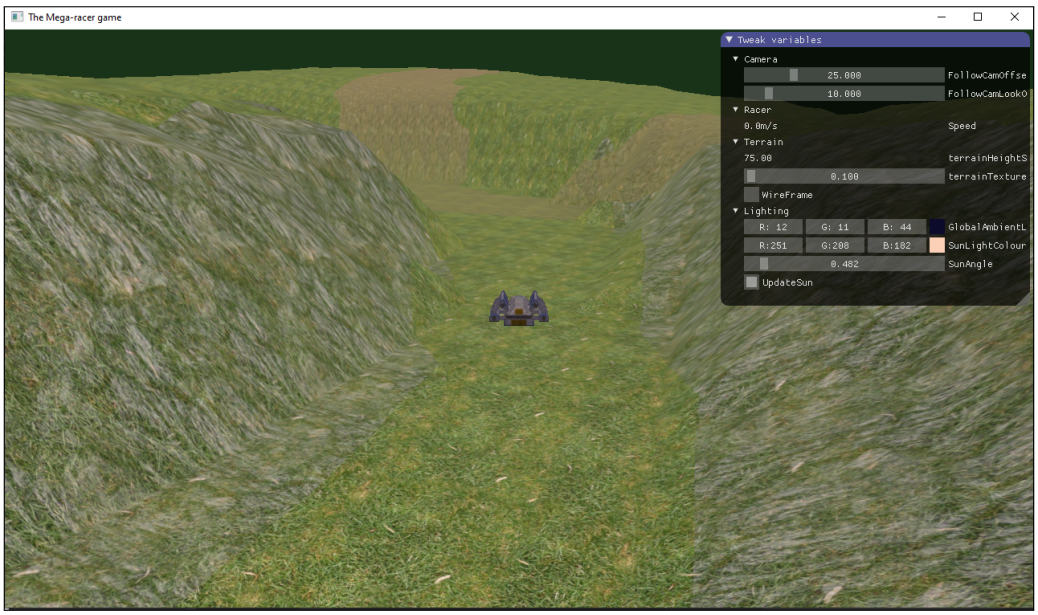

Figure 14: 2.1 - Road Texture (Mixed & No Offset)

1. The grass texture was overtaking the blend of the pavement. This is because I mixed the roadColour with the grassColour as previously done, instead I did the same as that for the grassColour and declared the materialColour of the road to be only roadColour
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 I also tried mixing the road and grass colours again but the sample problem occurred.

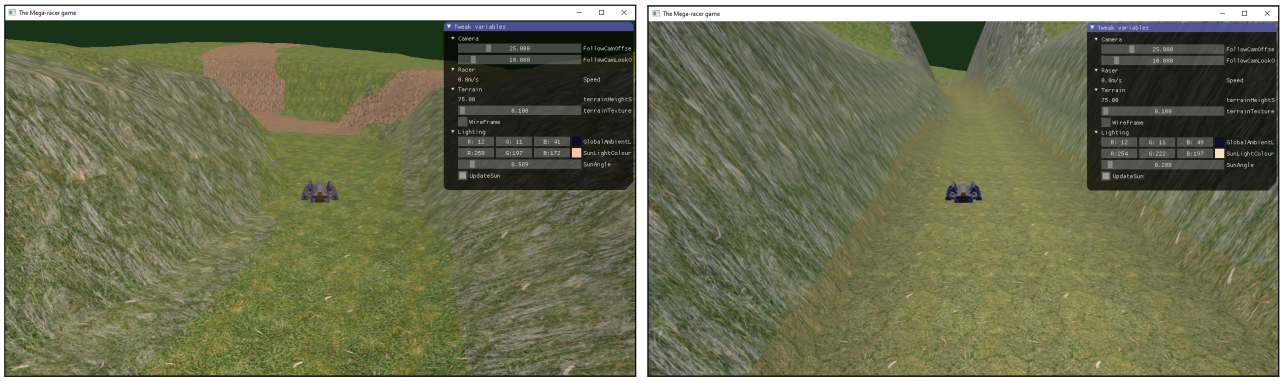

Figure 15: 2.1 - Road Texture (Not Mixed & No offset / Mixed & Offset)

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_xyO
if (blueChannel >= 0.9) {
    vec3 roadColour = texture(roadTexture, v2f_worldSpacePosition.xy * terrai
        materialColour = roadColour;
}
```

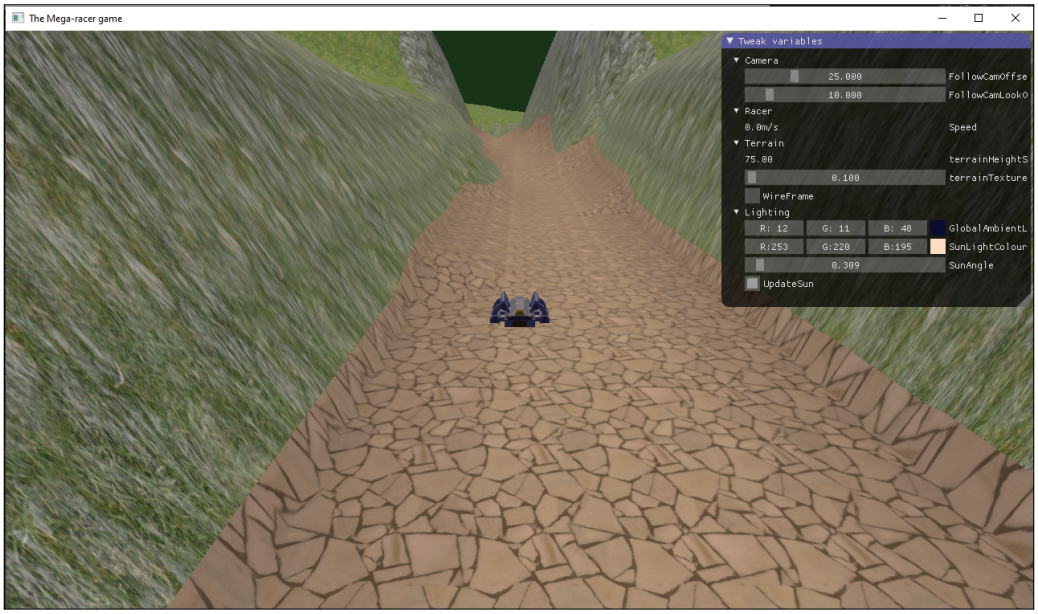

Figure 16: 2.1 - Road Texture

2.7 2.2 Add Fog

(1) Follow the tutorial) Followed the tutorial exactly. Set b to 0.005 because .....

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

Needed to also replace the fragmentColour to call this new function:

- 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;

```
#terrain.py
```

```
def load(...):
    ...
    fragmentShader =
        ...
        void main()
        {
            ...
            fragmentColor = vec4(toSrgb(applyFog(reflectedLight,
                                                 -v2f_viewSpacePosition.z)
                                        1.0);
        }
```

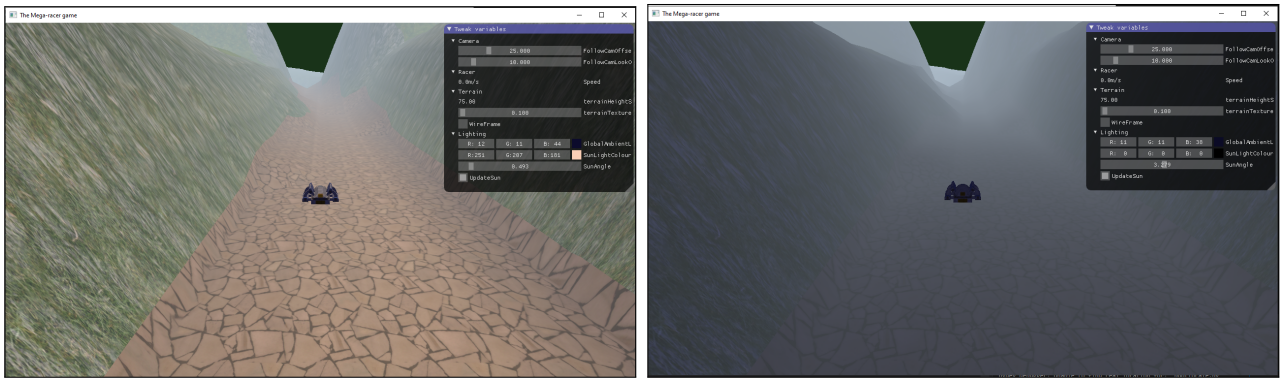

Figure 17: 2.2 - Basic

(2) Make colour sunLight and Ambient The image shows how dark this makes it and project notes suggest a combination of sunLightColour and globalAmbientLight. First I tried the addition of the two

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

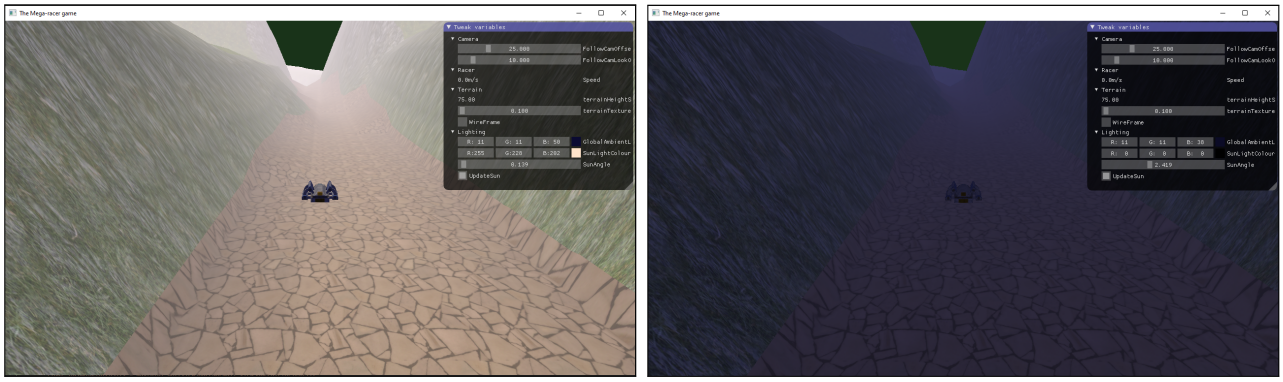

Figure 18: 2.2 - Fog Colour (Total)

Then I tried the average of the the two values

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

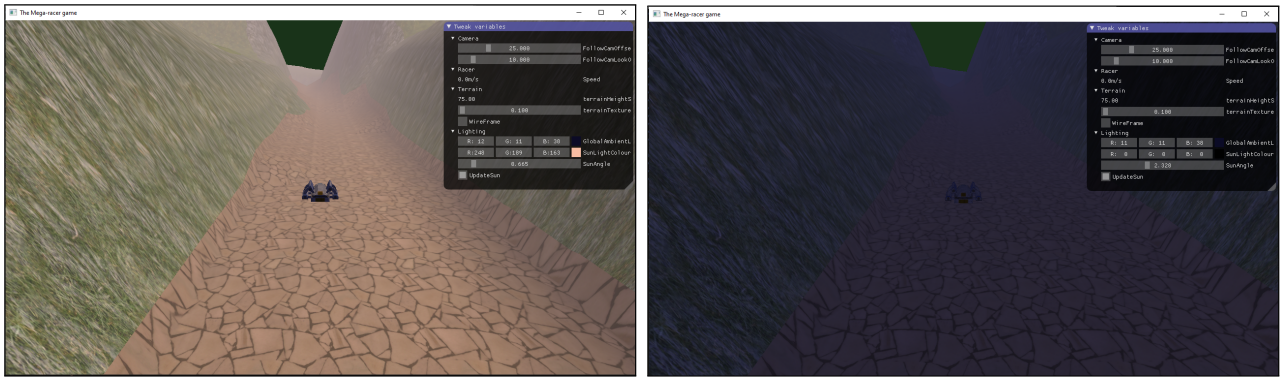

Figure 19: 2.2 - Fog Colour (Average)

(3) Harder version This version uses height based fog.

```
# 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 )
    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 ra
    {
        float b = 0.005;
        float c = 0.66;
        float fogAmount = c * exp(-rayOri.y*b) * (1.0-exp( -distance*rayDir.y
        vec3  fogColor  = (sunLightColour + globalAmbientLight) / 2.0;
        return mix( rgb, fogColor, fogAmount );
    }
```

Same as before, the call to the function also needs to be updated. There are a few more variables needed as well:

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

```
# terrain.python
def load(...):
    ...
    vertexShader =
        \dots
        uniform mat4 worldToViewTransform;
```

```
        out VertexData
        {
            ...
            vec3 cameraPosition;
            vec3 cameraToPointVector;
        }
        void main()
        {
            ...
            cameraPosition = vec3(worldToViewTransform[3][0], worldToViewTrans
            cameraToPointVector = normalize(positionIn - cameraPosition);
            ...
        }
    fragmentShader =
        in VertexData
        {
            ...
            vec3 cameraPosition;
            vec3 cameraToPointVector;
        }
        void main()
        {
            ...
            fragmentColor = vec4(toSrgb(applyFog(reflectedLight, -v2f_viewSpa
        }
```

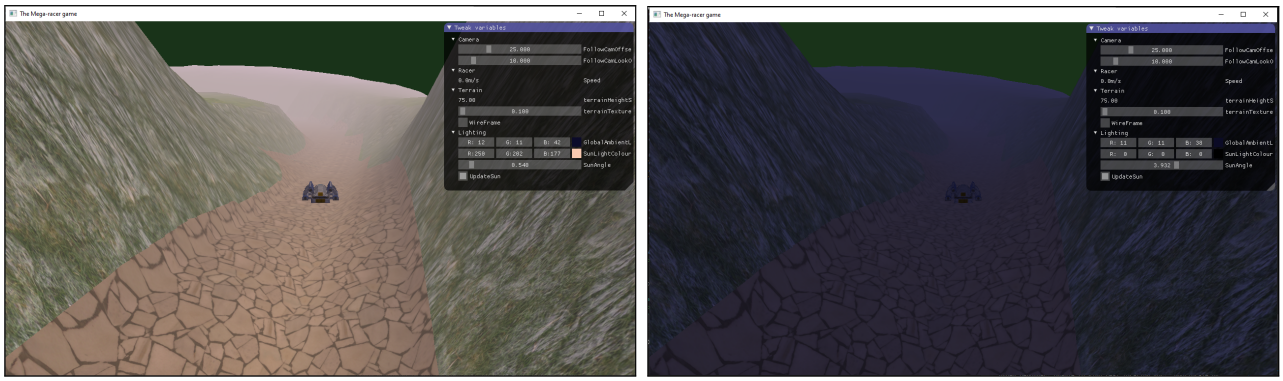

Figure 20: 2.2 - Height Fog

2.8 2.3 Props

(1) Create the Prop class Followed racer.py to start of with by importing all the same libraries, creating a Prop class and copying the variables from the Racer class. 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 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.headi
    rotationMatrix = lu.make_rotation_y(self.rotation)
    renderingSystem.drawObjModel(self.model, modelToWorldTransform * rotation
```

(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.t
    # Load rocks
    self.rockModel = ObjModel("data/rocks/rock_01.obj")
    self.rockList = self.loadPropList(self.rockModel, self.rockMax, terrain.r
```

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.

```
#mega_racer.py
from prop import PropManager
...
g_props = None
....
g_props = PropManager()
g_props.loadAllProps(g_terrain)
```

(3) Render all props

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

Again in the same area as the g_racer and g_terrain variables are rendered this render function will be called.

```
#mega_racer.py
def renderFrame(...):
    ...
    g_props.renderAllProps(view, g_renderingSystem)
```

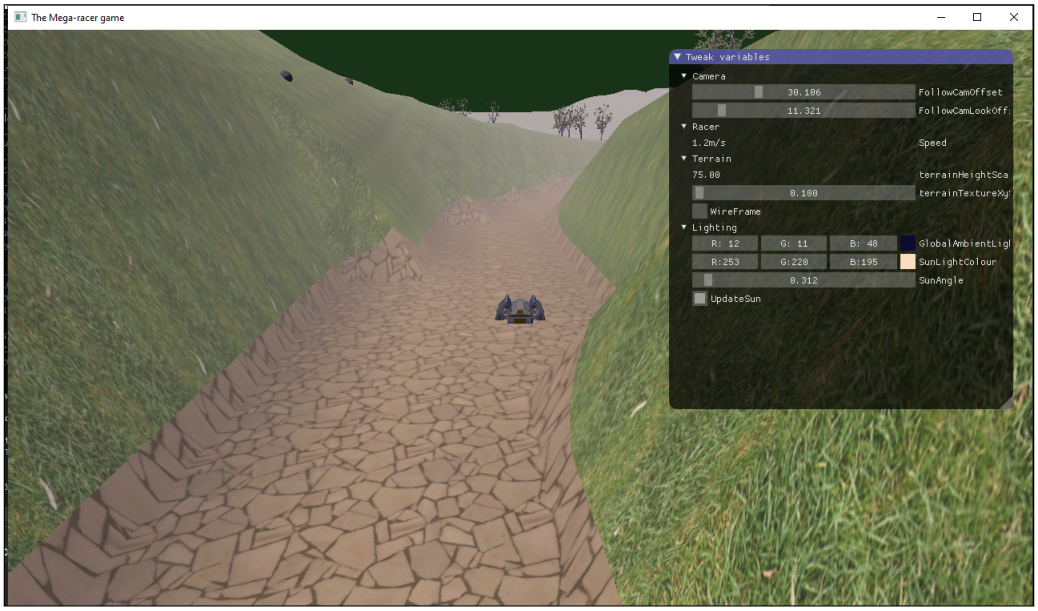

Figure 21: 2.3 - Props (trees and rocks)