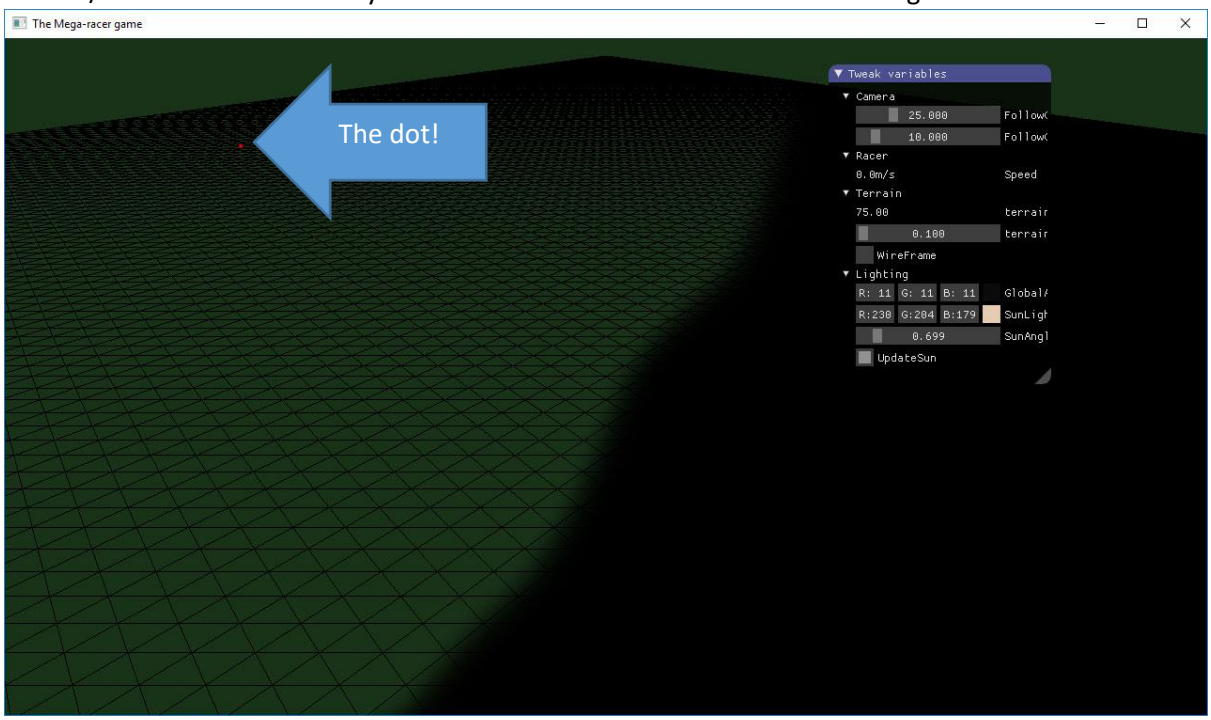
Mega Racer!

The aim of the project is to implement the *graphics* of a simple racing game that we provide the game logic for. There is already some placeholder graphics implemented so that you can see the game logic in action. If you start the file 'mega_racer.py' you should see a rather boring looking black field with a red dot (which you can drive around with the arrow keys). If you toggle the Terrain/WireFrame check box you will see that the field is made out of triangles.



The project, thus, is to make this all look a bit more interesting using the techniques we've learned in this course.

Make sure to read the general graphics project instructions available on Blackboard (in the item 'Summary') in the assessment section.

Some things to keep in mind

- The key result that your grade depends on is the report (not the code or exactly what you implemented)! So while you have fairly specific instructions here, document your implementation process, including being clear about when and why you use code from the labs (which you may do).
- You still need to write and submit a (very) brief project proposal. In addition to clearly stating that you are choosing to do the set project, you should outline the graphics techniques you plan to implement and why you think these are important or appropriate (there is no right or wrong here!).

Overview of the skeleton code

The skeleton code provided contains the necessary 'game' logic (you are not expected to invest any time in improving this). There are three main modules in the program, which take care of different aspects.

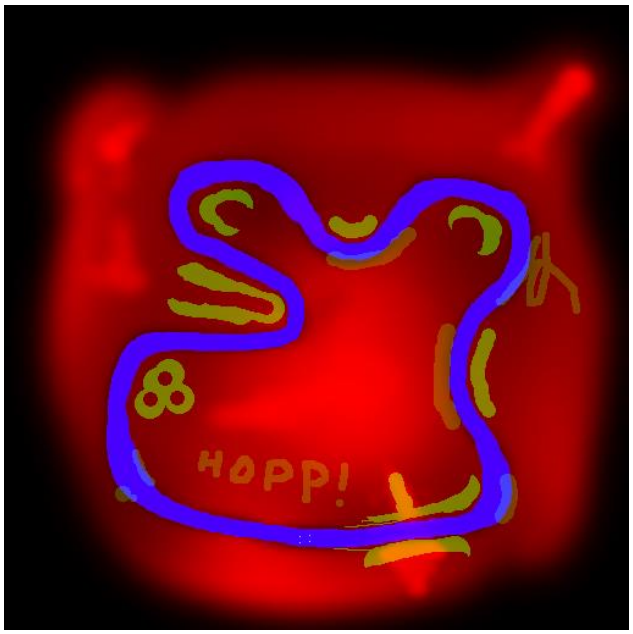
General design decisions

World space is defined as having the z-axis meaning 'up' – this departure from the labs is purely to reflect the fact (described below) that our race track data is described as a 2D image, where x and y naturally map to the image extents. This is nothing more than a convenience, and we might equally well have mapped the image to the x and z coordinates and used 'y' as up.

Coordinate units are given in metres. For scale, the track terrain is about 1 km2, and a racer is a couple of metres long.

Terrain ('`terrain.py`')

The terrain module deals with loading and drawing the terrain for the game. To make modelling easy the terrain is defined as a normal RGB image made in Photoshop. As shown here to the right, the red channel is used to denote height (more on this in Section 1.1), and the blue channel is used to encode the type of terrain.



Currently there is only two types of terrain, 'road' and 'rough' and they are used by the game logic to set the maximum speed for the racer.

The green channel is used to denote positions where things in the game are supposed to be placed. For example a green colour value of '255' is used to define start positions for the racer so we can make sure it starts on the track. The other values painted indicate where the reference demo exe places trees or rocks.

If you want to change the track, or make your own, there are a couple of things to be mindful of. In particular if you change the size, you need to make sure that filtering does not alter the values that are used for specific things (height is usually fine, but not the others).

Take your time to try and understand how the Terrain is implemented. The load function creates both the vertex data (positions and normals) as well as the topology (3 indices for each triangle that references the vertex data). Note how the image data is retained such that we can query it later to figure out the type of terrain.

The terrain has a special vertex and fragment shader, since the terrain is handled sufficiently differently from geometry loaded by the ObjModel class. Spend some time getting familiar with how the shaders work. Note that the fragment shader makes use of the common shader code declared in the rendering system (see below).

Racer (‘racer.py’)

The ‘Racer’ class implements the logic and rendering for the racer. It is much less complex than the Terrain as it does not manage its own geometry (we’ll rely on the ObjModel class for that). The vertex and fragment shader we use for this is in the main program (in the ‘RenderingSystem’ class) as it will be shared across all instances of ObjModel (if you chose to add some Props later).

However, it has a new function ‘update’ which is responsible for the update of the craft in response to user input. It is called by the main program each update step of the simulation. The parameter ‘dt’ is the number of seconds since the last update was called and is used to ensure movement is (mostly) consistent regardless of frame rate.

Main Program (‘mega_racer.py’)

The main program, implemented in ‘mega_racer.py’ declares a number of global variables to represent the state of the game, for example, an instance of the Terrain and Racer classes each and variable to represent the camera. The main loop towards the end of the program repeatedly calls the ‘update’ and ‘renderFrame’ functions until the window is closed.

The class ‘RenderingSystem’ is a helper class to be able to collect functions and data to do with rendering in one place that we can pass around to the other modules. A key reason for it to exist is to store the shared fragment shader code (in ‘commonFragmentShaderCode’). This is code that we wish to include in all our fragment shaders, to avoid code duplication and ensure consistency. OpenGL shaders do not have any support for modules, so we would have to build our own system for pre-processing the shaders. However, for our simple project a single shared string containing the shared code is sufficient. We only do this for the fragment shader since it is where most of the work will go, the vertex shaders are relatively simple. The function ‘setCommonUniforms’ helps setting a number of the global variables in the program as uniforms, for example those to do with lighting, and also ensures that the transformation matrices are set. Recall that we set the uniforms for each program object, and OpenGL ensures they are set for all shader stages that use them. The RenderingSystem also contains a shader for use with obj models and a convenience method for drawing ObjModels ‘drawObjModel’ that helps out by setting the common uniforms and the correct shader.

Deliverables

Tier 1 – Baseline grade (i.e., grade 4-5)

The deliverables in this tier can be seen as requirements for a passing grade and are almost only things we have already done in the labs. Implementing these will provide basic graphics to the game.

Note: as stated above (I know I'm going on about this a bit), the resulting grade will *depend primarily* on the quality of the report you write, *not* the visual quality of the graphics produced (or even the code)! You are required to hand in the code, but style and so on will not be part of the grade.

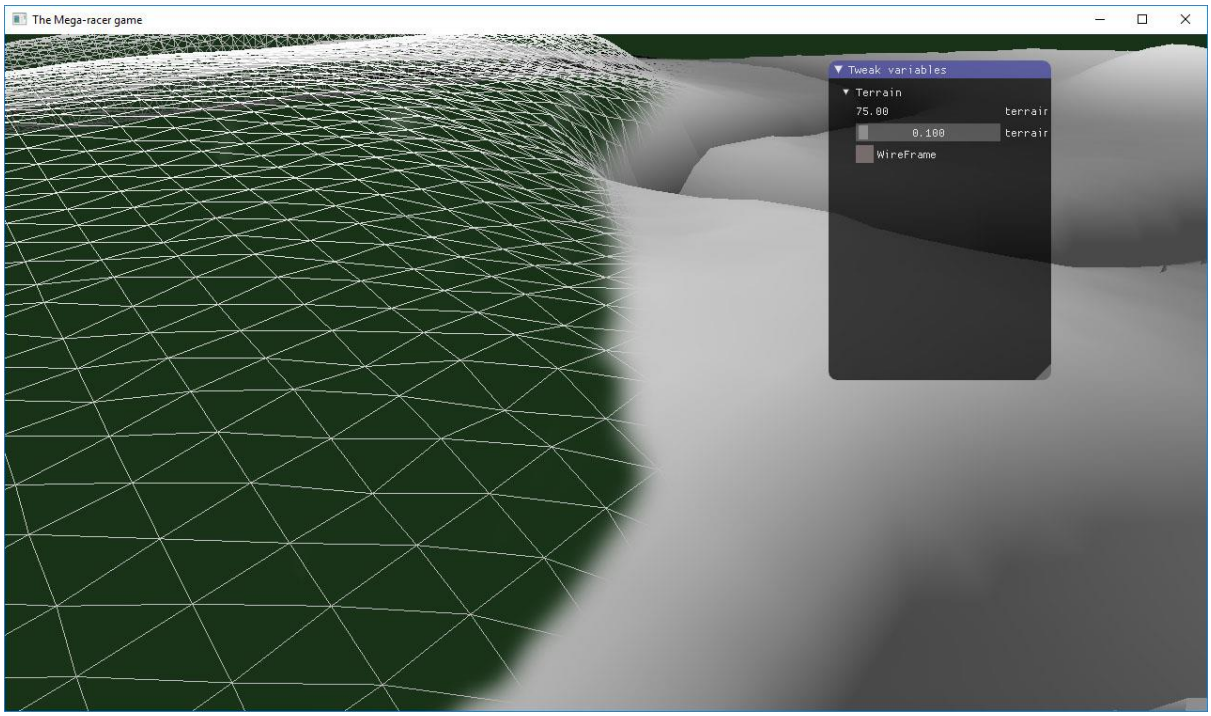
1.1 Scale the terrain grid

The terrain / ground is currently flat. The terrain/track geometry is defined in the image 'data/track_01.png' together with constants in the `Terrain` class, which is loaded as part of the game setup ('`g_terrain.load(…)`'). The variable 'xyScale' together with the x and y dimensions (in pixels) determine the size of the terrain, e.g., an image size of 128 x 128 and a Terrrain.xyScale = 8.0 gives a terrain with extent of 1024 x 1024 metres (Note that the reference C++ implementation uses a 512 x 512 pixel image and xyScale = 2.0, for the same dimensions but higher resolution terrain geometry).

The red colour channel contains the height at each point, represented using a byte, i.e., an integer in range [0,255], we interpret 0 as the lowest point, and 255 as the highest.

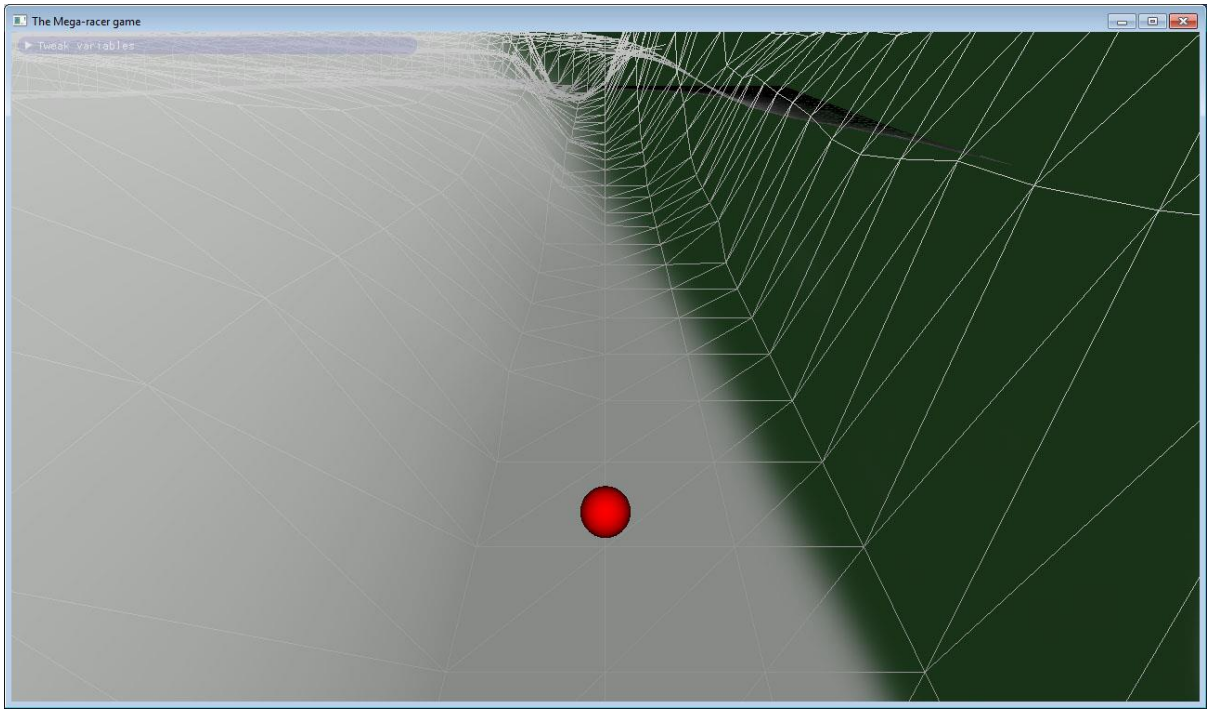
The terrain mesh uses a grid of vertices, where each vertex corresponds to one pixel. Currently, the x and y coordinates are set to one metre increments, and z is set to zero. Use the scale factor '`heightScale`' (a member of the `Terrain` class, so accessed as '`self.heightScale`' inside Terrain member functions, such as load) to calculate the z coordinate for each vertex.

Using the default values, the result should look like the below (which is a composite of using wire-frame drawing or not). The fragment shader uses the height, or world space-z coordinate, as the colour (normalized to the range [0,1]). Also, you will need to uncomment a line last in '`Racer.update`' to make the racer follow the terrain height.



1.2 Set up a camera to follow the racer

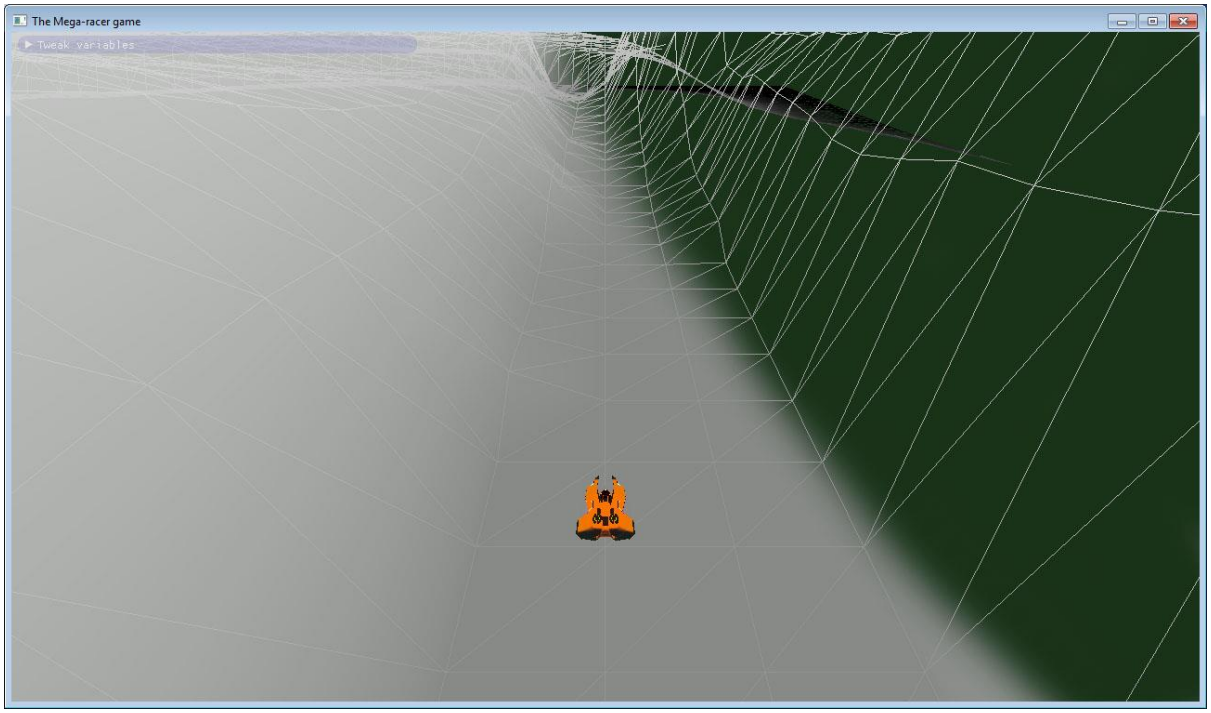
The view-point is currently fixed above the world using a hard coded location and direction. In the update function add code to position the camera such that it follows the racer at an offset behind and above, and looks at the racer. The variable 'g_followCamOffset' is intended to represent the distance behind and above that the camera should be placed. Additionally, the 'g_followCamLookOffset' variable is intended to be used to offset the look target above the racer – this creates a more flexible view, when the racer is not dead in the centre. As seen below, this makes it possible to keen the view fairly close to the racer and still show a reasonable distance ahead.



The result should look something like the above image. Note how the view-point has shifted from staring at the centre of the world to the starting location of the racer. The racer however, looks somewhat dull.

1.3 Place and orient a model for the racer

Chose a forwards direction (your own convention) and create a model-to-world transformation to position a model at the location of the player racer. The transformation should be such that it aligns with the forwards direction of the game object.



The model should be scaled, centred, and rotated such that it is consistent with the conventions chosen. The best way to do this is using modelling software such as Blender, which is free and can be run out of a single folder (no installation needed). Maya is a commercial program that is available to students for free (with some restrictions). If this appears too complicated, the model can also be scaled and rotated at run time though this makes the code more complicated and is best avoided. The racer shown in the picture below is provided in the skeleton, in the file 'data/racer_01.obj', though the 'data/racer_02.obj' is a bit more low-poly and might speed up loading time in particular. Note that it does not share the conventions for up and forwards shared with the world space we defined. You can find a lot of other models at opengameart.org.

The model should be loaded using the obj model loader provided. Note that the function 'renderingSystem.drawObjModel' is provided to help setting the common uniforms and provide a single shader that can be used for all ObjModel instances.

Document your choices and thoughts.

Tip: In lab_utils there is a very convenient function for this kind of purpose called 'make_mat4_from_zAxis'. It is similar to what was described in Lecure #2. Especially useful if you by some chance happen to pick the z-axis as the model-space forwards direction.

1.4 Texture the terrain

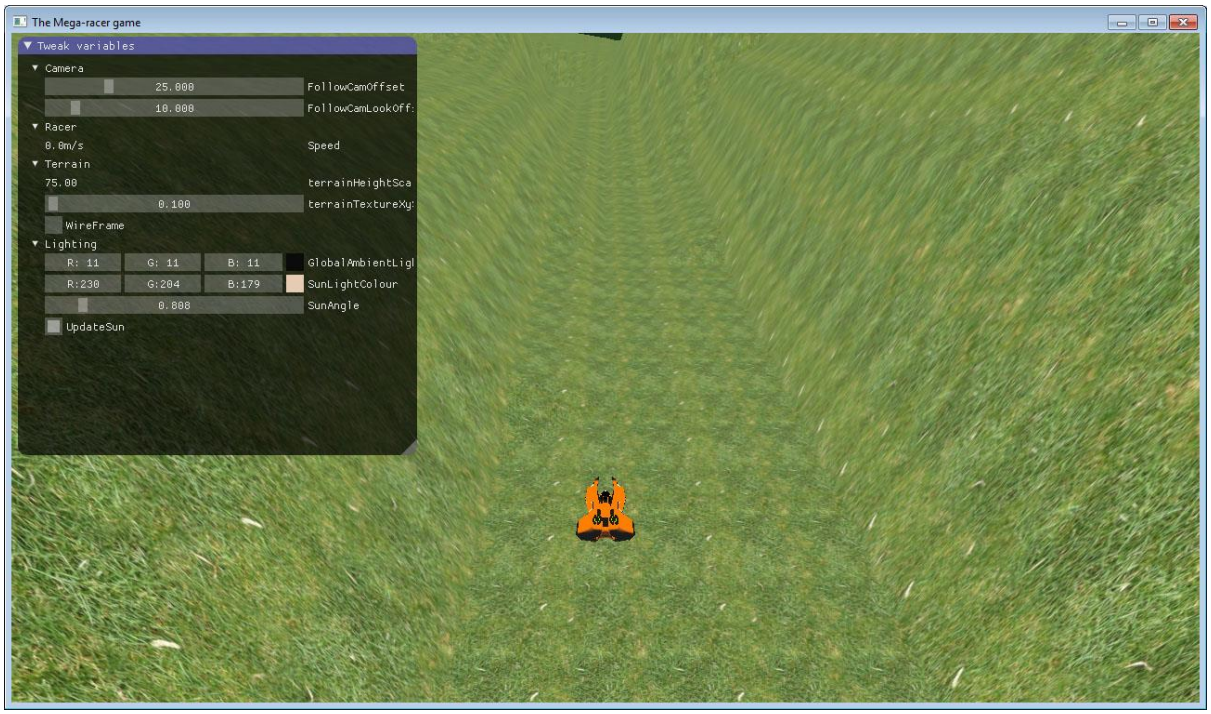
The terrain looks a bit(!) drab. Load the grass texture ('data/grass2.png') and create an OpenGL texture. Ensure the wrap mode is set to repeat as we wish to tile this texture over the whole terrain.

Add a sampler to the terrain fragment shader such that you can sample the texture. The terrain vertex data does not contain any explicit texture coordinates, but we don't need it since the x and y *spatial* world-space coordinates are completely regular – use these to sample the texture in the fragment shader.

The texture should repeat *every* metre, since we are using world space coordinates as texture coordinates and the texture space has range [0,1]. Use the member variable 'textureXyScale' which is already set as a uniform in the shader 'terrainTextureXyScale' to scale the texture coordinates.

Tip: You can call 'ObjModel.loadTexture' to load a texture and create an OpenGL texture.

The end result should look something like the below, with the default scale factor of 0.1, which means the texture should repeat every 10 metres. It does still not look exactly fantastic, but we're on the way!

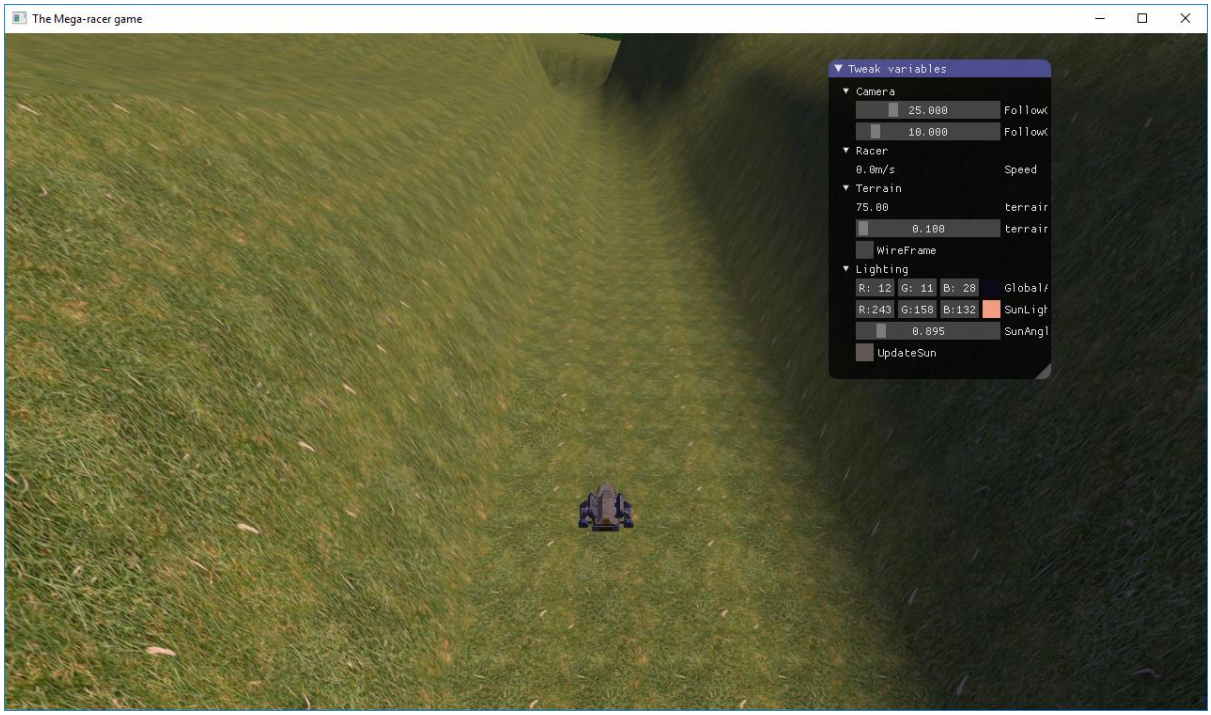


1.5 Lighting from the sun

The sun position is given in the variable 'g_sunPosition' and is updated in the game logic to orbit the flat world (just like in reality!). Implement a lambertian shading model in all the shaders (that is, the Obj-model shader and terrain shader), and make sure it is looks consistent. If your shading-model only has a lambertian term, things will be pitch black when not facing the light – introduce the appropriate lighting term to fix this in the simplest way possible. To ensure all shaders behave in the same way, implement the shading in the skeleton function '`computeShading`' in '`RenderingSystem.commonFragmentShaderCode`' – then you just need to call this function from each of the shaders since they already include this block of code.

The sun light colour and intensity is represented in the program 'g_sunLightColour' and this is already provided to the shaders as the uniform 'sunLightColour' (set by the 'setCommonUniforms'). The sun light varies depending on the angle of the sun, and is configured using the key-frames in 'g_sunKeyFrames' to produce a nice time-of-day effect. Most particularly, the intensity is zero when the light is below the horizon which takes care of a lot of weird looking artifacts.

The below picture shows the result (I've swapped the racer for a more low-polygon one now this example). Note how the shading is consistent on the racer and the terrain and not black in the dark regions due to that other lighting term… ☺



Tip: A common source of problems is that the *spaces* get mixed up. The sun is defined in **world space**. Make sure all parameters to the shading calculation are in the same space (whichever it is, usually **view space** is the most convenient for shading).

Tip: Be sure to check for when the light is behind a surface, by clamping as appropriate. Remember the dot product produces negative values for vectors with an angle larger than 90 degrees. Intuitively this should not happen for visible geometry in view-space, BUT it does because the shading normal is not the same as the geometry normal.

Tip: In your report, be sure to discuss the possible extensions and what could be done to make this a more complete model. What lighting effects are ignored for example?

Tier 2. Better grades

These deliverables require a bit more effort, and sometimes some own research. To get a grade above base line (i.e., above grade 5) you **must have a good go** at a few of these. The key thing is to try, and **document** the process and end result using appropriate terminology. A well-documented failure is almost certainly worth *more* than a poorly documented success! There is no exact definition of how many you need to do for a higher grade, they are not equal and you can spend more or less time exploring and document each suggested item. As a rule of thumb two or three ought to do it. You should pick a couple you think interesting and fun and then make sure you understand how it works! Don't hesitate in asking for help during the lab times if you get stuck.

> *A well-documented failure can be worth more than a poorly documented success!*
> *-Steve Gates*

Note: Some of the deliverables are considerably more difficult than others. Those are marked with an asterisk *. This should not discourage you from trying, but be prepared to have to do your own research.

2.1 Improve Terrain Textures

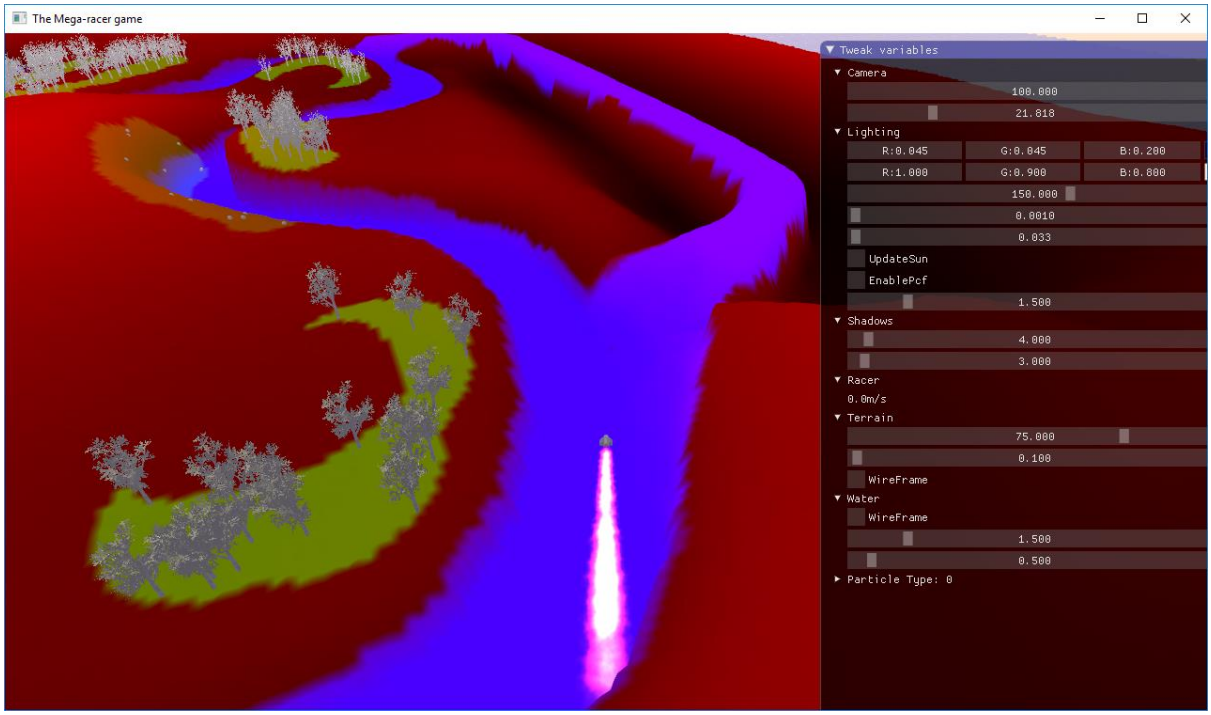
The terrain *still* looks a bit drab. There are two main things missing to which are implemented in the demo. The first one is a selection of a different texture based on the terrain material, this also provides an important visual cue to the player by showing where it will be fast to race.

The second one is a height and slope based texture selection to create variation in the non-road part of the terrain. This is actually the simpler part. Just load a few more textures in the same way as the grass texture (the demo loads "data/rock 2.png" and "data/rock 5.png" for the high and steep parts respectively), and create samplers in the fragment shader (again in the same way, just remember to use different texture units). To blend the textures in the shader, use the height to mix the high texture over the grass texture. To decide when to use the 'steep' texture, use the world space normal of the geometry, if it is near horizontal the surface is steep. You can pick a threshold value of when to start blending in this texture.

To make the road have its own texture we need to add yet another texture in the same was as before (demo uses "data/paving 5.png"), but to select when to use it in the shader we need to know the type of terrain. This is somewhat less straightforward as this information is not available to the shader (it is only available to the game logic). To address this a straightforward way is to create a new texture containing the map data itself, and then we can access the blue channel where this information is stored. Loading the texture is not much different from the others, except that since this texture stores data and not colours we should not load it as an SRGB texture (see comment in ObjModel.loadtexture). However, to sample it you must use texture coordinates that are normalized in the range 0-1 across the whole map. This can be computed from the world space x and y coordinates in the vertex shader and passed to the fragment shader.

Tip: To ensure things are working ok, you can modify the fragment shader to just output the terrain type as a colour. This is really a key way to debug shaders, crude but fairly effective. The below is a shot from the demo showing the terrain data used as texture (and no lighting or fog applied):

```
fragmentColor = vec4(terainDataSample, 1.0);
```



2.2 Add Fog

Fog is an important distance cue and gives a sense of scale to the scene. Implement distance-based for as described in the lectures. Use this source for more details. The reference implementation implements the distance (but not height-based) fog and uses an extinction coefficient 'g_fogExtinctionCoeff = 0.001f'. The other tricky part is the fog colour. Since fog done in this way is a rather big hack, there is no right answer, but intuitively it ought to be some combination of the direct sun and ambient light (just ambient leads to a very dark fog, but might be good for a dungeon).

For extra challenge, cred, and visual quality you might also implement the more advanced height based fog (also described in the blog linked). With this and an interactive fog height you should get some cool looking effects across the race track.

Implementing shadowing of the volumetric fog is not extremely hard, if you already have working shadow maps. You simply walk from the sample point in view space towards the camera and take samples of the shadow map visibility and add up the contribution of the segment. It is somewhat tricky to get right, and is typically rather computationally intensive if you want nice-looking volumetric shadows with this simple method.

2.3 Add Props!

The terrain is empty, oh so empty. As mentioned the Terrain already pulls out locations that are classified according to type (from the green channel). The provided ones are trees and rocks, but you are free to create any you like, using the remaining 251 values possible.

Create a new class 'Prop' ('Prop' is a term from the movie industry and refers to stuff you place in a scene that are not part of the scene or actors outfits, I suppose) to manage the loading and rendering of the props. The Props should have a position and a randomized rotation around the up vector (for variation). The Prop class ought to be a lot like the Racer class, except simpler since it doesn't need to be updated.

The idea is to select randomly from the list of possible positions in the Terrain class and have a pre-defined maximum number of each type. The created props should all be kept in a global list for easy rendering.

There is a model for a tree and a rock provided to get you started. In the demo implementation the models 'data/trees/birch_01_d.obj' and 'data/rocks/rock_01.obj' were used. Try to find some alternative models for each type and randomly select. Also try to find some other type of prop that might look cool and include it too, opengameart.org contains many useful models. You might need to load the model into blender or some other modelling tool to re-orient or scale. Also, I have noticed that the .obj models often provided at opengameart.org typically don't have a material definition file, unfortunately, which means you might have to re-export the model.

Note: You should make sure to load each unique model only once. Otherwise if you create 100 trees they will all have their own copy of textures and vertex data, which is highly inefficient. Often it is useful to have a 'manager' in between that can keep track of which models are loaded and ensure there is only ever one copy. The same really goes for textures or any other resource. Here it should be enough to load the model and the pass a reference to each prop that should use it.

2.4 Headlights

Implement a global spotlight and add it to the shaders. To make it work like a headlight set the position and orientation to using the racer position and orientation, it is a good idea to angle it down somewhat. One way to think about this is to define the position and orientation in the racer's object space and then transform that to world space.

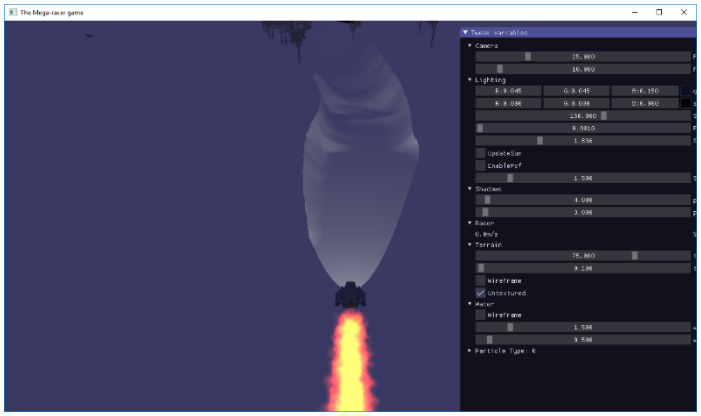


Figure 1. Headlight in the demo implementation. Visible clearly at night when there is no direct light from the sun. Texturing on the terrain is turned off.

A spotlight is a cone defined using a position, a direction and an angle. To test whether a point is within the cone can be done using a dot product. The spotlight should also have a colour. To look nice and reduce the x-ray effect you get from not having shadows, a range and falloff is also a good idea, the demo uses a linear falloff (highly unphysical!).

2.5 (More) complete shading model

Add other components to the shading model, to make it match what is described in the lectures.

2.6 Add a particle system

Fire and smoke coming out of the racer. More details to come!

2.7 Add Shadow Maps

To properly see spatial relationships, for example, how high the racer is, shadows are essential. The industry standard for real-time rendering is to use shadow maps. Create and render a shadow map for the sun, and use this in the terrain shader (at least) to check for occlusion of the light from the sun (i.e., shadow). Only render things that are supposed to cast shadow into the shadow map – for example if you added water it should not be drawn (but is would make sense for it to *receive* shadows).

Setting up the view is no more complicated than the camera used for the primary view. The sun has a location (since we use a point-light for this) and as a direction we can aim at the racer (which is at least where we know we want shadow, even if we'll waste a fair bit behind the racer…).

You can play around with some of the shadow map parameters in the reference demo implementation provided.

A fairly complete shadow map tutorial that can be used to work from can be found at opengl-tutorial.org (tutorial 16) – NOTE however that the method of avoiding depth-aliasing they use is not recommended. They add a small bias in the shader, which is cumbersome and inflexible. Instead as mentioned in Lecture #7, it is better to use the built in OpenGL offset and bias support when drawing the shadow map.

```
g_polyOffsetFactor = 4.0
g_polyOffsetUnits = 0.75
…
glPolygonOffset(g_polyOffsetFactor, g_polyOffsetUnits);
glEnable(GL_POLYGON_OFFSET_FILL);
```

Oh, and totally ignore the idea of using the backfaces! It sounds nice in theory but **just does not work!** Not sure why this idea refuses to die peacefully.

You might also want to have a go at implementing Percentage Closer Filtering (PCF). This just amounts to taking a number of samples around the sample position and averaging the result.

Tip: As a debug aid, to make sure the shadow map view is set up correctly, use its parameters to set the camera for rendering the visible geometry. You should then see the scene as seen from the sun. If you cannot see the some part of the world, it will not be drawn into the shadow map. Typical mistakes is using a too small **far** clip-distance.

Tip: The game sets the light intensity to 0 when the sun is below the horizon. To see how bad it would be otherwise, set the sun light intensity to 1 in the shader (or even in just return the result of the shadow map test, this is implemented in the demo toggle: the option 'ShowSmVisibility').The light will shine through the ground if back-face culling is used. However, turning it off will result in terrible *shadow-map aliasing*. The most sensible solution for this particular case is to dim the sun light as the sun nears the horizon, and such that it is zero when the sun is down. Another solution, which is already implemented, for the more general case is to ensure the geometry is *closed*, here one might put a large polygon underneath the terrain facing down. When drawn into the shadow map the front faces cast more or less correct shadows on the land.

2.8 Level of Detail (LOD)

Why not put a few thousand bunnies in the map and apply LOD meshes? ☺ (yes I'll put in a bit more here later! Most likely, or just ask…)

2.9 Water

Re-use the code to create the ground mesh to create a water mesh at z = 1 and make it extend a fair way from the land. Create a new shader that can look more like water. Animate the water surface using a sine wave, or several stacked sine waves that depend on time and position to create moving waves. Add a variable to control wave height and set that in the UI. To make it look really nice, try to

find some references on ocean shading and see what could be applied in the shader. For example, changing the transparency and colour based on the water depth might be nice.

Getting water to look really nice can take a lot of work! As you can see from the demo, I gave up long before reaching 'nice'…

2.10 View-frustum culling

Implement basic view-frustum culling, to avoid drawing objects that are not visible on screen. If you have not implemented loading more props, then this is kind of pointless, so that is a requirement for this task. For extra points divide the terrain into chunks and perform culling on these as well.

The first step here is to create AABBs for the models, this can be done once when loading each ObjModel, or for each prop or chunk of terrain. Then you need to form the bounding planes of the view frustum, and test the AABBs against each plane.

What are your expectations on performance / efficiency? How did they hold up?

Tip: Remember that you need to do this on the fly for each view, and that shadow map rendering would use a different view than the main view.

*2.11 Physically based shading / High-dynamic range**

Change the shading model to be physically based. As sources of information make use of the SIGGRAPH courses on physically based shading and the Frostbite presentations. Be aware that implementing this correctly is a complex task, so try to limit the scope of what you attempt. Just ensuring you can set the sun light and exposure to physically realistic values and control the night-time look through the exposure rather than fudging the light levels should provide sufficient challenge!

Making it actually look nice is also a process that involves modifying the content, in particular such that the textures and materials represent physically accurate reflectance amounts.

One important aspect is exposure. Since the frame buffer has a very limited range, just like a photographic image, you must control the exposure. Typically this is done as a post process and computed automatically based on the brightness of the frame buffer content. However, to simplify the implementation you are instead advised to manually control the exposure and apply this last in the fragment shaders.

2.12 Cube-map reflections on the racer

*2.13 Real-time cube-map reflections**

*2.14 Screen-space ambient occlusion**

*2.15 Deferred shading**