University of Milan

Department of computer science

Report of the Real-time Graphics Programming project

Author:
Diego Vallauri

Contents

1 Introduction and purpose of the project

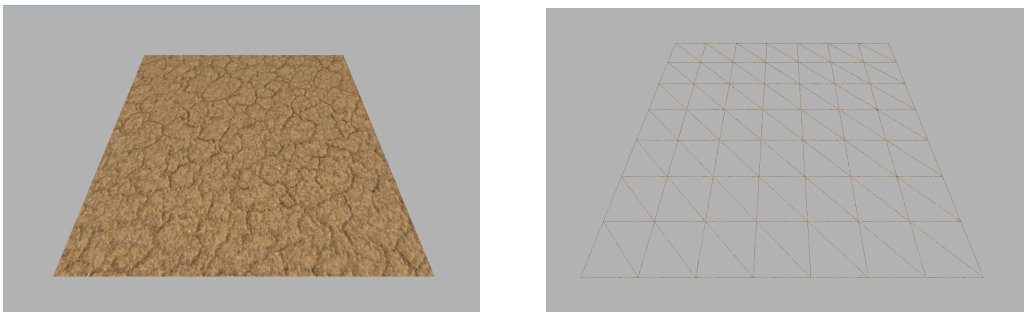
The aim of this project was to create an infinite procedural terrain with other aesthetic elements such as water and trees using **OpenGL**. In the following chapters I will describe the different components of the project and how I have implemented them.

2 Terrain

The terrain is the main topic of this project so let's start to introduce the different components.

2.1 Cell

The main part of the terrain is a cell, a cell is basically a square whose sides have a fixed size and equal number of vertices. As you can imagine, a cell at first is just a plane, as large as the multiplication **Size x Size** and with **Vertices x Vertices** number of vertices. Creating a plane is not a difficult task, so I didn't load a model but I generated it manually. First, I created the different vertices and stored them in a vector, then I generated the different indices to create the triangles of the mesh.

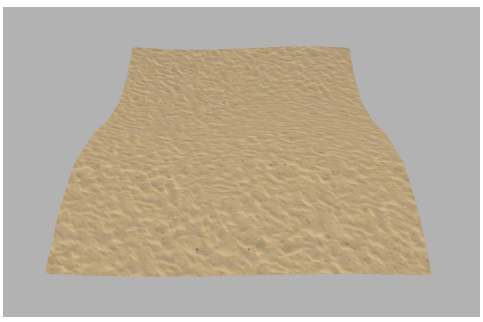


(a) Plane.

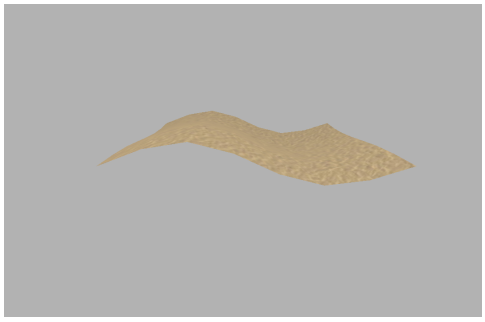
(b) Vertices, indices and faces.

Figure 1

Obviously I didn't want flat ground, so I had to introduce some noises. For this task I relied on an existing library called **FastNoiseLite** which implemented different types of noise function. You can set various parameters, such as the type of noise, the seed and the frequency. In this specific case I choose a Perlin noise type, a random seed and a noise frequency of 0.008. By changing these parameters you can get different terrains. In addition, the maximum height of the terrain must also be specified. Then, at the time of creation, each vertex generates a noise that will be multiplied by the maximum height.



(a) Front plane.



(b) Side plane.

Figure 2

For any other information about a cell, you can check the **terrain.h** file.

2.2 Terrain

To create a large terrain, I put several cells together, specifically I create a square of cells. Basically I just have to choose the number of cells per row/column and put them close together. What I get now is something pretty awful, because there is no correlation between the height of the adjacent cells but it is totally random.

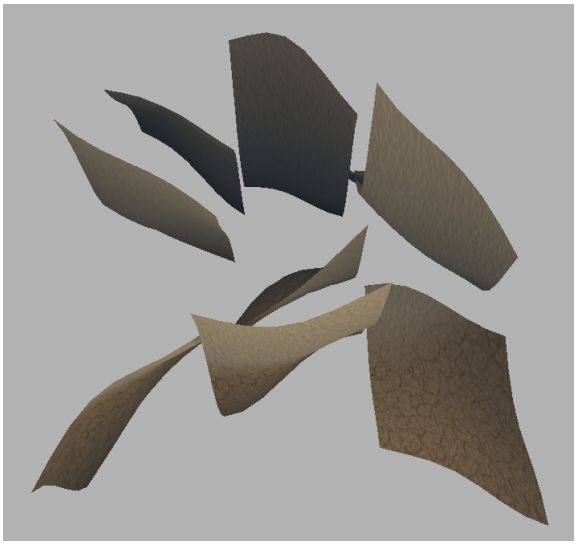


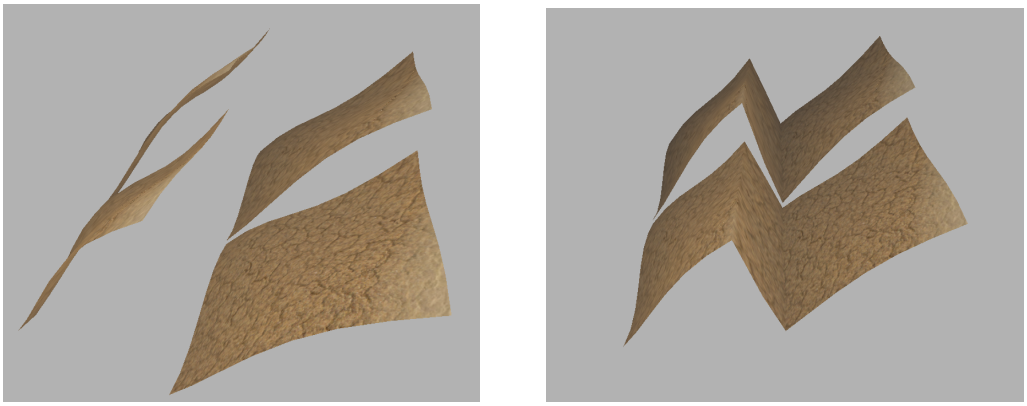
Figure 3: This image represents 9 cells (3 per row) with random heights.

2.3 Blending

The simple solution is to use a blending function between two adjacent cells. I can summarize this process in three parts.

2.3.1 First Part

Since I am doing this process on adjacent cells, they obviously need to reach the same height on the common side. The simplest way to accomplish this, is to use a simple lerp function that averages the heights of two common side vertices and assign it to them.



(a) Before. (b) After.

Figure 4

2.3.2 Second and third part

The second and third steps are actually the same, the only thing that changes is the cell that is used. In particular, it is necessary to propagate the height update obtained on the "hinge" side. We use again a lerp function but this time it's weighted based on the distance from the melt side. It works like this:

- I apply the blending from the melt side to the opposite side.
- I only change the value of the vertices farther from the "zipper" side. (Otherwise, the two cells will not be connected anymore).

- The blending is weighted, the greater the distance between the vertex and the "hinge" and the less propagation will affect the vertices. The most important thing is that the opposite side will not be affected by the blending as it is very important for connecting more than two cells.

As it is not very easy to explain the process, I have created the image below which I hope will make the process clearer for you. I also write the formula (although it's not that important) where I calculate the weight of the lerp function based on which vertex I am in at the moment (the **i** variable is a counter in the for loop). I also write the blend value with four vertices for each row / column in correspondence of each column. As you can see from the example, the first red vertex mixes 0.4 of its height with 0.6 of the new one, in fact it is closer to the hinge, so the blending is greater. Instead the second red vertex, since it is on the opposite side, will not be affected and its value will be the same.

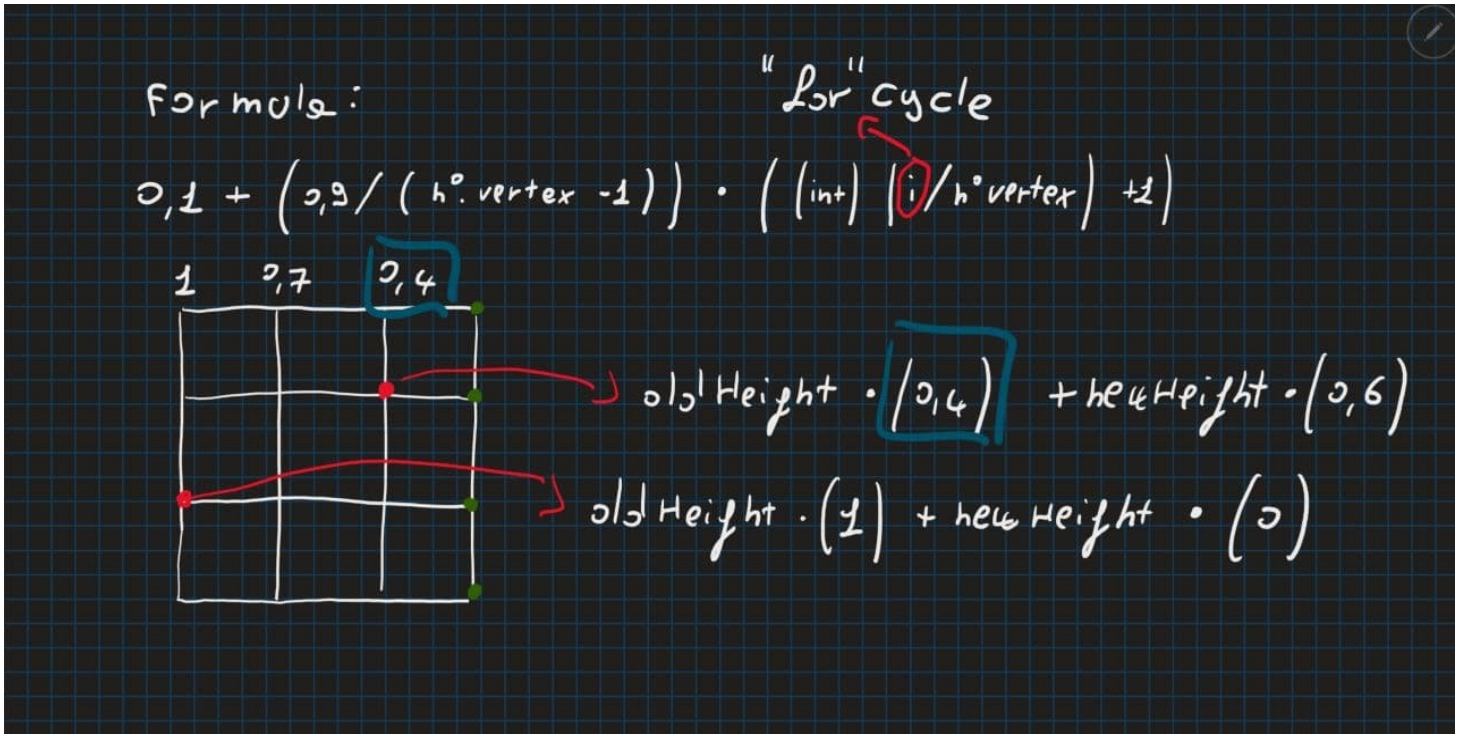
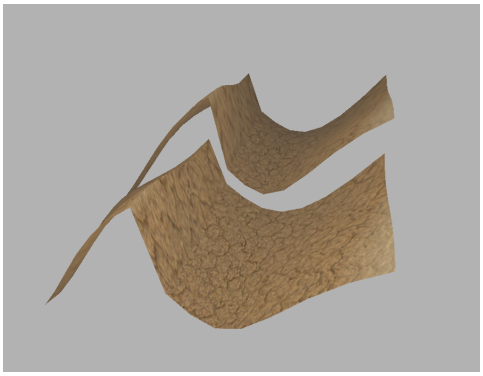
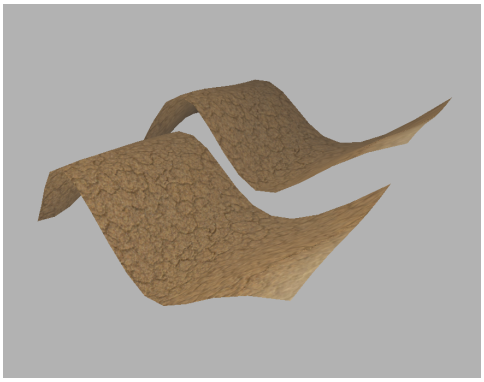


Figure 5: Visual explanation of the second/third part of the blending.

So we first apply this propagation to one of the two cell and then we do it for the other one (which is why the second and the third part are nearly the same).



(a) Second part.



(b) Third part.

Figure 6

2.3.3 Extra details

I need to explain two more things. The first is why I want to keep the opposite side unaffected by propagation. Mostly in normal terrain, there will be another cell attached to the opposite side where I have probably already applied the blending process. So I don't have to touch the hinge, otherwise the two cells will separate and I'll have to repeat the process for a potentially infinite number of times.

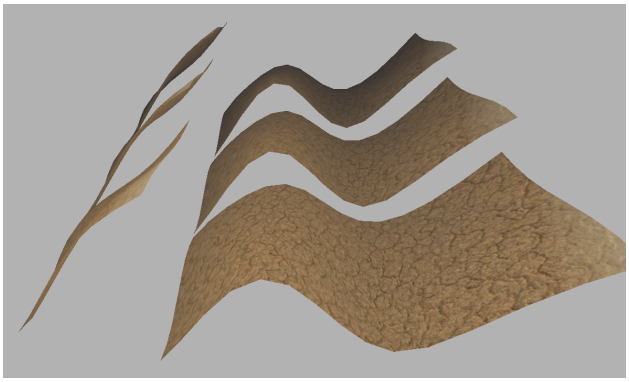
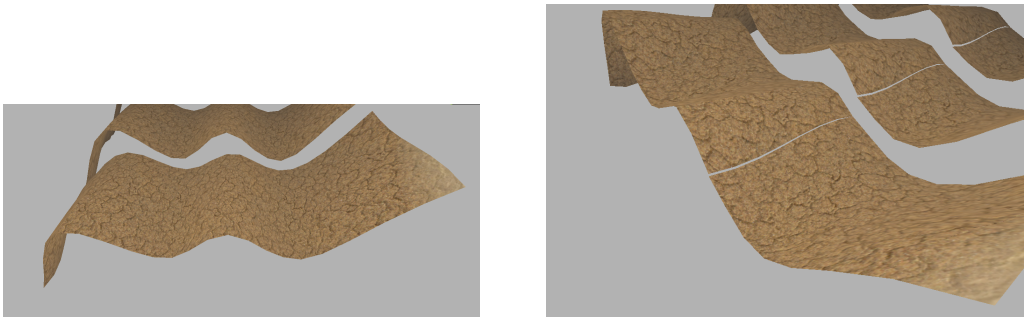


Figure 7: Before the blending of the left terrain.



(a) Good blending.

(b) Bad blending

Figure 8

The second thing is that I also have to interpolate the first cell with the last for each row / column. This is because my terrain will be infinite, so if I move a row / column of cells to the opposite side I need them to match the cell in that part. (Explanation of the moving terrain in the 2.5 section).

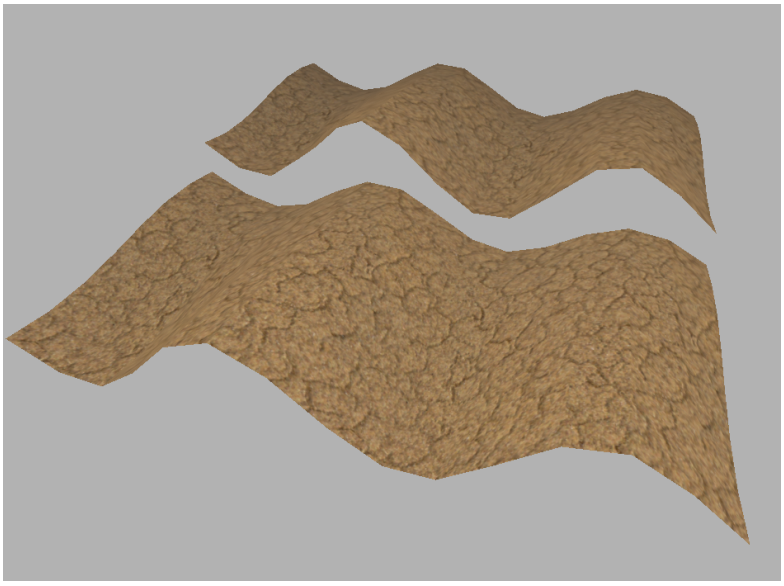


Figure 9: Horizontal blending between two cells. (Compare it with figure 6b)

2.4 Final blending

I have now blended two cells horizontally, but of course I need it vertically as well. The process is perfectly the same, the only thing to watch out for is not to mix the two interpolations and, before starting the vertical one, you must wait the end of the horizontal or vice versa.

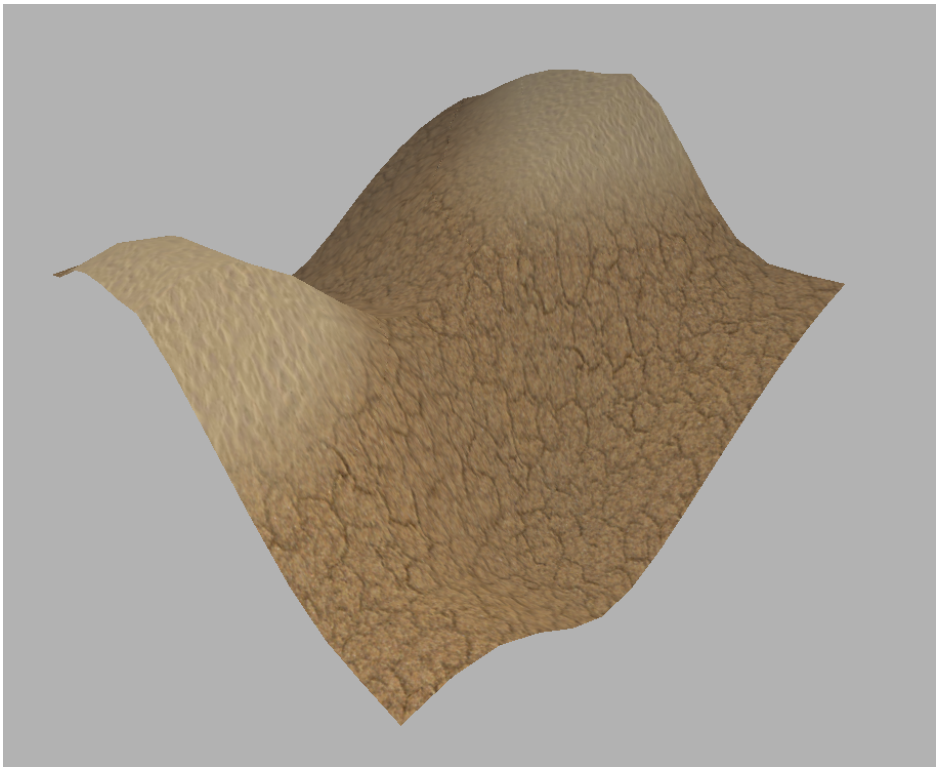


Figure 10: Example of horizontal and vertical blending between four cells. (The seed of the cells is different from the one used in the previous example).

2.5 Infinite terrain

I want my terrain to move with me, so that it gives the illusion of being infinite. As I said before, our terrain is made up of a fixed number of cells in each row / column so I need to figure out when is the right time to move a cell / column. To explain it, Let's see the image below.
Basically to move a row or a column, I checked when I overlapped another cell and based on the side that I overlapped, I moved a column or row on the opposite side. In the below case I am moving forward, so when I reached the end of the middle cell I move the last row to the first position. This is why I need to blend the last row / column as well, so they continue to match perfectly. In practice what I have to do is simply memorize the indices of the first cell of the last column and row (to get the first I just add the index of

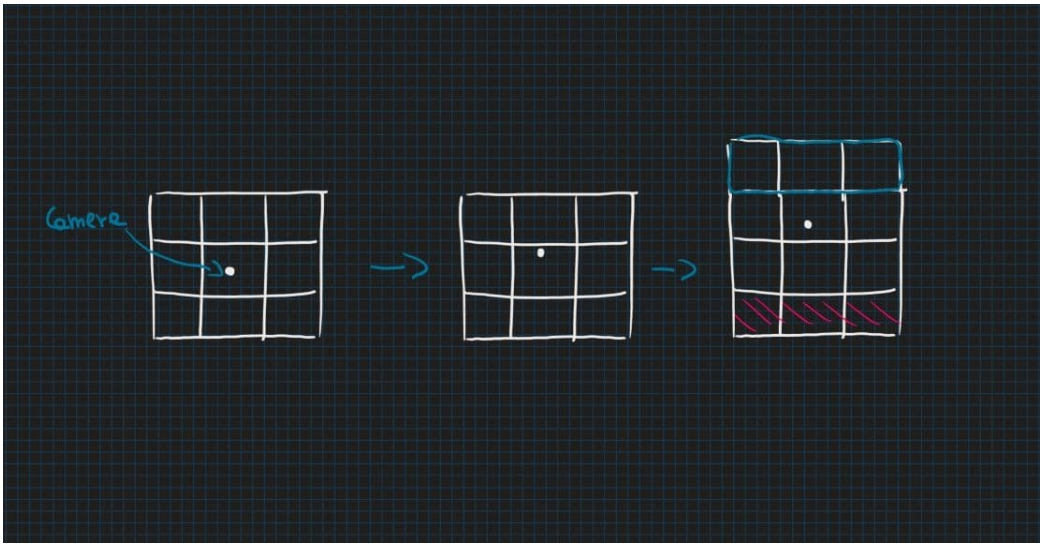


Figure 11: Infinite terrain moving schema.

a column / row and apply the module operator) so that I can change it real time and I always know which cells to move. Obviously the position of the camera is in the central cell. (To keep things simple, I assumed that each row and column had an odd number of cells, so there is always a unique center cell.)

2.6 Calculate the terrain normals.

To calculate the normal of a particular vertex, I used a simple method called the "finite difference method" which allowed me to calculate the components of the difference (x, y, z) in a very simple way. Let's start with the x component. I calculated the height near the selected vertex shifting the position in the four direction by a small amount. I first computed the left and right heights, then I subtracted the left one with the right one and obtained the x component. For the z component I did the same but instead of left and right I used the bottom and the top. Finally I set the y component to 2.0. I leave below the simple formula.

$$normal = vec3(heightLeft - heightRight, 2.0, heightDown - heightUp) \tag{1}$$

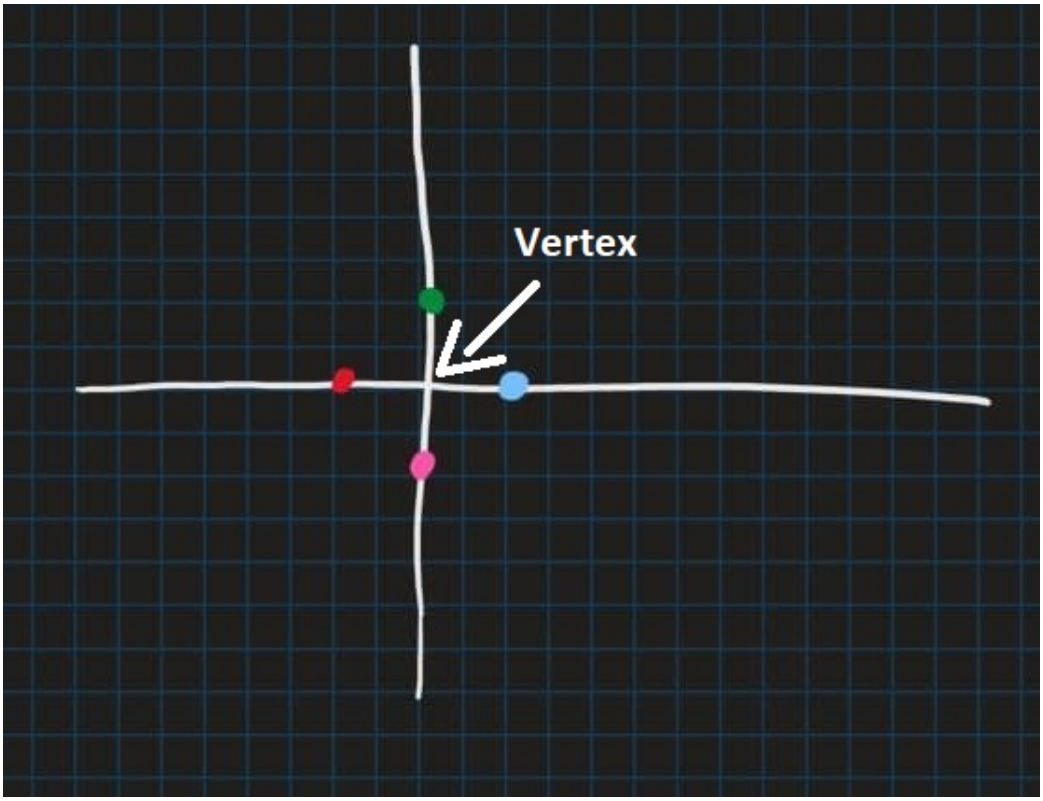


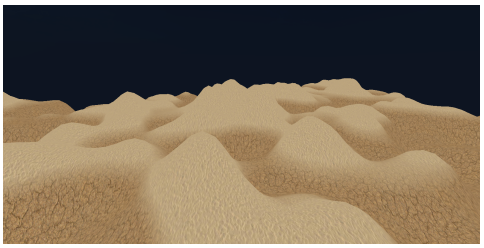
Figure 12: Color of the heights considered to calculate the normal in the central vertex.

After calculated the normal vector, I normalized it to create a versor.

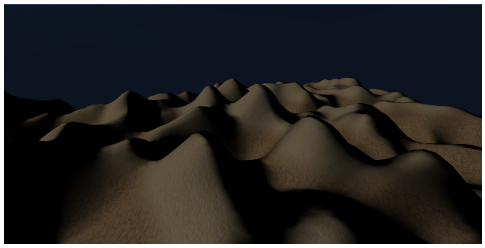
2.7 Terrain shader

2.7.1 Lambertian light

Inside the shader there is implemented a simple light model, the Lambertian one. Since it is very simple (the aim was only to check if normals were working, not to create a complex light model) and it isn't different from the one implemented in class, so I don't give any others details.



(a) No light.



(b) Lambertian model.

Figure 13

2.7.2 Multi-texturing

Instead of assigning a single texture, I used a texture for vertices below a certain height (e.g. up to heights of 15), then I used another texture for vertices above a different height (e.g. for heights greater than 115). For the central vertices I interpolated the two textures based on the height value.

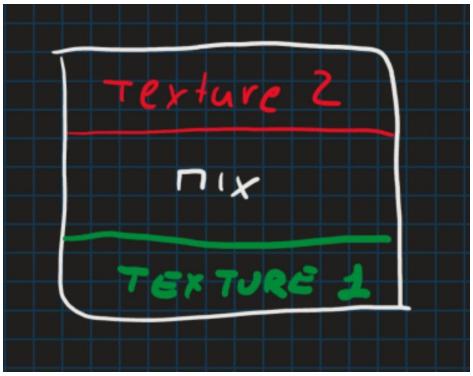
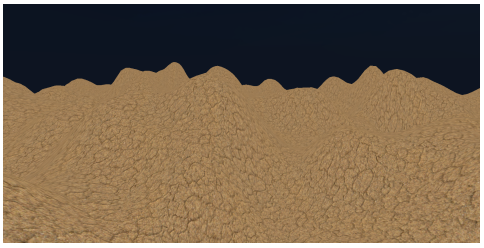
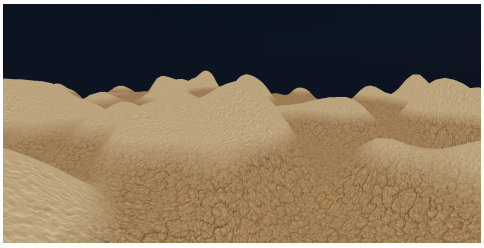


Figure 14: Multi-texturing schema.



(a) Terrain one texture.



(b) Terrain two textures.

Figure 15

2.8 References

If you want to find more about the things mentioned above, you can check **terrainManagement.h**, **terrainVert.vert**, **terrainFrag.frag** files.

3 Water

The water is basically a squared plane (built in the same way of the flat terrain). Each cell has a water plane that will be visible only if some vertex of the cell has an height greater than the terrain. After creating the plane, the first thing I implemented was the reflaction and the refraction on the plane.

(a) Plane. (b) Vertices, indices and faces.

Figure 16
3

3.1 Frame buffer objects

The frame buffer is used to render our scene with all the information from the different buffers, such as color and depth. You can create your own frame buffer object to render the scene and save it in a texture. In particular, every time we render the scene in our frame buffer, we will update our 2D texture. This is exactly what I need because we have to create a texture to project inside the plane. I'll quickly explain how I created the class for the frame buffer object (Full code in **waterFrameBuffer.h**).

First I defined a method to create the frame buffer using some OpenGL function.

```
GLuint createFrameBuffer() {
    GLuint frameBuffer;
    glGenFramebuffers(1, &frameBuffer);
    //generate name for frame buffer
    glBindFramebuffer(GL_FRAMEBUFFER, frameBuffer);
    //create the framebuffer
    glDrawBuffer(GL_COLOR_ATTACHMENT0);
    //indicate that we will always render to color attachment 0
    return frameBuffer;
}
```

Figure 17: Method for the creation of the frame buffer.

So I defined the two methods of creating texture attachment, one for colour and one for depth attachment. They are just a simple generation of a texture with a new "glFramebufferTexture" statement. (I only insert the screenshot of the attached colour since they are pretty much the same).

```
GLuint createTextureAttachment(int width, int height) {
    GLuint texture;
    glGenTextures(1, &texture);
    glBindTexture(GL_TEXTURE_2D, texture);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE); //?
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE); //?
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texture, 0);

    GLenum Status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
    if (Status != GL_FRAMEBUFFER_COMPLETE) {
        printf("FB tex error, status: 0x%x\n", Status);
        return false;
    }

    return texture;
}
```

Figure 18: Method for the creation of the texture.

Last I defined a method to create the depth buffer.

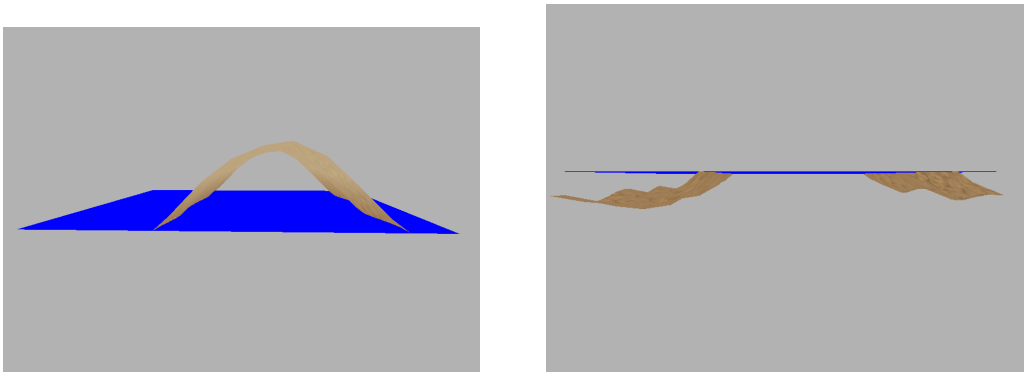
```
GLuint createDepthBufferAttachment(int width, int height) {
    GLuint depthBuffer;
    glGenRenderbuffers(1, &depthBuffer);
    glBindRenderbuffer(GL_RENDERBUFFER, depthBuffer);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, width, height);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthBuffer);
    return depthBuffer;
}
```

Figure 19: Method for the creation of depth buffer.

Finally I created two methods, one to bind the frame buffer and one to unbind it so that I can switch from the normal frame buffer to the one created by me. I stored two frame buffers in this class, one for reflection and one for refraction. Basically I'll render the scene and create my own texture that will be used by the main render to apply it to the water.

3.2 Clipping plane

For both refraction and reflection textures, you don't need to render the whole scene. Actually I need the one above the surface for the reflection and the one under the water for refraction. So I defined the clipping plane which allowed me to render only a part of the scene and obviously it is positioned at the same level of the water.
After enabling a clipping plane (**glEnable(GL_CLIP _DISTANCE0)**) in the main rendering, I use the **gl_ClipDistance[0]** variable in the shaders to define which vertex should be clipped. If this distance is positive, I should render this vertex, if negative no. To calculate it, I simply use the dot product of the world position (model matrix * position) and the horizontal plane, whose equation is **vec4**(0, -1/+1, waterHeight). The Y component will be +1 if we want to clip the lower part (reflection) or -1 if we want to clip the upper part (refraction)



(a) Clipping the below part.

(b) Clipping the above part.

Figure 20

3.3 Projective texture mapping

To apply the texture to the plane, I used **projective texture mapping** that is a texture mapping method that allows you to project a textured image onto a scene. I needed to find the correct UV coordinate and for that I just found the screen space coordinate on the water coordinates and I can use those exactly to sample the texture. Basically I need to calculate the normalized device coordinates and I can do it using the perspective division. I just need to divide the clip space component (projection matrix * view matrix * model matrix * position) into its **W** component and then divide by 2 and add 0.5 to find the correct coordinate system .

$$newCoord = vec2(clipspace.xy/clipspace.w)2.0 + 0.5) \tag{2}$$

Now I can sample the refraction and the reflection by using the X and Y components as texture coordinates. I also need to mention that the rendering for the reflection texture is done with the camera reversed and shifted slightly down to give the reflection effect. On the other hand, no changes are applied to refraction rendering.

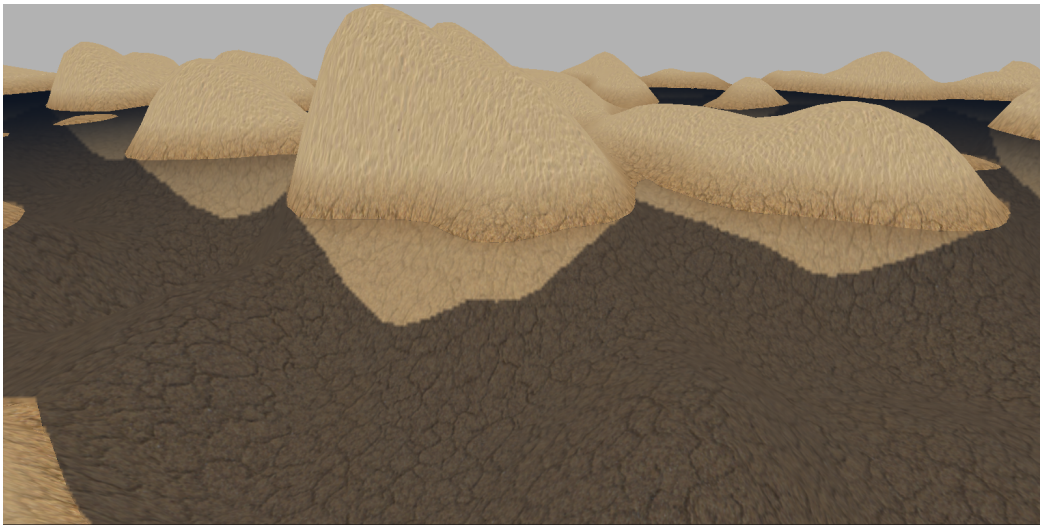


Figure 21: Reflection and Refraction texture mixed.

3.4 DuDv Maps

To create a more realistic water, I used a particular texture to distort the water.

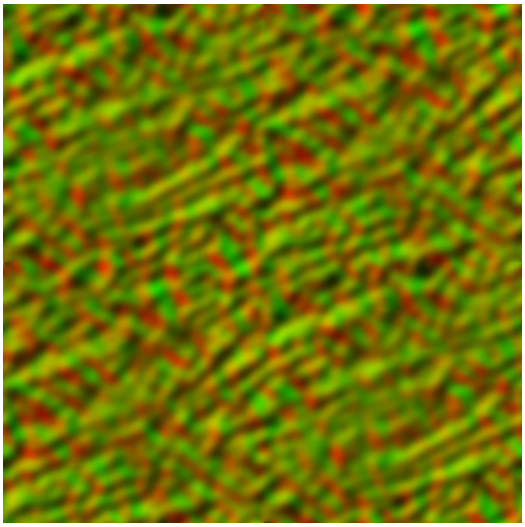


Figure 22: DuDv texture.

Basically, I try this new texture and used these coordinates to change the value of the previously calculated texture coordinates, in the x component and in the y component. To change the coordinates over time, I create a uniform variable called waterMovement to change the position of the coordinates at each frame. Since the value of our dudv texture is always positive, I applied a conversion to create value from -1 and 1, multiplied by 2 and subtracted 1. After adding the distortion, I clamped the result to have a value between 0 and 1.

```
vec2 distortedTexCoords = texture(waterDuDvTexture, vec2(outTexture.x + waterMovement, outTexture.y)).rg*0.1;
distortedTexCoords = outTexture + vec2(distortedTexCoords.x, distortedTexCoords.y+waterMovement);
vec2 totalDistortion = (texture(waterDuDvTexture, distortedTexCoords).rg * 2.0 - 1.0) * waveStrength;

refractionTextureCoor += totalDistortion;
refractionTextureCoor = clamp(refractionTextureCoor,0.001,0.999);

reflectionTextureCoor += totalDistortion;
reflectionTextureCoor = clamp(reflectionTextureCoor,0.001,0.999);
```

Figure 23: **waterShader.frag** distortion part.

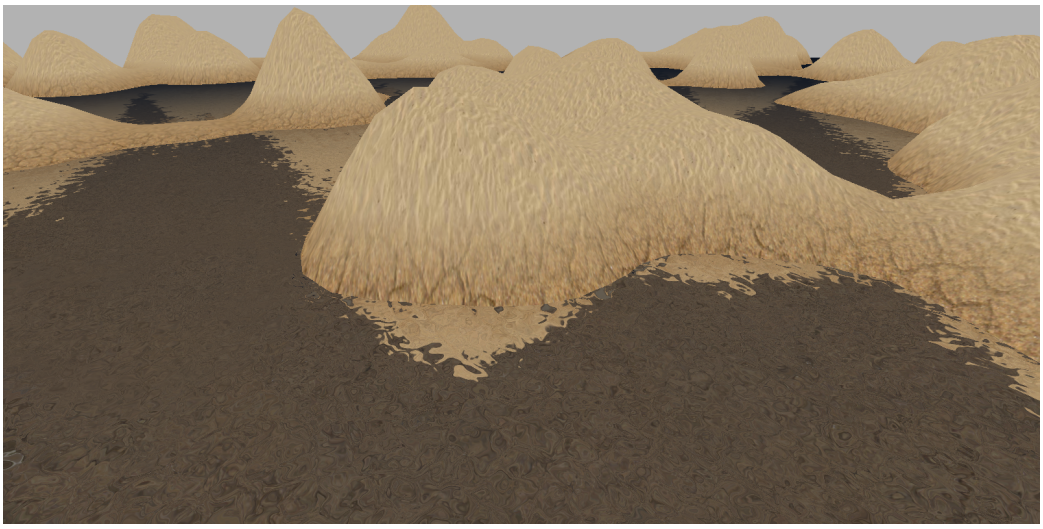


Figure 24: Water distortion.

3.5 Normal map

I want to simulate a light effect on the surface of the water. To do this I used a normal map which is similar to the dudv map above.

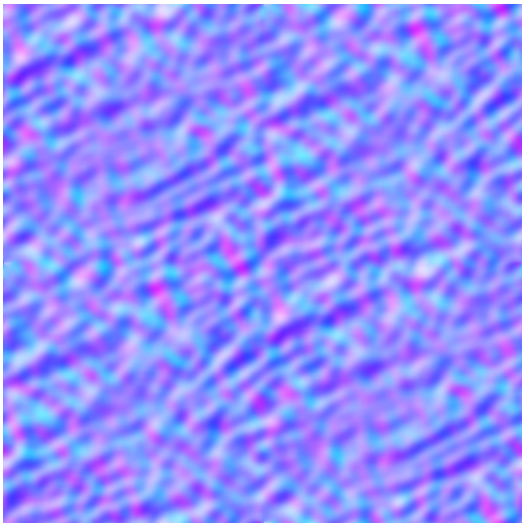


Figure 25: Normal texture.

In the shader, after sampling the texture, I calculated the normal vector using the red component for the X, the blue component for the Y and the green component for the Z. We want the Y component to always be positive, while the X and Z they can also be negative. So we multiplied the two coordinates by 2 and subtracted by 1 as we did before. After that I normalized the vectors.

$$\text{vec3 normal} = vec3(normalMap.r*2-1, normalMap.b, normalMap.g*2-1) \tag{3}$$

Next I passed as uniform the position of the camera, the position of the light and I added a few more parameters to create the specular lighting on the water like **shineDamper** to determine the distance from where the camera will see any difference in the brightness of the surface and **reflectivity** which determined the amount of reflected light. Then I calculated the vector from the light to the vertex using the world position (model * position) and the vector from the world position to the camera.

$$\text{vec3 fromLightVector} = normalize(WorldPosition - lightPos); \tag{4}$$

$$\text{vec3 viewVector} = normalize(cameraPos - WorldPosition); \tag{5}$$

Now I can first calculate the reflection of **fromLightVector** with the normal, then calculate the dot product of **viewVector** and the reflected light to understand how bright the surface will be at that point. After that we had to use the pow function between the point product calculated earlier and the **shineDamper**. Finally, we can multiply the light colour (passed as uniform), the calculated specular illumination and the reflectivity.

```
vec3 reflectedLight = reflect(normalize(fromLightVector), normal);
float specular = max(dot(reflectedLight, viewVector), 0.0);
specular = pow(specular, shineDamper);
vec3 specularHighlights = lightColour * specular * reflectivity;
```

Figure 26: Specular lighting code.

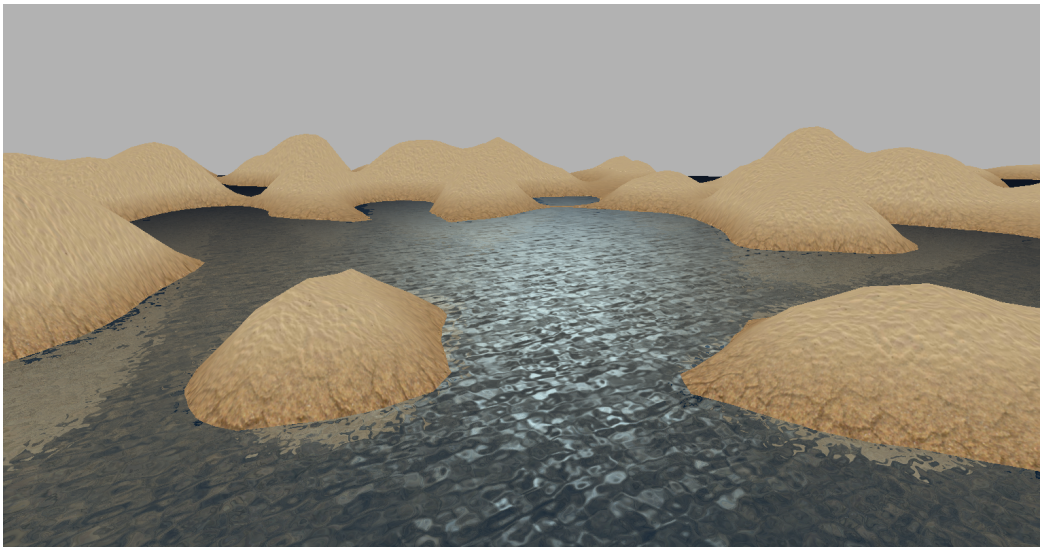


Figure 27: Specular lighting

3.6 Soft edges

There is one last problem to be solved and it is caused by the distortion, in fact, there are rendered parts that should not be in the water.

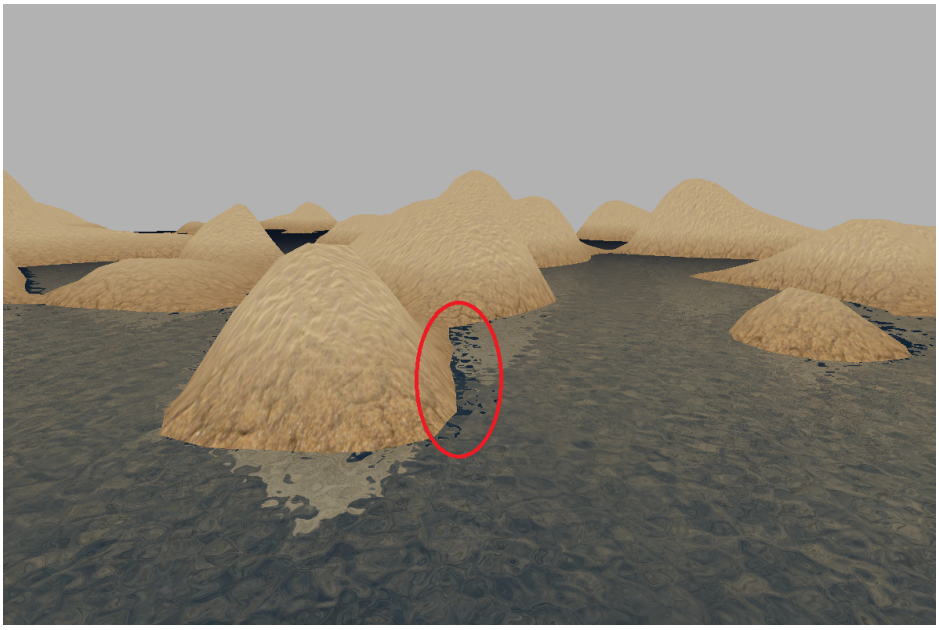


Figure 28: Picture that shows the problem

I want to calculate the distance between the bottom of the water and the water plane to the camera in the shader.

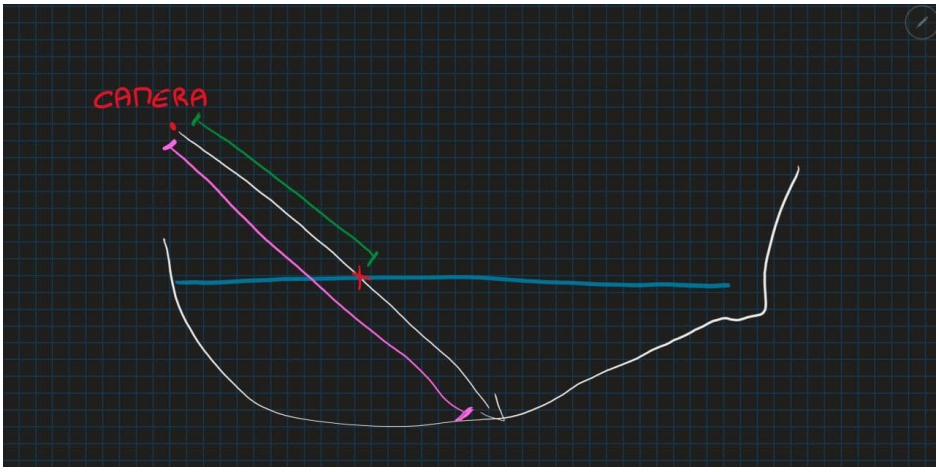


Figure 29: We want the subtraction between pink line and green line

To calculate the distance from the bottom of the water I will use the texture of the refractive depth calculated in the FBO. However, the **r** component

of the depth texture gives us a value between 0 and 1. To convert it to a distance I used a formula that uses the far and near plane. To calculate the distance from the plane I can use an OpenGl variable called **gl_FragCoord**, then take the Z component that gives us the depth of this fragment and then apply the same formula as above. Finally I subtract the two values.

```
float depth = texture(depthTexture,refractionTextureCoor).r;
float floorDistance = 2.0 * near * far / (far + near - (2.0 * depth - 1.0) * (far - near));
depth = gl_FragCoord.z;
float waterDistance = 2.0 * near * far / (far + near - (2.0 * depth - 1.0) * (far - near));
float waterDepth = floorDistance - waterDistance;
```

Figure 30: Code of the above passage.

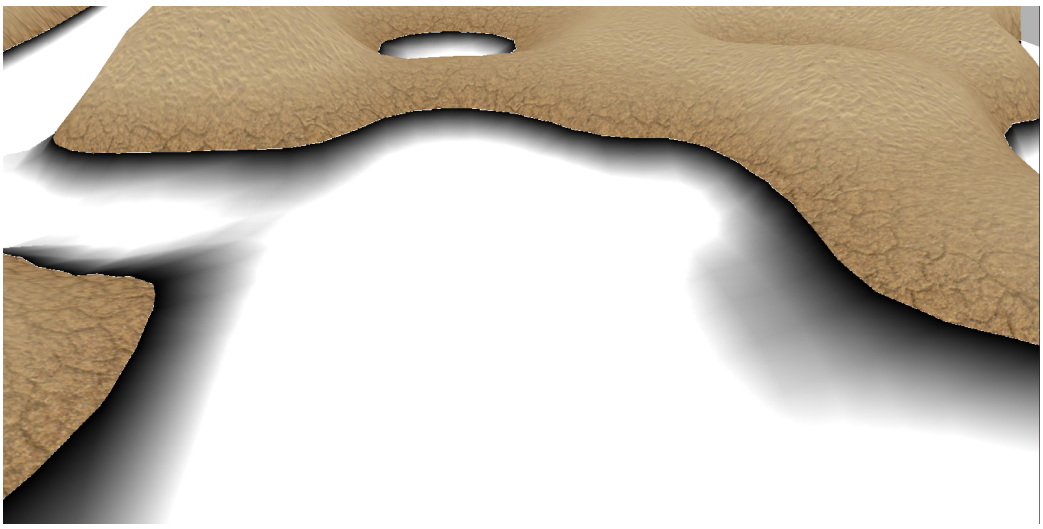


Figure 31: Water depth values applied in the water.

Now we want to use this information to change the water near the edges. When the depth value is 0, the water is transparent. So the closer you get to the value 1, the less transparent the water is. After the value 1 there is no transparency. Basically I just need to clamp my water depth (divided by a factor that determines how long the transition from 0 to 1 takes) and multiply it by the **totalDistortion** value in the figure **??** and the value **specularHighlights** in figure **??**.

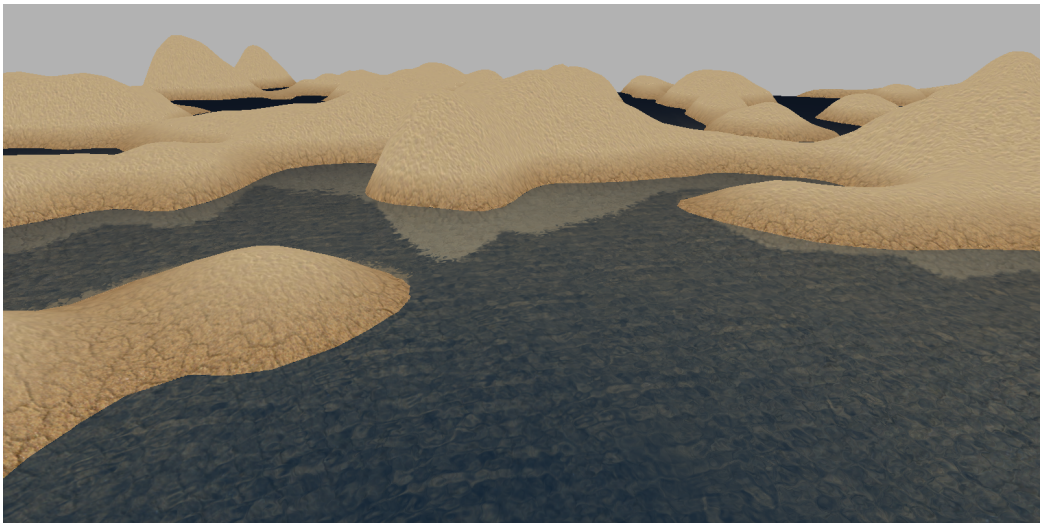


Figure 32: Water result after all stages.

4 Cube Map

I used a cube map to create the sky. One of the images I choose, contains moon to give the illusion that the light comes from there (the light source is indeed in the same direction).

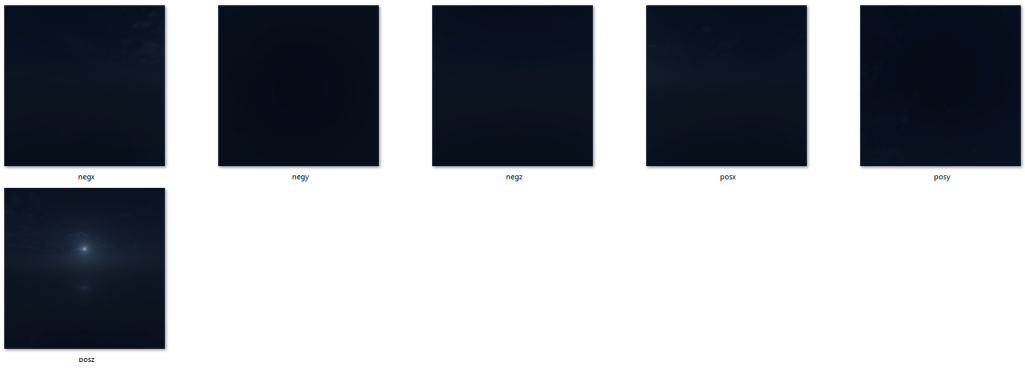


Figure 33: Images used for the cube map.

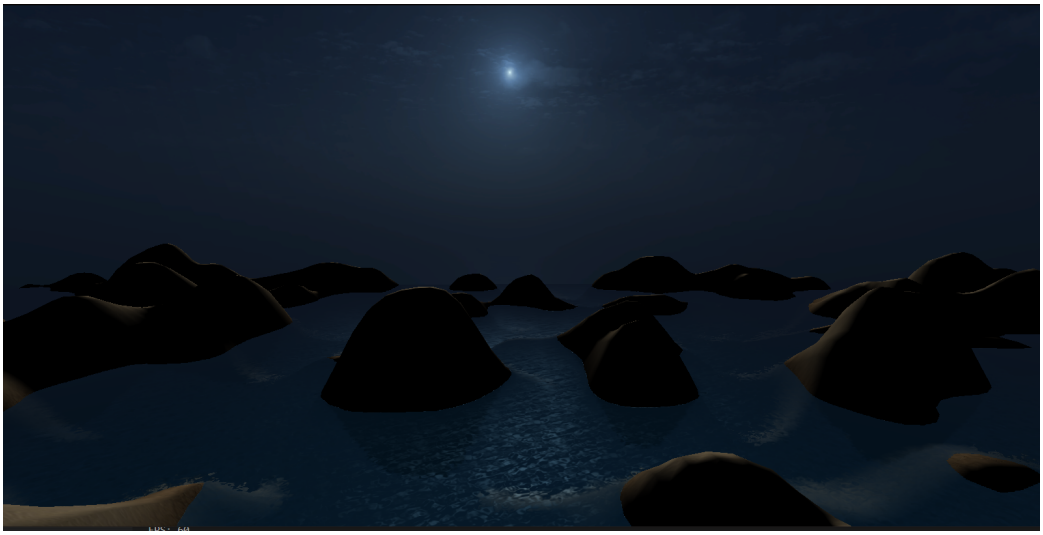


Figure 34: Cube map applied.

4.1 Fog

Right now I can see the movement of the cells. To avoid this I need to create a fog. The general idea is to blend objects with the colour of the sky, depending on how far they are from the camera. Near objects will have their normal colour, distant ones will have the same colour as the sky, and the objects in the middle will have a mixed colour. To do this I introduced a variable called (visibility) used in each shader, whose value goes from 0 (object in the fog) to 1 (normal colour). I used an exponential formula to calculate it which has two parameters: **density** and **gradient**. The first represents the thickness of the fog and increasing this value will decrease the overall visibility of the scene. The second determined how quickly visibility decreases with distance, and increasing this value makes the transition from 0 visibility to full visibility much smaller. To get the distance from the camera I used the built-in GLSL length function on the (positionRelativeToCam) variable (view matrix * model matrix * position).

After calculating the visibility I mixed the color value with the fog color (usually the sky color) using the visibility factor in the fragments shader.

$$\text{FragColour} = mix(vec4(fogColour, 1), FragColour, visibility); \quad (6)$$

```
float distance = length(positionRelativeToCam.xyz);
visibility = exp(-pow((distance*density),gradient));
visibility = clamp(visibility,0.0f,1.0f);
```

Figure 35: Code for the fog visibility calculation.

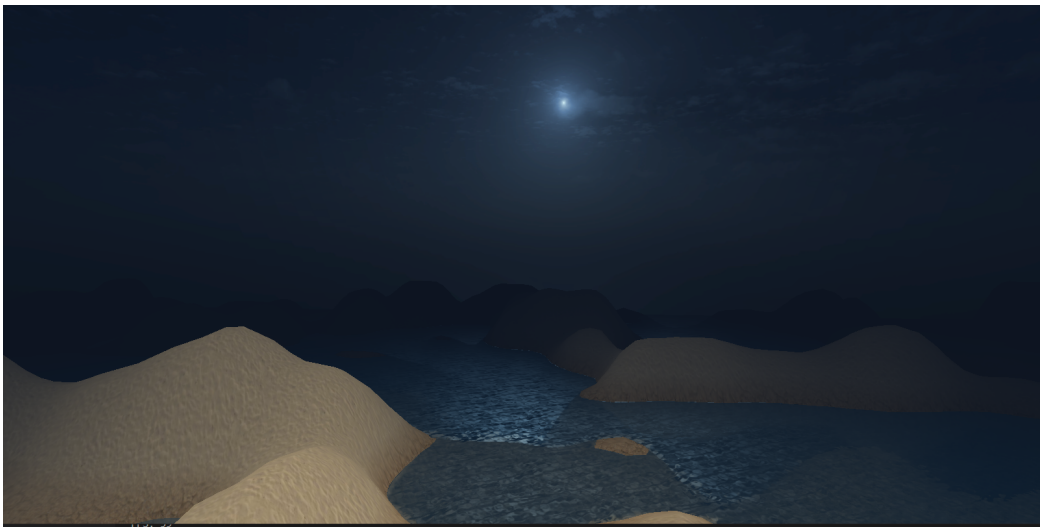


Figure 36: Fog with fixed colour.

Perhaps in the image it is not very visible but there is still a problem. Fog is a fixed colour, but the sky has several. In fact, I can still see the movement of the terrain. The solution is to blend the horizon of the cube map with the colour of the fog. I have defined a lower and an upper limit. As before, below the lower limit, the cube map will have the colour of fog, above the upper limit it will be normal, and in the middle it will be mixed.

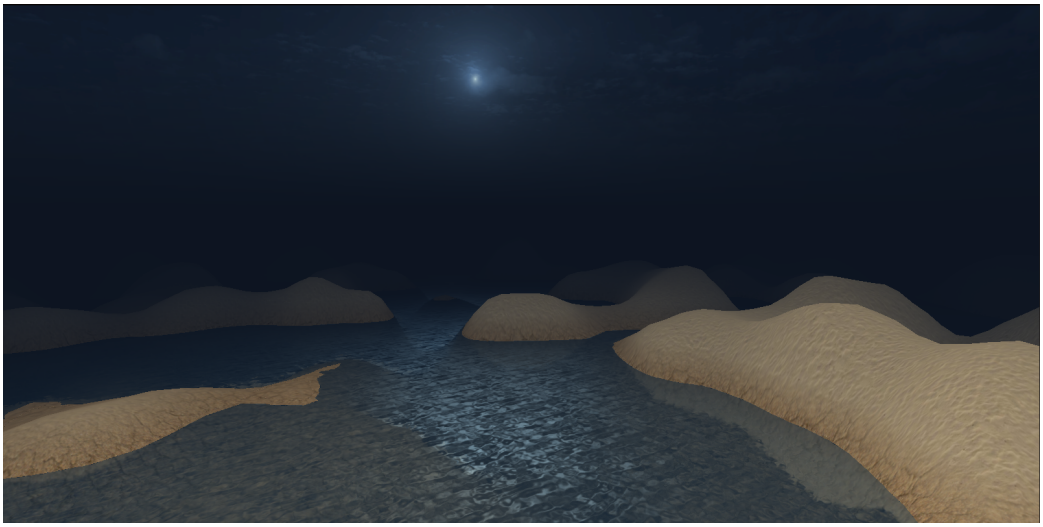


Figure 37: Final fog result.

5 Other elements

I decided to decorate the scene with other elements, in particular with a tree and a cactus. To keep things simple, I randomly assign one of the two patterns to each cell above the water. The only difficult thing is to calculate the height of the ground to position the object correctly.

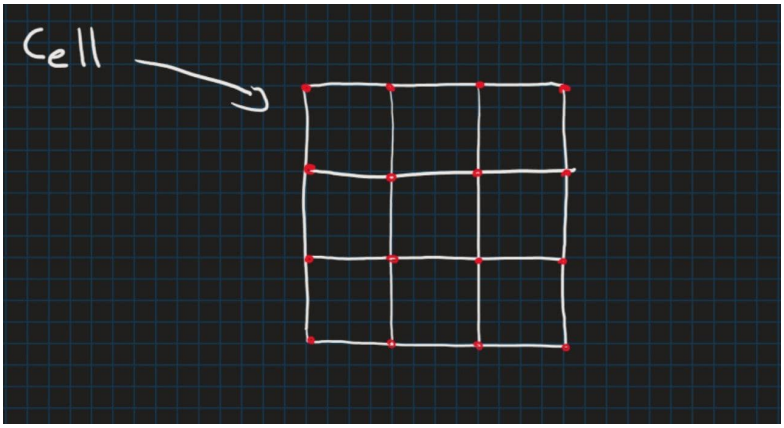


Figure 38: Grid square of a cell.

Given a position within a cell, I need to calculate the height of that point. First I need to find the size of a grid square. I divided the size of a cell by the number of vertices subtracted by one. For example in the image above there are four vertices and with a size of 500, the size of the grid square is 500/3. Now I can calculate which square of the grid is the object in by dividing the X and Y coordinates of the object's position with the grid size of the grid an floor the result. Now I want to calculate what the coordinates of the object are in that grid square. To do this I apply the mod operator between the position and size of the square and then divide it by the size of the square. Basically now I have the coordinate inside a square from 0 to 1. To understand what the object of the triangles is, I checked the X coordinate and I saw if it is less than or greater than 1 - Z. Finally I have my triangle and I have the height of the vertices and I can calculate the height of my points in several possible ways, for example using the barycentric interpolation. After doing this, we can easily place the different objects at the right height.

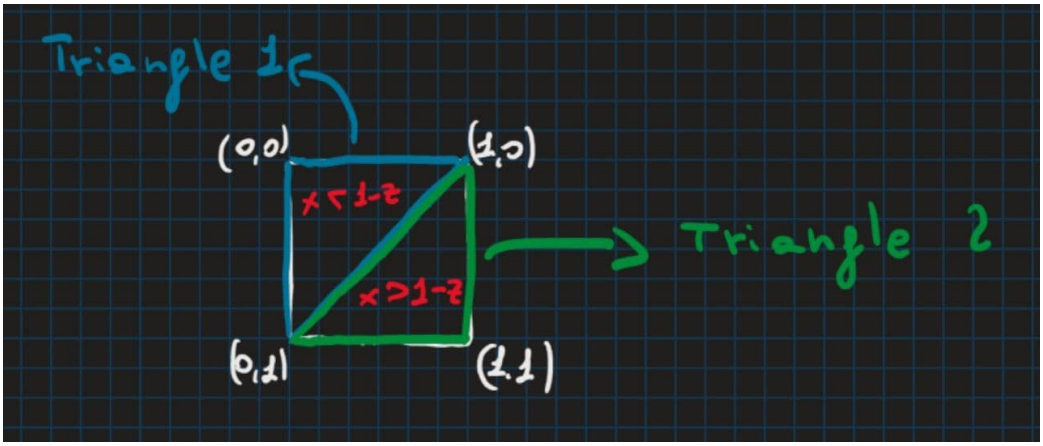


Figure 39: Grid square triangles division.

Figure 40: Trees and cactus positioned.

6 Performance

Last thing, I want to mentioned the performance of the terrain. With a size cell of 500 with 8 vertices and 21 cells for row (441 cells in total), the scene is rendered with a range beetween 58 and 60 FPS.

```
void calculateFPS(){
        frameCount++;
        finalTime = time(NULL);
        if((finalTime - initialTime) > 0){
            std::cout << "FPS: " << frameCount / (finalTime - initialTime) << std::endl;
            frameCount = 0;
            initialTime = finalTime;
        }
}
```

Figure 41: FPS calculation.

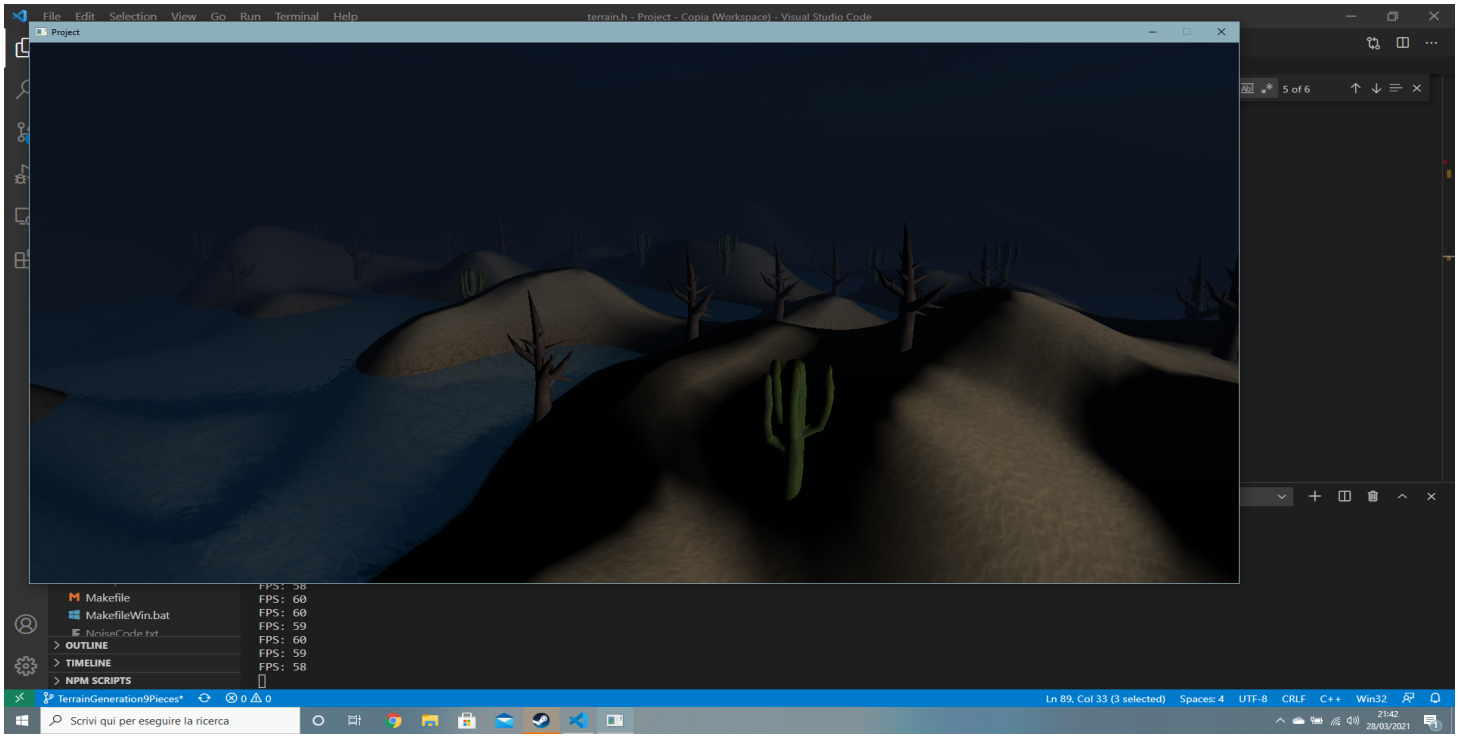


Figure 42: Scene with FPS below.

7 Conclusion

This is how I implemented everything. If you have any more questions I can answer them during the oral interview.

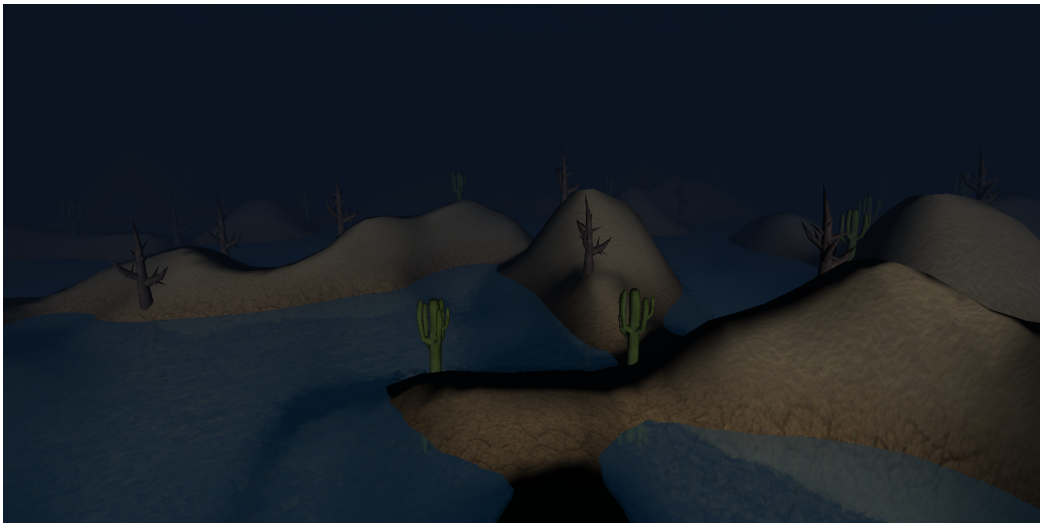


Figure 43: Final image3 (front view).

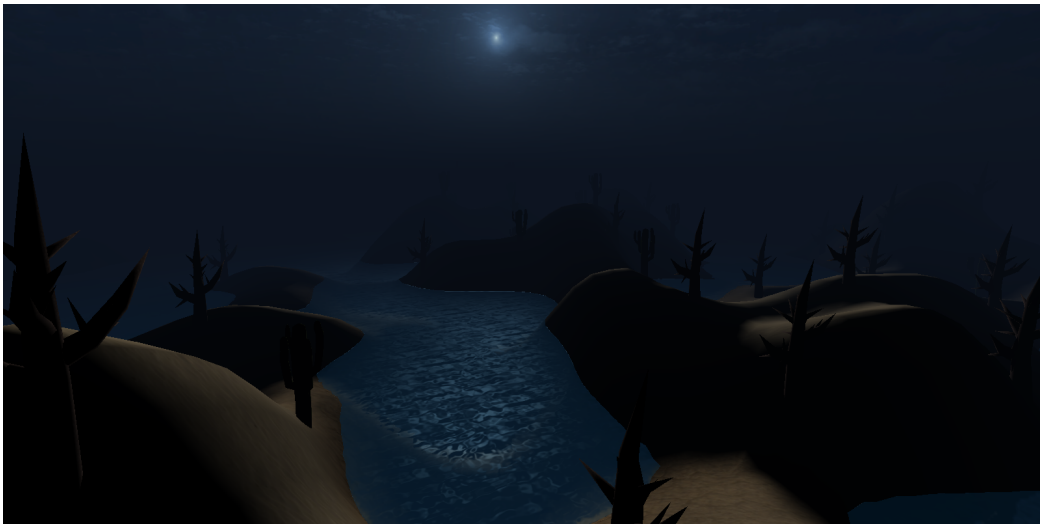


Figure 44: Final image (back view).