Terrain Engine

Minjing Yu

Contents

1 Required Tasks

The goal of this project is to construct a 3D scene and wander in it. The effect will look like what is shown in Figure 1.

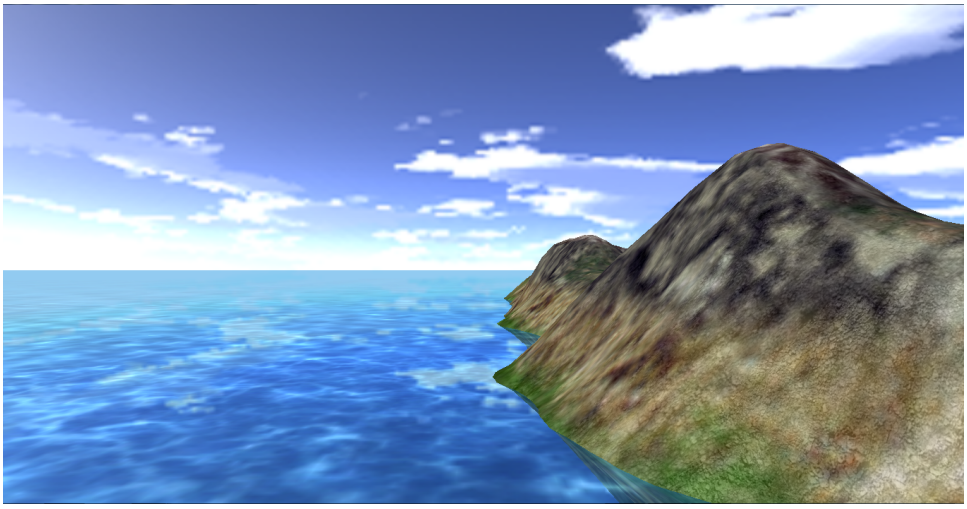

Figure 1: The whole effect of this project

To finish this project, the following techniques are necessary:

- Texture (for constructing skybox, water wave and the terrain).
- Model-View transformation (for control movement in the scene).
- Load terrain data from an image and render it.

2 Detail Description

This part will talk about the implementation in detail.

2.1 Skybox

Let's build up the scene by starting with the skybox, which includes blue sky with clouds and the sun. Besides, we will also create the flowing sea and the inverted reflection of the sky. It will look like what Figure 2 shows.

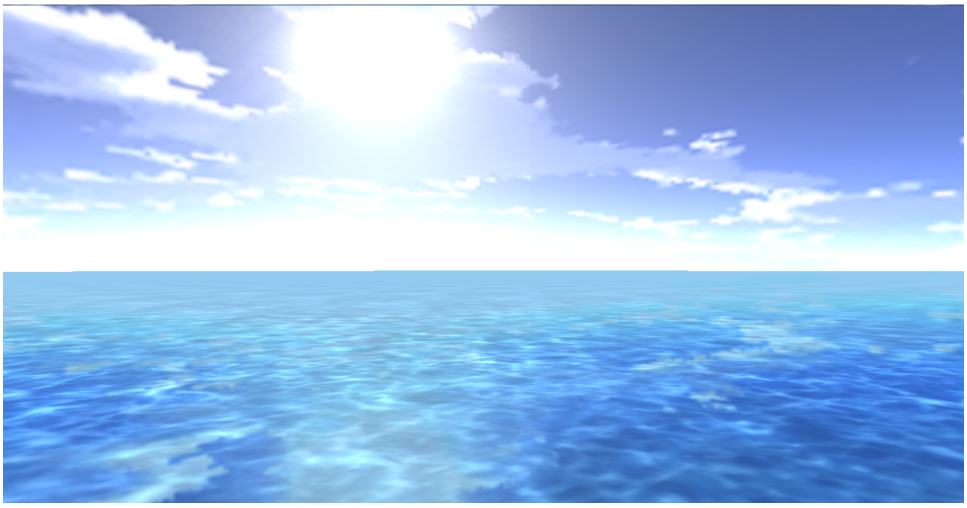

Figure 2: Skybox built using the images

2.1.1 Images used for texturing

Generally, to build up the sky, we basically just use images taken from the same position to different directions (front, back, left, right, top and bottom) and map them to the inside of a box, which we denote as *skybox*. And if our eyes are in the box, it will look like a panoramic view. The images we use in this project are shown in Figure 3.

The Cubemap, as a texture generation method that OpenGL provides, makes building up a skybox easier. But in this project, it may be wiser to avoid using it for the following reasons: First, 6 images are needed to build up a cubemap, and in our case we only have 5 because we'll leave the bottom one for the sea surface. You may think may be we can use the water wave as the bottom texture, however, since the resolution of it is different from the others, you will see some wrong effect if it is applied directly as part of cubemap (readjust the resolution may also not be an option considering we need to create the flowing effect of waves). Second, the cubemap in OpenGL is mainly used to create a reflection mirrored effect

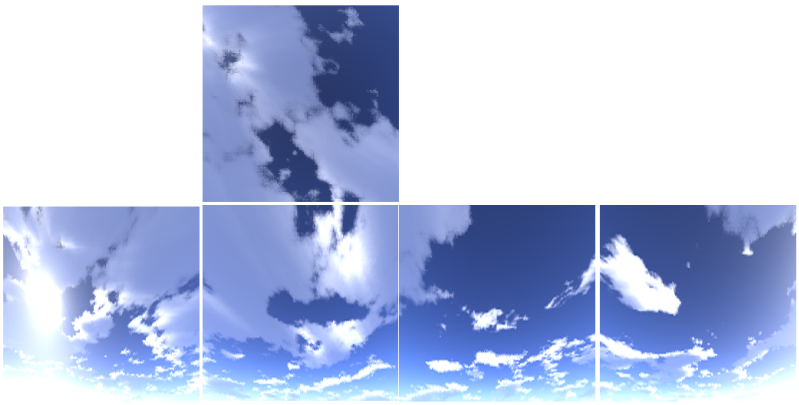

Figure 3: Images used for skybox

on the surface of an object, which in our project is not the purpose. So we can just manually use 5 2D textures and map them onto the inner-faces of a cube.

The images provided can be easily imagined how to map them onto the surface of a box. It should be pretty simple, but there still something else you should notice.

2.1.2 Parameters of the box

As a skybox for a large scene (in our case for example), the skybox may be quite large, which you need to adjust it yourself to get the most comfortable parameter. Another thing to notice is the ratio among the length, width and height of the skybox. Figure 4 is a screenshot for a $1 \times 1 \times 1$ textured skybox. You can see the sun looks very "thin" because the original image looks just like that (probably for the reason that the camera taking the pictures has a large fov). To make it better, we need to stretch the skybox a little. There is no standard for this, and it's your call to adjust the ratio to make the scene look normal.

2.1.3 Erasing edges

Another problem may occur is like what Figure 5 shows, i.e. the edges of the skybox are shown explicitly and reveal that the scene is actually just a box which is obviously not what we want. This problem is actually related to the wrapping mode of texture you set. Recall we've discussed it during the class:

Figure 4: Skybox using a standard cube

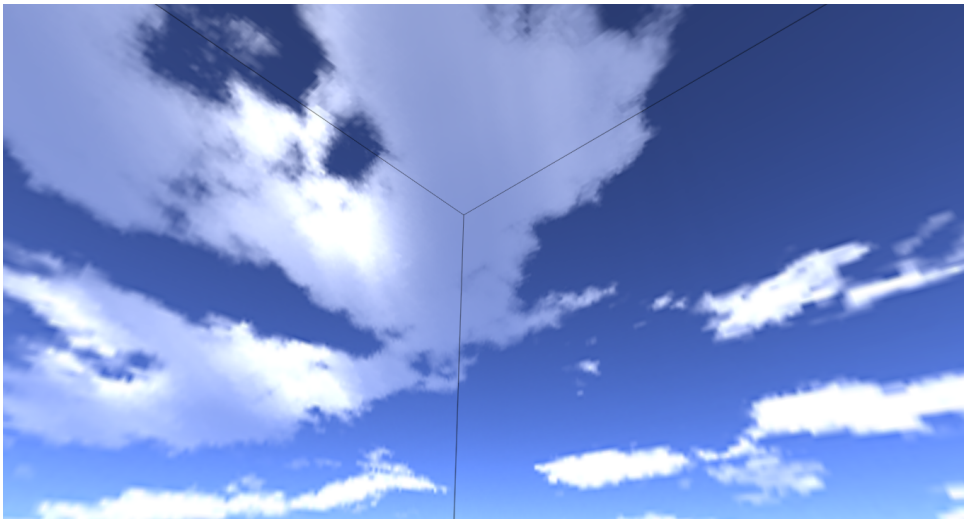

Figure 5: Edges of skybox give it away

```
void glTexParameterf(GLenum target, GLenum pname, GLfloat param)
```

With the texture target (say GL_TEXTURE_2D) and the texture coordinate applied wrapping (say GL_TEXTURE_WRAP_S and GL_TEXTURE_WRAP_T), we have four parameter:

```
GL_REPEAT GL_CLAMP GL_CLAMP_TO_EDGE GL_CLAMP_TO_BORDER
```

GL_CLAMP, which leads to the bad effect above, is the default value for most cases. It just uses the border color to set the texture on the border of the square. GL_CLAMP_TO_EDGE will be a good choice to ignore the texture on the border and erase the black border of the box.

Until now you should set up the skybox using the 5 sky images, probably like Figure 2 shows except for the sea.

2.2 Water wave

Though we build up the sky, the ground still be the pure color you choose. Here is another image shown in Figure 6 to create a water wave effect. This

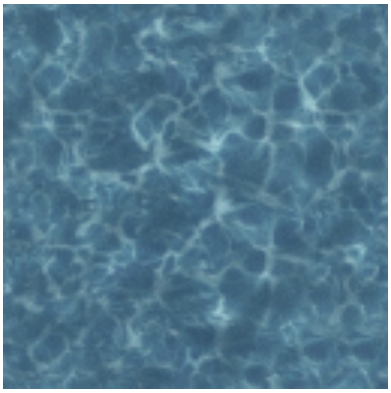

Figure 6: Water wave image

image's resolution (128×128) is smaller than the five above (256×256). Next, let's create the water wave step by step.

First, let's create a static wave. If you try to map the image to the bottom of the box with the texture coordinates ranging from 0.0 to 1.0, then the wave will look very big and not clear since the resolution is low. One way to avoid this is to map the image to a **part** of the bottom face and repeat it until it covers the whole bottom. There are many ways to do that, and a suggested one is to make the wrapping mode of this texture be GL_REPEAT and give a coordinate larger than 1.0 where you will assign 1.0 originally. That way the texture will repeat itself, and the times of repeating are decided by the maximal value you set for texture.

Since we have a water wave, tt's pretty easy to make it move. Instead of giving a fixed coordinate to the vertex of the bottom face, you should make it changeable. For example, if you want to move the wave in the x-direction, the x texture coordinate should be added to a value (say $waveshift$) every time the frame is refreshed.

Maybe you will ask: how to control the speed of the water flowing? Well, under the method of our implementation, the flowing speed is related to the frame refreshing speed. For example, if you're using Windows, a GetTickCount() function will help you to know how much time elapsed between the adjacent two frames, and you can use this value to decide how fast the water flowing. There are similar system calls for other platforms to make your water wave move the same speed even under different machines.

This part's job has not been done yet. We need to make the reflection of the sky, This will make the sea much more real. What we actually do is just reversing the skybox and redrawing it. But from the class we know that the reflected image is below the water wave and it'll not be seen if we enable depth test. So we need to mask the depth buffer before we render the reflection(or in our case, disable depth test is fine too). And blending is another technique we need so that the reflection and the water will be mixed. Note that the color of the sea is determined by the blending factors you choose. Try to use different parameters to achieve the most real effect you think.

Until now, we have a beautiful scene with blue sky, white cloud, the sun, waving sea and the reflection of the sky.

2.3 Terrain Model

Now let's see another significant part in this project — load the terrain model. We have three images to construct the terrain:

- **heightmap.bmp** stores the height information of the terrain.
- **terrain-texture3.bmp** is the color texture image. It gives the terrain the mountain, grass etc.
- **detail.bmp** gives the terrain texture more detail.

2.3.1 Loading height information

The heightmap.bmp is a gray-scale image storing the height information. What you need to do with it is different from the normal texture images: The value stored in every pixel represents the height of the terrain. You can load it as an array or a matrix. The SOIL library provides a function to load an image as an array, or you can load the .bmp file yourself if you are clear about its structure.

Having the height information at hand, we can get a point (x, y, z) in 3D coordinate system by its position (i, j) in the image and its height value h. A direct relation will be $(x, y, z) = (i, j, h)$, but some scaling will probably be necessary. With these points, we can render them either in triangles or quads by connecting the adjacent points.

Note that we don't actually need to form an explicit terrain model storage, since the height information is organized in a row (or column) major way as an array and we can get the point efficiently when we need it. Or you may want to use a specific structure to store the terrain. You can either design it yourself or use an open-source library[1] to help you concentrate on the rest part of this project.

[1] the Trimesh2 library is good:http://gfx.cs.princeton.edu/proj/trimesh2/, but you still need to read the .bmp file yourself

2.3.2 Texture mapping

Since we render the terrain the height information in a regular form, texture the color image should be easy. It's almost the same with the texture mapping we've practiced during classes. The only thing you may wonder is how to decide the texture coordinate to each vertex of the terrain model. That's not so difficult: texture coordinates in adjacent pixel has a constant increment (or decrement) of 1.0 divided by the width (or height) resolution of the gray-scale image.

Next thing is the detail information we need to add. Here comes the multi-texturing technique. We can use the color texture as the 1st texture (GL_TEXTURE0) and use the detail texture as the 2nd one (GL_TEXTURE1). We've talked about multi-texturing during the class. You may want to know how to choose the texture environment to combine the two textures. GL_DECAL and GL_REPLACE is clearly not what we need here. The left ones will be GL_ADD, GL_ADD_SIGNED, GL_MODULATE (default value) etc. Considering the clamping to the color value, modulating and simple adding will make the color weird, and the GL_ADD_SIGNED may be the best choice.

2.3.3 Put it in the sea

The textured terrain you load should look like the one shown in Figure 7. You can see that there is a circle of "water" around the terrain. You may want to cut this part off while putting the terrain in the water. There are also many ways to do that. One way is to use clip plane the cut it out. the other is a little more complicated: you can set the blending function carefully when you render the reflection of the sky and the water wave so that the color of the part of the terrain which is under the water will not be used in blending at all. Note that we also need a reflection of the terrain, and this procedure is similar to the rendering reflection of the skybox.

2.4 Interaction

A good wandering demo should have a fine interaction. One thing to note is that all the model view matrix operations in OpenGL is multiply a single matrix to the right of the current matrix, which means that the objects to be rendered is applied these operation in a reverse order. To make the successive operations applied later on, you need to maintain the model view matrix carefully.

Moreover, you may also want to make the interaction smooth, that is, when you forward in the scene, you will speed up slowly (and may be stable at certain speed) but not move to the next position suddenly; when you stop, you will slow down other than stop immediately. This requires you control the speed of the movement carefully too.

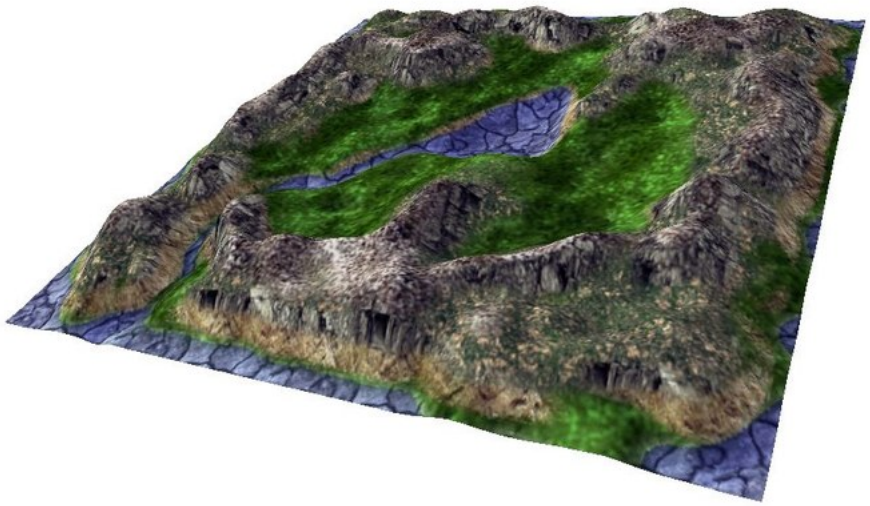

Figure 7: Textured terrain

3 Bonus

Any extra features can be considered for an extra credits. The difficulty and effect will determine how much extra you'll get. Please describe the features you implement in the final report in detail.