COSC3000 Visualization, Computer Graphics & Data Analysis

A 3D Scenery

Student Number: 44575986

Student Name: Yuzhe Jie

Introduction

Background

"Nobody could have predicted that in just 20 years, we would be immersing ourselves in realistic living cities, flying over gorgeous tropical islands and going head-to-head with astoundingly rendered characters – and not even being particularly impressed" by PC Pauls, 2010.

As stated in the comments, 3D object rendering is changing human being's lives as the advancement of computer graphics. Games and films usually contain lots of astonishing 3D background images which could be one reason of their popularity. With modern computers, 3D scenery which not only mimic but also goes nicer than reality can be created, and with nowadays' graphics rendering tools, each individual can access and get hands on these technologies to render their own 3D background images, as a starting point to learn computer graphics.

Aim

This project aims to render a 3D background scenery which contains hilly terrains, sky and a pond of reflective and refractive flowing water. This scenery can be see very often in nature and could possibly bring some relaxed and comfortable feeling for people. The finished scene can have versatile use. It can be used just as a simple background image in an online meeting, or it can even be used as a part of the background in a 3D game.

Methods

Software

OpenGL API was mainly used for rendering 2D and 3D vector graphics in this project. To design this project as a software program, java programming language plus its own game library called LWJGL which enables cross-platform access to OpenGL API (Lightweight Java Game Library) were used. Besides, Slick-Util library and PNGDecoder were adopted to load texture images for LWJGL.

Implementation

1.Some preparation

There are some important concepts and classes to be specified or created before rendering target object.

1.1 VBO, VAO and Raw model

VAO (Vertex Array Object) is an attribute list storing all the data of a model, such as vertex positions, vertex colours, normal vectors, texture coordinates etc. These data are first stored in a VBO (Vertex Buffer Object) by glGenBuffers(), glBufferData() and glBindBuffer() methods. The VBOs are simply an array buffer of data which may be type of float or integer, and will be put into a VAO once bounded by OpenGL. Each VAO is created by glGenVertexArrays() and glBindArrays() methods. It was stored in memory and can be called anytime by its ID. A raw model of an object contains its VAO id, and number of vertices.

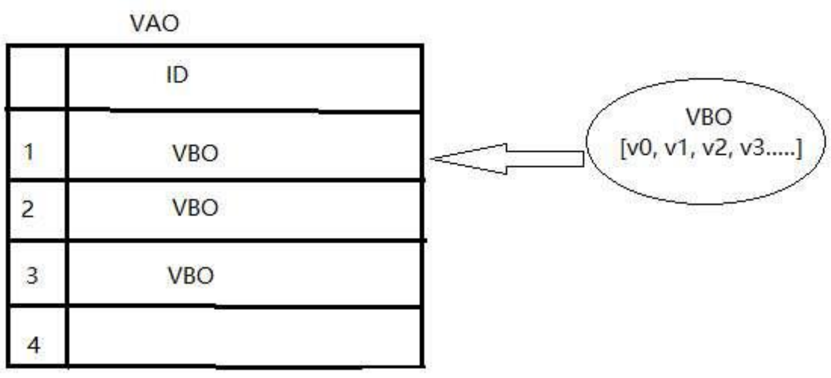


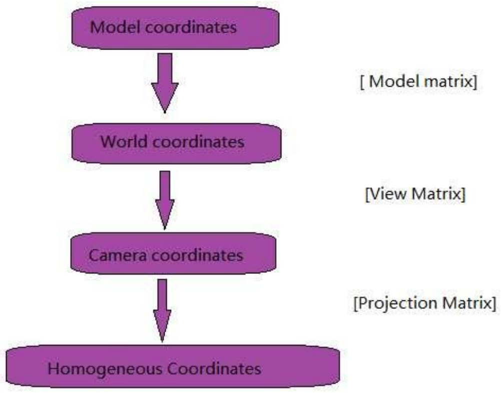
Figure 1 VAO & VBO

1.2 Camera, Light, Projection Matrix and View Matrix

A camera class represents the virtual camera in the scene world. It has 3 attributes namely position, pitch value (rotation around x axis), yaw value (rotation around y axis) and roll value (rotation around z axis). 10 keyboard keys are used to "move" the camera. When the camera is moving around, the scene is moving the opposite direction, this is achieved by the View Matrix. The View Matrix represents the transformation that is exact opposite to the camera's transformation. Another important matrix relating to camera is the Projection Matrix. It will give a wider view of objects in distance and also make them appear smaller when they are further away. Without a projection matrix, objects at similar x y coordinates but different z positions cannot be differentiated. Both View Matrix and Projection Matrix are created in the Renderer class (see below) and done matrix multiplication with object's position to give a suitable view in the shader file. However, projection matrix is only created once, while View Matrix updates every time the camera moves. Finally, Light class represents the light in the scene world, its position is set to (20000, 20000, 2000) and its colour is white (1,1,1). The light position and colour will do important calculations for water and terrain in the shader file.

$$\begin{bmatrix} \tan^{-1}(\frac{FOVx}{2}) & 0 & 0 & 0 \\ 0 & \tan^{-1}(\frac{FOVy}{2}) & 0 & 0 \\ 0 & 0 & -\frac{Z_{far}+Z_{near}}{Z_{far}-Z_{near}} & -\frac{2\,(Z_{near}\,Z_{far})}{Z_{far}-Z_{near}} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Figure 2 Projection Matrix



1.3 Vertex Shader and Fragment Shader

Vertex shader and Fragment shader are two type of shaders in GLSL. The vertex shader takes VAO model data as input and execute once for each vertex in the object that is being rendered. It outputs value for each vertex. The fragment shader is then take those values as input, execute one time for every pixel the object covered on the screen, calculate and output the final colour of the pixel. In this program, terrain, skybox and water object have their own shader files. All the shaders will take object position, texture coordinates and normal vectors. Uniform variables such as any transformation matrix, projection matrix, view matrix, model matrix, camera and light position will also be loaded from ShaderProgram (see below) to do the lighting calculation in the shader files.

1.4 Reneder and Shader Program

The Terrain, Skybox and Water object all have their specific shader and renderer class, but what they do are similar: the shader classes get the location of the uniform variables used in shader files and wait for the real time values from the renderer to load them into the GLSL. The shader classes are also the children of an abstract ShaderProgram class which binds the input variables of the vertex shader (glBindAttributeLocation()), activate and link the two shader programs (glCreateProgram(), glLinkProgram(), glValidateProgram()), start and stop the shader program (glUseProgram()). The renderer classes then use these shader objects to render shaders and textures.

2. Terrain

Terrain is a mesh of vertices with texture on it. It evenly spanned over the spaces (i.e. the x and z coordinates). For a flat terrain the y coordinate for each vertex is simply 0, for some hilly terrains however, the y coordinate need to be set to some positive value so a height is given. The terrain will then looks like there are hills and valleys on it.

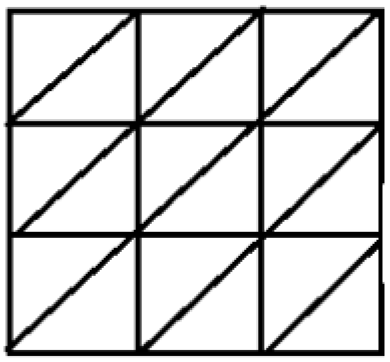


Figure 4 Sample terrain mesh

Figure 5 Terrain texture

In the world of the scene, the terrains are tiled in a grid. Each terrain tile has the same size (i.e. same number of vertices along its edge), and has its own positions in the plane which determines the terrain positions in the world (the coordinate (i, j) inside each terrain tile). So to generate such a tiled terrain in our world, first a Terrain class is needed to specify all the vertex positions (by assigning x = j * size / (num_of_vertex_along_edge), z = i * size / (num_of_vertex_along_edge), y value being the height will be set later), texture coordinates and normal vectors (for a flat terrain at first, all normals are (0,1,0)). Then a TerrainRenderer class will load and bind the texture to the whole grid. However, as the texture image is stretched to cover the whole grid, the resolution will

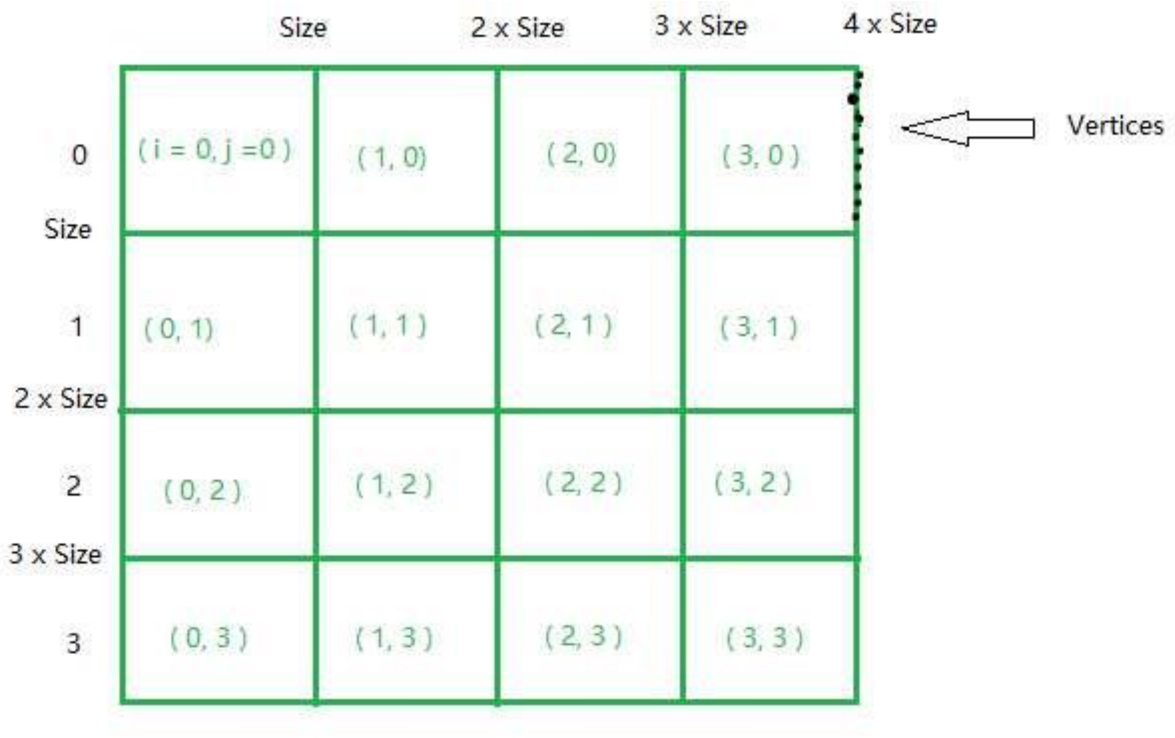


Figure 6: Terrain grid

become pretty low. This problem can be solved by making the texture image texturing each terrain tile, then the whole terrain grid will be that the same textures tiling together. To achieve that, simply times each texture coordinates to a value x, when the new texture coordinate is larger than 1, OpenGL will make that pixel to texture from the origin of the texture image. (i.e. texture for texture coordinate (1.1,0) is the same as (0.1, 0).) In this case 40 was chosen as x value, so there (128 / 40) * (128 / 40) = 9 full textures tilling together in the whole grid.

As mentioned before, to make the terrain hilly, the y coordinate need to assign some height values. A heightmap was adopted to cater for the needs. A heightmap is a grayscale image where the colour of each pixel on the heightmap represents the height of vertex on the corresponding place on the terrain. We use the height of the heightmap image to be the number of vertices on the edges of a terrain tile. To get the colour value of each pixel, simply use BufferImage.getRGB method for a given x and y coordinate on of the terrain. The actual height of each terrain vertex is just normalising the colour value and scale it again to make it look suitable on the scene, and keep that value bounded by a maximum height. For terrain vertex positions which are not within the range of heightmap's height, simply set their height to 0.

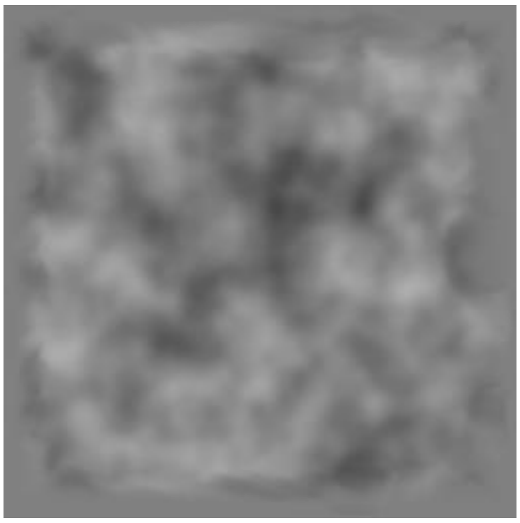
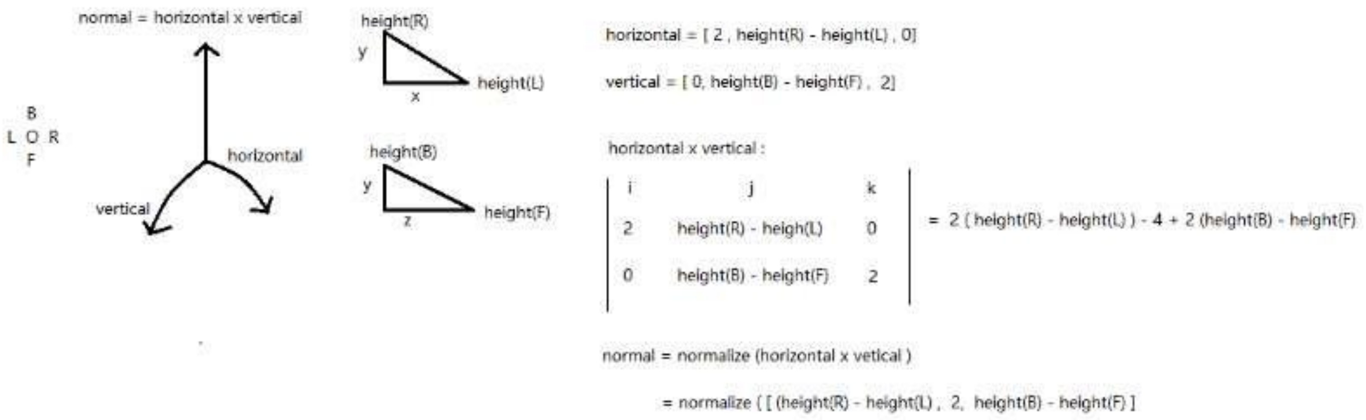


Figure 7: HeightMap

After creating such a hilly terrain, there is one problem left. The lights are constant over the whole terrain which is unrealistic. There should be some shades at regions of valleys or gully if overlook at a height. The lighting system is this world is determined by the light vector which is the position of the light minus the world (entity) position, plus the normal vector of the surface of that entity. If the dot product of these (normalized) two vectors is close to 1, it means they are quite parallel so the lighting on that part of the entity should be very bright. If they are close to 0, the two vectors are pointing quite different angles so the lighting should be darker at that region. Previously when the terrain's height was not given, the normal vector at any position of the terrain surface was (0,1,0), namely pointing straight upwards. Hence, to solve that problem, new normal vectors should be calculated for each vertex. There is one approximate way of doing this which is known "finite difference method". It uses the height difference of left and right neighbour of a given vertex to approximate the partial derivative vector in the x direction, and similarly a partial derivative vector in the y direction. The normal of the given vertex can be approximated by taking the cross product of the two partial derivative vectors.



$$\text{horizontal} = [\,2\,,\ \text{height(R)} - \text{height(L)}\,,\ 0]$$

$$\text{vertical} = [\,0,\ \text{height(B)} - \text{height(F)}\,,\ 2]$$

horizontal x vertical :

$$\begin{vmatrix} i & j & k \\ 2 & \text{height(R)} - \text{heigh(L)} & 0 \\ 0 & \text{height(B)} - \text{height(F)} & 2 \end{vmatrix} = 2\,(\,\text{height(R)} - \text{height(L)}\,) - 4 + 2\,(\text{height(B)} - \text{height(F)}$$

$$\text{normal} = \text{normalize (horizontal x vetical)}$$

$$= \text{normalize}\,(\,[\,(\text{height(R)} - \text{height(L)}\,,\ 2,\ \text{height(B)} - \text{height(F)}\,]$$

Figure 8: "finite difference method" for terrain normal

3. Skybox

The sky in the scene is also called a skybox. It is rendered by cube map textures. A cube map is a 3D cube where each face is a 2D texture, the whole box is counted as one texture with 6 sections (i.e. 6

separate sky images). The skybox was rendered as follows: Firstly a method is needed to use PNG Decoder to load the image into a byte buffer, and returned it as a TextureData object which stores the buffer as well as the height and width of the image. An empty textures was created by glGenTextures(). It will be activated and bounded by taking GL_TEXTURE_CUBE_MAP as a parameter into the glBindTexture method. glTexImage method taking TextureData is then used to texture the 6 faces of the skybox. When texturing the skybox, the texture coordinates is no longer a 2D coordinates in normal 2D texturing but a 3D coordinates being the same as vertex positions ((0,0,0) is the centre of the skybox). The vertex and fragment shaders of skybox does not contain other methods except determining the gl_Position and texture the cubeMap. The skyboxVertex file's input are all the vertices' positions of the cube, since there are 6 faces, each face is composed of 2 triangles, in total there are 12 triangles with 36 vertices. These vertices were specified in a SkyboxRenderer class, which renders the shaders and textures. One last thing of the skybox is that the edges should not be reached, when moving the camera around. It is expected that the camera is always "fixed" at the centre of the skybox. To do that simply remove the ViewMatrix in the skyboxVertexShader, (before was: gl_position = projectionMatrix * viewMatrix * vec4(position, 1.0)), thus the skybox's position is totally unaffected by the camera, instead of moving opposite towards camera's direction like the other objects do.

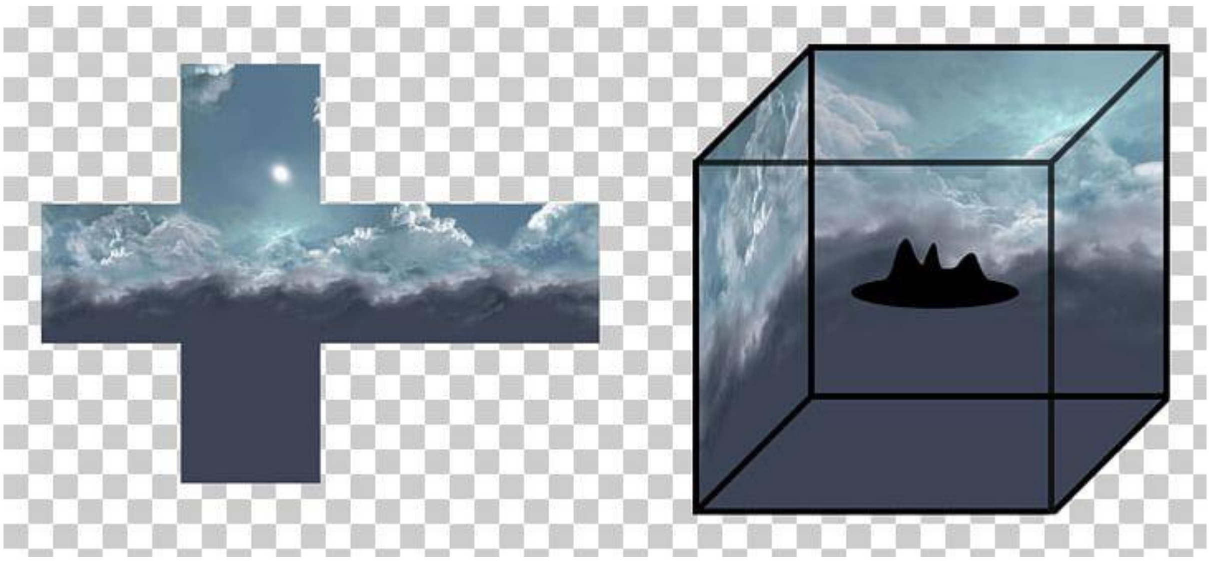


Figure 9 Sample skybox

4. Water

4.1 FBO & Reflection & Refraction

As introduced above, Reflection and refraction effects are both considered when rendering the water. An important concept which relates strongly with such rendering is the Frame Buffer Object (Fbos). Simply speaking, an fbo is the location where rendering data of an object stores before the display is updated to make them appear. Usually two buffers namely colour buffer and depth buffer were attached to an fbo. The colour buffer is basically a 2D array of pixel colours which will be shown on the screen, while the depth buffer stores depth information of each pixel in the colour buffer. So in every rendering the object will first be checked about the depth information in the depth buffer, the colour buffer will then discard some pixels which are behind and hidden by other pixels. The advantage of applying fbo in this rendering is that, besides the default fbo in OpenGL, another two fbos can be made to stores the textures in colour buffer and depth buffer. In this scenario, these will be reflection texture and refraction texture. The 2 textures will be rendered to the 2 fbos when bounded, and rendered to the default fbo when they are unbounded. The scene will then have water with both reflection and refraction textures from the default fbo (only from default fbo data will be rendered to display).

The reflection texture will render everything above the water surface while the refraction texture will render everything below the water surface. To achieve that, firstly glTexImage2D() method was called to give an empty texture which will be the up side down image of the scene. Those textures were attached with respective fbo by glFramebufferTexture(). Next, a tool called clip plane was used to cull the rendering. The reflection texture will then become the scene with stuff below the water surface being culled, and everything below water is the refraction texture.

4.2 dUdV

To make the water appear rippling, the texture coordinates needs to be distorted. A simple way is to add a constant vector to the reflection texture coordinate. For example, a vector vec2(1,0,0) will move the texture coordinate to the left so when texturing each time, the quad looks like a bit rippling. To get more complex and vigorous effect, a dUdV map comes into play. A dUdV map gives the offset of refraction in a texture using R, G, B colours with which representing one dimension. It is going to represent the distortion of different points on the water surface. The red and green offset value of each pixel can then be used to add to the reflection and refraction texture coordinates. In the water fragment shader, the dUdV map is called as the usual texture by distortion = texture(dudvMap, vec2(textureCoords.x, textureCoords.y)).rg. Then the refraction texture coordinate and reflection texture coordinate can add to that to make the water distorted.To control the amount of distortion, a constant value can be used to multiple the distortion to give the more realistic effect. Moreover, to create the effect that the water is moving, the water fragment shader can have a uniform variable which loads from water Renderer and adds to the x and y texture coordinate of the dUdV. This will move where the dUdV is sampled and makes the water looks like moving.

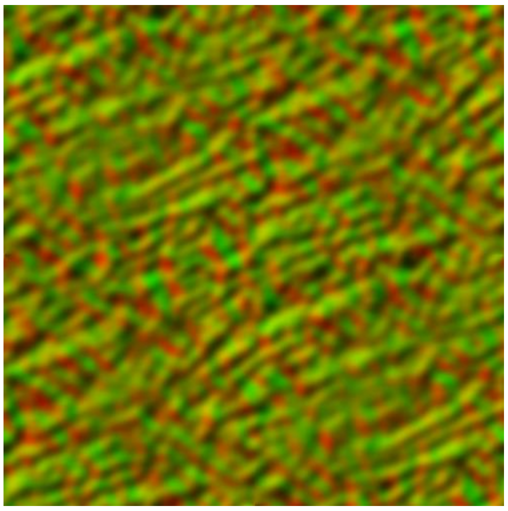


Figure10 dUdV map

4.3 Fresenel Effect

The Fresenel Effect describes how the reflectivity of water changes depending on the view. The water should look more transparent when looking above, and more reflective from a lower view angle. This effect is simply relates to two vectors, one is the normal vector of the surface, and the vector from camera to surface being the another one. The surface normal is always pointing upwards, if these two vectors are quite apart, it means the camera is located at a low angle and vice versa. Hence, the dot product is again calculated to measure the extent of diverseness. The value can then be taken as a factor to mix the reflection texture and refraction texture, by using the mix(reflectColuor, refractColour, factor) method.

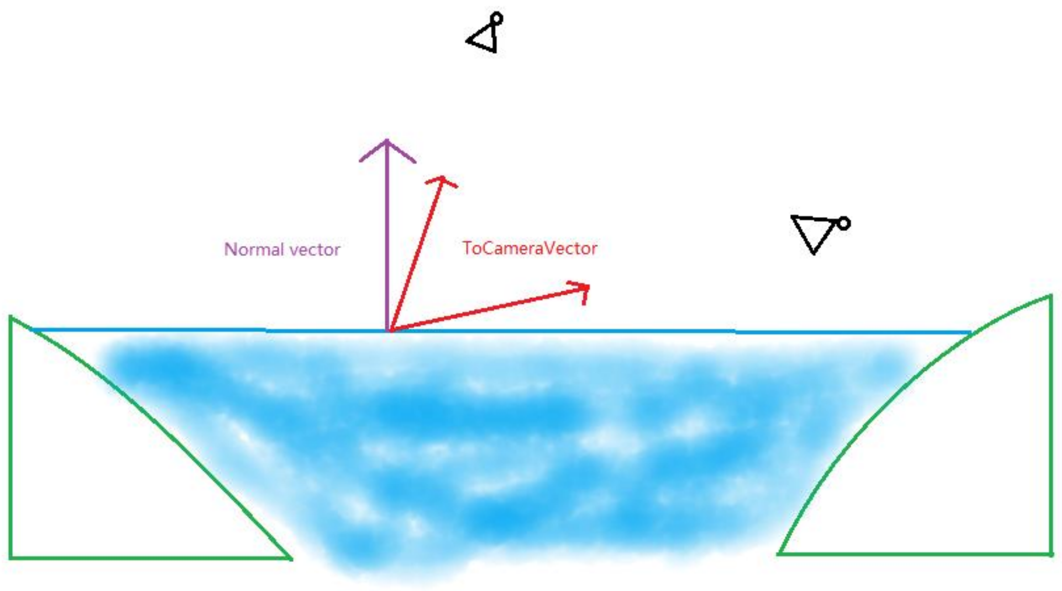


Figure 11 Fresenel Effect

Results and Discussion

Firstly, the program was running smoothly on a windows machine which supports up to OpenGL4.5 and with 16 Gigabytes RAM.

As shown in the picture, there are actually two terrains connecting together at origin, each with size 800 x 800. All vertices on the terrain grid were given a height value being either positive or negative with the help of the height map, which makes it looks like hills or valleys in the scene. Although the grass images are tiled consecutively over the terrain grid, It is hard to spot any repetitive patterns of the grass when looking closely. This will make the scene more realistic and less boring if applied as a game background.

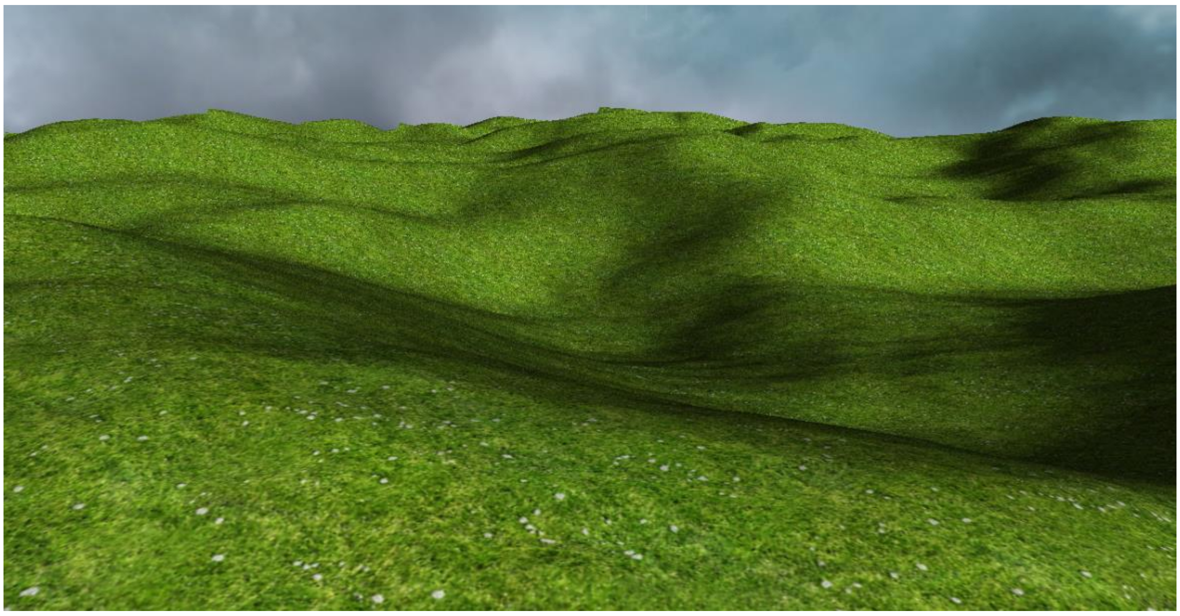


Figure 12 Terrain

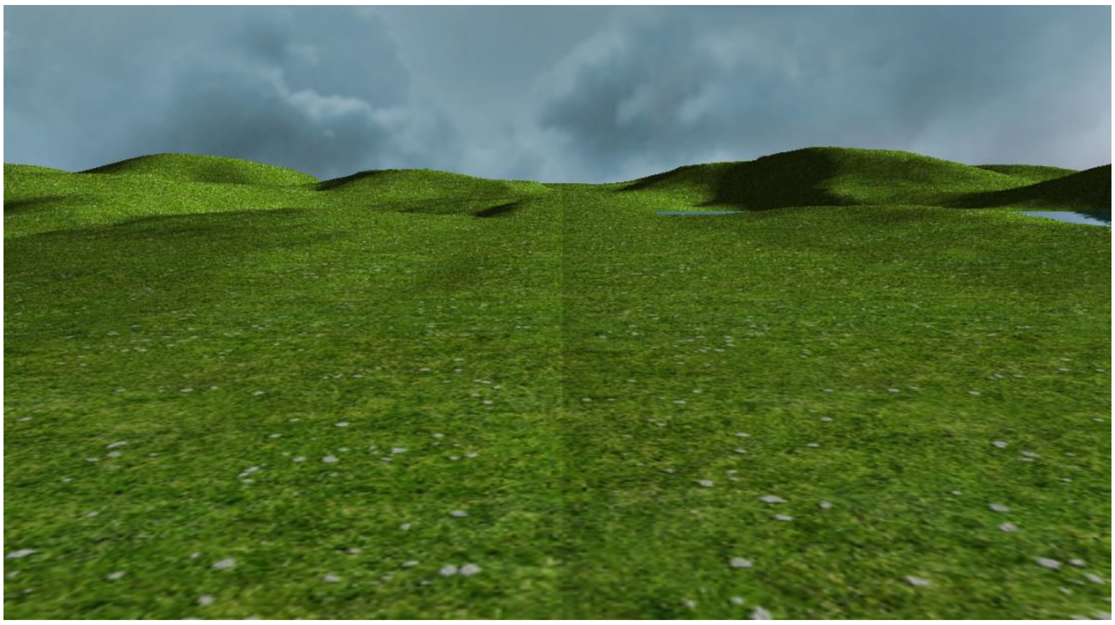


Figure 13 Two terrains connects at origin

The shades effect also works on the terrains. As shown in the picture, the shades mostly exist at the ridges where sun has less light rays directly cast onto. The “finite difference method” works well on approximating the normal of terrain surfaces.

Hence, in general, the terrain was rendered successfully.

As the picture of scene shows, the sky gives a cloudy and little gloomy backgrounds of the scene. The six images texturing the faces of the skybox are consistent thus hard to spot any edges nor the existence of the box. It is also a good effect that the edges of the skybox can never be reached, due to the transformation with the view matrix so the camera is always at the center of the box. The skybox’s rendering also reached the goal and able to apply to real game’s background.

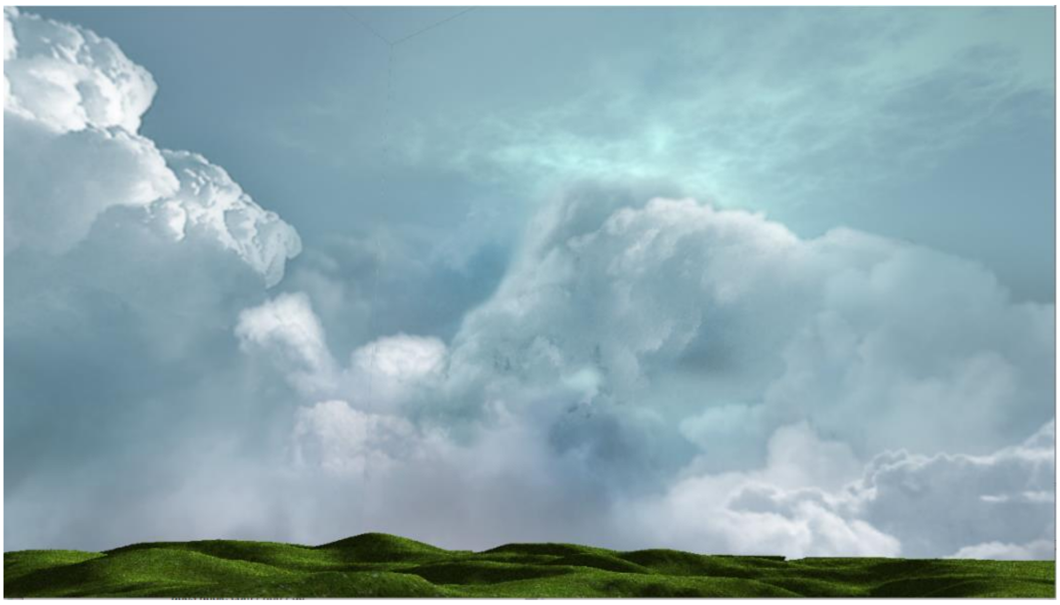


Figure14 sky

The water rendering was hard in this scene. Firstly it is necessary to find a suitable place to put the water quad to make it fit well with the surrounding terrains. The position for the water quad is (70, -70,0). At such place it looks like a flowing creek. The successful part of this rendering is that the moving and rippling effect are obvious, and the reflection was also successfully rendered. However, the refraction effect is almost unable to observe, the quad looks more like a mirror only showing the reflection of the sky.

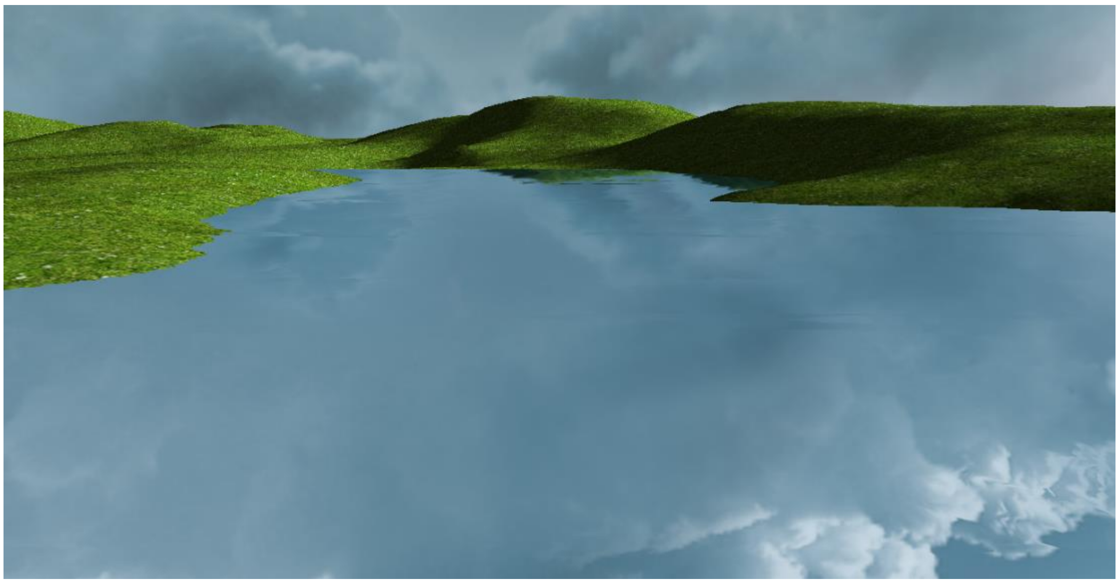


Figure 15 Water rippling

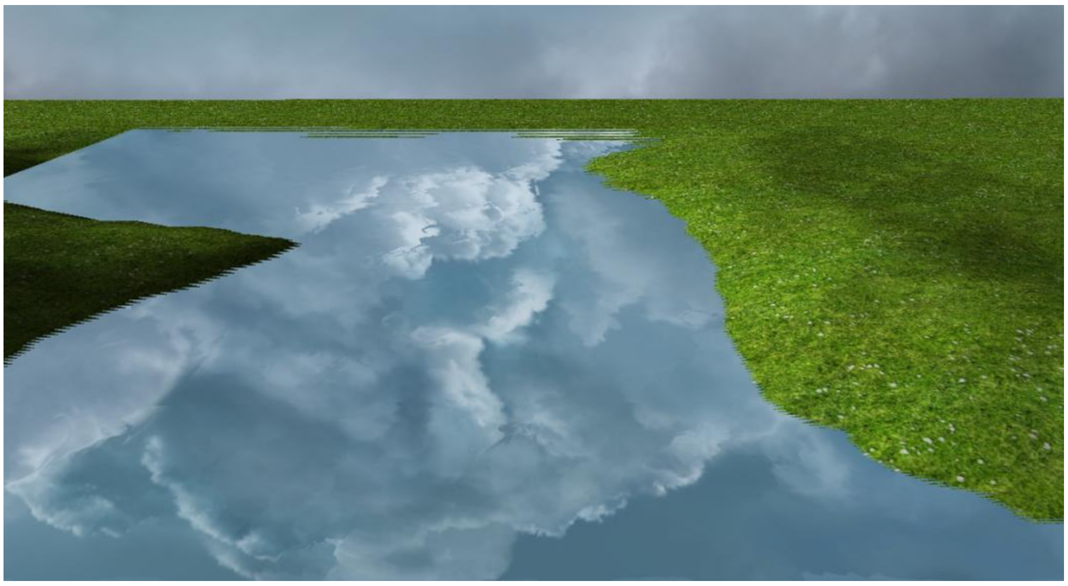


Figure 16 Water reflection

There are a few points where the scene can be improved or extended.

Firstly, the terrain is only the tiling of grass textures. One can think of adding more entities to make the terrain more vivid such as trees and shrubs. Another 3D modelling software may be needed to create those entities for the sake of reality. One issue after the rendering of 3D objects is the height of those 3D objects. Since the terrain uses a height map for its height value, the entities should be given the similar height if they were to put on the terrain and avoid collision. More objects will also make the water's reflection effect nicer.

Secondly, as can be observed, the sky is currently static as the cube map sample textures with the static texture coordinates. In reality, the clouds are more likely to move towards a certain direction with a speed. If one would like to achieve that effect, the technique which adds constant offset to the texture coordinates could possibly be adopted, similar to the water rippling effect.

The water should be improved as well. As argued before, the refraction effect is almost invisible. One possible reason could be the height of the water quad. The height namely the y value of the quad was set to 0, as illustrated before, when the refraction texture wants to texture the scene under the water surface using the clip plane, it possibly could not find something "meaningful" to texture. The terrain's height at that region could be also without any negative height, so the scene under the water surface could not capture the scene at the bottom of that terrain. The pattern of the terrain texture could also be a potential reason as it is quite simple comparing to the sky texture. Otherwise, some other more feasible methods should be taken completely.

The other issue related to water is its edges. As can be seen from the pictures above, the edges are sharp and one can think of ways to make them smoother when colliding with the terrain.

Conclusion

This project created a simple and basic 3D scenery containing terrain, sky and water. The rendering effect of terrain and sky are generally successfully as well as the reflection part of the water. The refraction part of the rendering however, fail to work properly somehow. The project also describes some techniques and structure of the rendering software. To apply this scenery to be the background in the real game, some suggestions and directions of improving were also argued.

Reference

1 https://imgbin.com/png/vbEw7F2h/skybox-shader-rain-cloud-png
2 https://www.techradar.com/au/news/gaming/the-evolution-of-3d-games-700995
3 http://www.codinglabs.net/article_world_view_projection_matrix.aspx
4 https://www.chinedufn.com/3d-webgl-basic-water-tutorial/
5 http://wiki.polycount.com/wiki/DuDv_map