# A Small Town

Mengyi Luo

M22luo

ID 20470845

July 24, 2017

# Contents

CS488/688

# Final Project

The purpose of this project was to implement a very simple RPG game by using OpenGL. The final goal is to meet all the goals and requirements that I set up for myself to make my project look like a very professional game.

## 1. Topics

- Adding Textures
- Bring Models
- The Third Person Camera - a special View Matrix
- Some Graphical Topics with Different Shaders
- Game Logic and UI design

## 2. Statement

A Small Town is a RPG game that allows player to control a character in the scene and overcome a challenge to win the game. Generally speaking , in my game a player is able to control a ball by keyboard inputs and let the ball collect eggs. When the egg's amount reach a certain amount, the player win the game. During the process, the player should be aware of an enemy - turtle. If the player is touched by a turtle, the total amount of collection on egg will be deducted, if it's zero, the player will lose the game. The basic requirement from the game logic is that the player should be able to use required inputs to interact with the game and the game should be able to give an appropriate display to illustrate the view or process of the game. So the game should be able to detect keyboard events to change the ball's position and a good camera position should also be controllable by mouse and be flexible to describe the scene. From the perspective of Computer Graphics, my scene in the game should be as much realistic as possible, otherwise the player wouldn't easily to understand the settings. The models and light should be well set up as a "common sense" to player. So the basic elements I chose to implement the game are texture mapping, shadow mapping, transparency, reflectivity and lens flare.

A Small Town is a very interesting game and challenging subject. It brings the player into the scene by making the scene as realistic as possible. It requires a lot of interactions with the game. One feature it has over OpenGL is that because it requires moving all the time and this should be reflected on the screen; player can also see the scene in different angles and it's totally decided by the player, so it will be like a "Controllable Animation" to player. Another nice feature of the game is that it has an interaction with songs. The BGM will be changed depending on the process of the game and the action of the ball.

# 3. Goals

I hope to implement a very simple and completed game for this project that at least has a "you win" window form and "you die" window form. The goals that I am aiming to achieve are the following:

Initially, I will make sure my texture mapping is working. Texture mapping is the simplest objective in the project, but it is also the most common objective we will apply in the project. Almost every chosen objective related to Computer Graphics will apply the technique from texture mapping, for example, reflection map use the texture mapping from skybox; lens flare is a series of 2D texture mapping that overlay to each other, etc. I will use A0 The Triangle assignment to test the functionality, and make sure my texture mapping is simply working in 2D. Usually the A0 to A4 assignment didn't require a lot of change on vertex shader and fragment shader. Thanks to this objective that let me get to know much details in shaders process that we make a new buffer in our code to store texture coordinate data and transfer texture coordinate info from out code to vertex shader and then to fragment shader.

My next goal is to modify my A3 assignment and successfully brings my new models in, including plane, ball, and water. Since I'm building a 3D game, so A3 Puppet assignment is the best starting point. I will search some free obj available models from some websites and let my code to read them. In this step, I will know what exactly an Obj file is; what the base structure are needed for an obj file. From my research that my code will need a group values for v, f, vn and vt; f should be much careful that f can only take 3 vertices because we use a triangle to describe a face. Some models has 4 vertices for a face, if we don't check that, my model will have some holes. Please see the

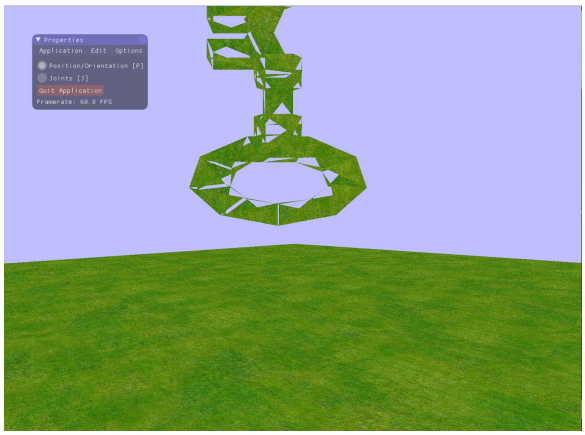


image on the right. In this goal, I also studied how to use Blender. This software is really helpful to make any models. More importantly, it will make sure your obj files have v,vn,vt and triangle f.

My third goal is to bring make my ball move and it's all controlled by player and bring in the third person camera. This is more complicated to achieve than the first two. It will require some calculations on object. For example, moving a ball require the game updates the position of the ball in real time. We need to collect the information of ball's current position and apply translation matrix to the current position and reset this to the model. This step is also required to work with keyboard events catchers. If key W is being pressed, the ball should move forward until the player take the hand off the keyboard. S is for moving backward. Key A and D is for rotating the ball either to the left or the right. The other tricky part is that camera's position would also be updated based on the location of the ball. Beside that, mouse is also an input source to transform the view matrix.

My fourth goal is let one object touch the other object and make some change to the scene to illustrate the change. Once an object is able to move, means that we can keep updating the position of an object; At least, we can track the position of an object. Therefore, we can take two positions and compare. Since the game is in three dimension, we need to use x, y, and z. If the player touch the egg, the egg should disappear. Then we will have an extra attribute *visibility*. If visibility is true, we will render the model; if it's not, we don't draw it. After those goals above are done, I have gained some basic knowledge of I will try other computer graphics technique including skybox, particle system, shadow mapping, lens flare, transparency and reflection.

My fifth goal is to bring the beautiful UI design. This requires a little bit of photoshop skills. By the way, display the graphic component on the screen is another 2D texture mapping technique.

My sixth goal is to bring the music in. Apply irrKlang library to make interactive songs. Choose some music that fit my game, and play it while the game is on.

My final goal is to make sure my game is a complete game on time. Make sure my game has a "you win" situation and "you die" situation. My game is allowed to restart and end. No bugs to stuck my game. Make sure all my basic objective are done. If possible, bring some new objectives.

# 4. Communication

**Input :** just run ./A5, no other inputs

**Interaction:**

| Keyboard | Mouse |
|---|---|
| **[Enter]** start the game<br>**[W]** moving forward<br>**[S]** moving backward<br>**[A]** rotate to the left<br>**[D]** rotate to the right<br>**[Space]** jump<br>**[Q]** quit the game | **[left key]** move around the ball<br>**[scroll]** zoom in or zoom out<br>**[right key]** adjust the height of camera |

**Output:** None. you will just see the game

# 5. Modules

## Code

Beside the modules that provided in A3, I have used the following modules.

**stb_image.cpp** read image, and give out image's data, height and width.

**Texture.cpp** contains a texture class. The main class can create an instance of texture, setUp a special image and bind it anytime when we want to use it in draw(). It has correlation with stb_image.cpp to read image's data.

**Object.cpp** contains code to bind with a scenenode. It is used to trace the position of an object/Scenenode. More importantly, object.cpp will provide functions that easily change the transformation/movement of a Scenenode, including moving (translation), and rotation. It has a correlation with SceneNode.cpp. Collision detection function also takes two objects as parameters.

**Camera.cpp** contains code to calculate the position of camera based on player's position. Be able to keep updating the position of camera based on the movement of player. And it is also be able to generate a new view matrix to the main code.

**Sun.cpp** interacted with Texture.cpp since a sun is also a piece of image ( 2D Texture). Contains code to draw a sun.

**LensFlare.cpp** contains code to draw lens flare. Lens flare is series of 2D texture mapping. So it has correlation with Texture.cpp.

**Skybox.cpp** contains code to draw a skybox. It's a different way of drawing a texture, so it doesn't have correlation with Texture.cpp. It's a way of drawing cubemap.

**Particle.cpp** contains code to draw particle system. We used a texture to represent bubbles. So it also requires a correlation with Texture.cpp. But didn't used this in the code because the function didn't work properly.

**Shadow.cpp** contains shadowbox class to calculate light view matrix. However, didn't use it.

**Gui.cpp** contains code put Graphic User Interface components on the screen. Contains info of texture (used to determine which image), position (x, y) on a 2D screen and scale for size.

## Shaders

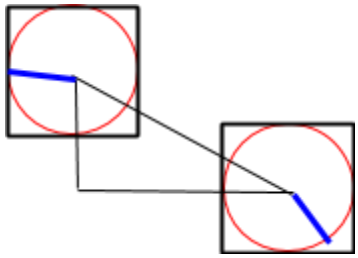
| Topics | Shaders |
|---|---|
| Tree, water, plane, houses, egg, earth, turtle | VertexShader.vs<br>FragmentShader.fs |
| Skybox | SkyboxVert.vs<br>SkyboxFrag.fs |
| Sun | sunVertex.vs<br>sunFragment.fs |
| Lens flare | flareVertex.cs |

| | |
|---|---|
| | flareFragment.fs |
| Shadow | shadowVS.vs<br>shadowFS.fs |
| Particle | ParticleVertexShader.vs<br>ParticleFragmentShader.fs |
| Gui | guiVertexShader.vs<br>guiFragmentShader.fs |

# 6. Technical Outline

This project will require implementing several algorithms. Most of them is the use of shader by GLSL. OpenGL 3.3+ will be used for its efficient buffers and multiple render targets. Personally, the challenge for me is how to successfully challenge the scene.

| Techniques | Explanation |
|---|---|
| Modeling the scene | The scene is settle down on a prairie (may or may not use perlin-noise). This requires texture mapping, skybox and imported other obj files to decorate my model.<br>Generally, I will use a lua file to generate the basic model. Some objects' model are described by obj file. The warning is that the obj file strictly requires values for 'v', 'vn', 'vt' and 'f'. Currently, I found that loading (too many) unused obj file before the main obj file would affect the quality of texture mapping. |
| UI | The user interface should be able to illustrate keyboard controls. Working with ImGUI framework. UI will direct players to see the difference of applying objectives. For example, player would be able to know the key or menu item to enable or disable the shadow mapping. |
| Texture Mapping | Make my objects in my game look reasonable. Objects including my ground, sky, player ball, water/lava. Used third party library called "stb_image.hpp" to load image data. The key point of this algorithm in OpenGl is the used of function glTexParameteri and some key parameters like GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T, GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MAG_FILTER, GL_REPEAT and so on. Texture binding on different objects should be well controlled. |
| Particle systems | I will apply particle system to label the position of keys. This will apply Buffer Object Streaming. Each particle object has three basic attributes. Position, speed and life. Each particle has its own position and direction to move, but not for a long time. We have to reserve a series of space on buffer, size in the maximum amount of particle. Frequently counting alive particles, finding unused particles and relocating them to be alive again. It will be a big buffer |

| | |
|---|---|
| | and update the value at run time. |
| Sound | When the player ball touch the key, it will make a background sound. Found a library called irrklang in an example. |
| Static collision detection | When the ball touch the key, it is a static collision detection, or called Radial Collision. It compared x and z position.<br>  <br>So it will be working on a 2d surface. Find a bounding box for each object and find the center of object and radius.<br>Pesudo code:<br>Bool Function Collision()<br>A = obj1.x - obj2.x<br>B = obj1.z - obj2.z<br>C = squareRoot(A^2 +B^2)<br>If C < obj1.size + obj2.size<br>Return true<br>Return false |
| Transparency | It will be illustrated either on the river or on the window of house. It depends which part can be done on time. In GLSL, when using texture mapping, fragment shader will call texture(sampler, texture coords) and it will return a vec4. The alpha is used to describe the transparency. |
| Shadows | All objects in the scene will have shadow due the light of sun. The technique will apply the shadow frame with frame buffer. |
| Reflection | It will be illustrated either on the river. “The reflection effect is a two pass effect. In the first pass, we draw in the stencil buffer the area in which the reflection should be visible and create a mask for drawing the reflection.” The reflection is the symmetry of the object in reference to the reflection area. |
| Lens flare | It’s another technique from texture mapping and required framework contains FreeImage 3.16.0, GLEW 1.11.0 |

# 7. Milestones

1 - Texture mapping, make sure it’s working

2 - Bring models, make sure I can bring any models I want. Make sure my texture is working with those models. Bring at least two models at the same scene

3 - Bring shadows

4 - Bring particle system. If two objects has collision, produce particle system
5 - Bring skybox and water plane, make sure my transparency and reflection is working
6 - Implement game logic
7 - Decorate the scene
8 - Bring gui, make sure it is playable
9 - Bring music songs

# 8.Organization

All files will be found in the subdirectories under the A5 directory. The subdirectories and files are indicated in the following.

A5
 |--- files--- : Main A5, Camera, GeometryNode, SceneNode, Gui, LensFlare, Object,
 |  Particle, Sound, Shadow, Skybox, stb_image, Sun, Texture
 |--- Directory Assets
  | ---- gui
  | --- gui images including gamestart.png, youlose.png and youwin.png
  | ---- house
  | --- house obj model and diffuse png
  | ---- lensFlare
  | --- flares images
  | ---- models
  | --- All obj models we need for our game
  | ---- shaders
  | --- include every fragment shader and vertex shader files
  | ---- skybox
  | --- include at least 6 skybox images
  | ---- sound
  | --- include all sound mp3 files
  | ---- terrain
  | --- include all textures for terrain including a blend map
  | ---- texture
  | --- include all textures for models in models folder
  | ---- tree
  | --- include obj file and diffuse png file for tree model
  | ---- turtle
  | --- include turtle obj file and diffuse png file

| ---file --- : ss.lua: luacode defines the game scene

# 9.Documentation

The file A5/README gives a brief installation guide. The file doc/README contains a user's guide to running the game.

# 10.Implementation

## Data Structures

To implement this game, I created several special classes to make my game is organizable when coding it. First of all, please let me introduce some new but basic attributes we need for A5.

| Type | Name | Purpose |
|---|---|---|
| Texture | xxx_tex | Used create texture. Be able to generate a valid Texture ID and bind the texture whenever after it called setUp() |
| ShaderProgram | m_xxx_shader | Used to link fragment shader and vertex shader. Draw specific model as it required to bind different buffers. Shaders are listed in 5.Modules |
| GLuint | m_vao_xxx | Used to bind with a vertex array |
| GLuint | m_vbo_xxx | Used to bind with a buffer |
| Skybox | skybox | Used to create a skybox |
| Sun | sun | Used to draw a sun |
| Camera | camera | Be able to generate a valid view matrix |
| Object | player | Used to link a scenenode that present our player |
| Object | egg,egg1,egg2 | Used to link eggs scenenode; Used to detect the position of eggs |

| | | |
|---|---|---|
| Vector <Object> | eggBasket | Easy to organize a group of eggs |
| Object | Turtle, turtle1, turtle2 | Same as above as egg Object |
| Vector<Object> | turtleBucket | Easy to organize a group of turtles |
| irrklang::ISound Engine * | bgm | Make bgm sound. |
| Gui | xxx | Graphic User interface components |
| vector<Gui> | guiBucket | Used to manage a group of gui, set up its position on display, make sure they are displayed properly |
| Particle[] | pConainer | Array of Particles instance. Used to allocate or free particles. A manager of all particles. |
| Static GLfloat * | particle_position_size | Used to hold all particle information and transfer to GPU. |
| bool | x_button_pressed | Used to detect when a button is pressed. True when it's being pressed; false otherwise. |
| FlareTexture | xxx | Create a flare by position, texture image and scale |
| vector<FlareTex ture> | List_of_flares | Holding all information about flares; easy to manage. |
| bool | blended , normalized | The value will be transferred to fragment shader. If it's true to a model, then the model should apply blended or normalized mapping. |
| bool | blendMap_option, shadow_option | Put on gui. If it's true, show the effect after applying related mapping |

## Actual Implementation

Here I will explain in more detail what I did to meet all of my goals and to implement the features of my project. As I stated previously, I plan to do texture mapping first as it is the very basic technique required.

## 1.Texture Mapping

Generally texture mapping is really simple. There is not too many thing to describe. The basic idea is that except for transfering v(position), vn (normal) value to GPU, we should also transfer vt(texture coordinate) value to GPU. and interestingly the range should be [0, 1]. Left bottom corner is (0 , 0) and right-up corner is (1, 1). The critical thing (to a beginner) is that, make sure we write texture functions to be exactly four functions, namely:

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,, GL_LINEAR);
```

In fragment shader, we must have uniform sampler2D samTex and bind with a unit number, GL_TEXTURE0 or GL_TEXTURE1, to sampler2D. Missing any of steps will cause a black texture. It also possible to cause a white texture. While I spent 5 days to find out the answer on the website but there is no good answer as it is too rare. A white texture is a default texture. It means you successfully transfer the data and bind the texture with a correct unit number. However, for some reasons, the image info is gone. As a beginner, I was trying to find out if any of my parameters for above functions was wrong because there was no other way to make that happen. The final result turned out, I make a constructor too early. I should wait until everything set up, like shaders should be linked, all uniforms should be activated etc. Two warnings here, each texture mapping will generate an unique texture ID that will be used in texture() function in fragment shader. The other warning is that if you have more than one textures, you need to assign an unique unit number to uniform sampler that will be used for active a specular texture position. For example, using GL_TEXTURE0 will activate the texture on unit 0, GL_TEXTURE1 will activate the texture on unit 1 and so on. Missing or reassign number will cause the shader crash.

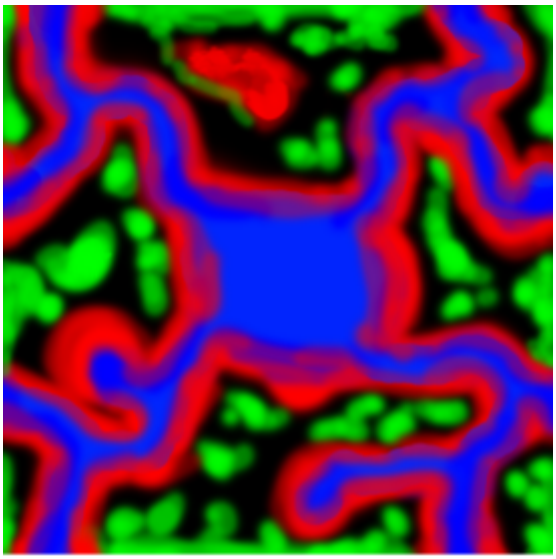


Figure 10 - 1 - 1

The other interesting thing about texture mapping is **multi-texturing** using blend map (figure 10 - 1 - 1). This is what I did for my plane. You can see that there are more than one texture on a single plane. We need to have a texture ID for each colour. And then we need to prepare another texture ID for blending map. In the blending map, there are only 4 colours, black (grass), red color(mud), green color(flower) and blue color(path). The fragment shader will read texture value and extra its' red, blue, and green value respectively to determine the transparency of mud, flower and path image. The final color of plane is the

sum of those colour.  See Figure 10 - 1- 2.

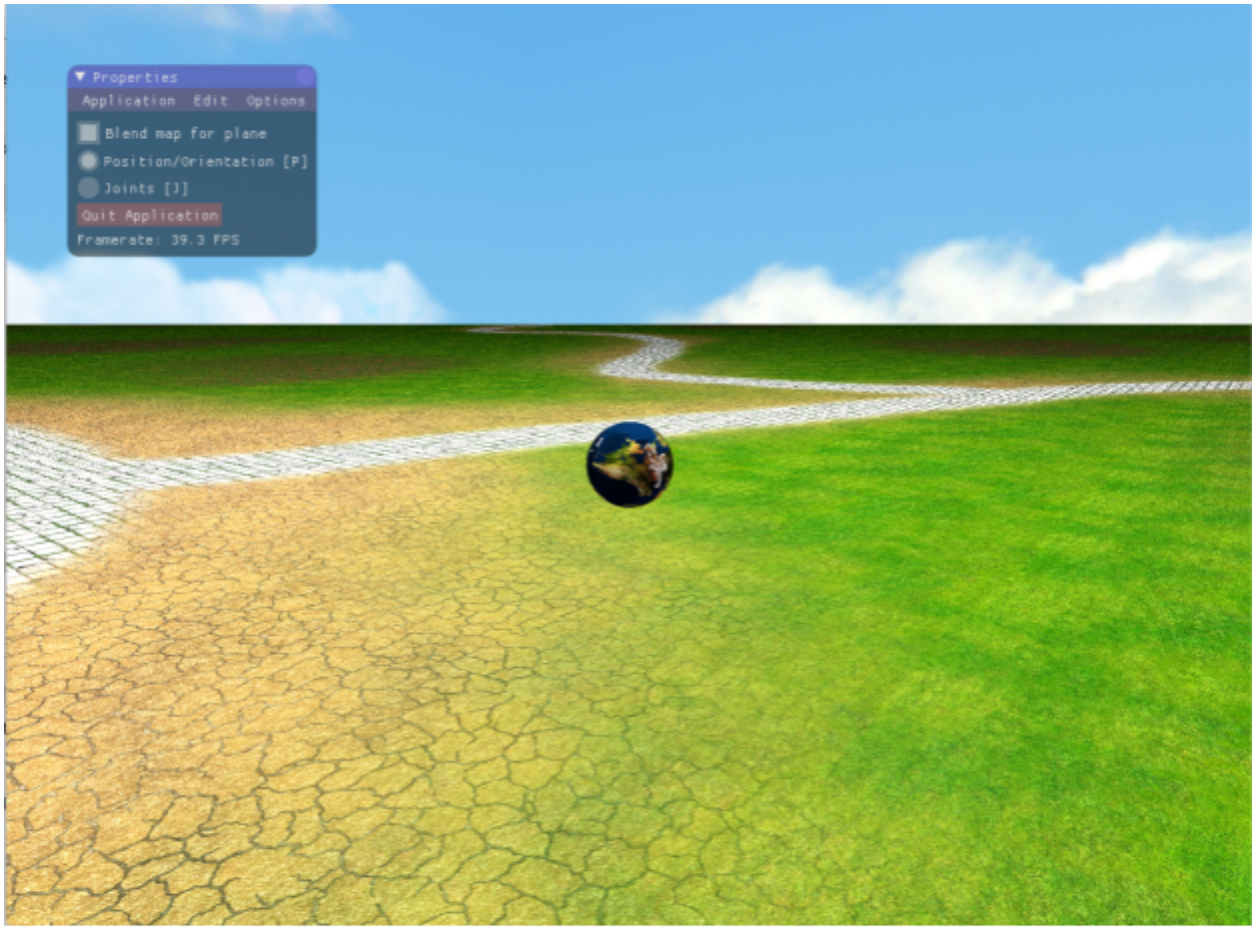



Figure 10 - 1 - 2

I will skip the loading models as it has been explained in Goals section.

**2. The third Person Camera**

The third person camera is a little bit of complicated as it requires knowledge of math. Camera.cpp as I mentioned has correlation with Object.cpp such that the game will keep tracking the position of the ball and updating the position of camera. Therefore, in the Object.cpp, there are some attributes that the camera class will need: the vector3 for position of the player and the rotation angle on y axis.

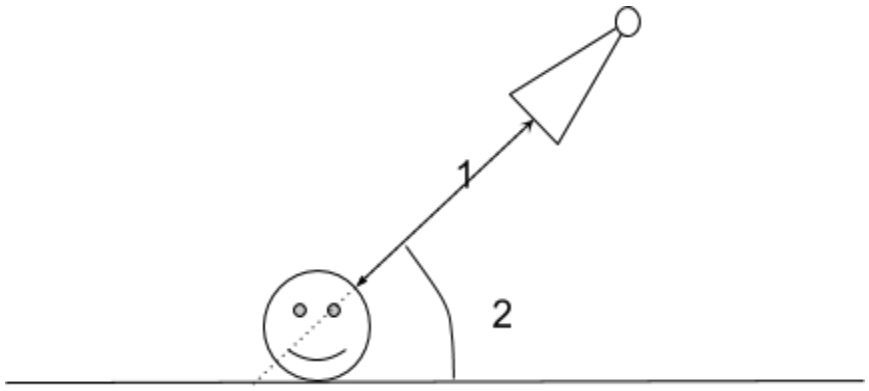



Figure 10 -2 - 1.1

Scroll the mouse will change the distance between camera and the ball, shown in number 1. Right mouse button will change the angle an 2. See figure 10-2-1.1 and figure 10-2-1.2.

Figure 10 - 2 - 1.2

Comparison between long distance and short distance. You can also see the blend map result for ground plane.

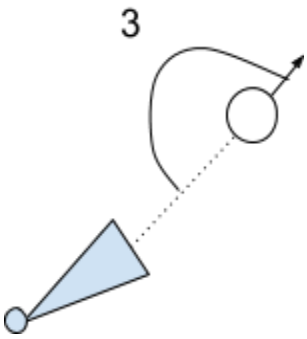



Figure 10 - 2 -2

Left mouse key will adjust the angle 3 between camera and the ball in figure 10 - 2 - 2. In the camera class, we have already have values for **pitch**, **yaw** and **roll**. And now, we will bring two more values, **distance (d)** between camera and player and **angle around player(a.a.p)**, and then update the value No.2 to pitch, and set up a default value for No.1. By using the **pitch ( θ )** value, we can calculate the horizontal distance (h = d * cos( θ )) and vertical distance (v = d * sin( θ )) by trigonometry. By getting these two values, we will then be able to calculate the position (x, z) of camera. x = h x sin(total angle) and z = h * cos(total angle), and total angle is the sum of Roty (rotation angle on y axis for player) and a.a.p (Total angle = roty + a.a.p). After we calculate the distance on 2D (x,z) between camera and player, we will add them together to calculate the position of camera. See figure 10 - 2 - 3.

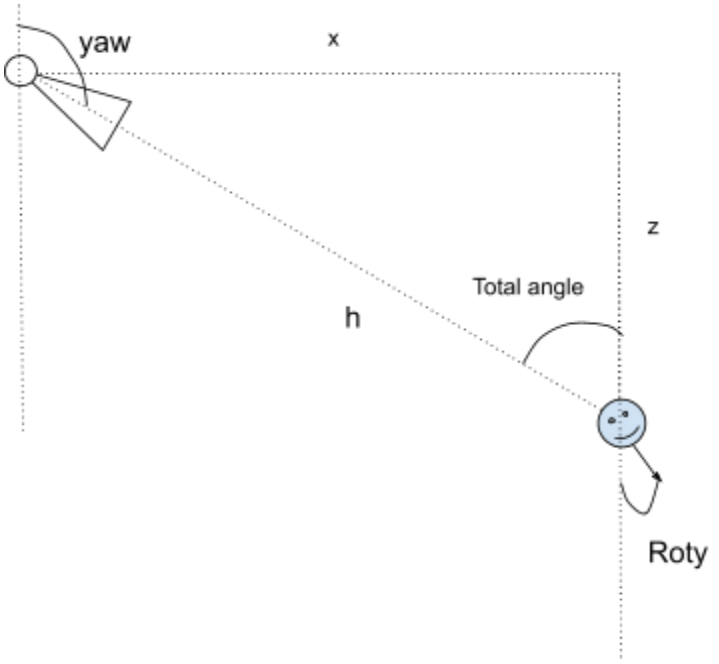



Figure 10 - 2 -3

We also need to update the yaw angle for camera by this yaw = 180 - total angle by parallel rule. Reference[1].

### 3. Shadow Mapping

I wanted to add shadow in my game is because shadow adds a great deal of realism to a lighted scene and make a spatial space to be easily observed and understandable. Overall, it would make my game looks really realistic. Shadow mapping requires two time's rendering the scene. The first time will detect the depth of each objects - namely, how each object is close to the camera. It will generate a depth map such that the objects that's closer to the camera will has a darker colour, the further gets the lighter colour. "Shadows are a result of the absence of light due to occlusion". If a light doesn't hit an object because some

other objects stays between the light and the object, then the object will be in shadow. We will used a depth buffer to store the depth value [0, 1] from a camera's point of view to a fragment. The closer the smaller value. Then, we store all these depth value in a texture and we call it depth map or shadow map. In my code, it's called **depthTexture**. Generally speaking, if we render a fragment at point p, we first need to determine if it's in a shadow. We first need to find a **Light View Matrix**, and put the point p into the matrix. Then get its z coordinate value which corresponds to depth value. We might also get another visible fragment that has a smaller depth value from one single light ray, and that means the point with smaller depth value will be higher than the one with larger number, and make the larger one in shadow. See Figure 10 - 3 - 1.

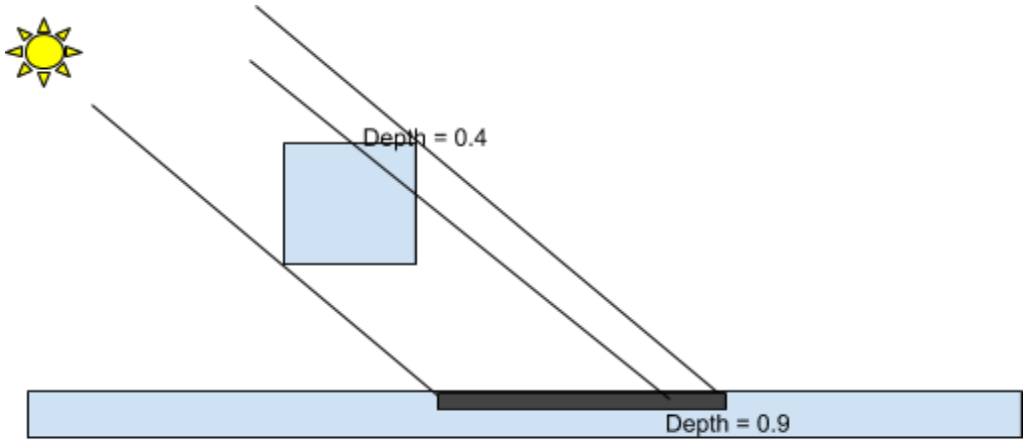



Figure 10 - 3 - 1

Since we are store values for a scene, we will use frame buffer. In the second time, we will render our model scene normally and use the generated depth map to check whether the those fragments are in the shadow or not. See figure 10-3-2 the shadow mapping results. Also see Reference [2][5].

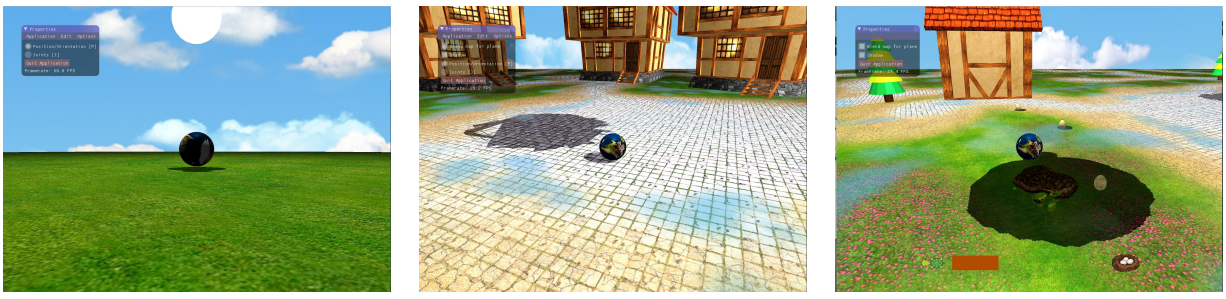


Figure 10 - 3 - 2

## 4. Reflection and Transparency - Skybox and Water

In order to implement a reflective map, we need to first implement a skybox. Skybox is a big cube such that every single models in our scene is put inside the box and we can observe a huge 2D texture mapping for 6 different faces. Therefore, for preparing valid images, we need to give 6 different orientational images of sky. It is easy and similar with texture mapping skills, except for binding parameter. We are binding with GL_TEXTURE_CUBE_MAP instead of GL_TEXTURE_2D. We need to put 6 individual 2D textures into an order with specific orientation. For example GL_TEXTUER_CUBE_MAP_POSITIVE_X is right and GL_TEXTURE_CUBE_MAP_NEGATIVE_X is left. After we read image's data, we put data to specific orientation:

```
data = stbi_load(textures_faces[i].c_str(), &width, &height, &nrChannels, 0); // read image
glTexImage2D( GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0, GL_RGB, width, height, 0, GL_RGB,
GL_UNSIGNED_BYTE, imagedata); // put data into specific orientation
```

We can simply add a water plane like adding a new model, just like what we did for terrain. Water also has its own texture. In order to make water to be transparent, we can simply change the alpha value to be less than 1 and greater than 0. Since in fragment shader, a fragment colour is represented in vector[r ,g, b, a], **a** represents the transparency. Beside that, since skybox is also another type of texture mapping, it would also generate a valid skybox texture ID. In reflection, we will transfer skybox texture ID to water's fragment shader and pass in as an uniform samplerCube parameter of texture(sampler, texture coordinates) function and call mix() function to generate a mix color of it's original color and reflected colour. You can see my FragmentShader.fs to find out the details. See Figure 10-4-1,2,3

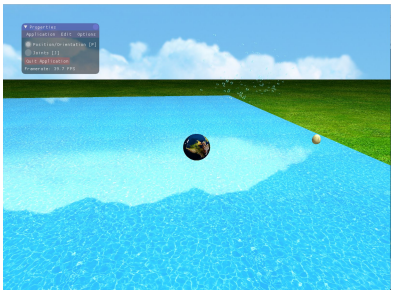


Figure 10 - 4 - 1
Water plane with water texture and reflected texture.

Figure 10 - 4 - 2
Water plane has transparency.

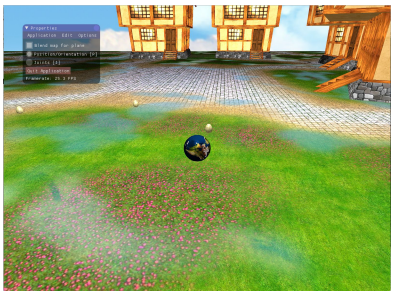


Figure 10 - 4 - 3
Water plane with blending map, split the water plane and make it like a scene after rain.

## 5. Particle System

We first created a particle class having position, velocity and life attributes. Each particle will have a random position and velocity. Life is usually short to particle, we count down from 1.0 to 0.0. In the particle system, we first preserve 100 spaces on buffer to generate particles. When we want generate some particles, we first find out a certain amount (let's say 10) and compare with the current amount of alive particles we have so far. If we need to generate more, we need to go over through each 100 positions on buffer by a pointer called **LastUsedParticle** to determine who need to be alive and who need to be dead. If the particle is alive, we just need to make sure that it's position is observable changed. If it's not, then we just need to change the position of particle to be far far away from camera. My particle system code will have a lot of similarity with Reference [3],[4],[5].

Figure 10-5-1 A single particle system

Figure 10-5-2 Two particle systems

## 6. Lens flare

Lens flare is a series of 2D texture mapping on the sky and if you shoot your camera at a specific direction, you will see an overlay textures that will look like lens flare. In every frame, we will calculate the 2D position of the light source on the display and then once we've done that we are going to take the imaginary line that goes from the light of 2D position through the center of the screen and then we render some 2D texture images all along this line with lens textures when rendering. We will also determine the brightness of the lens flare depending on the distance of the light from the center of the screen. For each

flare, we create a class that has scale (to determine the size), position (where we put the flare) and texture (what flare we want to use).

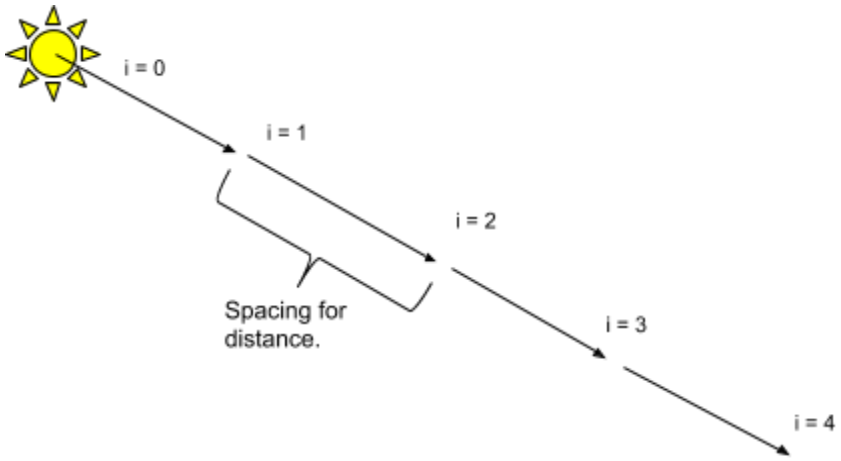



Figure 10-6-1

In order to do this we have several steps: first of all, we need to convert each objects from 3D position to 2D position. We have a sample code:

```
glm::vec2 A5::convertToScreenSpace(glm::vec3 worldPos, glm::mat4 viewMatrix, glm::mat4 perspectiveMatrix){
   glm::vec4 coords = glm::vec4(worldPos.x, worldPos.y, worldPos.z, 1.0f);
   coords = viewMatrix * coords;
   coords = perspectiveMatrix * coords;  //getting 3D position in Camera-view-matrix
   if (coords.w <= 0){
      return vec2(-999.0f);
   }
   float x = (coords.x / coords.w + 1.0f) / 2.0f;
   float y = 1.0f - ((coords.y / coords.w + 1.0f) / 2.0f);    //Convert 3D position to 2D position short it in range [-1, 1] for screen
display
   return glm::vec2(x,y);
}
```

Second of all, find each flare's 2d position by given sun's 2D position and sun's 2D direction. We can calculate each distance by using linear equation.

```
voidA5:: calcFlarePositions(glm::vec2 sunToCenter, glm::vec2 sunCoords){
   for (int i = 0; i < List_of_flares.size(); i++){
      glm::vec2 direction = glm::vec2(sunToCenter);
      float spacing = 0.4f;
      direction = direction * ((i) * spacing);
      glm::vec2 flarePos = sunCoords + direction;
      List_of_flares[i].setScreenPos(flarePos);
   }
}
```

Then, we render each flare :

```
for (int i = 0; i < flares.size(); i++){
        Texture tmpFlare = flares[i].getTexture();    // getting the correct texture of flare
        tmpFlare.Bind(0);                             // bind the correct texture
        float xScale = flares[i].getScale();          // determin x,y scale
        float yScale = xScale * (m_windowWidth / m_windowHeigh t);
        glm::vec2 centerPos = flares[i].getScreenPos();    // getting 2D position

        glm::vec4 answer = glm::vec4(centerPos.x,centerPos.y , xScale , yScale);

        location = m_flare_shader.getUniformLocation("transform");
        glUniform4fv(location,1, value_ptr(answer));

        glDrawArrays(GL_TRIANGLE_STRIP, 0,4);
    }
```

The result wasn't satisfying and I need to do more research:

Figure 10-6-2 The camera is too far away from the model.

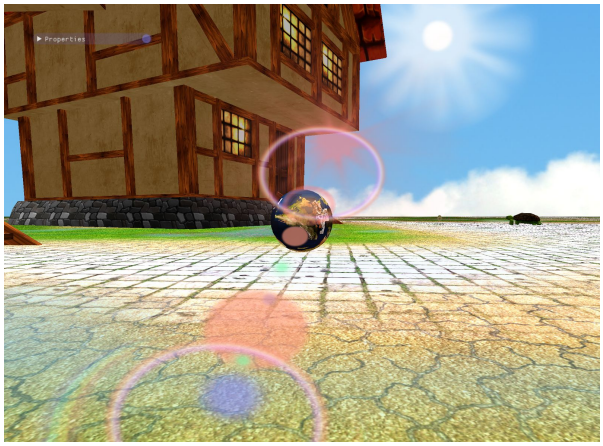


Figure 10-6-3 when the camera is set up in a good position. But this is still not a perfect flare such that the flare wasn't look like a light flare, just some pictures. And the last part of flare is missing.

## 7. Collision Detection

When the ball touch the key, it is a static collision detection, or called Radial Collision. It compared x and z position. See Figure 10-7-1.

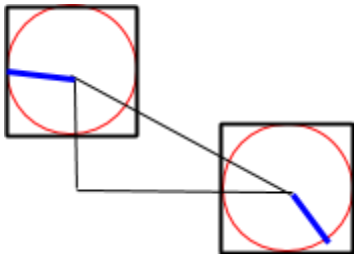


Figure 10-7-1

We find a bounding box for each object and find the center of object and its radius. Using this function to determine if they are touching to each other. If the distance between two object's center is smaller than the sum of radius, it will return true as it's touching to each other. Otherwise, return false.

```
bool A5::collision(Object obj1, Object obj2){
   if(obj1.isVisible() && obj2.isVisible()){
      float A = obj1.getObjectPosition().x - obj2.getObjectPosition().x;
      float B = obj1.getObjectPosition().z - obj2.getObjectPosition().z;
      float D = obj1.getObjectPosition().y - obj2.getObjectPosition().y;
      float C = sqrt(A*A + B*B);
      float E = sqrt(B*B + D*D);
      if (C < obj1.getObjectSize() + obj2.getObjectSize() && E < obj1.getObjectSize() + obj2.getObjectSize()){
         return true;
      }
      else return false;
   }
   return false;
}
```

Static and dynamic collision detection both use this formula, as the important thing is not to determine whether object's moving or not, it is what position of the object is currently at.

## 8. GUI - UI

GUI components that I made for my game is fairly another 2D texture mapping that has a fixed position on the display screen. It requires a little bit photoshop skills to design the interface.


Figure 10-8-1 Game start interface to illustrate player operations.

Figure 10-8-2 You lose window, can either press enter to restart or quit the game.


Figure 10-8-3 You win, press either enter to restart or q to quit

We just need to show those image at a certain time at position(0,0) with scale of (m_windowWidth, m_windowHeight).

One fun fact is the winning bar is interactive on the screen. If it's about to win, it will has a green long bar; If it's about to lose, it will show a short red bar. This technique is all done in guiVertexShader.vs and guiFragmentShader.fs by calculating the length and color of this rectangular at real time.

In renderGui():

```
mat4 mvpMat = mat4();

float lifebar_width = 0.4;
float lifebar_height = 0.05;
float lifeW = ((float)life/10) * lifebar_width;  // calculate a certain width that will use later to transform into
float move = -lifebar_width + lifeW;             // the correct position
// life bar
mvpMat *= glm::translate(mat4(), vec3(move, -0.8, 0.0));
mvpMat *=  glm::scale(mat4(), vec3( lifeW, lifebar_height, 0.0));

GLint location = m_gui_shader.getUniformLocation("transformationMatrix");
glUniformMatrix4fv(location, 1, GL_FALSE, value_ptr(mvpMat));

location = m_gui_shader.getUniformLocation("life");
glUniform1i(location, 1);
location = m_gui_shader.getUniformLocation("lifeValue");
```

```
glUniform1f(location, life/10.0);
glDrawArrays(GL_TRIANGLE_STRIP, 0,4);
```

Figure 10-8-4 Initially player has 3 lives (has 3 eggs)

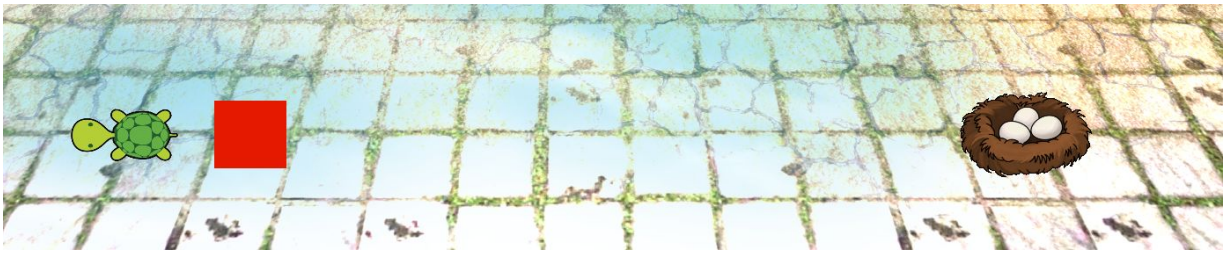
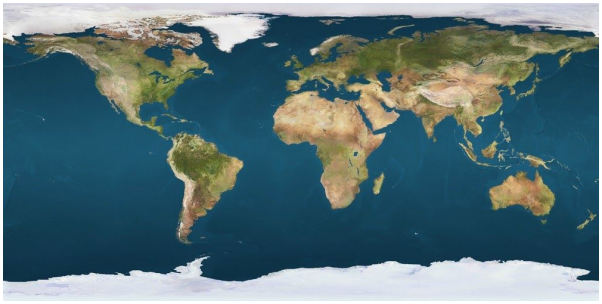


Figure 10-8-5 Player has only one life and about to lose. Need to get more eggs to win.

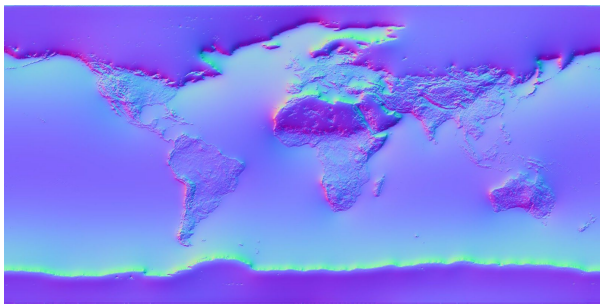
Figure 10-8-6 Player just needs to collecting two more eggs to win

## 9. Normal Mapping

Normal mapping is an extra objective that I added to my work in last two hours before deadline. I just quickly studied from Reference[6]. Basically we need Tangent and Bitangent for each vertex, and change the normal direction based on noraml map. In normal map, we use this formula normal = (2 * color) -1 to translate color value [0,1] we read from normal map to normal [-1, 1]. Generally, the texture map has a blue tone that means the normal is towards the “outside of the surface”.

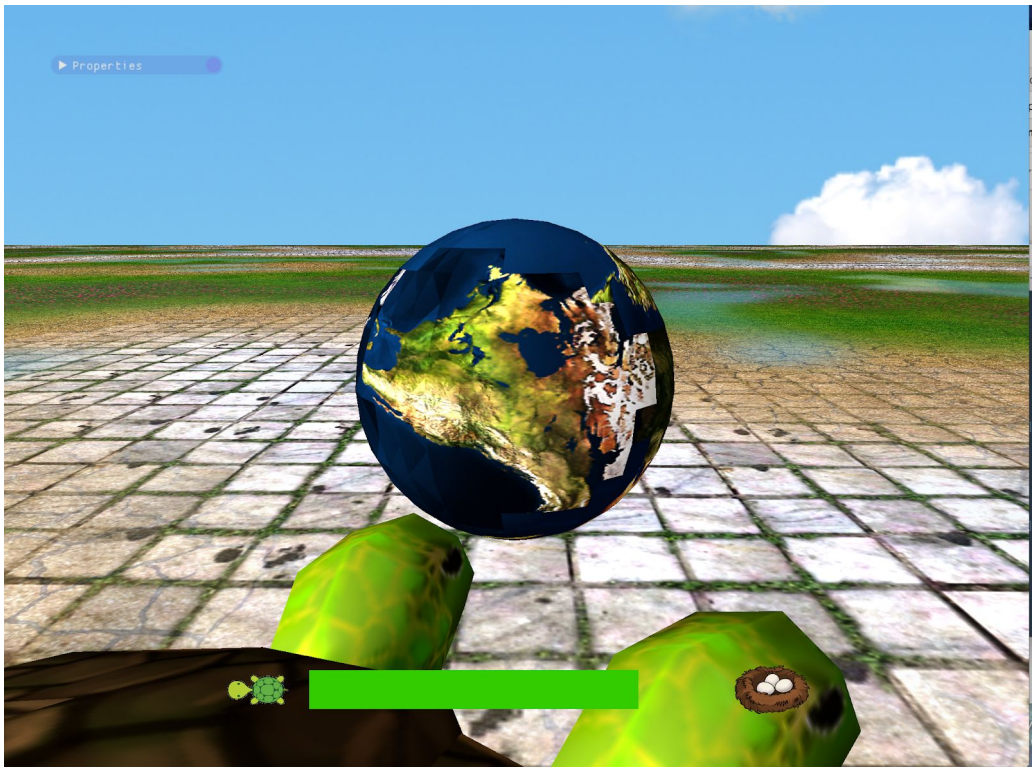


Earth normal texture image



Earth normal mapping image

You can find the different texture on earth from different light position.




# 11. Manual

The program was compiled and runs on Mac OSX, with CMAKE downloaded and has premake4 executable file. The main executable is called A5. I havn't implement any real error-handling if the user enters invalid inputs it won't be caught. In fact all uncertainty were a huge challenge to me, because it happened so unusually. Like one of my skybox image, it has a glitch without any reasons. I have checked that my executable cannot pass the compiler in CS488 lab. Some of uniform didn't pass the compiler in my shaders.

Images are screenshot .png files, and I have shown a lot of them in 10.Implematation section. I will just give a brief description about my screenshots

**Screenshot01.png** : The image demonstrates that I have successfully implement texture mapping, model the scene and lens flare.

**Screenshot02.png**: You can see the use of third person camera. Observing the distance between player and eggs.

**Screenshot03.png**: The image illustrates that I had a shadow mapping. However, it didn't work so well and unfortunately, this is the best I can do. You might also be able to see water effect such that water reflect the image about skybox.

**Screenshot04.png**: There is another view of water from different perspective. You can see how different on reflection of water.

**GameStart.png YouLose.png, YouWin.png**: illustrate that this game is a completed game.

**Screen Shot 2017-07-14 at 5.25.47 PM**: illustrate that I have successfully implemented particle system.

# Sources

In **Project**, **premake4 gmake** and **make**. In **A5**, the command **make** will compile my project and produce an exEcutable called **A5.**

# Executable

My executable is named **./A5**

# 12. Bibliography


[1] Karl (2014, Nov 15) *Third Person Camera, ThinMatrix.* Retrieved from : https://www.youtube.com/watch?v=PoxDDZmctnU&list=PLRIWtICgwaX0u7Rf9zkZhLoLuZVfUksDP&index=19

[2]Joey de Vries (2014, June) *Shadow Mapping, Learn OpenGL* Retrieved from: https://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping

[3]Opengl-Tutorial (2017, Jun) *Particle System, Opengl-Tutorial*. Retrieved from http://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/

[4]Joey de Vries (2014, June) *Particle System, Learn OpenGL.* Retrieved from https://learnopengl.com/#!In-Practice/2D-Game/Particles

[5]Zelda, CS488 OpenGL project Retrieve from https://github.com/longdtnguyen/zelda_openGL

[6]Opengl-Tutorial (2017, Jun) *Normal Mapping*, *Opengl-Tutorial* Retrieved from: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/

[7] Khronos Group, Chapter 9. Transformation, ***Opengl.org, 2016***
Reference: https://www.opengl.org/archives/resources/faq/technical/transformations.htm#tran0170

[8] Andrew S. Glassner *3D Computer Graphics: A User's Guide for Artists and Designers*, chapter 5 Lightning, OpenGL Programming Guide,*(New York: Design Press, 1989)*
Reference: http://www.glprogramming.com/red/chapter05.html