

COMP3055 Computer Graphics

Project Final Report

Chen Liao 20030694

1. Introduction and brief recap

● Introduction

The Taj Mahal is an enormous mausoleum complex commissioned in 1632 by the Mughal emperor Shah Jahan to house the remains of his beloved wife. Now the place has become a great place of interest in India for visitors worldwide. The major idea of this project is to replicate the construction of the Taj Mahal in 3D modeling with OpenGL based on C++ programming.

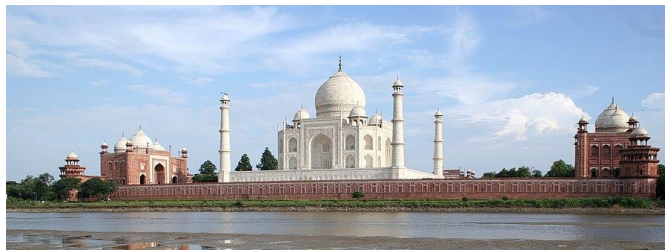


Figure 1: Landscape of the Taj Mahal

The overall landscape of the spot will be revived, including the main building, two side buildings, the fountain, the plants, and the garden.

● Specificities

1. Necessary data will be measured in Google Earth. A more simplified version of the model will be designed.
2. The overall code structure will be specified, following the basic idea of object orienting programming paradigm for C++.
3. Using hierarchical modeling to build the primitive model by using basic building blocks in OpenGL.
4. After finishing the main body, proper textures will be rendered to various objects.
5. Various animations will be implemented, mainly orienting the water system.
6. Lighting effect will be added, including ambient, diffuse, and specular
7. Diverse viewpoint will be designated to deliver a better presentation.

● Overall screen shot of the scene

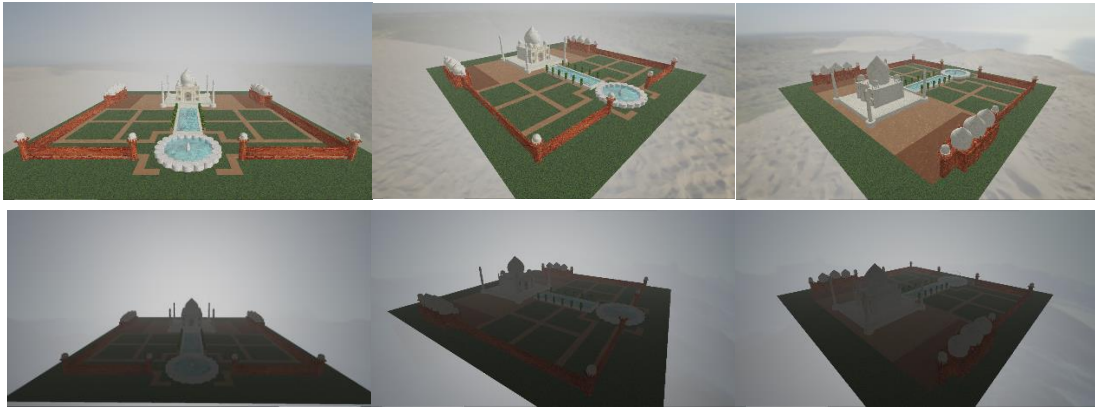


Figure 2: Overall project landscape in day mode (up) and in night mode (down)

2. How to use the program

● Dependencies

The project depends on one third-party, open-source library: `stb_image`. This library enables the program to read in images in different format compatibly. The actual use of this library will be discussed detailly in the fourth part, in this part, we only focus on how to install and activate this dependency.

`Stb_image` library follows pure head-file fashion, it implements all functions concretely in its head files. This means that we only need to include its source code into this project to depend on its functionalities rather than directly manipulating the project solution (.sln file). Firstly, its source code is linked to this project and included as a header as shown in figure 3.

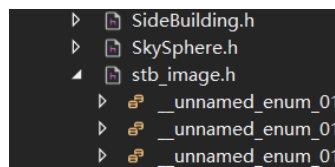


Figure 3: Header file of the `stb_image` library

`Stb_image` library also follows pre-compiling paradigm, which means that the header will be compiled ahead to make itself only contain relevant source code every time before the program is running. Therefore, a declaration is necessary before including the header. In this project, we use key word: `#define` in C++ to activate the pre-compiling process as shown in figure 4.

```
#include "Texture.h"
#define STB_IMAGE_IMPLEMENTATION
#include "stb_image.h"
#define GL_CLAMP_TO_EDGE 0x812F
```

Figure 4: Setting up environment for stb_image library

● Control the scene

The control of the program mainly lies on the control of camera perspectives. This program applies first-person-perspective camera, also known as FFP camera, its implementation details will be demonstrated in the following part. The FFP camera could allow people to fix the camera at an arbitrary position instead of some predefined ones. To do this, a set of instructions are provided to customize the camera:

- “w” on the keyboard moves the camera position forward.
- “s” on the keyboard moves the camera position backward.
- “a” on the keyboard moves the camera position leftward.
- “d” on the keyboard moves the camera position rightward.
- “q” on the keyboard moves the camera position upward.
- “e” on the keyboard moves the camera position downward.
- Mouse drag alters the direction the camera looks at.
- “Space” on the keyboard resets the camera to its original position.
- “n” on the keyboard switches the scene between day mode and night mode.
- “esc” on the keyboard controls the exit of the program.

3. How this program meets the requirements

● Hierarchical modeling (3D models and its transformations)

Most of the 3D objects in the scene is built through hierarchical modeling based on Glut library in OpenGL, including the main building, the side buildings, the towers, the walls, the fountains, and the water system. Here we take some building blocks as examples.

There are multiple object classes defined as building blocks, including roofs, pillars, and walls. The basic idea of drawing a curved roof is assembling a sphere and a cone following some geometric relationships as shown in Figure 5 (middle). Pillars are basically combined with cylinders with different top and bottom radiuses, as in figure 5 (left). In order to make a hollowed wall, a scaled cube and a triangled cylinder are combined to form the boundary of its surface and the inner carving as shown in figure 5 (right). Then, four of them form a gate with slopes downright into the inner carving. Corresponding 3D effects for these building blocks are demonstrated in figure 6.

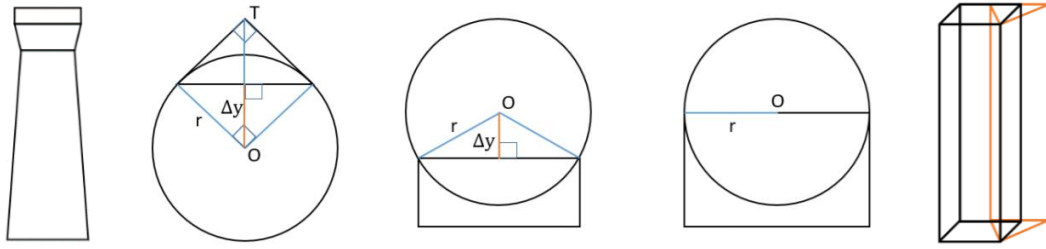


Figure 5: Geometric interpretations for some basic building blocks

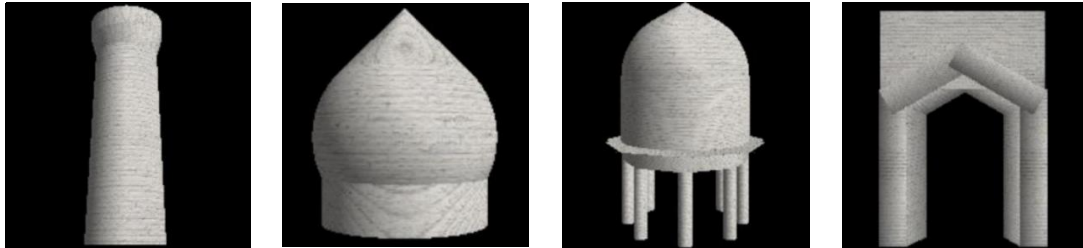


Figure 6: Corresponding 3D models in OpenGL environment

More complicated objects could build on these basic ones. For example, the main building aggregates five roofs in two types, four certain pillars and multiple hollowed walls with different sizes. Readers could refer to figure 7 to find out its decomposition.

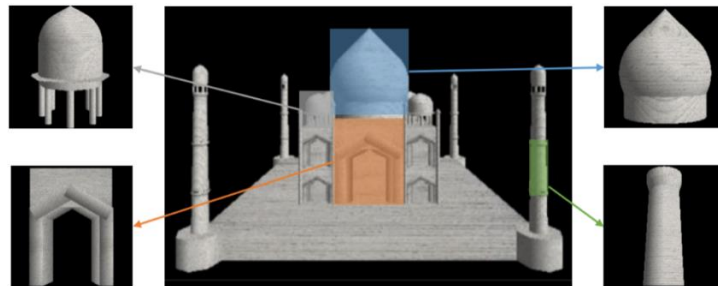


Figure 7: Main building decomposition

In addition to buildings, general implementation of the water system also relies on hierarchical modeling. However, instead of using Glu objects as for the buildings, water utilize GL Polygons. The basic idea of water implementation is to place multiple squares in juxtaposition forming a large surface. This will bring convenience to the later part when we add animation to mimic real water movements. Sample code for implementing water is shown in figure 8.

```

glBegin(GL_QUADS); // draw the water surface as quads
for (int j = 0; j < subdivDim; j++)
{
    s = -0.5f;
    for (int i = 0; i < subdivDim; i++)
    {
        glNormal3f(0.0f, 1.0f, 0.0f); // specify the quads normal

        // specify the first vertex coordinates and position relative to the start position x,y,z
        glVertex3f((float)sin(time + (double)s * 10 * radius + (float)sin(time + (double)s * 10 * radius, s));
        glVertex3f((float)sin(time + (double)s * 10 * radius + (float)sin(time + (double)s * 10 * radius, s));
        // specify the second vertex coordinates and position relative to the start position x,y,z
        glVertex3f((float)sin(time + (double)s * 10 * radius + (float)sin(time + (double)s * 10 * radius, s));
        glVertex3f((float)sin(time + (double)s * 10 * radius + (float)sin(time + (double)s * 10 * radius, s));
        // specify the third vertex coordinates and position relative to the start position x,y,z
        glVertex3f((float)sin(time + (double)s * 10 * radius + (float)sin(time + (double)s * 10 * radius, s));
        glVertex3f((float)sin(time + (double)s * 10 * radius + (float)sin(time + (double)s * 10 * radius, s));
        // specify the fourth vertex coordinates and position relative to the start position x,y,z
        glVertex3f((float)sin(time + (double)s * 10 * radius + (float)sin(time + (double)s * 10 * radius, s));
        glVertex3f((float)sin(time + (double)s * 10 * radius + (float)sin(time + (double)s * 10 * radius, s));
        // move the position of the quad up by one quadruple step in the s axis
        s = 1.0f / (float)subdivDim;
        // move the position of the quad up by one quadruple step in the s axis
    }
}
glEnd();

```

Figure 8: Sample code for implementing water surface under GL_QUADS

● Texturing

Extensive textures are deployed in this project to add verisimilitude to objects, ranging from marble to grass. There are mainly two techniques in achieving this. Firstly, object pointers are utilized. This technique is broadly used when applying to Glu objects. The basic idea is to initiate a `GLUQuadricObj` pointer before we use Glu objects. Then we could bind texture and shape to this pointer for display. A demo could be found in figure 9. The second technique involves object coordinates and texture coordinates, which is mainly deployed in polygons. Water, for example, binds texture with mapping polygon coordinates to texture coordinates, sample codes could be found in figure 10.

```
GLUQuadricObj* wall = gluNewQuadric();
gluQuadricTexture(wall, GLU_TRUE);
gluQuadricDrawStyle(wall, GLU_FILL);
glBindTexture(GL_TEXTURE_2D, texID);
```

Figure 9: A demo for binding texture using object pointers.

```
for (int j = 0; j <= xGridDims; j++)
for (int i = 0; i <= xGridDims; i++)
{
    // if the water is frozen then calculate texCoord based on the sample position
    // i+(xGridDims+1)*j gives the texture position of the ith sample on the jth row
    texCoords[(i + (xGridDims + 1) * j) * 2 + 0] = (float)i / (float)xGridDims;
    texCoords[(i + (xGridDims + 1) * j) * 2 + 1] = (float)j / (float)xGridDims;
}
```

Figure 10: Sample code for binding texture using coordinate mapping

As different in nature, various textures are selected. Most of the walls in Taj Mahal uses ivory marble. Therefore, a corresponding marble texture is bind to them with a surface color of ivory (RGB (1.0, 1.0, 0.96)). Some parts of the side building use red tiles on the surface, therefore, textures resembling red tiles are bind to them. Textures of middle-age carvings, bushes, woods, and grasslands are also deployed correspondingly. We kindly refer readers to figure 11 for the list of textures utilized.

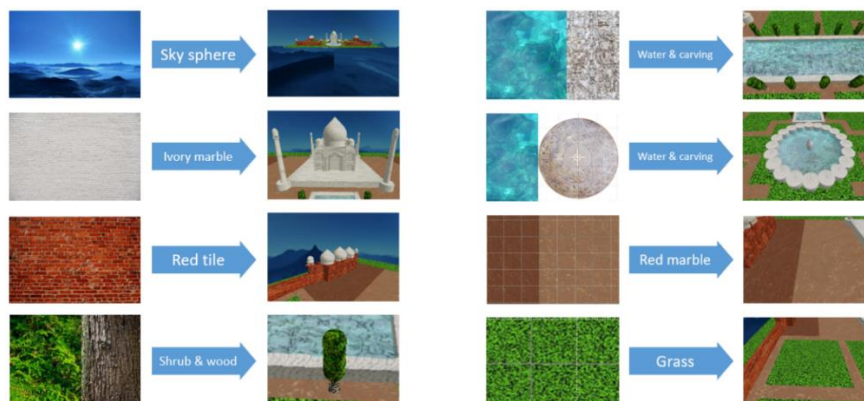


Figure 11: List of textures and their effects in the model

● Lighting

Standard OpenGL lighting system are utilized in this project. Following low-coupling programming, the setting up of the light system are separated and encapsulated in class

Scene. Ambient, diffuse, and specular, all three kinds of reflections are encompassed in this project. There are two different light settings, one is day light setting in the day mode, the other is the moon light setting in the night mode. Because this project intends to replicate an outdoor scene, directional light instead of spotlight are deployed to mimic natural lighting effect. Figure 12 shows specific values of parameters for the two settings.

```
GLfloat ambienceDay[] = { 0.4f, 0.4f, 0.4f, 1.0f };
GLfloat diffuseDay[] = { 0.6f, 0.6f, 0.6f, 1.0f };
GLfloat specularDay[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat positionDay[] = { 1.0f, 1.0f, 1.0f, 0.0f };

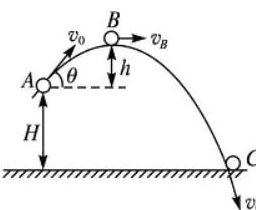
GLfloat ambienceNight[] = { 0.4f, 0.4f, 0.4f, 0.2f };
GLfloat diffuseNight[] = { 0.6f, 0.6f, 0.6f, 0.2f };
GLfloat specularNight[] = { 1.0f, 1.0f, 1.0f, 0.2f };
GLfloat positionNight[] = { 1.0f, 1.0f, 1.0f, 0.0f };
```

Figure 12: parameters for two light settings

● Animation

Animations in this project aggregates on water system of the whole landscape. There are two kinds of animations, one is the water-spraying effect of the fountain, the other is the wave simulation of the water surface. Since the project heavily relies on the GLUT library, no effective particle effect rendering pipeline is available to implement the effect of spraying water. Therefore, it needs to be simulated with oblique throwing motion of small water drops orienting diverse directions in GLUT. For the specific trajectory of each water ball, we decompose the movement into two orthogonal directions: a motion with constant velocity along X axis, and a free fall motion along Y axis. Set 1 demonstrates how we could mathematically interpretate the two sub motions and calculate the exact position of a water drop. The animation is repeated periodically within the time bound of animationTime. Therefore, we could achieve a continuous spraying effect. Figure 13 shows the critical part of implementing this animation.

$$d_x = v_0 \sin(\theta) t$$

$$d_y = v_0 \cos(\theta) t + \frac{gt^2}{2}$$


Set 1: Mathematically interpretation of the oblique throwing motion

```
float g = 9.8;
aT = fmod(aT + static_cast<float>(deltaTime) * 2.0, 3.5f);
x = 2.5 * aT;
//std::cout << "pos[0]: " << pos[0] << std::endl;
y = 15.0 * aT - 0.5 * g * aT * aT;
//std::cout << "pos[1]: " << pos[1] << std::endl;
```

Figure 13: critical implementation of the animation

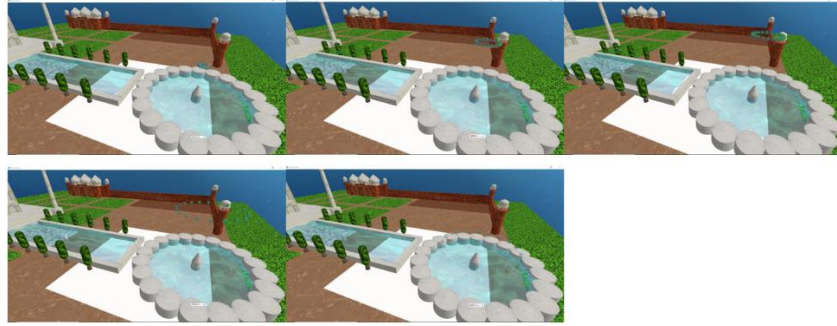


Figure 14: overall effect of the animation

The second animation lies on the wave simulation. Traditional water surface only animates through the dislocation of texturing binding on XoZ plane. Although delivering a visual effect of wave, fail in emulating its physical movement. Therefore, the animation along Y axis is necessary. We thus develop from the traditional animation. Instead of a whole piece of rectangle, the water surface was decomposed into many connected polygons which are bound to corresponding position of the texture, this is helpful in creating the variances in altitudes. The wave follows trigonometric functions in mathematics, $y = \sin(x)$ along positive direction and $y = \sin(z)$ in positive Z direction. Therefore, the altitude y of arbitrary position $p(x, z)$ is $y = \sin(x) + \cos(z)$. An intuitive illustration could be found in figure 15.

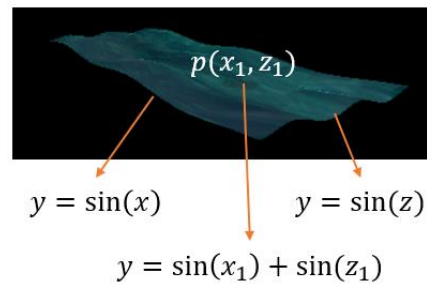


Figure 15: An intuitive illustration of the animation

● Viewpoint

In this project, first person perspective (FPP) is applied to the camera system. Instead of fixing viewpoints into several predefined places, FPP enables users to continuously move the camera and its perspective to arbitrary places. This adds more flexibility to the project. As mentioned in the previous chapter, the system intakes keyboard and mouse inputs to manipulate the camera. Sample codes could be referred to in figure 16.

```
void Camera::SetupCamera()
{
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(eyePosition[0], eyePosition[1], eyePosition[2],
             eyePosition[0] + vd[0], eyePosition[1] + vd[1], eyePosition[2] + vd[2],
             up[0], up[1], up[2]);
}
```

Figure 16: Camera system setup

● Code reliability

Considering the fact that many of these objects rely on duplicated building blocks, we thus define a bottom-up construction fashion. This means that following object-orienting programming, we first implement some basic object classes as building blocks, then we implement some upper object classes with instances from these rudimentary classes. This simplifies the implementation and reduces redundancy. An intuitive illustration of bottom-up code structure is shown in figure 17.

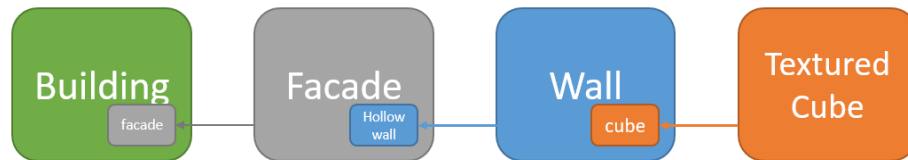


Figure 17: Sample of bottom-up code structure

Object orienting programming is strictly followed. To achieve low-coupling code structure, we encapsulated all object classes separately following similar templates. Higher order classes such as DisplayableObject, Animation and Input are extended, delivering generic functionalities to all object classes. In this project, addition, deletion, and modification of any object class will not affect other segment of the code, which ease the future development and constant maintenance. Readers could refer to figure 18 for some examples of object orienting programming and figure 19 for the overall class diagram.

```
// initiate sky sphere and ground
SkySphere* sky = new SkySphere0;
TexturedCube* ground = new TexturedCube("./texture/floor.png");

// initiate main building and two side buildings
MainBuilding* mb = new MainBuilding0;
SideBuilding* sbLeft = new SideBuilding0;
SideBuilding* sbRight = new SideBuilding0;

// initiate four towers along with the closure wall
Pillar* wallTower1 = new Pillar(3, "./texture/redWall.jpeg");
Pillar* wallTower2 = new Pillar(3, "./texture/redWall.jpeg");
Pillar* wallTower3 = new Pillar(3, "./texture/redWall.jpeg");
Pillar* wallTower4 = new Pillar(3, "./texture/redWall.jpeg");
Wall* wallLeft = new Wall(1, "./texture/redWall.jpeg");
Wall* wallRight = new Wall(1, "./texture/redWall.jpeg");
Wall* wallFL = new Wall(1, "./texture/redWall.jpeg");
Wall* wallFR = new Wall(1, "./texture/redWall.jpeg");
Wall* wallLFL = new Wall(1, "./texture/redWall.jpeg");
Wall* wallLFR = new Wall(1, "./texture/redWall.jpeg");
Wall* wallRFL = new Wall(1, "./texture/redWall.jpeg");
Wall* wallRFR = new Wall(1, "./texture/redWall.jpeg");

// initiate two fountains
Fountain* ft1 = new Fountain(1, "./texture/stoneWall3.bmp");
Fountain* ft2 = new Fountain(2, "./texture/stoneWall3.bmp");
Fountain* ft3 = new Fountain(2, "./texture/stoneWall3.bmp");
Fountain* ft4 = new Fountain(2, "./texture/stoneWall3.bmp");

// initiate two tree strips
Tree* ts1 = new Tree("./texture/wood.bmp",
                    "./texture/leaf.jpg", 6);
Tree* ts2 = new Tree("./texture/wood.bmp",
                    "./texture/leaf.jpg", 6);

// push objects into the object list
AddObjectToScene(text);
AddObjectToScene(sky);
AddObjectToScene(ground);
AddObjectToScene(ft1);
AddObjectToScene(ft2);
AddObjectToScene(water1);
AddObjectToScene(water2);
AddObjectToScene(sblLeft);
AddObjectToScene(sblRight);
AddObjectToScene(swallLeft);
AddObjectToScene(swallRight);
AddObjectToScene(swallFL);
AddObjectToScene(swallFR);
AddObjectToScene(swallLFL);
AddObjectToScene(swallLFR);
AddObjectToScene(swallRFL);
AddObjectToScene(swallRFR);
AddObjectToScene(ts1);
AddObjectToScene(ts2);
AddObjectToScene(mb);
AddObjectToScene(sbLeft);
AddObjectToScene(sbRight);
```

Figure 18: Samples of object orienting programming

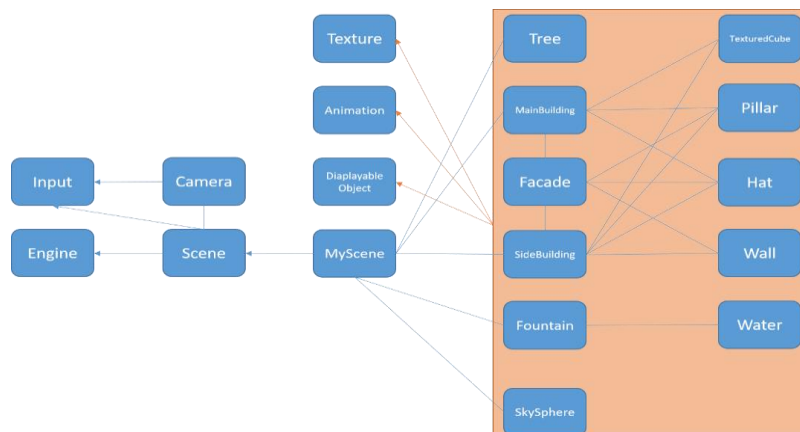


Figure 19: Overall class diagram of the project

Extensive and straightforward code comments are added. There are mainly two types of comments in this project, one is the header comments, appearing constantly at the top of each header file, illustrating the overview of the class, its functionalities, its category, and its dependencies, the other is inline comments, appearing arbitrary at places requiring further explanations. Figure 20 gives examples of a header comment and an in-text comment.

```
/* This class is a level 2 object class defining a fountain in the scene.
 * This class follows factory design pattern in object orienting programming
 * There are two kinds of fountain to be implemented:
 *   1. Round base fountain
 *   2. Rectangle baed fountain
 * Animations of the spraying water will be implemented in this class.
 *
 * category: instantiable object class, updateable animation class
 * Interface implemented: DiaplayableObject, Animation
 * Dependencies: Wall
 * Being depended by: MyScene
 */

//Constructor method (overload)
//Inputs:
// type: integer identifier for specifying the fountain type
// filename1: path of the first texture required by (@Wall) instance
// filename2: path of the second texture required by (@Water) instance
```

Figure 20: a header comment (up) and an in-text comment (down)

4. Creative ideas and extra contributions

● RGBA texturing

Because of the demanding texturing requirement in some part of the project. GRBA texturing involves besides normal RGB texturing. ‘A’ here stands for ‘Alpha’, different from RGB texturing, an RGBA texturing allows the texture itself to vary in its different parts of their transparency. In another word, one texture image could be fully transparent in some places, but visible in others. To achieve this, PNG texture format is required. Therefore, we need a new texture loader to read images in PNG format. In this project, stb_image library is deployed. This library allows us to load various format of image, including PNG. Therefore, the original BMP loader is replaced by stb_loader (see figure 21) which could deliver more texture choices for the project. For comparisons of different textures and the actual effect of RGBA textures, figure 22 could be referred to.

```
// flip the png image up-side-down due to its format
stbi_set_flip_vertically_on_load(true);
// use stbi loader to read in image file
pixelBuffer = stbi_load(fileName.c_str(), &width, &height, &channels, 0);
```

```
// Upload texture data
if (pixelBuffer) // selector: if has three channel, then RGB format, if four, then RGBA format.
{
    GLenum format;
    if (channels == 1)
        format = GL_RED;
    else if (channels == 3)
        format = GL_RGB;
    else if (channels == 4)
        format = GL_RGBA;

    glBindTexture(GL_TEXTURE_2D, texObject);
    glTexImage2D(GL_TEXTURE_2D, 0, format, width, height, 0, format, GL_UNSIGNED_BYTE, pixelBuffer);

    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, format == GL_RGBA ? GL_CLAMP_TO_EDGE : GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, format == GL_RGBA ? GL_CLAMP_TO_EDGE : GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
}
```

Figure 21: Implementation of the stb_loader

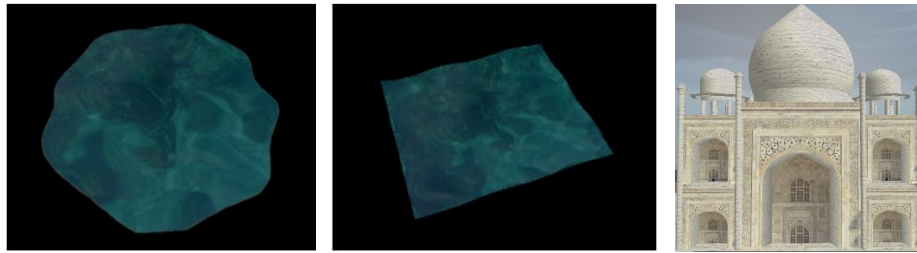
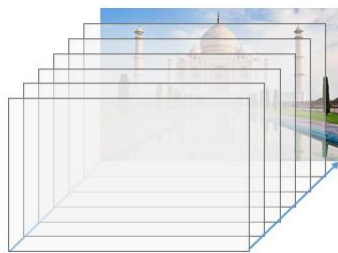


Figure 22: comparison between RGBA texture (left) and RGB texture (middle).
Another example of RGBA texture (right).

● The fog system

Because the Taj Mahal locates along the riverbank, a fog weather is common especially in the morning. Therefore, this is also replicated in this project to add verisimilitude. In this project, GL visibility system is utilized. The essence of the fog system in OpenGL is the decrease in visibility, which is a linear decrease with a factor α . Mathematically, it could be represented as: $V' = V \times (1 - n\alpha)$, where $n \propto d$, d is the distance between the position and the camera. A more intuitive interpretation could be found in figure 23. For implementation, we use `glBegin` to enable visibility model, then properties are bind to the model with `glFogfv`, `glFog`, and `glHint` as shown in figure 23. Here we refer readers to figure 24 for a comparison.



$$V' = V \times (1 - n\alpha)$$

s.t.:

$$n \propto d$$

```
// Enable fog effect
glEnable(GL_FOG);
GLfloat fogColor[] = { 0.8f, 0.8f, 0.8f };
glFogfv(GL_FOG_COLOR, fogColor);
glFogi(GL_FOG_MODE, GL_LINEAR);
glFogf(GL_FOG_DENSITY, 0.5f);
glFogf(GL_FOG_START, 0.0f);
glFogf(GL_FOG_END, 800.0f);
glHint(GL_FOG_HINT, GL_NICEST);
```

Figure 23: Intuitive illustration of GL fog model (left) and its implementation (right).

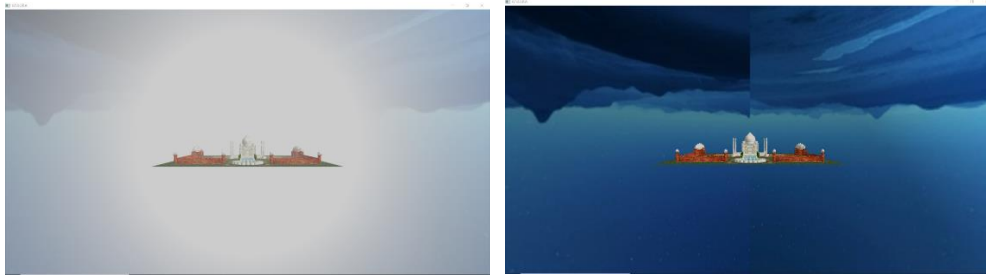


Figure 24: A comparison between with fog (left) and without fog (right)

● Switch between day and night

Another interesting feature in this project would be the switch between the day mode and the night mode. One can use “N” on the keyboard to switch back-and-forth between the two modes. In the day mode, the background is a day scenario with stronger light intensity in the scene, whereas in the night mode, the background changes into a night scenario with a weaker light intensity. This is achieved by binding keyboard event to the SkySphere class and the Scene class, which enforce them to switch into the other mode every time the certain key is pressed. Specific implementation of the event handling process could be found in figure 25 with a comparison between the scene in day mode and night mode in figure 26.

```
glEnable(GL_LIGHT0);
if (state % 2 == 1) {
    glEnable(GL_LIGHT1);
}
else {
    glDisable(GL_LIGHT1);
}

// the texture varies along with the state
if (state % 2 == 0) {
    glBindTexture(GL_TEXTURE_2D, privateID1);
}
else {
    glBindTexture(GL_TEXTURE_2D, privateID2);
}
```

Figure 25: Sample code of event handling process

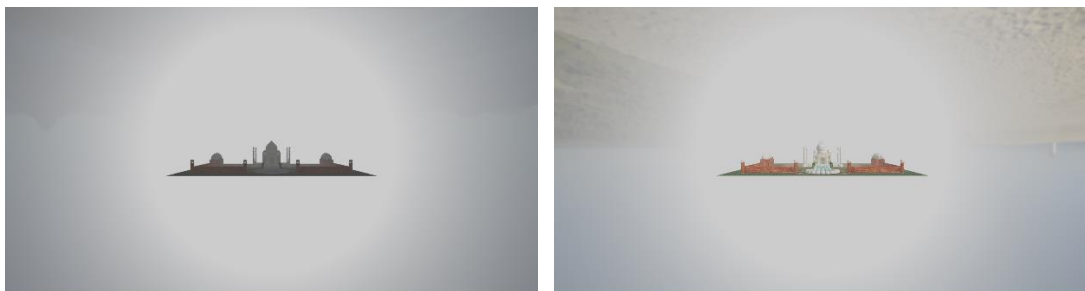


Figure 26: A comparison between night mode (left) and day mode (right)

5. Summary and conclusion

This project replicated the overall landscape of the Taj Mahal. To meet the requirement of the coursework, various efforts were devoted. From the perspective of the scene, delicate models assembled from basic structures in OpenGL are decorated with various textures. Multiple lighting and camera configurations were set up for a more

informative demonstration of the landscape. Animations of spraying water and water surface simulated fountains in the garden. Additionally, the use of RGBA texturing, the fog system, and the night mode could be concluded as creative ideas of this project. From the perspective of the code, stratified object structure with object orienting fashion gives the code a concise and straightforward structure. Header and in-text comments contributes to the readability of the code and are conducive for its further development and maintenance.