COMPUTER GRAPHICS COURSEWORK 3

Student Name: Yixin Qi
Student ID: 16522057
Date of Submission:
Word Count:

1 Introduction

In this project, a virtual scene of a coffee house (central perk) is displayed in a skybox. The scene is composed of nine main objects (sofa, chair, table, tea table, lamp, cup, teapot, and people), three positional lights and one animated directional light. The users can use keyboard to control the movements of people, door, and teapot as well as the position of camera. A view of this scene is demonstrated in *Figure 1.*

Figure 1: view of the virtual scene

The detailed implementations of the scene will be elaborated in the following sections with respect to six aspects: modelling, lighting, texture, animation and interaction.

2 Modelling

The six main objects in the scene are all constructed in different classes. The objects of the these classes will be added to the scene in the *Initialise()* function of *MyScene.cpp* using function *AddObjectToScene().* In this section, the implementations of the six objects will be demonstrated.

2.1 Sofa

In this scene, there are two sofas: the orange one (*Figure 2*) and the green one (*Figure 3*).

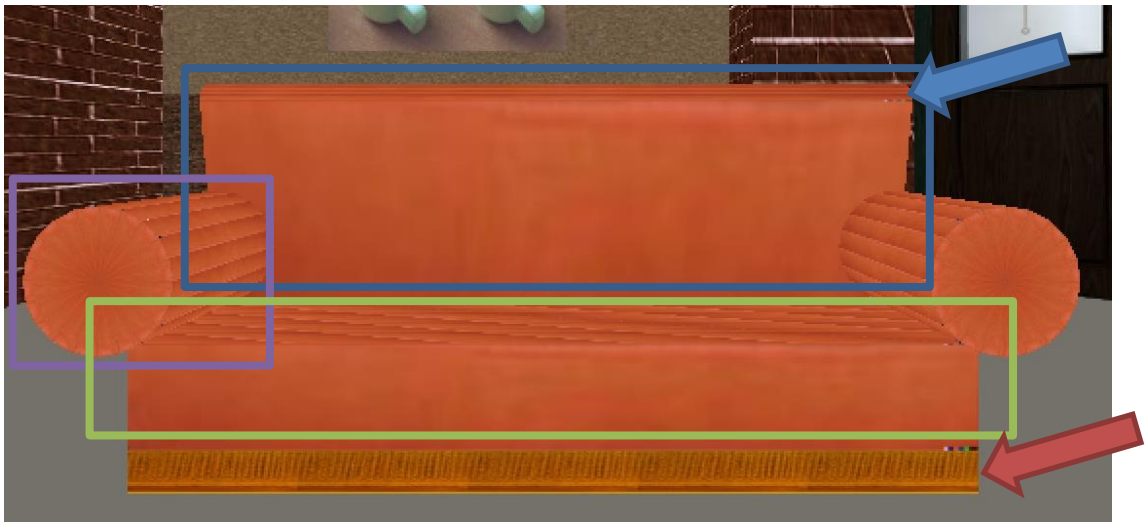

Figure 2: the orange sofa

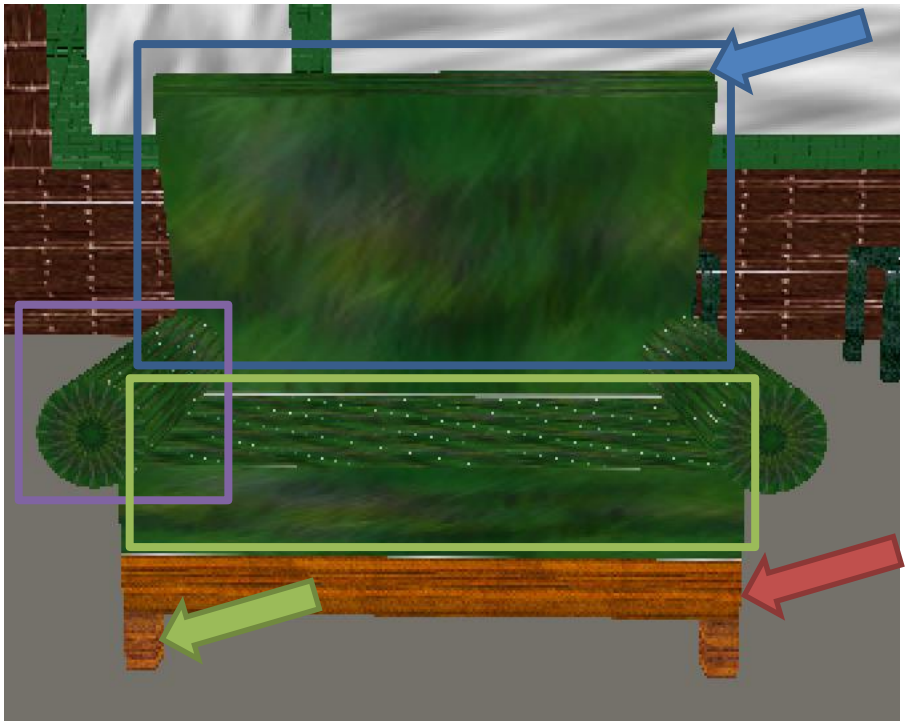

Figure 3: the green sofa.

These two sofas are generated from the same "Sofa" class since they are very similar in shapes and structures. As it can be seen from the figures, the main components of these two sofas are the ***bottom*** (highlighted in red arrows in *Figure 2* and *Figure 3*), cushion(highlighted in green squares *Figure 2* and *Figure 3*), ***handrails***(highlighted in purple squares in *Figure 2* and *Figure 3*),***back*** (highlighted in blue squares in *Figure 2* and *Figure 3*), and ***the half cylinder (the back-top)***(highlighted by blue arrows in *Figure 2* and *Figure 3*) that is placed on top of the back. Additionally, comparing to the orange sofa, the green sofa has four extra ***feet*** (highlighted by green arrow in *Figure 3*). These components are implemented in different functions, then they are combined to construct the sofa in the *drawSofa()* function. A fragment of this function for combining ***bottom, cushion and handrails*** is demonstrated in *Figure 4*. As it can be seen from the code*, glPushMatrix()* and *glPopMatrix()* functions are called every time a component is combined so that each component can be in their own transformation matrix. When one of the components is changed/moved, these individual transformation matrixes enable the rest of the components of sofa remain unchanged.

```
void Sofa::drawSofa() {

    glDisable(GL_CULL_FACE);
    if (toTexture) glEnable(GL_TEXTURE_2D);

    //bottom
    glPushMatrix();
    glTranslatef( 0.f,0.f+foot_height,0.f);
    draw_bottom();
    glPopMatrix();

    //cushion
    glPushMatrix();
    glTranslatef(0.f, bottom_height+foot_height, 0.f);
    draw_cushion();
    glPopMatrix();

    //handle_right
    glPushMatrix();
    glTranslatef(length / 2.f, (bottom_height + cushion_height+foot_height) + (length / 12.f), -width / 2.f+back_width);
    glRotatef(90.f, 1.f, 0.f, 0.f);
    draw_handle();
    glPopMatrix();

    //handle_left
    glPushMatrix();
    glTranslatef(-length / 2.f, (bottom_height+cushion_height+foot_height)+ (length / 12.f), -width/2.f+back_width);
    glRotatef(90.f, 1.f, 0.f, 0.f);
    draw_handle();
    glPopMatrix();
```

Figure 4: A fragment of the *drawSofa()* function.

Construction of *bottom, cushion, feet* and *back*: Each side of the sofas' bottom, cushion, feet and back components is a closed quadrilateral implemented using shape: **GL_QUADS.** The widths and lengths of these quadrilateral are defined in *MyScene.cpp* using setter functions while adding the sofas into the scene. Since the orange sofa does not need the ***feet***, the height of the ***feet*** for orange sofa is set to 0.

For the ***cushion, bottom, feet*** parts, the sides are implemented as rectangles with different lengths and widths. And for the ***back*** component, the front side (the side that is shown on *Figure 3* and *Figure 4*) and back side are implemented as upturned trapezoids while the left and right sides are also constructed as two rectangles. These four parts are implemented in function *draw_bottom()*, *draw_cushion(), draw_feet()* and *draw_back()* respectively. A code snippet for constructing the front side of the ***cushion*** is shown in *Figure 5.*

```
void Sofa::draw_cushion () {
    //cushion front
    glNormal3f(0.f, 0.f, 1.f);
    if (toTexture) glBindTexture(GL_TEXTURE_2D, texids[1]);
    glBegin(GL_QUADS);
    if (toTexture) glTexCoord2f(1.f, 1.f);
    glVertex3f(length / 2.f, 0, width / 2.f);
    if (toTexture) glTexCoord2f(1.f, 0.f);
    glVertex3f(length / 2.f, cushion_height, width / 2.f);
    if (toTexture) glTexCoord2f(0.f, 0.f);
    glVertex3f(-length / 2.f, cushion_height, width / 2.f);
    if (toTexture) glTexCoord2f(0.f, 1.f);
    glVertex3f(-length / 2.f, 0, width / 2.f);
    glEnd();
```

Figure 5: code snippet for constructing the front side of the sofa cushion.

Construction of *handrails* and *back-top*: The ***handrails*** of the sofa are implemented as closed-ends cylinders (*Figure 6*).

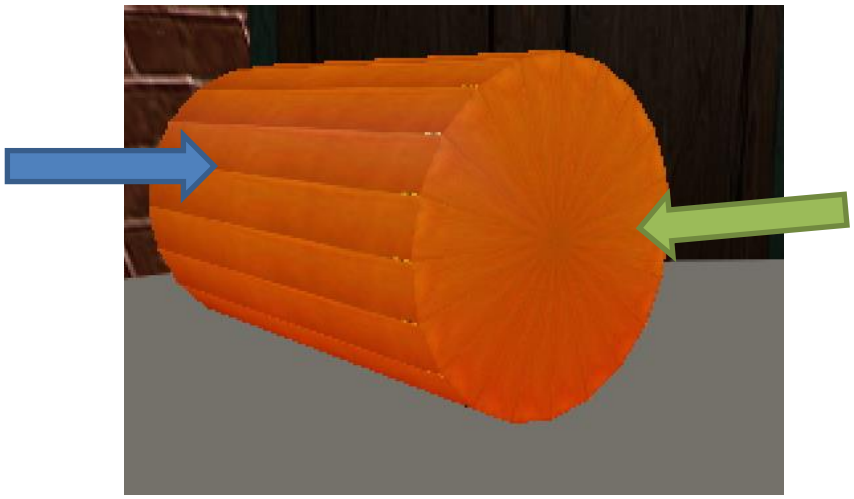

Figure 6: the right ***handrail*** of orange sofa

The body of the cylinders (highlighted by blue arrow) are constructed by 20 rectangles using **GL_QUADS**. The code for implementing the cylinder body is demonstrated in *Figure 7.*

```
void Sofa::draw_handle() {
    float r = length / 12.f;
    float res = 0.1*M_PI;           // resolution (in radians: equivalent to 18 degrees)
    float x = r, z = 0.f;           // initialise x and z on right of cylinder centre
    float t = 0.f;                  // initialise angle as 0
    float h = width;
    if (toTexture) glBindTexture(GL_TEXTURE_2D, texids[1]);
    do
    {

        glBegin(GL_QUADS);          // new QUAD
            // Create first points
        glNormal3f(x, 0.f, z);
        if (toTexture) glTexCoord2f(1.f, 1.f);
        glVertex3f(x, h, z);    // top
        if (toTexture) glTexCoord2f(0.f, 1.f);
        glVertex3f(x, 0.f, z);  // bottom
        // Iterate around circle
        t += res;               // add increment to angle
        x = r * cos(t);           // move x and z around circle
        z = r * sin(t);
        // Close quad
        glNormal3f(x, 0.f, z);
        if (toTexture) glTexCoord2f(0.f, 0.f);
        glVertex3f(x, 0.f, z);  // bottom
        if (toTexture) glTexCoord2f(1.f, 0.f);
        glVertex3f(x, h, z);    // top
        glEnd();                    // end shape
    } while (t <= 2 * M_PI);        // for a full circle (360 degrees)
```

Figure 7: The implementation of handrail's body.

As for the two closed ends of the ***handrails*** (highlighted by green arrow in *Figure 6*), they are composed by 20 triangles using the shape **GL_TRIANGLE_FAN**. The code for implementing the two ends are demonstrated in *Figure 8*.

```
//back
glBegin(GL_TRIANGLE_FAN);
if (toTexture) glTexCoord2f(0.5f, 1.f);
glVertex3f(0.f, 0.f, 0.f);
do
{
    // Create first points
   glNormal3f(0.f, 1.f, 0.f);
   glNormal3f(x, 0.f, z);
   if (toTexture) glTexCoord2f(0.f, 0.f);
   glVertex3f(x, 0.f, z);
   // Iterate around circle
   t += res;               // add increment to angle
   x = r * cos(t);           // move x and z around circle
   z = r * sin(t);
   // Close quad
   glNormal3f(x, 0.f, z);
   if (toTexture) glTexCoord2f(1.f, 0.f);
   glVertex3f(x, 0.f, z);  // bottom

                        // end shape
} while (t <= 2*M_PI);        // for a full circle (360 degrees)
glEnd();

//front
t = 0.f;
x = r, z = 0.f;           // initialise x and z on right of cylinder centre

glBegin(GL_TRIANGLE_FAN);
if (toTexture) glTexCoord2f(0.5f, 1.f);
glVertex3f(0.f, h, 0.f);
do
{
   glNormal3f(0.f, -1.f, 0.f);
   glNormal3f(x, 0.f, z);
   if (toTexture) glTexCoord2f(0.f, 0.f);
   glVertex3f(x, h, z);
   // Iterate around circle
   t += res;               // add increment to angle
   x = r * cos(t);           // move x and z around circle
   z = r * sin(t);
   // Close quad
   glNormal3f(x, 0.f, z);
   if (toTexture) glTexCoord2f(1.f, 0.f);
   glVertex3f(x, h, z);

                        // end shape
} while (t <= 2*M_PI);        // for a full circle (360 degrees)
glEnd();
```

Figure 8: the code for implementing the two ends of ***handrails.***

The ***back-top*** of the sofa is the half-cylinder that is placed on the top of the sofa ***back***. And example of this is shown in *Figure 9.* The implementation of the half cylinder is the same as the ***handrail.*** The body of the cylinder is constructed using rectangles while the two closed-ends are implemented by triangles. Since this is only a half-cylinder, it only uses 10 rectangles and 10 triangles.

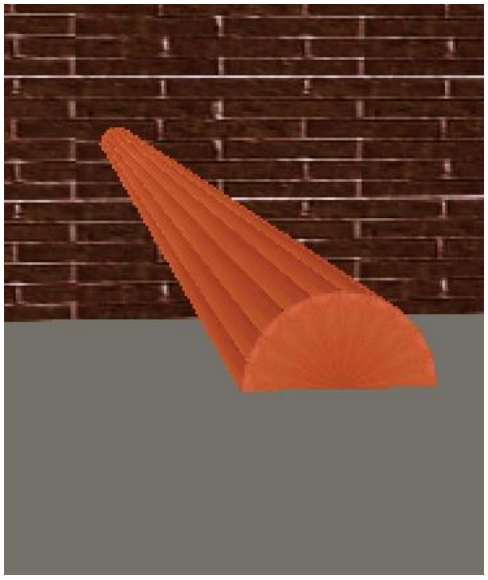

Figure 9: the ***back-top*** of the orange sofa.

2.2 Table, Cup, Teapot and Chair

The table, cup, teapot and chair that are implemented in the scene are demonstrated in *Figure 10* and *Figure 11*.

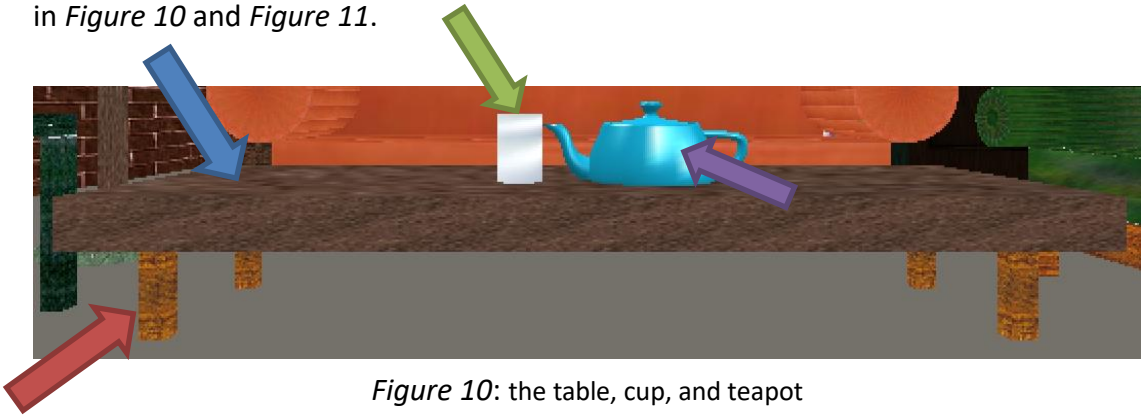

Figure 10: the table, cup, and teapot

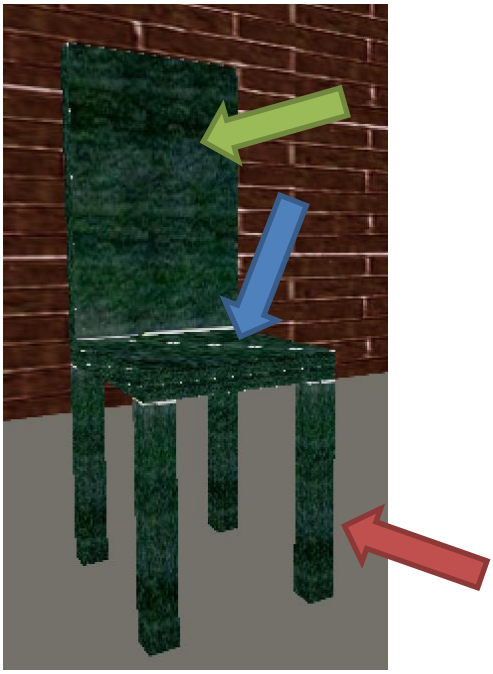

Figure 11: the chair

Table and Chair: The table and chair are defined in classes "Table" and "Chair" respectively. As shown in the figures, these table and chair both have one ***plane*** (highlighted by blue arrow in *Figure 10* and *Figure 11*) and four ***feet*** (highlighted by red arrow *Figure 10* and *Figure 11*). And the chair has an additional ***back*** comparing to the table (highlighted by green arrow in *Figure 11*). Same as the "Sofa" class, in these two classes, the ***feet, back*** and ***plane*** components are implemented in different functions and combined in *draw_Table()* and *drawChair()* functions to construct these two objects. The *drawChair()* function in "Chair" class that combines the ***feet, back*** and ***plane*** to construct the chair is demonstrated in *Figure 12*. Same as the *drawSofa()* function, these components are all assigned with a individual transformation matrix to enable the chair's and table's adaptivity for partial changes.

```
void Chair::drawChair() {

    glDisable(GL_CULL_FACE);
    if (toTexture) glEnable(GL_TEXTURE_2D);
    //left_fornt_foot
    glPushMatrix();
    glTranslatef(-length / 2.f, 0.f, width / 2.f - foot_width);
    draw_feet();
    glPopMatrix();

    //right_front_foot
    glPushMatrix();
    glTranslatef((length / 2.f - foot_length), 0.f, (width / 2.f - foot_width));
    draw_feet();
    glPopMatrix();

    //left_back_foot
    glPushMatrix();
    glTranslatef(-length / 2.f, 0.f, -width / 2.f);
    draw_feet();
    glPopMatrix();

    //right_back_foot
    glPushMatrix();
    glTranslatef((length / 2.f - foot_length), 0.f, -width / 2.f);
    draw_feet();
    glPopMatrix();
    //
    //bottom
    glPushMatrix();
    glTranslatef(0.f, foot_height, 0.f);
    draw_buttom();
    glPopMatrix();



    //back
    glPushMatrix();
    glTranslatef(-length / 2.f, foot_height  + bottom_height, -width / 2.f);
    draw_back();
    glPopMatrix();







    if (toTexture) {
        //glBindTexture(GL_TEXTURE_2D, 0);
        glDisable(GL_TEXTURE_2D);
    }
    glEnable(GL_CULL_FACE);
}
```

Figure 12: the code for constructing the chair using ***feet, plane and back***

As shown in graph *Figure 10* and *Figure 11*, each side of these ***plane, feet*** and ***back*** is implemented as a rectangles using **GL_QUADS**. A snippet of code for constructing the top side of table (the side pointed by the blue arrow in *Figure 10)* is shown in *Figure 13*.

```
void Table::draw_plane() {


    //plane_top
    glNormal3f(0, 1, 0);
    glBegin(GL_QUADS);
    if (toTexture) glTexCoord2f(0.f, 0.f);
    glVertex3f(table_length / 2.f, table_height, -table_width / 2.f);
    if (toTexture) glTexCoord2f(0.f, 1.f);
    glVertex3f(-table_length / 2.f, table_height, -table_width / 2.f);
    if (toTexture) glTexCoord2f(1.f, 0.f);
    glVertex3f(-table_length / 2.f, table_height, table_width / 2.f);
    if (toTexture) glTexCoord2f(1.f, 1.f);
    glVertex3f(table_length / 2.f, table_height, table_width / 2.f);
    glEnd();
```

Figure 13: a fragment of the *draw_plane()* function for constructing the top side of the table plane.

Cup and Teapot: The cup and teapot are highlighted in *Figure 10* by green and purple arrows. As shown in the figure, the cup is implemented a cylinder. But it should be noticed that to facilitate the animation of the cup that will be elaborated in the Animation Section (Section 5), the cup should be composed by ***two piled up cylinders***. A closer look of the cup is shown in *Figure 14.* The approximate poistions of these two cylinders that compose this cup are highlighted by blue and green squares in *Figure 14*.

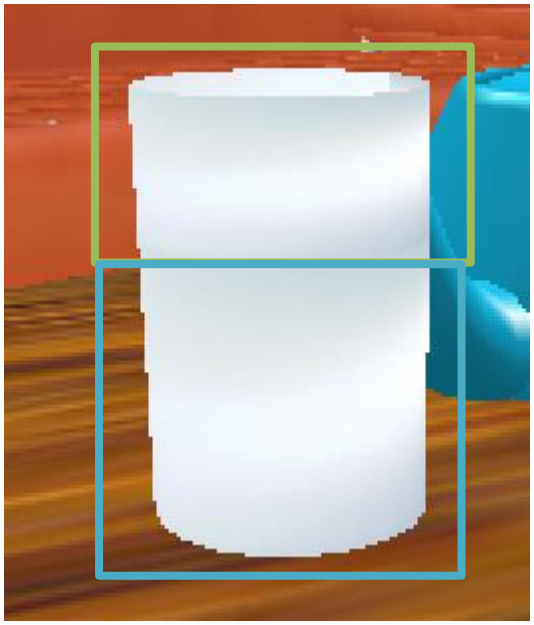

Figure 14: a closer look of the cup.

Same as the implementation for sofa's ***handrails,*** each of the cylinder of the cup is constructed by 20 rectangles using **GL_QUADS**. Then, for the closed bottom side of

the cup (the side that contacts the table), it is composed by 20 triangles using **GL_TRIANGLE_FAN.** As for the teapot object, it is implemented using the *glutSolidTeapot()* function.

It should be noticed that, since the cup and teapot are placed on the table, for convenience, the code for constructing the two objects are also put in the "Table" class. They are also assigned with individual transformation matrixes using *glPushMatrix()* and *glPopMatrix()* functions and combined with the rest components of the table in the *draw_Table()* function.

2.3 Lamp and Tea Table

The lamp and tea table in the scene are exhibited in *Figure 15* and *Figure 16,17.*

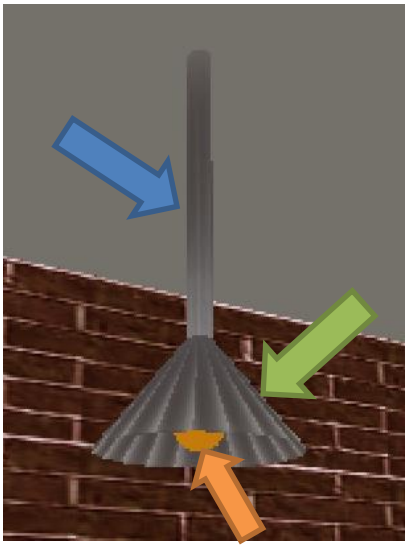

Figure 15: the lamp

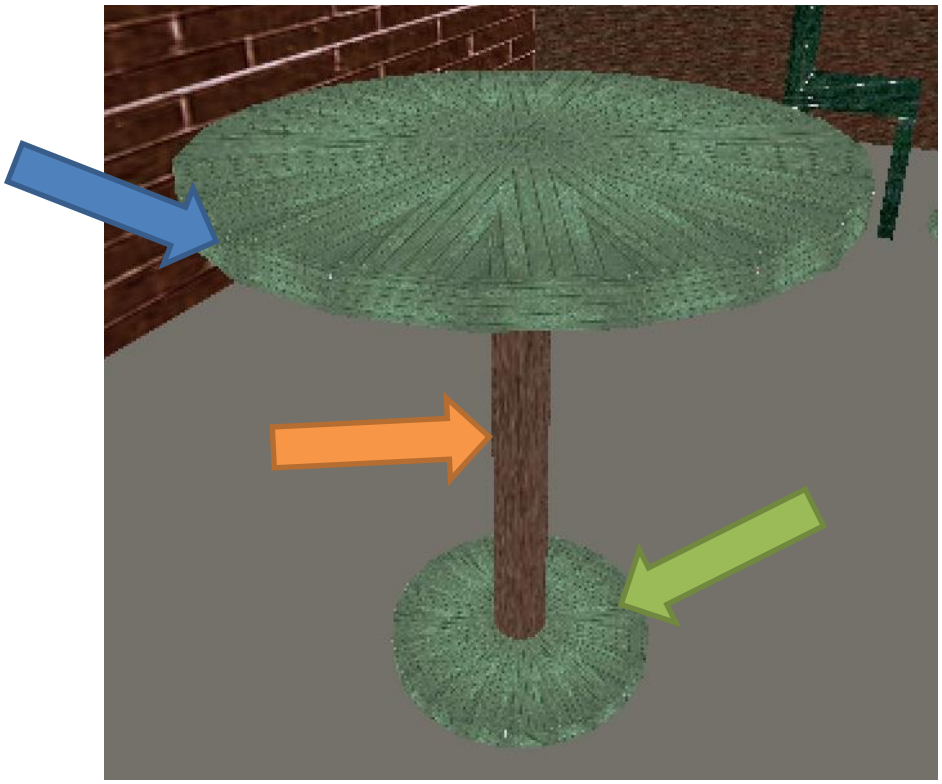

Figure 16: the tea table

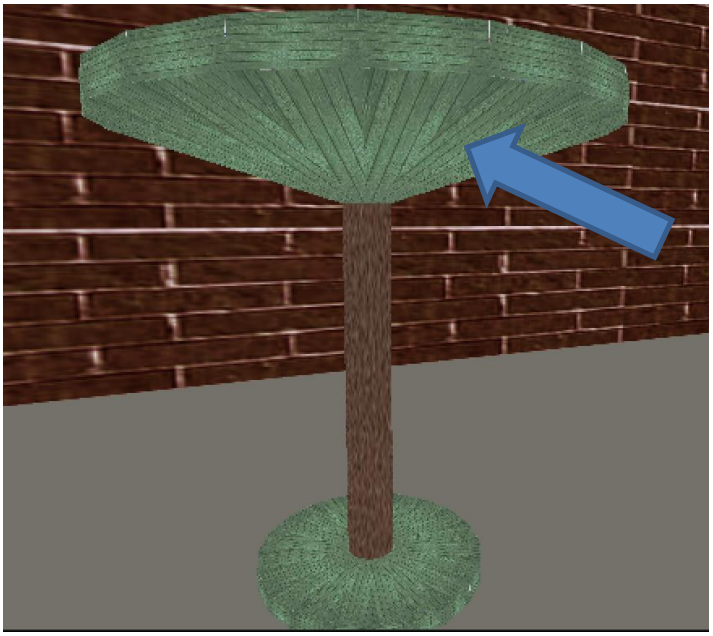

Figure 17: the tea table

Lamp: As it can be seen from the *Figure 15*, the lamp is composed by three parts: ***tube*** (highlighted by blue arrow in *Figure 15*)***, cover***(highlighted by green arrow in *Figure 15*) and ***light bulb*** (highlighted by orange arrow in *Figure 15*)**.**

As shown in the figure, the ***tube*** is implemented as a cylinder and it is constructed the same way as the sofa's ***handrails.*** It is composed by 20 rectangles that are drawn by the shape **GL_QUADS**.

As for the ***cover*** part, its implementation is a variation of the implementation of sofa's ***handrails' two closed-ends***. It is also constructed by 20 triangles that are implemented using **GL_TRIANGLE_FAN**. But since it is not a flat circle like the closed-ends, the middle point of the circle is higher the circle edge. In other words, for each triangle, the vertexes that are at the center of the circle are higher than the rest two.

The ***light bulb*** is implemented using the function *glutSolidSphere().* Since the color of the ***light bulb*** will be changed accordingly with the positional light that will be put at the lamp's position, the code for implementing the light bulb is placed in the "Lights" class (the details about the light and the color change of light bulb will be elaborated in Section 3).

Tea Table: As shown in *Figure 16* and *Figure 17*, the tea table is composed by ***top*** (pointed by blue arrow in *Figure 16*)***, bottom*** (pointed by green arrow in *Figure 16*)***, tube*** (pointed by orange arrow in *Figure 16*)***, and holder*** (pointed by blue arrow in *Figure 17*)***.***

As it can be seen, the ***top***, ***tube*** and ***bottom*** parts are all implemented as cylinders and the ***holder*** part is an upturned circular cone. For the ***top, bottom*** and ***tube***, the cylinders can again be implemented using 20 rectangles. Since the ***top*** and ***bottom*** are both closed-ends cylinders, the two ends of the cylinders are closed by circles that are generated using 20 triangles. As for the ***holder*** part, the implementation is the same as the implementation of lamp's ***cover*** part which is also constructed by 20 triangles.

Same as other objects in the scene, the lamp and tea table are also defined in their own classes: "Lamp" class and "Table2" class. Their components are also defined in different functions and combined in *drawLamp()* and *drawTable()* functions. Again, in these two functions, each component is assigned with a unique transformation matrix to facilitate further changes that may be applied to these two objects.

2.4 People

The people object implemented in the scene has been exhibited in *Figure 18*.

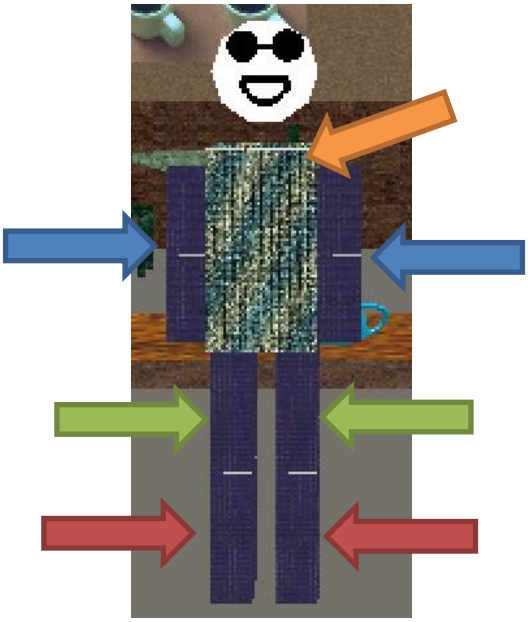

Figure 18: the people in the scene.

As shown in the figure, the people are composed by ***left arm*** (highlighted by blue arrow in *Figure 18*), ***right arm***(highlighted by blue arrow in *Figure 18*), ***upper left leg***(highlighted by green arrow in *Figure 18*), ***upper right leg***(highlighted by green arrow in *Figure 18*), ***lower left leg*** (highlighted by red arrow in *Figure 18*) , ***lower right leg*** (highlighted by red arrow in *Figure 18*), ***body*** (highlighted by orange arrow in *Figure 18)* and ***head***.

This object is defined in class "People". The body, arms and legs are all composed by cubes with each side being implemented as a rectangle drawn by **GL_QUADS**. These cubes are drawn in the function *drawCube().* While combining each part of the people in *drawPeople()* function, all the components of the people are included in the same transformation matrix as the ***body*** part. Then in this transformation matrix, the upper body (i.e. ***the arms***) are implemented in an individual sub-matrix. And inside this sub-matrix of upper body, the ***left and right arms*** are again implemented in different sub-matrixes. Then for the lower body part (***the legs***), it is also implemented in an individual sub-matrix. Again, inside this lower body sub-matrix, the left and right legs are implemented in different sub-matrixes. Further, inside these two matrixes for left and right legs, the ***lower left leg*** and ***lower right leg*** are again placed into sub-matrixes. The ***head*** part is implemented in the same matrix as the ***body.*** These components are arranged in this way to facilitate the further animation implementation of the people (details in Section 5).

2.5 Skybox

The whole scene is established in a skybox. The overlook of the skybox from above is displayed in *Figure 18*. As shown in the figure, the skybox constructs the walls, floor and ceiling of this scene. Each side of the skybox is constructed by rectangles that are drawn by **GL_QUADS**. This object is defined in class “Skybox”.

Figure 18: the overview of the skybox from above.

3 Lighting

There are three positional lights, one directional light in the scene. They are all defined in class “Lights”. The detailed implementation of these four lights will be elaborated in this section. It should be noticed that, all the objects in the scene are affected by the lighting except for the skybox (i.e. the floors and walls).

3.1 Directional Light

The scene under only directional light is displayed in *Figure 19*. This is an ***animated directional light,*** and an extra sphere is placed on the rooftop to indicate the direction of the light (highlighted by blue arrow in *Figure 19*). The details about the animation will be clarified in Section 5. The ***ambient, diffuse*** and ***specular*** parameters of the light are set to "*0.5, 0.5, 0.5, 1*", "*0.8, 0.8, 0.8, 1*", "1.0*, 1.0, 1.0, 1*" respectively.

Figure 19: the scene under directional light

3.2 Positional Lights

There are three positional lights in this scene. The scene under positional light 1, positional light 2 and positional light 3 are exhibited in *Figure 20, 21, 22*. The positions of the lights are clarified by three spheres pointed by green arrows in these three figures. The color of the sphere will be orange when the corresponding positional lights is on, otherwise, it is black. The scene under all three positional lights is displayed in *Figure 23*. The ***ambient***, ***diffuse*** and ***specular*** parameters of these three lights are set to "*0.2, 0.2, 0.2, 1*", "*0.5, 0.5, 0.5, 1*", "*1.0, 1.0, 1.0, 1*". These three positional lights share the same sets of positional light parameters and the ones for positional light 1 are exhibited in *Figure 24*.

Figure 20: the scene under positional light 1.

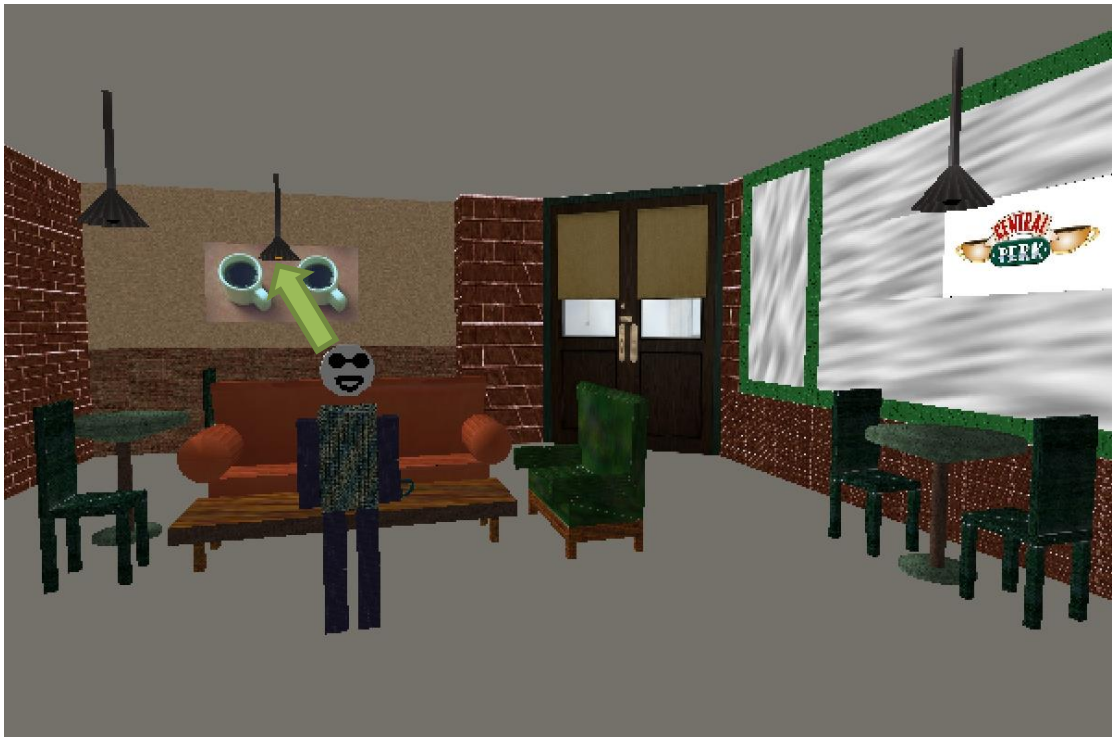

Figure 21: the scene under positional light 2.

Figure 22: the scene under positional light 3.

Figure 23: the scene under three positional lights.

```
//set positional light parameters
float direction[] = { 0.f,-1.f,0.f };
glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, direction);
glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 120.f);
glLightf(GL_LIGHT1, GL_LINEAR_ATTENUATION, 0.000002f);
```

Figure 24: the positional light parameters for positional light 1.

4 Texture

In this scene all the surfaces of all objects and skybox are covered by textures except for the people's head, teapot, floor, ceiling and the spheres that indicate the lights. In this section, detailed implementation of the textures will be introduced.

4.1 Texture Mapping for Skybox

As mentioned above, the skybox constructs the walls, floor and ceiling of this scene. The floor and ceiling are not covered by any texture, but the multiple textures are applied to the walls.

The following textures (*Figure 25, and Figure 26*) are applied to the skybox's walls. For the six textures in *Figure 25*, they are repeatedly mapped to corresponding surfaces by setting the texture coordinates to number that is larger than 1. The snippet of code for mapping texture on the left wall (highlighted by green arrow in *Figure 27*) is demonstrated in *Figure 28*. The highlighted lines of code indicate that this wall is filled with 25 copies of this texture (5 for the wall's x-axis direction and 5 for the wall's y-axis direction.).

Figure 25: the textures that have been mapped to the skybox's walls

Figure 26: the textures that have been mapped to the skybox's walls.

Figure 27: the overview of the skybox.

```
// LEFT SIDE
//glColor3f(0.9f, 0.9f, 1.0f); // blue
if (toTexture) glBindTexture(GL_TEXTURE_2D, texids[0]);
glBegin(GL_QUADS);
if (toTexture) glTexCoord2f(5.f, 5.f);
glVertex3f(-1.f, 1.f, -1.f);
if (toTexture) glTexCoord2f(0.f, 5.f);
glVertex3f(-1.f, 1.f, 1.f);
if (toTexture) glTexCoord2f(0.f, 0.f);
glVertex3f(-1.f, 0.f, 1.f);
if (toTexture) glTexCoord2f(5.f, 0.f);
glVertex3f(-1.f, 0.f, -1.f);
glEnd();
glColor3f(1.f, 1.f, 1.f);
```

Figure 28: the snippet of code for mapping the textures on the left wall of Skybox.

As for the textures shown in *Figure 26*, they are mapped to rectangles without being copied. The coordinates of the rectangle's vertices correspond to the texture coordinates: (0,0), (1,0), (1,1), (0,1). A snippet of code for mapping the texture on the left door (highlighted by blue arrow in *Figure 27*) is shown in *Figure 29*. In this figure, the texture coordinates are highlighted by green arrows.

```
//left door
if (toTexture) glBindTexture(GL_TEXTURE_2D, texids[5]);
glBegin(GL_QUADS);

if (toTexture) glTexCoord2f(0.f, 1.f);
glVertex3f(0.f, 1.f, 0.f);
if (toTexture) glTexCoord2f(0.f, 0.f);
glVertex3f(0.f, 0.f, 0.f);
if (toTexture) glTexCoord2f(1.f, 0.f);
glVertex3f(left_door_length, 0.f, 0.f);
if (toTexture) glTexCoord2f(1.f, 1.f);
glVertex3f(left_door_length, 1.f, 0.f);
glEnd();
glColor3f(1.f, 1.f, 1.f);
glPopMatrix();
```

Figure 29: the code for mapping the texture on the left door.

4.2 Texture Mapping for Objects

The textures that are applied to the objects' surfaces are displayed below.

Figure 30: the textures that are mapped to the two sofas.

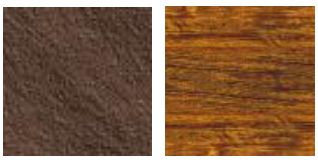

Figure 31: the textures that are mapped to the table.

Figure 32: the textures that are mapped to the tea table.

Figure 33: the texture that is mapped to the chair.

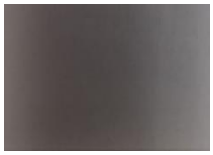

Figure 34: the texture that is mapped to the lamp.

Figure 35: the textures that are mapped to the people.

Figure 36: the texture that is mapped to the cup.

As introduced above, the surfaces of the objects are all composed by rectangles or triangles. For the rectangles except for the ones that construct the cup, the texture coordinate numbers for each vertex are all greater or equal to 1. And for the three vertices of triangles, they correspond to the texture coordinates: (0.5, 1), (0.0, 0.0), (1.0, 0.0).

As for the cup, to enable the smooth surface effect, the texture that is applied to the cup is divided into 20 parts along the x-axis. Then each of the part is applied to a rectangle that compose the cup. The code snippet for mapping the texture on the top cylinder of cup (as introduced in Section 2, the cup is composed by two cylinders) is elaborated in *Figure 37*. The lines of code for mapping texture are highlighted by blue arrows.

```
void Table::draw_cup() {

    //top
    float r = cup_rad;
    float res = 0.1*M_PI;           // resolution (in radians: equivalent to 18 degrees)
    float x = r, z = 0.f;           // initialise x and z on right of cylinder centre
    float t = 0.f;                  // initialise angle as 0
    float h = cup_height_top;
    glPushMatrix();
    glTranslatef(0.f,cup_height_buttom,0.f);
    if (toTexture) glBindTexture(GL_TEXTURE_2D, texids[2]);
    do
    {
        glBegin(GL_QUADS);          // new QUAD
            // Create first points

        glNormal3f(x, 0.f, z);
        if (toTexture) glTexCoord2f(t/(2*M_PI), 1.f);
        glVertex3f(x, h, z);
        if (toTexture) glTexCoord2f(t / (2 * M_PI), 0);
        glVertex3f(x, 0.f, z);
        // Iterate around circle
        t += res;               // add increment to angle
        x = r * cos(t);           // move x and z around circle
        z = r * sin(t);
        // Close quad
        glNormal3f(x, 0.f, z);
        if (toTexture) glTexCoord2f(t / (2 * M_PI), 0.f);
        glVertex3f(x, 0.f, z);  // bottom
        if (toTexture) glTexCoord2f(t / (2 * M_PI), 1);
        glVertex3f(x, h, z);    // top
        glEnd();                    // end shape
    } while (t <= 2 * M_PI);        // for a full circle (360 degrees)
    glPopMatrix();
```

Figure 37: the code for mapping texture on the top cylinder of cup.

5 Animation

In this scene, there are 5 animated objects: directional light, tea table, doors, teapot and people. All of them are sub-classes of "Animation".

5.1 Animated Directional Light

As mentioned above, the direction of directional light is animated and indicated by the sphere on the roof. To enable this animation, the position parameters of this light is updated in the *Update()* function in the "Light" class. The code for updating and enabling the light in *Update()* function is demonstrated in *Figure 38*.

```
void Lights::Update(const double& deltaTime) {

    //capture the time elapsed
    _runtime += deltaTime;

    //update the position of Light0/1/2
    _position[_LIGHT_0+0]= static_cast<GLfloat>(_radius0*cos(_runtime));
    _position[_LIGHT_0 + 2] = static_cast<GLfloat>(_radius0*sin(_runtime));


    //set the Light0
    glLightfv(GL_LIGHT0, GL_AMBIENT, &_ambient[_LIGHT_0]);      // set ambient parameter of light source
    glLightfv(GL_LIGHT0, GL_DIFFUSE, &_diffuse[_LIGHT_0]);      // set diffuse parameter of light source
    glLightfv(GL_LIGHT0, GL_SPECULAR, &_specular[_LIGHT_0]);    // set specular parameter of light source
    glLightfv(GL_LIGHT0, GL_POSITION, &_position[_LIGHT_0]);    // set direction vector of light source
    // Enable this lighting effects
    glEnable(GL_LIGHTING);  // enable scene lighting (required to enable a light source)
    light0 ? glEnable(GL_LIGHT0) : glDisable(GL_LIGHT0);
```

Figure 38: the code for updating the position and enabling the directional light.

5.2 Animated Door

In this scene, the audience (i.e. users) can open and close the left and right doors (highlighted by blue arrow in *Figure 38*) by pressing the key "l" and "r". The *Figure 39* shows the scene of opened left and right doors.

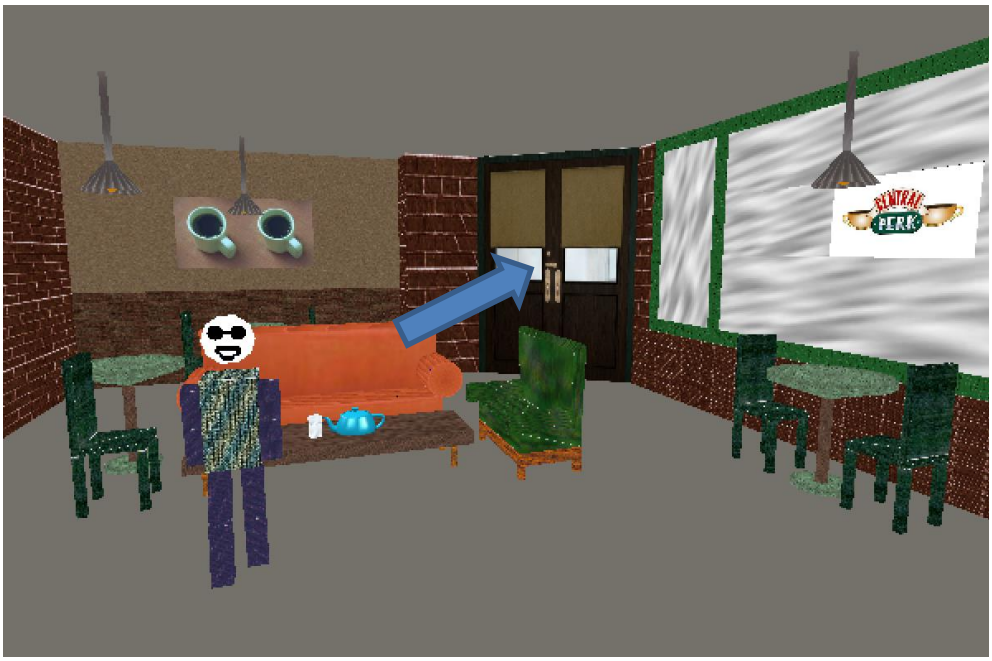

Figure 38: the scene that contains the doors

Figure 39: the opened left and right doors.

The animation of the doors is realized by rotations. In the *Update()* function of class "Skybox" (the doors are implemented as parts of the skybox), the rotation angles of the doors are related to the elapsed time after users trigger the animation. After a certain time (i.e. 3.6 seconds), the rotation angle will stop increasing and the doors' opening animation will stop. The code for implementing this animation is displayed in

Figure 40.

```
void Skybox::Update(const double& deltaTime) {

    //update the left door
    if (left_door_close) {
        left_runtime = 0;
    }

    if (!left_door_close) {
        cout << left_runtime;
    cout << "\n";
    if (left_runtime <= 3.6) {
        left_runtime = left_runtime + deltaTime;
    }
    }

    //update the right door

    if (right_door_close) {
        right_runtime = 0;
    }

    if (!right_door_close) {

        if (right_runtime <= 3.6) {
            right_runtime = right_runtime + deltaTime;
        }
    }


    left_door_close ? offset_left = left_runtime = 0 : offset_left= left_runtime*20 ;
    right_door_close ? offset_right = right_runtime = 0 : offset_right = right_runtime*20;
}
```

Figure 40: the code for implementing the door animation.

5.3 Animated Tea Table

In this scene, the users can trigger the “jump and fall back” animation of tea table by pressing the key “j”. The screen shot of the static and jumped table is displayed in *Figure 41* and *Figure 42,* respectively.

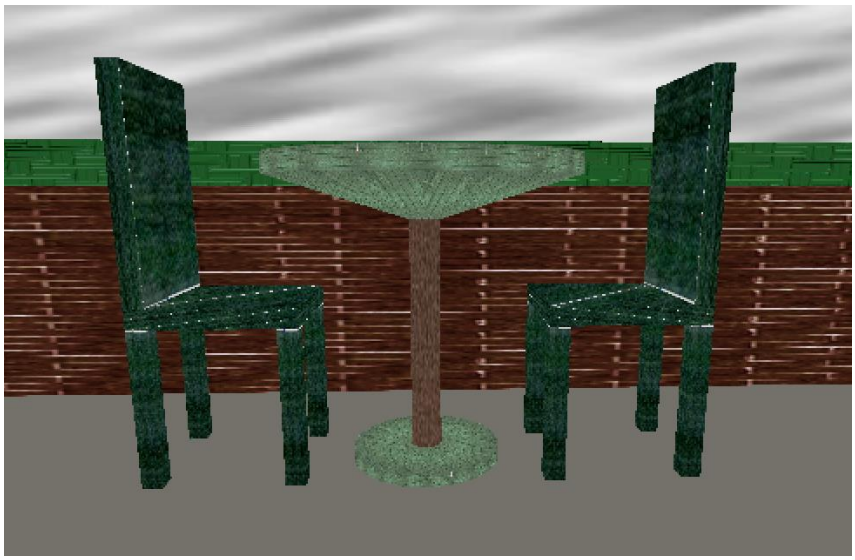

Figure 41: the static tea table.

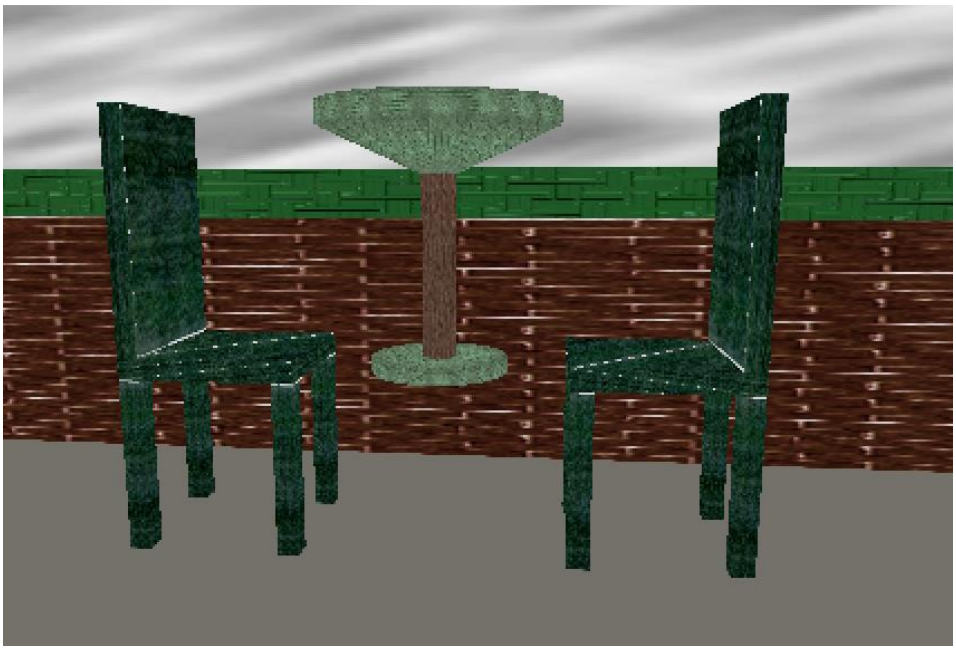

Figure 42: the jumped tea table.

The whole process happens in 1 second after users press the “j” key. The animation of jumping up and falling back down is divided into 13 equal frames (i.e. 13 equal time duration within 1 second). In the first 6 frames, the ***bottom, tube*** and ***top*** of the tea table are shrinked and the tea table will go up along the y-axis. Then the in the rest 7 frames, the three shrinked parts of the table will gradually increase their size until “recover” to the original size. And the tea table will also fall down along the y-axis to its original position. The code for implementing the first and last three frames (stages) of the animation is displayed in *Figure 43*.

```
void Table2::Update(const double& deltaTime) {

    //cout << deltaTime;

    if (jump) {
        _runtime = fmod(_runtime + deltaTime, animationTime);          // update time in animation cycle

        float stage = 13.f*_runtime / animationTime;          // calculate stage (out of 13)

        if (stage < 1 || stage>12)
        {

            coor_change = 0;
            tube_change = 0;
    //      slide_change = 0;
            top_rad_change = 0;
            if (stage > 12) {
                jump = false;
                _runtime = 0;
            }
        }
        else if (stage < 2 || stage>11)
        {

            coor_change = 20;
            tube_change = -10;
            buttom_change = -2;
            top_rad_change = -10;
        }
        else if (stage < 3 || stage>10)
        {

            coor_change = 40;
            tube_change = -20;
            buttom_change = -4;
            top_rad_change = -15;
        }
```

Figure 43: the code for implementing the first and last three frames of the tea table animation

5.4 Animated Teapot and Cup

The user can trigger the animation of teapot and cup by pressing the key “t”. In the animation, the teapot will first move up along the y-axis and then rotate along the z-axis to mimic the scene of pouring tea into the cup. After the teapot is rotated, the bottom half of the cup will turn green to mimic the poured tea. Then the teapot will rotate back along the z-axis and move down along the y-axis to go back to its original position and the green color of the cup will disappear. The static teapot and rotated teapot are demonstrated in *Figure 44* and *Figure 45.*

Figure 44: the static teapot and cup.

Figure 45: the rotated tea pot and filled cup

This whole animation lasts 3 seconds and is divided into 28 frames. For the first 7 frames, the teapot will move up along the y-axis. Then from frame 8 to 14, the teapot will rotate along the z-axis. And at the 14th frame, the color of the cup will change. After this, in frames 15 to 22, the teapot will rotate back along the z-axis and in frames 22 to 28, the teapot will move back to its original place. In the last frame, the color of the cup will be turned back. The code snippet for implementing the first and last 7 frames is displayed in *Figure 46*.

```
void Table::Update(const double& deltaTime) {

    if (tea) {
        _runtime = fmod(_runtime + deltaTime, animationTime);          // update time in animation cycle
        float stage = 28.f*_runtime / animationTime;          // calculate stage (out of 28)
        cout << stage;
        cout << "\n";
        if (stage < 1 || stage>27)
        {
            _v_teapot = 0.f;
            _r_teapot = 0.f;
            color = 2;

           if (stage > 12) {
               tea = false;
               _runtime = 0;
           }
        }
        else if (stage < 2 || stage>26)
        {
           _v_teapot = 10.f;
            _r_teapot = 0.f;

        }
        else if (stage < 3 || stage>25)
        {
            float _v_teapot = 20.f;
            float _r_teapot = 0.f;

        }
        else if (stage < 4 || stage>24)
        {
            _v_teapot = 30.f;
            _r_teapot = 0.f;

        }
        else if (stage < 5 || stage>23)
        {
            _v_teapot = 40.f;
            _r_teapot = 0.f;

        }
        else if (stage < 7 || stage>22)
        {
            _v_teapot = 50.f;
            _r_teapot = 0.f;

        }
```

Figure 46: the first and last frames for the teapot animation.

5.5 Animated People

In this scene, the users can use keys “f”, “c”, “v” and “b” to trigger the “walking” animation of the people and control its walking directions. And the user can also pause the “walking” animation by pressing key “p”. A screen shot of the walking people is shown in *Figure 47*.

Figure 47: the figure of walking people.

Since the people will keep walking until the users press the pause “p”, the continued walking is constructed by iterations of walking movements. One iteration of the walking animation takes up 1 second. In this second, the movements of the people include rising and curving the left leg, body moving forward, moving the right arm forward, moving the left arm backwards and curving the right leg, then dropping the left leg, body keeping moving forward, moving the right arm backward and moving forward the left arm then rising and curving the right leg. As mentioned in the Section 2, since all the components of people are in the same matrix as the body part, when body moves, the whole people will move. And for rising the legs and moving arms, since the left and right legs and arms are in different sub-matrices, other parts of the people will not be affected by their movements. Also, since the lower left and right legs are in sub-matrices of their corresponding legs’ matrices, when two leg matrices move, the upper parts of the legs will move as well as the lower parts. To enable the natural movements of people, the lower part will be further rotated in their own sub-matrices after the upper parts are moved along with the legs’ matrices to enable the effect of curving a leg.

This one iteration of movement is divided into 29 frames. In the first 14 frames, the left legs will rise and curve, the right arm will move forward, the body moves forward, the right leg will slightly curve and move backwards, the left arm will move backwards as well. Then in the next 14 frames, the left leg will drop, the right leg moves forward, the left arm moves forward, and the right arm moves backward. In the rest 14 frames, the right leg rises and curves, the left leg slightly curves and moves backward, the right arm moves backward and the left arm moves forward.

To enable the continued movement, the body matrix of the people (i.e. the matrix that contains all the parts of the people) will be implemented as a sub-matrix of another matrix (the “larger” matrix). During one movement iteration, the larger matrix will not move while the body matrix will move forward along the z-axis. When one iteration is finished, the distance that the people has moved is 56.f. In the last frame of the movement, the larger matrix will “catch up” the body matrix and overlap the direction and position of these two matrices. Then, at the second iteration, the people will keep walking forward until the iteration is over and the larger matrix catches up again. This enables the people keeping walking forward instead of moving forward and then back to the start place.

The code implementation of the first 4 frames of the people’s animation is elaborated in the *Figure 48.*

```
void People::Update(const double& deltaTime) {
    if (forward || back || left || right) {
        /*if (false) {*/
        _runtime = fmod(_runtime + deltaTime, animationTime);          // update time in animation cycle
        float stage = 29.f*_runtime / animationTime;          // calculate stage (out of 29)

        old_move = move;
        if (stage < 1) {
            counter = 0;
            left_arm_angle_1 = 3.f;
            right_arm_angle_1 = -3.f;
            left_leg_angle_1 = -6.f;
            left_leg_angle_2 = 2.f;
            //right_leg_angle_1 = 6.f;
            right_leg_angle_1 = 1.f;
            right_leg_angle_2 = 1.f;
            move = 2.f;



        }

        else if (stage < 2) {
            left_arm_angle_1 = 6.f;
            right_arm_angle_1 = -6.f;
            left_leg_angle_1 = -6.f;
            left_leg_angle_2 = 7.f;
            right_leg_angle_1 = 3.f;
            right_leg_angle_2 = 3.f;
            move = 4.f;
        }

        else if (stage < 3) {
            left_arm_angle_1 = 9.f;
            right_arm_angle_1 = -9.f;
            left_leg_angle_1 = -12.f;
            left_leg_angle_2 = 12.f;
            right_leg_angle_1 = 5.f;
            right_leg_angle_2 = 5.f;
            move = 6.f;
        }
        else if (stage < 4) {
            left_arm_angle_1 = 12.f;
            right_arm_angle_1 = -12.f;
            left_leg_angle_1 = -24.f;
            left_leg_angle_2 = 17.f;
            right_leg_angle_1 = 7.f;
            right_leg_angle_2 = 7.f;
            move = 8.f;
        }
```

Figure 48: the code for implementing the first four frames of the people's movement.

6 Interaction

6.1 Light Control

As demonstrated in the Animation Section, the users can use different keys to trigger animations. Also, they can control the lights using keyboard. They can turn on and turn off the three positional lights using the number 1 to 3 and turn on and off the directional light using number 0. They can also control if the objects will be affected by lighting by using the key "e". Additionally, they can control the on/off of all the lightings by pressing the key "n". These can be implemented because each lighting is controlled by a Boolean variable and the keys can change the variables to control the lightings' on and off in the *HandleKey()* function. There is also a Boolean variable for deciding if the objects will be affected by lightings and this variable is controlled by the key "e". A code snippet for controlling the three positional lights and one directional light using key board is displayed in the *Figure 49*.

```
void Lights::HandleKey(unsigned char key, int state, int x, int y) {
    if (!state) return;

    switch (key)
    {
    case 'n':
        light0  = !light0;
        light1  = !light1;
        light2  = !light2;
        light3 = !light3;
        break;
    case '0':
        light0 = !light0;
        break;

    case '1':
        light1 = !light1;
        break;

    case '2':
        light2 = !light2;
        break;

    case '3':
        light3 = !light3;
        break;



    }
}
```

Figure 49: the code for implementing keys “n”, “1”, “2”, “3”, “0” to control the three positional lights and one directional light.

6.2 Camera Control

Based on the existing camera controls, I added the function that the users can move the camera up and down using the up and down keys. This is implemented by adding two states in the “Camera” class’s *Update()* function. Theses two states will be adding or subtracting the current eyePosition with the up vector and they are controlled by the Boolean variables *upKey* and *downKey*. These two Boolean variables are then controlled by the up and down keys in the *HandleKey()* function. The code for implementing this function is displayed in *Figure 50*.

```
void Camera::Update(const double& deltaTime)
{
    float speed = 1.0f;

    if (aKey)
        sub(eyePosition, right, speed);

    if (dKey)
        add(eyePosition, right, speed);

    if (wKey)
        add(eyePosition, forward, speed);

    if (sKey)
        sub(eyePosition, forward, speed);

    if(upKey)
        add(eyePosition, up, speed);

    if(downKey)
        sub(eyePosition, up, speed);



    SetupCamera();
}
```

```
void Camera:: HandleSpecialKey(int key, int state, int x, int y) {
    switch (key) {
    case GLUT_KEY_DOWN:

        downKey = state;
        break;
    case GLUT_KEY_UP:

        upKey = state;
        break;
    }
}
```

Figure 50: the code for implementing the up and down keys to move the camera up and down.

Apart from these, another function for controlling camera is added to the scene. In the original settings, the users can reset the camera by press the space key. Now, another function is added to the space key so that when users press this key, they can see the scene from the back and press again they can reset the camera. The *Figure 51* shows the figure when the camera is set at the back of the scene.

Figure 50: the scene when the camera is set at the back.

To ensure that the users can still use the “w”, “s”, “a” ,“d” to control the forward, backward, left and right movements of the camera when the position of the camera is changed, the view direction vector (*vd*), forward vector (*forward*) and right vector (*right*) should all be updated. The code for implementing this in *HandleKey()* function is displayed in *Figure 51*.

```
case ' ':
    if (!state) return;
    back = !back;
    if (!back) {
        Reset();
    }
    else {
        eyePosition[0] = -250.f;
        eyePosition[1] = 500.f;
        eyePosition[2] = -550.f;

        vd[0] = 250.f;
        vd[1] = -300.f;
        vd[2] = 550.f;

        right[0] = -1.0f;
        right[1] = 0.0f;
        right[2] = 0.0f;

        forward[0] = 0.0f;
        forward[1] = 0.0f;
        forward[2] = 1.0f;


    }
```

Figure 51: update the directional vectors when the position of the camera is changed.