Lab05 for CPT205 Computer Graphics

Part 1. Viewing and Projections

This part will review and exercise coordinate systems and projections for rendering 3D scenes in OpenGL.

1.1 Coordinate System and Clipping Window

Let's think about how we describe objects in a two-dimensional (2D) or three-dimensional (3D) scene. Before specifying the position and size of an object, a reference frame (coordinate system) is set to measure and locate this object. If the points and lines are drawn on a computer screen (in 2D), a simple way is to specify the position of the rows and columns of these pixels (drawing elements) on the screen. For example, a full-HD screen has 1920 pixels from the left to right and 1080 pixels from the top to bottom. To specify a point in the middle of the screen, the point should be plotted at (960, 540), that is, 960th pixel from the left of the screen and 540th pixel down from the top of the screen.

In OpenGL, when drawing a scene in a window, the coordinate system should be specified and mapped into physical screen pixels. The most common coordinate system for 2D plotting is the **Cartesian coordinate system**, specified by an x coordinate and a y coordinate with the origin at (0, 0).

A window is measured physically in terms of **pixels**. Before drawing points, lines and polygons in a window, the specified coordinates are paired into screen coordinates. This mapping could be obtained by specifying the region of Cartesian space that occupies the window. This region is known as the **clipping region/window** (to be discussed in detail later during the module). In the 2D space, there are two ways to describe the clipping region. One is the minimum and maximum x and y values that are inside the window. The other is to specify the origin's location in relation to the window.

Figure 1 shows two common clipping regions (left: window's bottom-left corner at the origin and right: window's centre at the origin). The default mapping for Microsoft *Windows*™ window is with the x-coordinates increasing from the left to the right and the y-coordinates increasing from the top to the bottom. The location of the origin (relative to the screen) varies depending on whether you specify screen or client coordinates. Though useful for drawing text from top to bottom, this default mapping is not as convenient for drawing graphics, especially for OpenGL.

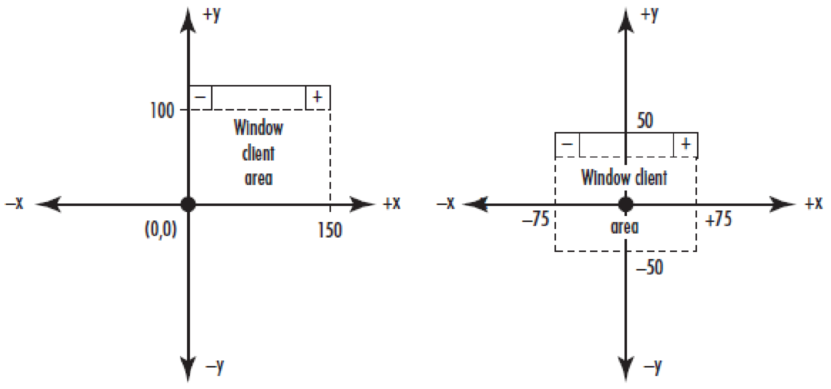


Figure 1. Clipping region [1]

1.2 Viewports

It is rare that the width and height of the clipping area exactly match those of the window in pixels. The coordinate system must be mapped from logical Cartesian coordinates to physical screen pixel coordinates. The mapping is specified by a setting known as the **viewport**. The viewport is the region within the window's area that is used for drawing the clipping area. The viewport simply maps the clipping area to a region of the window. Usually, the viewport is defined as the entire window, as shown in the left of Figure 2; however, it is not strictly necessary, and for example, the viewport can also be defined as the lower half of the window, as shown in the right of Figure 2.

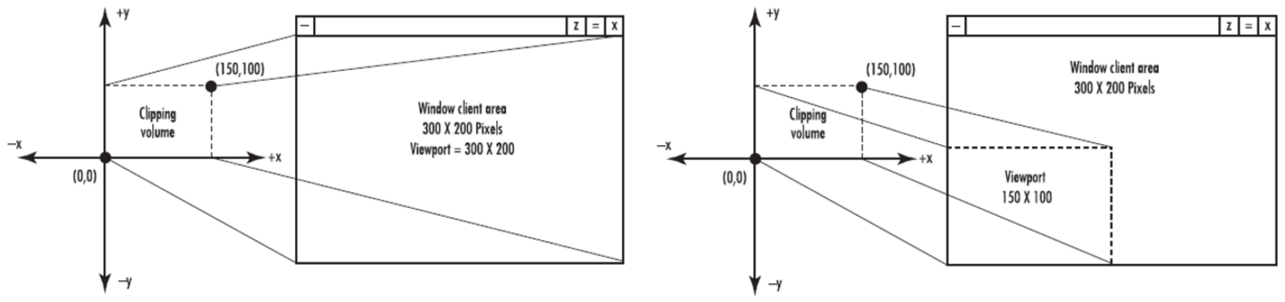


Figure 2. Viewport defined within the window area [1]

Therefore, in OpenGL, the viewport can be used to shrink or enlarge the image inside the window and display only a portion of the clipping area by setting the viewport to be larger than the window's area. The code below is an example for adjusting viewport in OpenGL, e.g., moving or change its size.

```
// Lab05_Viewport.cpp

#define FREEGLUT_STATIC
#include <math.h>
#include <GL/freeglut.h>
#include <iostream>

GLint scale = 1;
GLint stepx = 0;
GLint stepy = 0;
GLint width = 500;
GLint height = 500;

void myDisplay(void)
{
     glClear(GL_COLOR_BUFFER_BIT);
     glColor3f(1.0, 1.0, 1.0);
     glutWireTeapot(0.5);  // draw a wireframe teapot
     glFlush();
}

void idleDisplay()
{
     if (scale == 0)
          scale = 1;
     glViewport(stepx, stepy, width / scale, height / scale);

     glutPostRedisplay();  // force OpenGL to redraw the current window
}

void keyboard_input(unsigned char key, int x, int y)
{  // keyboard interaction
     if (key == 'q' || key == 'Q')
          exit(0);
     else if (key == 'l' || key == 'L')  // move the viewport to the left
          stepx--;
     else if (key == 'u' || key == 'U')  // move the viewport up
          stepy++;
     else if (key == 'r' || key == 'R')  // move the viewport to the right
          stepx++;
     else if (key == 'd' || key == 'D')  // move the viewport down
          stepy--;
     else if (key == 's' || key == 'S')  // shrink the viewport
          scale++;
     else if (key == 'e' || key == 'E')  // enlarge the viewport
          scale--;
}

int main(int argc, char** argv)
{
     glutInit(&argc, argv);
     glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
     glutInitWindowSize(width, height);
     glutCreateWindow("Teapot");

     glutDisplayFunc(myDisplay);
     glutKeyboardFunc(keyboard_input);
     glutIdleFunc(idleDisplay);
```

```
    glutMainLoop();
}
```

1.3 Projections

As discussed during the lecture, there are two main types of projection, **orthogonal projection** and **perspective projection**. For the orthogonal projection, a cubic or rectangular viewing volume is specified as the clipping volume. Anything outside this volume is not drawn. In addition, all objects with the same dimensions appear the same size, regardless whether they are far away or nearby. This projection is used in architectural design, computer-aided design (CAD), and 2D graphs. Texts / 2D overlays can be added on top of 3D graphic scene.

The code below shows an example of orthogonal projection in OpenGL.

```
// Lab05_Orthogonal_projection.cpp

#define FREEGLUT_STATIC
#include <math.h>
#include <GL/freeglut.h>
#include <iostream>

// Angles of rotation
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

// Change the view volume and viewport. This is called when the window is resized.
void ChangeSize(int w, int h)
{
    if (h == 0)  // Avoid division by zero
        h = 1;

    // Set viewport to window dimensions
    glViewport(0, 0, w, h);

    // Reset coordinate system
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Set viewing volume
    glOrtho(-100.0f, 100.0f, -100.0f, 100.0f, -200.0f, 200.0f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

// Setting up lighting and material parameters for enhanced rendering effect.
// This topic will be covered later on in the module so please skip this for now.
void SetupRC()
{
    // Light parameters and coordinates
    GLfloat whiteLight[] = { 0.45f, 0.45f, 0.45f, 1.0f };
    GLfloat sourceLight[] = { 0.25f, 0.25f, 0.25f, 1.0f };
    GLfloat lightPos[] = { -50.f, 25.0f, 250.0f, 0.0f };

    glEnable(GL_DEPTH_TEST);  // Hidden surface removal

    glEnable(GL_LIGHTING);  // Enable lighting

    // Setup and enable light 0
    glLightModelfv(GL_LIGHT_MODEL_AMBIENT, whiteLight);
    glLightfv(GL_LIGHT0, GL_AMBIENT, sourceLight);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, sourceLight);
    glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
    glEnable(GL_LIGHT0);

    glEnable(GL_COLOR_MATERIAL);  // Enable colour tracking

    // Set material properties to follow glColor values
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);  // Black background
}
```

```
// Respond to arrow keys
void SpecialKeys(int key, int x, int y)
{
	if (key == GLUT_KEY_UP)
		xRot -= 5.0f;

	if (key == GLUT_KEY_DOWN)
		xRot += 5.0f;

	if (key == GLUT_KEY_LEFT)
		yRot -= 5.0f;

	if (key == GLUT_KEY_RIGHT)
		yRot += 5.0f;

	xRot = (GLfloat)((const int)xRot % 360);  // Reminder of xRot divided by 360
	yRot = (GLfloat)((const int)yRot % 360);  // Reminder of yRot divided by 360

	// Refresh the window
	glutPostRedisplay();
}

// Draw scene
void RenderScene(void)
{
	// Clear the window with current clearing color
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	//Perform the depth test to render multiple objects in the correct order of Z-axis value
	glEnable(GL_DEPTH_TEST); // Hidden surface removal

	glPushMatrix();  // Save the matrix state and perform rotations
	glRotatef(xRot, 1.0f, 0.0f, 0.0f);
	glRotatef(yRot, 0.0f, 1.0f, 0.0f);
	glColor3f(1.0f, 1.0f, 1.0f);
	glutSolidTeapot(25);
	glPushMatrix();
	glTranslatef(0.0f, 0.0f, -100.0f);
	glColor3f(0.0f, 0.0f, 1.0f);
	glutSolidTeapot(25);
	glPopMatrix();
	glPopMatrix();  // Restore the matrix state

	glutSwapBuffers();
}

int main(int argc, char* argv[])
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
	glutInitWindowSize(800, 600);
	glutCreateWindow("Orthogonal Projection");
	glutReshapeFunc(ChangeSize);
	glutSpecialFunc(SpecialKeys);
	glutDisplayFunc(RenderScene);
	SetupRC();  // lighting and material effect
	glutMainLoop();

	return 0;
}
```

The perspective projection adds the effect that distant objects appear smaller than nearby objects of the same size. The viewing/clipping volume is often a frustum (a pyramid with the top shaved off). The objects nearer to the front of the viewing volume appear close to their original size, and on contrary, the objects nearer to the back of the viewing volume shrink as they are projected to the front of the volume, as shown in Figure 3. This projection provides the most realism for simulation and 3D animation.

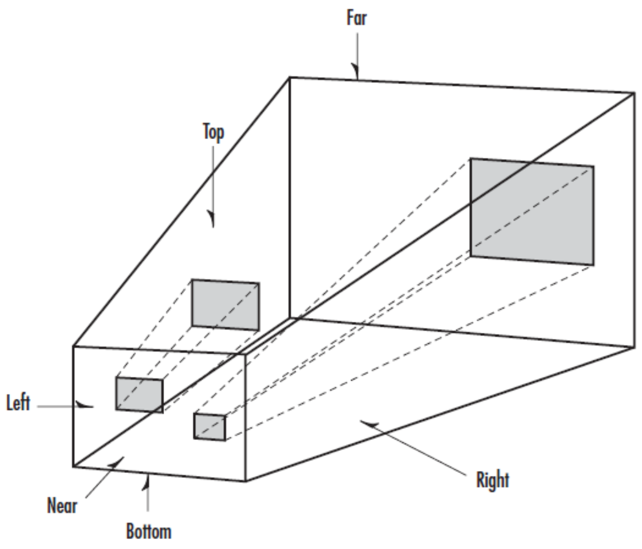


Figure 3. Viewing volume for perspective projection [1]

The code below provides an example for perspective projection in OpenGL.

```
// Lab05_Perspective_projection.cpp

#define FREEGLUT_STATIC
#include <math.h>
#include <GL/freeglut.h>
#include <iostream>

// Angles of rotation
static GLfloat xRot = 0.0f;
static GLfloat yRot = 0.0f;

// Change the view volume and viewport. This is called when the window is resized.
void ChangeSize(int w, int h)
{
     GLfloat fAspect;

     if (h == 0) // Avoid division by zero
          h = 1;

     glViewport(0, 0, w, h);  // Set viewport to window dimensions

     fAspect = (GLfloat)w / (GLfloat)h;

     glMatrixMode(GL_PROJECTION);  // Reset coordinate system
     glLoadIdentity();

     gluPerspective(60.0f, fAspect, 1.0, 600.0);  // Set viewing volume

     glMatrixMode(GL_MODELVIEW);
     glLoadIdentity();
}

// Setting up lighting and material parameters for enhanced rendering effect.
// This topic will be covered later on in the module so please skip this for now.
void SetupRC()
{
     // Light parameters and coordinates
     GLfloat  whiteLight[] = { 0.45f, 0.45f, 0.45f, 1.0f };
     GLfloat  sourceLight[] = { 0.25f, 0.25f, 0.25f, 1.0f };
     GLfloat   lightPos[] = { -50.f, 25.0f, 250.0f, 0.0f };

     glEnable(GL_DEPTH_TEST);  // Hidden surface removal

     glEnable(GL_LIGHTING);  // Enable lighting

     // Setup and enable light0
     glLightModelfv(GL_LIGHT_MODEL_AMBIENT, whiteLight);
     glLightfv(GL_LIGHT0, GL_AMBIENT, sourceLight);
     glLightfv(GL_LIGHT0, GL_DIFFUSE, sourceLight);
     glLightfv(GL_LIGHT0, GL_POSITION, lightPos);
```

```
	glEnable(GL_LIGHT0);

	glEnable(GL_COLOR_MATERIAL);  // Enable colour tracking

	// Set material properties to follow glColor values
	glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);

	glClearColor(0.0f, 0.0f, 0.0f, 1.0f);  // Black background
}

// Respond to arrow keys
void SpecialKeys(int key, int x, int y)
{
	if (key == GLUT_KEY_UP)
		xRot -= 5.0f;

	if (key == GLUT_KEY_DOWN)
		xRot += 5.0f;

	if (key == GLUT_KEY_LEFT)
		yRot -= 5.0f;

	if (key == GLUT_KEY_RIGHT)
		yRot += 5.0f;

	xRot = (GLfloat)((const int)xRot % 360);  // Reminder of xRot divided by 360
	yRot = (GLfloat)((const int)yRot % 360);  // Reminder of yRot divided by 360

	glutPostRedisplay();  // Refresh the window
}

// Draw scene
void RenderScene(void)
{
	// Clear the window with current clearing color
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	//Perform the depth test to render multiple objects in the correct order of Z-axis value
	glEnable(GL_DEPTH_TEST); // Hidden surface removal

	glPushMatrix();  // Save the matrix state and perform rotations
	glTranslatef(0.0f, 0.0f, -300.0f);
	glRotatef(xRot, 1.0f, 0.0f, 0.0f);
	glRotatef(yRot, 0.0f, 1.0f, 0.0f);
	glColor3f(1.0f, 1.0f, 1.0f);
	glutSolidTeapot(25);
	glPushMatrix();
	glTranslatef(0.0f, 0.0f, -100.0f);
	glColor3f(0.0f, 0.0f, 1.0f);
	glutSolidTeapot(25);
	glPopMatrix();
	glPopMatrix();  // Restore the matrix state
	glutSwapBuffers();
}

int main(int argc, char* argv[]) {
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
	glutInitWindowSize(800, 600);
	glutCreateWindow("Perspective Projection");
	glutReshapeFunc(ChangeSize);
	glutSpecialFunc(SpecialKeys);
	glutDisplayFunc(RenderScene);
	SetupRC();  // lighting and material effect
	glutMainLoop();

	return 0;
}
```

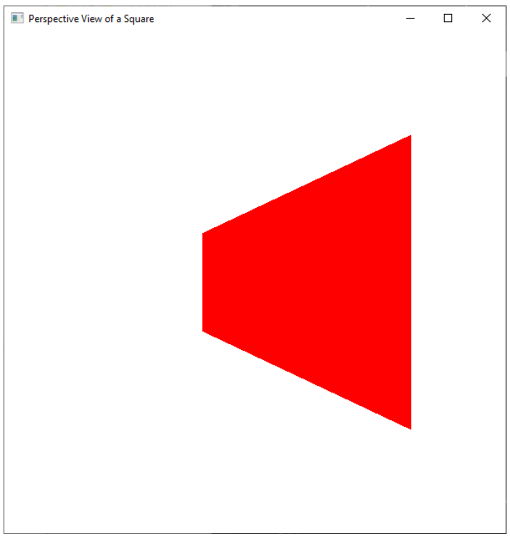
1.4 Sample Code for Lecture 05

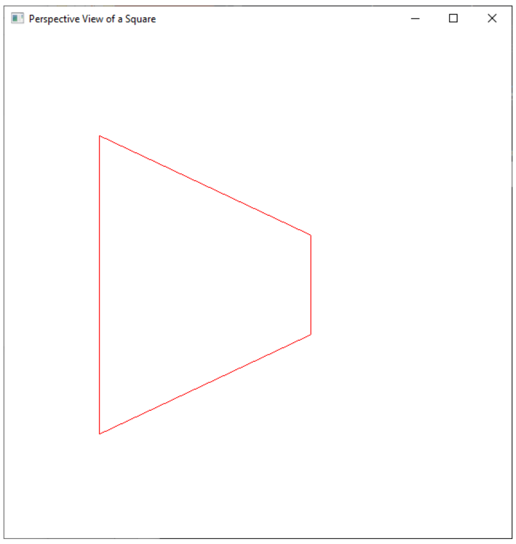
Task 1: Carefully read this sample code and the lecture ppt slides about this sample code.

Task 2: Run the sample code and try to understand the spatial relationship between the viewing co-ordinate system and the world co-ordinate system,

Task 3: Adjust some of the parameters in a sensible way and see what you get.

```cpp
// Smaple_Lecture05.cpp

#define FREEGLUT_STATIC
#include <gl/freeglut.h>

GLint winWidth = 600, winHeight = 600;          // initial display window size

GLfloat x0 = 100.0, y0 = 50.0, z0 = -50.0;      // viewing co-ordinate origin
GLfloat xref = 50.0, yref = 50.0, zref = 0.0;   // look-at point
GLfloat Vx = 0.0, Vy = 1.0, Vz = 0.0;           // view-up vector

// co-ordinate limits for clipping window
GLfloat xwMin = -40.0, ywMin = -60.0, xwMax = 40.0, ywMax = 60.0;

// positions for near and far clipping planes
GLfloat dnear = 25.0, dfar = 125.0;

void init(void)
{
    glClearColor(1.0, 1.0, 1.0, 0.0);

    // decides which matrix is to be affected by subsequent transform functions
    glMatrixMode(GL_MODELVIEW);
    // define camera / viewing coordinate system
    gluLookAt(x0, y0, z0, xref, yref, zref, Vx, Vy, Vz);

    glMatrixMode(GL_PROJECTION);  // projection transformations
    glFrustum(xwMin, xwMax, ywMin, ywMax, dnear, dfar);  // define perspective frustum
}

void displayFcn(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    // parameters for a square fill area
    glColor3f(1.0, 0.0, 0.0);
    glPolygonMode(GL_FRONT, GL_FILL);  // fill in front face of polygon
    glPolygonMode(GL_BACK, GL_LINE);  // keep back face of polygon as wireframe/line
    glBegin(GL_QUADS);
    glVertex3f(0.0, 0.0, 0.0);
    glVertex3f(100.0, 0.0, 0.0);
    glVertex3f(100.0, 100.0, 0.0);
    glVertex3f(0.0, 100.0, 0.0);
    glEnd();

    glFlush();
}

void reshapeFcn(GLint newWidth, GLint newHeight)
{
    glViewport(0, 0, newWidth, newHeight); // Set viewport to window dimensions
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition(50, 50);
    glutInitWindowSize(winWidth, winHeight);
    glutCreateWindow("Perspective View of a Square");

    init();
    glutDisplayFunc(displayFcn);
    glutReshapeFunc(reshapeFcn);
    glutMainLoop();
}
```

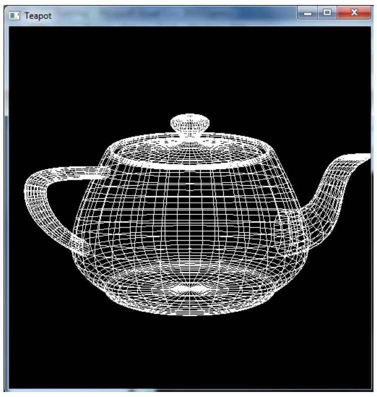




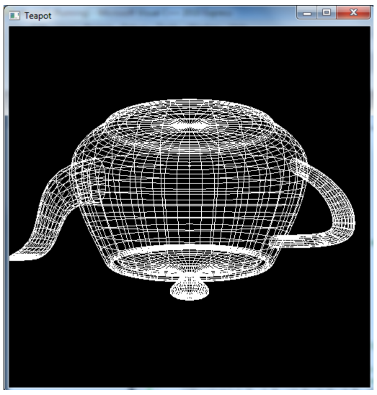
1.5 Viewing/Projection Functions

Task1: First carefully read and understand the following code. You will see the image below on running it.

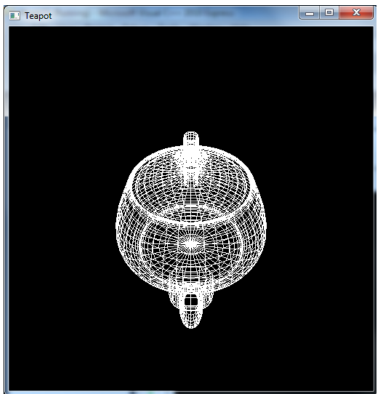
```
// Lab05_Projection_Functions.cpp
// A teapot for viewing

#define FREEGLUT_STATIC
#include <math.h>
#include <iostream>
#include <gl/freeglut.h>

void myDisplay(void)
{
     glClear(GL_COLOR_BUFFER_BIT);
     glColor3f(1.0, 1.0, 1.0);
     glutWireTeapot(2);  // draw a wireframe teapot
     glFlush();
}

void myinit(void)
{
     glShadeModel(GL_FLAT);  // Flat shading specified
}

void myReshape(GLsizei w, GLsizei h)
{
     glViewport(0, 0, w, h);  // define viewport

     glMatrixMode(GL_PROJECTION);  // projection transformations
     glLoadIdentity();  // clear the projection matrix
     glFrustum(-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);  // set the perspective frustum

     glMatrixMode(GL_MODELVIEW);  // back to model-view matrix to define camera
     glLoadIdentity();  // clear the model-view matrix
     gluLookAt(0, 0, 5, 0, 0, 0, 0, 1, 0);  // set up camera / viewing coordinate system
}

int main(int argc, char** argv)
{
     glutInit(&argc, argv);
     glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
     glutInitWindowSize(500, 500);
     glutCreateWindow("Teapot");

     myinit();
     glutReshapeFunc(myReshape);
     glutDisplayFunc(myDisplay);
     glutMainLoop();
}
```



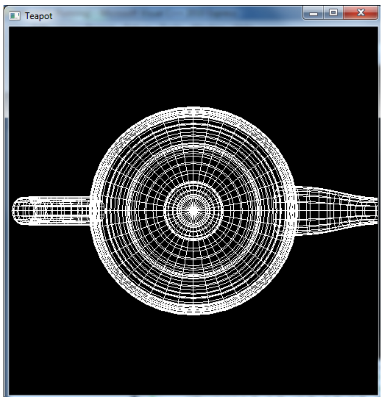
Task2: Specify different values for the **gluLookAt** function to see the following image.



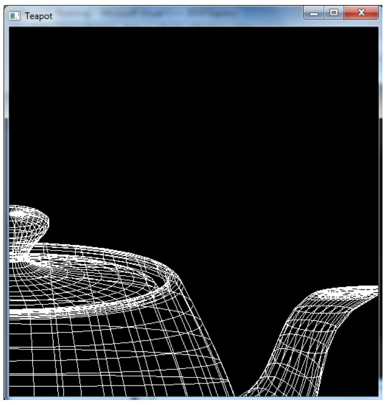
Task3: Specify different values for the **gluLookAt** function to see the following image.



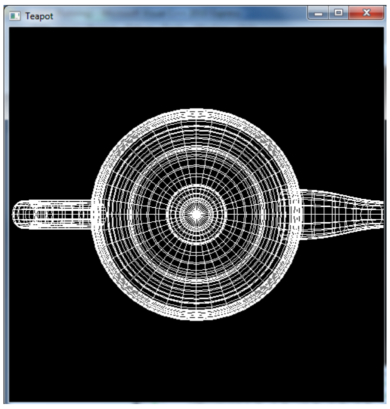
Task4: Specify different values for the **gluLookAt** function to see the following image.



Task5: Specify different values for the **glFrustum** function to see the following image.



Task6: Add the **glRotatef** function to see the following image.



Task7: You should keep in mind what functions are used (e.g., geometric transformations or projection transformations), how they work and can be called. You may now want to revisit the functions you have used.

Reference

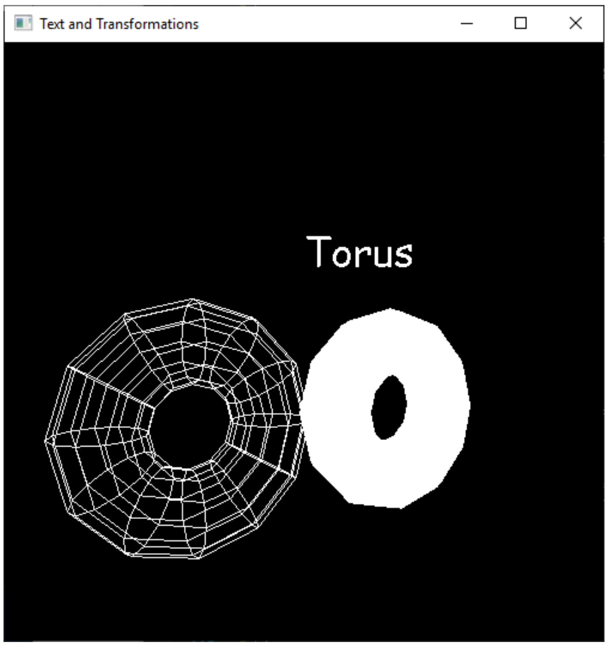
[1] Graham Sellers, Richard Wright Jr., Nicholas Haemel. OpenGL Super Bible: Comprehensive Tutorial and Reference. Pearson Education, 2015.

Part 2. Text and Transformations

```
// Lab05_Text_and_Transformations.cpp

#define FREEGLUT_STATIC
#include <math.h>
#include <GL/freeglut.h>
#include <iostream>
#include "windows.h"
#define MAX_CHAR 128

GLdouble x_p = 0;  // parameter to define camera position in x-direction
GLdouble z_p = 6;  // parameter to define camera position in z-direction
int flag = 1;  // flag for value ranges of x_p and z_p

void drawString(const char* str)
{
     static int isFirstCall = 1;  // lazy init flag
     static GLuint lists;   //base id of the generated display lists

     if (isFirstCall)
     {
          isFirstCall = 0;
          lists = glGenLists(MAX_CHAR);  // reserve MAX_CHAR consecutive list IDs
          // Build bitmap glyph lists for the font currently selected into the DC. The parameters represent:
          1) Device Context, 2) first glyph index, 3) number of glyph to build, 4) base list id.
          wglUseFontBitmaps(wglGetCurrentDC(), 0, MAX_CHAR, lists);
     }

// render each character by calling its display list
     for (;*str != '\0';++str)
     {
          glCallList(lists + *str);
     }
}

// Create and select a GDI font into the current Device Context (used by wglUseFontBitmap).
// Note: selectFont should be called before the first drawString so that glyph lists are built
//        for the intended typeface/size/charset.
void selectFont(int size, int charset, const char* face)
{
     // Create a logical GDI font and select it into the current device context
     HFONT hFont = CreateFontA(size, 0, 0, 0, FW_MEDIUM, 0, 0, 0, charset, OUT_DEFAULT_PRECIS,
     CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY, DEFAULT_PITCH | FF_SWISS, face);
     HFONT hOldFont = (HFONT)SelectObject(wglGetCurrentDC(), hFont);  // activate new font in DC
     DeleteObject(hOldFont);  // discard the previous font object
}

void myDisplay(void)
{
     glClear(GL_COLOR_BUFFER_BIT);
     glColor3f(1.0, 1.0, 1.0);

     glMatrixMode(GL_PROJECTION);  //projection transformation
     glLoadIdentity();// clear the matrix
     glFrustum(-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
     gluLookAt(x_p, 0, z_p, 0, 0, 0, 0, 1, 0);

     glMatrixMode(GL_MODELVIEW);  // back to modelview matrix
     glLoadIdentity();  // clear the matrix

     glColor3f(1.0, 1.0, 1.0);
     //draw Torus
     glPushMatrix();
     glTranslated(1.5, -1.0, 0.0);
     glutWireTorus(0.5, 1, 12, 12);  // inner radius, out radius, number of sides for each
     glPopMatrix();                 // radial section, number of radial divisions

     glPushMatrix();
     glTranslated(-1.5, -1.0, 0.0);
     glutSolidTorus(0.5, 1, 12, 12);
     glPopMatrix();

     //Draw 2D text
     selectFont(48, ANSI_CHARSET, "Comic Sans MS");
     glRasterPos2f(0.0, 1.0);
     drawString("Torus");
```

```c
    glFlush();
}

void myinit(void)
{
    glShadeModel(GL_FLAT);
}

void myReshape(GLsizei w, GLsizei h)
{
    glViewport(0, 0, w, h);  // define the viewport
}

void mouse_input(int button, int state, int x, int y)  // mouse interaction
{
    /*
    * Mouse handler:
    * On each LEFT click, move the camera to the next point (x_p, z_p) on the
    * circle x_p^2 + z_p^2 = 36 in the XZ plane (R = 6). We keep x_p in steps of 1
    * and recompute z_p = ±sqrt(36 - x_p^2). The sign is controlled by 'flag'
    * to traverse the full circle in four arcs:
    *   flag = 1 : z >= 0,  x : 0 to +6  (upper-right arc)
    *   flag = 2 : z <= 0,  x : +6 to 0   (lower-right arc)
    *   flag = 3 : z <= 0,  x : 0 to -6  (lower-left  arc)
    *   flag = 4 : z >= 0,  x : -6 to 0   (upper-left  arc)
    */
    if (state == GLUT_DOWN && button == GLUT_LEFT_BUTTON)
    {
        if (x_p < 6 && x_p >= 0 && flag == 1)  // Quadrant I:  x: 0 to +6,  z ≥ 0
        {
            x_p = x_p + 1;
            z_p = sqrt(36 - x_p * x_p);
        }
        else if (x_p == 6 && flag == 1)  // Hit (+6,0): switch to lower arc
        {
            x_p = x_p - 1;
            z_p = -sqrt(36 - x_p * x_p);
            flag = 2;
        }
        else if (x_p < 6 && x_p>0 && flag == 2)   // Quadrant IV: x: +6 to 0,  z ≤ 0
        {
            x_p = x_p - 1;
            z_p = -sqrt(36 - x_p * x_p);
        }
        else if (x_p == 0 && flag == 2)  // Cross x=0: continue to negative side
        {
            x_p = x_p - 1;
            z_p = -sqrt(36 - x_p * x_p);
            flag = 3;
        }
        else if (x_p > -6 && x_p < 0 && flag == 3)  // Quadrant III: x: 0 to −6, z ≤ 0
        {
            x_p = x_p - 1;
            z_p = -sqrt(36 - x_p * x_p);
        }
        else if (x_p == -6 && flag == 3)  // Hit (−6,0): switch back to upper arc
        {
            x_p = x_p + 1;
            z_p = sqrt(36 - x_p * x_p);
            flag = 4;
        }
        else if (x_p > -6 && x_p < 0 && flag == 4)   // Quadrant II: x: −6 to 0, z ≥ 0
        {
            x_p = x_p + 1;
            z_p = sqrt(36 - x_p * x_p);
        }
        else if (x_p == 0 && flag == 4)   // Completed a loop: reset to state 1s
        {
            x_p = x_p + 1;
            z_p = sqrt(36 - x_p * x_p);
            flag = 1;
        }
        glutPostRedisplay();
    }
}
```

```
int main(int argc, char** argv)
{
	glutInit(&argc, argv);
	glutInitDisplayMode(GLUT_SINGLE | GLUT_RGBA);
	glutInitWindowSize(500, 500);
	glutCreateWindow("Text and Transformations");
	myinit();
	glutReshapeFunc(myReshape);
	glutDisplayFunc(myDisplay);
	glutMouseFunc(mouse_input);
	glutMainLoop();
}
```

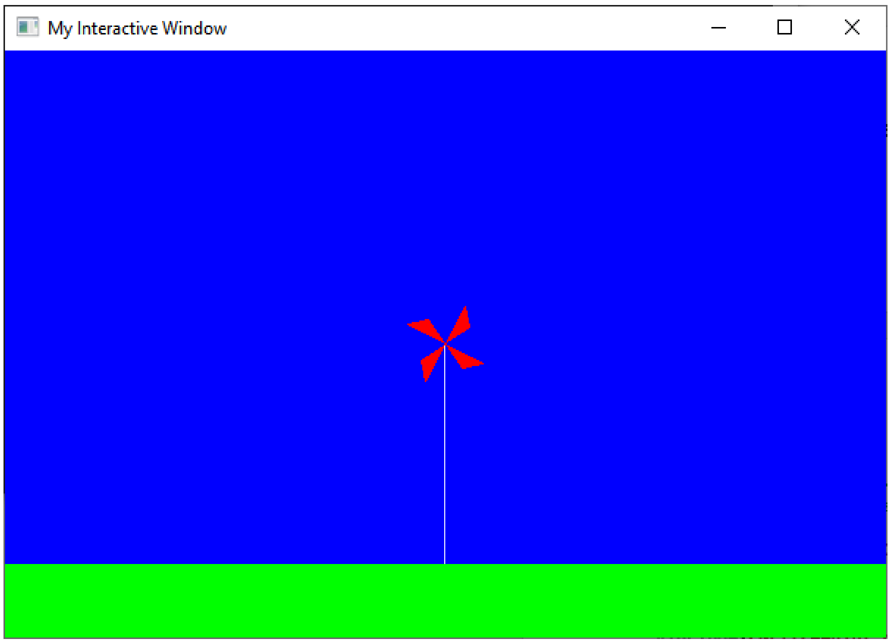


Part 3. Further Transformations and Creation of Geometry (Part 2a for Lab04)

1) Further Transformations

```
// Lab05_Further_transformations.cpp

#define FREEGLUT_STATIC
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <GL/freeglut.h>
#include <iostream>

using namespace std;
float r = 0;

//Task 2
GLfloat step = 1;  // declare step
int time_interval = 16;  // declare refresh interval in ms

void when_in_mainloop()  // idle callback function
{
	glutPostRedisplay();  // force OpenGL to redraw the current window
}

void OnTimer(int value)
{
	r += step;
	if (r >= 360)
		r = 0;
	else if (r <= 0)
		r = 359;

	glutTimerFunc(time_interval, OnTimer, 1);
}

//Task 3
void keyboard_input(unsigned char key, int x, int y) // keyboard interaction
{
	if (key == 'q' || key == 'Q')
		exit(0);
	else if (key == 'f' || key == 'F')// change direction of movement
		step = -step;
	else if (key == 's' || key == 'S')// stop
		step = 0;
	else if (key == 'r' || key == 'R')// set step
		step = 10;
}

//Task 4
void mouse_input(int button, int state, int x, int y)  // mouse interaction
{
	if (state == GLUT_DOWN && button == GLUT_LEFT_BUTTON && step >= -15)
		step -= 1;  // decrement step
	else if (state == GLUT_DOWN && button == GLUT_RIGHT_BUTTON && step <= 15)
		step += 1;  // increment step
}

void display(void)
{
	glMatrixMode(GL_PROJECTION);
	glLoadIdentity();
	gluOrtho2D(0, 600, 0, 400);

	glClearColor(0, 0, 1, 1);
	glClear(GL_COLOR_BUFFER_BIT);

	// draw a vertical line in white
	glPushMatrix();
	glTranslatef(300, 0, 0);
	glColor3f(1, 1, 1);
	glBegin(GL_LINES);
	glVertex2f(0, 50);
	glVertex2f(0, 200);
	glEnd();
	glPopMatrix();
```

```
    // draw a rectangle in green
    glPushMatrix();
    glTranslatef(0, 50, 0);
    glScalef(1, 2, 1);
    glColor3f(0, 1, 0);
    glBegin(GL_POLYGON);
    glVertex2f(0, 0);
    glVertex2f(0, -25);
    glVertex2f(600, -25);
    glVertex2f(600, 0);
    glEnd();
    glPopMatrix();

    // draw a windmill (4 triangles spaced 90° apart from each other) in red
    glPushMatrix();  // save the current model-view matrix (before 1st triangle)
    glColor3f(1, 0, 0);
    glTranslatef(300, 200, 0);
    glRotatef(r, 0, 0, 1);
    glBegin(GL_TRIANGLES);
    glVertex2f(0, 0);
    glVertex2f(0, 30);
    glVertex2f(10, 18);
    glEnd();
    glPushMatrix();  // save the current model-view matrix (1st triangle)
    glRotatef(90, 0, 0, 1);
    glBegin(GL_TRIANGLES);
    glVertex2f(0, 0);
    glVertex2f(0, 30);
    glVertex2f(10, 18);
    glEnd();
    glPushMatrix();  // save the current model-view matrix (2nd triangle)
    glRotatef(90, 0, 0, 1);
    glBegin(GL_TRIANGLES);
    glVertex2f(0, 0);
    glVertex2f(0, 30);
    glVertex2f(10, 18);
    glEnd();
    glPushMatrix();  // save the current model-view matrix (3rd triangle)
    glRotatef(90, 0, 0, 1);
    glBegin(GL_TRIANGLES);
    glVertex2f(0, 0);
    glVertex2f(0, 30);
    glVertex2f(10, 18);
    glEnd();
    glPopMatrix();  // discard the current model-view matrix (return to 3rd triangle)
    glPopMatrix();  // discard the current model-view matrix (return to 2nd triangle)
    glPopMatrix();  // discard the current model-view matrix (return to 1st triangle)
    glPopMatrix();  // discard the current model-view matrix (return to before 1st triangle)

    glutSwapBuffers();
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);

    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(600, 400);
    glutCreateWindow("My Interactive Window");

    glutDisplayFunc(display);

    //Task 2
    glutIdleFunc(when_in_mainloop);
    glutTimerFunc(time_interval, OnTimer, 1);

    //Task 3
    glutKeyboardFunc(keyboard_input);

    //Task 4
    glutMouseFunc(mouse_input);

    glutMainLoop();
}
```



2) Creation of Geometry

```
// Drawing a house with wireframe polygons and with filled polygons
// Lab05_Drawing_a_house.cpp

#define FREEGLUT_STATIC
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <GL/freeglut.h>

void define_to_OpenGL();
bool flag;  // flag for wireframe or filled house

void when_in_mainloop()  // idle callback function
{
     glutPostRedisplay();  // force OpenGL to redraw the current window
}

void keyboard_input(unsigned char key, int x, int y)  // keyboard interaction
{
     if (key == 'q' || key == 'Q')
     {
          exit(0);
     }
     else if (key == 'f' || key == 'F')
     {  //press ‘f’ to toggle polygons drawn in between ‘line’ and ‘fill’ modes
          if (flag == false)
               flag = true;
          else
               flag = false;
     }
}

void define_to_OpenGL()
{
     glClearColor(1, 1, 1, 1);
     glClear(GL_COLOR_BUFFER_BIT);

     if (flag == false)
          glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
     else if (flag == true)
          glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);

     //This draws reference axes
     glColor3f(0.0, 0.0, 0.0);
     glLineStipple(2, 0x5555);
     glEnable(GL_LINE_STIPPLE);
```

```
glBegin(GL_LINES);
glVertex2f(-1.0, 0.0);
glVertex2f(1.0, 0.0);
glEnd();
glBegin(GL_LINES);
glVertex2f(0.0, -1.0);
glVertex2f(0.0, 1.0);
glEnd();
glDisable(GL_LINE_STIPPLE);

//The wall
glColor3f(153.0 / 255.0, 51.0 / 255.0, 102.0 / 255.0);
glBegin(GL_POLYGON);
glVertex2f(-0.5, 0);
glVertex2f(-0.5, 0.5);
glVertex2f(-0.25, 0.5);
glVertex2f(-0.25, 0);
glEnd();

//The roof
glColor3f(255.0 / 255.0, 66.0 / 255.0, 14.0 / 255.0);
glBegin(GL_POLYGON);
glVertex2f(-0.9, 0);
glVertex2f(0, 0.5);
glVertex2f(0.9, 0);
glEnd();

//The chimney
glColor3f(153.0 / 255.0, 204.0 / 255.0, 255.0 / 255.0);
glBegin(GL_POLYGON);
glVertex2f(-0.5, -0.9);
glVertex2f(-0.5, 0);
glVertex2f(0.5, 0);
glVertex2f(0.5, -0.9);
glEnd();

//The window glass
float width = 0.1;
float height = 0.1;

glColor3f(255.0 / 255.0, 204.0 / 255.0, 153.0 / 255.0);
float x_shift = -0.3;
float y_shift = -0.3;
glBegin(GL_POLYGON);
glVertex2f(-width + x_shift, -height + y_shift);
glVertex2f(-width + x_shift, height + y_shift);
glVertex2f(width + x_shift, height + y_shift);
glVertex2f(width + x_shift, -height + y_shift);
glEnd();

x_shift = -0.1;
y_shift = -0.3;
glBegin(GL_POLYGON);
glVertex2f(-width + x_shift, -height + y_shift);
glVertex2f(-width + x_shift, height + y_shift);
glVertex2f(width + x_shift, height + y_shift);
glVertex2f(width + x_shift, -height + y_shift);
glEnd();

x_shift = -0.3;
y_shift = -0.5;
glBegin(GL_POLYGON);
glVertex2f(-width + x_shift, -height + y_shift);
glVertex2f(-width + x_shift, height + y_shift);
glVertex2f(width + x_shift, height + y_shift);
glVertex2f(width + x_shift, -height + y_shift);
glEnd();

x_shift = -0.1;
y_shift = -0.5;
glBegin(GL_POLYGON);
glVertex2f(-width + x_shift, -height + y_shift);
glVertex2f(-width + x_shift, height + y_shift);
glVertex2f(width + x_shift, height + y_shift);
glVertex2f(width + x_shift, -height + y_shift);
glEnd();
```

```
    //The windows edge
    glColor3f(0.0, 0.0, 0.0);
    x_shift = 0.3;
    y_shift = -0.3;
    glBegin(GL_LINES);
    glVertex2f(-width - x_shift, -height + y_shift);
    glVertex2f(0.0, -height + y_shift);
    glEnd();

    glBegin(GL_LINES);
    glVertex2f(width - x_shift, height + y_shift);
    glVertex2f(width - x_shift, -height * 3 + y_shift);
    glEnd();

    //The door
    glColor3f(102.0 / 255.0, 51.0 / 255.0, 0.0 / 255.0);
    glBegin(GL_POLYGON);
    glVertex2f(0.1, -0.9);
    glVertex2f(0.1, -0.2);
    glVertex2f(0.4, -0.2);
    glVertex2f(0.4, -0.9);
    glEnd();

    glFlush();
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutCreateWindow("My House");
    flag = false;
    glutIdleFunc(when_in_mainloop);
    glutDisplayFunc(define_to_OpenGL);
    glutKeyboardFunc(keyboard_input);
    glutMainLoop();
}
```