FACULTY OF COMPUTER AND INFORMATION SCIENCES, AIN SHAMS UNIVERSITY

Lab 3

3D Drawing and Transformations

OBJECTIVES:

1. **Be able to set the 3D viewing environment.**
2. **Be able to draw the cube.**
3. **Be able to use idle event to handle animation.**
4. **Be able to draw predefined opengl objects .**

1. SETTING THE 3D ENVIRONMENT

In the last two sections we should know how to set the 2D orthographic environment using the gluOrtho2D command, in this section we will show you how to set the 3D environment.

When working with 3D objects the answer to the question "how to convert from 3D drawings to 2D drawings on the screen?" is needed; In other words which projection will be used?

Orthographic Projection vs. Prespective Projection

Orthographic Projection: Simply, it ignores the z-component of the point. Good for engineering designs but not realistic because if objects far from the eye should appear smaller than objects close the eye. However, this doesn't happen in orthographic projection.

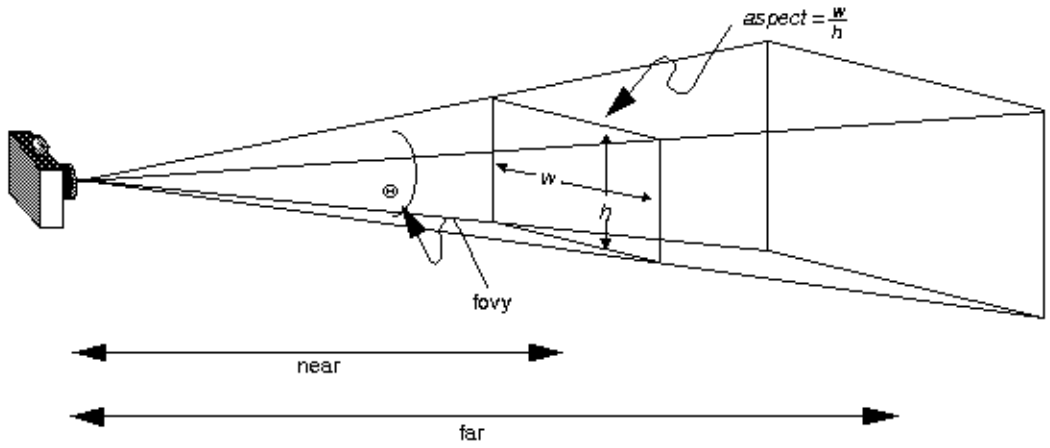
Prespective Projection: Things far from the eye look smaller. It tries to model the viewing of the human eye: First, a ray from the point to the eye is drawn, and its projection on the near plane is observed. The projection on the near plane is what we see on the screen.

The following code will show how to initialize OpenGl using perspective projection.

```
void SetTransformations()
{
      //set up the logical coordinate system of the window: [-
100, 100] x [-100, 100]
      glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    gluPerspective(60,
                                (double)800/ 600,
                                0.01,
                                1000);

    glEnable(GL_DEPTH_TEST);

// set the camera here

}
```

The command **gluPerspective()** creates a viewing volume of the shape of a frustum, by specifying the angle of the field of view in the *x-z* plane and the aspect ratio of the width to height (*x/y*). (For a square portion of the screen, the aspect ratio is 1.0.) These two

parameters are enough to determine an untruncated pyramid along the line of sight, as shown in the figure below. You also specify the distance between the viewpoint and the near and far clipping planes, thereby truncating the pyramid. Note that **gluPerspective()** is limited to creating frustums that are symmetric in both the *x*- and *y*-axes along the line of sight, but this is usually what you want.



The command **glEnable()** enables a given OpenGL capability, in our case we are enabling the depth buffer. The depth buffer holds the depth of each pixels of the frame. It is also called z buffer.

Depth buffer is associated with the Depth Test. For each pixel drawn, the Depth Test compares the current depth stored in the depth buffer with the depth of the new pixel to draw. Depending on the result of the test, the pixel is drawn or not.

We usually use the depth buffer to draw nearest pixels, pixels behind are not drawn.

Now, what we need is to define the camera, by defining the camera we are concerned with 3 main parameters, where the camera, where is it looking at and which orientation is it set.

The following code snippet shows how to set the camera in open

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(
                //cam pos
                30, 40, 20,
                //looking at
                0, 0, 0,
                //up dir
                0, 1, 0);
```

The command **gluLookAt** creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an *UP* vector.

The matrix maps the reference point to the negative *z* axis and the eye point to the origin. When a typical projection matrix is used, the center of the scene therefore maps to the center of the viewport. Similarly, the direction described by the *UP* vector projected onto the viewing plane is mapped to the positive *y* axis so that it points upward in the viewport. The *UP* vector must not be parallel to the line of sight from the eye point to the reference point.

2. DRAWING THE CUBE

Now after setting the 3D environment we are ready to draw the cube; a cube is drawn using 6 squares here we will show you how to utilize the transformations you learned the previous section to draw the cube.

```
void DrawCube()
{
            //+z square
            glPushMatrix();
            {
                glColor3f(1, 0, 0);
                glTranslatef(0, 0, .5f);
                DrawSquare();
            }
            glPopMatrix();
            //-z square
            glPushMatrix();
            {
                glColor3f(0, 1, 0);
                glTranslatef(0, 0, -.5f);
                DrawSquare();
            }
            glPopMatrix();
            //+x square
            glPushMatrix();
            {
                glColor3f(1, .5f, 0);
                glTranslatef(.5f, 0, 0);
                glRotatef(90, 0, 1, 0);
                DrawSquare();
            }
            glPopMatrix();
            //-x square
            glPushMatrix();
            {
                glColor3f(0, 1, .5f);
                glTranslatef(-.5f, 0, 0);
                glRotatef(90, 0, 1, 0);
                DrawSquare();
            }
            glPopMatrix();
            //+y square
            glPushMatrix();
            {
                glColor3f(.5f, 0, 1);
                glTranslatef(0, .5f, 0);
                glRotatef(90, 1, 0, 0);
                DrawSquare();
            }
```

```
            glPopMatrix();
            //-y square
            glPushMatrix();
            {
                glColor3f(1, 0, .5f);
                glTranslatef(0, -.5f, 0);
                glRotatef(90, 1, 0, 0);
                DrawSquare();
            }
            glPopMatrix();
        }

        /// <summary>
        /// Draws a 1x1 square with center at (0, 0).
        /// </summary>
        public void DrawSquare()
        {
            glBegin(GL_QUADS);
            glVertex2f(-.5f, -.5f);
            glVertex2f(.5f, -.5f);
            glVertex2f(.5f, .5f);
            glVertex2f(-.5f, .5f);
            glEnd();
        }
```

Finally the display funciton should looks like this:

```
void OnDisplay()
{
      glClearColor(0, 0, 0, 1);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
      {
        glScalef(10, 10, 10);
        // Inside OnPaint
           glRotatef(fRotAroundZ, 0, 1, 0);
        DrawCube();

        //glutWireTeapot(2);//size = 2
    }
    glPopMatrix();

    glutSwapBuffers();
}
```

3. USE IDLE EVENT TO HANDLE ANIMATION

Using the idle event illustrated in lab 2 we can create animation. To check the animation you should try to rotate the cube using the 'Q' and 'E' it will rotate the cube around the Y-axes.

Finally, add the rotation transform before the cube in the rendering method.

```
// Inside OnPaint
                glRotatef(angle, 0, 1, 0);
                DrawCube();
```

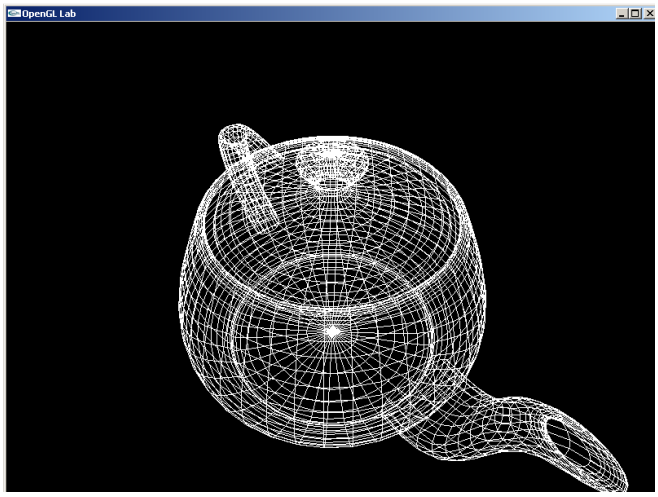
4. DRAWING PREDEFINED OPENGL OBJECTS

To draw any of the predefined shapes in the glut library we need first to initialize the glut library by making a call to the **glutInit()** command inside the `InitGraphics` which is done before.

Now, we can draw any of the shapes inside our rendering method (OnDisplay function), the following code snippet shows how to draw a wired Teapot.

```
// Inside OnPaint
glutWireTeapot(2);    //size = 2
```

The following screenshot was taken for a wired teapot of size 2.



All shapes are available in pairs wired and solid, some of the shapes available are listed below.

Cone, Cube, Sphere, Torus,Teapot and many others.