

A simple program

If you are able to run this program successfully, then you have installed OpenGL on your computer successfully. This is a simple program which displays a triangle in 2D. Try running this program and play with this a bit before jumping to next program.

```
#include "stdafx.h"
#include <GL/glut.h>

void display(void)
{
    /* clear all pixels */
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f(1.0f, 1.0f, 1.0f); //Defining color (white)
    glBegin(GL_LINE_LOOP);
        glVertex3f(5.0f, 5.0f, 0.0f);
        glVertex3f(25.0f, 5.0f, 0.0f);
        glVertex3f(25.0f, 25.0f, 0.0f);
    glEnd();

    glFlush (); // Try commenting glFlush()
}

void init (void)
{
    /* select clearing color */
    glClearColor (0.5, 0.5, 0.5, 0.0);

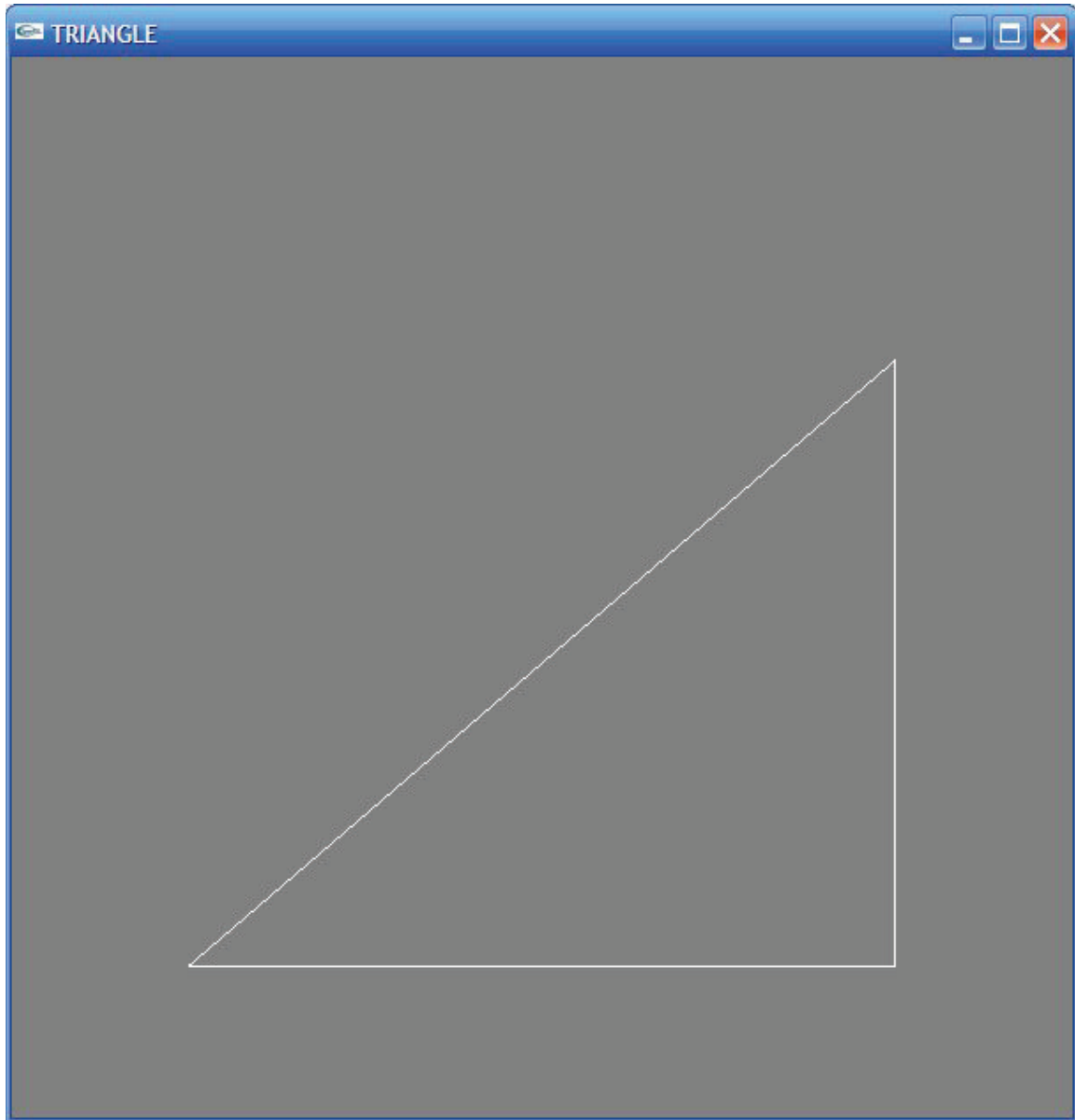
    /* initialize viewing values */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 30.0, 0.0, 35.0, -1.0, 1.0);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (600, 600);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("TRIANGLE");
    init ();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0; /* ANSI C requires main to return an int. */
}
```

simpleTriangle.cpp

We have used glBegin()/glEnd() in the above program, but most of the OpenGL implementations avoid using this paradigm. This is because of performance issues.

Output



Adding colors (Colored triangle)

Code assumptions

This is a simple 2D program which uses colors. The primitive type used for drawing a polygon is `GL_TRIANGLE`. We are making use of the three primitive colors red, green and blue. Each color is projected from each vertex of the triangle.

```
#include "stdafx.h"
#include <GL/glut.h>

void display(void)
{
    /* clear all pixels */
    glClear (GL_COLOR_BUFFER_BIT);

    /* draws a colorful triangle */

    glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 0.0f, 0.0f); // red
    glVertex3f(5.0f, 5.0f, 0.0f);
    glColor3f(0.0f, 1.0f, 0.0f); // green
    glVertex3f(25.0f, 5.0f, 0.0f);
    glColor3f(0.0f, 0.0f, 1.0f); // blue
    glVertex3f(25.0f, 25.0f, 0.0f);
    glEnd();

    glFlush ();
}

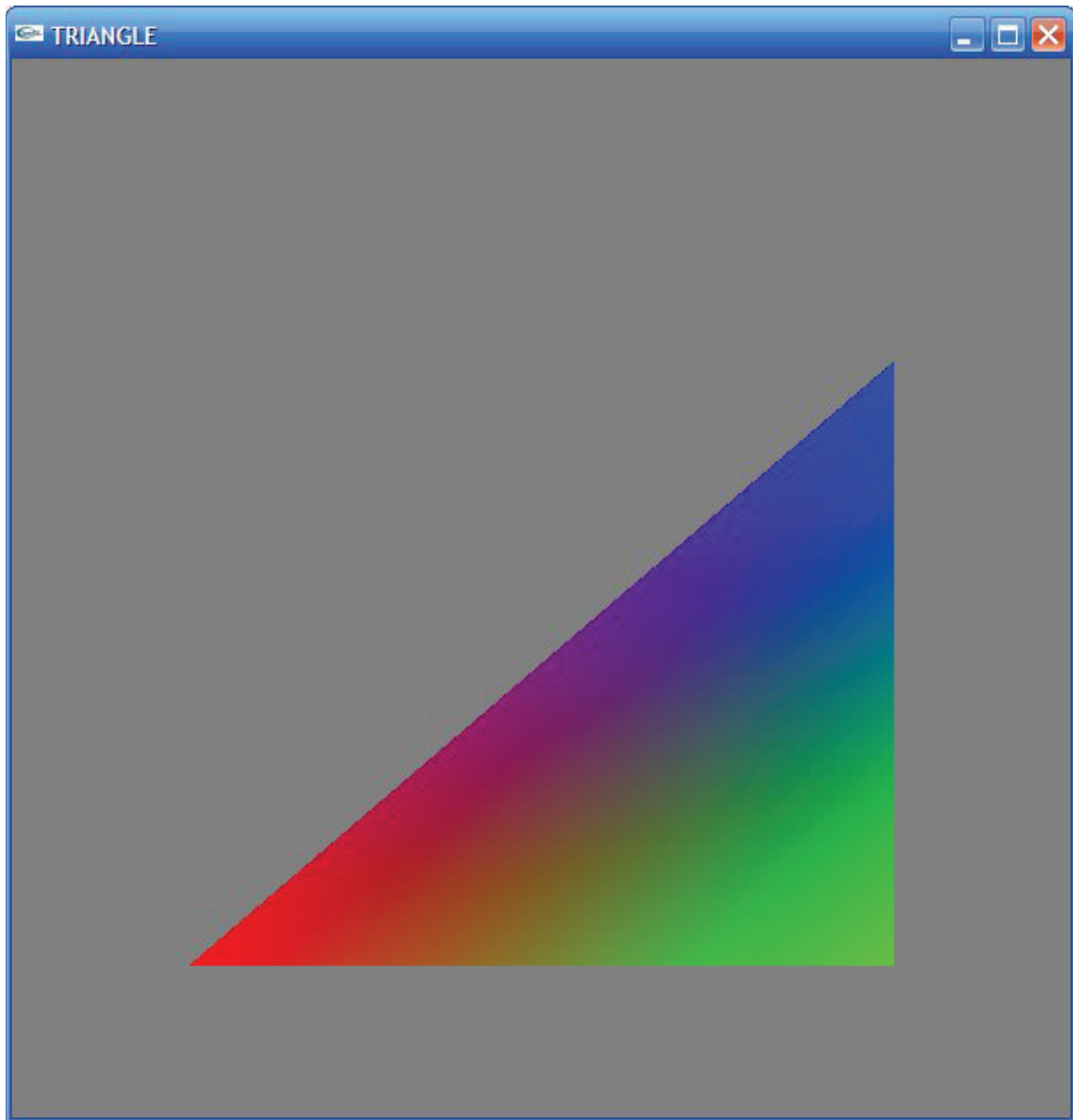
void init (void)
{
    glClearColor (0.5, 0.5, 0.5, 0.0);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 30.0, 0.0, 35.0, -1.0, 1.0);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (600, 600);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("TRIANGLE");
    init ();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0; /* ANSI C requires main to return int. */
}
```

Triangle.cpp

Output



Z Buffer

Code assumptions

In this program, we basically toggle between enabling/disabling *z-buffer*. We enable *z-buffer* or depth buffer using the following command, *glEnable(GL_DEPTH_TEST)*.

```
#include "stdafx.h"
#include <GL/glut.h>

int change = 0;

void display(void)
{
    /* clear all pixels */
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    if(change)
    {
        glEnable(GL_DEPTH_TEST); //enabling z-buffer
    }
    else
    {
        glDisable(GL_DEPTH_TEST); //Disabling z-buffer
    }

    glColor3f(1.0f, 1.0f, 0.5f);

    glutSolidTeapot (0.80);

    glColor3f(1.0f, 0.0f, 0.5f);

    glutWireTorus (0.5, 1.0, 20, 30);

    glFlush ();
    glutSwapBuffers();
}

void init (void)
{
    /* select clearing color */
    glClearColor (0.5, 0.5, 0.5, 0.0);

    /* initialize viewing values */
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(25.0, 1.0, 1.0, 100.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (5.0, 5.0, 15.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
```

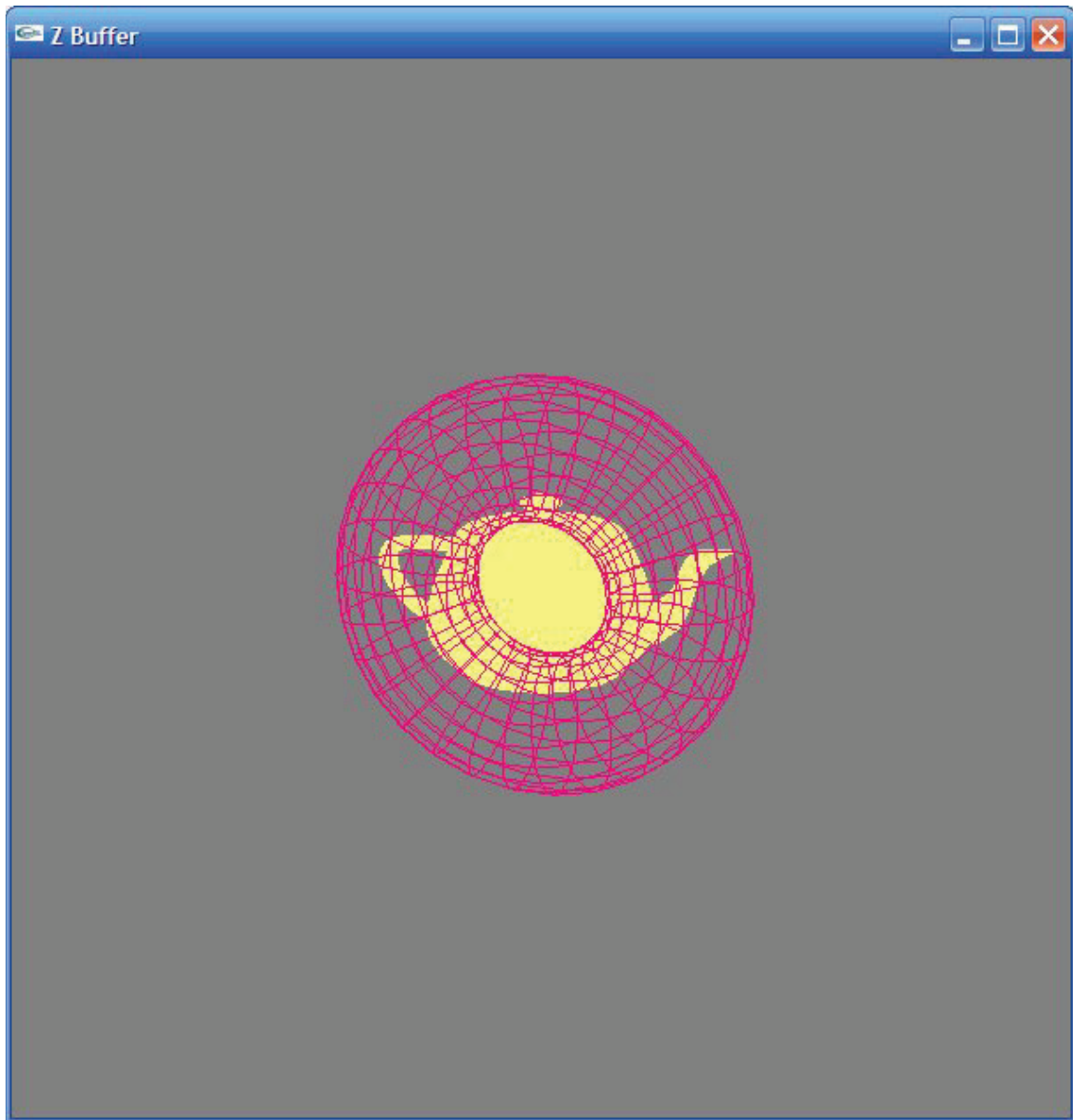
```

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 27:                // "esc" on keyboard
            exit(0);
            break;
        case 97:                // "a" on keyboard
            change = 1 - change;
            glutPostRedisplay();
            break;
    }
}

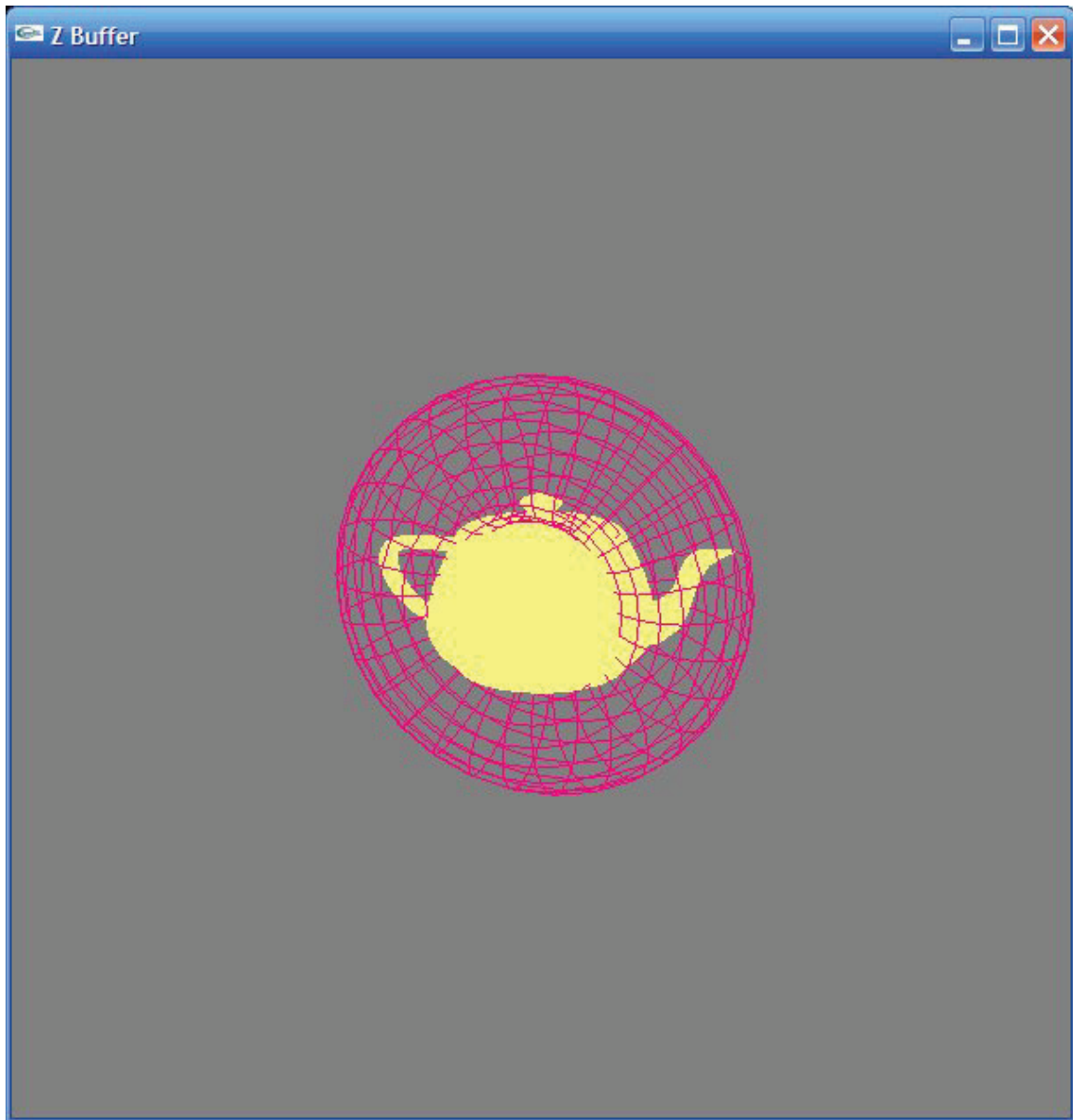
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize (600, 600);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("Z Buffer");
    init ();
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```

Output



Output



Pre-defined 3D objects

Code assumptions

This program makes use of some of the pre-defined 3D objects defined by OpenGL. We are using a *wire model* in order to draw these objects. An alternative model would be the *solid* model.

Some of the predefined 3-D shapes in OpenGL

```
glutWireCone (base length, height, slices, stacks);
glutWireSphere (radius, slices, stacks);
glutWireTorus (inner radius, outter radius, slices, stacks);

#include "stdafx.h"
#include <GL/glut.h>

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1.0f, 1.0f, 0.5f); //Defining color

    glutWireTorus (1.0, 3.0, 20, 30); // Creates a wire torus
    //glutWireTeapot(2.0);
    //glutWireCone(2.0, 3.0, 10, 30);

    glFlush ();
}

void init (void)
{
    glClearColor (0.5, 0.5, 0.5, 0.0);

    glEnable(GL_DEPTH_TEST); //enabling z-buffer

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

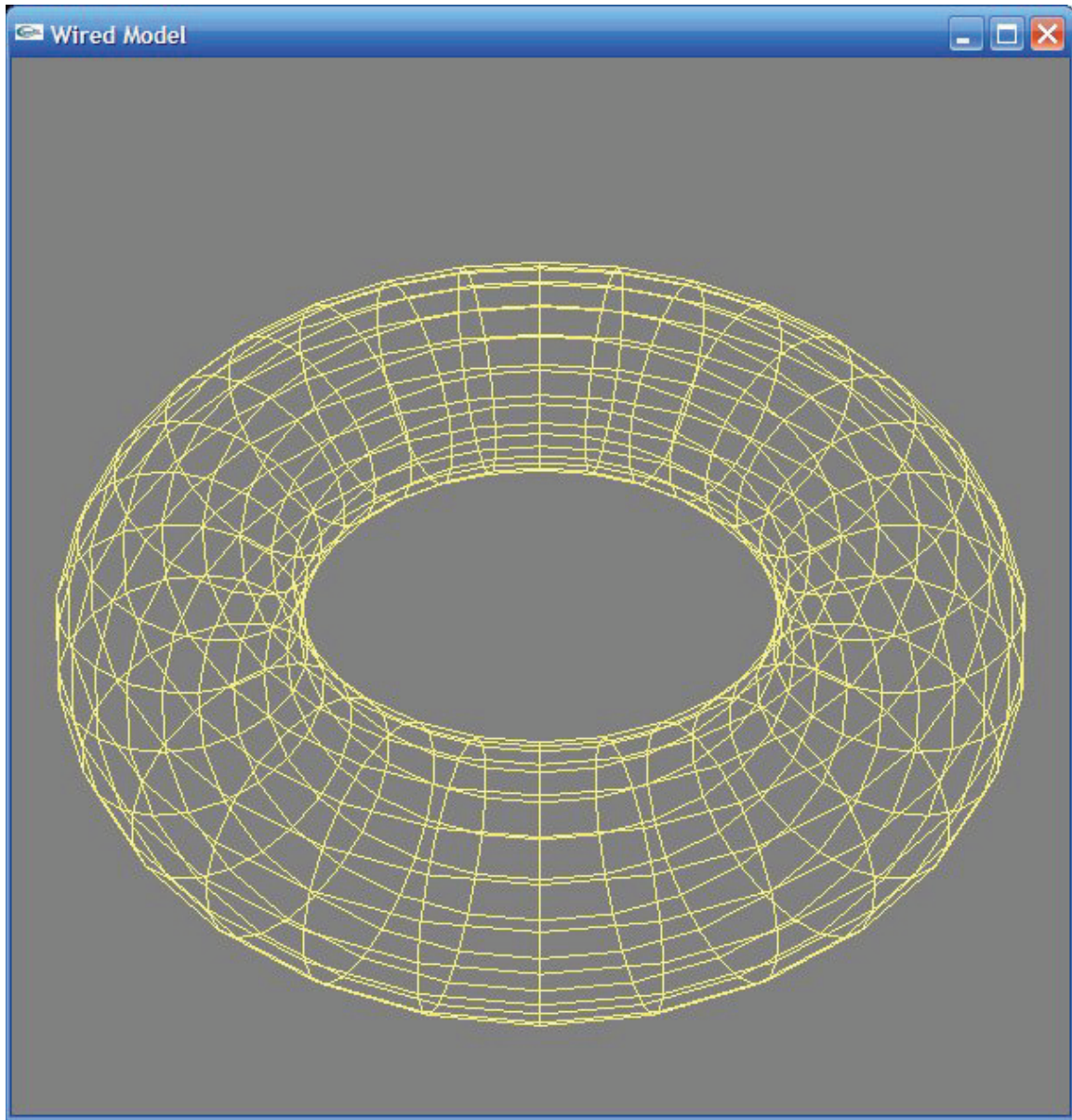
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (10.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (600, 600);
    glutInitWindowPosition (100, 100);
```

```
glutCreateWindow ("Wired Model");  
init ();  
glutDisplayFunc (display);  
glutMainLoop();  
return 0;  
}
```

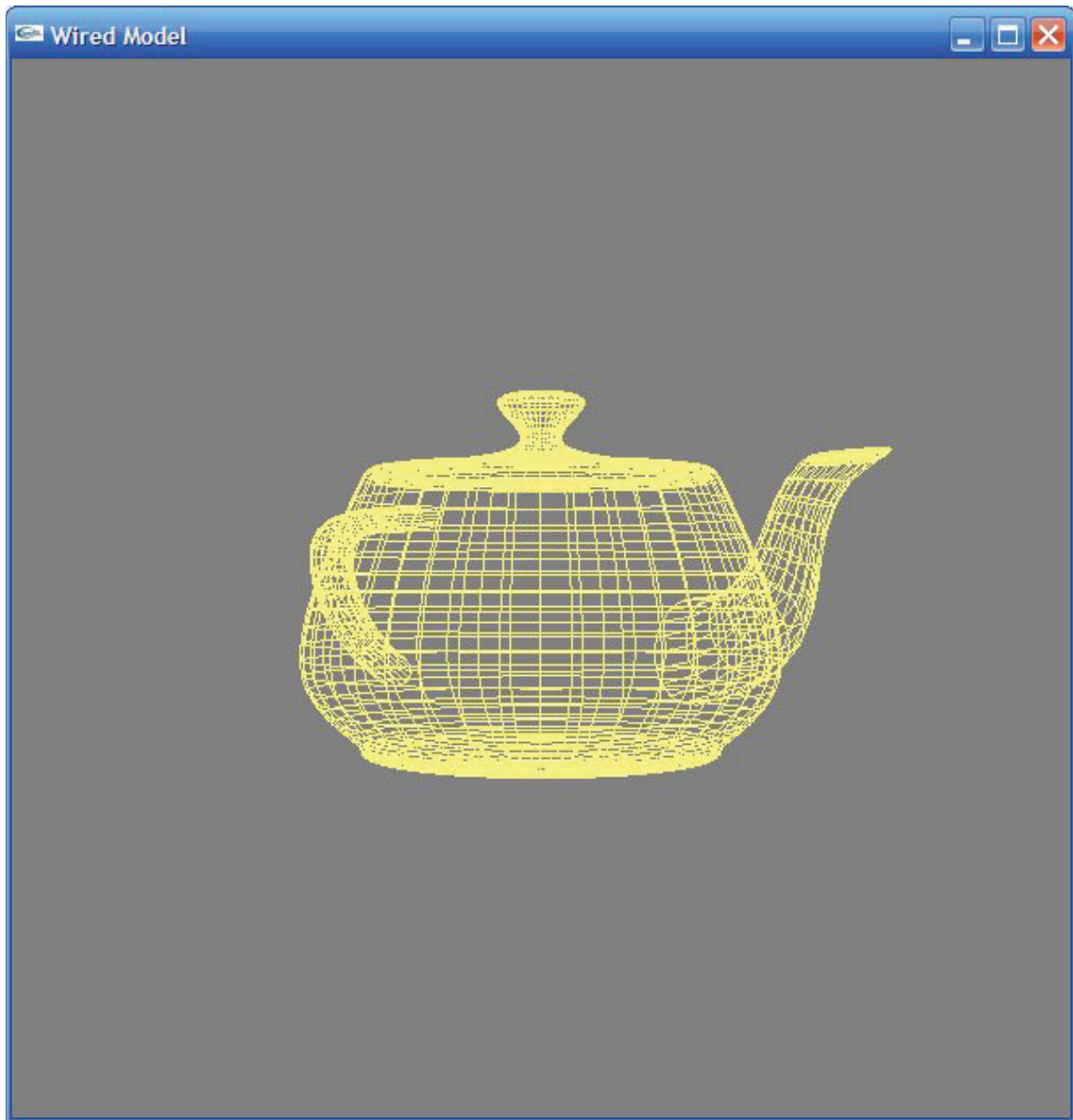
WiredModel.cpp

Output

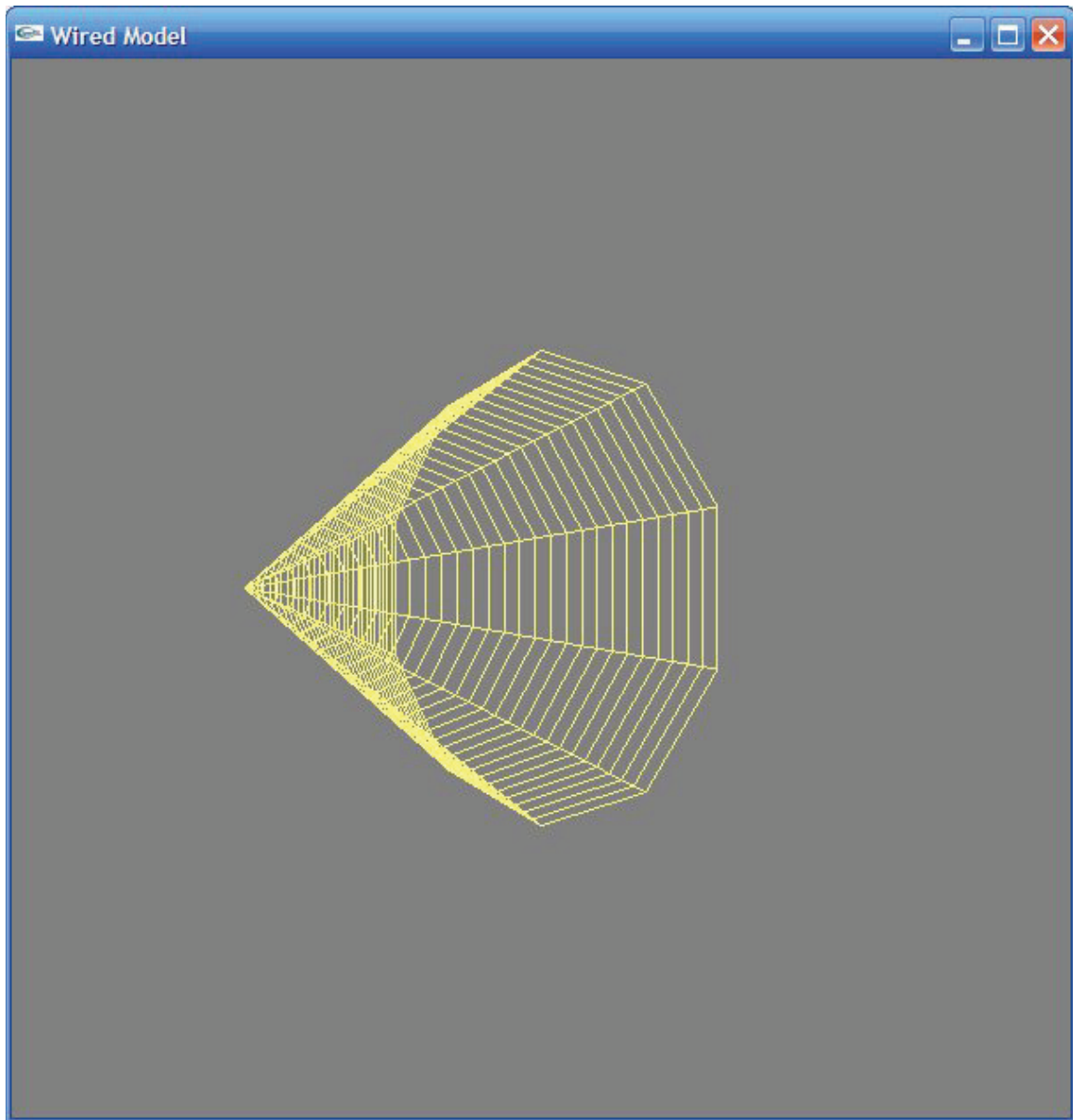


Uncomment other shapes and run the program.

Wired TeaPot



Wired Cone



Display Lists

Code assumptions

Display lists store a sequence of OpenGL commands in memory for later execution. The command *glCallList (listID)* executed the entire display list command sequence. Hence it is efficient to use display lists instead of using the same set of commands again and again. The *listID* is going to be unique for a set of commands.

```
#include "stdafx.h"
#include <GL/glut.h>

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1.0f, 1.0f, 0.5f); //Defining color

    GLuint listID;    // Creating a listID to differentiate between
                      // multiple display lists
    listID = glGenLists(1);

    glNewList( listID, GL_COMPILE );    // Definig display list starts

    glBegin( GL_QUADS );
        glColor3f( 1.0, 0.0, 0.0 );
        glVertex3f( 0.0, 0.0, 0.0 );
        glVertex3f( 0.0, 2.0, 0.0 );
        glColor3f( 0.0, 0.0, 1.0 );
        glVertex3f( 2.0, 2.0, 0.0 );
        glVertex3f( 2.0, 0.0, 0.0 );
    glEnd();

    glEndList();    // Defining Display List ends

    glCallList(listID); // Using the defined display list
    glRotatef(90.0, 1.0, 0.0, 0.0 );
    glCallList(listID); // Using the same display list after rotation
    glDeleteLists(listID, 1); // Deleting the display list if we no
                             // longer need it

    glFlush ();
    glutSwapBuffers();
}

void init (void)
{
    glClearColor (0.5, 0.5, 0.5, 0.0);

    /* initialize viewing values */
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);
}
```

```

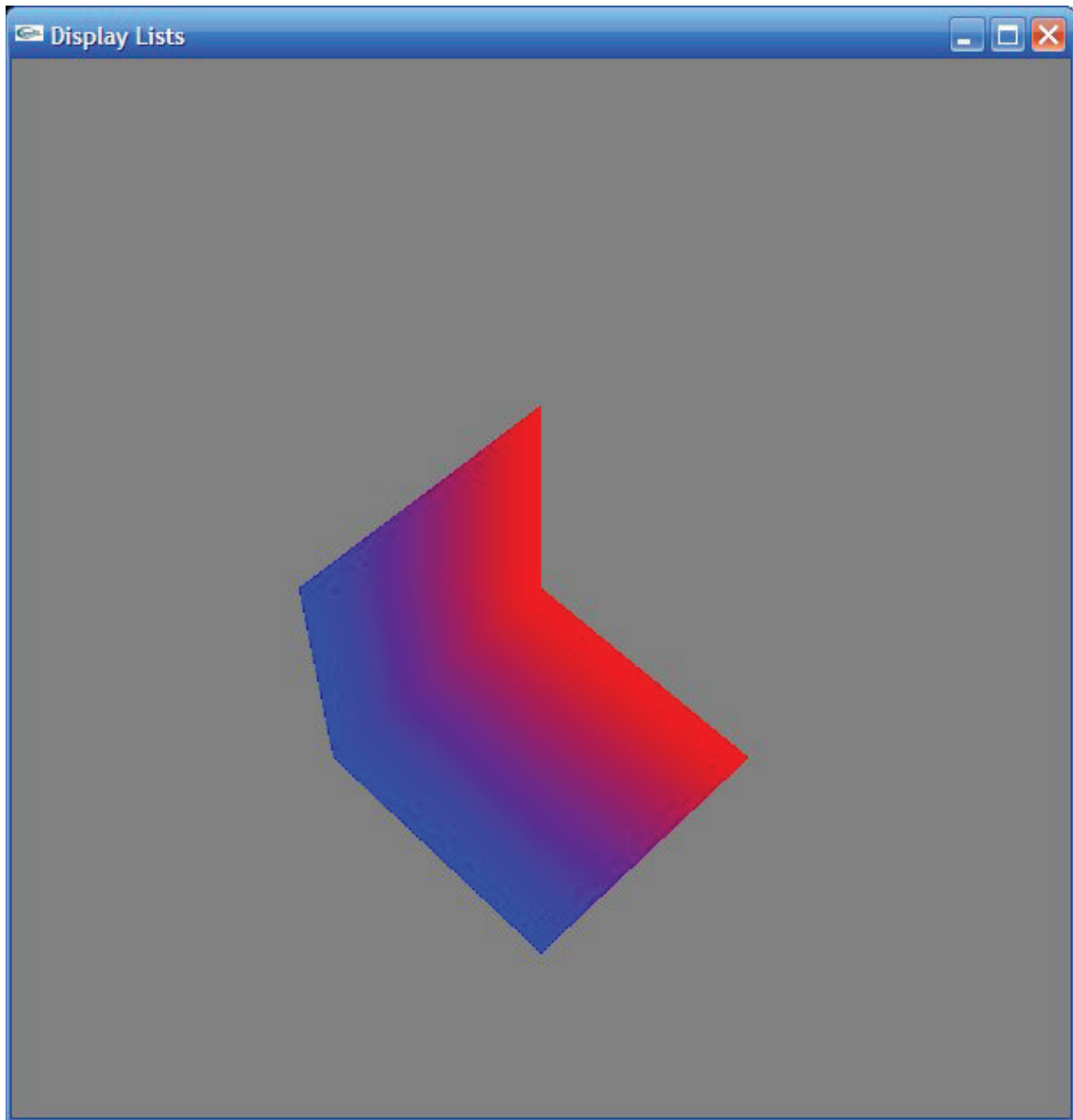
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (5.0, 5.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize (600, 600);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("Display Lists");
    init ();
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}

```

DisplayList.cpp

Output



Transformations

OpenGL implicitly converts a given set of x, y, z values into homogeneous coordinates by adding a value w . Homogeneous coordinates allow graphics systems to implement translation, scale, and rotation as matrix transformations. OpenGL uses the w value to effect perspective.

Translation

Code assumptions

The following code translates a sphere in the positive x direction. As we are calling the *glTranslatef()* function before drawing a sphere, the geometric location of the sphere depends on this transformation value. The sphere is reset after calling the transformation function two types.

The underlying mechanism is better understood if we practice the transformations using matrix model on a paper. OpenGL does all the calculations for us by just calling the *glTranslatef()* function.

```
#include "stdafx.h"
#include <GL/glut.h>

static double xVal=0.0;

void drawShape()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glColor3f(1.0f, 1.0f, 0.5f); //Defining color

    if(xVal<=2)
    {
        glTranslatef(xVal, 0.0, 0.0); // Translate by xVal in x
                                     direction
    }
    else
    {
        glTranslatef( -2 , 0.0, 0.0);
    }
    glutWireSphere(1.0, 30, 10);

    glFlush();
}

void display(void)
{
```

```

        glClearColor(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        drawShape();
        glFlush ();
    }

void init (void)
{
    glClearColor (0.5, 0.5, 0.5, 0.0);

    glEnable(GL_DEPTH_TEST); //enabling z-buffer

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (10.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
}

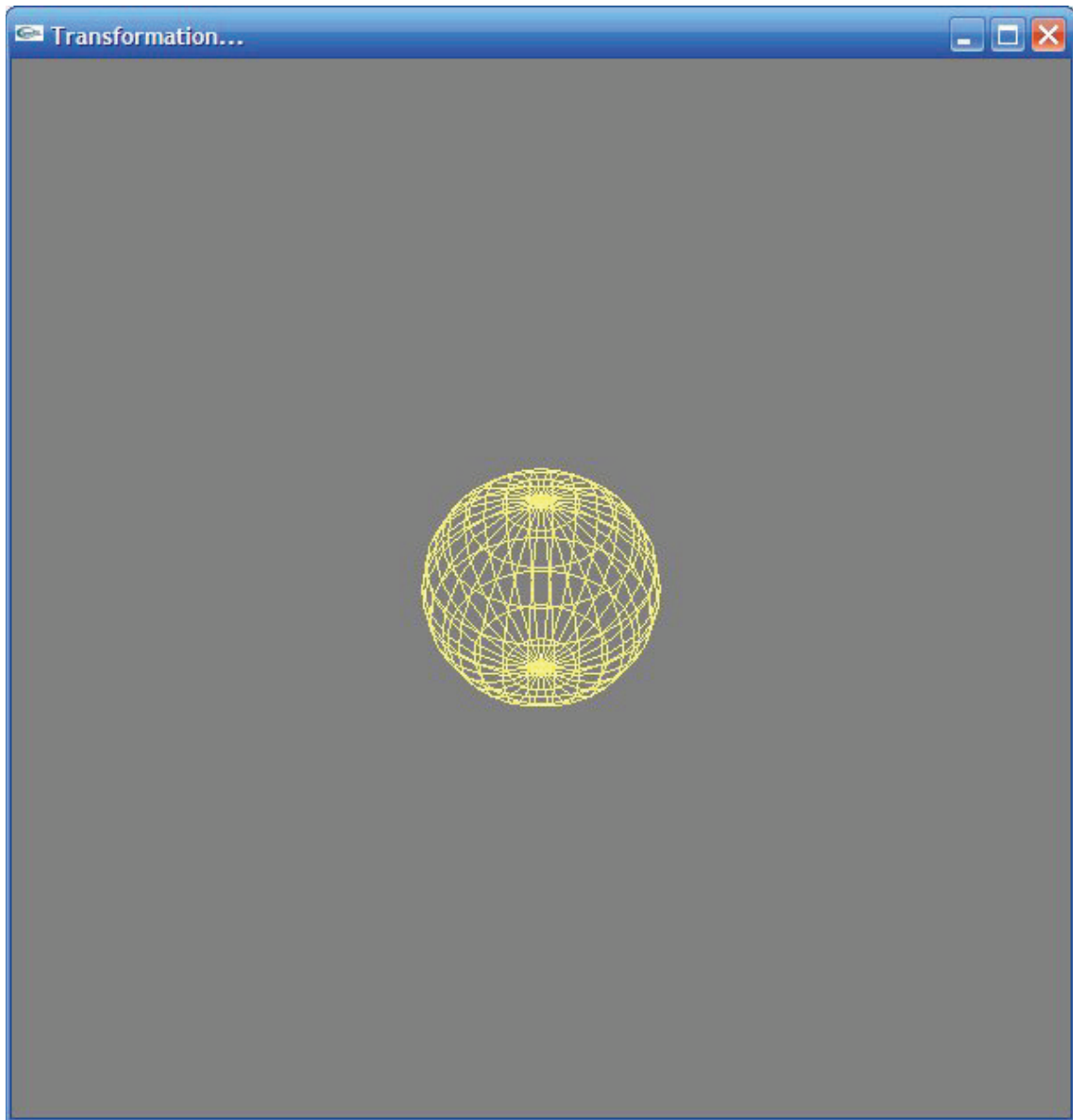
void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 27:          // "esc" on keyboard
            exit(0);
            break;
        case 97:          // "a" on keyboard
            if(xVal<=2)
            {
                xVal++;    // Incrementing xVal for translation
            }
            else
            {
                xVal=0.0;
            }
            glutPostRedisplay();
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (600, 600);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("Transformation...");
    init ();
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

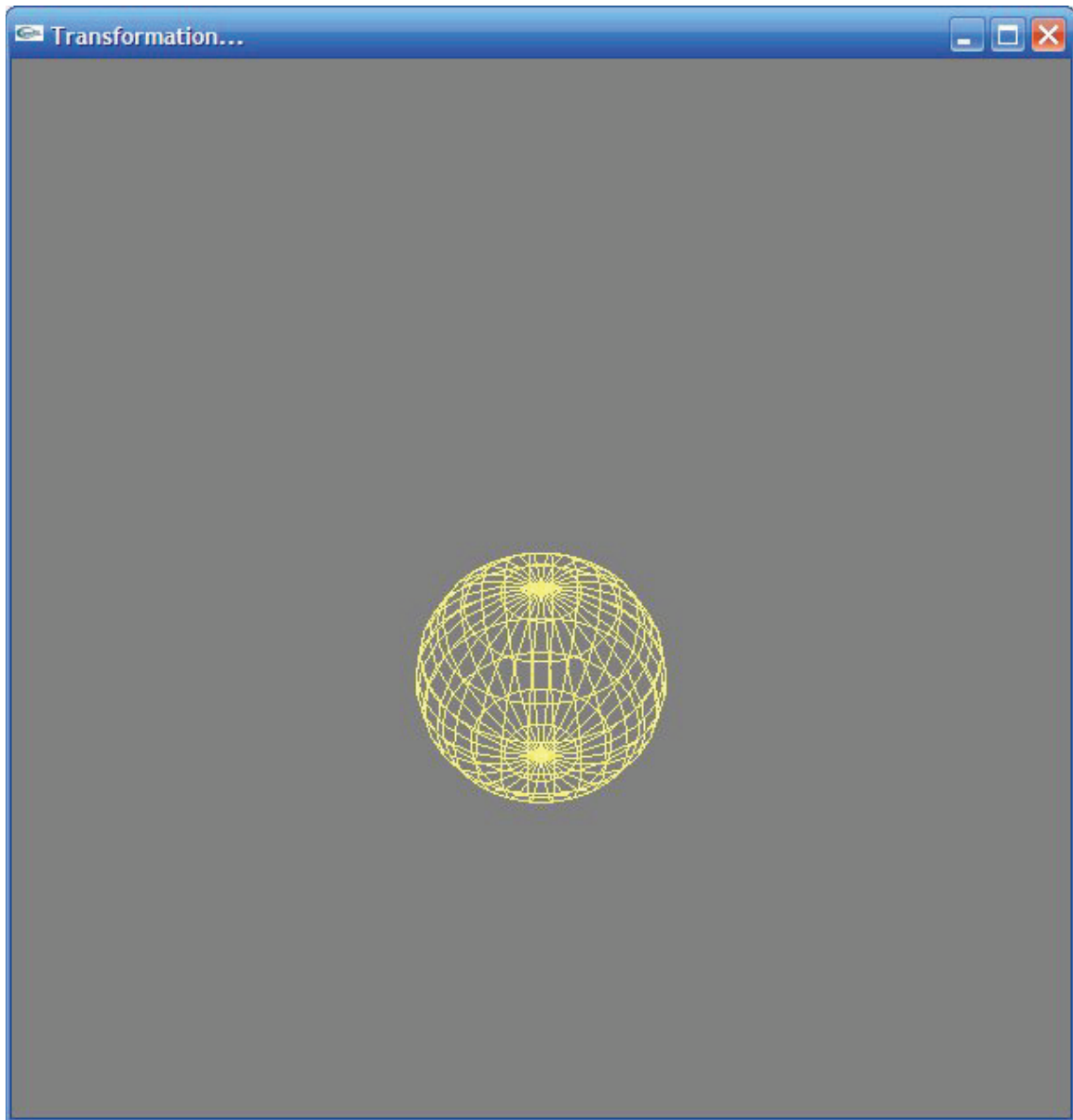
```

Translate1.cpp

Before translation



After translation



Key method used

```
void glTranslate{fd} (TYPE x, TYPE y, TYPE z);
```

Scaling

Code assumptions

The following code scales a cylinder along the y axis. As we are calling the *glScalef()* function before drawing a cylinder, changing the parameters of the function impacts the appearance of the cylinder.

The underlying mechanism is better understood if we practice scaling using the matrix model on a paper. OpenGL does all the calculations for us by just calling the *glScalef()* function. Leaving a particular parameter as 0 produces wrong results. Hence it we have to assign a value of 1 if we don't wish to scale the object in that direction.

```
#include "stdafx.h"
#include <GL/glut.h>

static double yVal=1.0;

void drawCyl()
{
    GLUQuadricObj* cyl;

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (30.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

    glColor3f(1.0f, 1.0f, 0.5f); //Defining color

    glScalef(1.0, yVal, 1.0); // Scale by yVal in y direction

    cyl = gluNewQuadric();
    gluQuadricDrawStyle(cyl, GLU_LINE);
    gluCylinder(cyl, 1.0, 1.0, 5, 35, 15);

    glFlush();
}

void display(void)
{
    /* clear all pixels */
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    drawCyl();

    glFlush ();
}
```

```

void init (void)
{
    glClearColor (0.5, 0.5, 0.5, 0.0);

    glEnable(GL_DEPTH_TEST); //enabling z-buffer

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

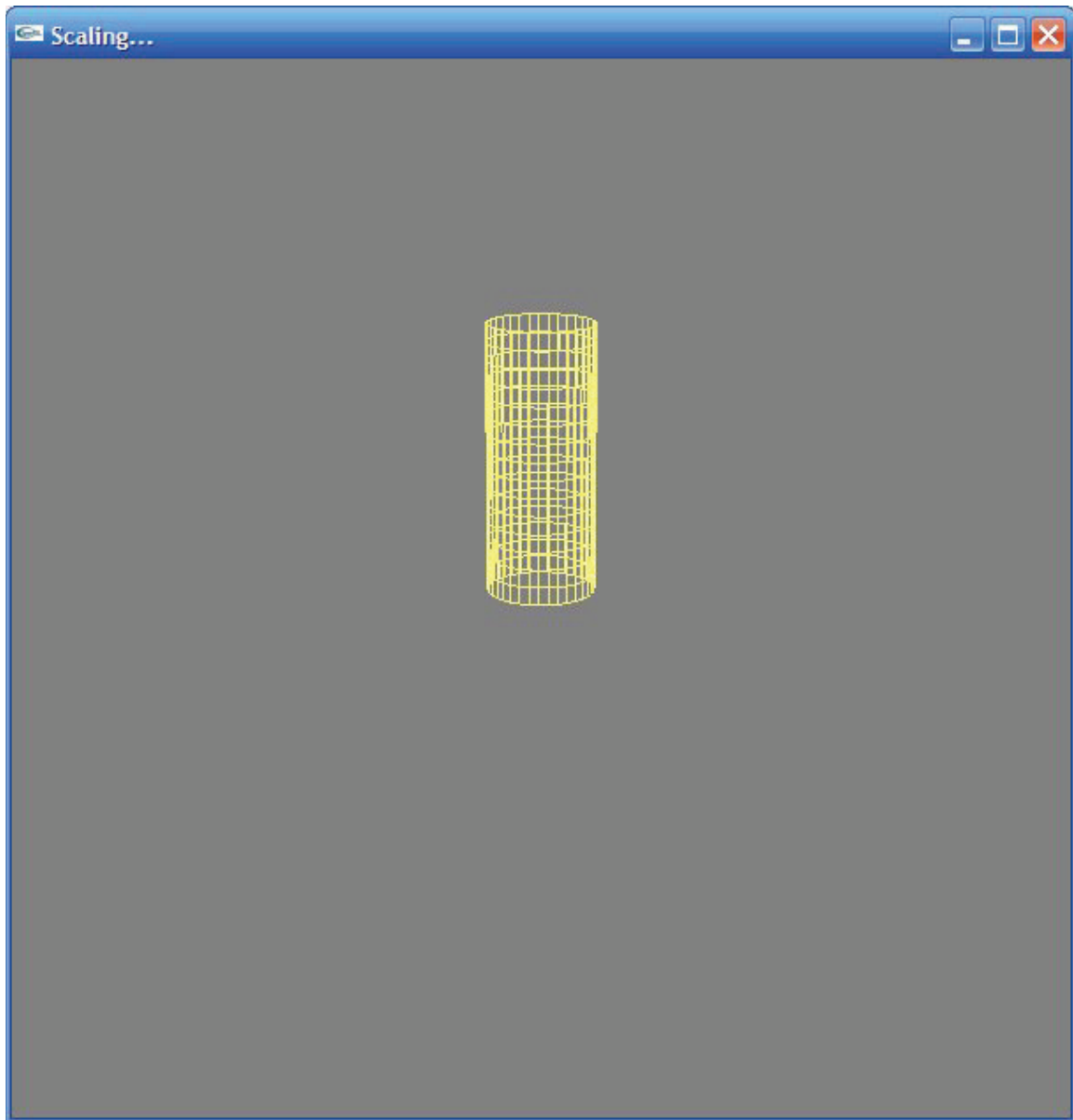
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (30.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 27:           // "esc" on keyboard
            exit(0);
            break;
        case 97:           // "a" on keyboard
            xVal++;
            glutPostRedisplay();
            break;
    }
}

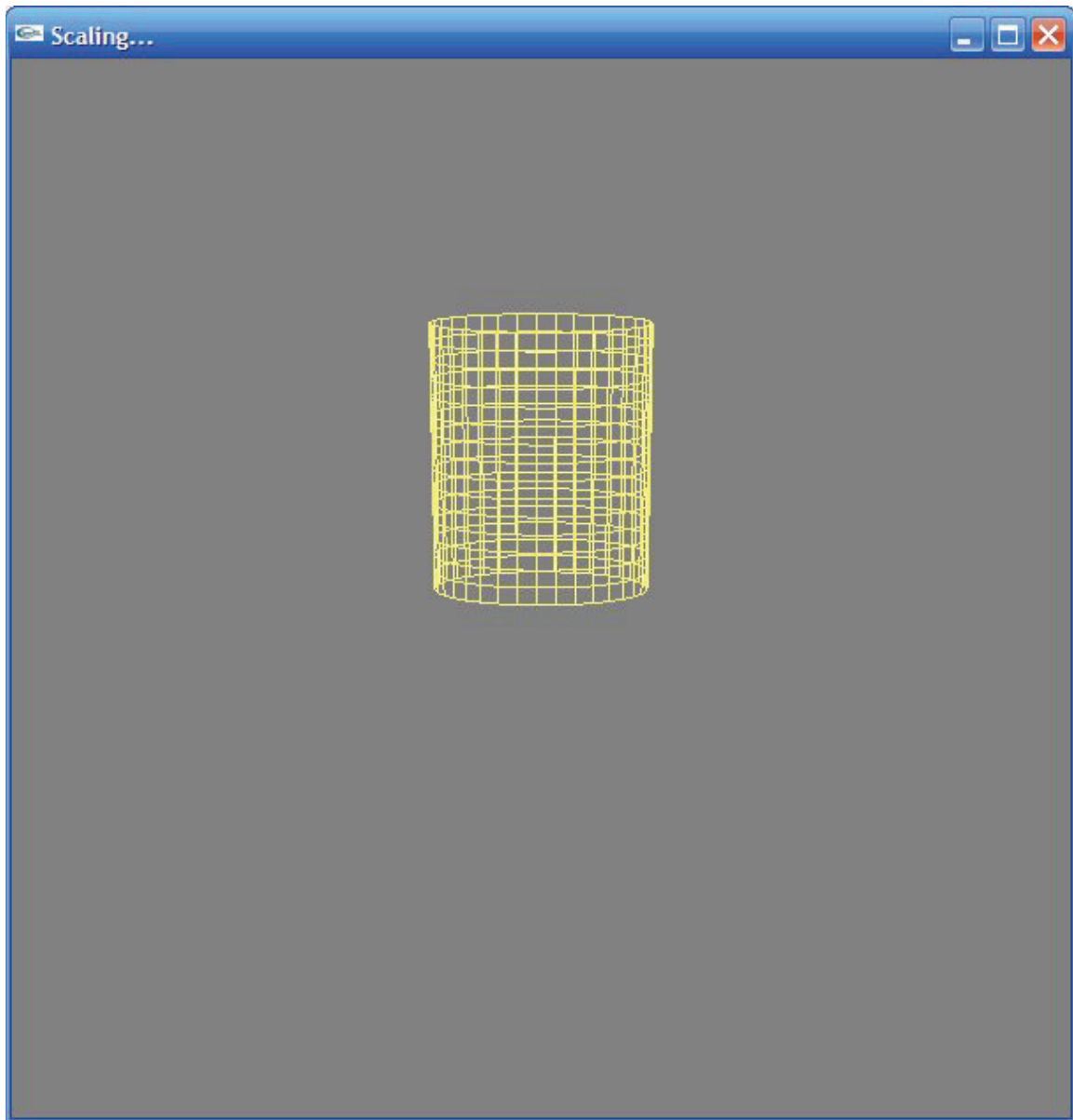
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (600, 600);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("Scaling...");
    init ();
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```

Before Scaling



After Scaling



Key method used

```
void glScale{fd} (TYPE x, TYPE y, TYPE z);
```


Rotation

Code assumptions

The following code rotates a cylinder around the x axis. As we are calling the *glRotatef()* function before drawing a cylinder, the rotation of the cylinder depends on the parameters passed to *glRotatef()*.

An additional parameter in *glRotatef()* is the *degrees*.

The underlying mechanism is better understood if we practice rotation using the matrix model on paper. OpenGL does all the calculations for us by just calling the *glRotatef()* function.

```
#include "stdafx.h"
#include <GL/glut.h>

static double deg=0.0;

void drawCyl()
{
    GLUQuadricObj* cyl;

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (30.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

    glColor3f(1.0f, 1.0f, 0.5f); //Defining color

    glRotatef(deg, 1.0, 0.0, 0.0); // Rotate by deg

    cyl = gluNewQuadric();
    gluQuadricDrawStyle(cyl, GLU_LINE);
    gluCylinder(cyl, 1.0, 1.0, 5, 35, 15);

    glFlush();
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    drawCyl();

    glFlush ();
}
```

```

void init (void)
{
    glClearColor (0.5, 0.5, 0.5, 0.0);

    glEnable(GL_DEPTH_TEST); //enabling z-buffer

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

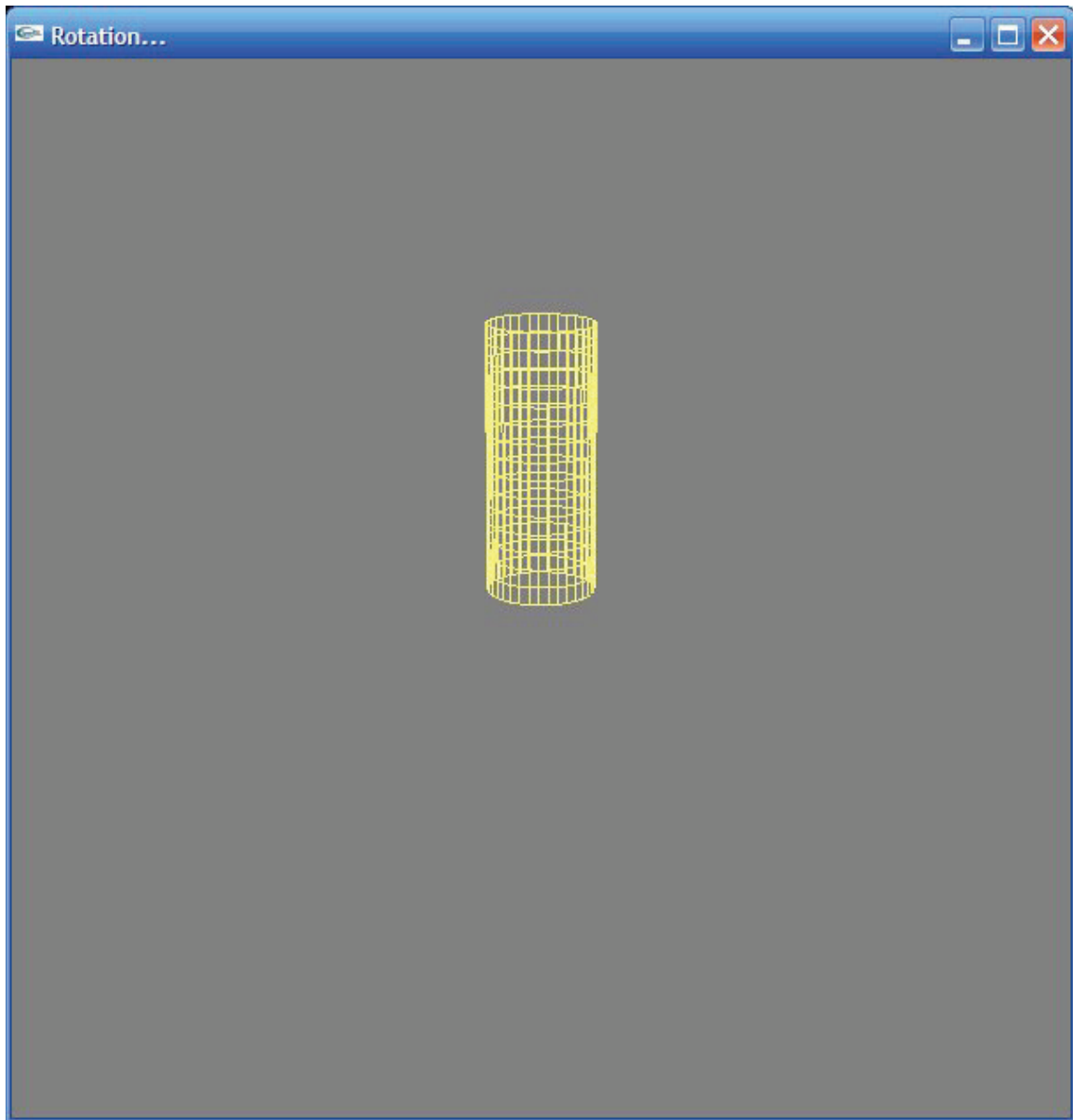
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (30.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 27:           // "esc" on keyboard
            exit(0);
            break;
        case 97:           // "a" on keyboard
            deg = deg+25.0;
            glutPostRedisplay();
            break;
    }
}

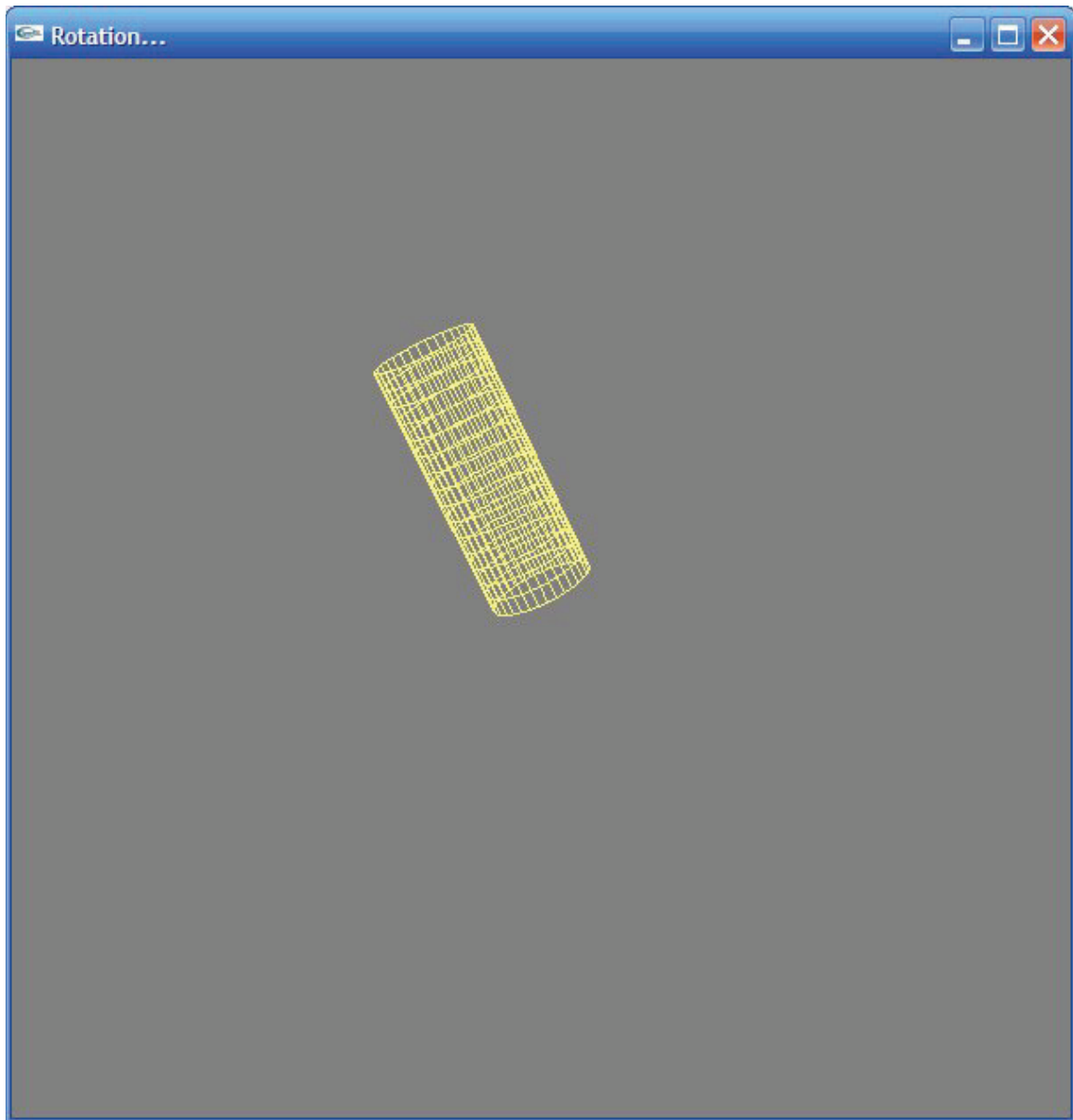
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (600, 600);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("Rotation...");
    init ();
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```

Before Rotation



After Rotation



Key method used

```
void glRotate{fd} (TYPE angle, TYPE x, TYPE y, TYPE z);
```

Lighting

Simple light (no material effect)

Code assumptions

The following code produces a simple light which in turn is projected onto a sphere. As this is the first lighting example, we are not concentrating on the material effects which are eventually described in a later section.

OpenGL implementations are required to have at least 8 primary lights ranging from *GL_LIGHT0* to *GL_LIGHT7*. In order to use lighting in the code, we have to first enable lighting by calling the routine *glEnable(GL_LIGHTING)*.

A *glLookAt()* routine sets the camera position and we have can set the position of the lighting using *glLightfv()*. This routine actually sets the direction of the light, but not the actual position of the light.

Though we are able to set a position for the light source, the light source behaves as if it is at an infinite distance.

```
#include "stdafx.h"
#include <GL/glut.h>

static double yVal=50.0;

void drawSphere()
{
    GLUQuadricObj* cyl;
    GLfloat light_position[] = { 0.0, 20.0, yVal, 10.0 };
    glLightfv(GL_LIGHT0, GL_POSITION, light_position); // Using
                                                         lighting (effects all the objects drawn below)

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (30.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

    cyl = gluNewQuadric();
    gluQuadricDrawStyle(cyl, GLU_FILL);
    gluSphere(cyl, 2.0, 50, 100);

    glFlush();
}
```

```

void display(void)
{
    /* clear all pixels */
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    drawSphere();

    glFlush ();
}

void init (void)
{
    glClearColor (0.5, 0.5, 0.5, 0.0);

    glEnable(GL_DEPTH_TEST); //enabling z-buffer

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (30.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

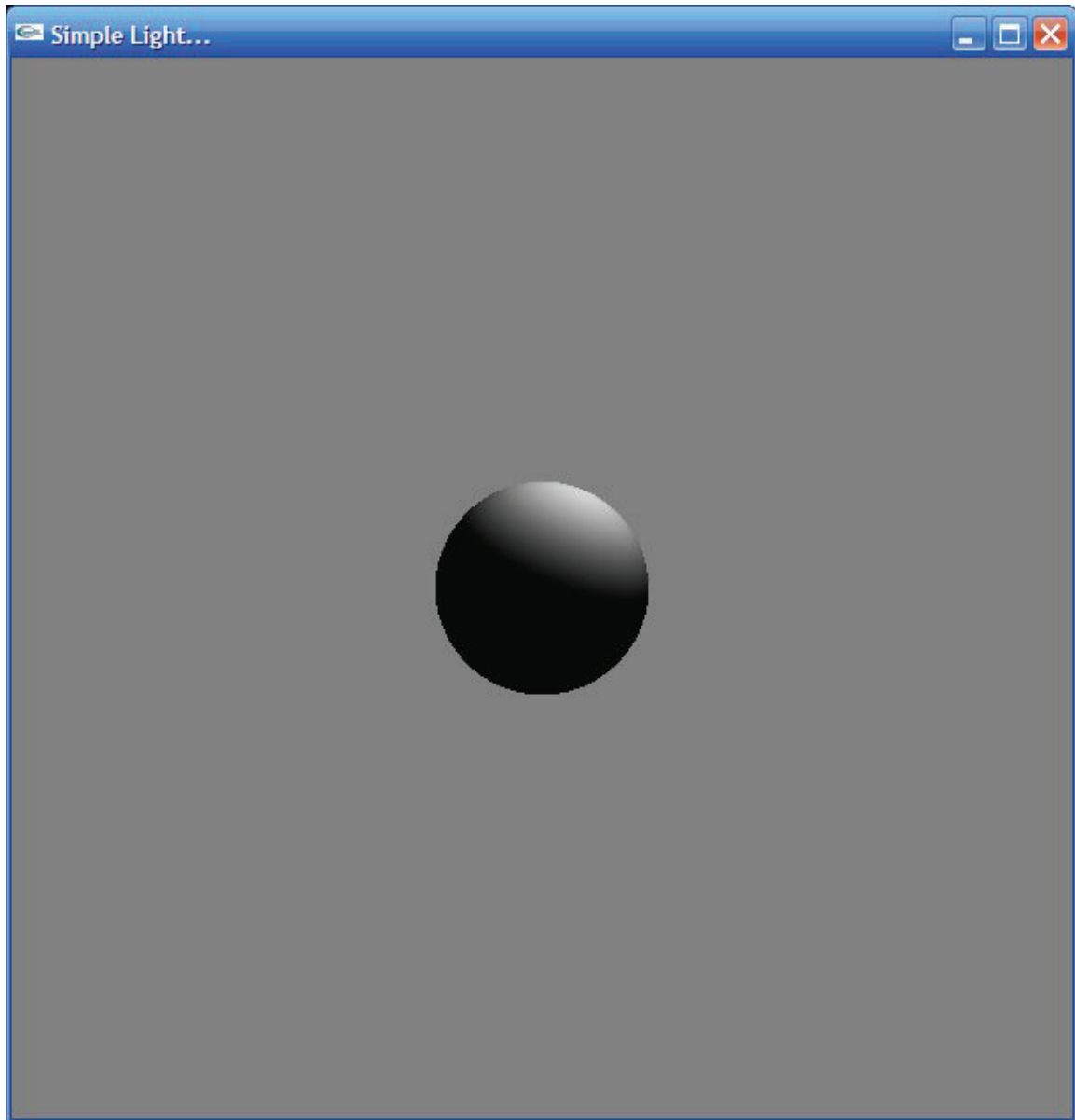
    glEnable(GL_LIGHTING); // Enable lighting
    glEnable(GL_LIGHT0); // Enable the first light (LIGHT0)
}

void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 27: // "esc" on keyboard
            exit(0);
            break;
        case 97: // "a" on keyboard
            yVal = yVal-5.0;
            glutPostRedisplay();
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (600, 600);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("Simple Light...");
    init ();
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```

Output



Key method used

void **glLightfv**(GLenum *light*, GLenum *pname*, const GLfloat **params*)

Lighting (material effect)

Material

Material parameters specify how a surface reflects light. Applications change OpenGL material parameters to emulate different colored materials, shiny and flat materials, high-gloss materials such as pool ball, etc.

The routine *glColor3f()*, sets the primary color and is a good idea to call this routine as long as we are not using lighting effects. The OpenGL lighting doesn't use the primary color, instead uses the material colors.

Hence it is important to introduce materials while talking about lighting.

Code assumptions

So far we have seen a simple lighting example which has no material effects. But, in order to make the scene more attractive, we have to make use of materials. In the following code, we have used a green material.

We make use of *GL_SPECULAR*, *GL_AMBIENT*, *GL_SHININESS* depending on the scene requirements.

The same parameters are also applicable to lighting methods.

```
#include "stdafx.h"
#include <GL/glut.h>

static double yVal=50.0;

void drawSphere()
{
    GLUQuadricObj* cyl;
    GLfloat light_position[] = { 0.0, 40.0, yVal, 0.0 };
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);

    GLfloat mat_specular[] = { 0.3f, 1.0f, 0.3f, 1.0f }; // Green
                                                         color material
    GLfloat mat_shininess[] = { 70.0 }; // Defines shininess

    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular); // Using
                                                         materials
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

    glMatrixMode (GL_MODELVIEW);
```



```

    glLoadIdentity();
    gluLookAt (30.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

    cyl = gluNewQuadric();
    gluQuadricDrawStyle(cyl, GLU_FILL);
    gluSphere(cyl, 2.0, 10, 10);

    glFlush();
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    drawSphere();

    glFlush ();
}

void init (void)
{
    glClearColor (0.5, 0.5, 0.5, 0.0);

    glEnable(GL_DEPTH_TEST); //enabling z-buffer
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (30.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
}

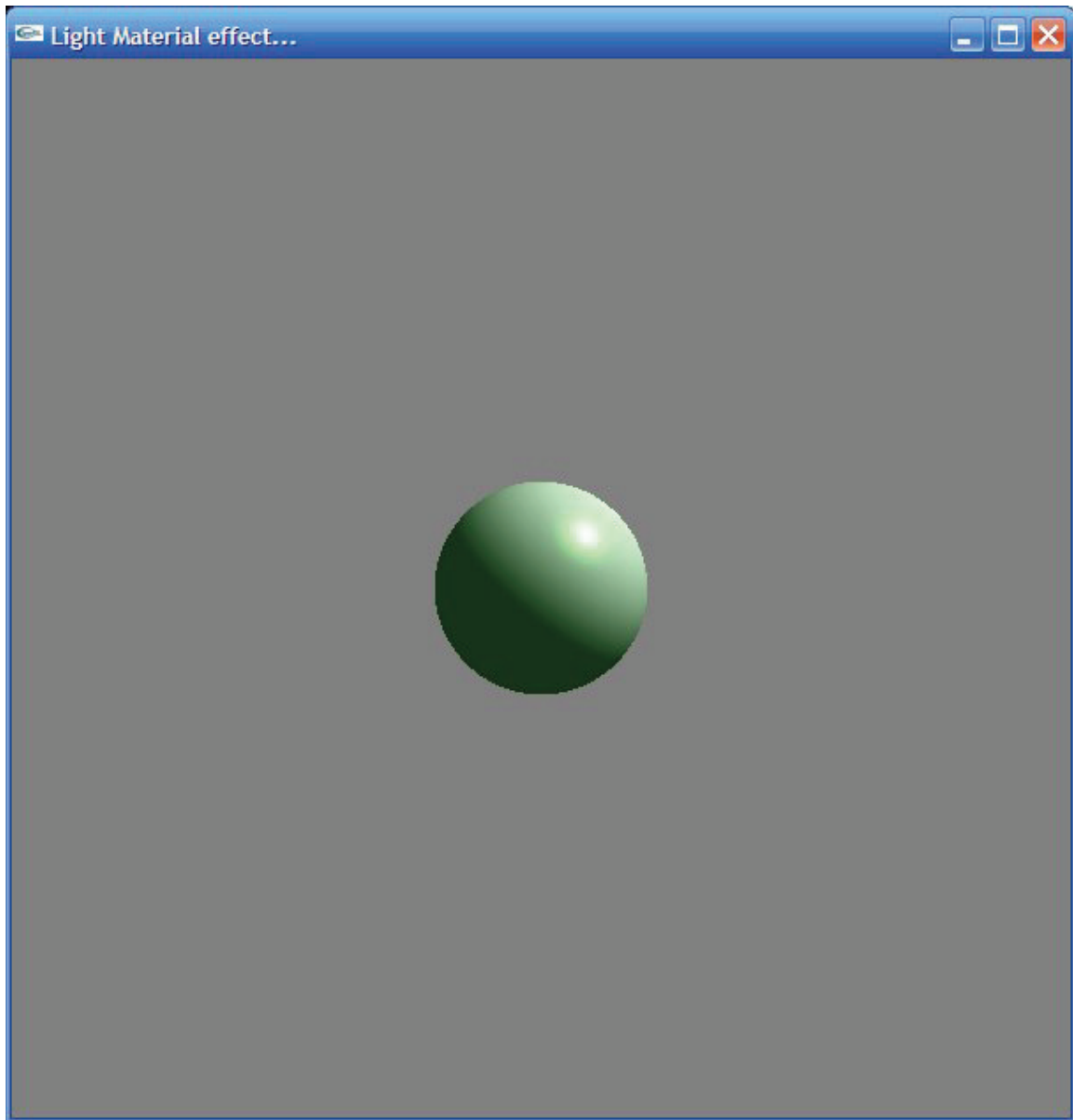
void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 27: // "esc" on keyboard
            exit(0);
            break;
        case 97: // "a" on keyboard
            yVal = yVal-5.0;
            glutPostRedisplay();
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (600, 600);

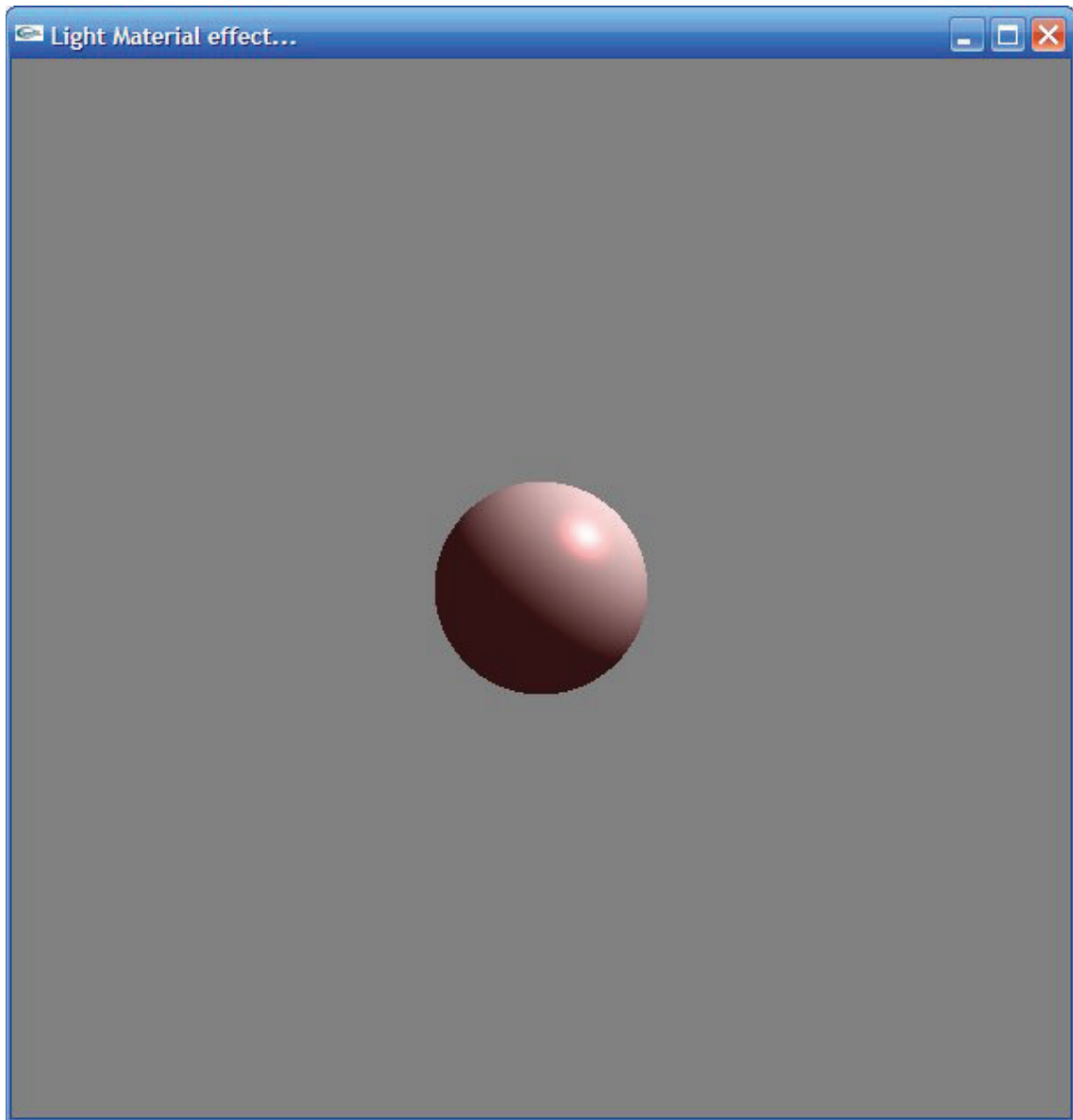
```

```
glutInitWindowPosition (100, 100);  
glutCreateWindow ("Light Material effect...");  
init ();  
glutDisplayFunc(display);  
glutKeyboardFunc(keyboard);  
glutMainLoop();  
return 0;  
}
```

Output



Different material



Spot Light

So far we have seen lighting effects which have the following properties:

- The light is being projected from an infinite distance.
- The rate of fading of the brightness at the circumference is very low.

But, we might want to produce lights, which have a position in the scene and are not projected from an infinite source unlike the normal lighting which we have seen so far. Like a study lamp or a spot light on a pool table, etc.

For a spot light, the ray emission is restricted to a cone area. Outside the cone, a spot light emits no light.

Hence in order to make the normal light behave like a spot light, we have to make some amendments to the code.

Code assumptions

The amendments made to the following code are the introduction of parameters like *GL_SPOT_CUTOFF*, which defines the angle of the cone at the center of which the spot light is placed. *GL_SPOT_EXPONENT*, defines the concentration of the light.

```
#include "stdafx.h"
#include <GL/glut.h>

static double yVal=1.0;

int spot(double a, double b, double c, double d, double e, double f)
{
    /*
    a, b and c -- x, y and z co-ordinates for light position
    d, e and f -- x, y and z co-ordinates for spot light position
    */

    GLfloat mat_specular[] = { 0.3, 1.0, 0.3, 1.0 };
    GLfloat mat_shininess[] = { 50.0 };
    GLfloat light_position[] = { a,b,c, 1.0 };
    GLfloat spotDir[] = { d,e,f };

    glClearColor (0.5, 0.5, 0.5, 0.0);

    glShadeModel (GL_SMOOTH);

    glLightfv(GL_LIGHT0, GL_SPECULAR, mat_specular);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

```

glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

// Definig spotlight attributes
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 95.0);
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT, 2.0);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spotDir);

glEnable(GL_COLOR_MATERIAL);
glEnable(GL_DEPTH_TEST);

return 0;
}

void drawSphere()
{
    GLUQuadricObj* cyl;
    glClearColor (0.5, 0.5, 0.5, 0.0);
    GLfloat light_position[] = { 50.0, 50.0 , 0.0, 1.0 };
    GLfloat mat_specular[] = { 0.3, 0.3, 1.0, 1.0 };
    GLfloat mat_shininess[] = { 100.0 };
    GLfloat spotDir[] = { 50.0, 30.0, 0.0 };

    glShadeModel (GL_SMOOTH);

    spot( yVal, 5.0, 1.5, 10.0, 1.0, 10.0);

    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);

    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (30.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

    cyl = gluNewQuadric();
    gluQuadricDrawStyle(cyl, GLU_FILL);

    glPushMatrix();
    gluSphere(cyl, 4.0, 1000, 1000);

    glPopMatrix();
    glFlush();
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    drawSphere();
}

```

```

        glFlush ();
    }

void init (void)
{
    glClearColor (0.5, 0.5, 0.5, 0.0);

    glEnable(GL_DEPTH_TEST); //enabling z-buffer

    /* initialize viewing values */
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (30.0, 0.0, 10.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

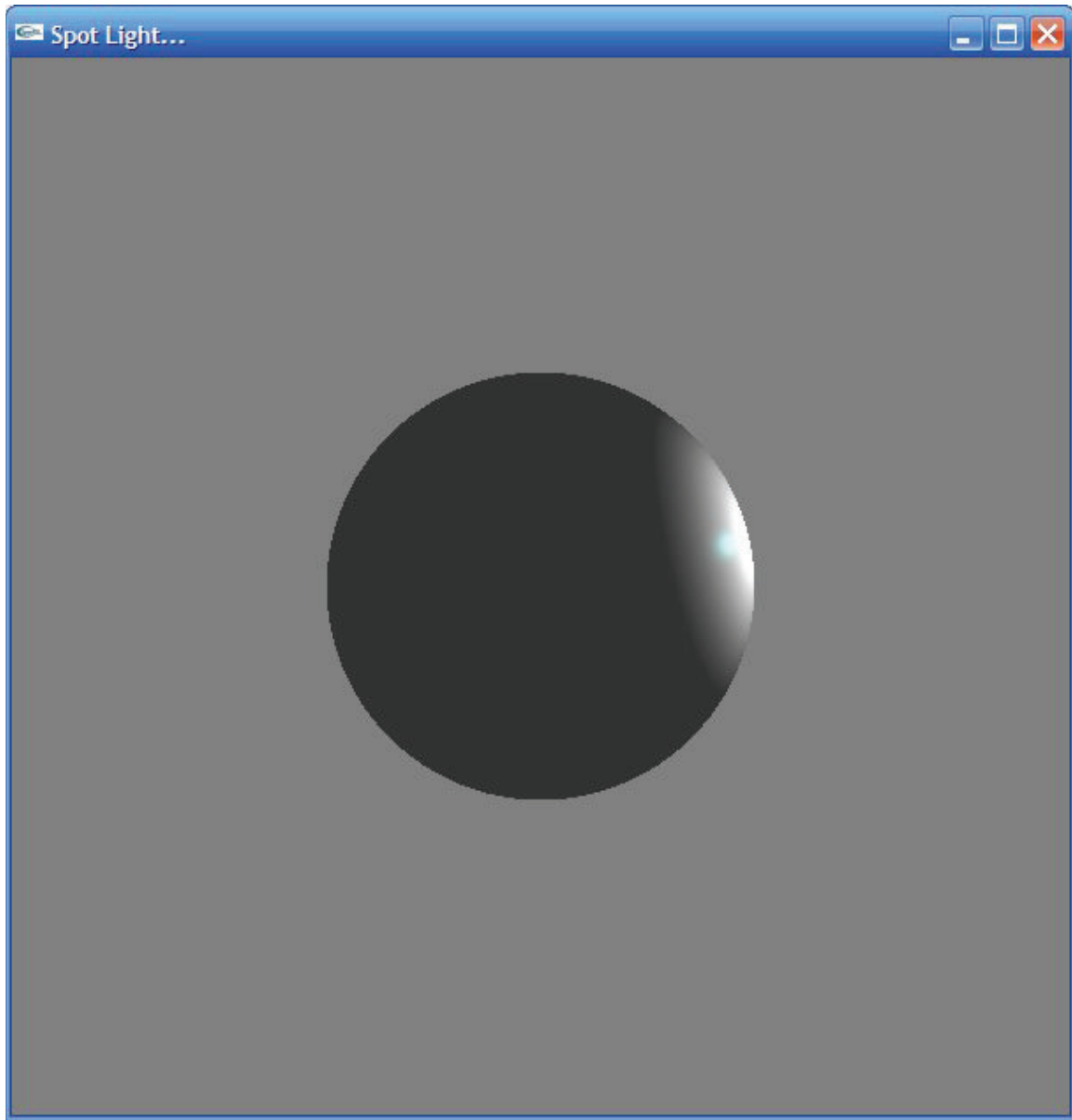
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
}

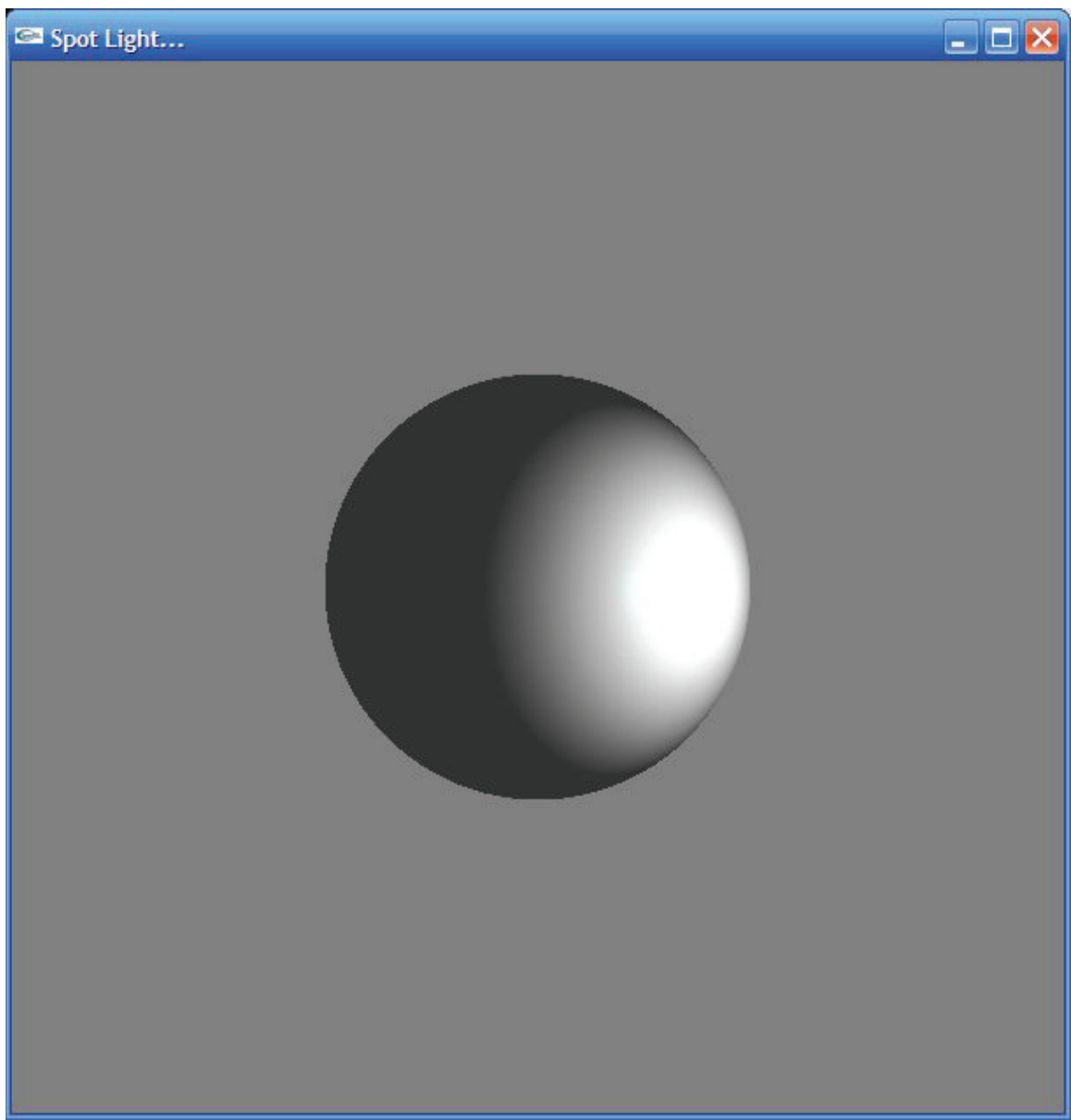
void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 27: // "esc" on keyboard
            exit(0);
            break;
        case 97: // "a" on keyboard
            yVal++;
            glutPostRedisplay();
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (600, 600);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("Spot Light...");
    init ();
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```

Output





Multiple Lights

To make the scene look much better, we have to make use of more than one light. So, all these lights have their own position in the 3D space and also have their own lighting effects.

Code assumptions

In the following code, we toggle between single light source and multiple light sources. As we are using more than one light source here, we have to enable the number of light sources we intend to use in the code.

The code enables two lights *GL_LIGHT0* and *GL_LIGHT1*. We enable/disable each light source for the toggling effect to take place.

```
#include "stdafx.h"
#include <GL/glut.h>

int change = 0;
void drawSphere()
{
    float mat_specular[] = { 0.3, 1.0, 0.3, 1.0 };
    float mat_shininess[] = { 100.0 };
    float mat_surface[] = { 1.0, 1.0, 0.0, 0.0 };

    float blue_light[] = { 0.0, 0.0, 1.0, 1.0 };
    float red_light[] = { 1.0, 0.0, 0.0, 1.0 };
    float light_position0[] = { 0.0, 40.0, 50.0, 0.0 };
    float light_position1[] = { 0.0, 40.0, -50.0, 0.0 };

    glClearColor (0.5, 0.5, 0.5, 0.0);
    glShadeModel (GL_SMOOTH);

    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_surface);

    glLightfv(GL_LIGHT0, GL_POSITION, light_position0);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, blue_light);
    glLightfv(GL_LIGHT1, GL_POSITION, light_position1);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, red_light);

    if(change)
    {
        glDisable(GL_LIGHT1);
    }
    else
    {
        glEnable(GL_LIGHT1);
    }
    glutSolidTeapot (0.80);
}
```

```

        glFlush();
    }

void display(void)
{
    /* clear all pixels */
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    drawSphere();

    glFlush ();
}

void init (void)
{
    /* select clearing color */
    glClearColor (0.5, 0.5, 0.5, 0.0);

    glEnable(GL_DEPTH_TEST); //enabling z-buffer

    /* initialize viewing values */
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(35.0, 1.0, 1.0, 100.0);

    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt (6.0, 4.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

    glEnable(GL_LIGHTING);

    //Defining two lights for multiple light effects
    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHT1);
}

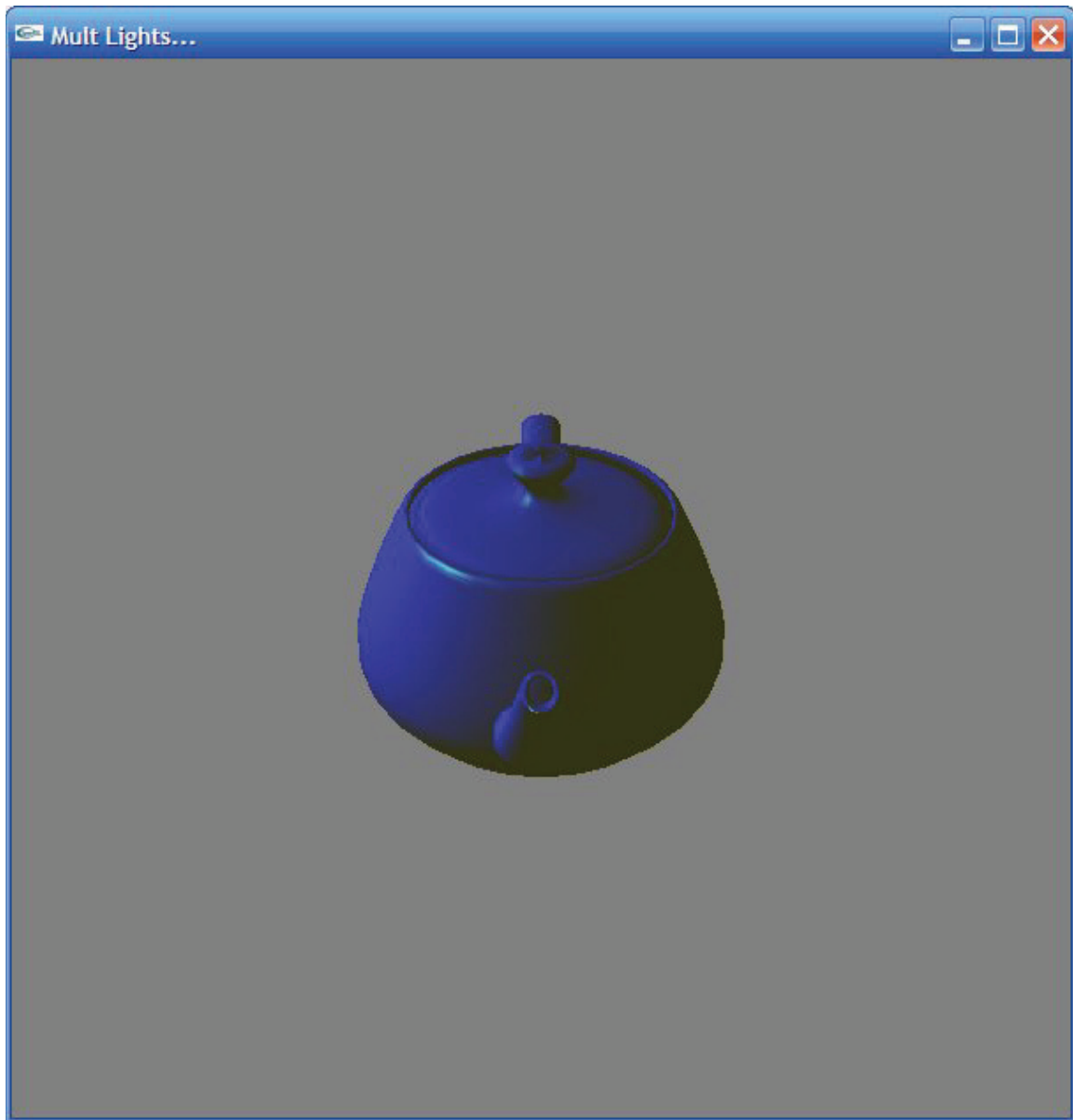
void keyboard(unsigned char key, int x, int y)
{
    switch (key) {
        case 27:                // "esc" on keyboard
            exit(0);
            break;
        case 97:                // "a" on keyboard
            change = 1 - change;
            glutPostRedisplay();
            break;
    }
}

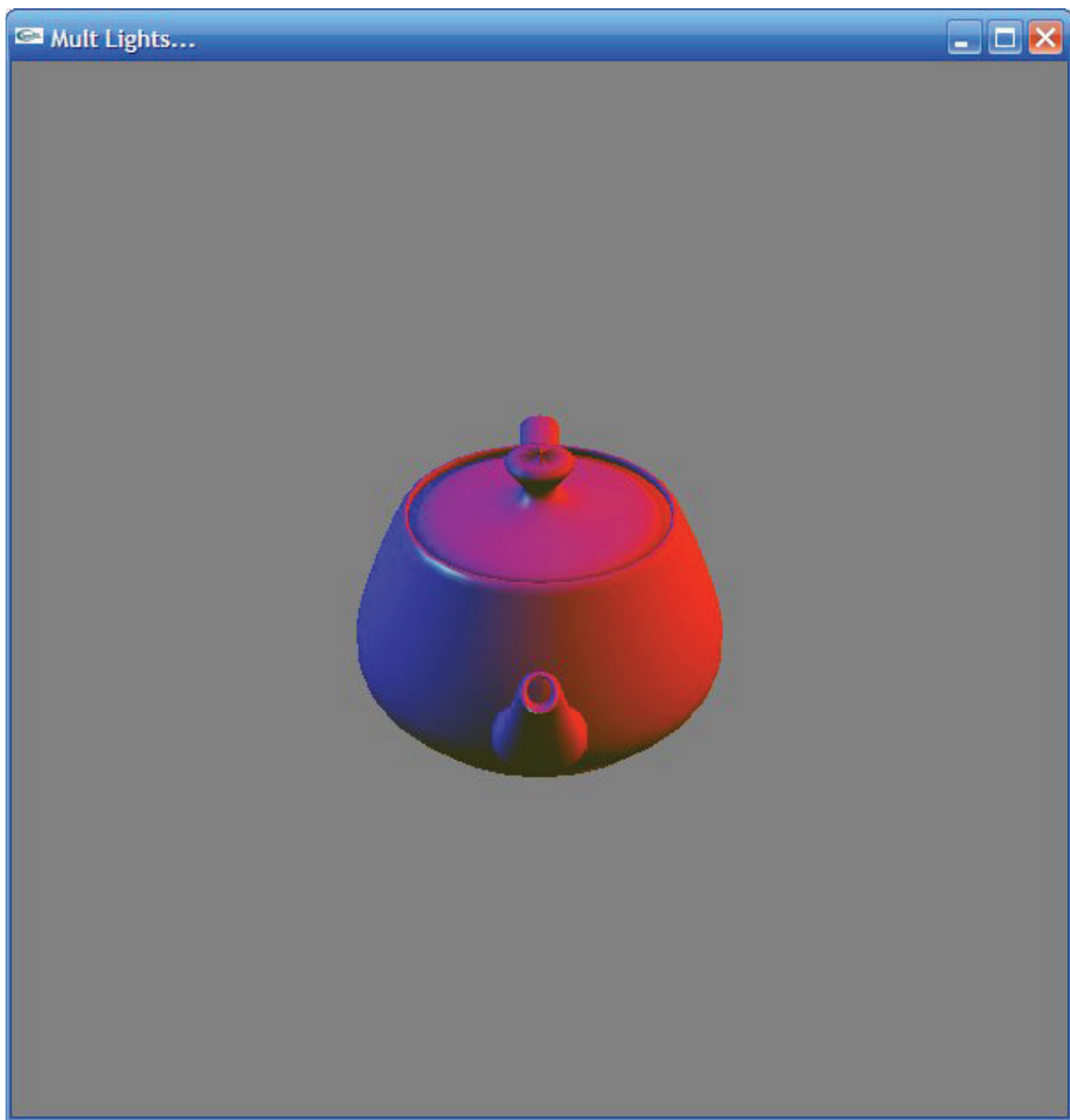
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (600, 600);

```

```
glutInitWindowPosition (100, 100);  
glutCreateWindow ("Mult Lights...");  
init ();  
glutDisplayFunc(display);  
glutKeyboardFunc(keyboard);  
glutMainLoop();  
return 0;  
}
```

Output





Textures

Texture mapping is the process of applying colors from an image to an object created in OpenGL. A simple example would be, applying a brick image onto a square which makes the square look like an original wall.

OpenGL supports four basic texture map types. 1D, 2D, 3D and cube map textures. The following code is an example of a 2D texture mapping, which provides a basic usage of textures.

Code Assumptions

The following are the commonly used steps to perform texture mapping.

1. Obtain an unused texture object identifier with *glGenTextures()*, and create a texture object using *glBindTexture()*.
2. Specify the recture image using *glTexImage2D()*.
3. Bind the texture object with *glBindTexture()*.
4. Enable texture mapping.

In the following code, we basically take a 2D bitmap image and map it onto a square created using the parameter *GL_QUADS*. We also use an uneven object tea pot to see how the texture mapping behaves.

```
#include "stdafx.h"
#include <GL/glut.h>
#include <stdio.h>
#include <gl\gl.h>
#include <gl\glu.h>

GLuint texture[2];

struct Image {
    unsigned long sizeX;
    unsigned long sizeY;
    char *data;
};

typedef struct Image Image;

#define      checkImageWidth 64
#define      checkImageHeight 64
GLubyte checkImage[checkImageWidth][checkImageHeight][3];

void makeCheckImage(void)
{
    int i, j, c;

    for (i = 0; i < checkImageWidth; i++) {
        for (j = 0; j < checkImageHeight; j++) {
            c = (((i&0x8)==0)^(j&0x8==0))*255;
            checkImage[i][j][0] = (GLubyte) c;
        }
    }
}
```

```

        checkImage[i][j][1] = (GLubyte) c;
        checkImage[i][j][2] = (GLubyte) c;
    }
}

int ImageLoad(char *filename, Image *image) {
    FILE *file;
    unsigned long size;           // size of the image in bytes.
    unsigned long i;             // standard counter.
    unsigned short int planes;    // number of planes in image
    (must be 1)
    unsigned short int bpp;      // number of bits per pixel
    (must be 24)
    char temp;                   // temporary color storage for
    bgr-rgb conversion.

    // make sure the file is there.
    if ((file = fopen(filename, "rb"))==NULL)
    {
        printf("File Not Found : %s\n",filename);
        return 0;
    }

    // seek through the bmp header, up to the width/height:
    fseek(file, 18, SEEK_CUR);

    // read the width
    if ((i = fread(&image->sizeX, 4, 1, file)) != 1) {
        printf("Error reading width from %s.\n", filename);
        return 0;
    }
    //printf("Width of %s: %lu\n", filename, image->sizeX);

    // read the height
    if ((i = fread(&image->sizeY, 4, 1, file)) != 1) {
        printf("Error reading height from %s.\n", filename);
        return 0;
    }
    //printf("Height of %s: %lu\n", filename, image->sizeY);

    // calculate the size (assuming 24 bits or 3 bytes per pixel).
    size = image->sizeX * image->sizeY * 3;

    // read the planes
    if ((fread(&planes, 2, 1, file)) != 1) {
        printf("Error reading planes from %s.\n", filename);
        return 0;
    }
    if (planes != 1) {
        printf("Planes from %s is not 1: %u\n", filename, planes);
        return 0;
    }

    // read the bitsperpixel
    if ((i = fread(&bpp, 2, 1, file)) != 1) {
        printf("Error reading bpp from %s.\n", filename);

```



```

    return 0;
}
if (bpp != 24) {
    printf("Bpp from %s is not 24: %u\n", filename, bpp);
    return 0;
}

// seek past the rest of the bitmap header.
fseek(file, 24, SEEK_CUR);

// read the data.
image->data = (char *) malloc(size);
if (image->data == NULL) {
    printf("Error allocating memory for color-corrected image data");
    return 0;
}

if ((i = fread(image->data, size, 1, file)) != 1) {
    printf("Error reading image data from %s.\n", filename);
    return 0;
}

for (i=0;i<size;i+=3) { // reverse all of the colors. (bgr -> rgb)
    temp = image->data[i];
    image->data[i] = image->data[i+2];
    image->data[i+2] = temp;
}

// we're done.
return 1;
}

Image * loadTexture()
{
    Image *image1;
    // allocate space for texture
    image1 = (Image *) malloc(sizeof(Image));
    if (image1 == NULL) {
        printf("Error allocating space for image");
        exit(0);
    }

    //pic.bmp is a 64x64 picture
    if (!ImageLoad("pic256.bmp", image1)) {
        exit(1);
    }
    return image1;
}

void myinit(void)
{
    glClearColor (0.5, 0.5, 0.5, 0.0);
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LESS);

```

```

Image *image1 = loadTexture();
if(image1 == NULL)
{
    printf("Image was not returned from loadTexture\n");
    exit(0);
}
makeCheckImage();

glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

// Create Texture
glGenTextures(2, texture);
glBindTexture(GL_TEXTURE_2D, texture[0]);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); //
scale linearly when image bigger than texture
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); //
scale linearly when image smalled than texture
glTexImage2D(GL_TEXTURE_2D, 0, 3, image1->sizeX, image1->sizeY, 0,
GL_RGB, GL_UNSIGNED_BYTE, image1->data);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);

glBindTexture(GL_TEXTURE_2D, texture[1]);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
glTexImage2D(GL_TEXTURE_2D, 0, 3, checkImageWidth,
    checkImageHeight, 0, GL_RGB, GL_UNSIGNED_BYTE,
    &checkImage[0][0][0]);
glEnable(GL_TEXTURE_2D);
glShadeModel(GL_FLAT);

}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindTexture(GL_TEXTURE_2D, texture[0]);
    glutSolidTeapot(1.0);

    glBindTexture(GL_TEXTURE_2D, texture[1]);

    glBegin(GL_QUADS);

    glTexCoord2f(0.0, 0.0);
        glVertex3f(1.0, -1.0, 0.0);
    glTexCoord2f(1.0, 0.0);
        glVertex3f(1.0, 1.0, 0.0);
    glTexCoord2f(1.0, 1.0);
        glVertex3f(2.41421, 1.0, -1.41421);
    glTexCoord2f(0.0, 1.0);
        glVertex3f(2.41421, -1.0, -1.41421);

```

```

        glEnd();
        glutSwapBuffers();
    }

void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 1.0*(GLfloat)w/(GLfloat)h, 1.0, 30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.6);
}

void keyboard (unsigned char key, int x, int y)
{
    switch (key) {
        case 27:          // "esc" on keyboard
            exit(0);
            break;
        default:          // "a" on keyboard
            break;
    }
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("Texture Mapping");
    myinit();
    glutReshapeFunc (myReshape);
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}

```

Output

