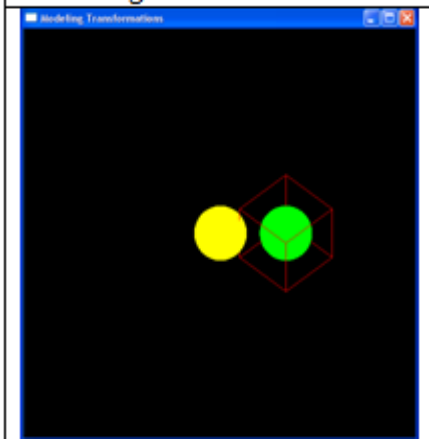
Aim: To create a scene by manipulating objects involved in the scene

Objectives:

1. To understand modeling transformations in OpenGL – Translate, rotate and scale
2. To combine several transformations to achieve a particular result

```
#include <stdlib.h>
#include <GL/freeglut.h>

void init(void);
void display(void);
void reshape (int w, int h);
void keyboard(unsigned char key, int x, int y);


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

   //Select Pixel Format/Device Context Attributes
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);

   //Configure Window
   glutInitWindowSize (512, 512);
   glutInitWindowPosition (100, 100);

   //Create the Window and Set Up Rendering Context
   glutCreateWindow ("Modeling Transformations");

   //Configure Rendering Context
   init();

   //Connect Callback Functions That Will Respond to Events
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutKeyboardFunc(keyboard);
```

Modeling Transformation



```
    //Start listening for events
    glutMainLoop();
    return 0;
}

void init(void)
{
    //Put OpenGL Initializing Code here
      glClearColor(0.0, 0.0, 0.0, 0.0);

      glShadeModel (GL_SMOOTH);

      glEnable(GL_DEPTH_TEST);
}

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

      glColor3f(1.0, 1.0, 0.0);
      glutSolidSphere (0.2, 100, 10);

      glPushMatrix();
            glTranslatef(0.5, 0.0, 0.0);
//          glScalef(0.5, 1.0, 1.0);
            glColor3f(0.0, 1.0, 0.0);
            glutSolidSphere (0.2, 100, 10);
```

```
            glRotatef(45.0, 1.0, 0.0, 0.0);
            glRotatef(45.0, 0.0, 1.0, 0.0);
            glColor3f(1.0, 0.0, 0.0);
            glutWireCube(0.5);
      glPopMatrix();
//    glutSolidTeapot(0.2);

   glutSwapBuffers();
}

void reshape (int w, int h)
{   //Put Resizing Code Here
      glViewport(0, 0, (GLsizei) w, (GLsizei) h);
      glMatrixMode(GL_PROJECTION);
      glLoadIdentity();
      glOrtho(-1.5, 1.5, -1.5, 1.5, -1.5, 1.5);
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
}

void keyboard(unsigned char key, int x, int y)
{
   switch (key) {
      case 27:
      case 'q':
      case 'Q':
         exit(0);
         break;
   }
}
```

Name

glClearColor — specify clear values for the color buffers

C Specification

```
void glClearColor( GLclampf red,
                   GLclampf green,
                   GLclampf blue,
                   GLclampf alpha);
```

Parameters

red, *green*, *blue*, *alpha*

Specify the red, green, blue, and alpha values used when the color buffers are cleared. The initial values are all 0.

Description

`glClearColor` specifies the red, green, blue, and alpha values used by glClear to clear the color buffers. Values specified by `glClearColor` are clamped to the range 0, 1 .

Name

glClear — clear buffers to preset values

C Specification

```
void glClear( GLbitfield mask);
```

Parameters

mask

Bitwise OR of masks that indicate the buffers to be cleared. The three masks are `GL_COLOR_BUFFER_BIT`, `GL_DEPTH_BUFFER_BIT`, and `GL_STENCIL_BUFFER_BIT`.

Description

`glClear` sets the bitplane area of the window to values previously selected by glClearColor, glClearDepthf, and glClearStencil

The pixel ownership test, the scissor test, dithering, and the buffer writemasks affect the operation of `glClear`. The scissor box bounds the cleared region. Blend function, stenciling, fragment shading, and depth-buffering are ignored by `glClear`.

`glClear` takes a single argument that is the bitwise OR of several values indicating which buffer is to be cleared.

The values are as follows:

`GL_COLOR_BUFFER_BIT`

Indicates the buffers currently enabled for color writing

`GL_DEPTH_BUFFER_BIT`

Indicates the depth buffer

`GL_STENCIL_BUFFER_BIT`

Indicates the stencil buffer

The value to which each buffer is cleared depends on the setting of the clear value for that buffer.

Using floating-point variables when dealing with OpenGL parameters we will use the function glColor3f which takes the R, G and B components of the color in the following manner.

```
glColor3f(R, G, B);
```

Each color component can range from 0.0f to 1.0f where 0.0f is no color and 1.0f is full color. For example to specify pure red you would do the following:

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

OpenGL doesn't have a specific function to assign a color to a specific vertex for instance, or a triangle at the time of the function call.

However, OpenGL has a CURRENT color which is always stored somewhere and all you can do is modify that current color. To explain this in more detail in

contrast to something else, imagine a different, non-OpenGL program which requires drawing pixels of different colors on the screen.

You specify the address someplace in the video memory and place a value of your color into that address, instantly the pixel lights up on the screen.

OpenGL, in contrast, doesn't work that way. It always stores a "current" color value at some place else that you don't need to worry about. What you do need to worry about is actually changing that value. All vertices of any primitive are drawn using the current color.

Just remember that that value is always stored somewhere and you need to modify it every time you need a new color. Initially OpenGL sets that current color value to white (R = 1.0f, G = 1.0f, B = 1.0f). That's exactly the reason why without specifying the color in previous tutorial the triangles you drew appeared white.

Now practice by drawing a red triangle. Remember that all of this should be typed inside the RenderFrame function

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

glBegin(GL_TRIANGLES);

glVertex3f(-1.0f, -0.5f, -4.0f);    // A
glVertex3f( 1.0f, -0.5f, -4.0f);    // B
glVertex3f( 0.0f,  0.5f, -4.0f);    // C

glEnd();
```

This will result in a red triangle rendered on the screen. Note how the current color is assigned to red, before drawing the triangle. It is possible to assign color outside the glBegin-glEnd block as well as inside. The following code assigns color to each vertex which results in a smoothly shaded triangle as illustrated below.

```
glBegin(GL_TRIANGLES);

glColor3f(1.0f, 0.0f, 0.0f);
glVertex3f(-1.0f, -0.5f, -4.0f);    // A

glColor3f(0.0f, 1.0f, 0.0f);
glVertex3f( 1.0f, -0.5f, -4.0f);    // B

glColor3f(0.0f, 0.0f, 1.0f);
glVertex3f( 0.0f,  0.5f, -4.0f);    // C

glEnd();
```

As you can see, you can change current color between calls to glVertex which in return assigns that color to the following vertex. This technique is called smooth-shading and can be somehwat considered the next step after flat-shading

Name

glRect — draw a rectangle

C Specification

```
void glRectd(GLdouble x1,
             GLdouble y1,
             GLdouble x2,
             GLdouble y2);
void glRectf(GLfloat x1,
             GLfloat y1,
             GLfloat x2,
             GLfloat y2);
void glRecti(GLint x1,
             GLint y1,
             GLint x2,
             GLint y2);
void glRects(GLshort x1,
             GLshort y1,
             GLshort x2,
             GLshort y2);
```

Parameters

x1, *y1*

Specify one vertex of a rectangle

x2, *y2*

Specify the opposite vertex of the rectangle

Description

`glRect` supports efficient specification of rectangles as two corner points. Each rectangle command takes four arguments, organized either as two consecutive pairs of x y coordinates or as two pointers to arrays, each containing an x y pair. The resulting rectangle is defined in the z = 0 plane.

`glRect`(*x1*, *y1*, *x2*, *y2*) is exactly equivalent to the following sequence:

```
glBegin(GL_POLYGON);
glVertex2(x1, y1);
glVertex2(x2, y1);
glVertex2(x2, y2);
glVertex2(x1, y2);
glEnd();
```

Name

glFlush — force execution of GL commands in finite time

C Specification
void **glFlush**(*void*);

Description

Different GL implementations buffer commands in several different locations, including network buffers and the graphics accelerator itself. glFlush empties all of these buffers, causing all issued commands to be executed as quickly as they are accepted by the actual rendering engine. Though this execution may not be completed in any particular time period, it does complete in finite time.

Name

glMatrixMode — specify which matrix is the current matrix

C Specification

`void` **`glMatrixMode`**`(` GLenum *mode*`);`

Parameters

mode

Specifies which matrix stack is the target for subsequent matrix operations. Three values are accepted: GL_MODELVIEW, GL_PROJECTION, and GL_TEXTURE. The initial value is GL_MODELVIEW. Additionally, if the ARB_imaging extension is supported, GL_COLOR is also accepted.

Description

glMatrixMode sets the current matrix mode. *mode* can assume one of four values:

GL_MODELVIEW

Applies subsequent matrix operations to the modelview matrix stack.

GL_PROJECTION

Applies subsequent matrix operations to the projection matrix stack.

GL_TEXTURE

Applies subsequent matrix operations to the texture matrix stack.

GL_COLOR

Applies subsequent matrix operations to the color matrix stack.

To find out which matrix stack is currently the target of all matrix operations, call glGet with argument GL_MATRIX_MODE. The initial value is GL_MODELVIEW.

Name

glLoadIdentity — replace the current matrix with the identity matrix

C Specification

```
void glLoadIdentity( void );
```

Description

`glLoadIdentity` replaces the current matrix with the identity matrix. It is semantically equivalent to calling glLoadMatrix with the identity matrix

1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1

but in some cases it is more efficient.

Name

glOrtho — multiply the current matrix with an orthographic matrix

C Specification

```
void glOrtho( GLdouble left,
              GLdouble right,
              GLdouble bottom,
              GLdouble top,
              GLdouble nearVal,
              GLdouble farVal );
```

Parameters

left, *right*

Specify the coordinates for the left and right vertical clipping planes.

bottom, *top*

Specify the coordinates for the bottom and top horizontal clipping planes.

nearVal, *farVal*

Specify the distances to the nearer and farther depth clipping planes. These values are negative if the plane is to be behind the viewer.

Description

`glOrtho` describes a transformation that produces a parallel projection.

.....

Name

glLineWidth — specify the width of rasterized lines

C Specification

`void` **`glLineWidth`**`(`GLfloat *width*`);`

Parameters

`width`

Specifies the width of rasterized lines. The initial value is 1.

Description

`glLineWidth` specifies the rasterized width of both aliased and antialiased lines. Using a line width other than 1 has different effects, depending on whether line antialiasing is enabled. To enable and disable line antialiasing, call glEnable and glDisable with argument `GL_LINE_SMOOTH`. Line antialiasing is initially disabled.

Name

glVertex — specify a vertex

C Specification

`void` **`glVertex2s`**`(` GLshort *x*,
GLshort *y*`);`

`void` **`glVertex2i`**`(` GLint *x*,
GLint *y*`);`

`void` **`glVertex2f`**`(` GLfloat *x*,
GLfloat *y*`);`

`void` **`glVertex2d`**`(` GLdouble *x*,
GLdouble *y*`);`

`void` **`glVertex3s`**`(` GLshort *x*,

```
                    GLshort y,
                    GLshort z);

void glVertex3i(    GLint x,
                    GLint y,
                    GLint z);

void glVertex3f(    GLfloat x,
                    GLfloat y,
                    GLfloat z);

void glVertex3d(    GLdouble x,
                    GLdouble y,
                    GLdouble z);

void glVertex4s(    GLshort x,
                    GLshort y,
                    GLshort z,
                    GLshort w);

void glVertex4i(    GLint x,
                    GLint y,
                    GLint z,
                    GLint w);

void glVertex4f(    GLfloat x,
                    GLfloat y,
                    GLfloat z,
                    GLfloat w);

void glVertex4d(    GLdouble x,
                    GLdouble y,
                    GLdouble z,
                    GLdouble w);
```

Parameters

x, *y*, *z*, *w*

Specify *x*, *y*, *z*, and *w* coordinates of a vertex. Not all parameters are present in all forms of the command.

Parameters

v

Specifies a pointer to an array of two, three, or four elements. The elements of a two-element array are x and y; of a three-element array, x, y, and z; and of a four-element array, x, y, z, and w.

Description

`glVertex` commands are used within glBegin/glEnd pairs to specify point, line, and polygon vertices. The current color, normal, texture coordinates, and fog coordinate are associated with the vertex when `glVertex` is called.

When only x and y are specified, z defaults to 0 and w defaults to 1. When x, y, and z are specified, w defaults to 1.

Name

glViewport — set the viewport

C Specification

```
void glViewport(GLint   x,
                GLint   y,
                GLsizei width,
                GLsizei height);
```

Parameters

x, *y*

Specify the lower left corner of the viewport rectangle, in pixels. The initial value is (0,0).

width, *height*

Specify the width and height of the viewport. When a GL context is first attached to a window, *width* and *height* are set to the dimensions of that window.

Description

`glViewport` specifies the affine transformation of x and y from normalized device coordinates to window coordinates.