Assignment No 13

Name : Sameer Manoj Brambhecha
Roll NO : 2111S
Batch : E-1
Date of Performance: 10-12-21
Date of Submission: 14-12-21

Title :- 3D Transformations Using Open GL.

Problem Statement:-
Write a C++ program to draw 3-D cube and perform following transformations on it using Open GL i) Scaling, ii) Translation, iii) Rotation about an axis (x/y/z).

Learning Objectives:- To implement Open GL functions and 3-D Transformation.

Learning Outcomes:→ After completion of this assignment, students will be able to implement OpenGL functions and 3-D ~~Transfoo~~ Transformations.

S/W and H/W Requirements:-
1.) Windows 10 OS
2.) Open Source C++ programming tool like G++/GCC, open GL.

Theory:

OpenGL Basics :→

Open Graphics library (OpenGL) is a cross-language, cross-platform API for rendering 2D and 3D Vector Graphics (use of polygons to represent image). OpenGL is a low level, widely supported modeling and rendering software package, available across all platforms. It can be used in a range of graphics applications, such as games, CAD design, or modelling. OpenGL API is designed mostly in hardware.

Since, ~~OPENGL~~ OpenGL is a graphics API and not a platform of its own, it requires a language to operate in and the language of choice is C++.

OpenGL syntax

- All functions have the form : gl*
- glVertex3f() - 3 means that this function take 3 arguements, and f means that the type of those arguements is float.
- glVertex2i() - 2 means that this function take 2 arguements, and i means that the type of those arguements is int.
- All variable types have the form GL*.
- In OpenGL program, it is better to use OpenGL variable types.
- GLfloat instead of float
- GLint instead of int.

OpenGL primitives:-

Drawing 2 lines:→

```
glBegin(GL_LINES);
glVertex3f(_,_,_); // Start pt. of line 1
glVertex3f(_,_,_); // End pt. of line 1.
```

glVertex3f(-,-,-); start pt of line 2.
glVertex3f(-,-,-); end pt of line 2.
glEnd();
We can replace GL_LINES with GL_POINTS, GL_LINELOOPS, GL_POLYGON, etc.

Open GL provides a consistent interface to the underlying graphics hardware. This abstraction allows a single program to run a different graphics hardware easily. A program written with Open GL can even be run in software (slowly) on machines with no graphics acceleration. OpenGL function names always begin with gl, such as glClear(), and they may end with characters that indicate the types of the parameters, for example glColor3f(GLfloat red, GLfloat green, GLfloat blue) takes three floating point color parameters and glColor4dv(const GLdouble *v) takes a pointer to an array that contains 4 double-precision floating point values. OpenGL constants begin with GL, such as GL_DEPTH. OpenGL also uses special names for types that are passed to its functions, such as GLfloat or GLint, the corresponding C types are compatible, that is float and int respectively.

GLU is the OpenGL utility library. It contains useful functions at a higher level than those provided by OpenGL, for example to draw complex shapes or set up cameras. All GLU functions are written on top of OpenGL. Like OpenGL, GLU functions names begin

with glu, and constants begin with ~~Glu~~ GLU.

~~Glut~~ GLUT, the Open GL Utility Toolkit, provides a system for setting up callbacks for interacting with the user and functions for dealing with the windowing system. This abstraction allows a program to run on different operating systems with only a recompile. Glut follows the convention of prepending function names with glut and constants with ~~Glut~~ GLUT.

Writing an Open GL Program with GLUT :=>
An Open GL program using the three libraries listed ~~before~~ above must include the appropriate headers. This requires the following three lines:

```
#include <GL/gl.h>
#include <GL/glu.h>
#include<GL/glut.h>
```

Before Open GL rendering calls can be made, some initialization has to be done. With ~~Glut~~ GLUT, this consists of initializing the GLUT library, initializing the display mode, creating the window and setting up callback functions. the following lines initialize a full color, double buffered display:

```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
```

Double bufferings means that there are two buffers, a front buffer and a back buffer. The front buffer is displayed to the user, while the back buffer is used for rendering operations. This prevents

flickering that would occur if we rendered directly to the front buffer.

Next, a window is created with GLUT that will contain the viewport which displays the OpenGL front buffer with the following three lines:

```
glutInitWindowPosition(px, py);
glutInitWindowSize(Sx, Sy);
glutCreateWindow(name);
```

To register callback functions, we simply pass the name of the function that handles the events to the appropriate GLUT function.

```
glutReshapeFunc(reshape);
glutDisplayFunc(display);
```

Here, the functions should have the following prototypes: `void reshape(int width, int height);`
`void display();`

In this example, when the user resizes the window, reshape is called by GLUT and when the display needs to be refreshed, the display function is called. For animation, an idle event handler that takes no arguements can be created to call the display function to constantly redraw the scene with `glutIdleFunc`. Once all the callbacks have been set up, a call to `glutMainLoop` allows the program to run.

In the display function, typically the image buffer is cleared, primitives are rendered to it, and the results are presented to the user. The following line clears the image buffer, setting each pixel color to the clear color, which can be configured to be any color:

```
glClear(GL_COLOR_BUFFER_BIT);
```

The next line sets the current rendering color to blue. OpenGL behaves like a state machine, so certain state such as the rendering color is saved by OpenGL and used automatically later as it is needed.

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

To render a primitive, such as a point, line, or polygon, OpenGL requires that a call to glBegin is made to specify the type of primitive being rendered.

```
glBegin(GL_LINES);
```

Only a subset of OpenGL commands is available after a call to glBegin. The main command that is used is glVertex, which specifies a vertex position. In GL_LINES mode, each pair of vertices define endpoints of a line segment. In this case, a line would be drawn from the point at (x_0, y_0) to (x_1, y_1).

```
glVertex2f(x0, y0); glVertex2f(x1, y1);
```

A call to glEnd completes rendering of the current primitive. `glEnd();` Finally, the back buffer needs to be swapped to the front buffer that the user will see, which GLUT can handle for us:

```
glutSwapBuffers();
```

✱ 3-D Transformations :→

① Translation :-

A position $P = (x, y, z)$ in 3-Dimensional space is translated to a location $P = (x, y, z)$ by adding translational distances tx, ty and tz to the cartesian coordinates of P:

$$x' = x + tx$$
$$y' = y + ty$$
$$z' = z + tz$$

Matrix form:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$\boxed{P' = T \cdot P}$$

② Scaling :- Scaling refers to changing the size of the object either by increasing or decreasing.

Let S_x, S_y and S_z be scaling factors along x, y and z axis.

$$\therefore x' = x * S_x$$
$$y' = y * S_y$$
$$z' = z * S_z$$

Matrix form

Matrix form:

$$[x' \; y' \; z' \; 1] = [x \; y \; z \; 1] \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boxed{P' = P \cdot S}$$

(3) Rotation :-

Unlike 2-D rotation, where all transformations are carried out in the xy plane, a 3-D rotation can be specified around any line in space. Therefore, for 3-D rotation we have to specify an axis of rotation about which the object is to be rotated along with the angle of rotation.

~~In~~ 3-D transformation matrix for each coordinate axis ~~of~~ rotation with homogeneous coordinate are as follows

1.) $R_z = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

2.) $R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

3.) $R_y = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

R for z axis rotation $P' = P \cdot R_z(\theta)$
for y axis rotation $P' = P \cdot R_y(\theta)$
for x axis rotation $P' = P \cdot R_x(\theta)$

Algorithm:→

① Algorithm for setIdentityM() function:→

1.) Start
2) Pass Matrix m as arguement to the function.
3.) Set m[i][j] = (i==j).
4.) Repeat step 3 for j=0 to j=3
5.) Repeat step 4 for j=0 to i=3.
6.) Stop.

② Algorithm for translate() function:→

1.) Start
2.) Pass tx, ty and tz as arguements to the function.
3.) Set output[i][0] equal to (input[i][0] + tx).
4.) Set Output[i][1] equal to (input[i][1] + ty)
5.) Set Output[i][2] equal to (input[i][2] + tz)
6.) Repeat steps 3, 4 and 5 for i=0 to i=8.
7.) Stop.

③ Algorithm for scale() function:→

1.) Start
2.) Pass sx, sy, sz as arguements to the function.
3.) Set theMatrix[0][0] = sx

4.) Set theMatrix[1][1] = sy
5.) Set theMatrix[2][2] equal to sz.
6.) Stop.

④ Algorithm for RotateX() function :→

1.) Start
2.) Pass 'angle' as arguement to the function
3.) Set angle equal to (angle * 3.14)/180
4.) Set theMatrix[1][1] equal to cos(angle)
5.) Set theMatrix[1][2] equal to sin(angle)
6.) Set theMatrix[2][1] equal to -sin(angle)
7.) Set theMatrix[2][2] equal to cos(angle)
8.) Stop.

⑤ Algorithm for RotateY() function :→

1.) Start
2.) Pass 'angle' as arguement to the function.
3.) Set angle equal to (angle * 3.14)/180
4.) Set theMatrix[0][0] equal to cos(angle)
5.) Set theMatrix[0][2] equal to -sin(angle)
6.) Set theMatrix[2][0] equal to sin(angle)
7.) Set theMatrix[2][2] equal to cos(angle)
8.) Stop.

⑥ Algorithm for RotateZ() function :→

1.) Start
2.) Pass 'angle' as arguement to the function.
3.) Set angle equal to (angle * 3.14)/180.

4.) Set theMatrix [0][0] = cos (angle)
5.) Set theMatrix [0][1] = -sin (angle)
6.) Set theMatrix [1][0] = sin (angle)
7.) Set theMatrix [1][1] = cos (angle)
8.) Stop.

(7.) Algorithm for MultiplyM() functions :-

1.) Start.
2.) Set output [i][j] equal to 0.
3.) Set output [i][j] equal to output [i][j] + input [i][k] * theMatrix [k][j]
4.) Repeat step 3 for k=0 to k=3
5.) Repeat step 2 and 4 for j=0 to j=3
6.) Repeat step 5 for i=0 to i=8.
7.) Stop.

8.) Algorithm for Axes() function :-

1.) Start.

8.) Algorithm for display() function :→

1.) Start.
2.) Call Axes() function to draw the axis (x and y).
3.) Set background color to white using glColor3f (1.0, 1.0, 1.0).
4.) Create a identity Matrix theMatrix. using the function setIdentityM().
5.) If case : 1, call translate (tx, ty, tz).
6.) If case 2 ; call scale (sx, sy, sz) and MultiplyM().

7.) If case 3, then case 1: , call Rotatex (angle)
8.) If case 2: call Rotatey (angle)
9.) If case 3: call Rotatez (angle)
10.) Outside ~~case~~ switch ~~case 3~~ statement 2, call MultiplyM() function.
11.) Call draw (output)
12.) Call glFlush().
13.) Stop.

9.) Algorithm for main() function:-

1.) Start
2.) Declare window ~~screen~~ size, name and position.
3.) Set display mode as GLUT_RGB.
4.) Call init() function.
5.) Display menu with 4 choices.
6.) Read user's choice.
7.) If choice == 1, Accept tx, ty, tz.
8.) If choice == 2; Accept sx, sy and sz.
9.) If choice == 3 > read choice for rotation.
10.) if rotch == 1, 2 or 3, accept angle.
11.) if choice == 4, Exit the code.
12.) Call glutDisplayfunc (display);
13.) Call glutMainLoop();
14.) Stop.

Conclusion:- Hence, we have successfully implemented 3-D Transformation using Open GL.