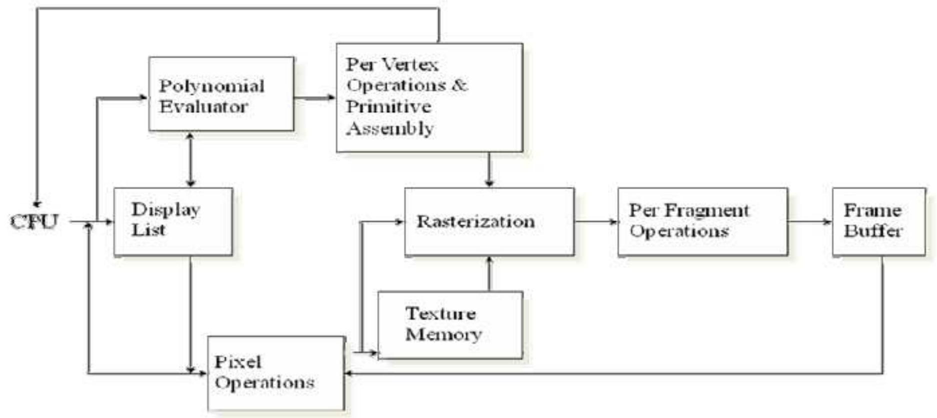
1 Introduction to OpenGL

What is OpenGL?

It is a window system independent, operating system independent graphics rendering API which is capable of rendering high-quality color images composed of geometric and image primitives. OpenGL is a library for doing computer graphics. By using it, you can create interactive applications which render high-quality color images composed of 3D geometric objects and images.
As OpenGL is window and operating system independent. As such, the part of your application which does rendering is platform independent. However, in order for OpenGL to be able to render, it needs a window to draw into. Generally, this is controlled by the windowing system on whatever platform you're working on. Summarizing the above discussion, we can say OpenGL is a software API to graphics hardware.

• Designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms.
• Procedural interface.
• No windowing commands.
• No high-level commands.



OpenGL as a Renderer

OpenGL is a library for rendering computer graphics. Generally, there are two operations that you do with OpenGL:
• draw something
• change the state of how OpenGL draws
OpenGL has two types of things that it can render: geometric primitives and image primitives. *Geometric primitives* are points, lines and polygons. *Image primitives* are bitmaps and graphics images (i.e. the pixels that you might extract from a JPEG image after you'veread it into your program.) Additionally, OpenGL links image and geometric

primitives together using *texture mapping*, which is an advanced topic we'll discuss this afternoon.
The other common operation that you do with OpenGL is *setting state.* "Setting state" is the process of initializing the internal data that OpenGL uses to render your primitives. It can be as simple as setting up the size of points and color that you want a vertex to be, to initializing multiple mipmap levels for texture mapping.

OpenGL and Related APIs

OpenGL is window and operating system independent. To integrate it into various window systems, additional libraries are used to modify a native window into an OpenGL capable window. Every window system has its own unique library and functions to do this.
Some examples are:

- GLX for the X Windows system, common on Unix platforms
- AGL for the Apple Macintosh
- WGL for Microsoft Windows
-

OpenGL also includes a utility library, GLU, to simplify common tasks such as: rendering quadric surfaces (i.e. spheres, cones, cylinders, etc.), working with NURBS and curves, and concave polygon tessellation.
Finally to simplify programming and window system dependence, we'll be using the freeware library, GLUT. GLUT, written by Mark Kilgard, is a public domain window system independent toolkit for making simple OpenGL applications. It simplifies the processof creating windows, working with events in the window system and handling animation.

2 GLUT (OpenGL Utility Toolkit)

Introduction

GLUT (pronounced like the glut in gluttony) is the OpenGL Utility Toolkit, a window system independent toolkit for writing OpenGL programs. It implements a simple windowing application programming interface (API) for OpenGL. GLUT makes it considerably easier to learn about and explore OpenGL programming. GLUT provides a portable API so you can write a single OpenGL program that works across all PC and workstation OS platforms.
GLUT is designed for constructing small to medium sized OpenGL programs. While GLUT is well-suited to learning OpenGL and developing simple OpenGL applications, GLUT is not a full-featured toolkit so large applications requiring sophisticated user interfaces are better off using native window system toolkits. GLUT is simple, easy, and small. The GLUT library has C, C++ (same as C), FORTRAN, and Ada programming bindings. The GLUT source code distribution is portable to nearly all OpenGL implementations and platforms. The current version is 3.7. Additional releases of the library are not anticipated.
GLUT is not open source. Mark Kilgard maintains the copyright. There are a number of newer and open source alternatives.
The toolkit supports:

- Multiple windows for OpenGL rendering

• Callback driven event processing
• Sophisticated input devices
• An 'idle' routine and timers
• A simple, cascading pop-up menu facility
• Utility routines to generate various solid and wire frame objects
• Support for bitmap and stroke fonts

Design Philosophy

GLUT simplifies the implementation of programs using OpenGL rendering. The GLUT application programming interface (API) requires very few routines to display a graphics scene rendered using OpenGL. The GLUT API (like the OpenGL API) is stateful.
Most initial GLUT state is defined and the initial state is reasonable for simple programs. The GLUT routines also take relatively few parameters. No pointers are returned. The only pointers passed into GLUT are pointers to character strings (all strings passed to GLUT are copied, not referenced) and opaque font handles.
The sub-APIs are:
Initialization.
Command line processing, window system initialization, and initial window creation state are controlled by these routines.
Beginning Event Processing.
This routine enters GLUT's event processing loop. This routine never returns, and it continuously calls GLUT callbacks as necessary.
Window Management.
These routines create and control windows.
Overlay Management.
These routines establish and manage overlays for windows.
Menu Management.
These routines create and control pop-up menus.
Callback Registration.
These routines register callbacks to be called by the GLUT event processing loop.
Color Index Colormap Management.
These routines allow the manipulation of color index colormaps for windows.
State Retrieval.
These routines allows programs to retrieve state from GLUT.
Font Rendering.
These routines allow rendering of stroke and bitmap fonts.

In GLUT's ANSI C binding, for most routines, basic types (`int`, `char*`) are used as parameters. In routines where the parameters are directly passed to OpenGL routines, OpenGL types (`GLfloat`) are used. The header files for GLUT should be included in GLUT programs with the following include directive:

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

Examples on OpenGL function formats:
Functions have prefix **gl** and initial capital letters for each word
glClearColor(), glEnable(), glPushMatrix() …
OpenGL data types
GLfloat, GLdouble, GLint, GLenum, …

glutInit

`glutInit` is used to initialize the GLUT library.
Usage
```
void glutInit(int *argcp, char **argv);
```

`argcp`
A pointer to the program's *unmodified* `argc` variable from `main`. Upon return, the value pointed to by `argcp` will be updated, because `glutInit` extracts any command line options intended for the GLUT library.

`argv`
The program's *unmodified* `argv` variable from `main`. Like `argcp`, the data for `argv` will be updated because `glutInit` extracts any command line options understood by the GLUT library.

Description
`glutInit` will initialize the GLUT library and negotiate a session with the window system.

During this process, `glutInit` may cause the termination of the GLUT program with an error message to the user if GLUT cannot be properly initialized. Examples of this situation include the failure to connect to the window system, the lack of window system support for OpenGL, and invalid command line options.

`glutInit` also processes command line options, but the specific options parse are window system dependent.

glutInitWindowPosition, glutInitWindowSize

`glutInitWindowPosition` and `glutInitWindowSize` set the *initial window position* and *size* respectively.

Usage
```
void glutInitWindowSize(int width, int height);
void glutInitWindowPosition(int x, int y);
```
`width`
Width in pixels.
`height`
Height in pixels.
`x`
Window X location in pixels.
`y`
Window Y location in pixels.

Description
Windows created by `glutCreateWindow` will be requested to be created with the current *initial window position* and *size*.

glutInitDisplayMode

`glutInitDisplayMode` sets the *initial display mode*.
Usage

```
void glutInitDisplayMode(unsigned int mode);
```

`mode`
Display mode, normally the bitwise *OR*-ing of GLUT display mode bit masks. See values below:

`GLUT_RGBA`
Bit mask to select an RGBA mode window. This is the default if neither `GLUT_RGBA` nor `GLUT_INDEX` are specified.

`GLUT_RGB`
An alias for `GLUT_RGBA`.

`GLUT_INDEX`
Bit mask to select a color index mode window. This overrides `GLUT_RGBA` if it is also specified.

`GLUT_SINGLE`
Bit mask to select a single buffered window. This is the default if neither

`GLUT_DOUBLE` or `GLUT_SINGLE` are specified.

`GLUT_DOUBLE`
Bit mask to select a double buffered window. This overrides `GLUT_SINGLE` if it is also specified.

`GLUT_ACCUM`
Bit mask to select a window with an accumulation buffer.

`GLUT_ALPHA`
Bit mask to select a window with an alpha component to the color buffer(s).

`GLUT_DEPTH`
Bit mask to select a window with a depth buffer.

`GLUT_STENCIL`
Bit mask to select a window with a stencil buffer.

`GLUT_MULTISAMPLE`
Bit mask to select a window with multisampling support. If multisampling is not available, a non-multisampling window will automatically be chosen. Note: both the OpenGL client-side and server-side implementations must support the `GLX_SAMPLE_SGIS` extension for multisampling to be available.

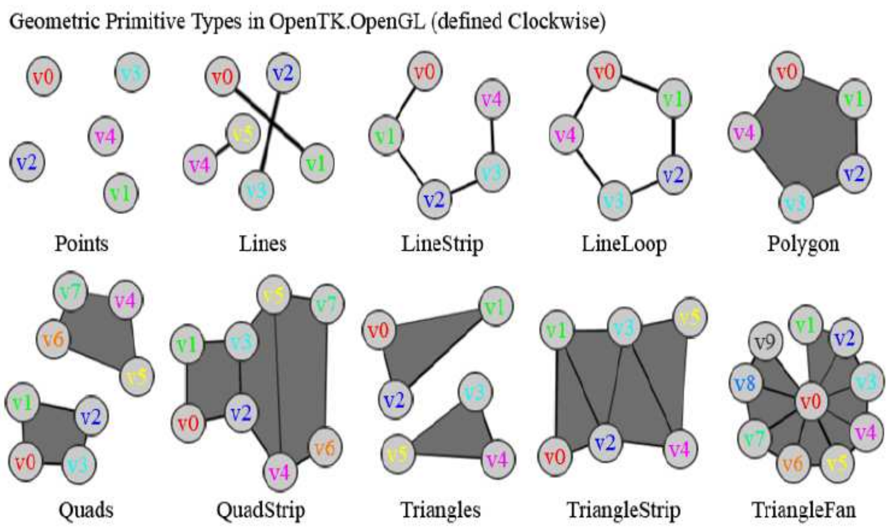
Description

The *initial display mode* is used when creating top-level windows, subwindows, and overlays to determine the OpenGL display mode for the to-be-created window or overlay. Note that `GLUT_RGBA` selects the RGBA color model, but it does not request any bits of alpha (sometimes called an *alpha buffer* or *destination alpha*) be allocated. To request alpha,specify `GLUT_ALPHA`. The same applies to `GLUT_LUMINANCE`.

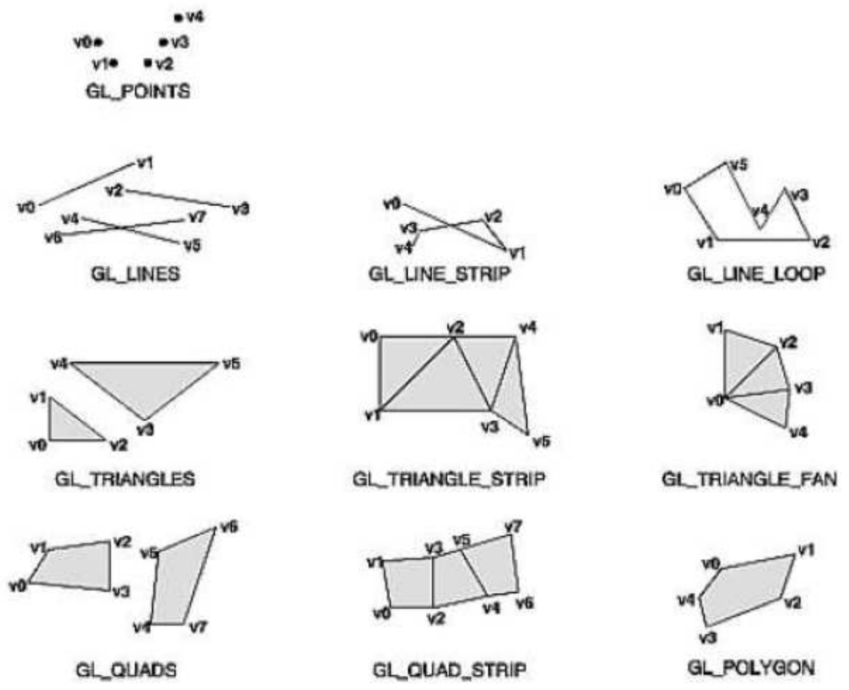
First Single OpenGL Program:

```
/* simple.c second version */
/* This program draws a white rectangle on a black background.*/
#include <glut.h> /* glut.h includes gl.h and glu.h*/
void display()
{
/* clear window */
glClear(GL_COLOR_BUFFER_BIT);
/* draw unit square polygon */
glBegin(GL_POLYGON);
glVertex2f(-0.5, -0.5);
glVertex2f(-0.5, 0.5);
glVertex2f(0.5, 0.5);
glVertex2f(0.5, -0.5);
glEnd();
/* flush GL buffers */
glFlush();
}
void init() // initialize colors
{
/* set clear color to black */
glClearColor(0.0, 0.0, 0.0, 0.0);
/* set fill color to white */
glColor3f(1.0, 1.0, 1.0);
}
void main(int argc, char** argv)
{
/* Initialize mode and open a window in upper left corner of/* screen */
/* Window title is name of program (arg[0]) */
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(500, 500); // Set window Size
glutInitWindowPosition(0, 0); //Set Window Position
glutCreateWindow("simple"); //Create Window and Set title
glutDisplayFunc(display); //Call the Displaying function
init(); //Initialize Drawing Colors
glutMainLoop(); //Keep displaying until program is closed.
}
```

3 Drawing Primitives



Value	Meaning
GL_POINTS	individual points
GL_LINES	pairs of vertices interpreted as individual line segments
GL_LINE_STRIP	series of connected line segments
GL_LINE_LOOP	same as above, with a segment added between last and first vertices
GL_TRIANGLES	triples of vertices interpreted as triangles
GL_TRIANGLE_STRIP	linked strip of triangles
GL_TRIANGLE_FAN	linked fan of triangles
GL_QUADS	quadruples of vertices interpreted as four-sided polygons
GL_QUAD_STRIP	linked strip of quadrilaterals
GL_POLYGON	boundary of a simple, convex polygon



glVertex

The main function (and probably the most used OpenGL function) is function named glVertex. This function defines a point (or a vertex) in your 3D world and it can vary from receiving 2 up to 4 coordinates.

glVertex2f(100.0f, 150.0f); defines a point at x = 100, y = 150, z = 0; this function takes only 2 parameters, z is always 0. glVertex2f can be used in special cases and won't be used a lot unless you're working with pseudo-2D sprites or triangles and points that always have to be constrained by the depth coordinate.

glVertex3f(100.0f, 150.0f, -25.0f); defines a point at x = 100, y = 150, z = -25.0f; this function takes 3 parameters, defining a fully 3D point in your world.

glBegin and **glEnd**

glVertex alone won't draw anything on the screen, it merely defines a vertex, usually of a more complex object. To really start displaying something on the screen you will have
to use two additional functions. These functions are
glBegin(int *mode*); and **glEnd**(*void*);

glBegin and **glEnd** delimit the vertices of a primitive or a group of like primitives. What this means is that everytime you want to draw a primitive on the screen you will first have to call glBegin, specifying what kind of primitive it is that you want to draw in the ***mode*** parameter of glBegin, and then list all vertices one by one (by sequentially calling glVertex) and finally call glEnd to let OpenGL know that you're done drawing a primitive. The parameter mode of the function glBegin can be one of the following:

GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
GL_QUADS
GL_QUAD_STRIP
GL_POLYGON

These flags are self-explanatory. As an example the code given below to shows how to draw some primitives.

```
// this code will draw a point located at [100, 100, -25]
glBegin(GL_POINTS);
glVertex3f(100.0f, 100.0f, -25.0f);
glEnd( );
// next code will draw a line at starting and ending coordinates
specified by glVertex3f
glBegin(GL_LINES);
glVertex3f(100.0f, 100.0f, 0.0f); // origin of the line
glVertex3f(200.0f, 140.0f, 5.0f); // ending point of the line
glEnd( );
// the following code draws a triangle
glBegin(GL_TRIANGLES);
glVertex3f(100.0f, 100.0f, 0.0f);
glVertex3f(150.0f, 100.0f, 0.0f);
glVertex3f(125.0f, 50.0f, 0.0f);
glEnd( );
// this code will draw two lines "at a time" to save
// the time it takes to call glBegin and glEnd.
glBegin(GL_LINES);
glVertex3f(100.0f, 100.0f, 0.0f); // origin of the FIRST line
glVertex3f(200.0f, 140.0f, 5.0f); // ending point of the FIRST line
glVertex3f(120.0f, 170.0f, 10.0f); // origin of the SECOND line
glVertex3f(240.0f, 120.0f, 5.0f); // ending point of the SECOND line
glEnd( );
```

BRESENHAM'S LINE DRAWING

AIM:

To implement Bresenham's line drawing Algorithm for drawing lines.

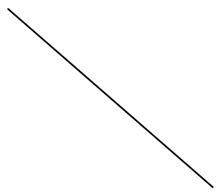
ALGORITHM:

```
Step 1: Start
Step 2: Get the values of the end points as(x1, y1) &(x2, y2)
Step 3: Assign x=x1, y=y1;
Step 4: Compute dx=x2-x1
Step 5: Compute dy=y2-y1
Step 6: Assign sx=x2-x1, sy=y2-y1
Step 7: If dy>dx then interchange the values of dx and dy and assign exch=1
Step 8: Compute p=2xdy-dx
Step 9: Put a pixel on(x,y)
Step 10: If exch=1, y=sy else x=x+sx
Step 11: If p>0 and exch =1, x=x+sx else y=y+sy, p=p-2xdx
Step 12: Compute p=p+2xdy
Step 13: Do steps (9) t0 (12) for dx times
Step 14: Stop
```

INPUT:

```
Enter starting value of X-Axis----->100
Enter starting value of Y-Axis----->100
Enter ending value of X-Axis----->200
Enter ending value of Y-Axis----->200
```

OUTPUT:



CIRCLE DRAWING

AIM:

To write a program to draw a circle using Bresenham’s circle drawing Algorithm.

ALGORITHM:

Step 1: Start

Step 2: Get the center point as (xc,yc),get the radius as r.

Step 3: Assign y=r,x=0

Step 4: Calculate p=3-2r

Step 5: If p<0,p=p+4x+6 else p=p+10+4(x-y) and y=y-1

Step 6: Increment x by 1

Step 7: Do steps (9) to (16)

Step 8: Repeat steps (5) to (9) until x<=y

Step 9: Put a pixel on (xc+x,yc+y,15);

Step 10: Put a pixel on (xc+x,yc-y,15);

Step 11: Put a pixel on (xc-x,yc+y,15);

Step 12: Put a pixel on (xc-x, yc-y, 15);

Step 13: Put a pixel on (xc+y,yc+x,15);

Step14: Put a pixel on (xc+y, yc-x, 15);

Step 15: Put a pixel on (xc-y, yc+x, 15);

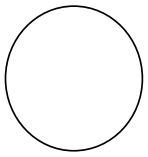
Step 16: Put a pixel on (xc-y, yc-x, 15);

Step 17: Stop

INPUT:

```
Enter X-Axis----->:100
Enter Y-Axis----->:200
Enter radius----->: 40
```

OUTPUT:



ELLIPSE DRAWING

AIM:

To write a program to draw a ellipse using Bresenham's ellipse drawing Algorithm.

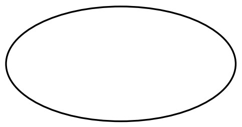
ALGORITHM:

```
Step 1: Start
Step 2: Get the center point as(x1, y1)
Step 3: Get the length of semi-major, semi-minor axes as r1 & r2
Step 4: Calculate t=pi/180
Step 5: Initialise i=0;
Step 6: Compute d=i*t
Step 7: Compute x=x1+y1*sin(d), y=y1+r2*cos(d).
Step 8: Put a pixel on(x,y)
Step 9: Increment I by 1
Step 10: Repeat steps(6) to (9) until i<360
Step 11: Stop
```

INPUT:

```
Enter the center co-or:100 150
Enter the radius1:40
Enter radius2:20
```

OUTPUT:



2D TRANSFORMATION

AIM:

To perform the 2D transformation such as translation, rotation, scaling, shearing, Reflection

ALGORITHM:

```
Step1: Declare the variables xa,ya,xa1,ya1 of array type.
Step2:Declare the variables gd,gm,n,i,op,tx,ty,xf,yf,rx,ry.
Step3: Initialise the graphics function.
Step4: Input the number of points.
Step5: Input the value of co-ordinate according to number of points.
Step6. Using switch statement selects the option to perform translation, rotation, scaling,   reflection
    and shearing.
Step7: Translation:
    a).input the translation vector
    b).add the translation vectors with the coordinates
        xa1[i]=xa[i]=tx, ya1[i]=ya[i]=ty,
    c).using the function line,display the object before and after translation.
Step8: Rotation
    a). input the rotation angle
    b). using formula theta=(theta*3.14)/180
    c).input the value of reference point
    d). calculate new coordinate point using formula
        xa1[i]=xf+(xa[i]-xf)*cos(theta)-(ya[i]-yf)*sin(theta),
        ya1[i]=yf+(xa[i]-xf)*sin(theta)-(ya[i]-yf)*cos(theta),
    e). using the function line,display the object before and after rotation.
Step9: Scaling:
    a).input the scaling factor and reference point
    b).calculate new coordinate point using formula
```

```
                xa1[i]=(xa[i]*sx+rx*(1-sx),
                 ya1 [i] = (ya[i]*sy+ry*(1-sy)
            c). using the function line, display the object before and after scaling.
   Step10: Shearing:
           a).input the shearing value and reference point.
              b). input the shear direction x or y
                  i).if direction x
                        xa1[i]=xa[i]+shx*(ya[i]-yref)
                  ii).otherwise
                        ya1[i]=ya[i]+shy*(xa[i]-xref)
                 iii). using the function line, display the object before and after shearing.
  Step11: Reflection:
            a).display the object before  reflection using the function line
            b). display the object after reflection using the function line
  Step12: Stop.
```

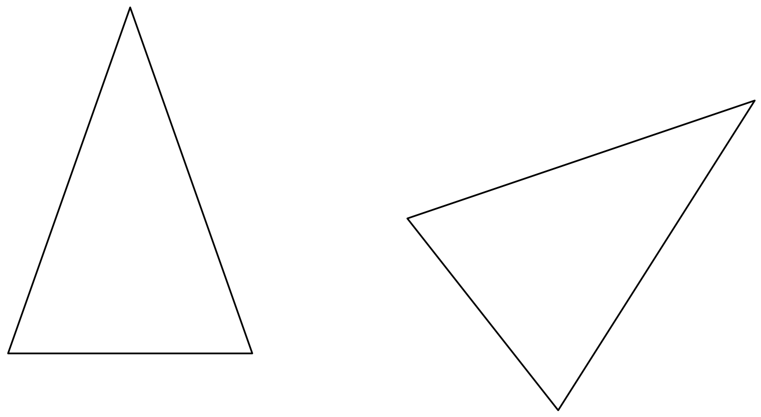
INPUT & OUTPUT:

```
enter the no of points:3
enter the coordinates 1:50 150
enter the coordinates 2:50 50
enter the coordinates 3:75 150

menu
1. translation
2. rotation
3. scaling
4.shearing
5.reflection
6.exit1
enter the translation vector:30 40
```

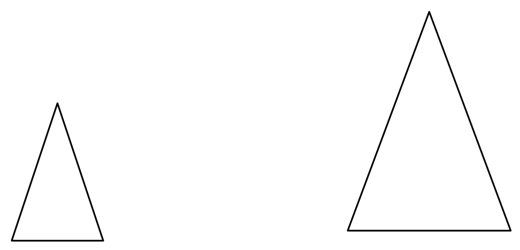
Before translation

After Translation

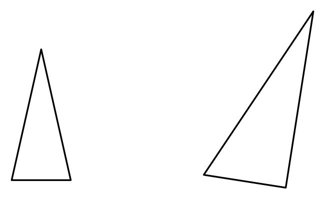
```
menu
1. translation
2. rotation
3. scaling
4.shearing
5.reflection
6.exit 2

enter the rotation angle:40

enter the reference points:100 100

before rotation            after  rotation
```



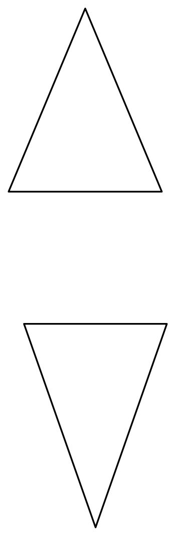
```
menu
1. translation
2. rotation
3. scaling
4.shearing
5.reflection
6.exit  3

Enter the scaling factor: 3 4
Enter the reference points: 30 40

Before scaling                          after scaling
```



```
menu
1. translation
2. rotation
3. scaling
4.shearing
5.reflection
6.exit  4

Enter the shear value: 3 4
Enter the reference point: 20 30
Enter the shear direction x or y: X

Before shearing                  After shearing
```



```
menu
1. translation
2. rotation
3. scaling
4.shearing
5.reflection
6.exit  5

Before reflection after reflection
```



```
menu
1. translation
2. rotation
3. scaling
4.shearing
5.reflection
6.exit  6
```

RESULT:

Thus the program is executed and verified.

COHEN-SUTHERLAND CLIPPING

AIM:

To implement Cohen-Sutherland clipping Algorithm.

ALGORITHM:

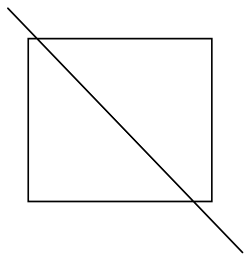
```
Step 1: Create a class sulc with functions drawwindow, drawline, setcode, visibility and
        reset endpoint.
Step 2: Using the function line set the parameters to draw window.
Step 3: Using the function defined in class sulc, setcode is used to save the line inside the
        window and to the line outside the window.
Step 4: Using the function visibility
        i).check the code to know the points inside or outside the window.
        ii).if the code value is zero the point is inside the window.
Step 5: Using the function reset end point
        i). if the code value for the line is outside the window.
        ii).reset the endpoint to the boundary of the window.
Step 6: Initialize the graphics functions
Step 7: Declare the variables x1, x2, y1, y2 of array type.
Step 8: Get the value of two endpoints x1, y1 and x2, y2 to draw the line.
Step 9: Using the object c, display the window before clipping.
Step 10: Using the function setcode, visibility display the clipped window only with lines
         inside the window class was displayed after clipping.
```

INPUT:

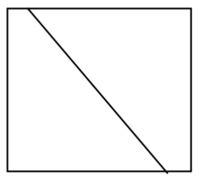
```
Enter the no.of lines:    1
Enter end-point1(x1,y1):30 40
Enter end-point1(x2,y2):300 400
```

OUTPUT:

Before clipping



After clipping



RESULT:

Thus the program is executed and verified.

3D- TRANSFORMATION

AIM:

To perform 3D transformations such as translation, rotation and scaling.

FUNCTIONS USED:

ALGORITHM:

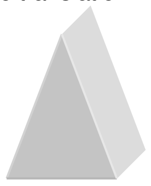
Step 1: Create a class cube with function draw cube.
Step 2: Use the function draw cube to draw a cube using eight points by means of
functions line.
Step 3: Declare the variables x1, y1, x2, y2, x3, y3, in array type which of data type int.

```
    Step 4:Declare the variables theta,op,ch,tx,ty,sx,sy,sz,lz+xy,zf,i,x,y,z.
    Step 5: Initialize graphics functions.
    Step 6: Input the first point in the cube.
    Step 7: Input the size of the edge.
    Step 8: Create an object to call the function.
Step 9: Using switch operation selects the operation to perform translation, rotation,    scaling.
    Step 10: Translation
        a).input the translation vectortx,ty,tz.
        b).calculate points using formula
            x3[i]=x1[i]+tx.
            y3[i]=y1[i]+ty
            z3[i]=z1[i]+tz.
            x4[i]=x3[i]+z3[i]/2
            y4[i]=y3[i]+z3[i]/2
    c).using the function line, display the object before and after translation.
    Step11: Rotation:
        a). input the rotation angle
        b). using formula theta=(theta*3.14)/180
        c).input the direction in x,y,z axis
       d). if the direction is along x axis,
            x3[i]=x1[i].
            y3[i]=y1[i]*cos(theta)-z1[i]*sin(theta),
            y3[i]=y1[i]*sin(theta)-z1[i]*cos(theta),
          if the direction is along yaxis,
            y3[i]=y1[i].
            z3[i]=z1[i]*cos(theta)-x1[i]*sin(theta),
            x3[i]=z1[i]*sin(theta)-x1[i]*cos(theta),
           if the direction is along  z axis,
            z3[i]=z1[i].
            x3[i]=x1[i]*cos(theta)-y1[i]*sin(theta),
            y3[i]=x1[i]*sin(theta)-y1[i]*cos(theta),
        e).calculate the points using the formula
            x4[i]=x3[i]+z3[i]/2
            y4[i]=y3[i]+z3[i]/2
        f). using the function line,display the object before and after rotation.
    Step12: Scaling:
        a).input the scaling factor and reference point
        b).calculate coordinates point using formula
           x3[i]=xf+(x1[i]*sx+xf*(1-sx),
           y3 [i] =yf+ (y1[i]*sy+yf*(1-sy)
           z3 [i] =zf+ (z1[i]*sz+zf*(1-sz)
    c). calculate the points using the formula
        x4[i]=x3[i]+z3[i]/2
        y4[i]=y3[i]+z3[i]/2
    d). using the function line, display the object before and after scaling.
Step13: Stop.
```

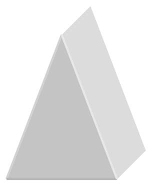
INPUT& OUTPUT:

```
Enter the point in the cube: 100 100 100
Enter the size of the edge: 50

Menu
1. translation
2. rotation
3. scaling
4. exit



Enter the choice:1
Enter the translation vector: 5,10,15
```

Before translation



After translation



RESULT:

Thus the program is executed and verified.

3D Shapes in Open GL

Introduction

All is perfectly fine if we want to draw standard 2D shapes in OpenGL, but the next step up is 3D shapes, and it is these shapes that will really make your application look impressive. The best part about 3D shapes, is that they are made up of 2D shapes, just drawn in 3D space.
While OpenGL provides methods for easily rendering 2D shapes, it doesn't provide any methods for shapes such as cubes, spheres, pyramids, etc. But all is not lost, you have two choices. The first choice is to create the shapes yourself, work out the vertices, determine which vertices you want to use to make faces, and hand code it all in. The next choice, not including 3D modeling applications and model loaders, is to use GLUT for simple 3D shapes. GLUT comes with the ability to render some extremely basic 3D shapes such as a cube, sphere (without texture coordinates), cone, torus/donut and the all famous teapot.
GLUT also lets us render this in both their wireframe version, or their regular filled version.
GLUT also comes with several other shapes, but they are not all that common. Lets take a look at the code required to render these more common shapes:

Cube

```
glutWireCube(double size);
glutSolidCube(double size);
```

This will create a cube with the same width, height and depth/length, each the length of the size parameter specified. This cube in GLUT also comes with surface normals but not texture coordinates.

Sphere

```
glutWireSphere(double radius, int slices, int stacks);
glutSolidSphere(double radius, int slices, int stacks);
```

The calls to create a sphere in GLUT require you to give a radius, which determines the size of the sphere, and the number of stacks and slices. The stacks and slices determine the quality of the sphere, and are the number of divisions in vertical and horizontal directions. The sphere does come with surface normals, but does not come with texture coordinates.

Cone

```
glutWireCone(double radius, double height, int slices, int stacks);
glutSolidCone(double radius, double height, int slices, int stacks);
```

If you want to create a cone, you would use the above GLUT calls. These calls are almost identical to that of the sphere code, we have a radius and the stacks and slices. But it also takes another parameter which defines the height of the cone. Also note that the radius of the cone, refers to the radius of the base of the cone. The cone will come with surface normals, but does not come with texture coordinates.

Torus

```
glutWireTorus(double inner_radius, double outer_radius, int sides,
int rings);
glutSolidTorus(double inner_radius, double outer_radius, int
sides,int rings);
```

A torus looks exactly like a donut. It has an inner radius, which specifies the size of the hole in the middle, an outer radius, which specifies the outer side of the torus from the centre (not the inner radius onwards), the sides specifies the number of sides in each radial section and finally, the rings specify how many radial divisions are used for the torus.

Teapot

```
glutWireTeapot(double size);
glutSolidTeapot(double size);
```

The all famous teapot was first created in 1975 by Martin Newell and is widely used for testing in computer graphics because it is round, has saddle points, can project a shadow onto itself and looks decent when it is untextured. All you have to do to create the teapot is specify the size for the teapot. The teapot comes with surface normals and texture coordinates which means it is perfect for testing bump mapping, environment mapping, and many other effects.