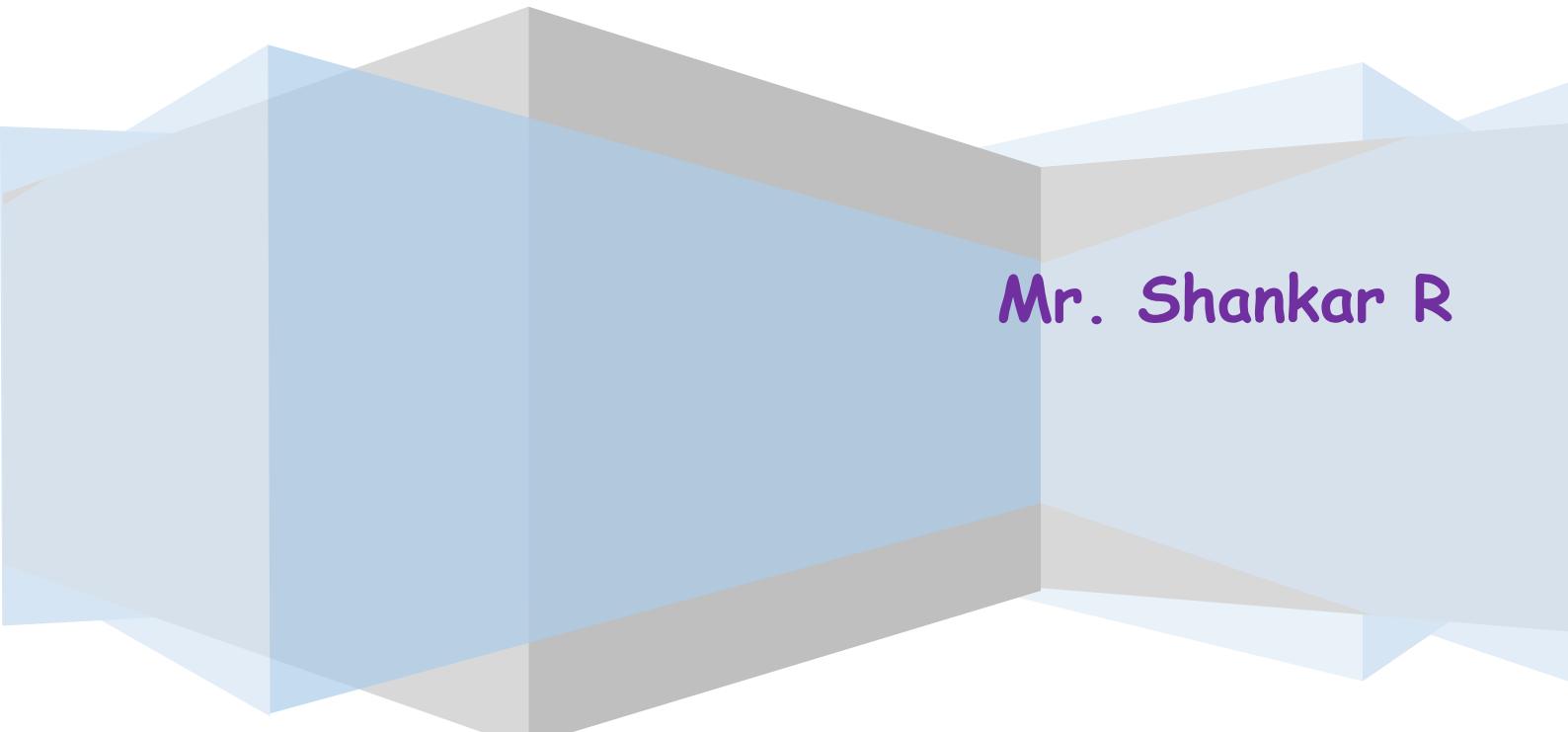


15CSL68 – Computer Graphics Lab Manual

A comprehensive package



Mr. Shankar R

SYLLABUS

PART A

Design, develop, and implement the following programs using OpenGL API

1. Implement Bresenham's line drawing algorithm for all types of slope.
2. Create and rotate a triangle about the origin and a fixed point.
3. Draw a colour cube and spin it using OpenGL transformation matrices.
4. Draw a color cube and allow the user to move the camera suitably to experiment with perspective viewing.
5. Clip a line using Cohen-Sutherland algorithm
6. To draw a simple shaded scene consisting of a tea pot on a table. Define suitably the position and properties of the light source along with the properties of the surfaces of the solid object used in the scene.
7. Design, develop and implement recursively subdivide a tetrahedron to form 3D sierpinski gasket. The number of recursive steps is to be specified by the user.
8. Develop a menu driven program to animate a flag using Bezier Curve algorithm.
9. Develop a menu driven program to fill the polygon using scan line algorithm

PART -B (MINI-PROJECT)

Student should develop mini project on the topics mentioned below or similar applications using OpenGL API. Consider all types of attributes like color, thickness, styles, font, background, speed etc., while doing mini project.

(During the practical exam: the students should demonstrate and answer Viva-Voce)

Sample Topics: Simulation of concepts of OS, Data structures, algorithms etc.

Conduction of Practical Examination:

1. All laboratory experiments from part A are to be included for practical examination.
2. Mini project has to be evaluated for 30 Marks as per 6(b).
3. Report should be prepared in a standard format prescribed for project work.
4. Students are allowed to pick one experiment from the lot.
5. Strictly follow the instructions as printed on the cover page of answer script.
6. Marks distribution: a) Part A: Procedure + Conduction + Viva:10 + 35 +5 =50 Marks b) Part B: Demonstration + Report + Viva voce = 15+10+05 = 30 Marks
7. Change of experiment is allowed only once and marks allotted to the procedure part to be made zero.

INTRODUCTION

Computer graphics are graphics created using computers and, more generally, the representation and manipulation of image data by a computer hardware and software. The development of computer graphics, or simply referred to as CG, has made computers easier to interact with, and better for understanding and interpreting many types of data. Developments in computer graphics have had a profound impact on many types of media and have revolutionized the animation and video game industry. 2D computer graphics are digital images—mostly from two-dimensional models, such as 2D geometric models, text (vector array), and 2D data. 3D computer graphics in contrast to 2D computer graphics are graphics that use a three-dimensional representation of geometric data that is stored in the computer for the purposes of performing calculations and rendering images.

OPEN GL

OpenGL is the most extensively documented 3D graphics API(Application Program Interface) to date. Information regarding OpenGL is all over the Web and in print. It is impossible to exhaustively list all sources of OpenGL information. OpenGL programs are typically written in C and C++. One can also program OpenGL from Delphi (a Pascal-like language), Basic, Fortran, Ada, and other languages. To compile and link OpenGL programs, one will need OpenGL header files. To run OpenGL programs one may need shared or dynamically loaded OpenGL libraries, or a vendor-specific OpenGL Installable Client Driver (ICD).

GLUT

The OpenGL Utility Toolkit (GLUT) is a library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system. Functions performed include window definition, window control, and monitoring of keyboard and mouse input. Routines for drawing a number of geometric primitives (both in solid and wireframe mode) are also provided, including cubes, spheres, and cylinders. GLUT even has some limited support for creating pop-up menus. The two aims of GLUT are to allow the creation of rather portable code between operating systems (GLUT is cross-platform) and to make learning OpenGL easier. All GLUT functions start with the glut prefix (for example, glutPostRedisplay marks the current window as needing to be redrawn).

KEY STAGES IN THE OPENGL RENDERING PIPELINE:

- **Display Lists**

All data, whether it describes geometry or pixels, can be saved in a *display list* for current or later use. (The alternative to retaining data in a display list is processing the data immediately - also known as *immediate mode*.) When a display list is executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode

- **Evaluators**

All geometric primitives are eventually described by vertices. Parametric curves and surfaces may be initially described by control points and polynomial functions called basis functions. Evaluators provide a method to derive the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, texture coordinates, colors, and spatial coordinate values from the control points.

- **Per-Vertex Operations**

For vertex data, next is the "per-vertex operations" stage, which converts the vertices into primitives. Some vertex data (for example, spatial coordinates) are transformed by 4×4 floating-point matrices. Spatial coordinates are projected from a position in the 3D world to a position on your screen. If advanced features are enabled, this stage is even busier. If texturing is used, texture coordinates may be generated and transformed here. If lighting is enabled, the lighting calculations are performed using the transformed vertex, surface normal, light source position, material properties, and other lighting information to produce a color value.

- **Primitive Assembly**

Clipping, a major part of primitive assembly, is the elimination of portions of geometry which fall outside a half-space, defined by a plane. Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending upon how the line or polygon is clipped. In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then viewport and depth (z coordinate) operations are applied. If culling is enabled and the primitive is a polygon, it then may be rejected by a culling test. Depending upon the polygon mode, a polygon may be drawn as points or lines.

The results of this stage are complete geometric primitives, which are the transformed and clipped vertices with related color, depth, and sometimes texture-coordinate values and guidelines for the rasterization step.

- **Pixel Operations**

While geometric data takes one path through the OpenGL rendering pipeline, pixel data takes a different route. Pixels from an array in system memory are first unpacked from one of a variety of formats into the proper number of components. Next the data is scaled, biased, and processed by a pixel map. The results are clamped and then either written into texture memory or sent to the rasterization step. If pixel data is read from the frame buffer, pixel-transfer operations (scale, bias, mapping, and clamping) are performed. Then these results are packed into an appropriate format and returned to an array in system memory.

There are special pixel copy operations to copy data in the framebuffer to other parts of the framebuffer or to the texture memory. A single pass is made through the pixel transfer operations before the data is written to the texture memory or back to the framebuffer.

- **Texture Assembly**

An OpenGL application may wish to apply texture images onto geometric objects to make them look more realistic. If several texture images are used, it's wise to put them into texture objects so that you can easily switch among them.

Some OpenGL implementations may have special resources to accelerate texture performance. There may be specialized, high-performance texture memory. If this memory is available, the texture objects may be prioritized to control the use of this limited and valuable resource.

- **Rasterization**

Rasterization is the conversion of both geometric and pixel data into *fragments*. Each fragment square corresponds to a pixel in the framebuffer. Line and polygon stipples, line width, point size, shading model, and coverage calculations to support antialiasing are taken into consideration as vertices are connected into lines or the interior pixels are calculated for a filled polygon. Color and depth values are assigned for each fragment square.

- **Fragment Operations**

Before values are actually stored into the framebuffer, a series of operations are performed that may alter or even throw out fragments. All these operations can be enabled or disabled.

The first operation which may be encountered is texturing, where a texel (texture element) is generated from texture memory for each fragment and applied to the fragment. Then fog calculations may be applied, followed by the scissor test, the alpha test, the stencil test, and the depth-buffer test (the depth buffer is for hidden-surface removal). Failing an enabled test may end the continued processing of a fragment's square. Then, blending, dithering, logical operation, and masking by a bitmask may be performed. Finally, the thoroughly processed fragment is drawn into the appropriate buffer, where it has finally advanced to be a pixel and achieved its final resting place.

- **OpenGL-Related Libraries**

OpenGL provides a powerful but primitive set of rendering commands, and all higher-level drawing must be done in terms of these commands. Also, OpenGL programs have to use the underlying mechanisms of the windowing system. A number of libraries exist to allow you to simplify your programming tasks, including the following:

The OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections, performing polygon tessellation, and rendering surfaces. This library is provided as part of every OpenGL implementation. Portions of the GLU are described in the *OpenGL*

For every window system, there is a library that extends the functionality of that window system to support OpenGL rendering. For machines that use the X Window System, the OpenGL Extension to the X Window System (GLX) is provided as an adjunct to OpenGL. GLX routines use the prefix glX. For Microsoft Windows, the WGL routines provide the Windows to OpenGL interface. All WGL routines use the prefix wgl. For IBM OS/2, the PGL is the Presentation Manager to OpenGL interface, and its routines use the prefix pgl.

The OpenGL Utility Toolkit (GLUT) is a window system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window system APIs. Open Inventor is an object-oriented toolkit based on OpenGL which provides objects and methods for creating interactive three-dimensional graphics applications. Open Inventor, which is written in C++, provides prebuilt objects and a built-in event model for user interaction, high-level application components for creating and editing three-dimensional scenes, and the ability to print objects and exchange data in other graphics formats. Open Inventor is separate from OpenGL.

- **GLUT, the OpenGL Utility Toolkit**

As you know, OpenGL contains rendering commands but is designed to be independent of any window system or operating system. Consequently, it contains no commands for opening windows or reading events from the keyboard or mouse. Unfortunately, it's impossible to write a complete graphics program without at least opening a window, and most interesting programs require a bit of user input or other services from the operating system or window system. In many cases, complete programs make the most interesting examples, so this book uses GLUT to simplify opening windows, detecting input, and so on. If you have an implementation of OpenGL and GLUT on your system, the examples in this book should run without change when linked with them.

In addition, since OpenGL drawing commands are limited to those that generate simple geometric primitives (points, lines, and polygons), GLUT includes several routines that create more complicated three-dimensional objects such as a sphere, a torus, and a teapot. This way, snapshots of program output can be interesting to look at. (Note that the OpenGL Utility Library, GLU, also has quadrics routines that create some of the same three-dimensional objects as GLUT, such as a sphere, cylinder, or cone.)

GLUT may not be satisfactory for full-featured OpenGL applications, but you may find it a useful starting point for learning OpenGL. The rest of this section briefly describes a small subset of GLUT routines so that you can follow the programming examples in the rest of this book.

OBJECTIVE AND APPLICATION OF THE LAB

The objective of this lab is to give students hands on learning exposure to understand and apply computer graphics with real world problems. The lab gives the direct experience to Visual Basic Integrated Development Environment (IDE) and GLUT toolkit. The students get a real world exposure to Windows programming API. Applications of this lab are profoundly felt in gaming industry, animation industry and Medical Image Processing Industry. The materials learned here will be useful in Programming at the Software Industry.

Setting up GLUT - main()

GLUT provides high-level utilities to simplify OpenGL programming, especially in interacting with the Operating System (such as creating a window, handling key and mouse inputs). The following GLUT functions were used in the above program:

- **glutInit:** initializes GLUT, must be called before other GL/GLUT functions. It takes the same arguments as the main().

```
void glutInit(int *argc, char **argv)
```

- **glutCreateWindow:** creates a window with the given title.

```
int glutCreateWindow(char *title)
```

- **glutInitWindowSize:** specifies the initial window width and height, in pixels.

```
void glutInitWindowSize(int width, int height)
```

- **glutInitWindowPosition:** positions the top-left corner of the initial window at (x, y). The coordinates (x, y), in term of pixels, is measured in window coordinates, i.e., origin (0, 0) is at the top-left corner of the screen; x-axis pointing right and y-axis pointing down.

```
void glutInitWindowPosition(int x, int y)
```

- **glutDisplayFunc:** registers the callback function (or event handler) for handling window-paint event. The OpenGL graphic system calls back this handler when it receives a window re-paint request. In the example, we register the function display() as the handler.

```
void glutDisplayFunc(void (*func)(void))
```

- **glutMainLoop:** enters the infinite event-processing loop, i.e, put the OpenGL graphics system to wait for events (such as re-paint), and trigger respective event handlers (such as display()).

```
void glutMainLoop()
```

- **glutInitDisplayMode:** requests a display with the specified mode, such as color mode (GLUT_RGB, GLUT_RGBA, GLUT_INDEX), single/double buffering (GLUT_SINGLE, GLUT_DOUBLE), enable depth (GLUT_DEPTH), joined with a bit OR '|'.

void **glutInitDisplayMode**(unsigned int displayMode)

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

The **glMatrixMode** function specifies which matrix is the current matrix.

- **void**

glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar);

The **glOrtho** function multiplies the current matrix by an orthographic matrix.

- **void glPointSize (GLfloat size);**

The **glPointSize** function specifies the diameter of rasterized points.

- **void glutPostRedisplay(void);**

glutPostRedisplay marks the current window as needing to be redisplayed.

- **void glPushMatrix (void);**

void glPopMatrix (void);

The **glPushMatrix** and **glPopMatrix** functions push and pop the current matrix stack.

- **GLint glRenderMode (GLenum mode);**

The **glRenderMode** function sets the rasterization mode.

- **void glRotatef (GLfloat angle, GLfloat x, GLfloat y, GLfloat z);**

The **glRotatef** functions multiply the current matrix by a rotation matrix.

- **void glScalef (GLfloat x, GLfloat y, GLfloat z);**

The **glScalef** functions multiply the current matrix by a general scaling matrix.

- **void glTranslatef (GLfloat x, GLfloat y, GLfloat z);**

The **glTranslatef** functions multiply the current matrix by a translation matrix.

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

The **glViewport** function sets the viewport.

- **void glEnable, glDisable();**

The **glEnable** and **glDisable** functions enable or disable OpenGL capabilities.

- **glutBitmapCharacter();**

The **glutBitmapCharacter** function used for font style.

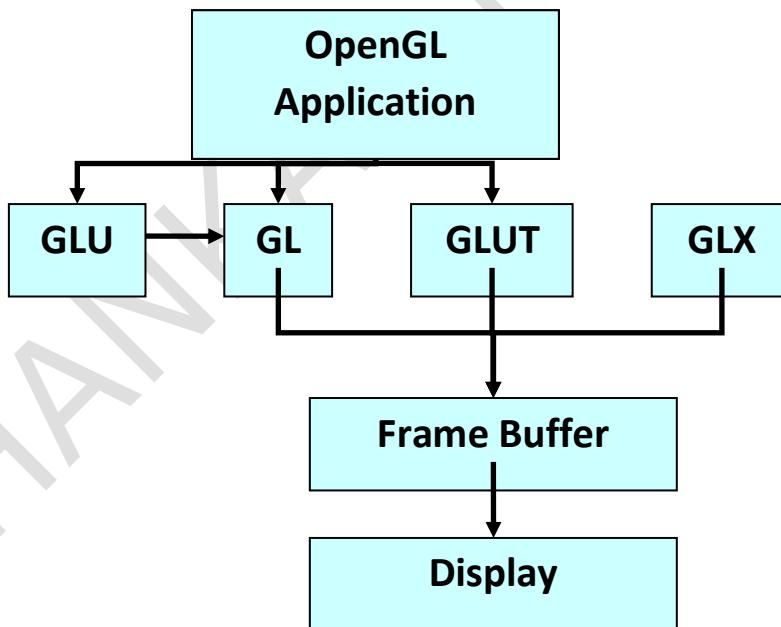
OpenGL Primitives

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

INTRODUCTION TO openGL

OpenGL is an API. OpenGL is nothing more than a set of functions you call from your program (think of as collection of .h files).

OpenGL Libraries:



OpenGL Hierarchy:

- ❖ Several levels of abstraction are provided
- ❖ GL
 - Lowest level: vertex, matrix manipulation
 - glVertex3f(point.x, point.y, point.z)
- ❖ GLU

- Helper functions for shapes, transformations
- gluPerspective(fovy, aspect, near, far)
- gluLookAt(0, 0, 10, 0, 0, 0, 0, 1, 0);
- ◆ GLUT
 - Highest level: Window and interface management
 - glutSwapBuffers()
 - glutInitWindowSize (500, 500);

OpenGL Implementations :

- ◆ OpenGL IS an API (think of as collection of .h files):
 - #include <GL/gl.h>
 - #include <GL/glu.h>
 - #include <GL/glut.h>
- ◆ Windows, Linux, UNIX, etc. all provide a platform specific implementation.
- ◆ Windows: opengl32.lib glu32.lib glut32.lib
- ◆ Linux: -l GL -l GLUT

Event Loop:

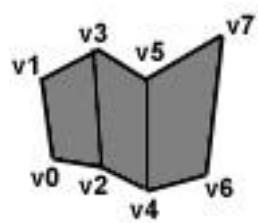
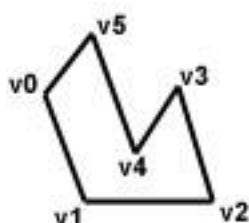
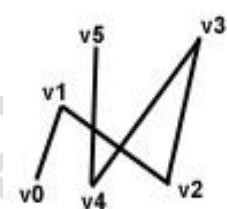
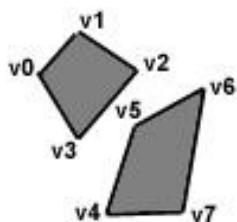
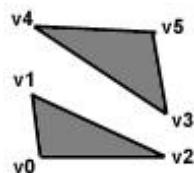
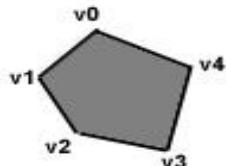
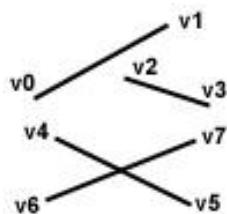
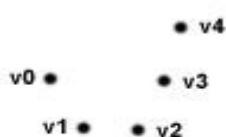
- OpenGL programs often run in an event loop:
 - Start the program
 - Run some initialization code
 - Run an infinite loop and wait for events such as
 - Key press
 - Mouse move, click
 - Reshape window
 - Expose event

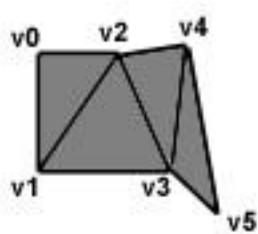
OpenGL Command Syntax (1) :

- OpenGL commands start with “gl”
- OpenGL constants start with “GL_”
- Some commands end in a number and one, two or three letters at the end (indicating number and type of arguments)
- A Number indicates number of arguments
- Characters indicate type of argument

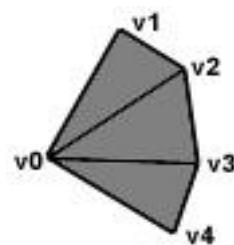
OpenGL Command Syntax (2)

- ‘f’ float
- ‘d’ double float
- ‘s’ signed short integer
- ‘i’ signed integer
- ‘b’ character
- ‘ub’ unsigned character
- ‘us’ unsigned short integer
- ‘ui’ unsigned integer

Ten gl Primitives:



`GL_TRIANGLE_STRIP`

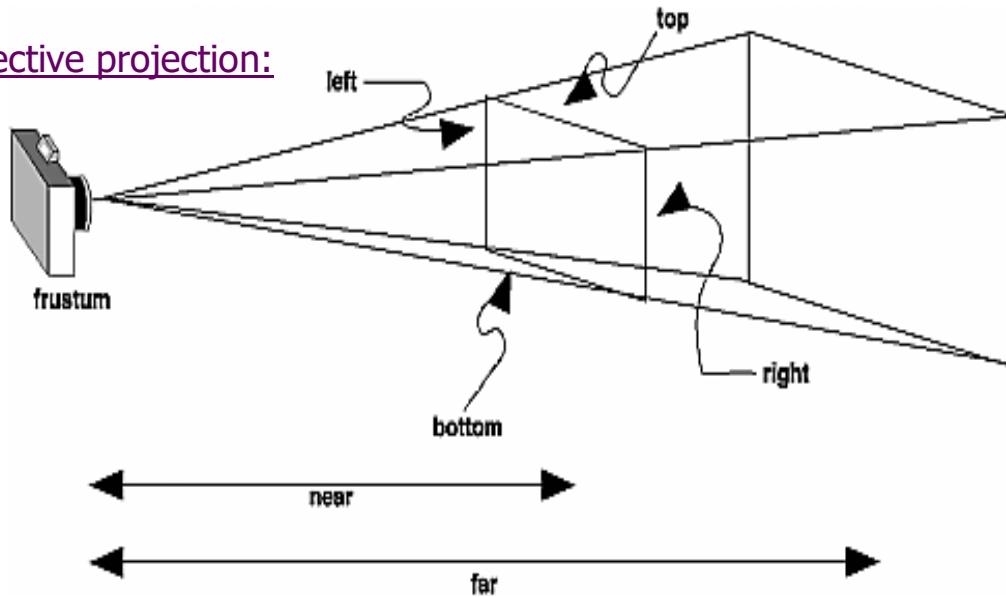


`GL_TRIANGLE_FAN`

SHANKARR, BMSIT&M

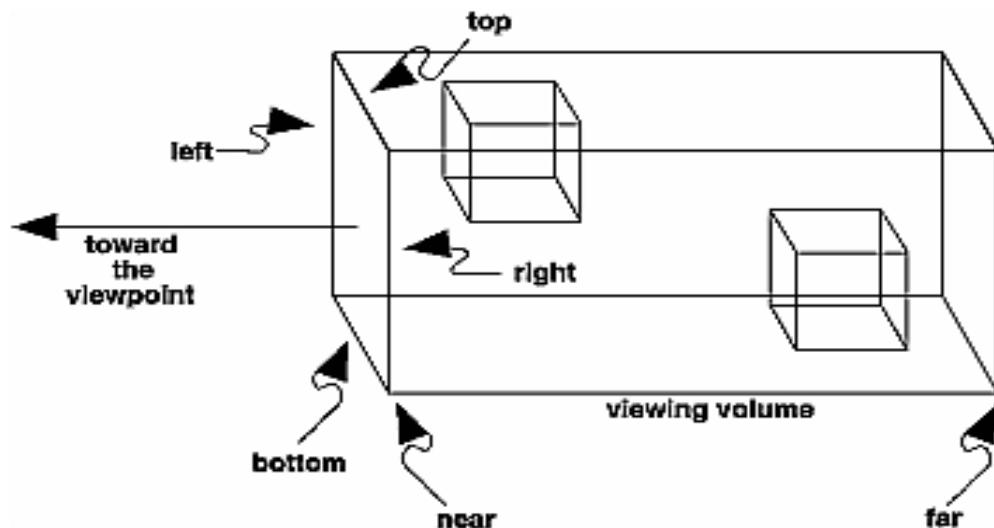
Projections in OpenGL

Perspective projection:



```
void glFrustum(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble near, GLdouble far);
```

Orthographic projection:



```
void glOrtho(GLdouble left, GLdouble right, GLdouble bottom,  
GLdouble top, GLdouble near, GLdouble far)
```

Viva questions and answers

1. What is Computer Graphics? Answer: Computer graphics are graphics created using computers and, more generally, the representation and manipulation of image data by a computer.
2. What is OpenGL? Answer: OpenGL is the most extensively documented 3D graphics API (Application Program Interface) to date. It is used to create Graphics.
3. What is GLUT? Answer: The OpenGL Utility Toolkit (GLUT) is a library of utilities for OpenGL programs, which primarily perform system-level I/O with the host operating system.
4. What are the applications of Computer Graphics? Answer: Gaming Industry, Animation Industry and Medical Image Processing Industries. The sum total of these industries is a Multi Billion Dollar Market. Jobs will continue to increase in this arena in the future.
5. Explain in brief 3D Sierpinski gasket? Answer: The Sierpinski triangle (also with the original orthography Sierpinski), also called the Sierpinski gasket or the Sierpinski Sieve, is a fractal named after the Polish mathematician Waclaw Sierpinski who described it in 1915. Originally constructed as a curve, this is one of the basic examples of self-similar sets, i.e. it is a mathematically generated pattern that can be reproducible at any magnification or reduction.
6. What is Liang-Barsky line clipping algorithm? Answer: In computer graphics, the Liang-Barsky algorithm is a line clipping algorithm. The Liang-Barsky algorithm uses the parametric equation of a line and inequalities describing the range of the clipping box to determine the intersections between the line and the clipping box. With these intersections it knows which portion of the line should be drawn.
7. Explain in brief Cohen-Sutherland line-clipping algorithm? Answer: The Cohen-Sutherland line clipping algorithm quickly detects and dispenses with two common and trivial cases. To clip a line, we need to consider only its endpoints. If both endpoints of a line lie inside the window, the entire line lies inside the window. It is trivially accepted and needs no clipping. On the other hand, if both endpoints of a line lie entirely to one side of the window, the line must lie entirely outside of the window. It is trivially rejected and needs to be neither clipped nor displayed.
8. Explain in brief scan-line area filling algorithm? Answer: The scanline fill algorithm is an ingenious way of filling in irregular polygons. The algorithm begins with a set of points. Each point is connected to the next, and the line between them is considered to be an edge of the polygon. The points of each edge are adjusted to ensure that the point with the smaller y value appears first. Next, a data structure is created that contains a list of edges that begin on each scanline of the image. The program progresses from the first scanline upward. For each line, any pixels that contain an intersection between this scanline and an edge of the polygon are filled in. Then, the algorithm progresses along the scanline, turning on when it reaches a polygon pixel and turning off when it reaches another one, all the way across the scanline.
9. Explain Midpoint Line algorithm Answer: The Midpoint line algorithm is an algorithm which determines which points in an n-dimensional raster should be plotted in order to form a close approximation to a straight line between two given points. It is commonly used to draw lines on a computer screen, as it uses only integer addition, subtraction and bit shifting, all of which are very cheap operations in standard computer architectures.
10. What is a Pixel? Answer: In digital imaging, a pixel (or picture element) is a single point in a raster image. The Pixel is the smallest addressable screen element; it is the smallest unit of picture which can be controlled. Each Pixel has its address. The address of Pixels corresponds to its coordinate. Pixels are normally arranged in a 2-dimensional grid, and are often represented using dots or squares.
11. What is Graphical User Interface? Answer: A graphical user interface (GUI) is a type of user interface item that allows people to interact with programs in more ways than typing such as computers; hand-held devices such as MP3 Players, Portable Media Players or Gaming devices; household appliances and office equipment with images rather than text commands.

12. What is the general form of an OpenGL program? Answer: There are no hard and fast rules. The following pseudocode is generally recognized as good OpenGL form.

```

program entrypoint { // Determine which depth or pixel
format should be used. // Create a window with the desired format. // Create a rendering context and make it current
with the window. // Set up initial OpenGL state. // Set up callback routines for window resize and window refresh. }
handle resize { glViewport(...); glMatrixMode(GL PROJECTION); glLoadIdentity(); // Set projection transform with
glOrtho, glFrustum, gluOrtho2D, gluPerspective, etc. } handle refresh { glClear(...); glMatrixMode(GL MODELVIEW);
glLoadIdentity(); // Set view transform with gluLookAt or equivalent // For each object (i) in the scene that needs to be
rendered: // Push relevant stacks, e.g., glPushMatrix, glPushAttrib. // Set OpenGL state specific to object (i). // Set
model transform for object (i) using glTranslatef, glScalef, glRotatef, and/or equivalent. // Issue rendering commands
for object (i). // Pop relevant stacks, (e.g., glPopMatrix, glPopAttrib.) // End for loop. // Swap buffers. }

```

13. What support for OpenGL does OpenNet, FreeBSD or Linux provide? Answer: The X Windows implementation, XFree86 4.0, includes support for OpenGL using Mesa or the OpenGL Sample Implementation. XFree86 is released under the XFree86 license. <http://www.xfree86.org/>

14. What is the AUX library? Answer: The AUX library was developed by SGI early in OpenGL's life to ease creation of small OpenGL demonstration programs. It's currently neither supported nor maintained. Developing OpenGL programs using AUX is strongly discouraged. Use the GLUT instead. It's more flexible and powerful and is available on a wide range of platforms. Very important: Don't use AUX. Use GLUT instead.

15. How does the camera work in OpenGL? Answer: As far as OpenGL is concerned, there is no camera. More specifically, the camera is always located at the eye space coordinate (0., 0., 0.). To give the appearance of moving the camera, your OpenGL application must move the scene with the inverse of the camera transformation.

16. How do I implement a zoom operation? Answer: A simple method for zooming is to use a uniform scale on the ModelView matrix. However, this often results in clipping by the zNear and zFar clipping planes if the model is scaled too large. A better method is to restrict the width and height of the view volume in the Projection matrix.

17. What are OpenGL coordinate units? Answer: Depending on the contents of your geometry database, it may be convenient for your application to treat one OpenGL coordinate unit as being equal to one millimeter or one parsec or anything in between (or larger or smaller). OpenGL also lets you specify your geometry with coordinates of differing values. For example, you may find it convenient to model an airplane's controls in centimeters, its fuselage in meters, and a world to fly around in kilometers. OpenGL's ModelView matrix can then scale these different coordinate systems into the same eye coordinate space. It's the application's responsibility to ensure that the Projection and ModelView matrices are constructed to provide an image that keeps the viewer at an appropriate distance, with an appropriate field of view, and keeps the zNear and zFar clipping planes at an appropriate range. An application that displays molecules in micron scale, for example, would probably not want to place the viewer at a distance of 10 feet with a 60 degree field of view.

18. What is Microsoft Visual Studio? Answer: Microsoft Visual Studio is an integrated development environment (IDE) for developing windows applications. It is the most popular IDE for developing windows applications or windows based software.

19. What does the .gl or .GL file format have to do with OpenGL? Answer: .gl files have nothing to do with OpenGL, but are sometimes confused with it. .gl is a file format for images, which has no relationship to OpenGL.

20. Who needs to license OpenGL? Who doesn't? Is OpenGL free software? Answer: Companies which will be creating or selling binaries of the OpenGL library will need to license OpenGL. Typical examples of licensees include hardware vendors, such as Digital Equipment, and IBM who would distribute OpenGL with the system software on their workstations or PCs. Also, some software vendors, such as Portable Graphics and Template Graphics, have a business in creating and distributing versions of OpenGL, and they need to license OpenGL. Applications developers do NOT need to license OpenGL. If a developer wants to use OpenGL that developer needs to obtain copies of a linkable OpenGL library for a particular machine. Those OpenGL libraries may be bundled in with the development and/or run-time options or may be purchased from a third-party software vendor, without licensing the source code or use of the OpenGL trademark.

21. How do we make shadows in OpenGL? Answer: There are no individual routines to control neither shadows nor an OpenGL state for shadows. However, code can be written to render shadows.

22. What is the use of glutInit? Answer: void glutInit(int *argc, char **argv); 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.

23. Describe the usage of glutInitWindowSize and glutInitWindowPosition? Answer: void glutInitWindowSize(int width, int height); void glutInitWindowPosition(int x, int y); Windows created by glutCreateWindow will be requested to be created with the current initial window position and size. The intent of the initial window position and size values is

to provide a suggestion to the window system for a window's initial size and position. The window system is not obligated to use this information. Therefore, GLUT programs should not assume the window was created at the specified size or position. A GLUT program should use the window's reshape callback to determine the true size of the window.

24. Describe the usage of glutMainLoop? Answer: void glutMainLoop(void); glutMainLoop enters the GLUT event processing loop. This routine should be called at most once in a GLUT program. Once called, this routine will never return. It will call as necessary any callbacks that have been registered.

SHANKAR, BMST&M

1. Implement Bresenham's Line drawing algorithm for all types of slope.

```
#include<GL/glut.h>
#include<stdio.h>
int x1, y1, x2, y2;

void draw_pixel(int x, int y)
{
    glColor3f(1.0,0.0,0.0);
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
}

void bresenham_line_draw(int x1, int y1, int x2, int y2)
{
    int dx = x2 - x1;                      // x difference
    int dy = y2 - y1;                      // y difference
    int m = dy/dx;                         // slope

    if (m < 1)
    {
        int decision_parameter = 2*dy - dx;
        int x = x1;                          // initial x
        int y = y1;                          // initial y
        if (dx < 0)                         // decide the first point and second point
        {
            x = x2;                          // making second point as first point
            y = y2;
            x2 = x1;
        }
        draw_pixel (x, y);                  // plot a point
        while (x < x2)                    // from 1st point to 2nd point
        {
            if (decision_parameter >= 0)
            {
                x = x+1;
                y = y+1;
                decision_parameter = decision_parameter + 2*dy - 2*dx * (y+1 - y);
            }
            else
            {
                x = x+1;
                y = y;
                decision_parameter = decision_parameter + 2*dy - 2*dx * (y - y);
            }
            draw_pixel (x, y);
        }
    }
}
```

```

else if (m > 1)
{
    int decision_parameter = 2*dx - dy;
    int x = x1;                      // initial x
    int y = y1;                      // initial y
    if (dy < 0)
    {
        x = x2;
        y = y2;
        y2 = y1;
    }
    draw_pixel (x, y);
    while (y < y2)
    {
        if (decision_parameter >= 0)
        {
            x = x+1;
            y = y+1;
            decision_parameter = decision_parameter + 2*dx - 2*dy * (x+1 - x);
        }
        else
        {
            y = y+1;
            x = x;
            decision_parameter = decision_parameter + 2*dx - 2*dy * (x - x);
        }
        draw_pixel(x, y);
    }
}

else if (m == 1)
{
    int x = x1;
    int y = y1;
    draw_pixel (x, y);
    while (x < x2)
    {
        x = x+1;
        y = y+1;
        draw_pixel (x, y);
    }
}

```

```

void init()
{
    glClearColor(1,1,1,1);
    gluOrtho2D(0.0, 500.0, 0.0, 500.0); // left ->0, right ->500, bottom ->0, top ->500
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    bresenhams_line_draw(x1, y1, x2, y2);
    glFlush();
}

int main(int argc, char **argv)
{
    printf("Enter Start Points (x1,y1)\n");
    scanf("%d %d", &x1, &y1); // 1st point from user

    printf("Enter End Points (x2,y2)\n");
    scanf("%d %d", &x2, &y2); // 2nd point from user

    glutInit(&argc, argv); // initialize graphics system
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB); //single buffered mode with RGB colour variants
    glutInitWindowSize(500, 500); // 500 by 500 window size
    glutInitWindowPosition(220, 200); // where do you wanna see your window
    glutCreateWindow("Bresenham's Line Drawing"); // the title of your window

    init(); // initialize the canvas

    glutDisplayFunc(display); // call display function

    glutMainLoop(); // run forever
}

```

OUTPUT

Case 1: m < 1

 "C:\Users\Shankara\Dropbox\CG\Lab Final\temp\1_line\bin\Debug\1_line.exe"

```

Enter Start Points (x1,y1)
0
0
Enter End Points (x2,y2)
300
100

```



Case 2: $m > 1$

"C:\Users\Shankara\Dropbox\CG\Lab Final\temp\1_line\bin\Debug\1_line.exe"

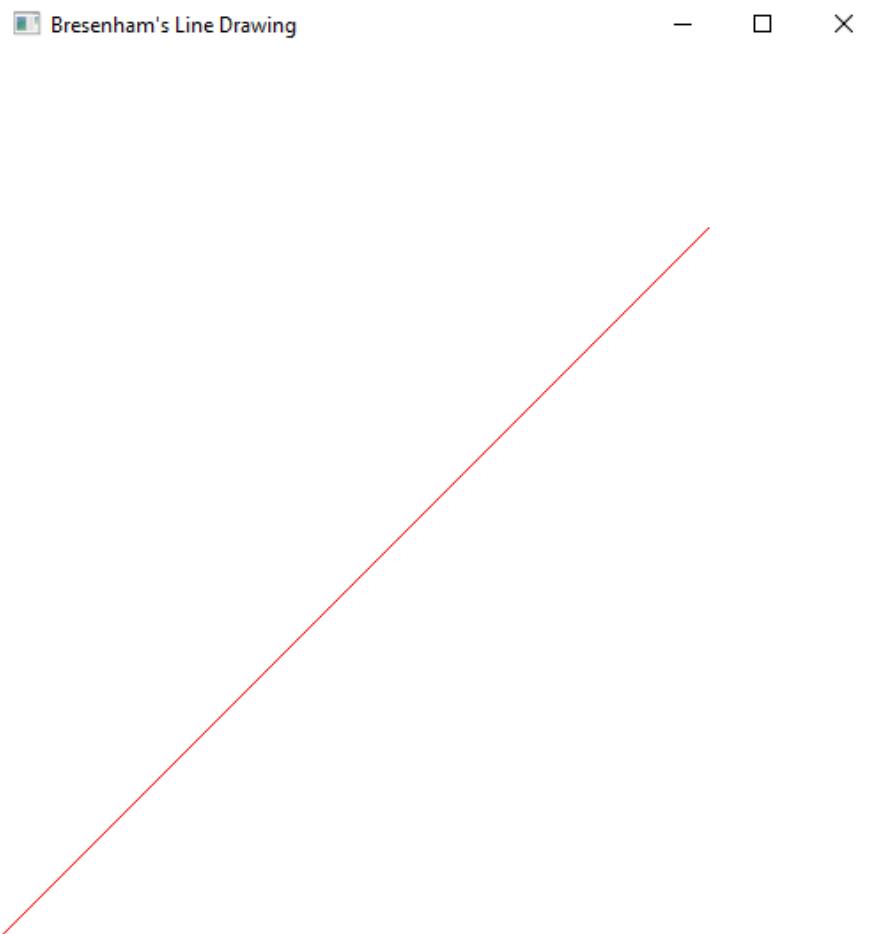
```
Enter Start Points (x1,y1)  
0  
0  
Enter End Points (x2,y2)  
50  
400
```



Case 3: $m == 1$

```
[C:\Users\Shankara\Dropbox\CG\Lab Final\temp\1_line\bin\Debug\1_line.exe]
Enter Start Points (x1,y1)
0
0
Enter End Points (x2,y2)
400
400
```

Bresenham's Line Drawing



2. Create and rotate a triangle about the origin and a fixed point.

```
#include<GL/glut.h>
#include<stdio.h>
int x,y;
int where_to_rotate=0;           // don't rotate initially
float rotate_angle=0;           // initial angle
float translate_x=0,translate_y=0; // initial translation

void draw_pixel(float x1, float y1)
{
    glPointSize(5);
    glBegin(GL_POINTS);
        glVertex2f(x1,y1);      // plot a single point
    glEnd();
}

void triangle(int x, int y)
{
    glColor3f(1,0,0);
    glBegin(GL_POLYGON);    // drawing a Triangle
        glVertex2f(x,y);
        glVertex2f(x+400,y+300);
        glVertex2f(x+300,y+0);
    glEnd();
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glLoadIdentity();

    glColor3f(1,1,1);          // mark origin point as white dot
    draw_pixel(0,0);           // plot origin - white colour

    if (where_to_rotate == 1)   // rotate around origin
    {
        translate_x = 0;        // no translation for rotation around origin
        translate_y = 0;
        rotate_angle += 1;       // the amount of rotation angle
    }
    if (where_to_rotate == 2)   // rotate around Fixed Point
    {
        translate_x = x;        // SET the translation to wherever the customer says
        translate_y = y;
        rotate_angle += 1;       // the amount of rotation angle
        glColor3f(0,0,1);        // mark the customer coordinate as blue dot
        draw_pixel(x,y);         // plot the customer coordinate - blue colour
    }
}
```

```

glTranslatef(translate_x, translate_y, 0); // ACTUAL translation +ve
glRotatef(rotate_angle, 0, 0, 1); // rotate
glTranslatef(-translate_x, -translate_y, 0); // ACTUAL translation -ve

triangle(translate_x,translate_y); // what to rotate? - TRIANGLE boss

glutPostRedisplay(); // call display function again and again
glutSwapBuffers(); // show the output
}

void init()
{
    glClearColor(0,0,0,1); //setting to black
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(-800, 800, -800, 800);
    glMatrixMode(GL_MODELVIEW);
}

void rotateMenu (int option)
{
    if(option==1)
        where_to_rotate=1; // rotate around origin

    if(option==2)
        where_to_rotate=2; // rotate around customer's coordinates

    if(option==3)
        where_to_rotate=3; // stop rotation
}

int main(int argc, char **argv)
{
    printf("Enter Fixed Points (x,y) for Rotation: \n");
    scanf("%d %d", &x, &y); // getting the user's coordinates to rotate

    glutInit(&argc, argv); // initialize the graphics system
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB); // SINGLE also works
    glutInitWindowSize(800, 800); // 800 by 800 size..you can change it
    glutInitWindowPosition(0, 0); // where do you wanna see your window
    glutCreateWindow("Create and Rotate Triangle"); // title

    init(); // initialize the canvas

    glutDisplayFunc(display); // call display function

    glutCreateMenu(rotateMenu); // menu items
    glutAddMenuEntry("Rotate around ORIGIN",1);
    glutAddMenuEntry("Rotate around FIXED POINT",2);
}

```

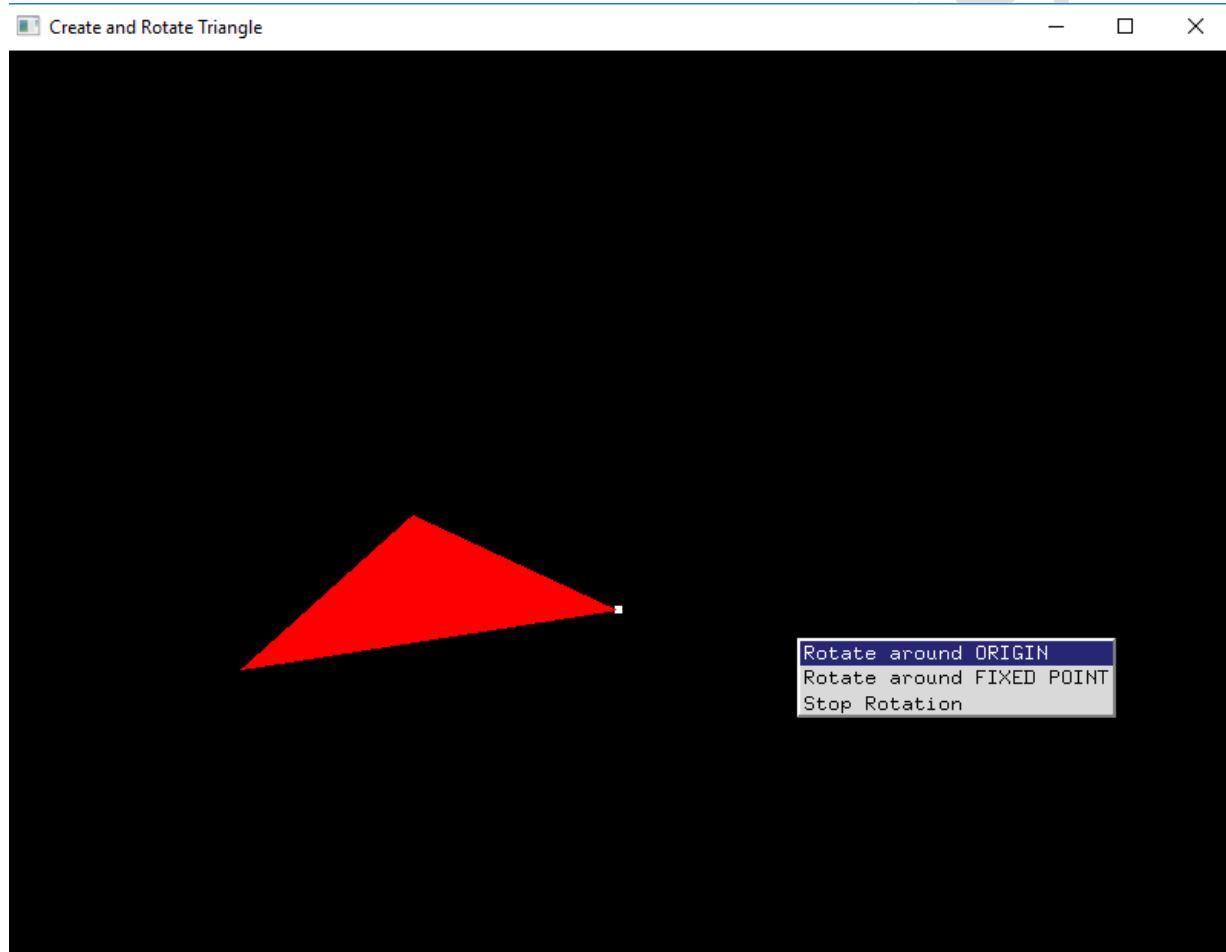
```
glutAddMenuEntry("Stop Rotation",3);
glutAttachMenu(GLUT_RIGHT_BUTTON);

glutMainLoop(); // run forever
}
```

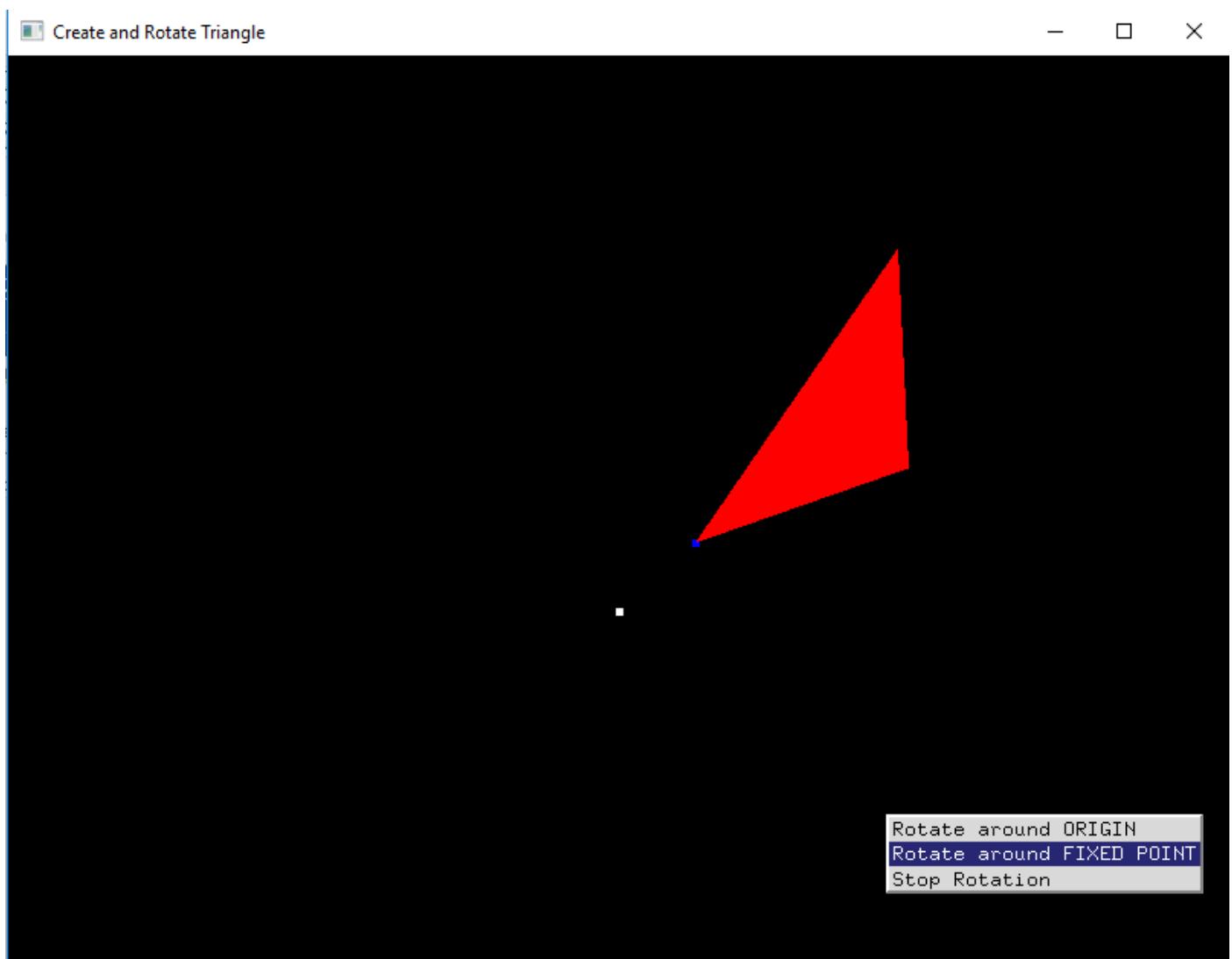
OUTPUT

```
[C:\Users\Shankara\Dropbox\CG\Lab Final\temp\2\bin\Debug\2.exe]
Enter Fixed Points (x,y) for Rotation:
100
100
```

if we select "Rotate around ORIGIN" option



if we select “Rotate around FIXED POINT” option



3. Program to draw a color cube and spin it using OpenGL transformation matrices.

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

```
GLfloat vertices[] = { -1, -1, -1,
                      1, -1, -1,
                      1, 1, -1,
                      -1, 1, -1,
                      -1, -1, 1,
                      1, -1, 1,
                      1, 1, 1,
                      -1, 1, 1
};

};
```

```
GLfloat colors[] = { 0, 0, 0, // white color
                     1, 0, 0, // red color .. so on for eight faces of cube
                     1, 1, 0,
                     0, 1, 0,
                     0, 0, 1,
                     1, 0, 1,
                     1, 1, 1,
                     0, 1, 1
};

};
```

```
GLubyte cubeIndices[] = {0, 3, 2, 1,
                        2, 3, 7, 6,
                        0, 4, 7, 3,
                        1, 2, 6, 5,
                        4, 5, 6, 7,
                        0, 1, 5, 4
};

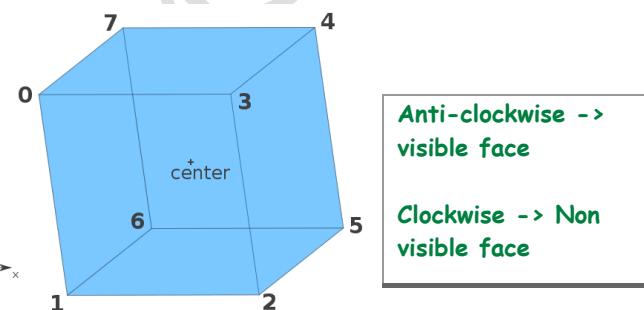
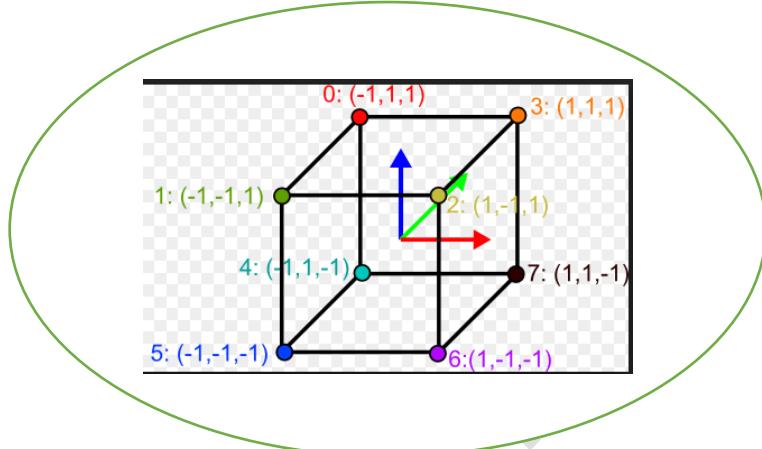
};
```

```
static GLfloat theta[] = {0, 0, 0}; // initial angles
static GLint axis=2; // let us assume the right mouse button has been clicked initially
```

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

    glRotatef (theta[0], 1, 0, 0); // first angle rotation via x axis
    glRotatef (theta[1], 0, 1, 0); // second angle rotation via y axis
    glRotatef (theta[2], 0, 0, 1); // third angle rotation via z axis

    glDrawElements(GL_QUADS,24,GL_UNSIGNED_BYTE,cubeIndices); // draw the cube
    glutSwapBuffers(); // show the output
}
```



glDrawElements() draws a sequence of primitives by hopping around vertex arrays with the associated array indices. It reduces both the number of function calls and the number of vertices to transfer. **glDrawElements()** requires 4 parameters. The first one is the type of primitive, the second is the number of indices of index array, the third is data type of index array and the last parameter is the address of index array.

```

void spinCube()
{
    theta[axis] += 2; // rotate every 2 degrees

    if (theta[axis] > 360) // if the rotation angle crosses 360 degrees, make it 0 degree
        theta[axis] -= 360;

    glutPostRedisplay(); // call display again
}

void mouse(int btn, int state, int x, int y)
{
    if (btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
        axis=0; // x axis rotation

    if (btn==GLUT_MIDDLE_BUTTON && state==GLUT_DOWN)
        axis=1; // y axis rotation

    if (btn==GLUT_RIGHT_BUTTON && state==GLUT_DOWN)
        axis=2; // z axis rotation
}

void myReshape(int w, int h)
{
    glViewport(0,0,w,h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if(w<=h)
        glOrtho (-2, 2, -2*(GLfloat)h/(GLfloat)w, 2*(GLfloat)h / (GLfloat)w, -10, 10);
    else
        glOrtho (-2*(GLfloat)w/(GLfloat)h, 2*(GLfloat)w / (GLfloat)h, -2, 2, -10, 10);

    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Spin a color cube");

    glutReshapeFunc(myReshape); // calls myReshape whenever we change the window size

    glutDisplayFunc(display); // call display function

    glutIdleFunc(spinCube); // whenever we are idle, calls spinCube function
}

```

Maintaining the ASPECT RATIO,
i.e., whenever we change the
window size, our output should
remain same, not distorted

```

glutMouseFunc(mouse);           // calls mouse function whenever we interact with mouse

glEnable(GL_DEPTH_TEST);       // enables depth - for 3D

glEnableClientState(GL_COLOR_ARRAY);    // enables colour and vertex properties
glEnableClientState(GL_VERTEX_ARRAY);

glVertexPointer(3, GL_FLOAT, 0, vertices); // glVertexPointer(size,type,stride,pointer)
glColorPointer(3, GL_FLOAT, 0, colors);    // glColorPointer(size,type,stride,pointer)

glColor3f(1, 1, 1);
glutMainLoop();
}

```

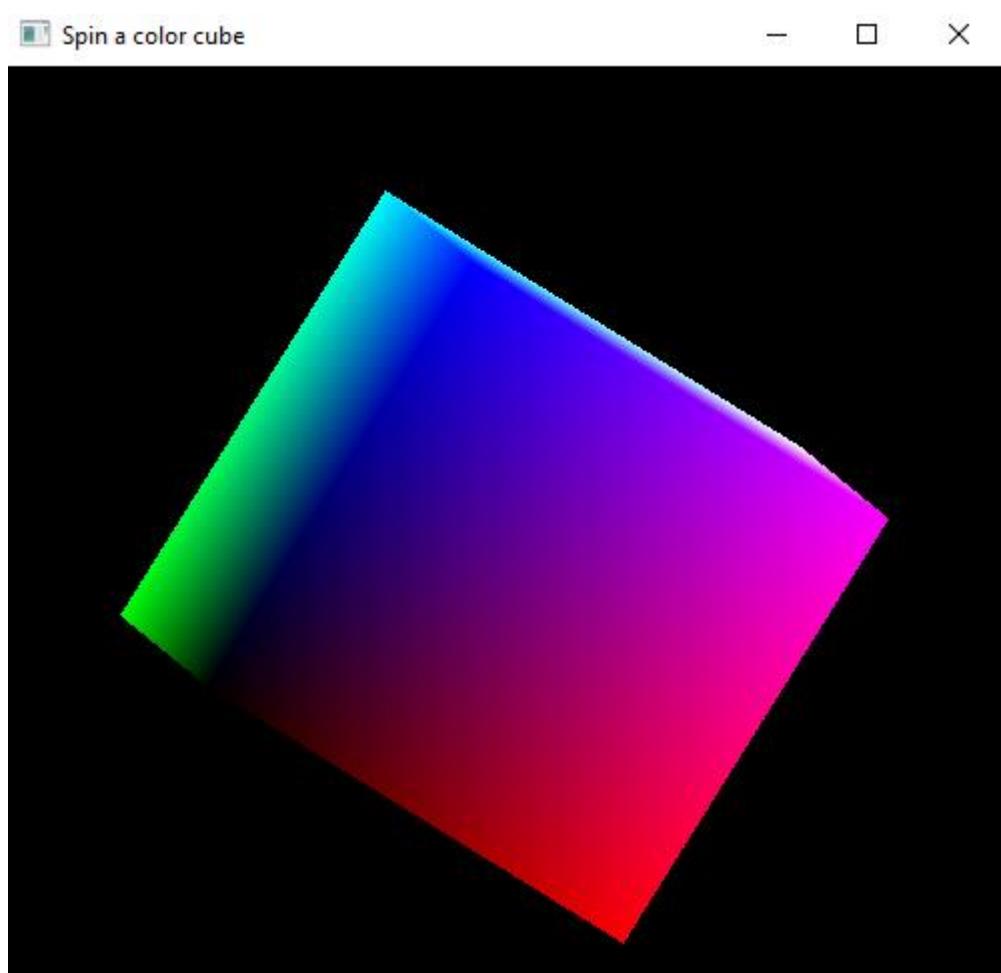
`void glEnableClientState (GLenum cap);`

`cap`

Specifies the capability to enable. Symbolic constants `GL_COLOR_ARRAY`, `GL_EDGE_FLAG_ARRAY`, `GL_FOG_COORD_ARRAY`, `GL_INDEX_ARRAY`, `GL_NORMAL_ARRAY`, `GL_SECONDARY_COLOR_ARRAY`, `GL_TEXTURE_COORD_ARRAY`, and `GL_VERTEX_ARRAY` are accepted.

`glVertexPointer` specifies the location and data format of an array of vertex coordinates to use when rendering. `size` specifies the number of coordinates per vertex, and must be 2, 3, or 4. `type` specifies the data type of each coordinate, and `stride` specifies the byte stride from one vertex to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays. `pointer` specifies a pointer to the first coordinate of the first vertex in the array. The initial value is 0.

`glColorPointer` specifies the location and data format of an array of color components to use when rendering. `size` specifies the number of components per color, and must be 3 or 4. `type` specifies the data type of each color component, and `stride` specifies the byte stride from one color to the next, allowing vertices and attributes to be packed into a single array or stored in separate arrays. `pointer` specifies a pointer to the first color of the first vertex in the array. The initial value is 0.

OUTPUT

Press left mouse button, middle and right ones and observe the change in the rotation

4. Program to draw a color cube and allow the user to move the camera suitably to experiment with perspective viewing.

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

GLfloat vertices[][3] = { {-1,-1,-1},
                           {1,-1,-1},
                           {1, 1,-1},
                           {-1, 1,-1},
                           {-1,-1, 1},
                           {1,-1, 1},
                           {1, 1, 1},
                           {-1, 1, 1}
                         };

GLfloat colors[][3] = { {1, 0, 0},
                        {1, 1, 0},
                        {0, 1, 0},
                        {0, 0, 1},
                        {1, 0, 1},
                        {1, 1, 1},
                        {0, 1, 1},
                        {0.5, 0.5, 0.5}
                      };

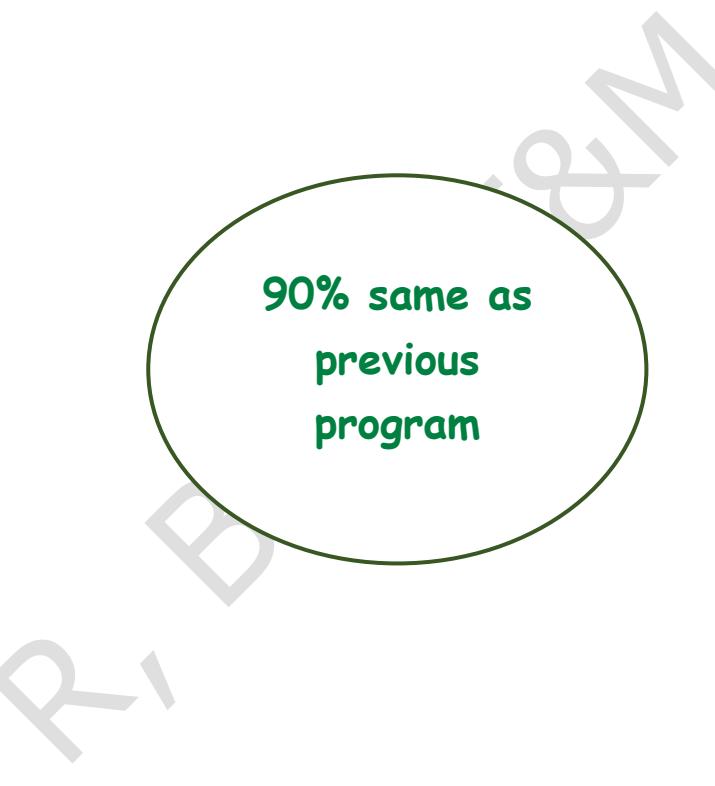
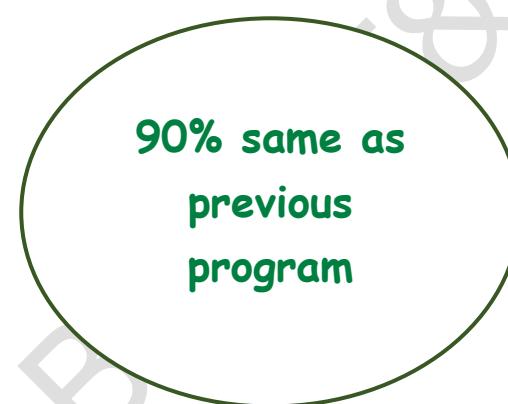
GLfloat theta[] = {0, 0, 0};
GLint axis = 2;
GLdouble viewer[] = {0, 0, 5}; // initial viewer location //

void polygon(int a, int b, int c, int d)
{
    glBegin(GL_POLYGON);
    glColor3fv(colors[a]);
    glVertex3fv(vertices[a]);

    glColor3fv(colors[b]);
    glVertex3fv(vertices[b]);

    glColor3fv(colors[c]);
    glVertex3fv(vertices[c]);

    glColor3fv(colors[d]);
    glVertex3fv(vertices[d]);
    glEnd();
}
```

90% same as
previous
program

```

void colorcube(void)
{
    polygon (0,3,2,1);
    polygon (0,4,7,3);
    polygon (5,4,0,1);
    polygon (2,3,7,6);
    polygon (1,2,6,5);
    polygon (4,5,6,7);
}

void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt (viewer[0],viewer[1],viewer[2], 0, 0, 0, 0, 1, 0);
    glRotatef (theta[0], 1, 0, 0);
    glRotatef (theta[1], 0, 1, 0);
    glRotatef (theta[2], 0, 0, 1);
    colorcube();
    glFlush();
    glutSwapBuffers();
}

void mouse(int btn, int state, int x, int y)
{
    if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        axis = 0;
    if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
        axis = 1;
    if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        axis = 2;

    theta[axis] += 2;

    if( theta[axis] > 360 )
        theta[axis] -= 360;
    display();
}

void keys(unsigned char key, int x, int y)
{
    if(key == 'x') viewer[0] -= 1;
    if(key == 'X') viewer[0] += 1;
    if(key == 'y') viewer[1] -= 1;
    if(key == 'Y') viewer[1] += 1;
    if(key == 'z') viewer[2] -= 1;
    if(key == 'Z') viewer[2] += 1;
    display();
}

```

```

void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if(w<=h)
        glFrustum(-2, 2, -2 * (GLfloat) h / (GLfloat) w, 2 * (GLfloat) h / (GLfloat) w, 2, 20);
    else
        glFrustum(-2, 2, -2 * (GLfloat) w / (GLfloat) h, 2 * (GLfloat) w / (GLfloat) h, 2, 20);
    glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutCreateWindow("Colorcube Viewer");

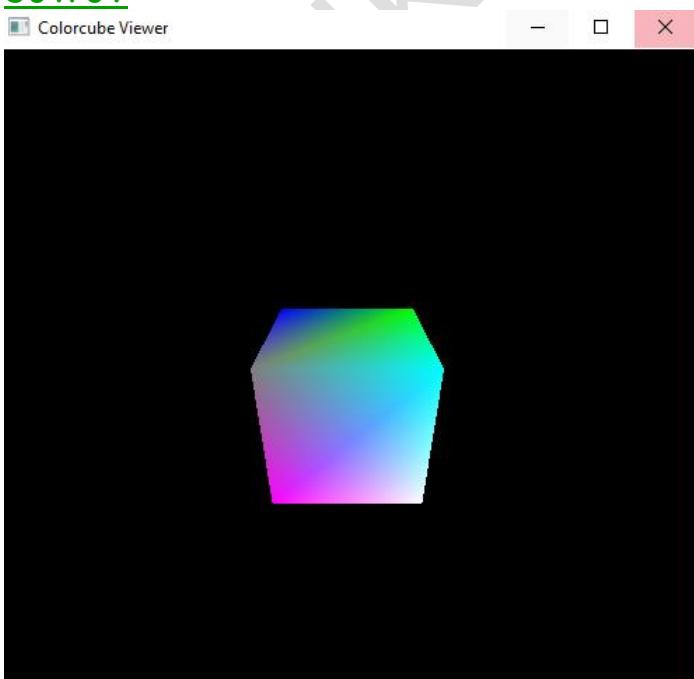
    glutReshapeFunc(myReshape);

    glutDisplayFunc(display);

    glutMouseFunc(mouse);

    glutKeyboardFunc(keys);

    glEnable(GL_DEPTH_TEST);
    glutMainLoop();
}
*****
```

OUTPUT

as and when you click left,
middle & right mouse
buttons, the cube rotates.

Also, press x, X

y, Y

z, Z

and observe the cube
rotation

5. Program to clip a line using Cohen-Sutherland line-clipping algorithm.

```

#include <stdio.h>
#include <GL/glut.h>
double xmin = 50, ymin = 50, xmax = 100, ymax = 100;           //window coordinates
double xvmin = 200, yvmin = 200, xvmax = 300, yvmax = 300; //viewport coordinates

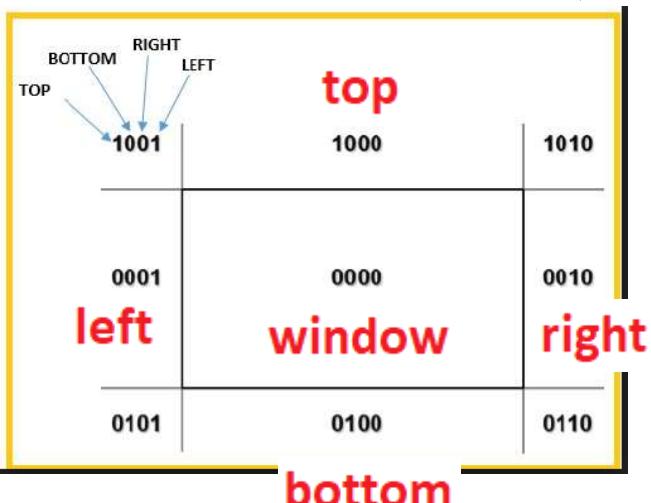
const int LEFT = 1;                                // assuming code words for LEFT, RIGHT, BOTTOM & TOP.
const int RIGHT = 2;
const int BOTTOM = 4;
const int TOP = 8;

int ComputeOutCode (double x, double y)
{
    int code = 0;
    if (y > ymax)                                //above the clip window
        code |= TOP;
    else if (y < ymin)                            //below the clip window
        code |= BOTTOM;
    if (x > xmax)                                //to the right of clip window
        code |= RIGHT;
    else if (x < xmin)                            //to the left of clip window
        code |= LEFT;
    return code;                                    //return the calculated code
}

void CohenSutherland(double x0, double y0, double x1, double y1)
{
    int outcode0, outcode1, outcodeOut;
    bool accept = false, done = false;
    outcode0 = ComputeOutCode (x0, y0);           //calculate the region of 1st point
    outcode1 = ComputeOutCode (x1, y1);           //calculate the region of 2nd point

    do
    {
        if (! (outcode0 | outcode1))
        {
            accept = true; //both the points
            done = true; //are inside the window
        }
        else if (outcode0 & outcode1)
            done = true; //both are outside
        else
        {
            double x, y;
            double m = (y1 - y0) / (x1 - x0);
            outcodeOut = outcode0 ? outcode0: outcode1;
            if (outcodeOut & TOP)
                y = m * (xmin - x0) + y0;
            if (outcodeOut & BOTTOM)
                y = m * (xmax - x0) + y0;
            if (outcodeOut & LEFT)
                x = m * (ymin - y0) + x0;
            if (outcodeOut & RIGHT)
                x = m * (yvmax - y0) + x0;
            if (y > yvmax)
                y = yvmax;
            if (y < yvmin)
                y = yvmin;
            if (x > xvmax)
                x = xvmax;
            if (x < xvmin)
                x = xvmin;
            glutPostRedisplay();
        }
    } while (!accept && !done);
}

```



```

if (outcodeOut & TOP)
{
    x = x0 + (1/m) * (ymax - y0);
    y = ymax;
}
else if (outcodeOut & BOTTOM)
{
    x = x0 + (1/m) * (ymin - y0);
    y = ymin;
}
else if (outcodeOut & RIGHT)
{
    y = y0 + m * (xmax - x0);
    x = xmax;
}
else
{
    y = y0 + m * (xmin - x0);
    x = xmin;
}

/* Intersection calculations are done,
go ahead and mark the clipped line */

if (outcodeOut == outcode0)
{
    x0 = x;
    y0 = y;
    outcode0 = ComputeOutCode (x0, y0);
}
else
{
    x1 = x;
    y1 = y;
    outcode1 = ComputeOutCode (x1, y1);
}

}

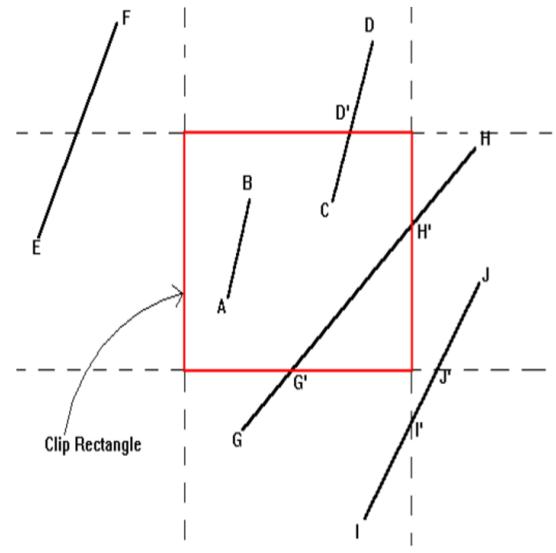
while (!done);

if (accept)
{
    double sx = (xvmax - xvmin) / (xmax - xmin);
    double sy = (yvmax - yvmin) / (ymax - ymin);

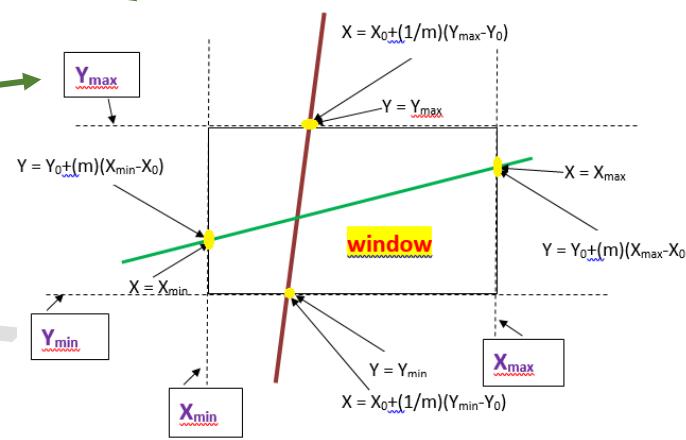
    double vx0 = xvmin + (x0 - xmin) * sx;
    double vy0 = yvmin + (y0 - ymin) * sy;

    double vx1 = xvmin + (x1 - xmin) * sx;
    double vy1 = yvmin + (y1 - ymin) * sy;
}

```



Calculating Intersection Points



Zooming (scaling) the clipping rectangle and the clipped line and show it to the customer. The customer can see both before and after clipping effects. See the output for better clarity.

$sx, sy \rightarrow$ scaling parameters

$vx0, vy0, vx1, vy1 \rightarrow$ line coordinates

```

glBegin(GL_LINE_LOOP);
    glVertex2f (xvmin, yvmin);
    glVertex2f (xvmax, yvmin);
    glVertex2f (xvmax, yvmax);
    glVertex2f (xvmin, yvmax);
glEnd();                                // draw the zoomed rectangle

glBegin(GL_LINES);
    glVertex2d (vx0, vy0);
    glVertex2d (vx1, vy1);
glEnd();                                  // draw the zoomed clipped line

}

void display()
{
    double x0 = 60, y0 = 20, x1 = 80, y1 = 120; // the line coordinates
    glClear (GL_COLOR_BUFFER_BIT);

    glColor3f(1, 1, 1);                         // white colour to draw line

    glBegin (GL_LINES);
        glVertex2d (x0, y0);
        glVertex2d (x1, y1);
    glEnd();                                    // draw the line that has to be clipped

    glBegin (GL_LINE_LOOP);
        glVertex2f (xmin, ymin);
        glVertex2f (xmax, ymin);
        glVertex2f (xmax, ymax);
        glVertex2f (xmin, ymax);
    glEnd();                                    // draw the clipping / viewing rectangle window

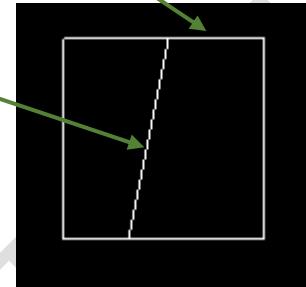
    CohenSutherland (x0, y0, x1, y1);          // call the algorithm

    glFlush ();                                 // show the output
}

void init()
{
    glClearColor (0, 0, 0, 1);                  //black background colour
    gluOrtho2D (0, 500, 0, 500);
}

```

// draw the zoomed clipped line



// draw the zoomed rectangle

// draw the line that has to be clipped

// draw the clipping / viewing rectangle window

// call the algorithm

// show the output

//black background colour

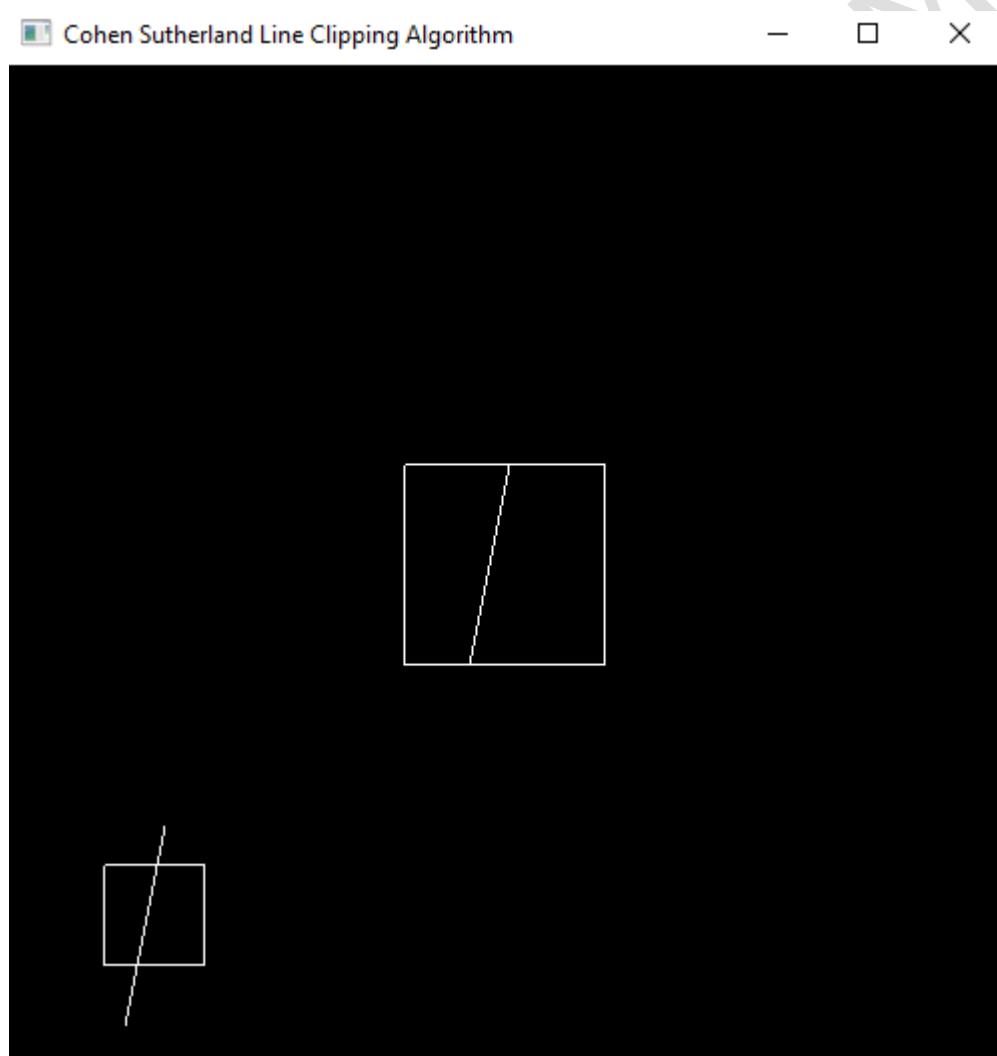
```
int main(int argc, char **argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (0, 0);
    glutCreateWindow ("Cohen Sutherland Line Clipping Algorithm");

    init();

    glutDisplayFunc(display);

    glutMainLoop();
}
```

OUTPUT



6. Program to draw a simple shaded scene consisting of a tea pot on a table. Define suitably the position and properties of the light source along with the properties of the properties of the surfaces of the solid object used in the scene.

```
#include<GL/glut.h>
void teapot(GLfloat x, GLfloat y, GLfloat z)
{
    glPushMatrix(); //save the current state
    glTranslatef(x, y, z); //move your item appropriately
    glutSolidTeapot(0.1); //render your teapot
    glPopMatrix(); //get back your state with the recent changes that you have done
}
```

```
void tableTop(GLfloat x, GLfloat y, GLfloat z) // table top which is actually a CUBE
```

```
{
    glPushMatrix();
    glTranslatef(x, y, z);
    glScalef(0.6, 0.02, 0.5);
    glutSolidCube(1);
    glPopMatrix();
}
```

glPushMatrix — pushes the current matrix stack. There is a stack of matrices for each of the matrix modes. In GL_MODELVIEW mode, the stack depth is at least 32. In the other modes, GL_COLOR, GL_PROJECTION, and GL_TEXTURE, the depth is at least 2. The current matrix in any mode is the matrix on the top of the stack for that mode. **glPushMatrix** pushes the current matrix stack down by one, duplicating the current matrix. That is, after a **glPushMatrix** call, the matrix on top of the stack is identical to the one below it. **glPopMatrix** pops the current matrix stack, replacing the current matrix with the one below it on the stack. Initially, each of the stacks contains one matrix, an identity matrix.

```
void tableLeg(GLfloat x, GLfloat y, GLfloat z) // table leg which is actually a CUBE
```

```
{
    glPushMatrix();
    glTranslatef(x, y, z);
    glScalef(0.02, 0.3, 0.02);
    glutSolidCube(1);
    glPopMatrix();
}
```

glutSolidCube(size) and **glutWireCube(size)** render a solid or wireframe cube respectively. The cube is centered at the modeling coordinates' origin with sides of length size.

```
void wall(GLfloat x, GLfloat y, GLfloat z) // wall which is actually a CUBE
```

```
{
    glPushMatrix();
    glTranslatef(x, y, z);
    glScalef(1, 1, 0.02);
    glutSolidCube(1);
    glPopMatrix();
}
```

```
void light() // set the lighting arrangements
```

```
{
    GLfloat mat_ambient[] = {1, 1, 1, 1}; // ambient colour
    GLfloat mat_diffuse[] = {0.5, 0.5, 0.5, 1};
    GLfloat mat_specular[] = {1, 1, 1, 1};
    GLfloat mat_shininess[] = {50.0f}; // shininess value
}
```

```

glMaterialfv (GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv (GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv (GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv (GL_FRONT, GL_SHININESS, mat_shininess);

GLfloat light_position[] = {2, 6, 3, 1};
GLfloat light_intensity[] = {0.7, 0.7, 0.7, 1};

glLightfv (GL_LIGHT0, GL_POSITION, light_position);
glLightfv (GL_LIGHT0, GL_DIFFUSE, light_intensity);

}

void display()
{
    GLfloat teapotP = -0.07, tabletopP = -0.15, tablelegP = 0.2, wallP = 0.5;
    glClear (GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();

    gluLookAt (-2, 2, 5, 0, 0, 0, 1, 0); // camera position & viewing

    light(); //Adding light source to your project

    teapot (0, teapotP, 0); //Create teapot

    tableTop (0, tabletopP, 0); //Create table's top

    tableLeg (tablelegP, -0.3, tablelegP); //Create 1st leg
    tableLeg (-tablelegP, -0.3, tablelegP); //Create 2nd leg
    tableLeg (-tablelegP, -0.3, -tablelegP); //Create 3rd leg
    tableLeg (tablelegP, -0.3, -tablelegP); //Create 4th leg

    wall (0, 0, -wallP); //Create 1st wall
    glRotatef (90, 1, 0, 0);

    wall (0, 0, wallP); //Create 2nd wall
    glRotatef (90, 0, 1, 0);

    wall (0, 0, wallP); //Create 3rd wall

    glFlush (); // show the output to the user
}

void init()
{
    glClearColor (0, 0, 0, 1); // black colour background
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    glOrtho (-1, 1, -1, 1, -1, 10);
    glMatrixMode (GL_MODELVIEW);
}

```

glMaterial — specify material parameters for the lighting model. fv means floating point vector

glMaterial takes three arguments. The first, face, specifies whether the GL_FRONT materials, the GL_BACK materials, or both GL_FRONT_AND_BACK materials will be modified. The second, pname, specifies which of several parameters in one or both sets will be modified. The third, params, specifies what value or values will be assigned to the specified parameter.

glLight sets the values of individual light source parameters. It takes 3 parameters - light, pname, params.

light names the light and is a symbolic name of the form GL_LIGHT i, where i ranges from 0 to the value of GL_MAX_LIGHTS - 1.

pname specifies one of ten light source parameters, again by symbolic name.

params is either a single value or a pointer to an array that contains the new values.

```

int main (int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(0, 0);
    glutCreateWindow("Teapot on a table");

    init();

    glutDisplayFunc(display);

    glEnable(GL_LIGHTING);      // enable the lighting properties
    glEnable(GL_LIGHT0);        // enable the light source

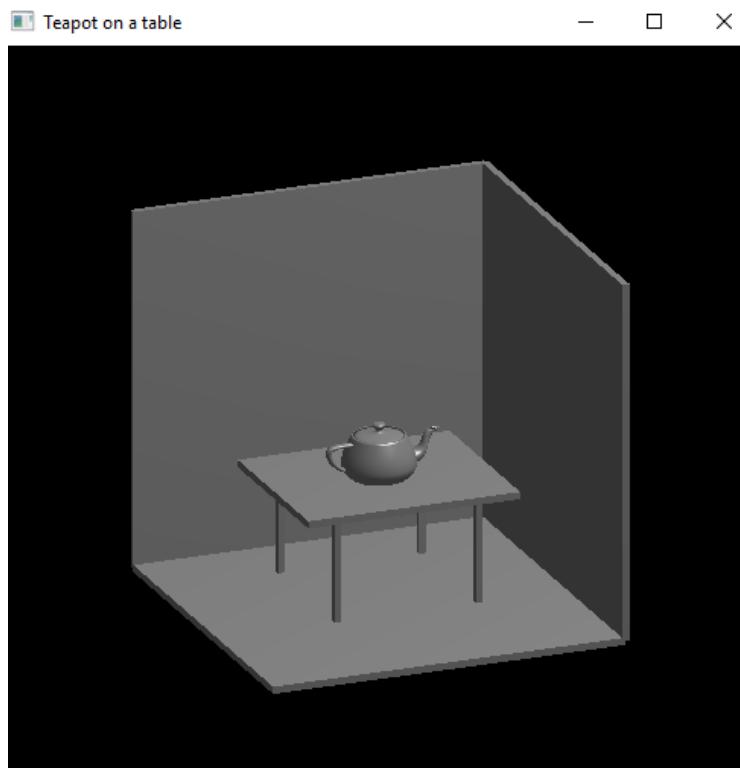
    glShadeModel(GL_SMOOTH);   // for smooth shading (select flat or smooth shading)

    glEnable(GL_NORMALIZE);    // If enabled and no vertex shader is active, normal vectors
                               // are normalized to unit length after transformation and before
                               // lighting.
    glEnable(GL_DEPTH_TEST);   // do depth comparisons and update the depth buffer.

    glutMainLoop();
}

```

OUTPUT



7. Program to recursively subdivide a tetrahedron to form **3D Sierpinski gasket**. The number of recursive steps is to be specified by the user

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

typedef float point[3];
point v[]={ {0, 0, 1}, {0, 1, 0}, {-1, -0.5, 0}, {1, -0.5, 0} };
int n;

void triangle(point a, point b, point c)
{
    glBegin(GL_POLYGON);
        glVertex3fv(a);
        glVertex3fv(b);
        glVertex3fv(c);
    glEnd();
}

void divide_triangle(point a, point b, point c, int n)
{
    point v1,v2,v3;
    int j;

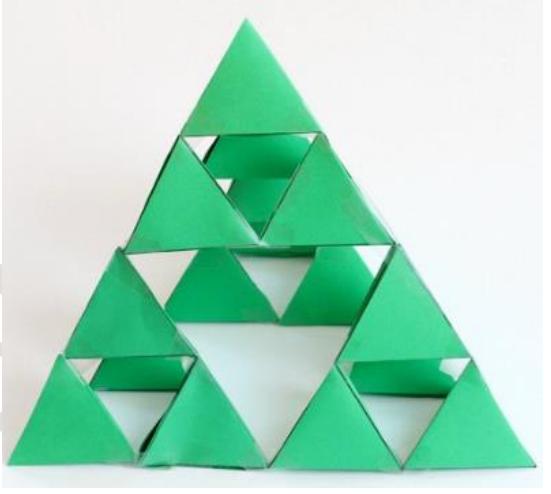
    if(n>0)
    {
        for(j=0; j<3; j++)
            v1[j] = (a[j]+b[j])/2; // calculate mid-point between a and b

        for(j=0; j<3; j++)
            v2[j] = (a[j]+c[j])/2; // calculate mid-point between a and c

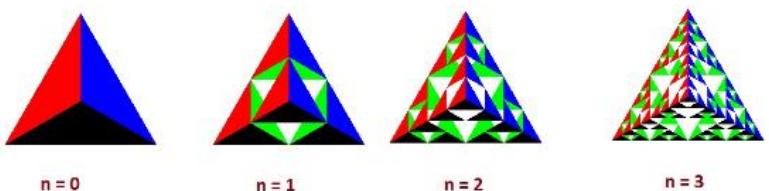
        for(j=0; j<3; j++)
            v3[j] = (c[j]+b[j])/2; // calculate mid-point between c and b

        divide_triangle(a,v1,v2,n-1); // divide triangle between points a, ab/2, ac/2 recursively
        divide_triangle(c,v2,v3,n-1);

        divide_triangle(b,v3,v1,n-1);
    }
    else
        triangle (a,b,c);// draw triangle
}
```



CG Lab 7 - Sierpinski Triangles



```

void tetrahedron(int n)
{
    glColor3f(1, 0, 0); // assign color for each of the side
    divide_triangle(v[0], v[1], v[2], n); // draw triangle between a, b, c

    glColor3f(0, 1, 0);
    divide_triangle(v[3], v[2], v[1], n);

    glColor3f(0, 0, 1);
    divide_triangle(v[0], v[3], v[1], n);

    glColor3f(0, 0, 0);
    divide_triangle(v[0], v[2], v[3], n);
}

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    tetrahedron(n);
    glFlush(); // show the output
}

void myReshape(int w,int h) // please see the earlier program for explanation on this
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    if(w<=h)
        glOrtho(-2, 2, -2*(GLfloat)h/(GLfloat)w, 2*(GLfloat)h/(GLfloat)w, -10, 10);
    else
        glOrtho(-2*(GLfloat)w/(GLfloat)h, 2*(GLfloat)w/(GLfloat)h, -2, 2, -10, 10);

    glMatrixMode(GL_MODELVIEW);
    glutPostRedisplay();
}

int main(int argc,char ** argv)
{
    printf("No of Recursive steps/Division: ");
    scanf("%d",&n);
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
    glutCreateWindow(" 3D Sierpinski gasket");

    glutReshapeFunc(myReshape);
}

```

```

glutDisplayFunc(display);      // call display function

glEnable(GL_DEPTH_TEST); // do depth comparisons and update the depth buffer.

glClearColor(1, 1, 1, 0);
glutMainLoop();

return 0;
}

```

OUTPUT

"C:\Users\Shankara\Dropbox\CG\Lab Final\temp\7_sierpinski\bin\Debug\7_sierpinski.exe"

No of Recursive steps/Division: 2

3D Sierpinski gasket - X



// for your information & understanding

CG Lab 7 - Sierpinski Triangles



n = 0



n = 1



n = 2



n = 3

8. Develop a menu driven program to animate a flag using Bezier curve algorithm.

Lets understand Bézier Curves first

Bézier curves are parametric curves that are generated with the control points. It is widely used in computer graphics and other related industry, as they appear reasonably smooth at all scales. Bézier curves was name after french engineer Pierre Bézier, who discovered it. Mathematically Bézier curves is represented as -

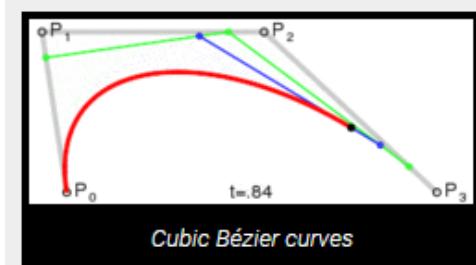
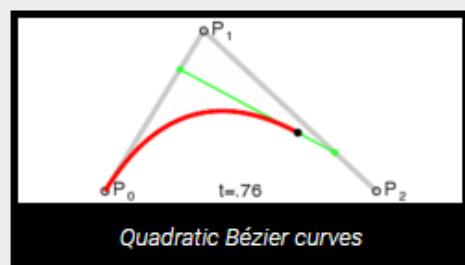
$$\sum_{k=0}^n P_i B_i^n(t)$$

Where P_i is the set of points and $B_i^n(t)$ represents the Bernstein polynomials which are given by –

$$B_i^n(t) = \binom{n}{i} (1-t)^{n-i} t^i$$

Where **n** is the polynomial degree, **i** is the index, and **t** is the variable.

Bézier curves are of different degree - linear curves, quadratic curve, cubic curve and high order curve.



So basically we need to calculate

Bezier curve = Bernstein Polynomial * For every point

Bezier curve = $(nCr) * (1-t)^{n-i} t^i$ * For every point

Where $n = (\text{number_of_control_points} - 1)$

$$= 4 - 1$$

$$n = 3$$

t ranges from 0 to 1

THE BASIC FLOW OF THIS CALCULATION IS:

Step 1: computeNcR

Step 2: bernstein_polynomial

Step 3: For every point

Finally - Multiply all

```

#include<GL/glut.h>
#include<stdio.h>
#include<math.h>
#define PI 3.1416
float theta = 0;

struct point
{
    GLfloat x, y, z;
};

int factorial (int n)
{
    if (n<=1)
        return (1);
    else
        n = n * factorial ( n-1 );
    return n;
}

void computeNcR (int n, int *hold_ncr_values)
{
    int r;
    for (r=0; r<=n; r++) //start from nC0, then nC1, nC2, nC3 till nCn
    {
        hold_ncr_values [r] = factorial (n) / ( factorial (n-r) * factorial (r) );
    }
}

void computeBezierPoints (float t, point *actual_bezier_point, int number_of_control_points,
                        point *control_points_array, int *hold_ncr_values) // 5 parameters
{
    int i, n = number_of_control_points - 1;

    float bernstein_polynomial;

    actual_bezier_point -> x = 0;
    actual_bezier_point -> y = 0;
    actual_bezier_point -> z = 0;

    for ( i=0; i<number_of_control_points; i++ )
    {
        bernstein_polynomial = hold_ncr_values [i] * pow(t, i) * pow( 1-t, n-i);

        actual_bezier_point->x += bernstein_polynomial * control_points_array [i].x;
        actual_bezier_point->y += bernstein_polynomial * control_points_array [i].y;
        actual_bezier_point->z += bernstein_polynomial * control_points_array [i].z;
    }
}

```

See the above explanation to understand this

```

void Bezier (point *control_points_array, int number_of_control_points, int number_of_bezier_points)
{
    point actual_bezier_point;
    float t;
    int *hold_ncr_values, i;

    hold_ncr_values = new int [number_of_control_points]; // to hold the nCr values

    computeNcR (number_of_control_points - 1, hold_ncr_values); // calculate nCr values

    glBegin (GL_LINE_STRIP);
        for(i=0; i<=number_of_bezier_points; i++)
        {
            t=float (i) / float (number_of_bezier_points);

            computeBezierPoints ( t, &actual_bezier_point, number_of_control_points,
                control_points_array, hold_ncr_values );// 5 parameters

            glVertex2f (actual_bezier_point.x, actual_bezier_point.y);
        }
    glEnd ();

    delete [] hold_ncr_values;
}

void display()
{
    glClear (GL_COLOR_BUFFER_BIT);
    int number_of_control_points= 4, number_of_bezier_points= 20;

    point control_points_array[4]= {{100, 400, 0}, {150, 450, 0}, {250, 350, 0},{300, 400, 0}};

    control_points_array[1].x += 50 * sin (theta * PI/180.0);
    control_points_array[1].y += 25 * sin (theta * PI/180.0); } // for animating the flag

    control_points_array[2].x -= 50 * sin ((theta+30) * PI/180.0);
    control_points_array[2].y -= 50 * sin ((theta+30) * PI/180.0); }

    control_points_array[3].x -= 25 * sin ((theta-30) * PI/180.0);
    control_points_array[3].y += sin ((theta-30) * PI/180.0); }

    theta += 2; //animating speed

    glPushMatrix ();
    glPointSize (5); // for plotting the point
}

```

See the above explanation to understand this

```

glColor3f (1, 0.4, 0.2); //Indian flag: Saffron color code
for (int i=0; i<50; i++)
{
    glTranslatef(0, -0.8, 0 );
    bezier(control_points_array, number_of_control_points, number_of_bezier_points);
}

glColor3f(1, 1, 1); //Indian flag: white color code
for(int i=0; i<50; i++)
{
    glTranslatef(0, -0.8, 0);
    bezier(control_points_array, number_of_control_points, number_of_bezier_points);
}

glColor3f(0, 1, 0); //Indian flag: green color code
for(int i=0; i<50; i++)
{
    glTranslatef(0, -0.8, 0);
    bezier(control_points_array, number_of_control_points, number_of_bezier_points);
}

glPopMatrix();

glLineWidth(5);

glColor3f(0.7, 0.5, 0.3); //pole colour

glBegin(GL_LINES);
    glVertex2f(100,400);
    glVertex2f(100,40);
glEnd();

glutPostRedisplay();      // call display again
glutSwapBuffers();        // show the output
}

void init ()
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0,500,0,500);
}

int main(int argc, char ** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowPosition(0, 0);
    glutInitWindowSize(500,500);
}

```

```
glutCreateWindow ("Bezier Curve - updated");

init ();

glutDisplayFunc (display);

glutMainLoop (0);
}
```

OUTPUT

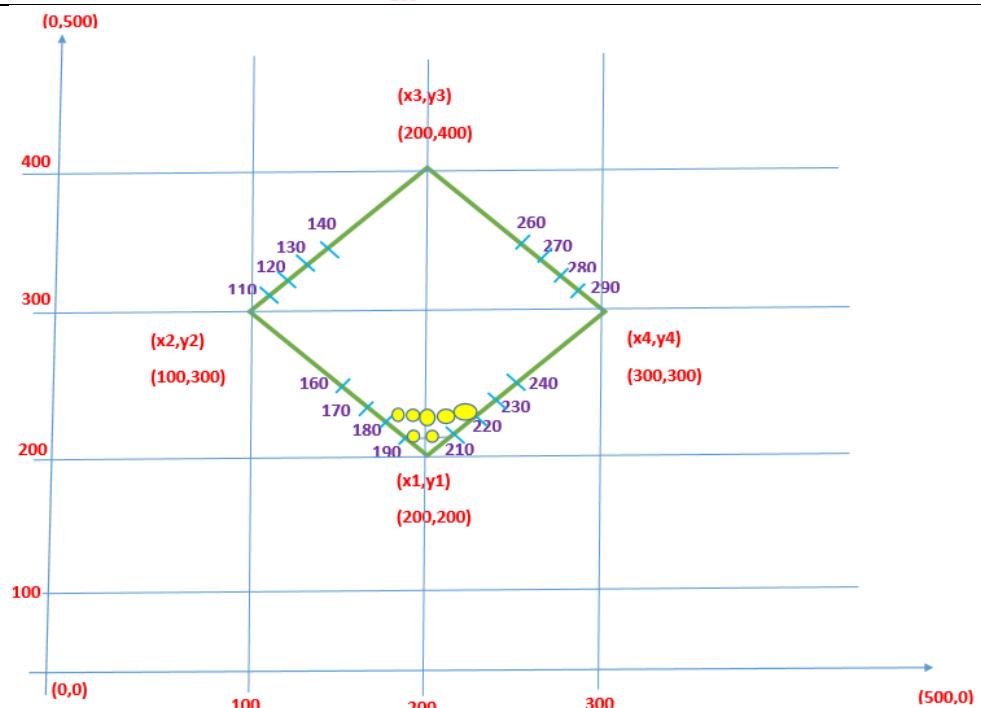
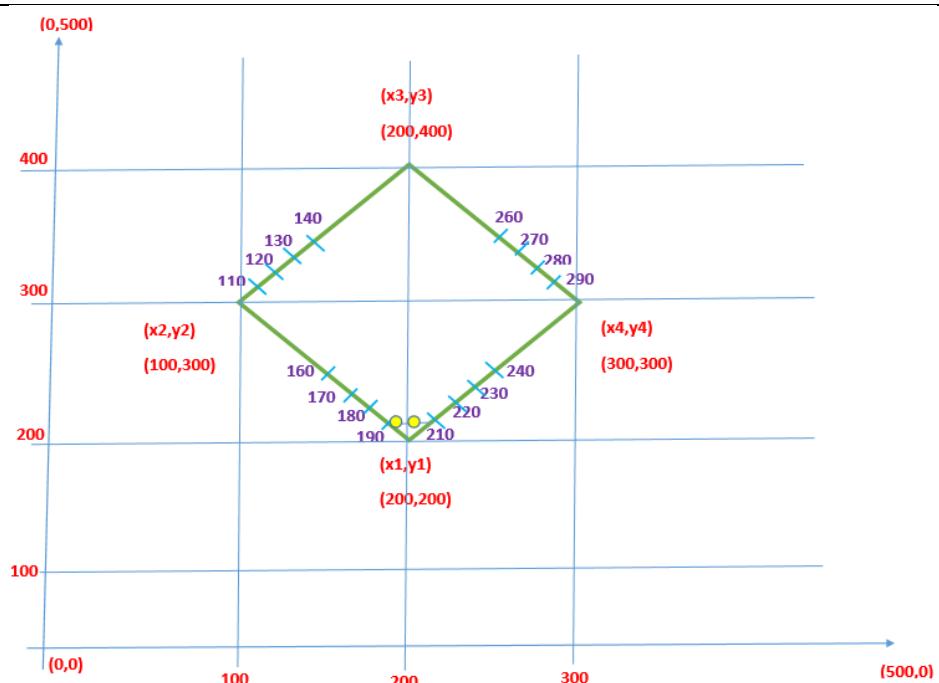


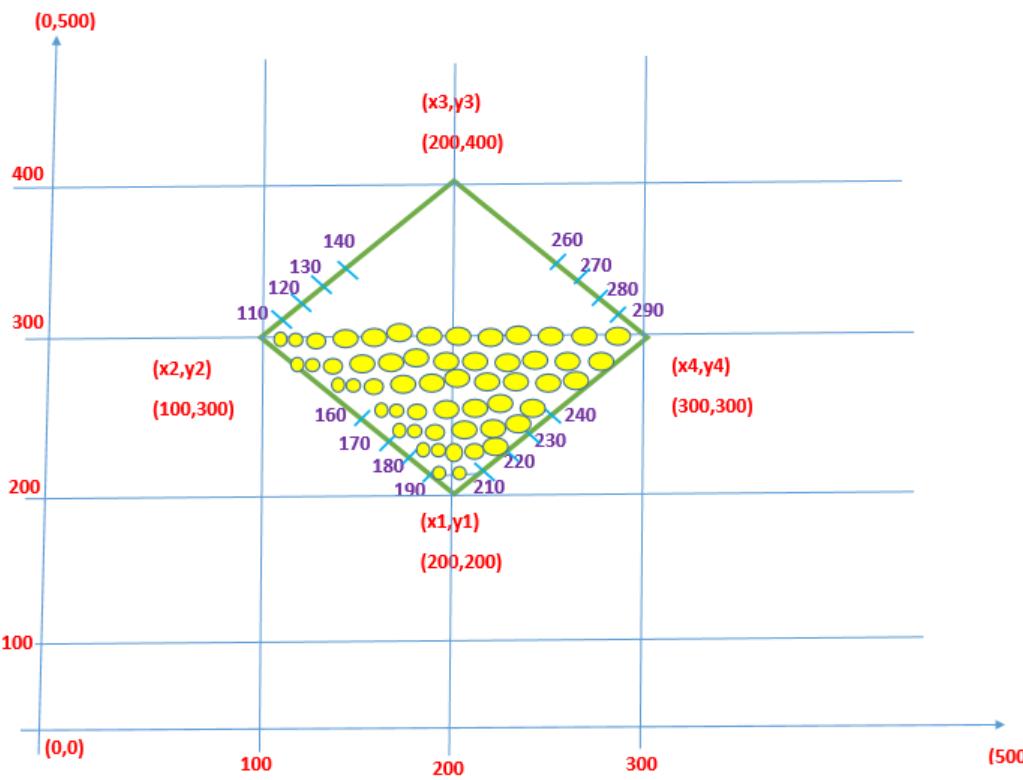
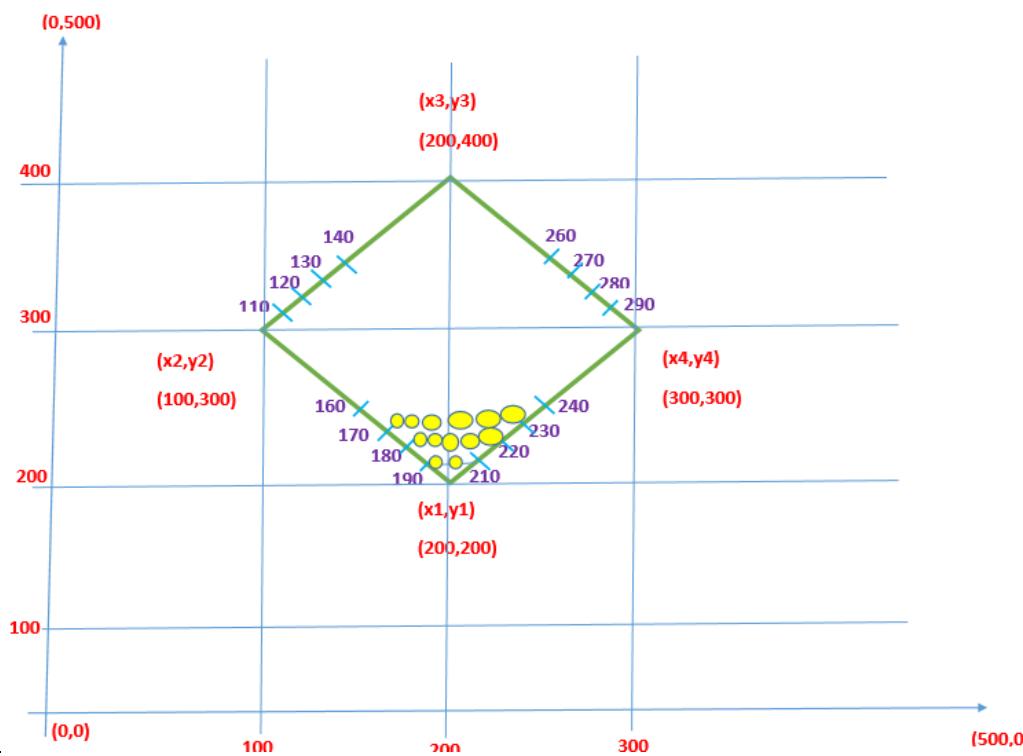
9. Develop a menu driven program to fill any given polygon using scan-line area filling algorithm.

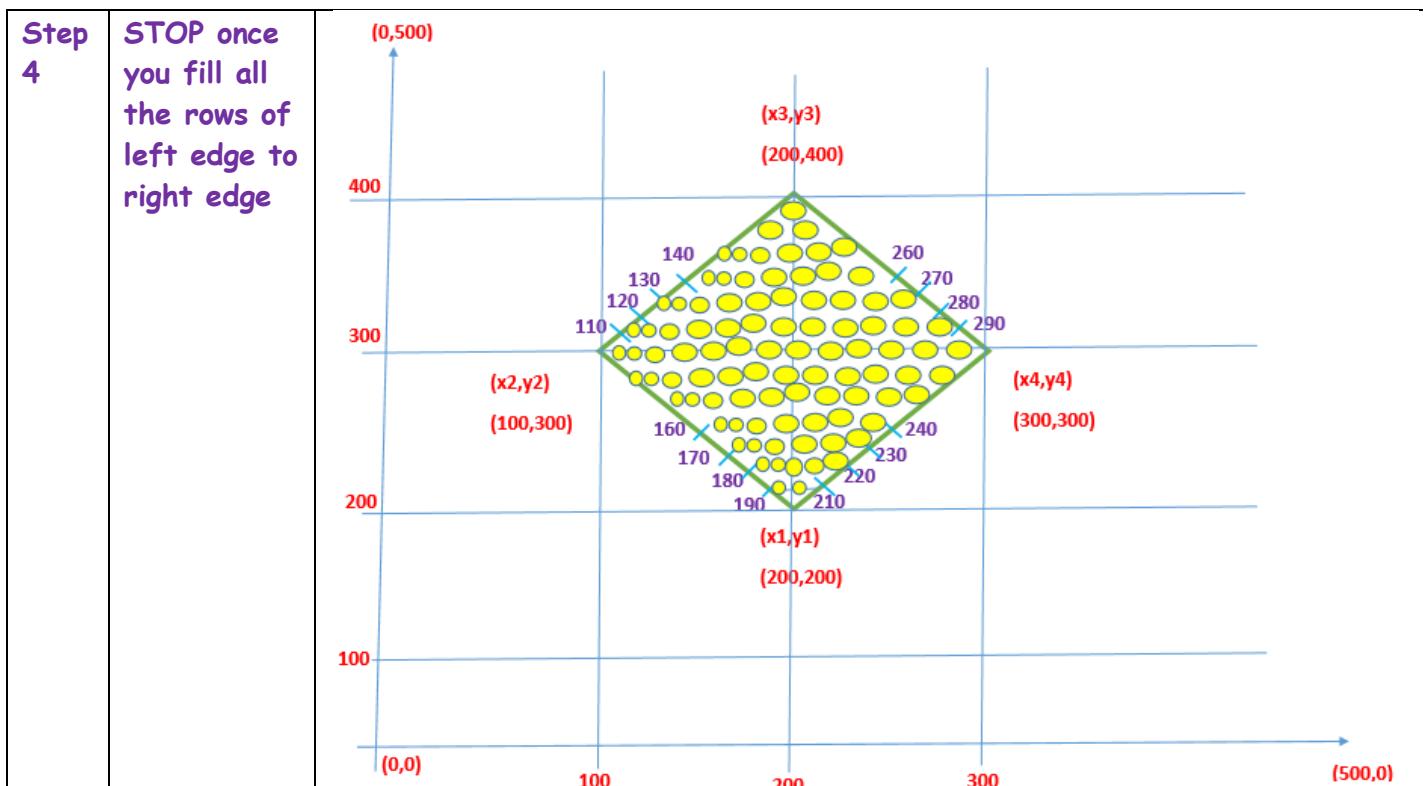
Step	Description	Output
Step 1	<p>Define the boundary -> x axis and y axis</p> <p>Also, define your polygon i.e., x_1, y_1 x_2, y_2 x_3, y_3 x_4, y_4</p>	
Step 2	Run 4 big for loops and mark your left edges and right edges	

**Step
3**

Start filling
from the
left edge to
right edge







You remember I had taught all the iterations via debug - breakpoints and an excel sheet which kinda looked like this after 4th iteration?

i	leftedge	rightedge
198	500	0
199	500	0
200	200	200
201	199	201
202	198	202
203	197	203
204	196	204
205	195	205
206	194	206
207	193	207
208	192	208
209	191	209
210	190	210
211	189	211
212	188	212
213	187	213
214	186	214
215	185	215

and so on....

```

#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>

float x1, x2, x3, x4, y1, y2, y3, y4; // our polygon has 4 lines - so 8 coordinates

void edgedetect(float x1, float y1, float x2, float y2, int *left_edge, int *right_edge)
{
    float x_slope, x, temp;
    int i;

    if ((y2-y1)<0) // decide where to start
    {
        temp = y1;
        y1 = y2;
        y2 = temp;

        temp = x1;
        x1 = x2;
        x2 = temp;
    }

    if ((y2-y1)!=0) // compute the values
        x_slope = (x2 - x1) / (y2 - y1);
    else
        x_slope = x2 - x1;

    x = x1;

    for (i = y1; i <= y2; i++) // fill the values
    {
        if (x < left_edge[i])
            left_edge[i] = x;

        if (x > right_edge[i])
            right_edge[i] = x;

        x = x + x_slope;
    }
}

void draw_pixel (int x, int y) // fill the polygon point by point (pixel by pixel)
{
    glColor3f (1, 1, 0); // fill the RHOMBUS in yellow colour
    glBegin (GL_POINTS);
        glVertex2i (x, y);
    glEnd ();
}

```

```

void scanfill (float x1, float y1, float x2, float y2, float x3, float y3, float x4, float y4)
{
    int left_edge[500], right_edge[500];
    int i, y;

    for (i = 0; i <= 500; i++)
    {
        left_edge [i] = 500;      // fill all the left_edge values as 500 initially
        right_edge [i] = 0;       // fill all the right_edge values as 0 initially
    }

    edgedetect (x1, y1, x2, y2, left_edge, right_edge); // first line
    edgedetect (x2, y2, x3, y3, left_edge, right_edge); // second line
    edgedetect (x3, y3, x4, y4, left_edge, right_edge); // third line
    edgedetect (x4, y4, x1, y1, left_edge, right_edge); // fourth line

    for (y = 0; y <= 500; y++) // now that you have calculated all values, start filling
    {                          // from left edge to right edge row by row pixel by pixel
        if (left_edge[y] <= right_edge[y])
        {
            for (i = left_edge[y]; i <= right_edge[y]; i++)
            {
                draw_pixel (i, y);
                glFlush ();
            }
        }
    }
}

void display()
{
    x1 = 200, y1 = 200;           // RHOMBUS coordinates
    x2 = 100, y2 = 300;
    x3 = 200, y3 = 400;
    x4 = 300, y4 = 300;

    glClear (GL_COLOR_BUFFER_BIT);

    glColor3f (0, 0, 1);          // blue RHOMBUS

    glBegin (GL_LINE_LOOP);       // draw the RHOMBUS
        glVertex2f (x1, y1);
        glVertex2f (x2, y2);
        glVertex2f (x3, y3);
        glVertex2f (x4, y4);
    glEnd ();
}

```

```

scanf (x1, y1, x2, y2, x3, y3, x4, y4);      // FILL the RHOMBUS
}

void init()
{
    glClearColor (1, 1, 1, 1);
    gluOrtho2D (0, 499, 0, 499);
}

int main (int argc, char** argv)
{
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (0, 0);
    glutCreateWindow ("Filling a Polygon using Scan-line Algorithm");

    init ();

    glutDisplayFunc (display);

    glutMainLoop ();
}

```

OUTPUT

Filling a Polygon using Scan-line Algorithm - X

