ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028

A T M E
College of Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

(ACADEMIC YEAR 2022-23)

LABORATORY MANUAL

SUBJECT: COMPUTER GRAPHICS LABORATORY

WITH MINI PROJECT

SUBJECT CODE: 18CSL67

SEMESTER: VI

2018 CBCS Scheme

INSTITUTIONAL MISSION AND VISION

Objectives

- To provide quality education and groom top-notch professionals, entrepreneurs and leaders for different fields of engineering, technology and management.
- To open a Training-R & D-Design-Consultancy cell in each department, gradually introduce doctoral and postdoctoral programs, encourage basic & applied research in areas of social relevance, and develop the institute as a center of excellence.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To develop academic, professional and financial alliances with the industry as well as the academia at national and transnational levels.
- To cultivate strong community relationships and involve the students and the staff in local community service.
- To constantly enhance the value of the educational inputs with the participation of students, faculty, parents and industry.

Vision

- Development of academically excellent, culturally vibrant, socially responsible and globally competent human resources.

Mission.

- To keep pace with advancements in knowledge and make the students competitive and capable at the global level.
- To create an environment for the students to acquire the right physical, intellectual, emotional and moral foundations and shine as torch bearers of tomorrow's society.
- To strive to attain ever-higher benchmarks of educational excellence.

Department of Computer Science & Engineering

Vision of the Department

- To develop highly talented individuals in Computer Science and Engineering to deal with real world challenges in industry, education, research and society.

Mission of the Department

- To inculcate professional behavior, strong ethical values, innovative research capabilities and leadership abilities in the young minds & to provide a teaching environment that emphasizes depth, originality and critical thinking.
- Motivate students to put their thoughts and ideas adoptable by industry or to pursue higher studies leading to research.

Program outcomes (POs)

Engineering Graduates will be able to:

- **PO1. Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems
- **PO2. Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
- **PO3. Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
- **PO4. Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
- **PO5. Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

- **PO6. The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice
- **PO7. Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
- **PO8. Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
- **PO9. Individual and team work:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
- **PO10. Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
- **PO11. Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
- **PO12. Life-long learning:** Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Educational Objectives (PEO'S):

1. Empower students with a strong basis in the mathematical, scientific and engineering fundamentals to solve computational problems and to prepare them for employment, higher learning and R&D.
2. Gain technical knowledge, skills and awareness of current technologies of computer science engineering and to develop an ability to design and provide novel engineering solutions for software/hardware problems through entrepreneurial skills.
3. Exposure to emerging technologies and work in teams on interdisciplinary projects with effective communication skills and leadership qualities.
4. Ability to function ethically and responsibly in a rapidly changing environment by Applying innovative ideas in the latest technology, to become effective professionals in Computer Science to bear a life-long career in related areas.

Program Specific Outcomes (PSOs)

1. Ability to apply skills in the field of algorithms, database design, web design, cloud computing and data analytics..
2. Apply knowledge in the field of computer networks for building network and internet based applications.

COMPUTER GRAPHICS LABORATORY WITH MINI PROJECT

LABORATORY

Subject Code	**:**	**18CSL67**	**CIE Marks**	**:**	**40**
Hours/Week	**:**	**0:2:2**	**SEE Marks**	**:**	**60**
Total Hours	**:**	**36**	**Exam Hours**	**:**	**03**

Credits - 2

Course objectives: This course (18CSL67) will enable students to:

- Demonstrate simple algorithms using OpenGL Graphics Primitives and attributes.
- Implementation of line drawing and clipping algorithms using OpenGL functions
- Design and implementation of algorithms Geometric transformations on both 2D and 3D objects.

Descriptions (if any): --

Installation procedure of the required software must be demonstrated, carried out in groups and documented in the journal.

Programs List:

PART A

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

1. Implement Brenham's line drawing algorithm for all types of slope.

   Refer: Text-1: Chapter 3.5
   Refer: Text-2: Chapter 8

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

   Refer: Text-1: Chapter 5-4

3. Draw a colour cube and spin it using OpenGL transformation matrices.

   Refer: Text-2: Modelling a Coloured Cube

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

   Refer: Text-2: Topic: Positioning of Camera

5. Clip a line using Cohen-Sutherland algorithm.

   Refer: Text-1: Chapter 6.7
   Refer: Text-2: Chapter 8

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.

   Refer: Text-2: Topic: Lighting and Shading

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.

   Refer: Text-2: Topic: sierpinski gasket.

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

   Refer: Text-1: Chapter 8-10

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 Open GL 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.
Laboratory outcomes: The students should be able to:

- Apply the concepts of computer graphics
- Implement computer graphics applications using OpenGL
- Animate real world problems using OpenGL

Conduction of Practical Examination:

- Experiment distribution
  - For laboratories having only one part: Students are allowed to pick one experiment from the lot with equal opportunity.
  - For laboratories having PART A and PART B: Students are allowed to pick one experiment from PART A and one experiment from PART B, with equal opportunity.
- Change of experiment is allowed only once and marks allotted for procedure to be made zero of the changed part only.
- Marks Distribution *(Courseed to change in accoradance with university regulations)*
  - For laboratories having only one part – Procedure + Execution + Viva-Voce: 15+70+15 =100 Marks
  - For laboratories having PART A and PART B
    i. Part A – Procedure + Execution + Viva = 6 + 28 + 6 = 40 Marks
    ii. Part B – Procedure + Execution + Viva = 9 + 42 + 9 = 60 Marks

CONTENT LIST

OpenGL

Introduction

OpenGL (Open Graphics Library) is an application program interface (API) that is used to define 2D and 3D computer graphics.

The interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL was developed by Silicon Graphics Inc. (SGI) in 1992 and is widely used in CAD, virtual reality, scientific visualization, information visualization, and flight simulation.

OpenGL's basic operation is to accept primitives such as points, lines and polygons, and convert them into pixels. This is done by a graphics pipeline known as the OpenGL state machine.

Features of OpenGL: -

- Geometric Primitives allow you to construct mathematical descriptions of objects.
- Color coding in RGBA (Red-Green-Blue-Alpha) or in color index mode.
- Viewing and Modeling permits arranging objects in a 3-dimensional scene, move our camera around space and select the desired vantage point for viewing the scene to be rendered.
- Texture mapping helps to bring realism into our models by rendering images of realistic looking surfaces on to the faces of the polygon in our model.
- Materials lighting OpenGL provides commands to compute the color of any point given the properties of the material and the sources of light in the room.
- Double buffering helps to eliminate flickering from animations. Each successive frame in an animation is built in a separate memory buffer and displayed only when rendering of the frame is complete.
- Anti-aliasing reduces jagged edges in lines drawn on a computer display. Jagged lines often appear when lines are drawn at low resolution. Anti-aliasing is a common computer graphics technique that modifies the color and intensity of the pixels near the line in order to reduce the artificial zig-zag.
- Gouraud shading is a technique used to apply smooth shading to a 3D object and provide subtle color differences across its surfaces.
- Z-buffering keeps track of the Z coordinate of a 3D object. The Z-buffer is used to keep track of the proximity of the viewer's object. It is also crucial for hidden surface removal
- Transformations: rotation, scaling, translations, perspectives in 3D, etc.

GLUT gives you the ability to create a window, handle input and render to the screen without being Operating System dependent.

The first things you will need are the OpenGL and GLUT header files and libraries for your current Operating System.

Once you have them setup on your system correctly, open your first C file and include them at the start of your program:

```
#include <GL/gl.h> //include the gl header le
#include <GL/glut.h>  //include the GLUT header
```

Now, just double check that your system is setup correctly, and try compiling your current file.

If you get no errors, you can proceed. If you have any errors, try your best to fix them. Once you are ready to move onto the next step, create a main() method in your current file.

Inside this is where all of your main GLUT calls will go.

The first call we are going to make will initialize GLUT and is done like so:
glutInit(&argc, argv); //initialize the program.

Keep in mind for this, that argc and argv are passed as parameters to your main method. You can see how to do this below.

```
int main(int argc,char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
glutInitWindowSize(500,500);
glutCreateWindow("");
glutDisplayFunc(display);
myinit(); glutMainLoop();
}
```

Once we have GLUT initialized, we need to tell GLUT how we want to draw. There are several parameters we can pass here, but we are going to stick them with most basic GLUT_SINGLE, which will give use a single buffered window.

```
glutInitDisplayMode(GLUT_SINGLE);
  //set up a basic display buffer (only singular for now)
```

The next two methods we are going to use, simply set the size and position of the GLUT window on our screen:
glutInitWindowSize (500, 500); //set the width and height of the window
glutInitWindowPosition (100, 100); // set the position of the window

And then we give our window a caption/title, and create it.

glutCreateWindow ("A basic OpenGL Window"); //set the caption for the window

We now have a window of the size and position that we want. But we need to be able to draw to it. We do this, by telling GLUT which method will be our main drawing method. In this case, it is a void method called display ()

glutDisplayFunc(display); //call the display function to draw our world
Finally, we tell GLUT to start our program. It does this by executing a loop that will continue until the program ends.
glutMainLoop(); //initialize the OpenGL loop cycle

So thus far, we have a window. But the display method that I mentioned is needed.
Let's take a look at this and dissect it.

```
void display(void)
{

glClearColor (0.0,0.0,0.0,1.0); //clear the color of the window
glClear (GL_COLOR_BUFFER_BIT); //Clear the Color Buffer (more buffers later on)
glLoadIdentity(); //load the Identity Matrix
gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0); //set the view
glFlush(); //flush it all to the screen
}
```

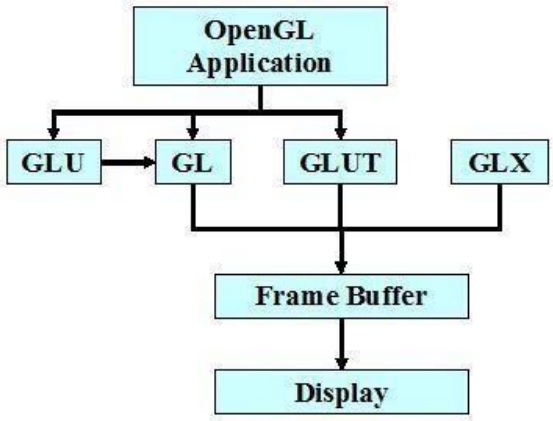
The first method, glClearColor will set the background color of our window. In this example, we are setting the background to black (RGB 0, 0, 0). The 1.0 value on the end is an alpha value and makes no difference at this stage as we don't have alpha enabled.

The next thing we want to do, is erase everything currently stored by OpenGL. We need to do this at the start of the method, as the method keeps looping over itself and if we draw something, we need to erase it before we draw our next frame. If we don't do this, we can end up with a big mess inside our buffer where frames have been drawn over each other.

The third method, glLoadIdentity resets our model view matrix to the identity matrix, so that our drawing transformation matrix is reset. From then, we will set the 'camera' for our scene. I am placing it 5 units back into the user so that anything we draw at 0, 0, 0 will be seen in front of us. The final thing we need to do is then user the current buffer to the screen so we can see it. This can be done with glFlush() as we are only using a single buffer.

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

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
- glutSwapBuers()
- glutInitWindowSize(500, 500);

OpenGL Implementations: OpenGL IS an API

#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 -lGLU -lglut
Event Loop:

OpenGL programs often run in an event loop:

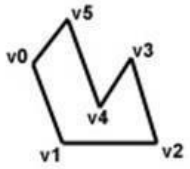
- Start the program
- Run some initialization code
- Run an infinite loop and wait for events such as Key press Mouse move, click Reshape window Expose event

- 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

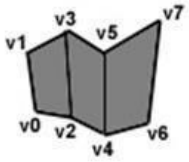
OpenGL Command Syntax (2)

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

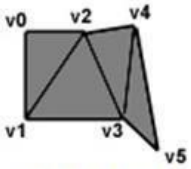
Ten gl Primitives



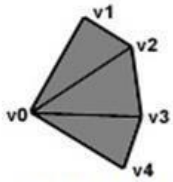
GL_LINE_LOOP



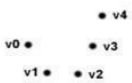
GL_QUAD_STRIP



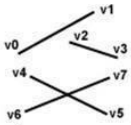
GL_TRIANGLE_STRIP



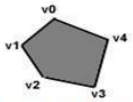
GL_TRIANGLE_FAN



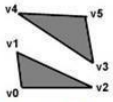
GL_POINTS



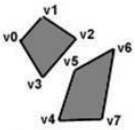
GL_LINES



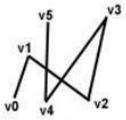
GL_POLYGON



GL_TRIANGLES



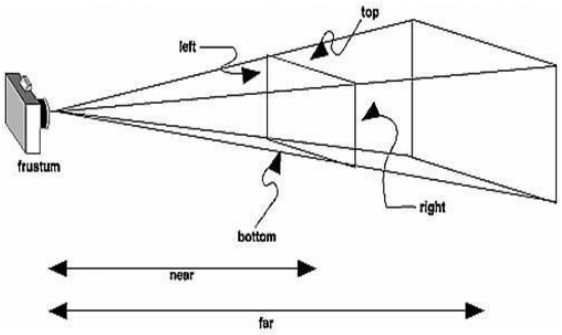
GL_QUADS



GL_LINE_STRIP

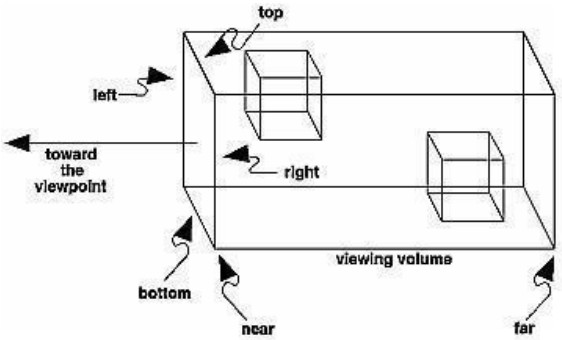
Projections in OpenGL

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, GLdou-ble near, GLdouble far)

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 defnition, 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 prefex (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 x 4 oating-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 geome-try which fall outside a half-space, dened 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 rst 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 bufer, 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 frame buffer to other parts of the frame buffer 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 frame buffer.

- 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 frame buffer. 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 frame buffer, 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. Un-fortunately, 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 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: species 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 innite 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 specied mode, such as color

- mode (GLUTRGB, GLUTRGBA, GLUTINDEX), single/double buering (GLUT SINGLE , GLUTDOUBLE), enable depth (GLUTDEPTH), joined with a bit OR '|'. void glu-tInitDisplayMode(unsigned int displayMode)

- void glMatrixMode (GLenum mode); The glMatrixMode function species 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 (GLoat size); The glPointSize function species 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.

- GLintglRenderMode (GLenum mode); The glRenderMode function sets the ras-terization mode.

- void glRotatef (GLoat angle, GLoat x, GLoat y, GLoat); The glRotatef functions multiply the current matrix by a rotation matrix.
- void glScalef (GLoat x, GLoat y, GLoat z); The glScalef functions multiply the current matrix by a general scaling matrix.

- void glTranslatef (GLoat x, GLoat y, GLoat z); The glTranslatef functions multiply the current matrix by a translation matrix.

- void glViewport (GLint x, GLint y, GLsizei width, GLsizei height); The glView-port function sets the viewport.

- void glEnable, glDisable(); The glEnable and glDisable functions enable or dis-able OpenGL capabilities.

- glutBitmapCharacter(); The glutBitmapCharacter function used for font style.

Introduction to frequently used OpenGL commands

glutInit

- glutInit is used to initialize the GLUT library.

Usage
void glutInit(int *argcp, char **argv);

argcp

- A pointer to the program's unmodied 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 unmodied 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. 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.

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_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_DEPTH

Bit mask to select a window with a depth buffer.
Description

- The initial display mode is used when creating top-level windows.

glutMainLoop

- glutMainLoop enters the GLUT event processing loop.

Usage
void glutMainLoop(void);
Description

- 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.

glutCreateWindow

- glutCreateWindow creates a top-level window.

Usage
int glutCreateWindow(char *name);

name

- ASCII character string for use as window name.

Description

- glutCreateWindow creates a top-level window. The name will be provided to the window system as the window's name. The intent is that the window system will label the window with the name.

glutPostRedisplay

- glutPostRedisplay marks the current window as needing to be redisplayed.

Usage
void glutPostRedisplay(void);

Description

- Mark the normal plane of current window as needing to be redisplayed. The next iteration through glutMainLoop, the window's display callback will be called to redisplay

the window's normal plane. Multiple calls to glutPostRedisplay before the next display callback opportunity generates only a single redisplay callback.

glutReshapeWindow

- glutReshapeWindow requests a change to the size of the current window.

Usage
void glutReshapeWindow(int width, int height);

width

- New width of window in pixels.

Height

- New height of window in pixels.

Description

- glutReshapeWindow requests a change in the size of the current window. The width and height parameters are size extents in pixels. The width and height must be positive values. The requests by glutReshapeWindow are not processed immediately. The request is executed after returning to the main event loop. This allows multiple glutReshapeWindow, glutPositionWindow, and glut-FullScreen requests to the same window to be coalesced.

glutDisplayFunc

- glutDisplayFunc sets the display callback for the current window.

Usage
void glutDisplayFunc(void (*func)(void));
func

- The new display callback function.

^

Description

- glutDisplayFunc sets the display callback for the current window. When GLUT determines that the normal plane for the window needs to be redisplayed, the display callback for the window is called. Before the callback, the current win-dow is set to the window needing to be redisplayed and (if no overlay display callback is registered) the layer in use is set to the normal plane. The display callback is called with no parameters. The entire normal plane region should be redisplayed in response to the callback (this includes ancillary buers if your program depends on their state).

glutReshapeFunc

- glutReshapeFunc sets the reshape callback for the current window.

Usage
void glutReshapeFunc(void (*func)(int width, int height));
func

- The new reshape callback function.

Description

- glutReshapeFunc sets the reshape callback for the current window. The reshape callback is triggered when a window is reshaped. A reshape callback is also triggered

immediately before a window's rst display callback after a window is created or whenever an overlay for the window is established. The width and height parameters of the callback specify the new window size in pixels. Before the callback, the current window is set to the window that has been reshaped.

glFlush

- glFlush - force execution of GL commands in finite time

^

Description

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

glMatrixMode

- glMatrixMode - specify which matrix is the current matrix

Usage
void glMatrixMode(GLenum mode)
Parameters

- Mode- Species which matrix stack is the target for subsequent matrix oper-ations. Three values are accepted: GL_MODELVIEW, GL_PROJECTION, and GL_TEXTURE. The default value is GL_MODELVIEW.

Description

- glMatrixMode sets the current matrix mode. Mode can assume one of three values: GL_MODELVIEW - Applies subsequent matrix operations to the mode matrix stack. GL_PROJECTION - Applies subsequent matrix operations to the projection matrix stack

gluOrtho2D

Usage
gluOrtho2D(left, right, bottom, top)

- Species the 2D region to be projected into the viewport. Any drawing outside the region will be automatically clipped away

Sample Programs

Creating a Window:

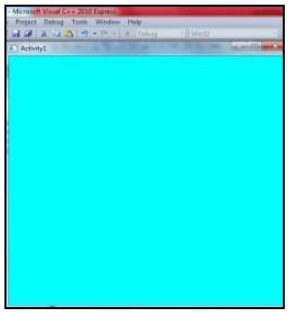
```
#include<GL/glut.h>
void display () /* callback function which is called when OpenGL needs to update the display */
{
 glClearColor (1.0,1.0,0.0,0.0);/*default color –black..... Now set to YELLOW */
 glClear (GL_COLOR_BUFFER_BIT);/*Clear the window-set the color of pixels in buffer*/
 glFlush(); /* Force update of screen */
}
void main (int argc, char **argv)
{
        glutInit (&argc, argv); /* Initialise OpenGL */
        glutCreateWindow ("Example1"); /* Create the window */
        glutDisplayFunc (display); /* Register the "display" function */
        glutMainLoop (); /* Enter the OpenGL main loop */
}
```

Activity 1:
Change the window size to 500, 500

Change the window position to 100, 100

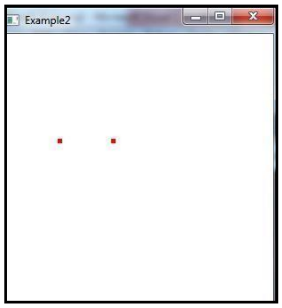
Change the window color to CYAN

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

void display (void)

{

        glClearColor (0.0,1.0,1.0,0.0);

        glClear (GL_COLOR_BUFFER_BIT);

        glFlush();

}

void main (int argc, char **argv)

{

        glutInit (&argc, argv); /* Initialise OpenGL */

        glutInitWindowSize(500,500);

        glutInitWindowPosition(100,100);

        glutCreateWindow ("Activity1"); /* Create the window */

        glutDisplayFunc (display); /* Register the "display" function */

        glutMainLoop (); /* Enter the OpenGL main loop */

}
```



Drawing pixels/points :

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

void display()

{

 glClear(GL_COLOR_BUFFER_BIT);

 glBegin(GL_POINTS);

 glVertex2i(100,300);

 glVertex2i(201,300);

 glEnd();

 glFlush();

}

void myinit()

{

 glClearColor(1.0,1.0,1.0,1.0);// set the window color to white

 glColor3f(1.0,0.0,0.0);// set the point color to red (RGB)

 glPointSize(5.0);// set the pixel size

 gluOrtho2D(0.0,500.0,0.0,500.0);// coordinates to be used with the
viewport(left,right,bottom,top)

}

void main(int argc, char** argv)

{

 glutInit(&argc,argv);

 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);// sets the initial display mode, GLUT
single-default

 glutInitWindowSize(300,300);

 glutInitWindowPosition(0,0);

 glutCreateWindow("Example2");

 glutDisplayFunc(display);

 myinit();
```



```
 glutMainLoop();

}
```

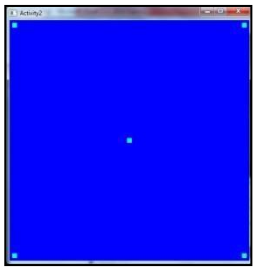
Activity 2

Change the window color to BLUE

Change the point color to CYAN

Change the point width to 10

Draw FIVE points: at 4 corners of the window and one more at the Centre of the window.

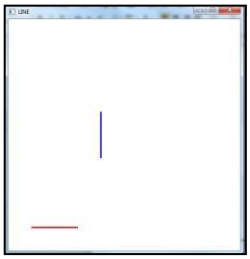


```
#include<GL/glut.h>

void display()

{

glClear(GL_COLOR_BUFFER_BIT);

glBegin(GL_POINTS);

glVertex2i(10,10);

glVertex2i(250,250);

glVertex2i(10,490);

glVertex2i(490,490);

glVertex2i(490,10);

 glEnd();

 glFlush();

}

void myinit()

{

 glClearColor(0.0,0.0,1.0,0.0);// set the window color to blue

 glColor3f(0.0,1.0,1.0);// set the point color to cyan (RGB)

 glPointSize(10.0);

 gluOrtho2D(0.0,500.0,0.0,500.0);// coordinates to be used with the viewport
(left,right,bottom,top)

}

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

```
{
 glutInit(&argc,argv);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
 glutInitWindowSize(500,500);
 glutInitWindowPosition(0,0);
 glutCreateWindow("Activity2");
 glutDisplayFunc(display);
 myinit();
 glutMainLoop();
}
```

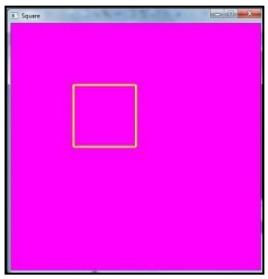
Drawing lines :

```
#include<GL/glut.h>
void display()
{
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0,0.0,0.0);  //draw the line with red color
glLineWidth(3.0); // Thickness of line
glBegin(GL_LINES);
  glVertex2d (50,50);       // to draw horizontal line in red color
  glVertex2d (150,50);
  glColor3f(0.0,0.0,1.0); //draw the line with blue color
  glVertex2d (200,200); // to draw vertical line in blue color
  glVertex2d (200,300);
  glEnd();
 glFlush();
}
```



```
void myinit()
{
 glClearColor(1.0,1.0,1.0,1.0);
 glColor3f(1.0,0.0,0.0);
 glPointSize(1.0);
 gluOrtho2D(0.0,500.0,0.0,500.0);
}
void main(int argc, char** argv)
{
 glutInit(&argc,argv);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
 glutInitWindowSize(500,500);
 glutInitWindowPosition(100,100);
 glutCreateWindow("LINE");
 glutDisplayFunc(display);
 myinit();
 glutMainLoop();
}
```

Activity 3



Change the window color to MAGENTA

Change the line color to YELLOW

Change the line width to width to 4

Draw a square using 4 lines

```
#include<GL/glut.h>
void display()
{
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0,1.0,0.0);
glLineWidth(4.0);
glBegin(GL_LINES);
glVertex2d (50, 100);
glVertex2d (100, 100);
glVertex2d (100, 100);
glVertex2d (100, 150);
glVertex2d (100, 150);
glVertex2d (50, 150);
glVertex2d (50, 150);
glVertex2d (50, 100);
glEnd();
glFlush();}
void myinit()
{
 glClearColor(1.0,0.0,1.0,1.0);
 gluOrtho2D(0.0,200.0,0.0,200.0);
}
```

```
void main(int argc, char** argv)
{
 glutInit(&argc,argv);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
 glutInitWindowSize(500,500);
 glutInitWindowPosition(10,100);
 glutCreateWindow("Square");
 glutDisplayFunc(display);
 myinit();
 glutMainLoop();
}
```

Drawing a square using LINE_LOOP:

```
#include<GL/glut.h>
void display()
{
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(0.0, 0.0, 1.0);
glLineWidth(3.0);
glBegin(GL_LINE_LOOP); // If you put GL_LINE_LOOP, it is only boundary.
glVertex2f(50, 50);
glVertex2f(200, 50);
glVertex2f(200, 200);
glVertex2f(50, 200);
glEnd();
glFlush();
}
```

```
void myinit()
{
 glClearColor(1.0,1.0,0.0,1.0);
 gluOrtho2D(0.0,499.0,0.0,499.0);
}
void main(int argc, char** argv)
{
 glutInit(&argc,argv);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
 glutInitWindowSize(300,300);
 glutInitWindowPosition(0,0);
 glutCreateWindow("LINE LOOP");
 glutDisplayFunc(display);
 myinit();
 glutMainLoop();
}
```

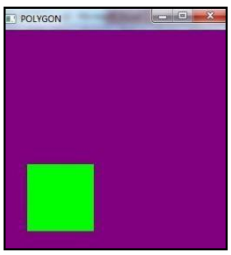
Activity 4

Change the window color to PURPLE

Change the line color to GREEN

Change the line width to width to 3

Draw a square using GL_POLYGON




```
#include<GL/glut.h>
void display()
{
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(0.0, 1.0, 0.0);// set line color to green
glLineWidth(3.0);
glBegin(GL_POLYGON);
```

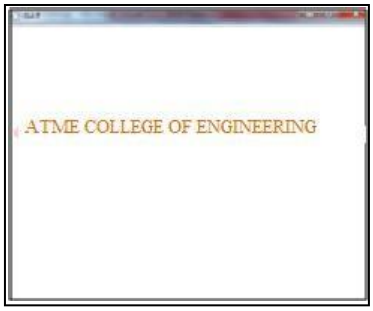
```
glVertex2f(50, 50);
glVertex2f(200, 50);
glVertex2f(200, 200);
glVertex2f(50, 200);
glEnd();
}
void myinit()
{
 glClearColor(0.5,0.0,0.5,1.0); // set window color to purple
 gluOrtho2D(0.0,499.0,0.0,499.0);
}
void main(int argc, char** argv)
{
 glutInit(&argc,argv);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
 glutInitWindowSize(300,300);
 glutInitWindowPosition(0,0);
 glutCreateWindow("POLYGON");
 glutDisplayFunc(display);
 myinit();
 glutMainLoop();
}
```

Drawing a right angled triangle using GL_TRINGLES

```
#include<GL/glut.h>
void display()

{

glClear(GL_COLOR_BUFFER_BIT);

glColor3f(0.0, 1.0, 0.0);// set line color to green

glLineWidth(3.0);

glBegin(GL_POLYGON);

glVertex2f(50, 50);

glVertex2f(200, 50);

glVertex2f(200, 200);

glVertex2f(50, 200);

glEnd();

glFlush();

}

void myinit()
{
 glClearColor(0.0,0.0,0.0,0.0);
 gluOrtho2D(0.0,499.0,0.0,499.0);

}
void main(int argc, char** argv)
{

 glutInit(&argc,argv);

 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);

 glutInitWindowSize(300,300);

 glutInitWindowPosition(0,0);

 glutCreateWindow("TRIANGLE");

 glutDisplayFunc(display);

 myinit();

 glutMainLoop();

}
```

Writing Text

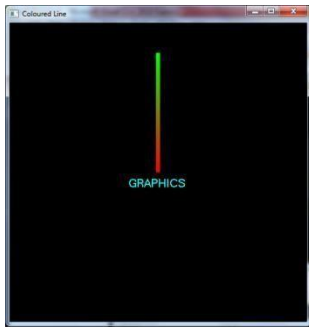
```
#include<GL/glut.h>

void output(GLfloat x,GLfloat y,char *text)

{

 char *p;

 glPushMatrix();

 glTranslatef(x,y,0);

 glScaled(0.2,0.2,0);

 for(p=text;*p;p++)

 glutStrokeCharacter(GLUT_STROKE_ROMAN,*p);

 glPopMatrix();

}

void display()

{

 glClear(GL_COLOR_BUFFER_BIT);

 output(10,300,"ATME COLLEGE OF ENGINEERING");

 glFlush();

}

void myinit()

{

 glClearColor(1.0,1.0,1.0,1.0);

 glColor3f(1.0,0.0,0.0);

 gluOrtho2D(0.0,499.0,0.0,499.0);

}

void main(int argc,char ** argv)

{

 glutInit(&2wargc,argv);

 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
```



```
 glutInitWindowSize(500,500);
 glutInitWindowPosition(0,0);
 glutCreateWindow("ATME");
 glutDisplayFunc(display);
 myinit();
 glutMainLoop();
}
```

Drawing colored line and writing Text

```
#include<GL/glut.h>
#include<string.h>
char *str= "GRAPHICS";
void display()
{
int i;
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0,0.0,0.0);
glLineWidth(10.0);
glBegin(GL_LINES);
glVertex2f(0.0,0.0);
glColor3f(0.0,1.0,0.0);
glVertex2f(0.0,0.8);
glEnd();
glColor3f(0.0,1.0,1.0);
glRasterPos2f(-0.2,-0.1);
//font type character to be displayed
for(i=0;i<strlen(str);i++)
glutBitmapCharacter(GLUT_BITMAP_HELVETICA_18,str[i]);
glFlush();
```



```
}
void myinit()
{
glClearColor(0.0,0.0,0.0,0.0);
gluOrtho2D(-1.0,1.0,-1.0,1.0);
}

void main(int argc, char **argv)
{
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_RGB|GLUT_SINGLE);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(0,0);
        glutCreateWindow("Coloured Line");
        myinit();
        glutDisplayFunc(display);
        glutMainLoop();
}
```

Activity 5

Change the window color to BLACK

Set the font to GLUT_STROKE_MONO_ROMAN

Display “GRAPICS IS FUN!” in YELLOW color

Display “REALLY FUN!” in RED color in the next line


```
#include<GL/glut.h>
void output(GLfloat x,GLfloat y,char *text)
```

```
{
 char *p;
 glPushMatrix();
 glTranslatef(x,y,0);
 glScaled(0.2,0.2,0);
 for(p=text;*p;p++)
 glutStrokeCharacter( GLUT_STROKE_MONO_ROMAN,*p);
 glPopMatrix();
}
void display()
{
 glClear(GL_COLOR_BUFFER_BIT);
 output(70,300,"GRAPHICS IS FUN!");
 glColor3f(1.0,0.0,0.0);
 output(120,250,"REALLY FUN!");
 glFlush();
}
void myinit()
{
 glClearColor(0.0,0.0,0.0,0.0);
 glColor3f(1.0,1.0,0.0);
 gluOrtho2D(0.0,499.0,0.0,499.0);
}
void main(int argc,char ** argv)
{
 glutInit(&argc,argv);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
 glutInitWindowSize(500,500);
```

```
 glutInitWindowPosition(0,0);
 glutCreateWindow("STROKE TEXT");
 glutDisplayFunc(display);
 myinit();
 glutMainLoop();
}
```

Drawing a colored square

```
#include<GL/glut.h>
void display()
{
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(0.0, 1.0, 0.0); // Green
glBegin(GL_POLYGON);
glVertex2f(100, 100);
glColor3f(1.0,0.0,0.0); // Red
glVertex2f(300, 100);
glColor3f(0.0,0.0,1.0); // Blue
glVertex2f(300, 300);
glColor3f(1.0,1.0,0.0); // Yellow
glVertex2f(100, 300);
glEnd();
glFlush();
}
void myinit()
{
 glClearColor(0.0,0.0,0.0,1.0);
```

```
 glColor3f(1.0,0.0,0.0);// Red
 gluOrtho2D(0.0,499.0,0.0,499.0);
}
void main(int argc, char** argv)
{
glutInit(&argc,argv);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
 glutInitWindowSize(500,500);
 glutInitWindowPosition(0,0);
 glutCreateWindow("COLORED SQUARE");
 glutDisplayFunc(display);
 myinit();
 glutMainLoop();
}
```

Creating 2 view ports

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

void display()

{

 glClear(GL_COLOR_BUFFER_BIT);

glViewport (5,-150,400,400);

glBegin(GL_POLYGON);

glColor3f(1.0,0.0,0.0);

glVertex2f(90,250);

glColor3f(0.0,1.0,0.0);

glVertex2f(250,250);

glColor3f(0.0,0.0,1.0);

glVertex2f(175,400);

glEnd();

glViewport (300,300,400,400);

glBegin(GL_POLYGON);

glColor3f(1.0,0.0,0.0);

glVertex2f(50,50);

glColor3f(0.0,1.0,0.0);

glVertex2f(250,50);

glColor3f(0.0,0.0,1.0);

glVertex2f(250,250);

glColor3f(0.0,1.0,1.0);

glVertex2f(50,250);

glEnd();

glFlush();

}
```

```
voidmyinit()
{
 glClearColor(0.0,0.0,0.0,1.0);
 gluOrtho2D(0.0,499.0,0.0,499.0);
}
void main(intargc, char** argv)
{
 glutInit(&argc,argv);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
 glutInitWindowSize(500,500);
 glutInitWindowPosition(0,0);
 glutCreateWindow("2 VIEW PORTS");
 glutDisplayFunc(display);
 myinit();
 glutMainLoop();
}
```

1. PART A

Design, develop, and implement the following programs in C/C++ using OpenGL API.

1.1 Bresenham's Line Drawing

Implement Brenham's line drawing algorithm for all types of slope.

1.1.1 PREAMBLE

Bresenham's line algorithm is an algorithm that determines the points of an n-dimensional raster that should be selected in order to form a close approximation to a straight line between two points. It is commonly used to draw line primitives in a bitmap image (e.g. on a computer screen), as it uses only integer addition, sub-traction and bit shifting, all of which are very cheap operations in standard computer architectures.

It is an incremental error algorithm. It is one of the earliest algorithms developed in the field of computer graphics. An extension to the original algorithm may be used for drawing circles.

While algorithms such as Wu's algorithm are also frequently used in modern computer graphics because they can support antialiasing, the speed and simplicity of Bresenham's line algorithm means that it is still important.

The algorithm is used in hardware such as plotters and in the graphics chips of modern graphics cards. It can also be found in many software graphics libraries. Because the algorithm is very simple, it is often implemented in either the rmware or the graphics hardware of modern graphics cards.

Concepts Used:

- Line equations
- Interger arithmetic

Algorithm:

Pseudo Code OpenGL commands Familiarised :

- glutInit
- glutInitDisplayMode
- glClearColor
- glColor
- glPointSize
- glOrtho

Program 1: Implement Brenham's line drawing algorithm for all types of slope.

Program Objective:

- creation of 2D objects
- Implement GLU and GLUT functions
- To have the detailed knowledge of the graphics Brenham's line algorithm.

/* Brenham's line */

```
#include<GL/glut.h>
#include<stdio.h>
int x1,x2,y1,y2;
void myInit()
{
glClearColor(0.0,0.0,0.0,0.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0,500,0,500);
}

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

void draw_line(int x1,int x2,int y1,int y2)
{
int dx,dy,i,e;
int incx,incy,inc1,inc2;
int x,y;
dx=x2-x1;
dy=y2-y1;
if(dx<0)
dx=-dx;
if(dy<0)
dy=-dy;
incx=1;
if(x2<x1)
incx=-1;
incy=1;
if(y2<y1)
incy=-1;
x=x1;
y=y1;
if(dx>dy)
{
draw_pixel(x,y);
e=2*dy-dx;
```

```
inc1=2*(dy-dx);
inc2=2*dy;
for(i=0;i<dx;i++)
{
if(e>=0)
{
y+=incy;
e+=inc1;
}

else
e+=inc2;
x+=incx;
draw_pixel(x,y);
}
}
else
{
draw_pixel(x,y);
e=2*dx-dy;
inc1=2*(dx-dy);
inc2=2*dx;
for(i=0;i<dy;i++)
{
if(e>=0)
{
x+=incx;
e+=inc1;
}
else
e+=inc2;
y+=incy;
draw_pixel(x,y);
}
}
}

void myDisplay()
{
glClear(GL_COLOR_BUFFER_BIT);
draw_line(x1,x2,y1,y2);
glFlush();
}
int main(int argc,char **argv)
{
printf("enter x1,x2,y1,y2\n");
scanf("%d%d%d%d",&x1,&x2,&y1,&y2);
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(500,500);
glutInitWindowPosition(0,0);
glutCreateWindow("win");
```

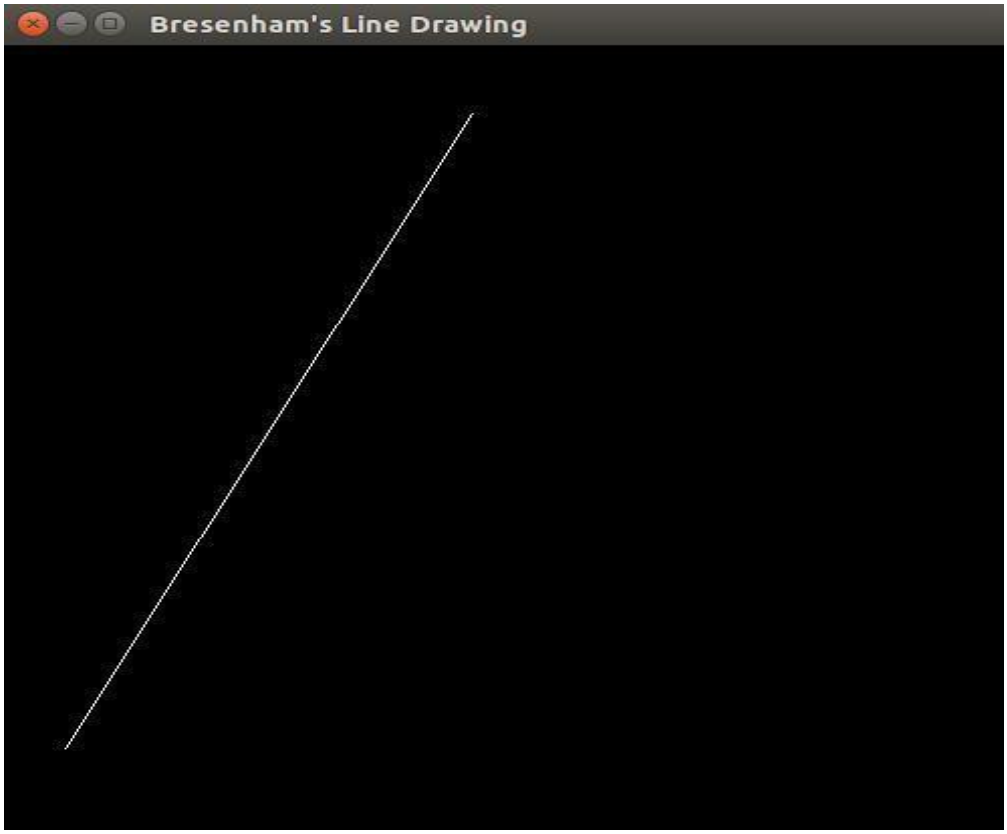
```
glutDisplayFunc(myDisplay);
myInit();
glEnable(GL_DEPTH_TEST);
glClearColor(0.0,0.0,0.0,0.0);
glutMainLoop();
}
```

RUN:

```
gcc 1.c -lglut -lGL –lGLU
./a.out
```

```
Enter x1, x2, y1, y2
40 140
40 280
```

SAMPLE OUTPUT:



Program Outcome:

- Ability to Design and develop 2D objects for different graphics applications.
- Use matrix algebra in computer graphics and implement Brenham's line algorithm.
- Analyze and evaluate the use of openGL methodss in practical applications of 2D representations.

1.2 Triangle Rotation

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

1.2.1 PREAMBLE

In linear algebra, a rotation matrix is a matrix that is used to perform a rotation in Euclidean space. For example the matrix rotates points in the xy-Cartesian plane counterclockwise through an angle about the origin of the Cartesian coordinate system. To perform the rotation using a rotation matrix R, the position of each point must be represented by a column vector v, containing the coordinates of the point.

A rotated vector is obtained by using the matrix multiplication Rv. Since matrix multiplication has no effect on the zero vector (i.e., on the coordinates of the origin), rotation matrices can only be used to describe rotations about the origin of the coordinate system.

Rotation matrices provide a simple algebraic description of such rotations, and are used extensively for computations in geometry, physics, and computer graphics. In 2-dimensional space, a rotation can be simply described by an angle of rotation, but it can be also represented by the 4 entries of a rotation matrix with 2 rows and 2 columns. In 3-dimensional space, every rotation can be interpreted as a rotation by a given angle about a single fixed axis of rotation (see Euler's rotation theorem), and hence it can be simply described by an angle and a vector with 3 entries. However, it can also be represented by the 9 entries of a rotation matrix with 3 rows and 3 columns.

The notion of rotation is not commonly used in dimensions higher than 3; there is a notion of a rotational displacement, which can be represented by a matrix, but no associated single axis or angle.

Concepts Used: Algorithm: Pseudo Code OpenGL commands Familiarised:

- glutInit
- glutInitDisplayMode
- glClearColor
- glColor
- glPointSize
- glOrtho

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

Program Objective:

- creation of 2D objects
- Create 2D Triangle with basic transformation operations like rotation.
- Use mathematical and theoretical principles i.e transformation matrices of computer graphics to draw and rotate Triangle object.

/* Triangle Rotation */

```
#include<GL/glut.h>
#include<stdio.h>
#include<math.h>
GLfloat triangle[3][3]={{200.0,300.0,250.0},{200.0,200.0,300.0},{0.0,0.0,0.0}};
float matRot[3][3]={{0.0},{0.0},{0.0}};
GLfloat res[3][3]={{0.0},{0.0},{0.0}};
GLfloat theta=0.0;
void Rotate_point()
{
int i,j,l;
matRot[0][0]=cos(theta);
matRot[0][1]=-sin(theta);
matRot[0][2]=0;
matRot[1][0]=sin(theta);
matRot[1][1]=cos(theta);
matRot[1][2]=0;
matRot[2][0]=0;
matRot[2][1]=0;
matRot[2][2]=1;
for(i=0;i<3;i++)
for(j=0;j<3;j++)
{
res[i][j]=0;
for(l=0;l<3;l++)
{
res[i][j]+=matRot[i][l]*triangle[l][j];
}
}
}
void inp_triangle()
{
glBegin(GL_LINE_LOOP);
glVertex2f(triangle[0][0],triangle[1][0]);
glVertex2f(triangle[0][1],triangle[1][1]);
glVertex2f(triangle[0][2],triangle[1][2]);
glEnd();
glFlush();
}
void Out_triangle()
{
```

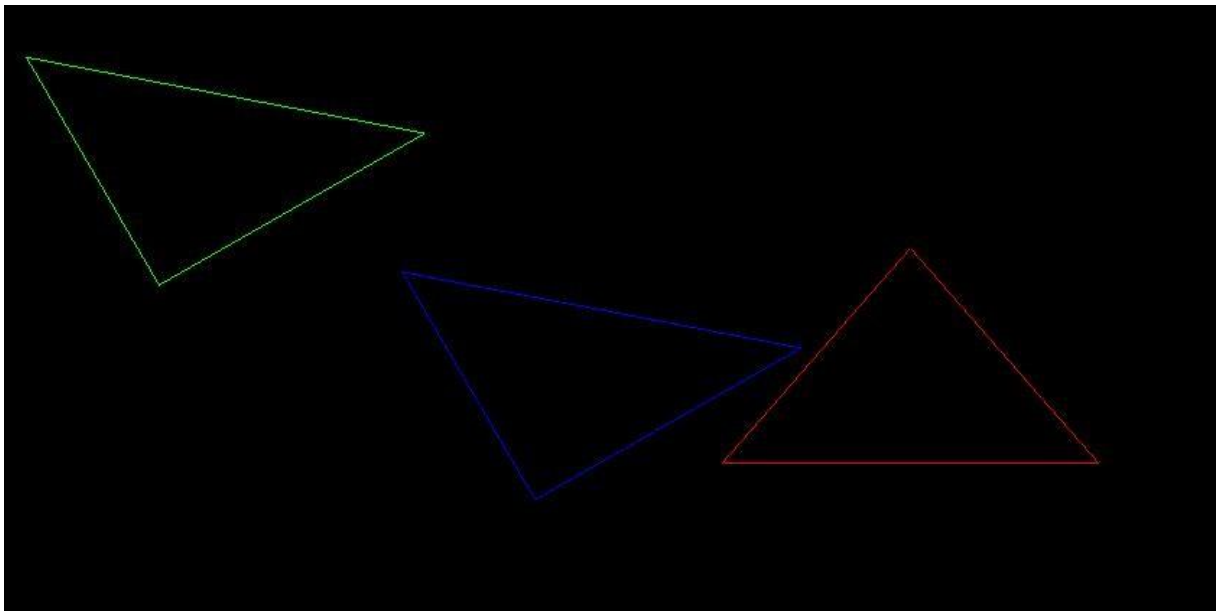
```
glBegin(GL_LINE_LOOP);
glVertex2f(res[0][0],res[1][0]);
glVertex2f(res[0][1],res[1][1]);
glVertex2f(res[0][2],res[1][2]);
glEnd();
glFlush();
}

void display()
{
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0,0.0,0.0);
inp_triangle();
glColor3f(0.0,1.0,0.0);
glTranslatef(50.0,0.0,0.0);
Rotate_point(); Out_triangle();
glPushMatrix();
glColor3f(0.0,0.0,1.0);
glTranslatef(300.0,100.0,0.0);
Rotate_point();
glTranslatef(-200.0,-200.0,0.0);
Out_triangle();
glPopMatrix();
glFlush();
}
void reshape(GLint w,GLint h)
{
glViewport(0,0,w,h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0,500,0,500);
glutPostRedisplay();
glFlush();
}
int main(int argc,char **argv)
{
printf("enter the angle of rotation\n");
scanf("%f",&theta);
theta=theta*3.14/180;
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(800,800);
glutInitWindowPosition(0,0);
glutCreateWindow("2d");
glutDisplayFunc(display);
glEnable(GL_DEPTH_TEST);
glutReshapeFunc(reshape);
glClearColor(0.0,0.0,0.0,0.0);
glutMainLoop();
}
```

RUN:

g++ 2.cc -lglut -lGL -lGLU
./a.out

enter the angle of rotation
45

SAMPLE OUTPUT:



Program Outcome:

- Ability to Design and develop 2D objects for different graphics applications.
- Use matrix algebra in computer graphics and implement fundamental transformations involved in transformation operation.
- Ability to draw objects like Triangle using basic objects like points and lines.
- Analyze and evaluate the use of openGL methodss in practical applications of 2D representations.

1.3 Color cube and spin

Draw a color cube and spin it using OpenGL transformation matrices.

1.3.1 PREAMBLE

Concepts Used:
- Data structures for representing a cube
- Rotation Transformation

Algorithm:
Modeling a color cube with simple data structures
- Define vertices with centre of cube as origin
- Define normals to identify faces
- Define colors

Rotating the cube
- Define angle of rotation [or accept it as input]
- Define/Input rotation axis
- Use swap-buffer to ensure smooth rotation

Using callback functions to indicate either or both of the following:
- Angle of rotation
- Axis of rotation

1. Define global arrays for vertices and colors
GLfloat vertices [][3] = {{-1.0,-1.0,-1.0}, {1.0,-1.0,-1.0}, {1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0, 1.0}, {1.0,-1.0, 1.0}, {1.0, 1.0, 1.0}, {-1.0, 1.0, 1.0}};
GLfloat colors[][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0}, {1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0}, {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};

2. Draw a polygon from a list of indices into the array vertices and use color corresponding to first index

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

3. Draw cube from faces.

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

```
polygon(1,2,6,5);
polygon(4,5,6,7);
polygon(0,1,5,4);
}
```

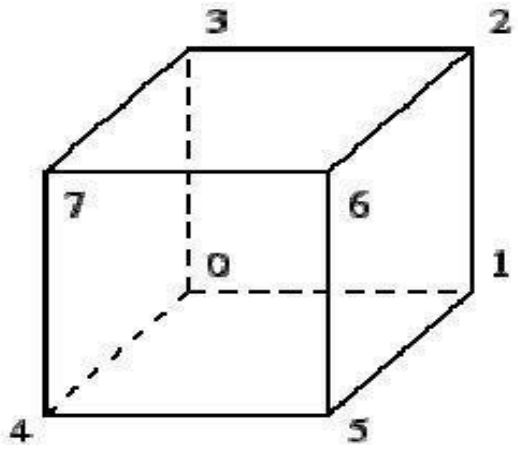
4. Define the Display function to clear frame buffer, Z-buffer, rotate cube and draw, swap buffers.

```
void display(void)
{
/* display callback, clear frame buffer and z buffer,
rotate cube and draw, swap buffers */
glLoadIdentity();
glRotatef(theta[0], 1.0, 0.0, 0.0);
glRotatef(theta[1], 0.0, 1.0, 0.0);
glRotatef(theta[2], 0.0, 0.0, 1.0);
colorcube();
glutSwapBuffers();

}
```

5. Define the spin cube function to spin cube 2 degrees about selected axis.
6. Define mouse callback to select axis about which to rotate.
7. Define reshape function to maintain aspect ratio.

```
void myReshape(int w, int h)
{
glViewport(0, 0, w, h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <= h)
glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
else
glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
glMatrixMode(GL_MODELVIEW);
}
```

8. Define main function to create window and to call all function.



OpenGL Commands Familiarized:

- glRotate
- glMatrixMode
- glMouseFunc. and/or glKeyboardFunc
- glEnable
- glutSwapBuffer
- glutIdleFunc

Program 3: Draw a color cube and spin it using OpenGL transformation matrices.

Program Objective:

- creation of 3D objects
- Create 3D-color cube with basic transformation operations like rotation.
- Use mathematical and theoretical principles i.e transformation matrices of computer graphics to draw the color cube object.

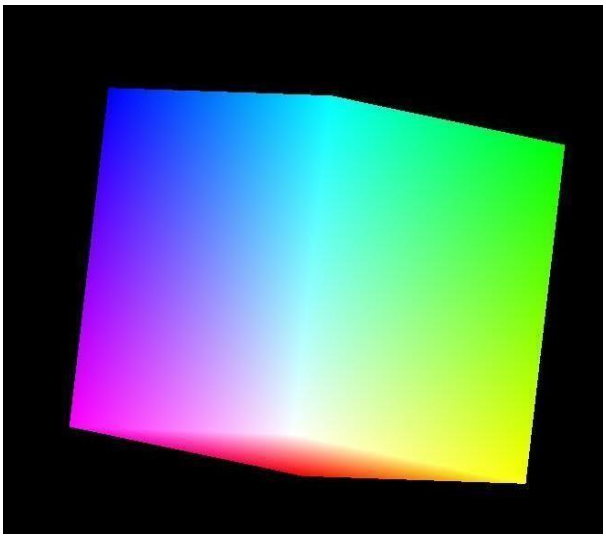
/* Color cube and spin */

```
#include<GL/glut.h>
#include<stdio.h>
GLfloat vertices[][3]={{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},{1.0,1.0,-1.0},{-1.0,1.0,-1.0},
{-1.0,-1.0,1.0},{1.0,-1.0,1.0},{1.0,1.0,1.0},{-1.0,1.0,1.0}};
GLfloat colors[][3]={{0.0,0.0,0.0},{0.0,0.0,1.0},{0.0,1.0,0.0},{0.0,1.0,1.0},{1.0,0.0,0.0},
{1.0,0.0,1.0},{1.0,1.0,0.0},{1.0,1.0,1.0}};
static GLfloat theta[]={0.0,0.0,0.0};
static GLint axis=2;
static GLdouble viewer[]={0,0,5};
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();
}
void colorcube()
{
polygon(0,3,2,1);
polygon(2,3,7,6);
polygon(1,2,6,5);
polygon(0,4,5,1);
polygon(4,5,6,7);
polygon(0,3,7,4);
}

void display()
{ glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
;
glLoadIdentity();
glRotatef(theta[0],1.0,0.0,0.0);
glRotatef(theta[1],0.0,1.0,0.0);
glRotatef(theta[2],0.0,0.0,1.0);
colorcube();
```

```
glFlush();
glutSwapBuffers();
}
void spincube()
{
theta[axis]+=2.0;
if(theta[axis]>360.0)
theta[axis]-=360.0;
glutPostRedisplay();
}
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;
spincube();
}
void myreshape(int w,int h)
{
glViewport(0,0,w,h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if(w<=h)
glOrtho(-2.0,2.0,-2.0*(GLfloat)h/(GLfloat)w,2.0*(GLfloat)h/(GLfloat)w,-10.0,10.0);
else
glOrtho(-2.0*(GLfloat)w/(GLfloat)h,2.0*(GLfloat)w/(GLfloat)h,-2.0,2.0,-10.0,10.0);
glMatrixMode(GL_MODELVIEW);
glutPostRedisplay();
}
int main(int argc,char**argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
glutCreateWindow("Color cube");
glutReshapeFunc(myreshape);
glutDisplayFunc(display);
glutIdleFunc(spincube);
glutMouseFunc(mouse);
glEnable(GL_DEPTH_TEST);
glClearColor(0.0,0.0,0.0,0.0);
glutMainLoop();
}
```

RUN:
gcc 3.c -lglut -lGL -lGLU
./a.out

SAMPLE OUTPUT:



Program Outcome:

- Ability to Design and develop 3D objects for different graphics applications.
- Use matrix algebra in computer graphics and implement fundamental transformations involved in transformation operation.
- Ability to draw objects like cube using basic objects like points and lines.

1.4 Color cube with perspective viewing

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

1.4.1 PREAMBLE

Concepts Used:

- Data structures for representation of a cube
- Perspective viewing
- Defining and moving the camera
- Input functions using Keyboard and mouse

Algorithm:

1. Modeling a color cube with simple data structures

- Define vertices with centre of cube as origin
- Define normals to identify faces
- Define colors

2. Camera Manipulations

- Define initial camera position - take care to define outside the cube.

3. Callback Function

- Handling mouse inputs - use the 3 mouse buttons to define the 3 axes of rotation
- Handling Keyboard Inputs - use the 3 pairs of keys to move the viewer along +ive and -ive directions of the X and Y axes respectively.

Pseudo Code

1. Define global arrays for vertices and colors

```
GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0}, {1.0,1.0,-1.0},
{-1.0,1.0,-1.0}, {-1.0,-1.0,1.0}, {1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-
1.0,1.0,1.0}};
GLfloat colors[][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
{1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0}, {1.0,0.0,1.0},
{1.0,1.0,1.0}, {0.0,1.0,1.0}};
```

2. Draw a polygon from a list of indices into the array vertices and use color corresponding to first index

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

3. Draw cube from faces.

```
void colorcube( )
{
polygon(0,3,2,1);
```

```
polygon(2,3,7,6);
polygon(0,4,7,3);
polygon(1,2,6,5);
polygon(4,5,6,7);
polygo n(0,1,5,4);
}
```

4. Intialize the theta value and initial viewer location and axis static GLfloat theta[]={0.0,0.0,0.0}
static Glint axis =2;
static GLdouble viewer[]={0.0,0.0,5,0}

5. Define display function to upadate viewer position in model view matrix

```
void display(void)
{
/* display callback, clear frame buffer and z buffer,
rotate cube and draw, swap buffers */
glLoadIdentity();
gluLookAt(viewer[0],viewer[1],viewer[2],0.0,0.0,0.0,0.0,1.0,0.0);
glRotatef(theta[0], 1.0, 0.0, 0.0);
glRotatef(theta[1], 0.0, 1.0, 0.0);
glRotatef(theta[2], 0.0, 0.0, 1.0);
colorcube();
glutSwapBuffers();
}
```

6. Define mouse callback function to rotate about axis.
7. Define key function to move viewer position .use x,X,y,Y z,Z keys to move viewer position.
8. Define reshape function to maintain aspect ratio and then use perspective view.
9. Define main fuction to create window and to call all function.

OpenGL Commands Familiarized:

- glutInit
- glutInitDisplayMode
- glClearColor
- glColor
- glPointSize
- gluOrtho2D

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

Program Objective:

- creation of 3D objects
- Create 3D-color cube with basic transformation operations like rotation.
- Use mathematical and theoretical principles i.e transformation matrices of computer graphics to draw and rotate the color cube object.
- Use matrix algebra in computer graphics and implement fundamental algorithms and transformations involved in viewing models.

/* Color cube with perspective viewing */

```
#include<GL/glut.h>
#include<stdio.h>
GLfloat vertices[][3]={{-1.0,-1.0,-1.0},
{1.0,-1.0,-1.0},
{1.0,1.0,-1.0},
{-1.0,1.0,-1.0},
{-1.0,-1.0,1.0},
{1.0,-1.0,1.0},
{1.0,1.0,1.0},
{-1.0,1.0,1.0}};
GLfloat colors[][3]={{0.0,0.0,0.0},
{0.0,0.0,1.0},
{0.0,1.0,0.0},
{0.0,1.0,1.0},
{1.0,0.0,0.0},
{1.0,0.0,1.0},
{1.0,1.0,0.0},
{1.0,1.0,1.0}};
static GLfloat theta[]={0.0,0.0,0.0};
static GLint axis=2;
static GLdouble viewer[]={0,0,5};
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();
}
void colorcube()
{
polygon(0,3,2,1);
```

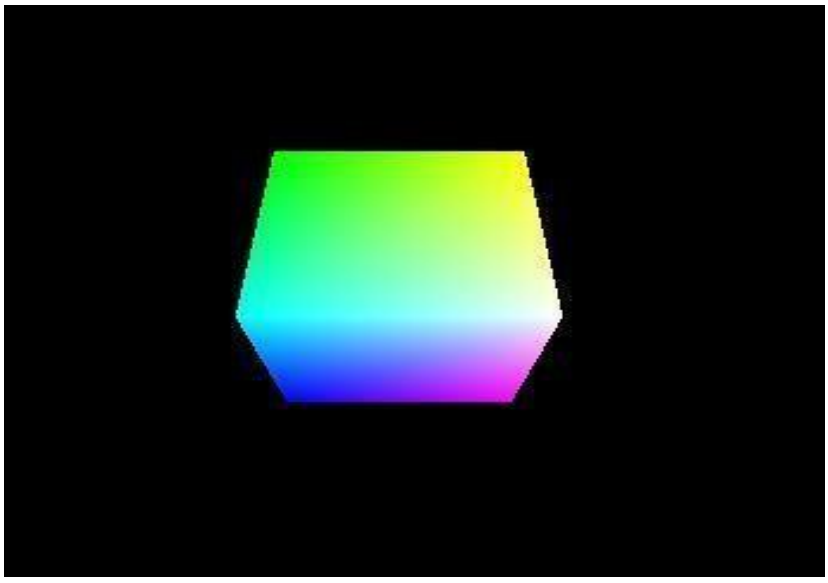
```
polygon(2,3,7,6);
polygon(1,2,6,5);
polygon(0,4,5,1);
polygon(4,5,6,7);
polygon(0,3,7,4);
}
void display()
{ glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
;
glLoadIdentity();
gluLookAt(viewer[0],viewer[1],viewer[2],0,0,0,0,1,0);
glRotatef(theta[0],1.0,0.0,0.0);
glRotatef(theta[1],0.0,1.0,0.0);
glRotatef(theta[2],0.0,0.0,1.0);
colorcube();
glFlush();
glutSwapBuffers();
}
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*(GLfloat)w/(GLfloat)h,2*(GLfloat)w/(GLfloat)h,-2,2,2,20);
glMatrixMode(GL_MODELVIEW);
glutPostRedisplay();
}
void myMouse(int btn,int state,int x,int y)
{
if(btn==GLUT_LEFT_BUTTON&&state==GLUT_DOWN)
axis=0;
if(btn==GLUT_RIGHT_BUTTON&&state==GLUT_DOWN)
axis=1;
if(btn==GLUT_MIDDLE_BUTTON&&state==GLUT_DOWN)
axis=2;
theta[axis]+=2.0;
if(theta[axis]>360.0)
theta[axis]-=360.0;
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();
```

```
}
int main(int argc,char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
glutCreateWindow("colorcube");
glutReshapeFunc(myreshape);
glutDisplayFunc(display);
glutMouseFunc(myMouse);
glutKeyboardFunc(keys);
glEnable(GL_DEPTH_TEST);
glClearColor(0.0,0.0,0.0,0.0);
glutMainLoop();
}
```

RUN:

```
gcc 4.c -lglut -lGL -lGLU
./a.out
```

SAMPLE OUTPUT:



Program Outcome:

- Ability to Design and develop 3D objects for different graphics applications.
- Use matrix algebra in computer graphics and implement fundamental transformations involved in viewing models.
- Analyze and evaluate the use of openGL methodss in practical applications of 3D representations.

1.4 Cohen-Sutherland

Clip a line using Cohen-Sutherland algorithm.

1.5.1PREAMBLE

The Cohen-Sutherland algorithm uses a divide-and-conquer strategy. The line segment's endpoints are tested to see if the line can be trivially accepted or rejected. If the line cannot be trivally accepted or rejected, an intersection of the line with a window edge is determined and the trivial reject/accept test is repeated.

This process is continued until the line is accepted. To perform the trivial acceptance and rejection tests, we extend the edges of the window to divide the plane of the window into the nine regions. Each end point of the line segment is then assigned the code of the region in which it lies.

Concepts Used:

Data structures for representing a square and line

Algorithm:

Every line endpoint is assigned a 4 bit Region code. The appropriate bit is set depending on the location of the endpoint with respect to that window component as shown below:

1001	1000	1010
0001	0000	0010
0101	0100	0110

Endpoint Left of window then set bit 1 Endpoint Right of window then set bit 2 Endpoint Below window then set bit 3 Endpoint Above window then set bit 4

1. Given a line segment with endpoint P1=(x1,y1) and P2=(x2,y2)

2. Compute the 4-bit codes for each endpoint. If both codes are 0000,(bitwise OR of the codes yields 0000) line lies completely inside the window: pass the endpoints to the draw routine.
If both codes have a 1 in the same bit position (bitwise AND of the codes is not 0000), the line lies outside the window. It can be trivially rejected.

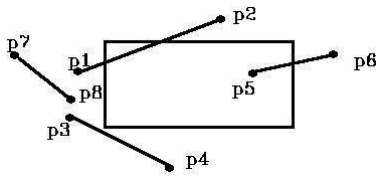
3. If a line cannot be trivially accepted or rejected, at least one of the two endpoints must lie outside the window and the line segment crosses a window edge. This line must be clipped at the window edge before being passed to the drawing routine.

4. Examine one of the endpoints, say. Read's 4-bit code in order: Left-to-Right, Bottom-to-Top.

5. When a set bit (1) is found, compute the intersection I of the corresponding window edge with the line from to. Replace with i and repeat the algorithm.

1.6

Example1

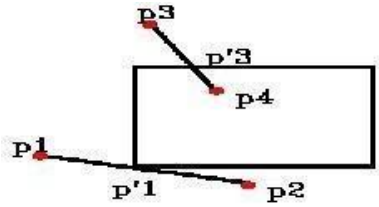


Can determine the bit code by testing the endpoints with window as follows:
If x is less than Xwmin then set bit 1
If x is greater than Xwmax then set bit 2
If y is less than Ywmin then set bit 3
If y is greater than Ywmax then set bit 4

Note: can use 4 element Boolean matrix and set C [Left] = true / false (1/0). If both endpoints = 0000 (in window) then display line. If both endpoints have a bit set in same position (P7, P8) then the line is completely outside the window and is rejected.So: can do logical AND of region codes and reject if result is 0000 Can do logical OR of region codes and accept if result = 0000
For the rest of the lines we must check for intersection with window. May still be outside, e.g. P3 - P4 in the above image. If point is to Left of window then compute intersection with Left window boundary. Do the same for Right, Bottom, and Top.

Then recompute region code restest. So the algorithm is as follows:
1. Compute region code for endpoints
2. Check for trivial accept or reject
3. If step 2 unsuccessful then compute window intersections in order:\\ Left, Right, Bottom, Top (only do 1)
4. Repeat steps 1, 2,3 until done.

Example2:



I. 1) P1 = 0001 2) no 3) P1 = P'1 P2 = 1000
1) P'1= 0000 2) no 3) P2 = P' 2 P2 = 1000
III. 1) P'1 = 0000 2) Yes - accept & display P'2 = 0000

Look at how to compute the line intersections for
P'1: m = dy/dx = (y1 - y'1)/(x1 - x'1) P1(x1, y1)P'1(x'1, y'1) or y'1 = y1 + m(x'1 - x1)
but for Left boundary x'1 = Xwmin for Right boundary x'1 = Xwmax

Similarly for Top / Bottom, e.g. P'3
x'3 = x3 + (y'3 - y3) / m
For Top y'3 = Ywmax for Bottom y'3 = Ywmin

OpenGL Commands Familiarized :

- glMatrixMode
- glLoadIdentity
- gluOrtho2D
- glFlush
- glColor3f
- glBegin

Program 5: Clip a lines using Cohen-Sutherland algorithm

Program Objective:

- creation of 2D objects
- To have the detailed knowledge of the graphics Cohen-Sutherland line-clipping algorithm.
- Use mathematical and theoretical principles of computer graphics to draw the and clip the line object.
- Implement GLU and GLUT functions

/* Cohen-Sutherland */

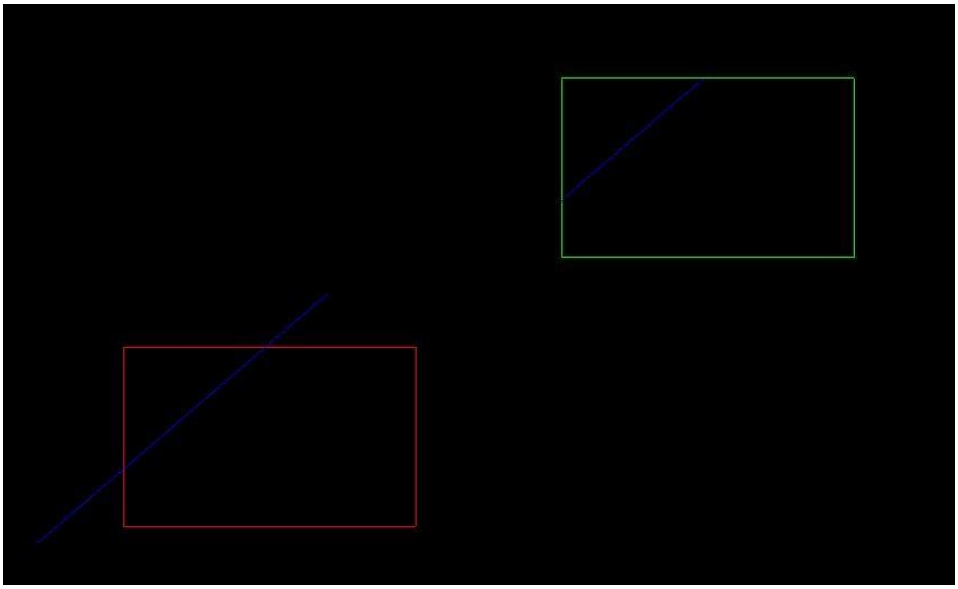
```
#include<GL/glut.h>
#include<stdio.h>
#define outcode int
const int L=1,R=2,B=4,T=8;
float x0,y00,x1,y11;
float xmin=50.0,ymin=50.0,xmax=150.0,ymax=150.0,xvmin=200,yvmin=200,xvmax=300,
yvmax=300;

outcode compute(float x,float y)
{
outcode code=0;
if(y>ymax)   code=T;
else if(y<ymin) code=B;
else if(x>xmax) code=R;
else if(x<xmin) code=L;
return code;
}

void cohen(float x0,float y00,float x1,float y11)
{
outcode outcode0,outcode1,outcodeout;
int accept=0,done=0;
outcode0=compute(x0,y00);
outcode1=compute(x1,y11);
do
{
if(!(outcode0| outcode1))
{
accept=1,done=1;
}
else if(outcode0&outcode1)
done=1;
else
{
outcodeout=outcode0?outcode0:outcode1;
float x,y;
if(outcodeout&T)
{
```

```
x=x0+(ymax-y00)*(x1-x0)/(y11-y00);y=ymax;
}
else if(outcodeout&B)
{
x=x0+(ymin-y00)*(x1-x0)/(y11-y00);
y=ymin;
}
else if(outcodeout&R)
{
y=y00+(xmax-x0)*(y11-y00)/(x1-x0);x=xmax;
}
else
{
y=y00+(xmin-x0)*(y11-y00)/(x1-x0);
x=xmin;
}
if(outcodeout==outcode0)
{
x0=x;
y00=y;
outcode0=compute(x0,y00);
}
else
{
x1=x,y11=y;
outcode1=compute(x1,y11);
}
}
}
while(!done);
if(accept)
{
float sx,sy,vx0,vx1,vy00,vy11;
sx=(xvmax-xvmin)/(xmax-xmin);
sy=(yvmax-yvmin)/(ymax-ymin);
vx0=xvmin+(x0-xmin)*sx;
vy00=yvmin+(y00-ymin)*sy;
vx1=xvmin+(x1-xmin)*sx;
vy11=yvmin+(y11-ymin)*sy;
glColor3f(0.0,1.0,0.0);
glBegin(GL_LINE_LOOP);
glVertex2f(xvmin,yvmin);
glVertex2f(xvmax,yvmin);
glVertex2f(xvmax,yvmax);
glVertex2f(xvmin,yvmax);
glEnd();
glColor3f(0.0,0.0,1.0);
glBegin(GL_LINES);
glVertex2f(vx0,vy00);
glVertex2f(vx1,vy11);
```

```
glEnd();
}
}

void display()
{
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0,0.0,0.0);
glBegin(GL_LINE_LOOP);
glVertex2f(xmin,ymin);
glVertex2f(xmax,ymin);
glVertex2f(xmax,ymax);
glVertex2f(xmin,ymax);
glEnd();
glColor3f(0.0,0.0,1.0);
glBegin(GL_LINES);
glVertex2f(x0,y00);
glVertex2f(x1,y11);
glEnd();
cohen(x0,y00,x1,y11);
glFlush();
}
void
init(){ glClearColor(0.0,0.0,0.0,0.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0,499.0,0.0,499.0);
}

int main(int argc,char
**argv){ printf("Enter the line
coordinates\n");
scanf("%f%f%f%f",&x0,&y00,&x1,&y11);
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(500,500);
glutInitWindowPosition(0,0);
glutCreateWindow("cohen");
glutDisplayFunc(display);
init();
glutMainLoop();
}
```

RUN:
gcc 5.c -lglut -lGL -lGLU
./a.out

Enter the line coordinates
20 40
120 180
SAMPLE OUTPUT:



Program Outcome:

- Ability to Design and develop 2D objects for different graphics applications.
- Use matrix algebra in computer graphics and implement Cohen-Sutherland line-clipping algorithm. .
- Analyze and evaluate the use of openGL methodss in practical applications of 2D representations.

1.6 Tea pot

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.

1.6.1 PREAMBLE

Concepts Used:
- Translation, Scaling
- Material Properties, Light Properties
- Depth in 3D Viewing
- Using predefined GLU library routines

Algorithm:

Algorithm[High Level]:
1. Recognise the objects in the problem domain and their components
- Teapot
- Table
  Table top
  4Legs
- Walls
  Left wall
  Right wall
  Floor

2. If depth testing is NOT used ensure that the background is drawn first
3. Enable lighting first - since lighting is applied at the time objects are drawn.
4. Draw the objects.

Algorithm[Low Level]:
Main Function:
- Initialization for Display, Mode , Window
- Enable lighting, shading, depth test
- Register display and callback function
- Define viewport appropriately
- Call mainloop

Display function:
- Define Material properties
- Define lighting properties
- Set the camera by defining
- projection parameters
- camera parameters
- Plan the required centre position for each of the components
- Draw each component using translation, scaling and rotation as required

Pseudo Code

1. Include glut header files.
2. Define wall function as following

```
void wall(double thickness)
{
glPushMatrix();
glTranslated(0.5,0.5*thickness,0.5);
glScaled(1.0,thickness,1.0);
glutSolidCube(1.0);
gPopMatrix();
}
```

3. Draw one tableleg using tableLeg function as following

```
void tableLeg(double thick,double len)
{
glpushMatrix();
glTranslated(0,len/2,0);
glScaled(thick,len,thick);
glutsolidCube(1.0);
glPopMatrix();
}
```

4. Draw the table using table function.
i) draw the table top using glutSolidCube(1.0) function.Before this use gltranslated and glScaled function to fix table in correct place.
ii) Draw four legs by calling the function tableleg .before each call use gltranslated function to fix four legs in correct place.

5. Define the function displaySolid
i) Initialize the properties of the surface material and set the light source properties.
ii) Set the camera position.
iii) Draw the teapot using glutSolidTeapot(0.08).before this call gltranslated and glRotated.
iv) Call table function and wall function
v) Rotate wall about 90 degree and then call wall function.
vi) Rotate wall about -90 degree and then call wall function.

6. Define the main function to create window and enable lighting function using
   glEnable(GL_LIGHTING)

OpenGL Commands Familiarized :
- glPushMatrix, glPopMatrix
- glTranslate, glScale, glRotate
- glMaterial, glLight
- gluLookAt
- glutSolidTeapot, glutSolidCube

Program 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.

Program Objective:

- creation of 3D objects
- Use 3D teapot objects to draw a simple shaded scene consisting of a tea pot on a table.
- Use matrix algebra in computer graphics and implement fundamental algorithms and transformations involved in viewing models.

```
/* Tea pot */
#include<GL/glut.h>
void wall(double thickness)
{
glPushMatrix();
glTranslated(0.5,0.5*thickness,0.5);
glScaled(1.0,thickness,1.0);
glutSolidCube(1.0);
glPopMatrix();
}
void tableleg(double thick,double len)
{
glPushMatrix();
glTranslated(0,len/2.0,0);
glScaled(thick,len,thick);
glutSolidCube(1.0);
glPopMatrix();
}
void table(double topwid,double topthick,double legthick,double leglen)
{
glPushMatrix();
glTranslated(0,leglen,0);
glScaled(topwid,topthick,topwid);
glutSolidCube(1.0);
glPopMatrix();
double dist=0.95*topwid/2-legthick/2;
glPushMatrix();
glTranslated(dist,0,dist);
tableleg(legthick,leglen);
glTranslated(0,0,-2*dist);
tableleg(legthick,leglen);
glTranslated(-2*dist,0,2*dist);
tableleg(legthick,leglen);
glTranslated(0,0,-2*dist);
tableleg(legthick,leglen);
glPopMatrix();
}
```

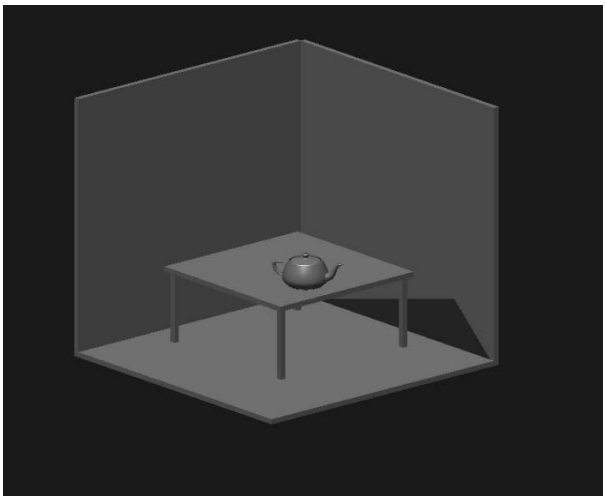
```
void display()
{
GLfloat mat_ambient[]={0.7f,0.7f,0.7f,1.0f};
GLfloat mat_diffuse[]={0.5f,0.5f,0.5f,1.0f};
GLfloat mat_specular[]={1.0f,1.0f,1.0f,1.0f};
GLfloat mat_shininess[]={50.0f};
glMaterialfv(GL_FRONT,GL_AMBIENT,mat_ambient);
glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
GLfloat Lightintensity[]={1.0f,1.0f,1.0f,1.0f};
GLfloat Lightposition[]={2.0f,6.0f,3.0f,0.0f};
glLightfv(GL_LIGHT0,GL_POSITION,Lightposition);
glLightfv(GL_LIGHT0,GL_DIFFUSE,Lightintensity);
double winht=1.0;
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-winht*64/48.0,winht*64/48.0,-winht,winht,0.1,100.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(2.3,1.3,2.0,0.0,0.25,0.0,0.0,1.0,0.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glPushMatrix();
glTranslated(0.5,0.38,0.5);
glRotated(60,0,1,0);
glutSolidTeapot(0.08);
glPopMatrix();
glPushMatrix();
glTranslated(0.4,0,0.4);
table(0.6,0.02,0.02,0.3);
glPopMatrix();
wall(0.02);
glPushMatrix();
glRotated(90.0,0.0,0.0,1.0);
wall(0.02);
glPopMatrix();
glPushMatrix();
glRotated(-90.0,1.0,0.0,0.0);
wall(0.02);
glPopMatrix();
glFlush();
}
int main(int argc,char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB |GLUT_DEPTH);
glutCreateWindow("Tea pot");
glutInitWindowSize(500,500);
glutInitWindowPosition(0,0);
glutDisplayFunc(display);
glEnable(GL_DEPTH_TEST);
glEnable(GL_LIGHTING);
```

```
glEnable(GL_LIGHT0);
glEnable(GL_NORMALIZE);
glClearColor(0.0,0.0,0.0,0.0);
glutMainLoop();
}
```

RUN:

```
gcc 6.c -lglut -lGL -lGLU
./a.out
```

SAMPLE OUTPUT:



Program Outcome:

- Ability to Design and develop 3D objects for different graphics applications.
- Use matrix algebra in computer graphics and implement fundamental transformations involved in viewing models.
- Ability to draw objects like table basic objects like points and lines.
- Analyze and evaluate the use of openGL methodss in practical applications of 3D representations.

1.7 3D Sierpinski gasket

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.

1.7.1 PREAMBLE

Sierpinski's Triangle is a very famous fractal that's been seen by most advanced math students. This fractal consists of one large triangle, which contains an infinite amount of smaller triangles within. The infinite amount of triangles is easily understood if the fractal is zoomed in many levels. Each zoom will show yet more previously unseen triangles embedded in the visible ones.

Creating the fractal requires little computational power. Even simple graphing calculators can easily make this image. The fractal is created pixel by pixel, using random numbers; the fractal will be slightly different each time due to this. Although, if you were to run the program repeatedly, and allow each to use an infinite amount of time, the results would be always identical. No one has an infinite amount of time, but the differences in the finite versions are very small.

A fractal is generally "a rough or fragmented geometric shape that can be split into parts, each of which is (at least approximately) a reduced-size copy of the whole".

To generate this fractal, a few steps are involved.

We begin with a triangle in the plane and then apply a repetitive scheme of operations to it (when we say triangle here, we mean a blackened, filled-in' triangle). Pick the midpoints of its three sides. Together with the old vertices of the original triangle, these midpoints define four congruent triangles of which we drop the center one. This completes the basic construction step. In other words, after the first step we have three congruent triangles whose sides have exactly half the size of the original triangle and which touch at three points which are common vertices of two contiguous triangles. Now we follow the same procedure with the three remaining triangles and repeat the basic step as often as desired. That is, we start with one triangle and then produce 3, 9, 27, 81, 243, triangles, each of which is an exact scaled down version of the triangles in the preceding step.

Concepts Used:

- Data structures for representing 3D vertices
- Tetrahedron sub-division using mid-points

Algorithm:

- Input: Four 3D vertices of tetrahedron, no. of divisions
- Recursively sub-divide the each triangle by finding the mid-point Pseudo Code

1. Define and initializes the array to hold the vertices as follows:
GLfloatvertices[4][3]={{0.0,0.0,0.0},{25.0,50.0,10.0},{50.0,25.0,25.0},{25.0,10.0,25.0}};

2. Define and initialize initial location inside tetrahedron. GLfloat p[3] = {25.0,10.0,25.0};

3. Define Triangle function that uses points in three dimensions to display one triangle
   Use glBegin(GL_POLYGON)
   glVertex3fv(a) glVertex3fv(b)
   glVertex3fv(c) to display points. glEnd();
4. Subdivide a tetrahedron using divide_triangle function

i) if no of subdivision(m) > 0 means perform following functions
ii) Compute six midpoints using for loop.
iii) Create 4 tetrahedrons by calling divide_triangle function
iv) Else draw triangle at end of recursion.

5. Define tetrahedron function to apply triangle subdivision to faces of tetrahedron by calling the function divide_triangle.
6. Define display function to clear the color buffer and to call tetrahedron function.
7. Define main function to create window and to call display function.

OpenGL commands Familiarised :

- glutInit
- glutInitDisplayMode
- glClearColor
- glColor
- glPointSize
- glOrtho

Program 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.

Program Objective:

- creation of 3D objects
- Implement GLU and GLUT functions
- Create Sierpinski gasket
- Implement the concepts of recursion to create Sierpinski gasket

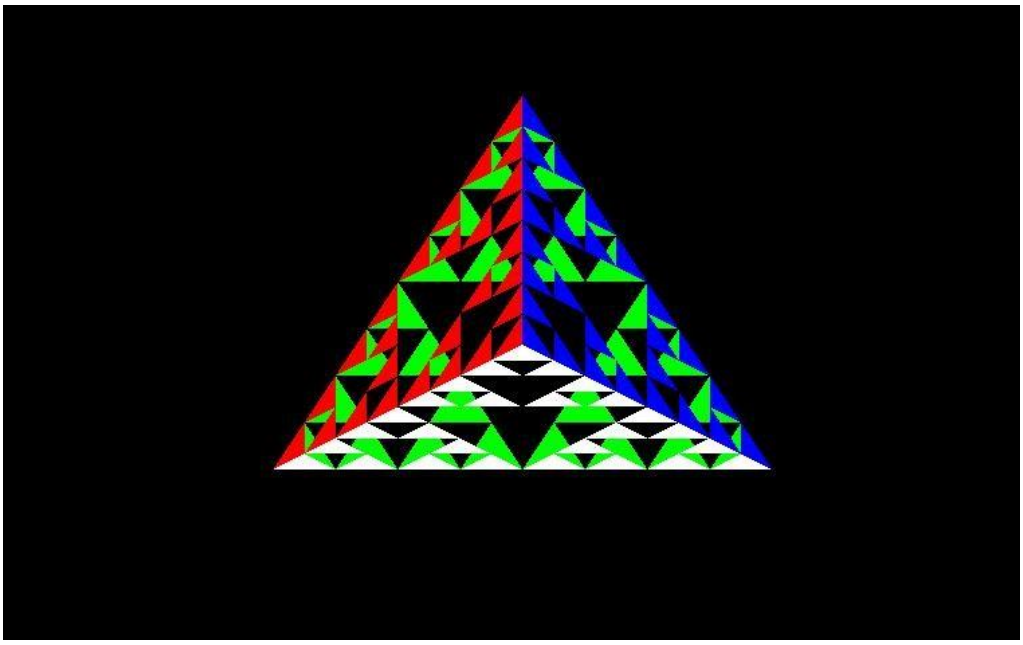
```
/* 3 D Sierpinski gasket */
#include <GL/glut.h>
#include <stdio.h>
typedef float point[3];
point v[] = {{0.0,0.0,1.0},{0.0,1.0,0.0},{-1.0,-0.5,0.0},{1.0,-0.5,0.0}};
int n;
void triangle(point a, point b, point c)
{
glBegin(GL_POLYGON);
glVertex3fv(a);
glVertex3fv(b);
glVertex3fv(c);
  glEnd();
}
void divide(point a, point b, point c,int m)
{
point v1, v2, v3;
int j;
if(m > 0)
{
for(j = 0; j < 3; j++)
v1[j] = (a[j] + b[j]) / 2;
for(j = 0; j < 3; j++)
v2[j] = (a[j] + c[j]) / 2;
for(j=0; j<3;  j++)
v3[j] = (b[j] + c[j]) / 2;
divide(a, v1, v2, m - 1);
divide(c, v2, v3, m - 1);
divide(b, v3, v1, m - 1);
}
else(triangle(a, b, c));
}

void tetra(int m)
{
glColor3f(1.0, 0.0, 0.0);
divide(v[0], v[1], v[2], m);
glColor3f(0.0, 1.0, 0.0);
divide(v[3], v[2], v[1], m);
glColor3f(0.0, 0.0, 1.0);
```

```
divide(v[0], v[3], v[1], m);
glColor3f(1.0, 1.0, 1.0);
divide(v[0], v[2], v[3], m);
}

void display()
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
tetra(n);
glFlush();
}

void myreshape(int w, int h)
{
glViewport(0,0,w,h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if(w <= h)
glOrtho(-2.0,2.0,-2.0*(GLfloat)h/(GLfloat)w,2.0*(GLfloat)h/(GLfloat)w,-10.0,10.0);
else
glOrtho(-2.0*(GLfloat)w/(GLfloat)h,2.0*(GLfloat)w/(GLfloat)h,-2.0,2.0,-10.0,10.0);
glMatrixMode(GL_MODELVIEW);
glutPostRedisplay();
}
int main(int argc, char **argv)
{
printf("Enter number of divisions");
scanf("%d", &n);
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
glutCreateWindow("GASKET");
glutReshapeFunc(myreshape);
glutDisplayFunc(display);
glEnable(GL_DEPTH_TEST);
glClearColor(0.0,0.0,0.0,0.0);
glutMainLoop();
 }
```

RUN:
gcc 7.c -lglut -lGL -lGLU
./a.out

Enter number of divisions
3

SAMPLE OUTPUT:



Program Outcome:

- Analyze and evaluate the use of openGL methodss in practical applications of 3D representations.
- Ability to explain the mathematical principles of recursion and to draw basic Sierpinski gasket object.

1.8 Bezier Curve

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

1.8.1 PREAMBLE

Bezier curve is discovered by the French engineer Pierre Bezier. These curves can be generated under the control of other points. Approximate tangents by using control points are used to generate curve. The Bezier curve can be represented mathematically as

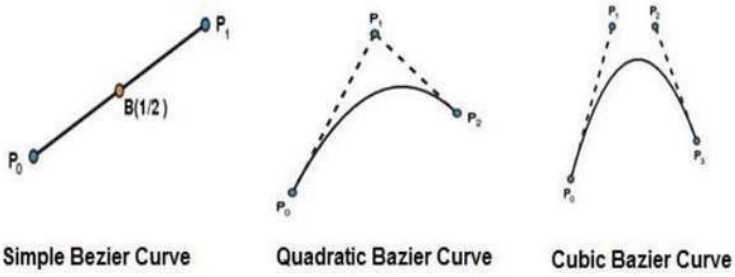
$$\sum_{k=0}^{n} P_i B_i^n(t)$$

Where pi is the set of points and Bni(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.

The simplest Bezier curve is the straight line from the point P0 to P1. A quadratic Bezier curve is determined by three control points. A cubic Bezier curve is deter-mined by four control points.



Simple Bezier Curve Quadratic Bazier Curve Cubic Bazier Curve

Bezier curves generally follow the shape of the control polygon, which consists of the segments joining the control points and always pass through the first and last control points. They are contained in the convex hull of their defining control points. The degree of the polynomial defining the curve segment is one less that the number of defining polygon point. Therefore, for 4 control points, the degree of the polynomial is 3, i.e. cubic polynomial. A Bezier curve generally follows the shape of the defining polygon. The direction of the tangent vector at the end points is same as that of the vector determined by first and last segments. The convex hull property for a Bezier curve ensures that the polynomial smoothly follows the control points. No straight line intersects a Bezier curve more times than it intersects its control polygon. They are invariant under an affine transformation. Bezier curves exhibit global control means moving a control point alters the shape of the whole curve. A given Bezier curve can be subdivided at a point t=t0 into two Bezier segments which join together at the point corresponding to the parameter value t=t0.

Concepts Used: Algorithm: OpenGL Commands Familiarized:

- glMatrixMode
- glLoadIdentity
- gluOrtho2D
- glFlush
- glColor3f
- glBegin

Program 8: Develop a menu driven program to animate a flag using Bezier Curve

Program Objective:

- creation of 2D objects
- Use mathematical and theoretical principles of computer graphics to animate a flag using Bezier Curve algorithm
- Implement GLU and GLUT functions

/* Bezier Curve */

```
#include<GL/glut.h>
#include<stdio.h>
#include<math.h>
#define  pi 3.14
struct
{
GLfloat x,y,z;
}
typedef wcpt;

void bino(GLint n,GLint *c)
{
GLint k,j;
for(k=0;k<=n;k++)
{ c[k]=
1;
for(j=n;j>=k+1;j--)c[k]*=j;
for(j=n-k;j>=2;j--)
c[k]/=j;
}
}

void combez(GLfloat u,wcpt *bezpt,GLint nctrpt,wcpt *ctrlpt,GLint *c)
{
GLint k,n=nctrpt-1;
GLfloat blendfunc;
bezpt->x=bezpt->y=bezpt->z=0.0;
for(k=0;k<nctrpt;k++)
{
blendfunc=c[k]*pow(u,k)*pow(1-u,n-k);
bezpt->x+=ctrlpt[k].x*blendfunc;
bezpt->y+=ctrlpt[k].y*blendfunc;
bezpt->z+=ctrlpt[k].z*blendfunc;
}
}
```

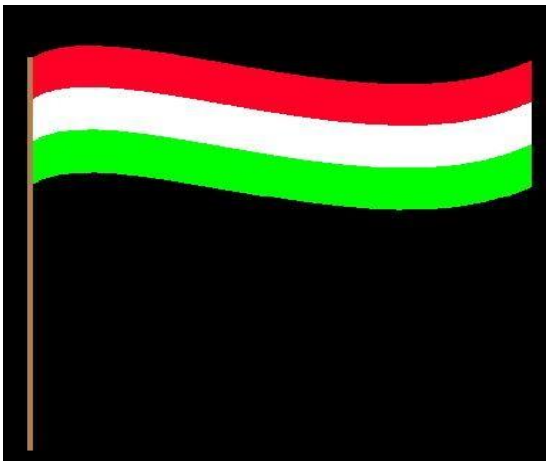
```
void bezier(wcpt *ctrlpt,GLint nctrpt,GLint nBezcurpts)
{
wcpt bezcurpt;
GLfloat u;
GLint *c,k;
c=new GLint[nctrpt];
bino(nctrpt-1,c);
glBegin(GL_LINE_STRIP);
for(k=0;k<=nBezcurpts;k++)
{
u=GLfloat(k)/GLfloat(nBezcurpts);
combez(u,&bezcurpt,nctrpt,ctrlpt,c);
glVertex2f(bezcurpt.x,bezcurpt.y);
}
glEnd();
delete[]c;
}

void display()
{
GLint nctrpt=4,nBezcurpts=20;
static float theta=0;
wcpt ctrlpt[4]={{20,100,0},{30,110,0},{50,90,0},{60,100,0}};
ctrlpt[1].x+=10*sin(theta*pi/180);
ctrlpt[1].y+=5*sin(theta*pi/180);
ctrlpt[2].x-=10*sin((theta+30)*pi/180);
ctrlpt[2].y-=10*sin((theta+30)*pi/180);
ctrlpt[3].x-=4*sin(theta*pi/180);
ctrlpt[3].y-=sin((theta+30)*pi/180);
theta+=3;
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0,1.0,1.0);
glPushMatrix();
glLineWidth(5);
glColor3f(1,0,0.15);
for(int i=0;i<8;i++)
{
glTranslatef(0,-0.8,0);
bezier(ctrlpt,nctrpt,nBezcurpts);
}
glColor3f(1,1,1);
for(int i=0;i<8;i++)
{
glTranslatef(0,-0.8,0);
bezier(ctrlpt,nctrpt,nBezcurpts);
}
glColor3f(0,1,0);
for(int i=0;i<8;i++)
{
glTranslatef(0,-0.8,0);
bezier(ctrlpt,nctrpt,nBezcurpts);
}
```

```
glPopMatrix();
glColor3f(0.7,0.5,0.3);
glLineWidth(5);
glBegin(GL_LINES);
glVertex2f(20,100);
glVertex2f(20,40);
glEnd();
glFlush();
glutPostRedisplay();
glutSwapBuffers();
}

void Reshape(GLint w,GLint h)
{
glViewport(0,0,w,h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0,130,0,130);
glutPostRedisplay();
glFlush();
}

int main(int argc,char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
glutInitWindowSize(500,500);
glutInitWindowPosition(0,0);
glutCreateWindow("bazier");
glutDisplayFunc(display); glutReshapeFunc(Reshape);
glClearColor(0.0,0.0,0.0,0.0);
glutMainLoop();
}
```

RUN:

```
g++ 8.cc -lglut -lGL –lGLU
./a.out
```

SAMPLE OUTPUT



Program Outcome:

- Ability to Design and develop 2D objects for different graphics applications.
- Use matrix algebra in computer graphics and implement Bezier Curve algorithm.
- Analyze and evaluate the use of openGL methodss in practical applications of 2D representations.

1.9 Scan-line

Develop a menu driven program to fill the polygon using scan line algorithm.

1.9.1 PREAMBLE

The scan conversion algorithm works as follows
i. Intersect each scanline with all edges
ii. Sort intersections in x
iii. Calculate parity of intersections to determine in/out iv. Fill the "in" pixels

Special cases to be handled: i. Horizontal edges should be excluded ii. For vertices lying on scanlines, i. count twice for a change in slope. ii. Shorten edge by one scanline for no change in slope

- Coherence between scanlines tells us that
- Edges that intersect scanline y are likely to intersect y + 1
- X changes predictably from scanline y to y + 1

We have 2 data structures: Edge Table and Active Edge Table
- Traverse Edges to construct an Edge Table
- Eliminate horizontal edges
- Add edge to linked-list for the scan line corresponding to the lower vertex.

Store the following:
- yupper : last scanline to consider
- xlower: starting x coordinate for edge
- 1/m: for incrementing x; compute
- Construct Active Edge Table during scan conversion. AEL is a linked list of active edges on the current scanline, y. Each active edge line has the following information
- yupper : last scanline to consider
- xlower: edge's intersection with current y
- 1/m: x increment

The active edges are kept sorted by x Concepts Used:
- Use the straight line equation y2-y1=m(x2-x1) to compute the x values corresponding to lines increment in y value.
- Determine which are the left edge and right edges for the the closed polygon.for each value of y within the polygon.
- Fill the polygon for each value of y within the polygon from x=left edge to x=right edge.

Algorithm:
1. Set y to the smallest y coordinate that has an entry in the ET; i.e, y for the first nonempty bucket.
2. Initialize the AET to be empty.
3. Repeat until the AET and ET are empty:
3.1 Move from ET bucket y to the AET those edges whose y_min = y (entering edges).
3.2 Remove from the AET those entries for which y = y_max (edges not involved in the next scanline), the sort the AET on x (made easier because ET is presorted).

3.3 Fill in desired pixel values on scanline y by using pairs of x coordinates from AET.
3.4 Increment y by 1 (to the coordinate of the next scanline).
3.5 For each nonvertical edge remaining in the AET, update x for the new y.

Extensions:
1. Multiple overlapping polygons - priorities
2. Color, patterns Z for visibility
// Function scanfill
Inputs: vertices of the polygon.
Output: filled polygon.
Processing :
1. Initialize array LE to 500 and RE to 0 for all values of y(0-->499)
2. Call function EDGEDETECT for each edge of the polygon one by one to set the value of x for each value of y within that line.
3. For each value of y in the screen draw the pixels for every value of x provided. It is greater than right edge value of x.

//function Display
Inputs:Globally defined vertices of Polygon
Output: Filled polygon display

Processing:
1. Draw the polygon using LINE,LOOP
2. Fill the polygon using SCANFILL

//Function EDGEDETECT:
Inputs:
1. End co-ordinates of edge
2. Address of LE and RE
3. Output: The updated value of x for left edge of the polygon and the right edge of the Polygon.

Processing:
1. Find the inverse of the slope.
2. Starting from the lesser integer value of y to the greater integer value of y.
3. Compute the value of x for left edge and right edge and update the value of x for both the edges.

OpenGL Commands Familiarized:
- glutInit
- glutInitDisplayMode
- glClearColor
- glColor
- glPointSize
- gluOrtho2D

Program 9: Develop a menu driven program to fill the polygon using scan line algorithm

Program Objective:

- creation of 2D objects
- To have the detailed knowledge of the graphics scan-line area filling algorithm
- Use mathematical and theoretical principles of computer graphics to draw and fill colors to the object.
- Implement GLU and GLUT functions

/* Scan-line */

```
#include<GL/glut.h>
float x1,x2,x3,x4,y1,y2,y3,y4;
void draw_pixel(int x,int y)
{
glColor3f(1.0,0.0,1.0);
glPointSize(1.0);
glBegin(GL_POINTS);
glVertex2i(x,y);
glEnd();
}
void edgedetect(float x1,float y1,float x2,float y2,int *le,int *re)
{
float temp,x,mx;
int i;
if(y1>y2)
{
temp=x1,x1=x2,x2=temp;
temp=y1,y1=y2,y2=temp;
}
if(y1==y2)
mx=x2-x1;
else
mx=(x2-x1)/(y2-y1);  x=x1;
for(i=(int)y1;i<(int)y2;i++)
{
if(x<(float)le[i])
le[i]=(int)x;
if(x>(float)re[i])
re[i]=(int)x;
x+=mx;
}
}
void Scanfill(float x1,float y1,float x2,float y2,float x3,float y3,float x4,float y4)
{
int le[500],re[500],i,j;
for(i=0;i<500;i++)
le[i]=500,re[i]=0;
edgedetect(x1,y1,x2,y2,le,re);
```

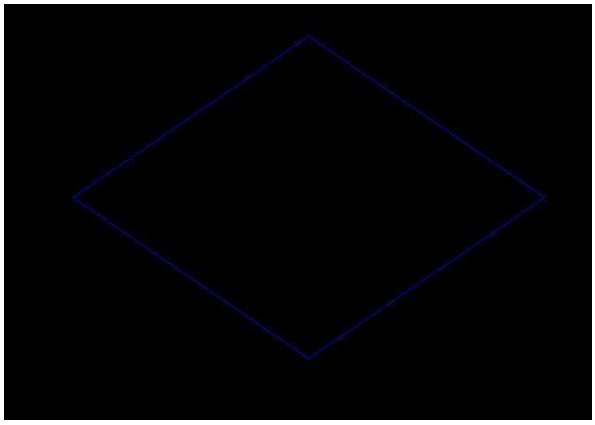
```
edgedetect(x2,y2,x3,y3,le,re);
edgedetect(x3,y3,x4,y4,le,re);
edgedetect(x4,y4,x1,y1,le,re);
for(j=0;j<500;j++)
{
if(le[j]<=re[j])
for(i=le[j];i<re[j];i++)
draw_pixel(i,j);
}
}
void filloption(GLint selectedoption)
{
switch(selectedoption)
{
case 1:
Scanfill(x1,y1,x2,y2,x3,y3,x4,y4);
break;
case 2:
exit(0);
 break;
}
}
void Reshape(GLint w,GLint h)
{
glViewport(0,0,w,h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0,500,0,500);
glutPostRedisplay();
glFlush();
}
void display()
{ x1=250.0,y1=200.0,x2=150.0,y2=300.0,x3=250.0,y3=400.0,x4=350.0,y4=300.
0;
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(0.0,0.0,1.0);
glBegin(GL_LINE_LOOP);
glVertex2f(x1,y1);
glVertex2f(x2,y2);
glVertex2f(x3,y3);
glVertex2f(x4,y4);
glEnd(); glFlush();
}

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

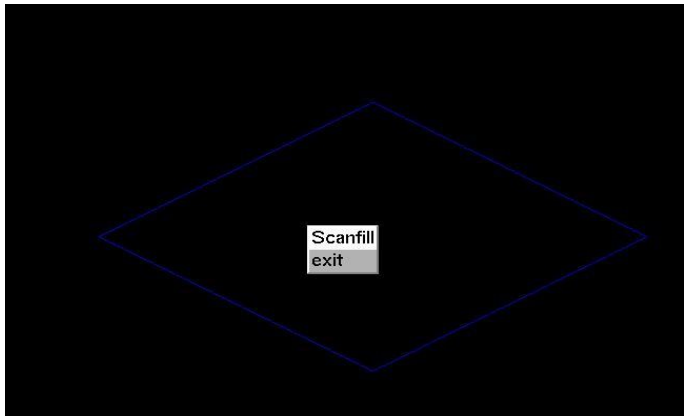
```
glutDisplayFunc(display);
glutCreateMenu(filloption);
glutAddMenuEntry("Scanfill",1);
glutAddMenuEntry("exit",2);
glutAttachMenu(GLUT_RIGHT_BUTTON);
glutReshapeFunc(Reshape);
glClearColor(0.0,0.0,0.0,0.0);
glutMainLoop();
}
```

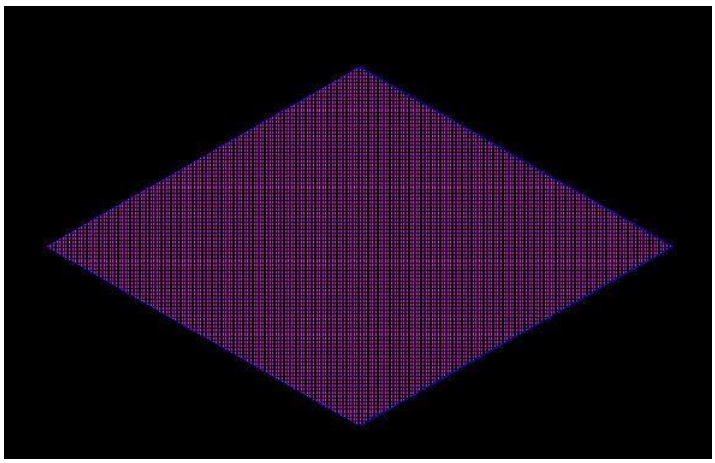
RUN:

```
gcc 9.c -lglut -lGL -lGLU
./a.out
```

SAMPLE OUTPUT







Program Outcome:

- Ability to Design and develop 2D objects for different graphics applications.
- Use matrix algebra in computer graphics and implement scan-line area filling algorithm.
- Analyze and evaluate the use of openGL methodss in practical applications of 2D representations.

PART B

Develop a suitable Graphics package to implement the skills learnt in the theory and the exercises indicated in PART A. Use the OpenGL. Criteria for CG Project Students per batch should be THREE or less.

One Three-Dimensional OpenGL Graphics Project using features from at least THREE CATEGORIES listed below:

Category I
- Input and Interaction
- Menus, Display Lists

Category II
- Transformations
- Camera Movement

Category III
- Coloring
- Texturing
- Lighting/ Shading

Category IV
- Animation
- Hidden Surface Removal

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 Pro-gram 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 breif 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 mathemati-cian 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 magnication 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 ll 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 o 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 o-ce 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 specic to object (i).
// Set model transform for object (i) using glTranslatef, glScalef, glRotatef, and/orequivalent.
// Issue rendering commands for object (i).
// Pop relevant stacks, (e.g., glPopMatrix, glPopAttrib.)
// End for loop.
// Swap buers.
}
```

13. What support for OpenGL does Open,Net,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 in-stead. It's more exible 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 specically, 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 diering values. For example, you may nd it convenient to model an airplane's controls in centimeters, its fuselage in meters, and a world to y around in kilometers. OpenGL's ModelView matrix can then scale these dierent coordinate sys-tems 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 eld 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 eld 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 le format have to do with OpenGL?

Answer: .gl les have nothing to do with OpenGL, but are sometimes confused with it. .gl is a le 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 OpenGLtrademark.

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 *argcp, 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 specied 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.