R V COLLEGE OF ENGINEERING
(Autonomous Institution Affiliated to VTU, Belagavi)
BENGALURU – 560 059

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

COMPUTER GRAPHICS LAB (12CS72)

VII SEM - B.E (CSE)

INSTRUCTORS MANUAL

2018-19

R V College of Engineering, Bengaluru- 560 059

(Autonomous Institution Affiliated to VTU, Belagavi)

Department of Computer Science and Engineering

Vision

To achieve leadership in the field of Computer Science & Engineering by strengthening fundamentals and facilitating interdisciplinary sustainable research to meet the ever growing needs of the society.

Mission

- To evolve continually as a centre of excellence in quality education in computers and allied fields.
- To develop state-of-the-art infrastructure and create environment capable for interdisciplinary research and skill enhancement.
- To collaborate with industries and institutions at national and international levels to enhance research in emerging areas.
- To develop professionals having social concern to become leaders in top-notch industries and/or become entrepreneurs with good ethics.

Program Educational Objectives (PEOs)

PEO1: Develop Graduates capable of applying the principles of mathematics, science, core engineering and Computer Science to solve real-world problems in interdisciplinary domains.

PEO2: To develop the ability among graduates to analyze and understand current pedagogical techniques, industry accepted computing practices and state-of-art technology.

PEO3: To develop graduates who will exhibit cultural awareness, teamwork with professional ethics, effective communication skills and appropriately apply knowledge of societal impacts of computing technology.

PEO4: To prepare graduates with a capability to successfully get employed in the right role / become entrepreneurs to achieve higher career goals or takeup higher education in pursuit of lifelong learning.

Program Outcomes (POs)

PO1: Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization for the solution of complex engineering problems.

PO2: Problem analysis: Identify, formulate, research literature, and analyse 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 public health and safety, and 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 the 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 Specific Outcomes (PSOs)

PSO1: System Analysis and Design

The student will be able to:

1. Recognize and appreciate the need of change in computer architecture, data organization and analytical methods in the evolving technology.
2. Learn the applicability of various systems software elements for solving design problems.
3. Identify the various analysis & design methodologies for facilitating development of high quality system software products with focus on performance optimization.
4. Display team participation, good communication, project management and document skills.

PSO2: Product Development

The student will be able to:

1. Demonstrate the use of knowledge and ability to write programs and integrate them with the hardware/software products in the domains of embedded systems, databases /data analytics, network/web systems and mobile products.
2. Participate in planning and implement solutions to cater to business – specific requirements displaying team dynamics and professional ethics.
3. Employ state-of-art methodologies for product development and testing / validation with focus on optimization and quality related aspects.

Course Outcomes (COs)

CO1: Understand and explore the basic concepts of the pipeline architecture, OPENGL and CUDA basics.

CO2: Analyze and identify the methods required for computer representation of objects.

CO3: Design applications using OPEN GL and CUDA libraries.

CO4: Implement geometric construction techniques for graphics applications.

Mapping of Course Outcomes with Program Outcomes

	CO	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	Understand and explore the basic concepts of the pipeline architecture, OPEN GL and CUDA basics.	M	M	M	-	-	-	-	-	M	M	-	L
CO2	Analyze and identify the methods required for computer representation of objects.	H	H	H	H	L	-	-	-	H	H	-	L
CO3	Design applications using OPEN GL and CUDA libraries.	H	H	H	H	M	-	-	-	H	-	-	L
CO4	Implement geometric construction techniques for graphics applications.	H	H	H	L	M	-	-	-	H	M	-	L

LIST OF PROGRAMS

Implement the following programs in C/C++ with OpenGL/ CUDA Libraries:

Sl. No.	Program
1.	Write a program to generate a line using Bresenham's line drawing technique. Consider slopes greater than one and slopes less than one. User can specify inputs through keyboard/mouse.
2.	Write a program to generate a circle using Bresenham's circle drawing technique. User can specify inputs through keyboard/mouse.
3.	Write a program to create a cylinder and parallelepiped by extruding a circle and quadrilateral respectively. Allow the user to specify the circle and the quadrilateral through keyboard/mouse.
4.	Write a program to recursively subdivides a tetrahedron to form 3D Sierpinski gasket. The number of recursive steps is to be specified at execution time.
5.	Write a program to create a house like figure and rotate it about a given fixed point using OpenGL/CUDA transformation functions.
6.	Write a program to create a house like figure and reflect it about an axis defined by y=mx+c using OpenGL/CUDA transformation functions.
7.	Write a program to create a square /rectangle and rotate continuously. Use two windows to demonstrate single buffering and the double buffering.
8.	Write a program to implement the Cohen-Sutherland line clipping algorithm. Make provision to specify the input for multiple lines, window for clipping and viewport for displaying the clipped image

9.	Write a program to implement the Liang-Barsky line clipping algorithm. Make provision to specify the input for multiple lines, window for clipping and viewport for displaying the clipped image.
10.	Write a program to implement the Cohen-Hodgeman polygon clipping algorithm. Make provision to specify the input polygon and window for clipping.
11.	Write a program to fill any given polygon using scan-line area filling algorithm.
12.	Write a program to model a car like figure using display lists.
13.	Write a program to create a color cube and spin it using OpenGL transformations.
14.	Write a program to generate a Limacon, Cardiod, Three-Leaf, spiral.
15	Write a program to construct Bezier curve. Control points are supplied through keyboard/ mouse

Rubrics for CG Lab

Each program is evaluated for 10 marks.

Lab Write-up and Execution rubrics (Max: 6 marks)					
Sl. No	**Criteria**	**Measuring methods**	**Excellent**	**Good**	**Poor**
1	**Understanding of problem statement. (2 Marks)** **CO1**	Observations	Student exhibits thorough understanding of program requirements and applies suitable Computer Graphics concepts for the problem. **(1.5 - 2 M)**	Student has sufficient understanding of program requirements and applies suitable Computer Graphics concepts for the problem. **(0.5 -1 M)**	Student does not have a clear understanding of program requirements and is unable to apply suitable Computer Graphics Concepts for the problem. **(0 M)**
2	**Execution and Testing (2 Marks)** **CO3, CO4**	Implementation Skills	Student demonstrates the execution of the program with all possible conditions. **(1.5 - 2 M)**	Student demonstrates the execution of the program with Few conditions. **(0.5 -1 M)**	Student has not executed the program. **(0 M)**
3	**Results and Documentation (2 Marks)** **CO4**	Observations	Documentation with appropriate comments and output is covered in data sheets and manual. **(1.5 - 2 M)**	Documentation with only few comments and only few output cases is covered in data sheets and manual. **(0.5 -1 M)**	Documentation with no comments and no output cases is covered in data sheets and manual. **(0 M)**
Viva Voce rubrics (Max: 4 marks)					
1	**Conceptual Understanding (2 Marks)** **CO1**	Viva Voce	Explains thoroughly the Computer Graphics Concepts and Algorithms **(1.5 - 2 M)**	Adequately explains the Computer Graphics Concepts and Algorithms **(0.5 -1 M)**	Unable to explain the Computer Graphics Concepts and Algorithms **(0 M)**
2	**Use of appropriate Computer Graphics APIs (1 Mark)** **CO2, CO3**	Viva Voce	Thoroughly explains the usage of appropriate API for the given problem. **(1 M)**	Sufficiently explains the usage of appropriate API for the given problem. **(0.5 M)**	Unable to explain the usage of appropriate API for the given problem. **(0 M)**
3	**Communication of Concepts (1 Mark)** **CO1, CO2**	Viva Voce	Communicates the concept used in problem solving well. **(1 M)**	Sufficiently communicates the concepts used in problem solving. **(0.5 M)**	Unable to communicate the concepts used in problem. **(0 M)**

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 simplify the 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.
The OpenGL Utility Toolkit (GLUT) is a window system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window system APIs.

Include Files:

For all OpenGL applications, User needs to include the gl.h header file in every file. Almost all OpenGL applications use GLU, the aforementioned OpenGL Utility Library, which requires inclusion of the glu.h header file. So almost every OpenGL source file begins with

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

If user is directly accessing a window interface library to support OpenGL, such as GLX, AGL, PGL, or WGL, User must include additional header files. For example, if User are calling GLX, User may need to add these lines to Userr code

```
#include <X11/Xlib.h>
#include <GL/glx.h>
```

If User is using GLUT for managing User window manager tasks, User should include

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

Note that glut.h includes gl.h, glu.h, and glx.h automatically, so including all three files is redundant. GLUT for Microsoft Windows includes the appropriate header file to access WGL.

GLUT, The OpenGL Utility Toolkit:

OpenGL contains rendering commands but is designed to be independent of any window system or operating system. Consequently, it contains no commands for opening windows or reading events from the keyboard or mouse. Unfortunately, it is 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 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 user may find it a useful starting point for learning OpenGL.

Window Management

Five routines perform tasks necessary to initialize a window.

glutInit(int **argc*, char ***argv*) initializes GLUT and processes any command line arguments (for X, this would be options like -display and -geometry). **glutInit()** should be called before any other GLUT routine.

glutInitDisplayMode(unsigned int *mode*) specifies whether to use an *RGBA* or color-index color model. User can also specify whether he wants a single- or double-buffered window. (If working in color-index mode, user want to load certain colors into the color map; use **glutSetColor()** to do this.) Finally, user can use this routine to indicate that user want the window to have an associated depth, stencil, and/or accumulation buffer. For example, if user want a window with double buffering, the RGBA color model, and a depth buffer, user might call **glutInitDisplayMode**(*GLUT_DOUBLE* | *GLUT_RGB* | *GLUT_DEPTH*).

glutInitWindowPosition(int *x*, int *y*) specifies the screen location for the upper-left corner of user window.

glutInitWindowSize(int *width*, int *size*) specifies the size, in pixels, of user window.

int **glutCreateWindow**(char **string*) creates a window with an OpenGL context. It returns a unique identifier for the new window. Until **glutMainLoop()** is called (see next section), the window is not yet displayed.

The Display Callback
glutDisplayFunc(void (* *func*)(void)) is the first and most important event callback function. Whenever GLUT determines the contents of the window need to be redisplayed, the callback function registered by **glutDisplayFunc()** is executed. Therefore, user should put all the routines user need to redraw the scene in the display callback function.
If program changes the contents of the window, sometimes user will have to call **glutPostRedisplay**(void), which gives **glutMainLoop()** a nudge to call the registered display callback at its next opportunity.

Running the Program
The very last thing–User must do is call **glutMainLoop**(void). All windows that have been created are now shown, and rendering to those windows is now effective. Event processing begins, and the registered display callback is triggered. Once this loop is entered, it is never exited.

OPENGL Data Types.

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

Handling Input Events
The following routines are used to register callback commands that are invoked when specified events occur.
glutReshapeFunc(void (* *func*)(int *w*, int *h*)) indicates what action should be taken when the window is resized.
glutKeyboardFunc(void (* *func*)(unsigned char *key*, int *x*, int *y*)) and **glutMouseFunc**(void (* *func*)(int *button*, int *state*, int *x*, int *y*)) allow user to link a keyboard key or a mouse button with a routine that is invoked when the key or mouse button is pressed or released.
glutMotionFunc(void (* *func*)(int *x*, int *y*)) registers a routine to call back when the mouse is moved while a mouse button is also pressed.

Managing a Background Process
User can specify a function that is to be executed if no other events are pending - for example, when the event loop would otherwise be idle - with **glutIdleFunc**(void (* *func*)(void)). This routine takes a pointer to the function as its only argument. Pass in NULL (zero) to disable the execution of the function.

Drawing Three-Dimensional Objects
GLUT includes several routines for drawing these three-dimensional objects:

cone	icosahedron	teapot
cube	octahedron	tetrahedron
dodecahedron	sphere	torus

User can draw these objects as wireframes or as solid shaded objects with surface normals defined.
For example, the routines for a cube and a sphere are as follows:
void **glutWireCube**(GLdouble *size*);
void **glutSolidCube**(GLdouble *size*);
void **glutWireSphere**(GLdouble *radius*, GLint *slices,* GLint *stacks*);
void **glutSolidSphere**(GLdouble *radius*, GLint *slices,* GLint *stacks*);
All these models are drawn centered at the origin of the world coordinate system.

State Management and Drawing Geometric Objects

Objectives

Clear the window to an arbitrary color.
Force any pending drawing to complete.
Draw with any geometric primitive - points, lines, and polygons - in two or three dimensions.
Turn states on and off and query state variables.
Control the display of those primitives - for example, draw dashed lines or outlined polygons.
Specify normal vectors at appropriate points on the surface of solid objects.
Use *vertex arrays* to store and access a lot of geometric data with only a few function calls.
Save and restore several state variables at once.

Although User can draw complex and interesting pictures using OpenGL, they are all constructed from a small number of primitive graphical items.

This section has the following major sub-sections:

"A Drawing Survival Kit" explains how to clear the window and force drawing to be completed. It also gives User basic information about controlling the color of geometric objects and describing a coordinate system.

"Describing Points, Lines, and Polygons" shows User what the set of primitive geometric objects is and how to draw them.

"Basic State Management" describes how to turn on and off some states (modes) and query state variables.

"Displaying Points, Lines, and Polygons" explains what control User have over the details of how primitives are drawn - for example, what diameter points have, whether lines are solid or dashed, and whether polygons are outlined or filled.

"Normal Vectors" discusses how to specify normal vectors for geometric objects and (briefly) what these vectors are for.

"Vertex Arrays" shows User how to put lots of geometric data into just a few arrays and how, with only a few function calls, to render the geometry it describes. Reducing function calls may increase the efficiency and performance of rendering.

"Attribute Groups" reveals how to query the current value of state variables and how to save and restore several related state values all at once.

"Some Hints for Building Polygonal Models of Surfaces" explores the issues and techniques involved in constructing polygonal approximations to surfaces.

One thing to keep in mind is that with OpenGL, unless User specify otherwise, every time User issue a drawing command, the specified object is drawn. This might seem obvious, but in some systems, User first make a list of things to draw. When the list is complete, the User tells the graphics hardware to draw the items in the list. The first style is called *immediate-mode* graphics and is the default OpenGL style. In addition to using immediate mode, User can choose to save some commands in a list (called a *display list*) for later drawing. Immediate-mode graphics are typically easier to program, but display lists are often more efficient.

A Drawing Survival Kit:
This section explains how to clear the window in preparation for drawing, set the color of objects that are to be drawn, and force drawing to be completed. None of these subjects has anything to do with geometric objects in a direct way, but any program that draws geometric objects has to deal with these issues.

Clearing the Window
Drawing on a computer screen is different from drawing on paper in that the paper starts out white, and all User have to do is draw the picture. On a computer, the memory holding the picture is usually filled with the last picture User drew, so User typically need to clear it to some background color before User start to draw the new scene. The color User use for the background depends on the application. For a word processor, User might clear to white (the color of the paper) before User begin to draw the text. Sometimes User might not need to clear the screen at all; for example, if the image is the inside of a room, the entire graphics window gets covered as User draw all the walls. First, a special command to clear a window can be much more efficient than a general-purpose drawing command. In addition, OpenGL allows User to set the coordinate system, viewing position, and viewing direction arbitrarily, so it might be difficult to figure out an appropriate size and location for a window-clearing rectangle. Finally, on many machines, the graphics hardware consists of multiple buffers in addition to the buffer containing colors of the pixels that are displayed. These other buffers must be cleared from time to time, and it is convenient to have a single command that can clear any combination of them. User must also know how the colors of pixels are stored in the graphics hardware known as *bit planes*.
There are two methods of storage. Either the red, green, blue, and alpha (RGBA) values of a pixel can be directly stored in the bit planes, or a single index value that references a color lookup table is stored.
RGBA color-display mode is more commonly used. As an example, these lines of code clear an RGBA mode window to black:

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
```

The first line sets the clearing color to black, and the next command clears the entire window to the current clearing color. The single parameter to **glClear()** indicates which buffers are to be cleared. In this case, the program clears only the color buffer, where the image displayed on the screen is kept.
Typically, User set the clearing color once, early in application, and then clear the buffers as often as necessary. OpenGL keeps track of the current clearing color as a state variable rather than requiring User to specify it each time a buffer is cleared.

For example, to clear both the color buffer and the depth buffer, User would use the following sequence of commands:

```
glClearColor(0.0, 0.0, 0.0, 0.0);
glClearDepth(1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

In this case, the call to **glClearColor()** is the same as before, the **glClearDepth()** command specifies the value to which every pixel of the depth buffer is to be set, and the parameter to the **glClear()** command now consists of the bitwise OR of all the buffers to be cleared. The following summary of **glClear()** includes a table that lists the buffers that can be cleared, their names.

void ***glClearColor****(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha);*
Sets the current clearing color for use in clearing color buffers in RGBA mode.
The red, green, blue, and alpha values are clamped if necessary to the range [0,1]. The default clearing color is (0, 0, 0, 0), which is black.
void ***glClear****(GLbitfield mask);*
Clears the specified buffers to their current clearing values. The mask argument is a bitwise-ORed combination of the values.

Buffer	**Name**
Color buffer	GL_COLOR_BUFFER_BIT
Depth buffer	GL_DEPTH_BUFFER_BIT
Accumulation buffer	GL_ACCUM_BUFFER_BIT
Stencil buffer	GL_STENCIL_BUFFER_BIT

Before issuing a command to clear multiple buffers, user have to set the values to which each buffer is to be cleared if user want something other than the default RGBA color, depth value, accumulation color, and stencil index. In addition to the **glClearColor()** and **glClearDepth()** commands that set the current values for clearing the color and depth buffers, **glClearIndex()**, **glClearAccum()**, and **glClearStencil()** specify the *color index*, accumulation color, and stencil index used to clear the corresponding buffers.
OpenGL allows user to specify multiple buffers because clearing is generally a slow operation, since every pixel in the window (possibly millions) is touched, and some graphics hardware allows sets of buffers to be cleared simultaneously. Hardware that does not support simultaneous clears performs them sequentially. The difference between
`glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);` and

```
glClear(GL_COLOR_BUFFER_BIT);
glClear(GL_DEPTH_BUFFER_BIT);
```

is that although both have the same final effect, the first example might run faster on many machines. It certainly wii not run more slowly.

Specifying a Color:

With OpenGL, the description of the shape of an object being drawn is independent of the description of its color. Whenever a particular geometric object is drawn, it is drawn using the currently specified coloring scheme. The coloring scheme might be as simple as "draw everything in fire-engine red," or might be as complicated as "assume the object is made out of blue plastic, that there is a yellow spotlight pointed in such and such a direction, and that there is a general low-level reddish-brown light everywhere else." In general, an OpenGL programmer first sets the color or coloring scheme and then draws the objects. Until the color or coloring scheme is changed, all objects are drawn in that color or using that coloring scheme. This method helps OpenGL achieve higher drawing performance than would result if it did not keep track of the current color.

For example, the pseudocode

```
set_current_color(red);
draw_object(A);
draw_object(B);
set_current_color(green);
set_current_color(blue);
draw_object(C);
```

draws objects A and B in red, and object C in blue. The command on the fourth line that sets the current color to green is wasted.

To draw geometric primitives that can be seen, however, user need some basic knowledge of how to set the current color; this information is provided in the next paragraphs.

To set a color, use the command **glColor3f()**. It takes three parameters, all of which are floating-point numbers between 0.0 and 1.0. The parameters are, in order, the red, green, and blue *components* of the color. User can think of these three values as specifying a "mix" of colors: 0.0 means do not use any of that component, and 1.0 means use all of that component.

Thus, the code `glColor3f(1.0, 0.0, 0.0);`
makes the brightest red the system can draw, with no green or blue components. All zeros makes black; in contrast, all ones makes white. Setting all three components to 0.5 yields gray (halfway between black and white). Here are eight commands and the colors they would set.

```
glColor3f(0.0, 0.0, 0.0); black
glColor3f(1.0, 0.0, 0.0); red
glColor3f(0.0, 1.0, 0.0); green
glColor3f(1.0, 1.0, 0.0); yellow
glColor3f(0.0, 0.0, 1.0); blue
glColor3f(1.0, 0.0, 1.0); magenta
glColor3f(0.0, 1.0, 1.0); cyan
glColor3f(1.0, 1.0, 1.0); white
```

User might have noticed earlier that the routine to set the clearing color, **glClearColor()**, takes four parameters, the first three of which match the parameters for **glColor3f()**. The fourth parameter is the alpha value. For now, set the fourth parameter of **glClearColor()** to 0.0, which is its default value.

Forcing Completion of Drawing:
As in "OpenGL Rendering Pipeline", most modern graphics systems can be thought of as an assembly line. The main central processing unit (CPU) issues a drawing command.
Perhaps other hardware does geometric transformations. Clipping is performed, followed by shading and/or texturing. Finally, the values are written into the bitplanes for display. In high-end architectures, each of these operations is performed by a different piece of hardware that is been designed to perform its particular task quickly. In such an architecture, there is no need for the CPU to wait for each drawing command to complete before issuing the next one. While the CPU is sending a vertex down the pipeline, the transformation hardware is working on transforming the last one sent, the one before that is being clipped, and so on. In such a system, if the CPU waited for each command to complete before issuing the next, there could be a huge performance penalty.
In addition, the application might be running on more than one machine. For example, suppose that the main program is running elsewhere (on a machine called the client) and that user is viewing the results of the drawing on his workstation or terminal (the server), which is connected by a network to the client. In that case, it might be horribly inefficient to send each command over the network one at a time, since considerable overhead is often associated with each network transmission. Usually, the client gathers a collection of commands into a single network packet before sending it. Unfortunately, the network code on the client typically has no way of knowing that the graphics program is finished drawing a frame or scene. In the worst case, it waits forever for enough additional drawing commands to fill a packet, and user never see the completed drawing.
For this reason, OpenGL provides the command **glFlush()**, which forces the client to send the network packet even though it might not be full. Where there is no network and all commands are truly executed immediately on the server, **glFlush()** might have no effect. However, if user is writing a program to work properly both with and without a network, include a call to **glFlush()** at the end of each frame or scene. Note that **glFlush()** does not wait for the drawing to complete - it just forces the drawing to begin execution, thereby guaranteeing that all previous commands *execute* in finite time even if no further rendering commands are executed. There are other situations where **glFlush()** is useful.
*void **glFlush**(void);*
Forces previously issued OpenGL commands to begin execution, thus guaranteeing that they complete in finite time.

A few commands - for example, commands that swap buffers in double-buffer mode – automatically flush pending commands onto the network before they can occur.
If **glFlush()** is not sufficient for user, **glFinish()** can be used. This command flushes the network as **glFlush()** does and then waits for notification from the graphics hardware or network indicating that the drawing is complete in the frame buffer. User might need to use **glFinish()** if user want to synchronize tasks – for example, to make sure that user's three-dimensional rendering is on the screen before he uses Display PostScript to draw labels on top of the rendering. Another example would be to ensure that the drawing is complete before it begins to accept user input. After user issue a **glFinish()** command, graphics process is blocked until it

receives notification from the graphics hardware that the drawing is complete. Keep in mind that excessive use of **glFinish()** can reduce the performance of application, especially if running over a network, because it requires round-trip communication. If **glFlush()** is sufficient for user needs, use it instead of **glFinish()**.

void ***glFinish****(void);*
Forces all previously issued OpenGL commands to complete. This command does not return until all effects from previous commands are fully realized.

Coordinate System Survival Kit

Whenever user initially open a window or later move or resize that window, the window system will send an event to notify him. If user is using GLUT, the notification is automated; whatever routine has been registered to **glutReshapeFunc()** will be called. User must register a callback function that will Reestablish the rectangular region that will be the new rendering canvas Define the coordinate system to which objects will be drawn. To create a simple, basic two-dimensional coordinate system into which user can draw a few objects.
Call **glutReshapeFunc**(**reshape**), where **reshape()** is the following function shown in Example below.

Example : Reshape Callback Function
```
void reshape (int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluOrtho2D (0.0, (GLdouble) w, 0.0, (GLdouble) h);
}
```
The internals of GLUT will pass this function two arguments: the width and height, in pixels, of the new, moved, or resized window. **glViewport()** adjusts the pixel rectangle for drawing to be the entire new window. The next three routines adjust the coordinate system for drawing so that the lower-left corner is (0, 0), and the upper-right corner is (*w*, *h*).
To explain it another way, think about a piece of graphing paper. The *w* and *h* values in **reshape()** represent how many columns and rows of squares are on graph paper. Then user have to put axes on the graph paper. The **gluOrtho2D()** routine puts the origin, (0, 0), all the way in the lowest, leftmost square, and makes each square represent one unit. Now when user render the points, lines, and polygons, they will appear on this paper in easily predictable squares.

Describing Points, Lines, and Polygons:

This section explains how to describe OpenGL geometric primitives. All geometric primitives are eventually described in terms of their *vertices* - coordinates that define the points themselves, the endpoints of line segments, or the corners of polygons. The next section discusses how these primitives are displayed and what control user have over their display.

Points, Lines, and Polygons

User probably have a fairly good idea of what a mathematician means by the terms *point*, line, and polygon. The OpenGL meanings are similar, but not quite the same.

One difference comes from the limitations of computer-based calculations. In any OpenGL implementation, floating-point calculations are of finite precision, and they have round-off errors. Consequently, the coordinates of OpenGL points, lines, and polygons suffer from the same problems.
Another more important difference arises from the limitations of a raster graphics display. On such a display, the smallest displayable unit is a pixel, and although pixels might be less than 1/100 of an inch wide, they are still much larger than the mathematician's concepts of infinitely small (for points) or infinitely thin (for lines). When OpenGL performs calculations, it assumes points are represented as vectors of floating-point numbers. However, a point is typically (but not always) drawn as a single pixel, and many different points with slightly different coordinates could be drawn by OpenGL on the same pixel.

Points
A point is represented by a set of floating-point numbers called a vertex. All internal calculations are done as if vertices are three-dimensional. Vertices specified by the user as two-dimensional (that is, with only x and y coordinates) are assigned a z coordinate equal to zero by OpenGL.

OpenGL works in the homogeneous coordinates of three-dimensional projective geometry, so for internal calculations, all vertices are represented with four floating-point coordinates (x, y, z, w). If w is different from zero, these coordinates correspond to the Euclidean three-dimensional point (x/w, y/w,z/w). User can specify the w coordinate in OpenGL commands, but that's rarely done. If the w coordinate is not specified, it is understood to be 1.0.

Lines
In OpenGL, the term *line* refers to a *line segment*, not the mathematician's version that extends to infinity in both directions. There are easy ways to specify a connected series of line segments, or even a closed, connected series of segments. In all cases, though, the lines constituting the connected series are specified in terms of the vertices at their endpoints, two Connected Series of Line Segments.

Polygons
Polygons are the areas enclosed by single closed loops of line segments, where the line segments are specified by the vertices at their endpoints. Polygons are typically drawn with the pixels in the interior filled in, but user can also draw them as outlines or a set of points.
In general, polygons can be complicated, so OpenGL makes some strong restrictions on what constitutes a primitive polygon. First, the edges of OpenGL polygons cannot intersect (a mathematician would call a polygon satisfying this condition a *simple polygon*). Second, OpenGL polygons must be *convex*, meaning that they cannot have indentations. Stated precisely, a region is convex if, given any two points in the interior, the line segment joining them is also in the interior. OpenGL, however, does not restrict the number of line segments making up the boundary of a convex polygon. Note that polygons with holes cannot be described. They are no convex, and they cannot be drawn with a boundary made up of a single closed loop. Be aware that if user present OpenGL with a non convex filled polygon, it might not draw it as expected. For instance, on most systems no more than the convex hull of the polygon would be filled. On some systems, less than the convex hull might be filled.

Valid and Invalid Polygons

The reason for the OpenGL restrictions on valid polygon types is that it is simpler to provide fast polygon-rendering hardware for that restricted class of polygons. Simple polygons can be rendered quickly. Many real-world surfaces consist of no simple polygons, non convex polygons, or polygons with holes. Since all such polygons can be formed from unions of simple convex polygons, some routines to build more complex objects are provided in the GLU library. These routines take complex descriptions and tessellate them, or break them down into groups of the simpler OpenGL polygons that can then be rendered.
Since OpenGL vertices are always three-dimensional, the points forming the boundary of a particular polygon do not necessarily lie on the same plane in space. (Of course, they do in many cases - if all the *z* coordinates are zero, for example, or if the polygon is a triangle.) If a polygon's vertices do not lie in the same plane, then after various rotations in space, changes in the viewpoint, and projection onto the display screen, the points might no longer form a simple convex polygon. For example, imagine a four-point *quadrilateral* where the points are slightly out of plane, and look at it almost edge-on. User can get a non simple polygon that resembles a bow tie, which is not guaranteed to be rendered correctly. User can always avoid the problem by using triangles, since any three points always lie on a plane.

Rectangles
OpenGL provides a filled-rectangle drawing primitive, **glRect*()**. User can draw a rectangle as a polygon, as described in "OpenGL Geometric Drawing Primitives," but particular implementation of OpenGL might have optimized **glRect*()** for rectangles.
void ***glRect****{sifd}(TYPEx1, TYPEy1, TYPEx2, TYPEy2);*
void ***glRect****{sifd}****v****(TYPE*v1, TYPE*v2);*
Draws the rectangle defined by the corner points (x1, y1) and (x2, y2). The rectangle lies in the plane z=0 and has sides parallel to the x- and y-axes. If the vector form of the function is used, the corners are given by two pointers to arrays, each of which contains an (x, y) pair.

Note that although the rectangle begins with a particular orientation in three-dimensional space (in the *x-y* plane and parallel to the axes), User can change this by applying rotations or other transformations.

Curves and Curved Surfaces

Any smoothly curved line or surface can be approximated - to any arbitrary degree of accuracy – by short line segments or small polygonal regions. Thus, subdividing curved lines and surfaces sufficiently and then approximating them with straight line segments or flat polygons makes them appear curved.
Even though curves are not geometric primitives, OpenGL does provide some direct support for subdividing and drawing them.

Specifying Vertices

With OpenGL, all geometric objects are ultimately described as an ordered set of vertices. User use the **glVertex*()** command to specify a vertex.

*void **glVertex**{234}{sifd}[v](TYPEcoords);*
*Specifies a vertex for use in describing a geometric object. User can supply up to four coordinates (x, y, z, w) for a particular vertex or as few as two (x, y) by selecting the appropriate version of the command. If User uses a version that does not explicitly specify z or w, z is understood to be 0 and w is understood to be 1. Calls to **glVertex*()** are only effective between a **glBegin()** and **glEnd()** pair.*

Example below provides some examples of using **glVertex*()**.
Example : Legal Uses of glVertex*()

```
glVertex2s(2, 3);
glVertex3d(0.0, 0.0, 3.1415926535898);
glVertex4f(2.3, 1.0, -2.2, 2.0);
GLdouble dvect[3] = {5.0, 9.0, 1992.0};
glVertex3dv(dvect);
```

The first example represents a vertex with three-dimensional coordinates (2, 3, 0). (Remember that if it isn't specified, the *z* coordinate is understood to be 0.) The coordinates in the second example are (0.0, 0.0, 3.1415926535898) (double-precision floating-point numbers). The third example represents the vertex with three-dimensional coordinates (1.15, 0.5, -1.1). (Remember that the *x, y*, and *z* coordinates are eventually divided by the *w* coordinate.) In the final example, *dvect* is a pointer to an array of three double-precision floating-point numbers.
On some machines, the vector form of **glVertex*()** is more efficient, since only a single parameter needs to be passed to the graphics subsystem. Special hardware might be able to send a whole series of coordinates in a single batch. If User machine is like this, it is an advantage to arrange User data so that the vertex coordinates are packed sequentially in memory. In this case, there may be some gain in performance by using the vertex array operations of OpenGL.

OpenGL Geometric Drawing Primitives

User still need to know how to tell OpenGL to create a set of points, a line, or a polygon from those vertices. To do this, User bracket each set of vertices between a call to **glBegin()** and a call to **glEnd()**. The argument passed to **glBegin()** determines what sort of geometric primitive is constructed from the vertices.

Example : Filled Polygon

```
glBegin(GL_POLYGON);
glVertex2f(0.0, 0.0);
glVertex2f(0.0, 3.0);
glVertex2f(4.0, 3.0);
glVertex2f(6.0, 1.5);
glVertex2f(4.0, 0.0);
glEnd();
```

Drawing a Polygon or a Set of Points
If User had used GL_POINTS instead of GL_POLYGON, the primitive would have been simply the five points. Table (below) function summary for **glBegin()** lists the ten possible arguments and the corresponding type of primitive.

void ***glBegin****(GLenum mode); Marks the beginning of a vertex-data list that describes a geometric primitive. The type of primitive is indicated by mode, which can be any of the values shown in*

Geometric Primitive Names and Meanings

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

void ***glEnd****(void);*
Marks the end of a vertex-data list.

As User read the following descriptions, assume that *n* vertices (v0, v1, v2, ... , vn-1) are described between a **glBegin()** and **glEnd()** pair.

GL_POINTS	Draws a point at each of the *n* vertices.
GL_LINES	Draws a series of unconnected line segments. Segments are drawn between v0 and v1, between v2 and v3, and so on. If *n* is odd, the last segment is drawn between vn-3 and vn-2, and vn-1 is ignored.
GL_LINE_STRIP	Draws a line segment from v0 to v1, then from v1 to v2, and so on, finally drawing the segment from vn-2 to vn-1. Thus, a total of *n-1* line segments are drawn. Nothing is drawn unless *n* is larger than 1. There are no restrictions on the vertices describing a line strip (or a line loop); the lines can intersect arbitrarily.
GL_LINE_LOOP	Same as GL_LINE_STRIP, except that a final line segment is drawn from vn-1 to v0, completing a loop.

GL_TRIANGLES	Draws a series of triangles (three-sided polygons) using vertices v0, v1,v2, then v3, v4, v5, and so on. If *n* is is not an exact multiple of 3, the final one or two vertices are ignored.
GL_TRIANGLE_STRIP	Draws a series of triangles (three-sided polygons) using vertices v0, v1, v2, then v2, v1, v3 (note the order), then v2, v3, v4, and so on. The ordering is to ensure that the triangles are all drawn with the same orientation so that the strip can correctly form part of a surface. Preserving the orientation is important for some operations, such as culling. *n* must be at least 3 for anything to be drawn.
GL_TRIANGLE_FAN	Same as GL_TRIANGLE_STRIP, except that the vertices are v0, v1,v2, then v0, v2, v3, then v0, v3, v4, and so on.
GL_QUADS	Draws a series of quadrilaterals (four-sided polygons) using vertices v0,v1, v2, v3, then v4, v5, v6, v7, and so on. If *n* is not a multiple of 4, the final one, two, or three vertices are ignored.
GL_QUAD_STRIP	Draws a series of quadrilaterals (four-sided polygons) beginning with v0, v1, v3, v2, then v2, v3, v5, v4, then v4, v5, v7, v6, and so on. *n* must be at least 4 before anything is drawn. If *n* is odd, the final vertex is ignored.
GL_POLYGON	Draws a polygon using the points v0, ... , vn-1 as vertices. *n* must be at least 3, or nothing is drawn. In addition, the polygon specified must not intersect itself and must be convex. If the vertices don't satisfy these conditions, the results are unpredictable.

Restrictions on Using glBegin() and glEnd()

The most important information about vertices is their coordinates, which are specified by the **glVertex*()** command. User can also supply additional vertex-specific data for each vertex - a color, a normal vector, texture coordinates, or any combination of these - using special commands. In addition, a few other commands are valid between a **glBegin()** and **glEnd()** pair.

Valid Commands between glBegin() and glEnd()

Command	**Purpose of Command**
glVertex*()	set vertex coordinates
glColor*()	set current color
glIndex*()	set current color index
glNormal*()	set normal vector coordinates
glTexCoord*()	set texture coordinates
glEdgeFlag*()	control drawing of edges
glMaterial*()	set material properties
glArrayElement()	extract vertex array data
glEvalCoord*(), glEvalPoint*()	generate coordinates
glCallList(), glCallLists()	execute display list(s)

No other OpenGL commands are valid between a **glBegin()** and **glEnd()** pair, and making most other OpenGL calls generates an error. Some vertex array commands, such as **glEnableClientState()** and **glVertexPointer()**, when called between **glBegin()** and **glEnd()**, have undefined behavior but do not necessarily generate an error. (Also, routines related to OpenGL, such as **glX*()** routines have undefined behavior between **glBegin()** and **glEnd()**.) These cases should be avoided, and debugging them may be more difficult.
Note, however, that only OpenGL commands are restricted; User can certainly include other programming-language constructs (except for calls, such as the aforementioned **glX*()** routines). For example below draws an outlined circle.
Example : Other Constructs between glBegin() and glEnd()

```
#define PI 3.1415926535898
GLint circle_points = 100;
glBegin(GL_LINE_LOOP);
for (i = 0; i < circle_points; i++) {
angle = 2*PI*i/circle_points;
glVertex2f(cos(angle), sin(angle));
}
glEnd();
```

Note: This example is not the most efficient way to draw a circle, especially if user intend to do it repeatedly. The graphics commands used are typically very fast, but this code calculates an angle and calls the **sin()** and **cos()** routines for each vertex; in addition, there is the loop overhead.
Unless they are being compiled into a display list, all **glVertex*()** commands should appear between some **glBegin()** and **glEnd()** combination. (If they appear elsewhere, they don't accomplish anything.) If they appear in a display list, they are executed only if they appear between a **glBegin()** and a **glEnd()**.

Although many commands are allowed between **glBegin()** and **glEnd()**, vertices are generated only when a **glVertex*()** command is issued. At the moment **glVertex*()** is called, OpenGL assigns the resulting vertex the current color, texture coordinates, normal vector information, and so on. To see this, look at the following code sequence. The first point is drawn in red, and the second and third ones in blue, despite the extra color commands.

```
glBegin(GL_POINTS);
glColor3f(0.0, 1.0, 0.0); /* green */
glColor3f(1.0, 0.0, 0.0); /* red */
glVertex(...);
glColor3f(1.0, 1.0, 0.0); /* yellow */
glColor3f(0.0, 0.0, 1.0); /* blue */
glVertex(...);
glVertex(...);
glEnd();
```

User can use any combination of the 24 versions of the **glVertex*()** command between **glBegin()** and **glEnd()**, although in real applications all the calls in any particular instance tend to be of the same form. If user vertex-data specification is consistent and repetitive (for example, **glColor***, **glVertex***, **glColor***, **glVertex***,...), user may enhance program's performance by using vertex arrays.

Viewing

Objectives

View a *geometric model* in any orientation by transforming it in three-dimensional space.
Control the location in three-dimensional space from which the model is viewed.
Clip undesired portions of the model out of the scene that's to be viewed.
Manipulate the appropriate matrix stacks that control model transformation for viewing and project the model onto the screen.
Combine multiple transformations to mimic sophisticated systems in motion, such as a solar system or an articulated robot arm.
Reverse or mimic the operations of the geometric processing pipeline.

Already it is explained how to instruct OpenGL to draw the geometric models user want displayed in scene. Now User must decide how to position the models in the scene, and must choose a vantage point from which to view the scene. User can use the default positioning and vantage point, but most likely user want to specify them.

User want to remember that the point of computer graphics is to create a two-dimensional image of three-dimensional objects (it has to be two-dimensional because it is drawn on a flat screen), but needs to think in three-dimensional coordinates while making many of the decisions that determine what gets drawn on the screen. A common mistake people make when creating three-dimensional graphics is to start thinking too soon that the final image appears on a flat, two-dimensional screen. Avoid thinking about which pixels need to be drawn, and instead try to visualize three-dimensional space. Create models in some three-dimensional universe that lies deep inside User computer, and let the computer do its job of calculating which pixels to color.

A series of three computer operations convert an object's three-dimensional coordinates to pixel positions on the screen. Transformations, which are represented by matrix multiplication, include modeling, viewing, and projection operations. Such operations include rotation, translation, scaling, reflecting, orthographic projection, and perspective projection. Generally, User use a combination of several transformations to draw a scene.
Since the scene is rendered on a rectangular window, objects (or parts of objects) that lie outside the window must be clipped. In three-dimensional computer graphics, clipping occurs by throwing out objects on one side of a clipping plane.

Finally, a correspondence must be established between the transformed coordinates and screen pixels. This is known as a *viewport* transformation.
All of these operations, and how to control them are discussed in the following sections:

"Overview: The Camera Analogy" gives an overview of the transformation process by describing the analogy of taking a photograph with a camera, presents a simple example program that transforms an object, and briefly describes the basic OpenGL transformation commands.
"Viewing and Modeling Transformations" explains in detail how to specify and to imagine the effect of viewing and modeling transformations. These transformations orient the model and the camera relative to each other to obtain the desired final image.

"Projection Transformations" describes how to specify the shape and orientation of the *viewing volume*. The viewing volume determines how a scene is projected onto the screen (with a perspective or orthographic projection) and which objects or parts of objects are clipped out of the scene.
"Viewport Transformation" explains how to control the conversion of three-dimensional model coordinates to screen coordinates.
"Troubleshooting Transformations" presents some tips for discovering why User might not be getting the desired effect from modeling, viewing, projection, and viewport transformations.
"Manipulating the Matrix Stacks" discusses how to save and restore certain transformations. This is particularly useful when drawing complicated objects that are built up from simpler ones.
"Additional Clipping Planes" describes how to specify additional clipping planes beyond those defined by the viewing volume.

Overview: The Camera Analogy
The transformation process to produce the desired scene for viewing is analogous to taking a photograph with a camera.

1. Set up User tripod and pointing the camera at the scene (viewing transformation).
2. Arrange the scene to be photographed into the desired composition (modeling transformation).
3. Choose a camera lens or adjust the zoom (projection transformation).
4. Determine how large User want the final photograph to be - for example, User might want it enlarged (viewport transformation).

After these steps are performed, the picture can be snapped or the scene can be drawn.

Note that these steps correspond to the order in which User specify the desired transformations in program, not necessarily the order in which the relevant mathematical operations are performed on an object's vertices. The viewing transformations must precede the modeling transformations in code, but User can specify the projection and viewport transformations at any point before drawing occurs.

To specify viewing, modeling, and projection transformations, construct a 4 × 4 matrix **M**, which is then multiplied by the coordinates of each vertex *v* in the scene to accomplish the transformation **v'=Mv** (Remember that vertices always have four coordinates (*x, y, z, w*), though in most cases *w* is 1 and for two-dimensional data *z* is 0.) Note that viewing and modeling transformations are automatically applied to surface normal vectors, in addition to vertices. (Normal vectors are used only in eye coordinates.)
This ensures that the normal vector's relationship to the vertex data is properly preserved.
The viewing and modeling transformations user specify are combined to form the model view matrix, which is applied to the incoming object coordinates to yield eye coordinates. Next, if User has specified additional clipping planes to remove certain objects from the scene or to provide cutaway views of objects, these clipping planes are applied.
After that, OpenGL applies the projection matrix to yield *clip coordinates*.
This transformation defines a viewing volume; objects outside this volume are clipped so that they are not drawn in the final scene. After this point, the perspective division is performed by dividing coordinate values by *w*, to produce *normalized device coordinates*. Finally, the

transformed coordinates are converted to window coordinates by applying the viewport transformation. User can manipulate the dimensions of the viewport to cause the final image to be enlarged, shrunk, or stretched.
User might correctly suppose that the *x* and *y* coordinates are sufficient to determine which pixels need to be drawn on the screen. However, all the transformations are performed on the *z* coordinates as well.

This way, at the end of this transformation process, the *z* values correctly reflect the depth of a given vertex (measured in distance away from the screen). One use for this depth value is to eliminate unnecessary drawing. For example, suppose two vertices have the same *x* and *y* values but different *z* values. OpenGL can use this information to determine which surfaces are obscured by other surfaces and can then avoid drawing the hidden surfaces.

A Simple Example: Drawing a Cube : Example draws a cube that is scaled by a modeling transformation. The viewing transformation, **gluLookAt()**, positions and aims the camera towards where the cube is drawn. A projection transformation and a viewport transformation are also specified.
The succeeding sections contain the complete, detailed discussion of all OpenGL's transformation commands.

The Viewing Transformation
Recall that the viewing transformation is analogous to positioning and aiming a camera. Before the viewing transformation can be specified, the current matrix is set to the identity matrix with **glLoadIdentity()**. This step is necessary since most of the transformation commands multiply the current matrix by the specified matrix and then set the result to be the current matrix. If User do not clear the current matrix by loading it with the identity matrix, user continue to combine previous transformation matrices with the new one supplied. In some cases, User do want to perform such combinations, but User also need to clear the matrix sometimes.
After the matrix is initialized, the viewing transformation is specified with **gluLookAt()**. The arguments for this command indicate where the camera (or eye position) is placed, where it is aimed, and which way is up. The arguments used here place the camera at (0, 0, 5), aim the camera lens towards (0, 0, 0), and specify the up-vector as (0, 1, 0). The up-vector defines a unique orientation for the camera.
If **gluLookAt()** was not called, the camera has a default position and orientation. By default, the camera is situated at the origin, points down the negative *z*-axis, and has an up-vector of (0, 1, 0). So in Example 3-1, the overall effect is that **gluLookAt()** moves the camera 5 units along the z-axis.

The Modeling Transformation
User use the modeling transformation to position and orient the model. For example, User can rotate, translate, or scale the model - or perform some combination of these operations. In Example, **glScalef()** is the modeling transformation that is used. The arguments for this command specify how scaling should occur along the three axes. If all the arguments are 1.0, this command has no effect. In Example, the cube is drawn twice as large in the *y* direction. Thus, if one corner of the cube had originally been at (3.0, 3.0, 3.0), that corner would wind up being

drawn at (3.0, 6.0, 3.0). The effect of this modeling transformation is to transform the cube so that it isn't a cube but a rectangular box.

Note:
Change the **gluLookAt()** call to the modeling transformation **glTranslatef()** with parameters (0.0, 0.0, -5.0). The result should look exactly the same as when User used **gluLookAt()**.

The effects of these two commands similar:
Note that instead of moving the camera (with a viewing transformation) so that the cube could be viewed, User could have moved the cube away from the camera (with a modeling transformation). This duality in the nature of viewing and modeling transformations is why user need to think about the effect of both types of transformations simultaneously. It does not make sense to try to separate the effects, but sometimes it is easier to think about them one way rather than the other. This is also why modeling and viewing transformations are combined into the *modelview matrix* before the transformations are applied.
Also note that the modeling and viewing transformations are included in the **display()** routine, along with the call that is used to draw the cube, **glutWireCube()**. This way, **display()** can be used repeatedly to draw the contents of the window if, for example, the window is moved or uncovered, and User have ensured that each time, the cube is drawn in the desired way, with the appropriate transformations. The potential repeated use of **display()** underscores the need to load the identity matrix before performing the viewing and modeling transformations, especially when other transformations might be performed between calls to **display()**.

The Projection Transformation

Specifying the projection transformation is like choosing a lens for a camera. User can think of this transformation as determining what the field of view or viewing volume is and therefore what objects are inside it and to some extent how they look. This is equivalent to choosing among wide-angle, normal, and telephoto lenses, for example. With a wide-angle lens, User can include a wider scene in the final photograph than with a telephoto lens, but a telephoto lens allows to photograph objects as though they are closer to user than they actually are.
In addition to the field-of-view considerations, the projection transformation determines how objects are *projected* onto the screen, as its name suggests. Two basic types of projections are provided by OpenGL, along with several corresponding commands for describing the relevant parameters in different ways. One type is the *perspective* projection, which matches how User see things in daily life.
Perspective makes objects that are farther away appear smaller; for example, it makes railroad tracks appear to converge in the distance. If User is trying to make realistic pictures, he will have to choose perspective projection, which is specified with the **glFrustum()** command.
The other type of projection is orthographic, which maps objects directly onto the screen without affecting their relative size. Orthographic projection is used in architectural and computer-aided design applications where the final image needs to reflect the measurements of objects rather than how they might look. Architects create perspective drawings to show how particular buildings or interior spaces look when viewed from various vantage points; the need for orthographic projection arises when blueprint plans or elevations are generated, which are used in the construction of buildings.

Before **glFrustum()** can be called to set the projection transformation, some preparation needs to happen. As in the **reshape()** routine, the command called **glMatrixMode()** is used first, with the argument GL_PROJECTION. This indicates that the current matrix specifies the projection transformation; the following transformation calls then affect the projection matrix. A few lines later **glMatrixMode()** is called again, this time with GL_MODELVIEW as the argument. This indicates that succeeding transformations now affect the modelview matrix instead of the projection matrix.
Note that **glLoadIdentity()** is used to initialize the current projection matrix so that only the specified projection transformation has an effect. Now **glFrustum()** can be called, with arguments that define the parameters of the projection transformation. In this example, both the projection transformation and the viewport transformation are contained in the **reshape()** routine, which is called when the window is first created and whenever the window is moved or reshaped. This makes sense, since both projecting (the width to height aspect ratio of the projection viewing volume) and applying the viewport relate directly to the screen, and specifically to the size or aspect ratio of the window on the screen.

Note:
Change the **glFrustum()** call in Example to the more commonly used Utility Library routine **gluPerspective()** with parameters (60.0, 1.0, 1.5, 20.0). Then experiment with different values, especially for *fovy* and *aspect*.

The Viewport Transformation

Together, the projection transformation and the viewport transformation determine how a scene gets mapped onto the computer screen. The projection transformation specifies the mechanics of how the mapping should occur, and the viewport indicates the shape of the available screen area into which the scene is mapped. Since the viewport specifies the region the image occupies on the computer screen, User can think of the viewport transformation as defining the size and location of the final processed photograph - for example, whether the photograph should be enlarged or shrunk. The arguments to **glViewport()** describe the origin of the available screen space within the window - (0,0) in this example - and the width and height of the available screen area, all measured in pixels on the screen. This is why this command needs to be called within **reshape()** - if the window changes size, the viewport needs to change accordingly. Note that the width and height are specified using the actual width and height of the window; often, User want to specify the viewport this way rather than giving an absolute size.

Drawing the Scene

Once all the necessary transformations have been specified, user can draw the scene (that is, take the photograph). As the scene is drawn, OpenGL transforms each vertex of every object in the scene by the modeling and viewing transformations. Each vertex is then transformed as specified by the projection transformation and clipped if it lies outside the viewing volume described by the projection transformation. Finally, the remaining transformed vertices are divided by *w* and mapped onto the viewport.

General-Purpose Transformation Commands

This section discusses some OpenGL commands useful to specify desired transformations. Already seen a couple of these commands, **glMatrixMode()** and **glLoadIdentity()**. The other two commands described here - **glLoadMatrix*()** and **glMultMatrix*()** - allow User to specify any transformation matrix directly and then to multiply the current matrix by that specified matrix. More specific transformation commands - such as **gluLookAt()** and **glScale*()** – are described in later sections.
As described in the preceding section, User need to state whether User want to modify the modelview or projection matrix before supplying a transformation command. User choose the matrix with **glMatrixMode()**. When nested sets of OpenGL commands that might be called repeatedly are used , remember to reset the matrix mode correctly.

*void **glMatrixMode**(GLenum mode);*
Specifies whether the modelview, projection, or texture matrix will be modified, using the argument GL_MODELVIEW, GL_PROJECTION, or GL_TEXTURE for mode. Subsequent transformation commands affect the specified matrix. Note that only one matrix can be modified at a time. By default, the modelview matrix is the one that's modifiable, and all three matrices contain the identity matrix.

User use the **glLoadIdentity()** command to clear the currently modifiable matrix for future transformation commands, since these commands modify the current matrix. Typically, User always call this command before specifying projection or viewing transformations, but User might also call it before specifying a modeling transformation.

*void **glLoadIdentity**(void);*
Sets the currently modifiable matrix to the 4 × 4 identity matrix.
To specify explicitly a particular matrix to be loaded as the current matrix, use **glLoadMatrix*()**. Similarly, use **glMultMatrix*()** to multiply the current matrix by the matrix passed in as an argument. The argument for both these commands is a vector of sixteen values (*m*1, *m*2, ... ,*m*16) that specifies a matrix **M.**

Remember that User might be able to maximize efficiency by using display lists to store frequently used matrices (and their inverses) rather than recomputing them. (OpenGL implementations often must compute the inverse of the modelview matrix so that normals and clipping planes can be correctly transformed to eye coordinates.)
Note : If user is programming in C and declare a matrix as *m*[4][4], then the element *m[i][j]* is in the *i*th column and *j*th row of the OpenGL transformation matrix. This is the reverse of the standard C convention in which *m[i][j]* is in row *i* and column *j*. To avoid confusion, declare matrices as *m*[16].

*void **glLoadMatrix**{fd}(const TYPE *m);*
Sets the sixteen values of the current matrix to those specified by m.

*void **glMultMatrix**{fd}(const TYPE *m);*
Multiplies the matrix specified by the sixteen values pointed to by m by the current matrix and stores the result as the current matrix.

Note: All matrix multiplication with OpenGL occurs as follows: Suppose the current matrix is **C** and the matrix specified with **glMultMatrix*()** or any of the transformation commands is **M**. After multiplication, the final matrix is always **CM**. Since matrix multiplication is not generally commutative, the order makes a difference.

Modeling Transformations

The three OpenGL routines for modeling transformations are **glTranslate*()**, **glRotate*()**, and **glScale*()**. These routines transform an object by moving, rotating, stretching, shrinking, or reflecting it. All three commands are equivalent to producing an appropriate translation, rotation, or scaling matrix, and then calling **glMultMatrix*()** with that matrix as the argument. However, these three routines might be faster than using **glMultMatrix*()**. OpenGL automatically computes the matrices for User.
In the command summaries that follow, each matrix multiplication is described in terms of what it does to the vertices of a geometric object using the fixed coordinate system approach, and in terms of what it does to the local coordinate system that's attached to an object.

Translate
*void **glTranslate**{fd}(TYPEx, TYPE y, TYPEz);*
Multiplies the current matrix by a matrix that moves (translates) an object by the given x, y, and z values (or moves the local coordinate system by the same amounts).

Rotate
*void **glRotate**{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);*
Multiplies the current matrix by a matrix that rotates an object (or the local coordinate system) in a counterclockwise direction about the ray from the origin through the point (x, y, z). The angle parameter specifies the angle of rotation in degrees.

Scale
*void **glScale**{fd}(TYPEx, TYPE y, TYPEz);*
Multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axes. Each x, y, and z coordinate of every point in the object is multiplied by the corresponding argument x, y, or z. With the local coordinate system approach, the local coordinate axes are stretched, shrunk, or reflected by the x, y, and z factors, and the associated object is transformed with them.

Using **glScale*()** decreases the performance of lighting calculations, because the normal vectors have to be renormalized after transformation.
Note: A scale value of zero collapses all object coordinates along that axis to zero. It's usually not a good idea to do this, because such an operation cannot be undone. Mathematically speaking, the matrix cannot be inverted, and inverse matrices are required for certain lighting operations. Sometimes collapsing coordinates does make sense, however; the calculation of shadows on a planar surface is a typical application. In general, if a coordinate system is to be collapsed, the projection matrix should be used rather than the modelview matrix.

LIST OF PROGRAMS

Implement the following programs in C/C++ with OpenGL/ CUDA Libraries:

1. Write a program to generate a line using Bresenham's line drawing technique. Consider slopes greater than one and slopes less than one. User can specify inputs through keyboard/mouse.

2. Write a program to generate a circle using Bresenham's circle drawing technique. User can specify inputs through keyboard/mouse.

3. Write a program to create a cylinder and parallelepiped by extruding a circle and quadrilateral respectively. Allow the user to specify the circle and the quadrilateral through keyboard/mouse.

4. Write a program to recursively subdivides a tetrahedron to form 3D Sierpinski gasket. The number of recursive steps is to be specified at execution time.

5. Write a program to create a house like figure and rotate it about a given fixed point using OpenGL/CUDA transformation functions.

6. Write a program to create a house like figure and reflect it about an axis defined by y=mx+c using OpenGL/CUDA transformation functions.

7. Write a program to create a square /rectangle and rotate continuously. Use two windows to demonstrate single buffering and the double buffering.

8. Write a program to implement the Cohen-Sutherland line clipping algorithm. Make provision to specify the input for multiple lines, window for clipping and viewport for displaying the clipped image.

9. Write a program to implement the Liang-Barsky line clipping algorithm. Make provision to specify the input for multiple lines, window for clipping and viewport for displaying the clipped image.
10. Write a program to implement the Cohen-Hodgeman polygon clipping algorithm. Make provision to specify the input polygon and window for clipping.
11. Write a program to fill any given polygon using scan-line area filling algorithm.
12. Write a program to model a car like figure using display lists.
13. Write a program to create a color cube and spin it using OpenGL transformations.
14. Write a program to generate a Limacon, Cardiod, Three-Leaf, spiral.
15. Write a program to construct Bezier curve. Control points are supplied through keyboard/ mouse

LAB PROGRAMS

Program 1

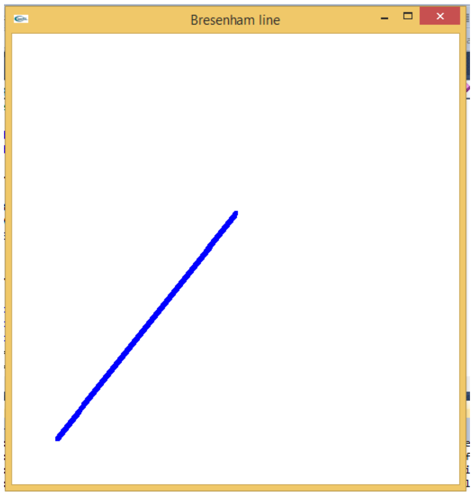
1. **Write a program to generate a line using Bresenham's line drawing technique. Consider slopes greater than one and slopes less than one. User can specify inputs through keyboard/mouse.**

```
#define BLACK 0
#include <stdio.h>
#include <GL/glut.h>

//GLint Point[4] = {0};
int x1, x2, y1, y2;
void draw_pixel(int x,int y,int value)
{
        glBegin(GL_POINTS);
        glVertex2i(x,y);
        glEnd();
}

void bres(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,BLACK);
                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,BLACK);
                }
        }
        else
        {
                draw_pixel(x,y,BLACK);
                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,BLACK);
                }
        }
}

void display()
{
        glClear(GL_COLOR_BUFFER_BIT);
        bres(x1, y1, x2, y2);
        glFlush();
}
void myinit()
{
        glClearColor(1.0,1.0,1.0,1.0);
        glColor3f(1.0,0.0,0.0);
        glPointSize(1.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0,499.0,0.0,499.0);
}

void main(int argc, char** argv)
{
//      int x1, x2, y1, y2;
```

```
        printf("Enter points: x1, y1, x2, y2");
        scanf("%d %d %d %d", &x1,&x2,&y1,&y2);
/*      Point[0] = x1;
        Point[1] = y1;
        Point[2] = x2;
        Point[3] = y2;*/
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(0,0);
        glutCreateWindow("Bresenham,s Algorithm");
        glutDisplayFunc(display);
        myinit();
        glutMainLoop();
}
```

Output and Observation:

Input two coordinates 100 100 200 200



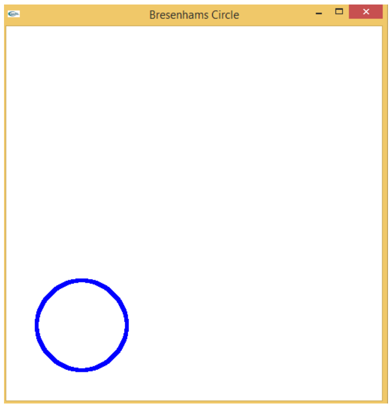
Program 2

2. **Write a program to generate a circle using Bresenham's circle drawing technique. User can specify inputs through keyboard/mouse.**

```
#include<stdio.h>
#include<gl/glut.h>
int xc,yc,r;
void drawCircle(int xc,int yc,int x,int y)
{
        glBegin(GL_POINTS);
        glVertex2i(xc+x,yc+y);
        glVertex2i(xc-x,yc+y);
        glVertex2i(xc+x,yc-y);
        glVertex2i(xc-x,yc-y);
        glVertex2i(xc+y,yc+x);
        glVertex2i(xc-y,yc+x);
        glVertex2i(xc+y,yc-x);
        glVertex2i(xc-y,yc-x);
        glEnd();
}

void circleBres(int xc,int yc,int r)
{

        int x=0,y=r;
        int d=3-2*r;
        while(x<y)
        {
                drawCircle(xc,yc,x,y);
                x++;
                if(d<0)
                        d=d+4*x+6;
                else
                {
                        y--;
                        d=d+4*(x-y)+10;
                }
                drawCircle(xc,yc,x,y);

   }

}

void display()
{
        int j;
        glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
        circleBres(xc,yc,r);
        glFlush();
}

void myinit()
{
        glClearColor(1.0,1.0,1.0,0);
        glColor3f(0,0,1.0);
        glPointSize(5.0);
        gluOrtho2D(0.0,500,0.0,500);
}

void main(int argc,char *argv[])
{
        int j;
        printf("Enter coord of centre of circle & radius: ");
        scanf("%d%d%d",&xc,&yc,&r);
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB|GLUT_DEPTH);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(100,100);
        glutCreateWindow("Bresenhams Circle");
        glutDisplayFunc(display);
        myinit();
        glutMainLoop();
}
```

Output and Observation:

Coordinates 100 100
Radius 60

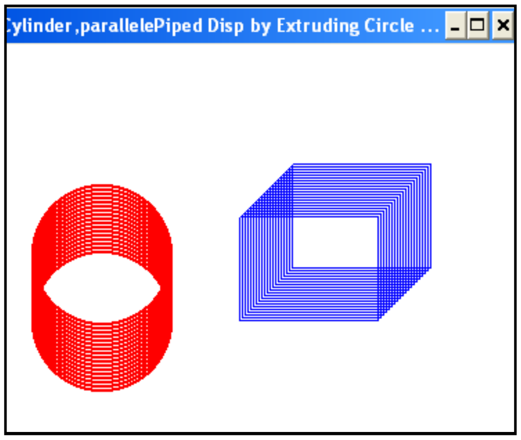


Program 3

3. **Write a program to create a cylinder and parallelepiped by extruding a circle and quadrilateral respectively. Allow the user to specify the circle and the quadrilateral through keyboard/mouse.**

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

void draw_pixel(GLint cx, GLint cy)
{
        glColor3f(1.0,0.0,0.0);
        glBegin(GL_POINTS);
                glVertex2i(cx,cy);
        glEnd();
}

void plotpixels(GLint h, GLint k, GLint x, GLint y)
{
        draw_pixel(x+h,y+k);
        draw_pixel(-x+h,y+k);
        draw_pixel(x+h,-y+k);
        draw_pixel(-x+h,-y+k);
        draw_pixel(y+h,x+k);
        draw_pixel(-y+h,x+k);
        draw_pixel(y+h,-x+k);
        draw_pixel(-y+h,-x+k);
}

void Circle_draw(GLint h, GLint k, GLint r)  // Midpoint Circle Drawing Algorithm
{
        GLint d =  1-r, x=0, y=r;
         while(y > x)
         {
                plotpixels(h,k,x,y);
                if(d < 0)
                d+=2*x+3;
                else
                 {
                        d+=2*(x-y)+5;
                        --y;
                 }
                 ++x;
         }
        plotpixels(h,k,x,y);
}

void Cylinder_draw()
{
        GLint xc=100, yc=100, r=50, i,n=50;
for(i=0;i<n;i+=3)
        Circle_draw(xc,yc+i,r);
}

void parallelepiped(int x1,int x2,int y1,int y2)
{
        glColor3f(0.0, 0.0, 1.0);
        glBegin(GL_LINE_LOOP);
                glVertex2i(x1,y1);
                glVertex2i(x2,y1);
                glVertex2i(x2,y2);
                glVertex2i(x1,y2);
        glEnd();
}

void parallelepiped_draw()
{
        int x1=200,x2=300,y1=100,y2=175, i, n=40;
        for(i=0;i<n;i+=2)
        parallelepiped(x1+i,x2+i,y1+i,y2+i);
}

void init(void)
{
        glClearColor(1.0,1.0,1.0,0.0);
        glMatrixMode(GL_PROJECTION);
        gluOrtho2D(0.0,400.0,0.0,300.0);
}

void display(void)
{
        glClear(GL_COLOR_BUFFER_BIT);
        glColor3f(1.0,0.0,0.0);
        Cylinder_draw();
        parallelepiped_draw();
        glFlush();
}

void main(int argc, char **argv)
{
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(400,300);
        glutCreateWindow("Cylinder,parallelePiped Disp by Extruding Circle &Quadrilaterl ");
        init();
        glutDisplayFunc(display);
        glutMainLoop();
}
```

Output and Observation:



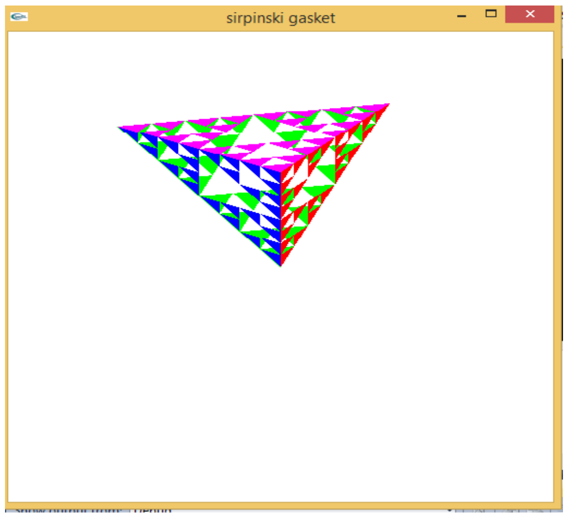
Program 4

4. **Write a program to recursively subdivides a tetrahedron to form 3D Sierpinski gasket. The number of recursive steps is to be specified at execution time.**

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

/* initial triangle */
typedef GLfloat point[3];
point v[]={{30.0, 50.0, 100.0}, {0.0, 450.0, -150.0},
        {-350.0, -400.0, -150.0}, {350., -400., -150.0}};

int n; /* number of recursive steps */

void triangle( point a, point b, point c)

/* display one triangle  */
{
   glBegin(GL_TRIANGLES);
     glVertex3fv(a);
     glVertex3fv(b);
     glVertex3fv(c);
          glEnd();
}

void divide_triangle(point a, point b, point c, int m)
{
/* triangle subdivision using vertex numbers */
   point v0, v1, v2;
   int j;
   if(m>0)
    {
      for(j=0; j<3; j++) v0[j]=(a[j]+b[j])/2;
      for(j=0; j<3; j++) v1[j]=(a[j]+c[j])/2;
      for(j=0; j<3; j++) v2[j]=(b[j]+c[j])/2;
      divide_triangle(a, v0, v1, m-1);
      divide_triangle(c, v1, v2, m-1);
      divide_triangle(b, v2, v0, m-1);
   }
   else(triangle(a,b,c));
 /* draw triangle at end of recursion */
}

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

void display()
{
          glClearColor (1.0, 1.0, 1.0,1.0);
   glClear(GL_COLOR_BUFFER_BIT);
     tetra(n);
   glFlush();
}
void myinit()
{
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  glOrtho(-499.0, 499.0, -499.0, 499.0,-499.0,499.0);
  glMatrixMode(GL_MODELVIEW);
   }

int main(int argc, char **argv)
{
  n=5;
  glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
  glutInitWindowSize(500, 500);
  glutCreateWindow("3D Gasket");
  glutDisplayFunc(display);
          myinit();
  glutMainLoop();
}
```

Output and Observation:

Division n = 5



Program 5

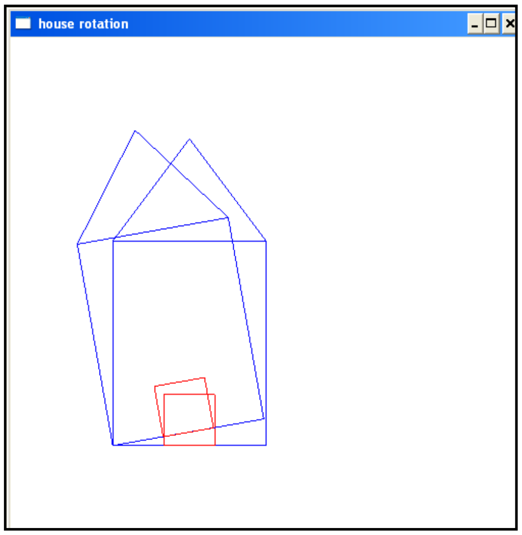
5. **Write a program to create a house like figure and rotate it about a given fixed point using OpenGL/CUDA transformation functions.**

```
#define BLACK 0
#include <stdio.h>
#include <math.h>
#include <GL/glut.h>

GLfloat house[3][9]={{100.0,100.0,175.0,250.0,250.0,150.0,150.0,200.0,200.0},
                                        {100.0,300.0,400.0,300.0,100.0,100.0,150.0,150.0,100.0},
                                        {1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0}
                                       };
GLfloat arbitrary_x=100.0;
GLfloat arbitrary_y=100.0;
GLfloat rotation_angle;


void drawhouse()
{
glColor3f(0.0, 0.0, 1.0);

glBegin(GL_LINE_LOOP);
  glVertex2f(house[0][0],house[1][0]);
  glVertex2f(house[0][1],house[1][1]);
  glVertex2f(house[0][3],house[1][3]);
  glVertex2f(house[0][4],house[1][4]);
  glEnd();
glColor3f(1.0,0.0,0.0);
  glBegin(GL_LINE_LOOP);
  glVertex2f(house[0][5],house[1][5]);
  glVertex2f(house[0][6],house[1][6]);
  glVertex2f(house[0][7],house[1][7]);
  glVertex2f(house[0][8],house[1][8]);
  glEnd();
glColor3f(0.0, 0.0, 1.0);
  glBegin(GL_LINE_LOOP);
  glVertex2f(house[0][1],house[1][1]);
  glVertex2f(house[0][2],house[1][2]);
  glVertex2f(house[0][3],house[1][3]);
  glEnd();
}
```

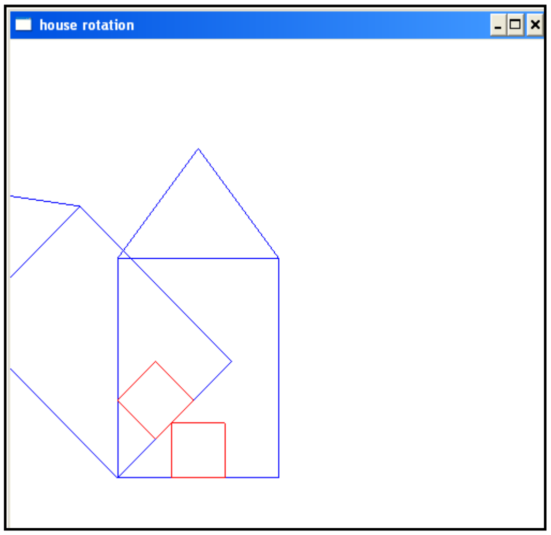
```
void display()
{

glClear(GL_COLOR_BUFFER_BIT);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
drawhouse();

glTranslatef(arbitrary_x,arbitrary_y,0.0);
glRotatef(rotation_angle,0.0,0.0,1.0);
glTranslatef(-(arbitrary_x),-(arbitrary_y),0.0);

drawhouse();

glFlush();
}

void myinit()
{
        glClearColor(1.0,1.0,1.0,1.0);
        glColor3f(1.0,0.0,0.0);
        glPointSize(1.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0,499.0,0.0,499.0);
}

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

        printf("Enter the rotation angle\n");
        scanf("%f", &rotation_angle);
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(0,0);
        glutCreateWindow("house rotation");
        glutDisplayFunc(display);
        myinit();
        glutMainLoop();
}
```

Output and Observation:

```
Enter the rotation angle
10
```



```
Enter the rotation angle
         45
```



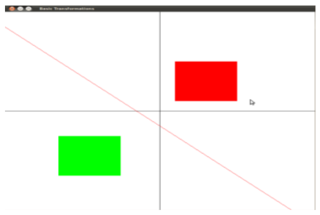
Program 6

6. **Write a program to create a house like figure and reflect it about an axis defined by y=mx+c using OpenGL/CUDA transformation functions.**

```
#define BLACK 0
#include <stdio.h>
#include <math.h>
#include <GL/glut.h>

GLfloat house[3][9]={{100.0,100.0,175.0,250.0,250.0,150.0,150.0,200.0,200.0},
                                             {100.0,300.0,400.0,300.0,100.0,100.0,150.0,150.0,100.0},
                                             {1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0,1.0}
                                            };
GLfloat M[]={0.0,1.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,1.0};
GLfloat arbitrary_x=100.0;
GLfloat arbitrary_y=100.0;
//GLfloat rotation_angle;


void drawhouse()
{
glColor3f(0.0, 0.0, 1.0);

glBegin(GL_LINE_LOOP);
  glVertex2f(house[0][0],house[1][0]);
  glVertex2f(house[0][1],house[1][1]);
  glVertex2f(house[0][3],house[1][3]);
  glVertex2f(house[0][4],house[1][4]);
  glEnd();
glColor3f(1.0,0.0,0.0);
  glBegin(GL_LINE_LOOP);
  glVertex2f(house[0][5],house[1][5]);
  glVertex2f(house[0][6],house[1][6]);
  glVertex2f(house[0][7],house[1][7]);
  glVertex2f(house[0][8],house[1][8]);
  glEnd();
glColor3f(0.0, 0.0, 1.0);
  glBegin(GL_LINE_LOOP);
  glVertex2f(house[0][1],house[1][1]);
  glVertex2f(house[0][2],house[1][2]);
  glVertex2f(house[0][3],house[1][3]);
  glEnd();
}
```

```
void display()
{

glClear(GL_COLOR_BUFFER_BIT);
glMatrixMode(GL_MODELVIEW);
drawhouse();

glLoadIdentity();
glLoadMatrixf(M);

//glTranslatef(200.0,1.0,0.0);
//glTranslatef(1.0,400.0,0.0);
//glScalef(1.0,-1.0,1.0);
//glRotatef(rotation_angle,0.0,0.0,1.0);
//glTranslatef(-(arbitrary_x),-(arbitrary_y),0.0);

drawhouse();

glFlush();
}

void myinit()
{
        glClearColor(1.0,1.0,1.0,1.0);
        glColor3f(1.0,0.0,0.0);
        glPointSize(1.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0,499.0,0.0,499.0);
}

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

        //printf("Enter the rotation angle\n");
        //scanf("%f", &rotation_angle);
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(0,0);
        glutCreateWindow("house reflection");
        glutDisplayFunc(display);
        myinit();
        glutMainLoop();
}
```

Output and Observation:



Program 7

7. **Write a program to create a square /rectangle and rotate continuously. Use two windows to demonstrate single buffering and the double buffering.**

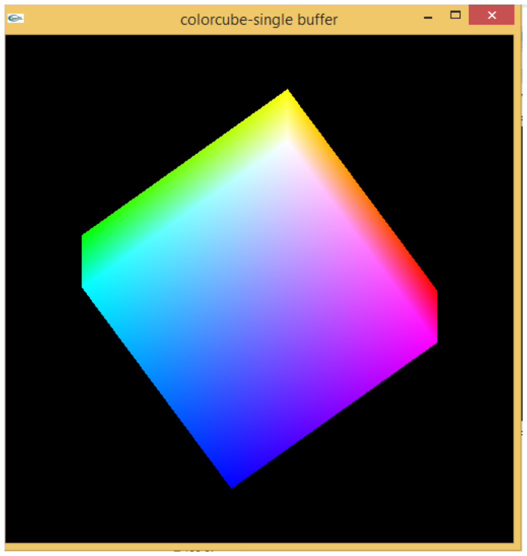
```
#include <stdlib.h>
#include <GL/glut.h>

        GLfloat vertices[] = {-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 normals[] = {-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[] = {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};

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

static GLfloat theta[] = {0.0,0.0,0.0};
static GLint axis = 2;

void display(void)
{
/* display callback, clear frame buffer and z buffer,
  rotate cube and draw, swap buffers */

 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);

 glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, cubeIndices);

 glBegin(GL_LINES);
  glVertex3f(0.0,0.0,0.0);
  glVertex3f(1.0,1.0,1.0);
 glEnd();

 glFlush();
        glutSwapBuffers();
}
```

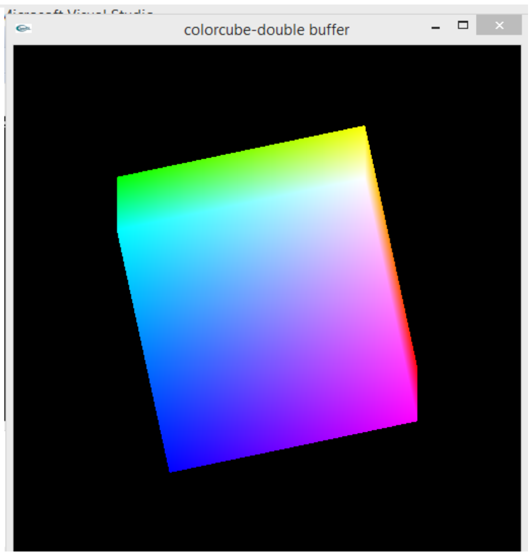
```
void spinCube()
{

/* Idle callback, spin cube 2 degrees about selected axis */

        theta[axis] += 2.0;
        if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
        glutPostRedisplay();
}

void mouse(int btn, int state, int x, int y)
{

/* mouse callback, selects an axis about which to rotate */

        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;
}

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);
}

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

/* need both double buffering and z buffer */

   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutCreateWindow("colorcube");
   glutReshapeFunc(myReshape);
   glutDisplayFunc(display);
```

```
        glutIdleFunc(spinCube);
        glutMouseFunc(mouse);
        glEnable(GL_DEPTH_TEST); /* Enable hidden--surface--removal */
                glEnableClientState(GL_COLOR_ARRAY);
                glEnableClientState(GL_NORMAL_ARRAY);
                glEnableClientState(GL_VERTEX_ARRAY);
                glVertexPointer(3, GL_FLOAT, 0, vertices);
        glColorPointer(3,GL_FLOAT, 0, colors);
        glNormalPointer(GL_FLOAT,0, normals);
                glColor3f(1.0,1.0,1.0);
        glutMainLoop();
    }
```

Output and Observation:

Single buffer **Double buffer**





Program 8

8. Write a program to implement the Cohen-Sutherland line clipping algorithm. Make provision to specify the input for multiple lines, window for clipping and viewport for displaying the clipped image

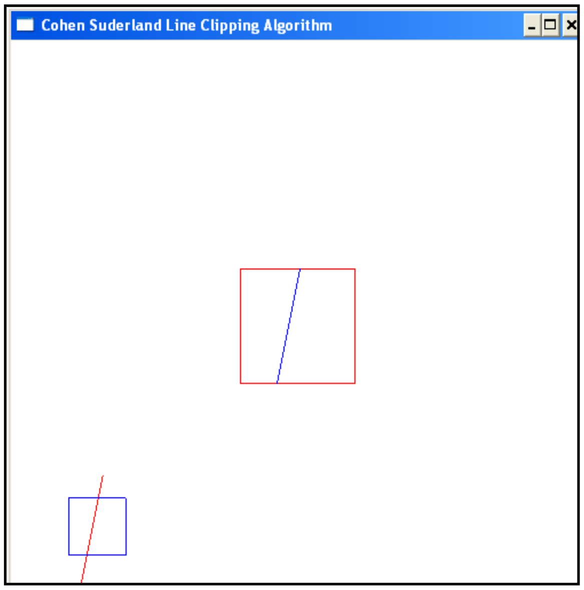
```
// Cohen-Suderland Line Clipping Algorithm with Window to viewport Mapping
 #include <stdio.h>
#include <GL/glut.h>

#define outcode int
double xmin=50,ymin=50, xmax=100,ymax=100; // Window boundaries
double xvmin=200,yvmin=200,xvmax=300,yvmax=300; // Viewport boundaries
//bit codes for the right, left, top, & bottom
const int RIGHT = 8;
const int LEFT = 2;
const int TOP = 4;
const int BOTTOM = 1;

//used to compute bit codes of a point
outcode ComputeOutCode (double x, double y);

//Cohen-Sutherland clipping algorithm clips a line from
//P0 = (x0, y0) to P1 = (x1, y1) against a rectangle with
//diagonal from (xmin, ymin) to (xmax, ymax).
void CohenSutherlandLineClipAndDraw (double x0, double y0,double x1, double y1)
{
        //Outcodes for P0, P1, and whatever point lies outside the clip rectangle
        outcode outcode0, outcode1, outcodeOut;
        bool accept = false, done = false;

        //compute outcodes
        outcode0 = ComputeOutCode (x0, y0);
        outcode1 = ComputeOutCode (x1, y1);

        do{
                if (!(outcode0 | outcode1))      //logical or is 0 Trivially accept & exit
                {
                        accept = true;
                        done = true;
                }
                else if (outcode0 & outcode1)   //logical and is not 0. Trivially reject and exit
                        done = true;
                else
                {
```

```
                //failed both tests, so calculate the line segment to clip
                //from an outside point to an intersection with clip edge
                double x, y;

                //At least one endpoint is outside the clip rectangle; pick it.
                outcodeOut = outcode0? outcode0: outcode1;

                //Now find the intersection point;
                //use formulas y = y0 + slope * (x - x0), x = x0 + (1/slope)* (y - y0)
                if (outcodeOut & TOP)          //point is above the clip rectangle
                {
                        x = x0 + (x1 - x0) * (ymax - y0)/(y1 - y0);
                        y = ymax;
                }
                else if (outcodeOut & BOTTOM)  //point is below the clip rectangle
                {
                        x = x0 + (x1 - x0) * (ymin - y0)/(y1 - y0);
                        y = ymin;
                }
                else if (outcodeOut & RIGHT)   //point is to the right of clip rectangle
                {
                        y = y0 + (y1 - y0) * (xmax - x0)/(x1 - x0);
                        x = xmax;
                }
                else                           //point is to the left of clip rectangle
                {
                        y = y0 + (y1 - y0) * (xmin - x0)/(x1 - x0);
                        x = xmin;
                }

                //Now we move outside point to intersection point to clip
                //and get ready for next pass.
                if (outcodeOut == outcode0)
                {
                        x0 = x;
                        y0 = y;
                        outcode0 = ComputeOutCode (x0, y0);
                }
                else
                {
                        x1 = x;
                        y1 = y;
                        outcode1 = ComputeOutCode (x1, y1);
                }
        }
}while (!done);
```

```
        if (accept)
        {           // Window to viewport mappings
                double sx=(xvmax-xvmin)/(xmax-xmin); // Scale parameters
                double sy=(yvmax-yvmin)/(ymax-ymin);
                double vx0=xvmin+(x0-xmin)*sx;
                double vy0=yvmin+(y0-ymin)*sy;
                double vx1=xvmin+(x1-xmin)*sx;
                double vy1=yvmin+(y1-ymin)*sy;
                        //draw a red colored viewport
                glColor3f(1.0, 0.0, 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); // draw blue colored clipped line
                glBegin(GL_LINES);
                        glVertex2d (vx0, vy0);
                        glVertex2d (vx1, vy1);
                glEnd();
        }
}

//Compute the bit code for a point (x, y) using the clip rectangle
//bounded diagonally by (xmin, ymin), and (xmax, ymax)
outcode ComputeOutCode (double x, double y)
{
        outcode code = 0;
        if (y > ymax)           //above the clip window
                code |= TOP;
        else if (y < ymin)      //below the clip window
                code |= BOTTOM;
        if (x > xmax)           //to the right of clip window
                code |= RIGHT;
        else if (x < xmin)      //to the left of clip window
                code |= LEFT;
        return code;
}

void display()
{
double x0=60,y0=20,x1=80,y1=120;
glClear(GL_COLOR_BUFFER_BIT);
//draw the line with red color
```

```
glColor3f(1.0,0.0,0.0);
//bres(120,20,340,250);
glBegin(GL_LINES);
                                glVertex2d (x0, y0);
                                glVertex2d (x1, y1);
                    glEnd();

//draw a blue colored window
glColor3f(0.0, 0.0, 1.0);

glBegin(GL_LINE_LOOP);
  glVertex2f(xmin, ymin);
  glVertex2f(xmax, ymin);
  glVertex2f(xmax, ymax);
  glVertex2f(xmin, ymax);
glEnd();
CohenSutherlandLineClipAndDraw(x0,y0,x1,y1);
glFlush();
}
void myinit()
{
          glClearColor(1.0,1.0,1.0,1.0);
          glColor3f(1.0,0.0,0.0);
          glPointSize(1.0);
          glMatrixMode(GL_PROJECTION);
          glLoadIdentity();
          gluOrtho2D(0.0,499.0,0.0,499.0);
}

void main(int argc, char** argv)
{
          //int x1, x2, y1, y2;
          //printf("Enter End points:");
          //scanf("%d%d%d%d", &x1,&x2,&y1,&y2);

          glutInit(&argc,argv);
          glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
          glutInitWindowSize(500,500);
          glutInitWindowPosition(0,0);
          glutCreateWindow("Cohen Suderland Line Clipping Algorithm");
          glutDisplayFunc(display);
          myinit();
          glutMainLoop();
}
```

Output and Observation:



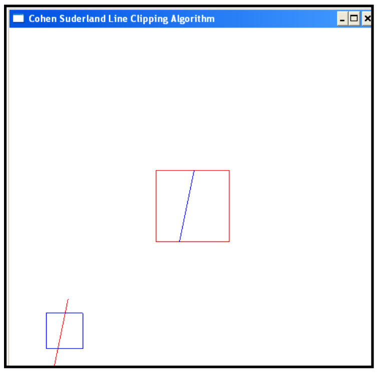
Program 9

9. **Write a program to implement the Liang-Barsky line clipping algorithm. Make provision to specify the input for multiple lines, window for clipping and viewport for displaying the clipped image.**

```
// Liang-Barsky Line Clipping Algorithm with Window to viewport Mapping */
#include <stdio.h>
#include <GL/glut.h>

double xmin=50,ymin=50, xmax=100,ymax=100; // Window boundaries
double xvmin=200,yvmin=200,xvmax=300,yvmax=300; // Viewport boundaries

int cliptest(double p, double q, double *t1, double *t2)
{ double t=q/p;
  if(p < 0.0)    // potentially enry point, update te
  {
            if( t > *t1) *t1=t;
            if( t > *t2) return(false); // line portion is outside
  }
  else
  if(p > 0.0)    //  Potentially leaving point, update tl
  {
            if( t < *t2) *t2=t;
            if( t < *t1) return(false); // line portion is outside
  }
  else
            if(p == 0.0)
            {
                      if( q < 0.0) return(false); // line parallel to edge but outside
            }
 return(true);
}

void LiangBarskyLineClipAndDraw (double x0, double y0,double x1, double y1)
{
          double dx=x1-x0, dy=y1-y0, te=0.0, tl=1.0;
          if(cliptest(-dx,x0-xmin,&te,&tl))  // inside test wrt left edge
          if(cliptest(dx,xmax-x0,&te,&tl)) // inside test wrt right edge
          if(cliptest(-dy,y0-ymin,&te,&tl)) // inside test wrt bottom edge
          if(cliptest(dy,ymax-y0,&te,&tl)) // inside test wrt top edge
          {
                    if( tl < 1.0 )
                    {
                              x1 = x0 + tl*dx;
                              y1 = y0 + tl*dy;
                    }
                    if( te > 0.0 )
                    {   x0 = x0 + te*dx;
                              y0 = y0 + te*dy;
                    }

                                // Window to viewport mappings
                    double sx=(xvmax-xvmin)/(xmax-xmin); // Scale parameters
                    double sy=(yvmax-yvmin)/(ymax-ymin);
                    double vx0=xvmin+(x0-xmin)*sx;
                    double vy0=yvmin+(y0-ymin)*sy;
                    double vx1=xvmin+(x1-xmin)*sx;
                    double vy1=yvmin+(y1-ymin)*sy;
                              //draw a red colored viewport
                    glColor3f(1.0, 0.0, 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); // draw blue colored clipped line
                    glBegin(GL_LINES);
                              glVertex2d (vx0, vy0);
                              glVertex2d (vx1, vy1);
                    glEnd();
          }
          }// end of line clipping


void display()
{
double x0=60,y0=20,x1=80,y1=120;
glClear(GL_COLOR_BUFFER_BIT);
//draw the line with red color
glColor3f(1.0,0.0,0.0);
//bres(120,20,340,250);
glBegin(GL_LINES);
                              glVertex2d (x0, y0);
                              glVertex2d (x1, y1);
                    glEnd();

//draw a blue colored window
glColor3f(0.0, 0.0, 1.0);
```

```
glBegin(GL_LINE_LOOP);
  glVertex2f(xmin, ymin);
  glVertex2f(xmax, ymin);
  glVertex2f(xmax, ymax);
  glVertex2f(xmin, ymax);
glEnd();
LiangBarskyLineClipAndDraw(x0,y0,x1,y1);
glFlush();
}
void myinit()
{
        glClearColor(1.0,1.0,1.0,1.0);
        glColor3f(1.0,0.0,0.0);
        glPointSize(1.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0,499.0,0.0,499.0);
}


void main(int argc, char** argv)
{
        //int x1, x2, y1, y2;
        //printf("Enter End points:");
        //scanf("%d%d%d%d", &x1,&x2,&y1,&y2);

        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(0,0);
        glutCreateWindow("Liang Barsky Line Clipping Algorithm");
        glutDisplayFunc(display);
        myinit();
        glutMainLoop();
}
```

Output and Observation:



Program 10

10. Write a program to implement the Cohen-Hodgeman polygon clipping algorithm. Make provision to specify the input polygon and window for clipping.

```
#include <windows.h>
#include <gl/glut.h>

struct Point{
   float x,y;
} w[4],oVer[4];
int Nout;

void drawPoly(Point p[],int n){
   glBegin(GL_POLYGON);
   for(int i=0;i<n;i++)
      glVertex2f(p[i].x,p[i].y);
   glEnd();
}

bool insideVer(Point p){
      if((p.x>=w[0].x)&&(p.x<=w[2].x))
         if((p.y>=w[0].y)&&(p.y<=w[2].y))
            return true;
      return false;
}

void addVer(Point p){
   oVer[Nout]=p;
   Nout=Nout+1;
}

Point getInterSect(Point s,Point p,int edge){
   Point in;
   float m;
   if(w[edge].x==w[(edge+1)%4].x){ //Vertical Line
      m=(p.y-s.y)/(p.x-s.x);
      in.x=w[edge].x;
      in.y=in.x*m+s.y;
   }
   else{//Horizontal Line
      m=(p.y-s.y)/(p.x-s.x);
      in.y=w[edge].y;
      in.x=(in.y-s.y)/m;
```

```
    }
    return in;
}

void clipAndDraw(Point inVer[],int Nin){
    Point s,p,interSec;
    for(int i=0;i<4;i++)
    {
        Nout=0;
        s=inVer[Nin-1];
        for(int j=0;j<Nin;j++)
        {
            p=inVer[j];
            if(insideVer(p)==true){
                if(insideVer(s)==true){
                    addVer(p);
                }
                else{
                    interSec=getInterSect(s,p,i);
                    addVer(interSec);
                    addVer(p);
                }
            }
            else{
                if(insideVer(s)==true){
                    interSec=getInterSect(s,p,i);
                    addVer(interSec);
                }
            }
            s=p;
        }
        inVer=oVer;
        Nin=Nout;
    }
    drawPoly(oVer,4);
}

void init(){
    glClearColor(0.0f,0.0f,0.0f,0.0f);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0,100.0,0.0,100.0,0.0,100.0);
    glClear(GL_COLOR_BUFFER_BIT);
    w[0].x =20,w[0].y=10;
    w[1].x =20,w[1].y=80;
    w[2].x =80,w[2].y=80;
    w[3].x =80,w[3].y=10;
}
void display(void){
    Point inVer[4];
    init();
    // As Window for Clipping
    glColor3f(1.0f,0.0f,0.0f);
    drawPoly(w,4);
    // As Rect
    glColor3f(0.0f,1.0f,0.0f);
    inVer[0].x =10,inVer[0].y=40;
    inVer[1].x =10,inVer[1].y=60;
    inVer[2].x =60,inVer[2].y=60;
    inVer[3].x =60,inVer[3].y=40;
    drawPoly(inVer,4);
    // As Rect
    glColor3f(0.0f,0.0f,1.0f);
    clipAndDraw(inVer,4);
    // Print
    glFlush();
}

int main(int argc,char *argv[]){
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    glutInitWindowSize(400,400);
    glutInitWindowPosition(100,100);
    glutCreateWindow("Polygon Clipping!");
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

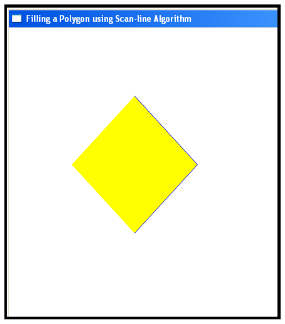
Program 11

11. Write a program to fill any given polygon using scan-line area filling algorithm.

Source Code :

```
#include <GL/glut.h>
#include<Windows.h>
float x1,x2,x3,x4,y1,y2,y3,y4;
void edgedetect(float x1,float y1,float x2,float y2,int *le,int *re)
{
        float mx,x,temp;
        int i;
        if((y2-y1)<0)
        {
                temp=y1;y1=y2;y2=temp;
                temp=x1;x1=x2;x2=temp;
        }
        if((y2-y1)!=0)
        mx=(x2-x1)/(y2-y1);
        else mx=x2-x1;
        x=x1;

        for(i=y1;i<=y2;i++)
        {
                if(x<(float)le[i])
                le[i]=(int)x;
                if(x>(float)re[i])
                re[i]=(int)x;
                x+=mx;
        }
}
void draw_pixel(int x,int y)
{
        glColor3f(1.0,0.0,0.0);
        Sleep(10);                // To set the delay time
        glBegin(GL_POINTS);
        glVertex2i(x,y);
        glEnd();
        glFlush();
}
void scanfill(float x1,float y1,float x2,float y2,float x3,float y3,float x4,float y4)
{
        int le[500],re[500],i,y;
        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(y=0;y<500;y++)
        {
                if(le[y]<=re[y])
                for(i=(int)le[y];i<(int)re[y];i++)
                draw_pixel(i,y);
        }

}
void display()
{
        x1=200.0;y1=200.0;x2=100.0;y2=300.0;x3=200.0;y3=400.0;x4=300.0;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();
        scanfill(x1,y1,x2,y2,x3,y3,x4,y4);
        glFlush();
}

void myinit()
{
        glClearColor(1.0,1.0,1.0,1.0);
        glColor3f(1.0,0.0,0.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        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);
        glutCreateWindow("Filling a Polygon using Scan-line Algorithm");
        glutDisplayFunc(display);
        myinit();
        glutMainLoop();
}
```

Output and Observation:

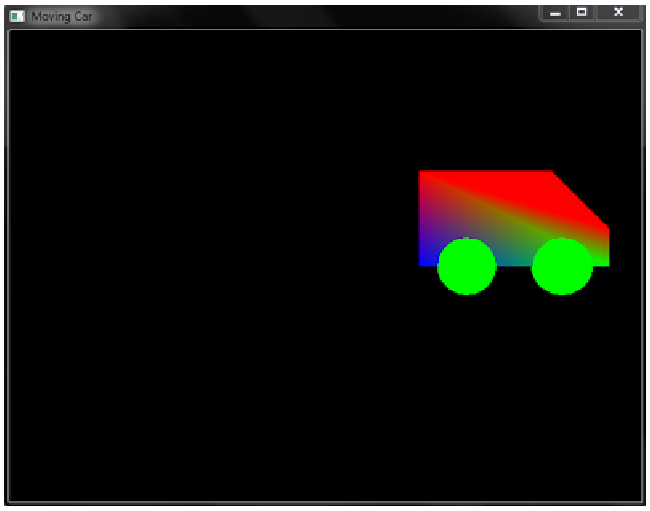


Program 12

12. Write a program to model a car like figure using display lists.

```
#include <GL/glut.h>    // Header File For The GLUT Library
#include <GL/gl.h>    // Header File For The OpenGL32 Library
#include <GL/glu.h>    // Header File For The GLu32 Library
//#include <unistd.h>     // Header File For sleeping.

/* ASCII code for the escape key. */
#define ESCAPE 27

/* The number of our GLUT window */
int window;

/* rotation angle for the triangle. */
float rtri = 0.0f;

/* rotation angle for the quadrilateral. */
float rquad = 0.0f;

/* A general OpenGL initialization function.  Sets all of the initial parameters. */
// We call this right after our OpenGL window is created.
void InitGL(int Width, int Height)
{
  // This Will Clear The Background Color To Black
  glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
  glClearDepth(1.0);               // Enables Clearing Of The Depth Buffer
  glDepthFunc(GL_LESS);               // The Type Of Depth Test To Do
  glEnable(GL_DEPTH_TEST);            // Enables Depth Testing
  glShadeModel(GL_SMOOTH);            // Enables Smooth Color Shading

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();               // Reset The Projection Matrix

  gluPerspective(45.0f,(GLfloat)Width/(GLfloat)Height,0.1f,100.0f);

  glMatrixMode(GL_MODELVIEW);
}

/* The function called when our window is resized (which shouldn't happen, because we're fullscreen) */
void ReSizeGLScene(int Width, int Height)
{
  if (Height==0)               // Prevent A Divide By Zero If The Window Is Too Small
    Height=1;

  glViewport(0, 0, Width, Height);        // Reset The Current Viewport And Perspective Transformation

  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(45.0f,(GLfloat)Width/(GLfloat)Height,0.1f,100.0f);
  glMatrixMode(GL_MODELVIEW);
}

float ballX = -0.5f;
float ballY = 0.0f;
float ballZ = 0.0f;

void drawBall(void) {
        glColor3f(0.0, 1.0, 0.0); //set ball colour
        glTranslatef(ballX,ballY,ballZ); //moving it toward the screen a bit on creation
        //glRotatef(ballX,ballX,ballY,ballZ);
        glutSolidSphere (0.3, 20, 20); //create ball.
        glTranslatef(ballX+1.5,ballY,ballZ); //moving it toward the screen a bit on creation
        glutSolidSphere (0.3, 20, 20); //
        }


/* The main drawing function. */
void DrawGLScene()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);        // Clear The Screen And The
Depth Buffer
  glLoadIdentity();                // Reset The View

  glTranslatef(rtri,0.0f,-6.0f);        // Move Left 1.5 Units And Into The Screen 6.0

  //glRotatef(rtri,1.0f,0.0f,0.0f);        // Rotate The Triangle On The Y axis
  // draw a triangle (in smooth coloring mode)
  glBegin(GL_POLYGON);                // start drawing a polygon
  glColor3f(1.0f,0.0f,0.0f);            // Set The Color To Red
  glVertex3f(-1.0f, 1.0f, 0.0f);        // Top left
  glVertex3f(0.4f, 1.0f, 0.0f);

  glVertex3f(1.0f, 0.4f, 0.0f);

  glColor3f(0.0f,1.0f,0.0f);            // Set The Color To Green
  glVertex3f( 1.0f,0.0f, 0.0f);         // Bottom Right
  glColor3f(0.0f,0.0f,1.0f);            // Set The Color To Blue
  glVertex3f(-1.0f,0.0f, 0.0f);// Bottom Left

  //glVertex3f();
  glEnd();                  // we're done with the polygon (smooth color interpolation)
  drawBall();

  rtri+=0.005f;                   // Increase The Rotation Variable For The Triangle
  if(rtri>2)
     rtri=-2.0f;
  rquad-=15.0f;                   // Decrease The Rotation Variable For The Quad
  // swap the buffers to display, since double buffering is used.
  glutSwapBuffers();
}

/* The function called whenever a key is pressed. */
void keyPressed(unsigned char key, int x, int y)
{
    /* sleep to avoid thrashing this procedure */
   // usleep(100);

    /* If escape is pressed, kill everything. */
    if (key == ESCAPE)
    {
    /* shut down our window */
    glutDestroyWindow(window);

    /* exit the program...normal termination. */
    exit(0);
    }
}

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

  glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_ALPHA | GLUT_DEPTH);

  /* get a 640 x 480 window */
  glutInitWindowSize(640, 480);

  /* the window starts at the upper left corner of the screen */
  glutInitWindowPosition(0, 0);

  /* Open a window */
  window = glutCreateWindow("Moving Car");

  /* Register the function to do all our OpenGL drawing. */
  glutDisplayFunc(&DrawGLScene);

  /* Go fullscreen.  This is as soon as possible. */
  //glutFullScreen();

  /* Even if there are no events, redraw our gl scene. */
  glutIdleFunc(&DrawGLScene);

  /* Register the function called when our window is resized. */
  glutReshapeFunc(&ReSizeGLScene);

  /* Register the function called when the keyboard is pressed. */
  glutKeyboardFunc(&keyPressed);
  /* Initialize our window. */
  InitGL(640, 480);

  /* Start Event Processing Engine */
  glutMainLoop();

  return 1;
}
```

Output and Observation:



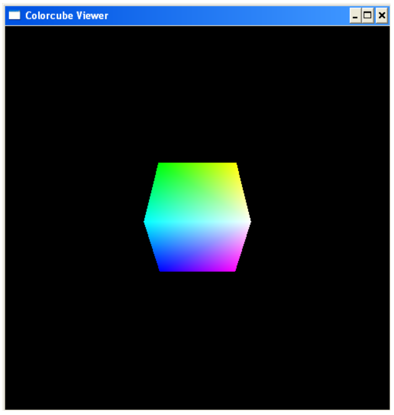
Program 13

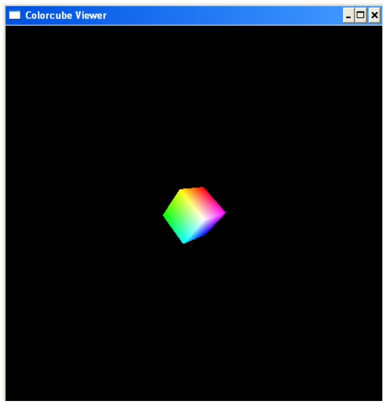
13. Write a program to create a color cube and spin it using OpenGL transformations.

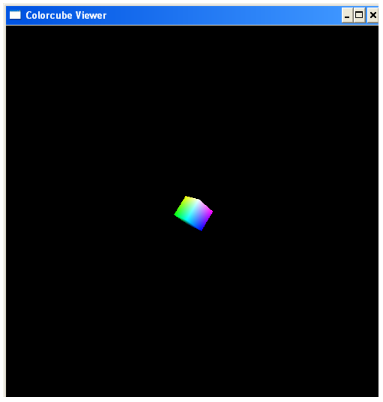
```
#include <GL/glut.h>

GLfloat vertices[8][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[8][3] =   {         {0.0,0.0,1.0}, {1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0},
                                   {0.0,0.0,0.0},{1.0,0.0,0.0}, {1.0,1.0,0.0}, {0.0,1.0,0.0}
                         };

GLfloat theta[] = {0.0,0.0,0.0};
GLint axis = 2;
GLdouble viewer[]= {0.0, 0.0, 5.0}; /* initial viewer location */

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

void colorcube()
{
        polygon(0,3,2,1); // front face – counter clockwise
        polygon(4,5,6,7); // back face – clockwise

        polygon(2,3,7,6); // front face – counter clockwise
        polygon(1,5,4,0); // back face – clockwise

        polygon(1,2,6,5); // front face – counter clockwise
        polygon(0,4,7,3); // back face – clockwise

}


void display(void)
{
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        // Update viewer position in modelview matrix
        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();
        glFlush();
        glutSwapBuffers();
}
void mouse(int btn, int state, int x, int y)
{
        if(btn==GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        axis = 0;
        if(btn==GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
        axis = 1;
        if(btn==GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
        axis = 2;
        theta[axis]+= 2.0;
        if( theta[axis] > 360.0 ) theta[axis]-= 360.0;
        display();
}

void keys(unsigned char key, int x, int y)
{
        if(key == 'x') viewer[0]-= 1.0;
        if(key == 'X') viewer[0]+= 1.0;
        if(key == 'y') viewer[1]-= 1.0;
        if(key == 'Y') viewer[1]+= 1.0;
        if(key == 'z') viewer[2]-= 1.0;
        if(key == 'Z') viewer[2]+= 1.0;
        display();
}

void myReshape(int w, int h)
{
        glViewport(0, 0, w, h);
        /* Use a perspective view */
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        if(w<=h)
        glFrustum(-2.0, 2.0, -2.0 *(GLfloat) h/(GLfloat) w,2.0*(GLfloat) h/(GLfloat) w, 2.0, 20.0);
        else
        glFrustum(-2.0, 2.0, -2.0 *(GLfloat)w/(GLfloat) h,2.0* (GLfloat) w / (GLfloat) h, 2.0, 20.0);
        glMatrixMode(GL_MODELVIEW);
}

void  main(int argc, char **argv)
{
        glutInit(&argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
        glutInitWindowSize(500, 500);
        glutCreateWindow("Colorcube Viewer");
        glutReshapeFunc(myReshape);
        glutDisplayFunc(display);
        glutMouseFunc(mouse);
        glutKeyboardFunc(keys);
        glEnable(GL_DEPTH_TEST);
        glutMainLoop();
}
```

Output and Observation:

Rotation through X axis **Perspective view with Key ‘x’** **Perspective view with Key ‘y’**







Program 14

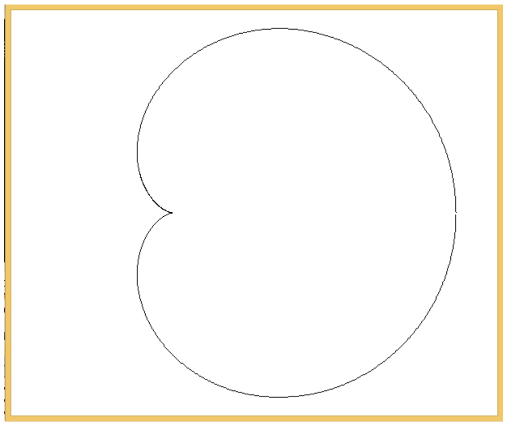
14. Write a program to generate a Limacon, Cardiod, Three leaf curve, Spiral.

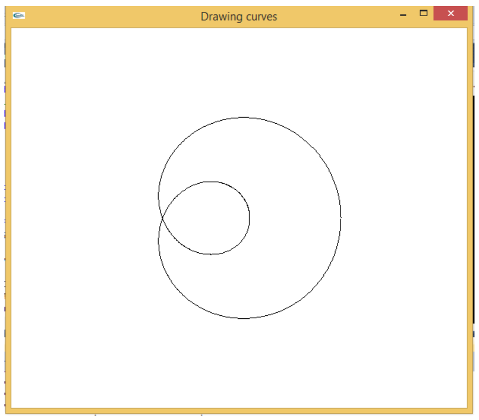
```
// rvce_lcts.cpp : Defines the entry point for the console application.
//

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


struct screenPt
{
        int x;
        int y;
};
typedef enum {limacon=1,cardioid=2,threeLeaf=3,spiral=4} curveName;
int w=600,h=500;

void myinit(void)
{
        glClearColor(1.0,1.0,1.0,1.0);
        glMatrixMode(GL_PROJECTION);
        gluOrtho2D(0.0,200.0,0.0,150.0);
}
void lineSegment(screenPt p1, screenPt p2)
{
        glBegin(GL_LINES);
                glVertex2i(p1.x,p1.y);
                glVertex2i(p2.x,p2.y);
        glEnd();
}
void drawCurve(int curveNum)
{
        const double twoPi=6.283185;
        const int a=175,b=60;
        float r,theta,dtheta=1.0/float (a);
        int x0=200,y0=250;
        screenPt curvePt[2];

        glColor3f(0.0,0.0,0.0);
        curvePt[0].x=x0;
        curvePt[0].y=y0;

        switch(curveNum)
        {
                case limacon:  curvePt[0].x+=a+b; break;
                case cardioid:  curvePt[0].x+=a+a; break;
                case threeLeaf:         curvePt[0].x+=a;   break;
                case spiral:   break;
                default: break;
        }
        theta=dtheta;
        while(theta<twoPi)
        {
                switch(curveNum)
                {
                        case limacon:  r=a*cos(theta)+b; break;
                        case cardioid:  r=a*(1+cos(theta)); break;
                        case threeLeaf:         r=a*cos(3*theta);   break;
                        case spiral:   r=(a/4.0)*theta;  break;
                        default:            break;
                }
        curvePt[1].x=x0+r*cos(theta);
        curvePt[1].y=y0+r*sin(theta);
        lineSegment(curvePt[0],curvePt[1]);

        curvePt[0].x=curvePt[1].x;
        curvePt[0].y=curvePt[1].y;
        theta+=dtheta;
        }
}


void mydisplay()
{
        int curveNum;

        glClear(GL_COLOR_BUFFER_BIT);
        printf( "\n Enter the integer value corresponding to \n");
        printf( "one of the following curve names. \n");
        printf( "Press any other key to exit. \n");
        printf("\n 1-limacon,2-cardioid,3-threeleaf,4-spiral:");
        scanf("%d", &curveNum);

                if (curveNum==1 || curveNum==2 || curveNum==3 || curveNum==4)
                        drawCurve(curveNum);
        glFlush();
}
void myreshape(int nw,int nh)
{
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0,(double)nw,0.0,(double)nh);
        glClear(GL_COLOR_BUFFER_BIT);
}
void main(int argc, char** argv)
{
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
        glutInitWindowSize(w,h);
        glutInitWindowPosition(100,100);
        glutCreateWindow("Drawing curves");
        myinit();
        glutDisplayFunc(mydisplay);
        glutReshapeFunc(myreshape);

        glutMainLoop();
}
```

Output and Observation:

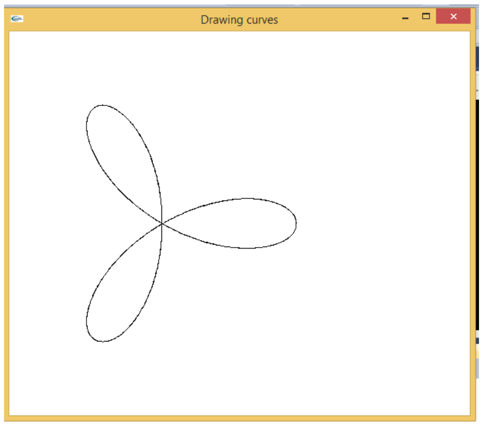
Limacon

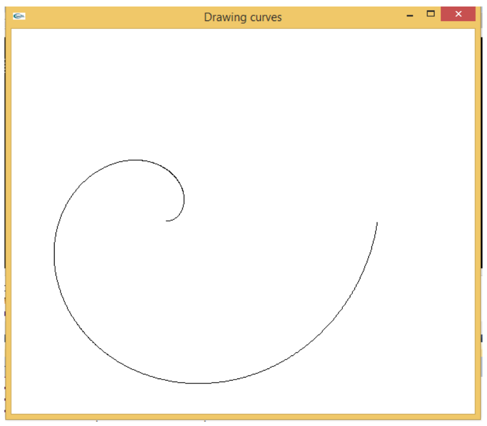
Cardioid





Three leaf **Spiral**





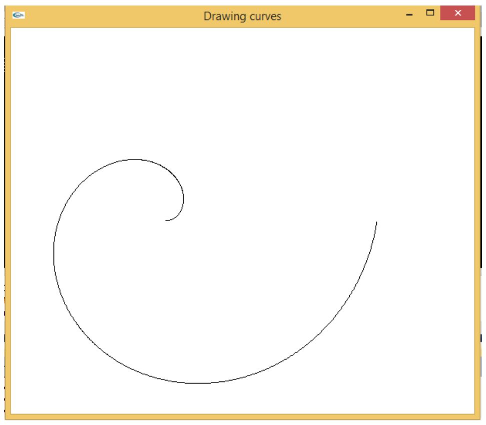
Program 15

15. **Write a program to construct Bezier curve. Control points are supplied through keyboard/ mouse.**

```
#include<iostream>
#include<math.h>
using namespace std;
#include<gl/glut.h>

float f,g,r,x[4],y[4];

void myinit()
{
	glClearColor(1.0,1.0,1.0,1.0);
	glColor3f(1.0,0.0,0.0);
	glPointSize(5.0);
	gluOrtho2D(0.0,800,0.0,800);
}

void draw_pixel(float x,float y)
{

	glBegin(GL_POINTS);
	glVertex2f(x,y);
	glEnd();

}

/*void drawcircle(float xc,float yc)
{
	draw_pixel(xc,yc);
	draw_pixel(yc,xc);
	draw_pixel(-yc,xc);
	draw_pixel(-xc,yc);
	draw_pixel(xc,-yc);
	draw_pixel(yc,-xc);
	draw_pixel(-xc,-yc);
	draw_pixel(-yc,-xc);
}*/

/*void draw_circle(float x1,float x2,float rad)
{
	glClear(GL_COLOR_BUFFER_BIT);

	float x=rad,y=0,theta,k;
	for(theta=0.0;theta<=45;theta=theta+0.25)
```

```
        {
                x=rad*cos(theta);
                y=rad*sin(theta);
                drawcircle(x+x1,y+x2);
        }
        for(k=1;k<rad;k++)
                draw_circle(x1,x2,k);
        glFlush();
}*/

void display()
{
        glClear(GL_COLOR_BUFFER_BIT);
        int i;
        double t;
        glColor3f(0.0,0.0,0.0);
        glBegin(GL_POINTS);
        for (t = 0.0; t < 1.0; t += 0.0005)
        {
                double xt = pow(1-t, 3)*x[0]+3*t*pow(1-t,2)*x[1]+3*pow(t,2)*(1-t)*x[2]+pow(t,
3)*x[3];
                double yt = pow(1-t,3)*y[0]+3*t*pow(1-t,2)*y[1]+3*pow(t,2)*(1-
t)*y[2]+pow(t,3)*y[3];
                glVertex2f(xt,yt);
        }
        glColor3f(1.0,1.0,0.0);
        for (i=0; i<4; i++)
                glVertex2f(x[i], y[i]);
        glEnd();
        glFlush();
}
int main(int argc,char **argv)
{
        cout<<"Enter x coordinates\n";
        cin>>x[0]>>x[1]>>x[2]>>x[3];
        cout<<"Enter y coordinates\n";
        cin>>y[0]>>y[1]>>y[2]>>y[3];
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
        glutInitWindowSize(500,500);
        glutInitWindowPosition(100,100);
        glutCreateWindow("new window");
        glutDisplayFunc(display);
        myinit();
        glutMainLoop();
}
```

Output and Observation:


Drawing curves