JSS Mahavidyapeetha

JSS Academy of Technical Education

Kengeri - Uttarahalli Main Road, Bengaluru - 560060

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

A MANUAL FOR

VI SEMESTER

COMPUTER GRAPHICS LABORATORY WITH MINI PROJECT (18CSL67)

Prepared by

Vikhyath K B, Assistant Professor, Department of CSE

Pavitra G S , Assistant Professor, Department of CSE

Impana K P , Assistant Professor, Department of CSE

Approved by

Dr. Naveen N C

Professor and Head, Dept. of CSE

JSS Mahavidyapeetha

JSS Academy of Technical Education, Bengaluru

Department of Computer Science & Engineering

Vision:

To be a distinguished academic and research Department in the field of Computer Science and Engineering for enabling students to be highly competent professionals to meet global challenges.

Mission:

1. Impart quality education in Computer Science and Engineering through state-of-the art learning environment and committed faculty with research expertise.

2. Train students to become the most sought after professionals in the field of Information Technology by providing them strong theoretical foundation with adequate practical training.

3. Provide a conducive environment for faculty and students to carry out research and innovation in collaboration with reputed research institutes and industry.

4. Inculcate human values and professional ethics among students to enable them to become good citizens and serve the society.

Program Educational Objectives

1. Graduates shall possess essential skills to adapt to emerging technologies & environment to solve world problem.

2. Graduates shall have required technical competency for pursuing higher studies & Research.

3. Graduates shall have essential communication and managerial skills to become competent professionals and entrepreneurs.

Program Specific Outcomes

1. Apply the principles of basic engineering science and acquire the hardware and software aspects of computer science.

2. Solve the real world problems using modelling for a specific computer system and architecture.

3. Ability to design and develop applications using various software and hardware tools.

4. Exhibit the practical competence using broad range of programming languages.

Program Outcomes (POs)

Engineering Graduates will be able to:

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

3. **Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.

4. **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.

5. **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.

6. **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.

7. **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.

8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **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.

12. **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.

COMPUTER GRAPHICS LABORATORY WITH MINI PROJECT / 18CSL67

Exam Hours: 03

SEE Marks: 60

CIE Marks: 40

Semester: 6th Semester

Total Number of Lab Contact Hours: 36

Number of Contact Hours / Week: 0:2:2

Course Objectives

This Course will enable students to

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

Syllabus

PART A

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

1. Implement Brenham's line drawing algorithm for all types of slope.
Refer:Text-1: Chapter 3.5
Refer:Text-2: Chapter 8

2. Create and rotate a triangle about the origin and a fixed point.
Refer:Text-1: Chapter 5-4

3. Draw a colour cube and spin it using OpenGL transformation matrices.
Refer:Text-2: Modelling a Coloured Cube

4. Draw a color cube and allow the user to move the camera suitably to experiment with perspective viewing.
Refer:Text-2: Topic: Positioning of Camera

5. Clip a lines using Cohen-Sutherland algorithm
Refer:Text-1: Chapter 6.7
Refer:Text-2: Chapter 8

6. To draw a simple shaded scene consisting of a tea pot on a table. Define suitably the position and properties of the light source along with the properties of the surfaces of the solid object used in the scene.
Refer:Text-2: Topic: Lighting and Shading

7. Design, develop and implement recursively subdivide a tetrahedron to form 3D sierpinski gasket. The number of recursive steps is to be specified by the user.
Refer: Text-2: Topic: sierpinski gasket.

8. Develop a menu driven program to animate a flag using Bezier Curve algorithm
Refer: Text-1: Chapter 8-10

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

PART – B (MINI PROJECT)

Student should develop mini project on the topics mentioned below or similar applications using Open GL API. Consider all types of attributes like color, thickness, styles, font, background, speed etc., while doing mini project.
(During the practical exam: the students should demonstrate and answer Viva-Voce)
Sample Topics: Simulation of concepts of OS, Data structures, algorithms etc.

Laboratory Outcomes: The student should be able to:
1. Apply the concepts of computer graphics
2. Implement computer graphics application using OpenGL
3. Animate real world problems using OpenGL

Conduction of Practical Examination:

- Experiment distribution
    - For laboratories having only one part: students are allowed to pick one experiment from the lot with equal opportunity.
    - For laboratories having PART A and PART B: Students are allowed to pick one experiment from PART A and one experiment from PART B, with equal opportunity.

- Change of experiment is allowed only once and marks allotted for procedure to be made zero of the change pert only.
- Marks distribution(Courseed to change in accordance eith university regulations)
- For laboratories having only one part – Procedure + Execution + Viva-Voce: 15+70+15=100 Marks.
- For laboratories having PART A and PART B

i. Part A – Procedure + Execution + Viva = 6+28+6=40 Marks
ii. Part B – Procedure + Execution + Viva = 9 + 42 +9 = 60 Marks

Reference books:

1. Donald Hearn & Pauline Baker: Computer Graphics-OpenGL Version,3 rd Edition, Pearson Education,2011

2. Edward Angel: Interactive computer graphics- A Top Down approach with OpenGL, 5th edition. Pearson Education, 2011

3. M M Raikar, Computer Graphics using OpenGL, Fillip Learning / Elsevier, Bangalore / New Delhi (2013)

Course Outcomes

C316.1	Apply various computer graphics algorithms to build a 2D and 3D models.	L3
C316.2	Apply geometric transformations techniques to 2D / 3D models.	L3
C316.3	Apply viewing, lighting and shading techniques to 2D / 3D scenes.	L3
C316.4	Build computer graphics applications for real world problems using OpenGL by incorporating team spirit and professional attitude.	L3

CO – PO Mapping:

		POs											
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**
CG Laboratory	**C316.1**	3	2	2	1	1	-	-	-	-	-	-	-
	C316.2	3	2	2	1	1	-	-	-	-	-	-	-
	C316.3	3	2	2	1	1	-	-	-	-	-	-	-
	C316.4	3	2	2	1	1	-	-	1	1	1	1	-
	All COs	**3**	**2**	**2**	**1**	**1**	**-**	**-**	**1**	**1**	**1**	**1**	**-**

CO – PSO Mapping:

		PSOs			
		1	**2**	**3**	**4**
CG Laboratory	**C316.1**	2	1	2	1
	C316.2	2	1	2	1
	C316.3	2	1	2	1
	C316.4	**2**	**2**	**2**	**2**
	All COs	**2**	**1.25**	**2**	**1.25**

Justification for CO-PO matrix

Course Outcomes	Program Outcomes	Co-relation level	Justification
C316.1	PO1	3	High knowledge of mathematics, engineering fundamentals and engineering specialization required to apply computer graphics algorithms.
	PO2	2	Moderate knowledge on principles of mathematics and engineering sciences are required to analyze computer graphics algorithms.
	PO3	2	Apply computer graphics algorithms to provide the solutions to the complex engineering problems.
	PO4	1	Fair research based knowledge and methods are required to form a conclusion on algorithms.
	PO5	1	Implement computer graphics algorithms using suitable software packages.
C316.2	PO1	3	High knowledge of mathematics, engineering fundamentals and engineering specialization required to apply geometric transformations techniques.
	PO2	2	Moderate knowledge on principles of mathematics and engineering sciences are required to apply geometric transformations.
	PO3	2	Apply geometric transformations techniques to provide solutions to complex engineering problems.
	PO4	1	Fair research based knowledge and methods are required to form a conclusion on geometric transformations problems.
	PO5	1	Identifying suitable packages for implementing geometric transformations on 2D and 3D models.
C316.3	PO1	3	High knowledge of mathematics, engineering fundamentals are required to apply viewing, lighting and shading techniques on two and three dimensional scenes.
	PO2	2	Moderate knowledge on principles of mathematics and engineering sciences are required to apply algorithms on lighting and shading in computer graphics.
	PO3	2	Fair knowledge on suitable computer graphics algorithms required to design of complex engineering problems in viewing and lighting.
	PO4	1	Research based knowledge and methods are required to analyze and understand viewing and lighting problems.
	PO5	1	Implement viewing, lighting and shading techniques on suitable packages.
C316.4	PO1	3	High knowledge of mathematics, engineering fundamentals and engineering specialization required to build computer graphics applications.
	PO2	2	Moderate knowledge on principles of mathematics and engineering sciences are required to apply to build computer graphics applications.

	PO3	2	Design computer graphics applications to solve complex engineering problems.
	PO4	1	Fair research based knowledge required to analyse the computer graphics applications.
	PO5	1	Develop computer graphics applications using suitable tool.
	PO8	1	Each team should apply ethical principles and responsibilities to carry out project work.
	PO9	1	Build a computer graphics application as an individual or in a team using programming languages.
	PO10	1	Computer graphics application should be well documented and presented.
	PO11	1	Building a computer graphics application requires proper understanding of engineering and project management principles.

Justification for CO-PSO matrix

Course Outcomes	Program Specific Outcome	Co-relation level	Justification
C316.1	**PSO1**	2	Principles of engineering science are required to apply various algorithms in computer graphics.
	PSO2	1	Apply suitable computer graphics algorithms to solve real world problems.
	PSO3	2	Design various computer graphics algorithms using suitable software packages.
	PSO4	1	Implement computer graphics algorithms using broad range of programming languages.
C316.2	**PSO1**	2	Hardware and software aspects of computer are required to apply geometric transformations on 2D and 3D models.
	PSO2	1	Apply geometric transformations to solve real world problems.
	PSO3	2	Design geometric transformations techniques on suitable software package.
	PSO4	1	Implement geometric transformation techniques using broad range of programming languages.
C316.3	**PSO1**	2	Hardware and software aspects of computer are required to apply viewing, lighting and shading techniques.
	PSO2	1	Apply lighting and shading techniques to solve real world problems.
	PSO3	2	Design and develop applications for lighting and shading techniques using software packages.
	PSO4	1	Implement viewing, lighting and shading using broad range of programming languages.
C316.4	**PSO1**	2	Hardware and software aspects of computer are required to build graphics applications.
	PSO2	2	Build graphical applications to solve real world problems.
	PSO3	2	Different tools are used to build graphical applications.
	PSO4	2	Different programming language syntax are used to build graphical applications.

Table of Contents

INTRODUCTION

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

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

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

FEATURES OF OpenGL :

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

OpenGL is a software interface to graphics hardware. The API mainly tries to focus on using the GPU to achieve hardware-accelerated rendering. The OpenGL is the core library which provides a powerful but primitive set of rendering commands. All OpenGL library functions begin with the letters **gl** and capitalized words (example functions: glClear, glClearColor etc.)

OpenGL doesn't provide high-level commands for describing models of three-dimensional objects. Commands for performing windowing tasks or obtaining user input are included in OpenGL. However, sophisticated libraries that provides these features are built on top of OpenGL. Two such libraries are namely, GLUT **(**OpenGL Utility Toolkit) and GLU **(**OpenGL Utility Library)**.**

- GLUT is an auxiliary library that handles window management(window definition, window creation, window control)OS system calls (mouse buttons, movement, keyboard, etc), and callbacks. GLUT routines begin with the prefix glut.

- GLU is an auxiliary library that is included with OpenGL and built using low-level OpenGL commands. It contains routines for setting up viewing and projection matrices. GLU routines begin with the prefix glu.

- All constants in OpenGL use the prefix "GL_". For example- glBegin(GL_POLYGON).

Header (.h) files: For all OpenGL programs, the header file glut.h needs to be included in the beginning of the program as: #include <GL/glut.h> //(glut.h includes gl.h and glu.h automatically)

Frame Buffer: A frame buffer is a portion of RAM containing colour values for every pixel (pixmap) that can be displayed on the screen in the form of 1's and 0's. An additional alpha channel is sometimes used to retain information about pixel transparency.

Therefore, a frame buffer is a digital device and the CRT is an analog device. Therefore, a conversion from a digital representation to an analog signal must take place when information is read from the frame buffer and displayed on the raster CRT graphics device. For this we can use a digital to analog converter (DAC).Each pixel in the frame buffer must be accessed and digital values representing the pixel colour should be converted to an analog voltage for the electron gun before it is visible on the raster CRT, as depicted in fig.1.

The total amount of the memory required to drive the frame buffer depends on the resolution of the output signal, and on the colour depth and palette size.

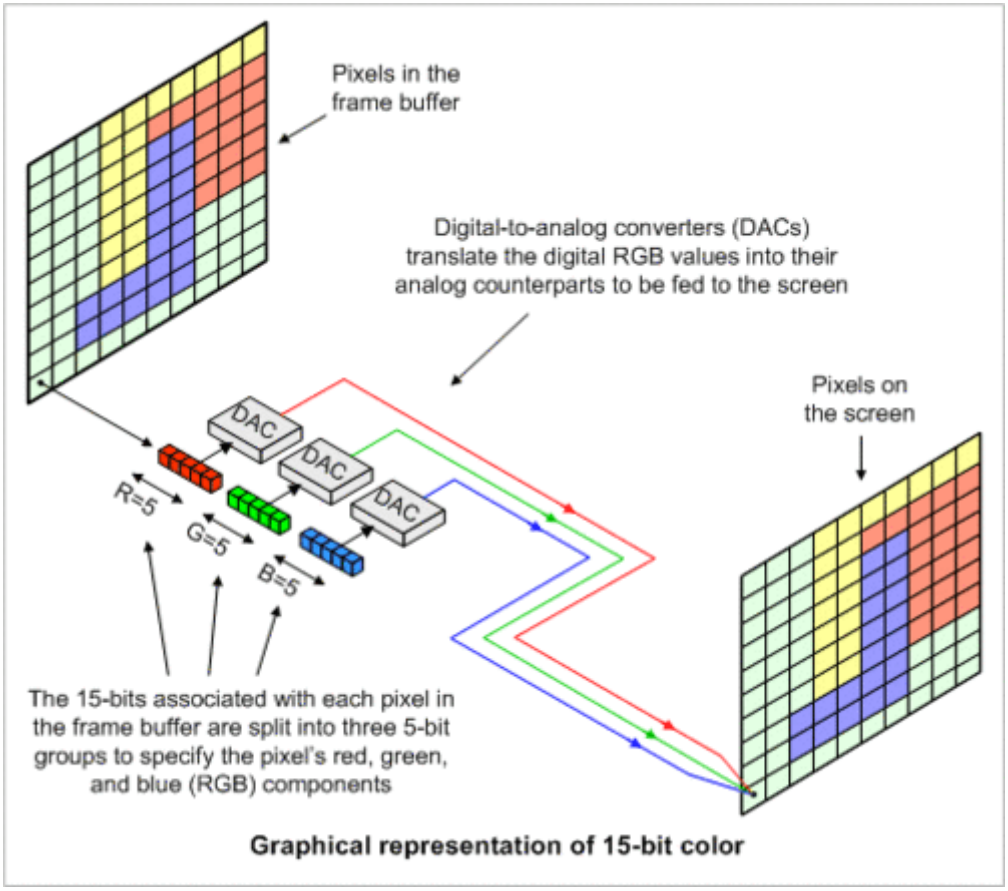


Fig.1 Displaying colors stored in frame buffer on the screen

Double Buffer: Double buffering provides two complete color buffers for use in drawing. One buffer is displayed while the other buffer is being drawn into. When the drawing is complete, the two buffers are swapped so that the one that was being viewed is now used for drawing. The swap is almost instantaneous, and thus provides a means of performing animation, like the way a sequence of still photographs, rapidly displayed, appear to depict a moving object.

GLUT Functions

1) glutInit : glutInit is used to initialize the GLUT library.

Syntax:- **glutInit (int argc, char **argv);**

argc -

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

argv -

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

***2) glutInitWindowPosition, glutInitWindowSize* :**

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

Syntax:- **void glutInitWindowSize(int width, int height);**
void glutInitWindowPosition(int x, int y);

width : Width in pixels.
Height:- Height in pixels.
x :- Window X location in pixels.
Y :- Window Y location in pixels.

3) glutInitDisplayMode:- glutInitDisplayMode sets the initial display mode.

Syntax:- **void glutInitDisplayMode(unsigned int mode);**

mode :- Display mode, normally the bitwise OR-ing of GLUT display mode bit masks. Mode can take following values.

Values	Meaning
GLUT_RGBA	Bit mask to select an RGBA mode window. This is the default if neither GLUT_RGBA nor GLUT_INDEX are specified.
GLUT_RGB	An alias for GLUT_RGBA.
GLUT_INDEX	Bit mask to select a color index mode window. This overrides GLUT_RGBA if it is also specified.
GLUT_SINGLE	Bit mask to select a single buffered window. This is the default if neither
GLUT_DOUBLE	Bit mask to select a double buffered window. This overrides GLUT_SINGLE if it is also specified.
GLUT_DEPTH	Bit mask to select a window with a depth buffer.

4) glutMainLoop:- glutMainLoop enters the GLUT event processing loop.
Syntax:- **void glutMainLoop(void);**

5) glutCreateWindow:- glutCreateWindow creates a top-level window.
Syntax:- **int glutCreateWindow(char *name);**
name :- ASCII character string for use as window name.

6)glutPositionWindow:- glutPositionWindow requests a change to the position of the current window.
Syntax:- **void glutPositionWindow(int x, int y);**

x :- New X location of window in pixels.
Y :- New Y location of window in pixels.

7) glutReshapeFunc :- glutReshapeFunc sets the reshape callback for the current window.
Syntax:- **void glutReshapeFunc(void (*func)(int width, int height));**
func :- The new reshape callback function.

8)glutDisplayFunc:- glutDisplayFunc sets the display callback for the current window.
Syntax:- **void glutDisplayFunc(void (*func)(void));**
func :- The new display callback function.

9) glutKeyboardFunc :- glutKeyboardFunc sets the keyboard callback for the current window.
Syntax:- **void glutKeyboardFunc(void (*func)(unsigned char key, int x, int y));**
func :- The new keyboard callback function.

10) glutMouseFunc:- glutMouseFunc sets the mouse callback for the current window.
Syntax:- void glutMouseFunc(void (*func)(int button, int state, int x, int y));
func :- The new mouse callback function.

GL Functions

1) glBegin & glEnd :- The **glBegin** and **glend** functions delimit the vertices of a primitive or a group of like primitives.
Syntax **:- void glBegin(GLenum mode);**
mode :- The primitive or primitives that will be created from vertices presented between **glBegin**and the subsequent **glEnd**. The following are accepted symbolic constants and their meaning.

Value	Meaning
GL_POINTS	Treats each vertex as a single point. Vertex *n* defines point *n*. *N* points are drawn.
GL_LINES	Treats each pair of vertices as an independent line segment. Vertices *2n - 1* and *2n* define line *n*. *N/2* lines are drawn.
GL_LINE_STRIP	Draws a connected group of line segments from the first vertex to the last. Vertices *n* and *n+1* define line *n*. *N - 1* lines are drawn.
GL_LINE_LOOP	Draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices *n* and *n + 1* define line *n*. The last line, however, is defined by vertices *N* and *1*. *N* lines are drawn.
GL_TRIANGLES	Treats each triplet of vertices as an independent triangle. Vertices *3n - 2*, *3n - 1*, and *3n* define triangle *n*. *N/3* triangles are drawn.
GL_TRIANGLE_STRIP	Draws a connected group of triangles. One

	triangle is defined for each vertex presented after the first two vertices. For odd *n*, vertices *n*, *n + 1*, and *n + 2* define triangle *n*. For even *n*, vertices *n + 1*, *n*, and *n + 2* define triangle *n*. *N - 2* triangles are drawn.
GL_TRIANGLE_FAN	Draws a connected group of triangles. one triangle is defined for each vertex presented after the first two vertices. Vertices *1*, *n + 1*, *n + 2* define triangle *n*. *N - 2* triangles are drawn.
GL_QUADS	Treats each group of four vertices as an independent quadrilateral. Vertices *4n - 3*, *4n - 2*, *4n - 1*, and *4n* define quadrilateral *n*. *N/4*quadrilaterals are drawn.
GL_QUAD_STRIP	Draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices *2n - 1*, *2n*, *2n + 2*, and *2n + 1* define quadrilateral *n*. *N/2 - 1* quadrilaterals are drawn. Note that the order in which vertices are used to construct a quadrilateral from strip data is different from that used with independent data.
GL_POLYGON	Draws a single, convex polygon. Vertices *1* through *N* define this polygon.

2) glclear :- The **glClear** function clears buffers to preset values. OpenGL offers four types of buffers. However we use only Color buffer(Frame buffer) and Depth buffer in the lab programs.

Syntax:- **void glClear(GLbitfield mask);**
mask :- Bitwise OR operators of masks that indicate the buffers to be cleared. The four masks are as follows.

Value	Meaning
GL_COLOR_BUFFER_BIT	The buffers currently enabled for color writing
GL_DEPTH_BUFFER_BIT	The depth buffer.
GL_ACCUM_BUFFER_BIT	The accumulation buffer.
GL_STENCIL_BUFFER_BIT	The stencil buffer.

3) glClearColor:- The **glClearColor** function specifies clear values for the color buffers.

Syntax **:-void glClearColor(red, green, blue, alpha);**
red :- The red value that **glClear** uses to clear the color buffers. The default value is zero.
green :-The green value that **glClear** uses to clear the color buffers. The default value is zero.
blue :- The blue value that **glClear** uses to clear the color buffers. The default value is zero.
alpha :-The alpha value that **glClear** uses to clear the color buffers. The default value is zero.

4) glColor3d :- Sets the current color.

Syntax :- **void glColor3i(GLint red, GLint green, GLint blue);**
red :- The new red value for the current color.
green :-The new green value for the current color.
blue :- The new blue value for the current color.

5) glColor3fv:- Sets the current color from an already existing array of color values.

Syntax :- **void glColor3fv(const GLfloat *v);**
V:- A pointer to an array that contains red, green, and blue values.

6) glEnable, glDisable :- The **glEnable** and **glDisable** functions enable or disable OpenGL capabilities.

Syntax **:- void glEnable(GLenum cap);**
void glDisable(GLenum cap);
cap :- Both **glEnable** and **glDisable** take a single argument, *cap*, which can assume one of the following values:

Value	Meaning
GL_DEPTH_TEST	If enabled, do depth comparisons and update the depth buffer..
GL_LINE_SMOOTH	If enabled, draw lines with correct filtering. If disabled, draw aliased lines.
GL_NORMALIZE	If enabled, normal vectors specified with glNormal are scaled to unit length after transformation.
glEnable(GL_LIGHTING); glEnable(GL_LIGHT0);	To enable lights from a single source (Light0).i.e., white light.

7) glFlush:- The **glFlush** function forces execution of OpenGL functions in finite time.

Syntax:- **void glFlush(void);**
This function has no parameters.

8) glFrustum:- The **glFrustum** function multiplies the current matrix by a perspective matrix.

Syntax :- **void glFrustum(GLdouble left, GLdouble right, GLdouble bottom,**
GLdouble top, GLdouble zNear, GLdouble zFar);

left :- The coordinate for the left-vertical clipping plane.
right :- The coordinate for the right-vertical clipping plane.
bottom :- The coordinate for the bottom-horizontal clipping plane.
top :- The coordinate for the bottom-horizontal clipping plane.
zNear :-The distances to the near-depth clipping plane. Must be positive.
zFar :- The distances to the far-depth clipping planes. Must be positive.

9) glLightfv:- The **glLightfv** function returns light source parameter values.

Syntax:- **void glLightfv(GLenum light, GLenum pname, const GLfloat *params);**
Light: The identifier of a light. The number of possible lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form GL_LIGHT*i* where *i* is a value: 0 to GL_MAX_LIGHTS - 1.
pname : A single-valued light source parameter for *light*. The following symbolic names are accepted. *param* : Specifies the value that parameter *pname* of light source *light* will be set to.

Value	Meaning
GL_AMBIENT	The *params* parameter contains four floating-point values that specify the ambient RGBA intensity of the light. Floating-point values are mapped directly. The default ambient light intensity is (0.0, 0.0, 0.0,1.0).
GL_DIFFUSE	The *params* parameter contains four floating-point values that specify the diffuse RGBA intensity of the light. Floating-point values are mapped directly.. The default diffuse intensity is (0.0, 0.0, 0.0, 1.0) for all lights other than light zero. The default diffuse intensity of light zero is (1.0, 1.0, 1.0, 1.0).
GL_SPECULAR	The *params* parameter contains four floating-point values that specify the specular RGBA intensity of the light. Floating-point values are mapped directly.. The default specular intensity is (0.0, 0.0, 0.0, 1.0) for all lights other than light zero. The default specular intensity of light zero is (1.0, 1.0, 1.0, 1.0).
GL_POSITION	The *params* parameter contains four floating-point values that specify the position of the light in homogeneous object coordinates. Both integer and floating-point values are mapped directly.

10) glLoadIdentity :- The **glLoadIdentity** function replaces the current matrix with the identity matrix.

Syntax :- **void glLoadIdentity(void);**

11)glPushMatrix & glPopMatrix:- The **glPushMatrix** and **glPopMatrix** functions push and pop the current matrix stack.

Syntax:- **void glPopMatrix(void);**

12) glMatrixMode:- The **glMatrixMode** function specifies which matrix is the current matrix.

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

mode :- The matrix stack that is the target for subsequent matrix operations. The *mode* parameter can assume one of three values.

Value	Meaning
GL_MODELVIEW	Applies subsequent matrix operations to the modelview matrix stack.
GL_PROJECTION	Applies subsequent matrix operations to the projection matrix stack.
GL_TEXTURE	Applies subsequent matrix operations to the texture matrix stack.

13) glOrtho:- The **glOrtho** function multiplies the current matrix by an orthographic matrix.

Syntax:- **void glOrtho(GLdouble left, GLdouble right, GLdouble bottom, GLdouble top, GLdouble zNear, GLdouble zFar);**

left :-The coordinates for the left vertical clipping plane.
right :- The coordinates for theright vertical clipping plane.
Bottom:- The coordinates for the bottom horizontal clipping plane.
top :- The coordinates for the top horizontal clipping plans.
zNear :- The distances to the nearer depth clipping plane. This distance is negative if the plane is to be behind the viewer.
zFar :- The distances to the farther depth clipping plane. This distance is negative if the plane is to be behind the viewer.

14) glPointSize :- The **glPointSize** function specifies the diameter of rasterized points.

Syntax :- **void glPointSize(GLfloat size);**
*Size:-*The diameter of rasterized points. The default is 1.0.

15) glPushMatrix & glPopMatrix:- The **glPushMatrix** and **glPopMatrix** functions push and pop the current matrix stack.
Syntax:- **void glPopMatrix(void);**

16) glRotatef:- The **glRotatef** function multiplies the current matrix by a rotation matrix.

Syntax :- **void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);**
angle :- The angle of rotation, in degrees.
X :- The *x* coordinate of a vector.
y :- The *y* coordinate of a vector.
z :- The *z* coordinate of a vector.

17) glScalef :- The **glScaled** and **glScalef** functions multiply the current matrix by a general scaling matrix.

Syntax :- **void glScalef(GLfloat x, GLfloat y, GLfloat z);**
x :- Scale factors along the *x* axis.
y :- Scale factors along the *y* axis.
z :- Scale factors along the *z* axis.

18) glTranslatef :- The **glTranslatef** function multiplies the current matrix by a translation matrix.

Syntax:- **void glTranslatef(GLfloat x, GLfloat y, GLfloat z);**
x :- The *x* coordinate of a translation vector.
y :- The *y* coordinate of a translation vector.
z :- The *z* coordinate of a translation vector.

19) glVertex2d :- Specifies a vertex.

*Syntax***:- void glVertex2d(GLdouble x, GLdouble y);**
x :- Specifies the x-coordinate of a vertex.
y :-Specifies the y-coordinate of a vertex.

20) glViewport :- The **glViewport** function sets the viewport.

Syntax:- **void glViewport(GLint x, GLint y, GLsizei width, GLsizei height);**
x :- The lower-left corner of the viewport rectangle, in pixels. The default is (0,0).
y :- The lower-left corner of the viewport rectangle, in pixels. The default is (0,0).
Width :-The width of the viewport. When an OpenGL context is first attached to a window, *width*and *height* are set to the dimensions of that window.
height :- The height of the viewport. When an OpenGL context is first attached to a window, *width*and *height* are set to the dimensions of that window.

GLU Functions

1) gluLookAt :- The **gluLookAt** function defines a viewing transformation.

Syntax:- **void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez,**
GLdouble centerx, GLdouble centery,GLdouble centerz,
GLdouble upx, GLdouble upy, GLdouble upz);
eyex :- The position of the eye point.
eyey :-The position of the eye point.
eyez :- The position of the eye point.
centerx :- The position of the reference point.
centery :- The position of the reference point.

centerz :- The position of the reference point.
upx :- The direction of the up vector.
upy :- The direction of the up vector.
upz :- The direction of the up vector.

2) gluOrtho2D :- The **gluOrtho2D** function defines a 2-D orthographic projection matrix.

Syntax:- **void gluOrtho2D (GLdouble left, GLdouble right, GLdouble GLdouble top, GLdouble bottom);**
left :- The coordinate for the left vertical clipping plane.
right :- The coordinate for the right vertical clipping plane.
top :- The coordinate for the top horizontal clipping plane.
bottom :- The coordinate for the bottom horizontal clipping plane.

PROGRAM -1

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

```
#include<GL/glut.h>
#include<stdio.h>
int x1,y1,x2,y2;
void minit()
{
glClear(GL_COLOR_BUFFER_BIT);
glClearColor(0.0,0.0,0.0,1.0);
glMatrixMode(GL_PROJECTION);
gluOrtho2D(0,500,0,500);
}
void dp(int x,int y)
{
glBegin(GL_POINTS);
glVertex2i(x,y);
glEnd();
}
void dl(int x1,int x2,int y1,int y2)
{
int x,y,cx,cy,dx,dy,c1,c2,i,e;
dx=x2-x1;
dy=y2-y1;
if(dx<0) dx=-dx;
if(dy<0) dy=-dy;
cx=1;
if(x2<x1)
cx=-1;
cy=1;
if(y2<y1)
cy=-1;
x=x1;
y=y1;
if(dx>dy)
{
printf("%d\t%d\n",x,y);
dp(x,y);
e=2*dy-dx;
c1=2*(dy-dx);
c2=2*dy;
for(i=0;i<dx;i++)
{
if(e>=0)
```

```
{
y+=cy;
e+=c1;
}
else
e+=c2;
x+=cx;
printf("%d\t%d\n",x,y);
dp(x,y);
}
}
else
{
printf("%d\t%d\n",x,y);
dp(x,y);
e=2*dy-dx;
c1=2*(dx-dy);
c2=2*dx;
for(i=0;i<dy;i++)
{
if(e>=0)
{
x+=cx;
e+=c1;
}
else
e+=c2;
y+=cy;
printf("%d\t%d\n",x,y);
dp(x,y);
}
}
}
void mdisplay()
{
dl(x1,x2,y1,y2);
glFlush();
}
int main(int argc,char **argv)
{
glutInit(&argc,argv);
printf("enter 1st point\n");
scanf("%d%d", &x1, &y1);
printf("enter 2nd point\n");
scanf("%d%d", &x2, &y2);
```

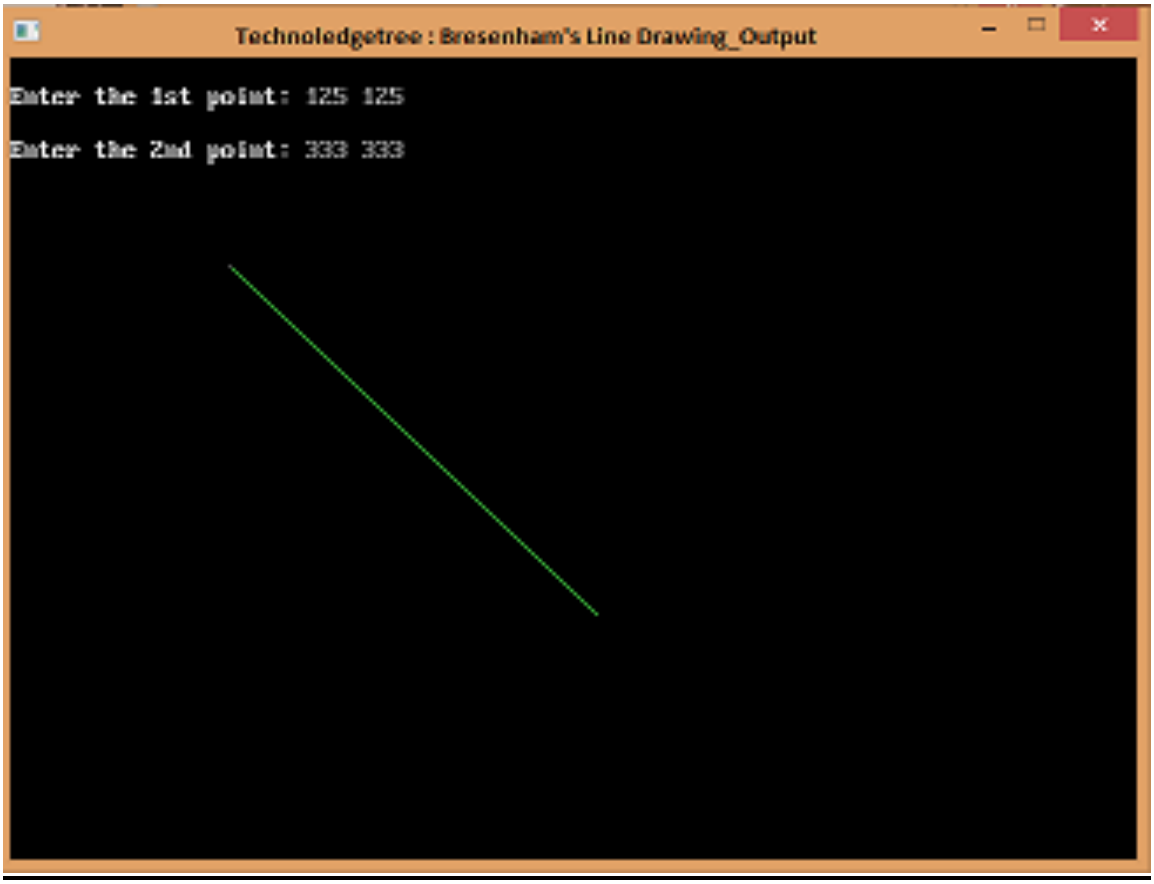
```
printf("\n\n\n\n");
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(1000,1000);
glutInitWindowPosition(0,0);
glutCreateWindow("bresenhams");
minit();
glutDisplayFunc(mdisplay);
glutMainLoop();
}
```

Output command

To create file - gedit filename.c

To compile file - gcc filename.c -lGL -lGLU -lglut

To execute - ./a.out



PROGRAM -2

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

```
#include<GL/glut.h>
#define NULL 0
static GLfloat angle=90;
int sb,db;
void dd()
{
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1,0,0);

glBegin(GL_LINES);
glVertex2f(-2,0);
glVertex2f(2,0);
glVertex2f(0,2);
glVertex2f(0,-2);
glEnd();

glColor3f(1,0,1);
glBegin(GL_TRIANGLES);
glVertex2f(0.3,0.2);
glVertex2f(0,0);
glVertex2f(0.2,0.3);
glEnd();

glColor3f(0,1,0);
glRotatef(90,0,0,1);
glBegin(GL_TRIANGLES);
glVertex2f(0.3,0.2);
glVertex2f(0,0);
glVertex2f(0.2,0.3);
glEnd();
glutSwapBuffers();
}

void ds()
{
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1,0,0);
glBegin(GL_LINES);
glVertex2f(-2,0);
glVertex2f(2,0);
glVertex2f(0,2);
```

```
glVertex2f(0,-2);
glEnd();

glColor3f(1,0,1);
glBegin(GL_TRIANGLES);
glVertex2f(0.3,0.2);
glVertex2f(0.6,0.2);
glVertex2f(0.6,0.6);
glEnd();

glPushMatrix();
glTranslatef(0.3,0.2,0.0);
glRotatef(90,0,0,1);
glTranslatef(-0.3,-0.2,0.0);

glColor3f(0,1,0);
glBegin(GL_TRIANGLES);
glVertex2f(0.3,0.2);
glVertex2f(0.6,0.2);
glVertex2f(0.6,0.6);
glEnd();

glPopMatrix();
}

void sd()
{
glutSetWindow(sb);
glLoadIdentity();
glutSetWindow(db);
glLoadIdentity();
glutPostRedisplay();
}

void minit()
{
glClearColor(1,1,1,1);
glColor3f(0,1,1);
glShadeModel(GL_FLAT);
}

void myres(int w, int h)
{
glViewport(0,0,w,h);
glMatrixMode(GL_PROJECTION);
```

```
glLoadIdentity();
if(w<=h)
glOrtho(-1,1,-1*(GLfloat)h/(GLfloat)w,1*(GLfloat)h/(GLfloat)w,-1,1);
else
glOrtho(-1*(GLfloat)w/(GLfloat)h,1*(GLfloat)w/(GLfloat)h,
-1,1,-1,1);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

int main(int argc, char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(300,350);
glutInitWindowPosition(700,300);
sb=glutCreateWindow("FPR");
minit();
glutDisplayFunc(ds);
glutReshapeFunc(myres);
glutIdleFunc(sd);
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
glutInitWindowSize(300,350);
glutInitWindowPosition(400,0);
db=glutCreateWindow("OR");
minit();
glutDisplayFunc(dd);
glutReshapeFunc(myres);
glutIdleFunc(sd);
glutMainLoop();
return 0;
}
```

Output command

To create file - gedit filename.c

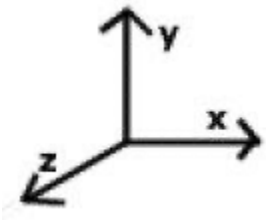
To compile file - gcc filename.c -lGL -lGLU -lglut

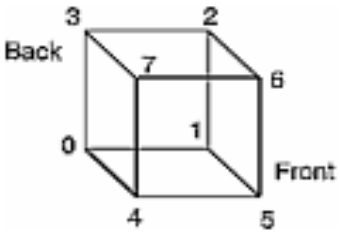
To execute - ./a.out

PROGRAM -3

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

In the right-handed coordinate system OpenGL uses, the X axis points right, the Y axis up, and the Z axis points forward. Cube has six faces and eight vertices.





To define the position of each vertex of the cube (0, 1, 2, 3, 4, 5, 6, 7) as given below:

-1.0,-1.0,-1.0 //left, bottom, back – vertex 0

1.0,-1.0,-1.0 //right, bottom, back– vertex 1

1.0, 1.0,-1.0 //right, top, back– vertex 2

-1.0, 1.0,-1.0 //left, top, back– vertex 3

-1.0,-1.0, 1.0 //left, bottom, front– vertex 4

1.0,-1.0, 1.0 //right, bottom, front– vertex 5

1.0, 1.0, 1.0 //right, top, front– vertex 6

-1.0, 1.0, 1.0 //left, top, front– vertex 7

This program demonstrates double buffering for flicker free animation. While content of one buffer is being displayed, the pixel values of the cube for the next position is computed and stored in the other buffer. Then the buffers are swapped.

The glutInitDisplayMode function is used to set up the display mode using the following flags:

- **GLUT_RGB** specifies we want an RGB colour buffer in our window
- **GLUT_DOUBLE** specifies we want a double buffer. Double buffering enables us to finish drawing before our image is sent to the screen, preventing flicker.
- **GLUT_DEPTH** specifies we want a depth buffer. The depth buffer ensures that objects near the camera will always be on top of those further away.

glutMouseFunc(*void(*callback)(int button*, *int state*, *int x*, *int y)*);

Whenever a mouse button is pressed or released in an OpenGLUT window, OpenGLUT checks if that window has a mouse-button (Mouse) callback registered. If so,it gives the event to the handler. Parameter button is the button number, starting from 0. State is GLUT_UP or GLUT_DOWN to indicate the button's new state. The other parameters are the mouse coordinates.

void glutIdleFunc(void (*func)(void));

GLUTprogram performs background processing tasks or continuous animation when window system events are not being received. If enabled, the idle callback is continuously called when events are not being received. In this program the function spincube( ) is continuously called till a keyboard hit.

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

glRotate produces a rotation of *angle* degrees around the vector x, y, z . The current matrix is multiplied by a rotation matrix with the product replacing the current matrix.' f ' signifies the parameters passed are float values.

//program to rotate a cube

```
#include<GL/glut.h>
float v[][3]={{0,0,0},{1,0,0},{1,1,0},{0,1,0},{0,0,1},{1,0,1},{1,1,1},{0,1,1}};
float theta[]={0,0,0};
int axis=0;

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

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

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

void spinCube()
{
theta[axis]+=5;
if(theta[axis]>360)
theta[axis]=0;
display();
}

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;
}

void myReshape(int w, int h)
{
glViewport(0,0,w,h);
```

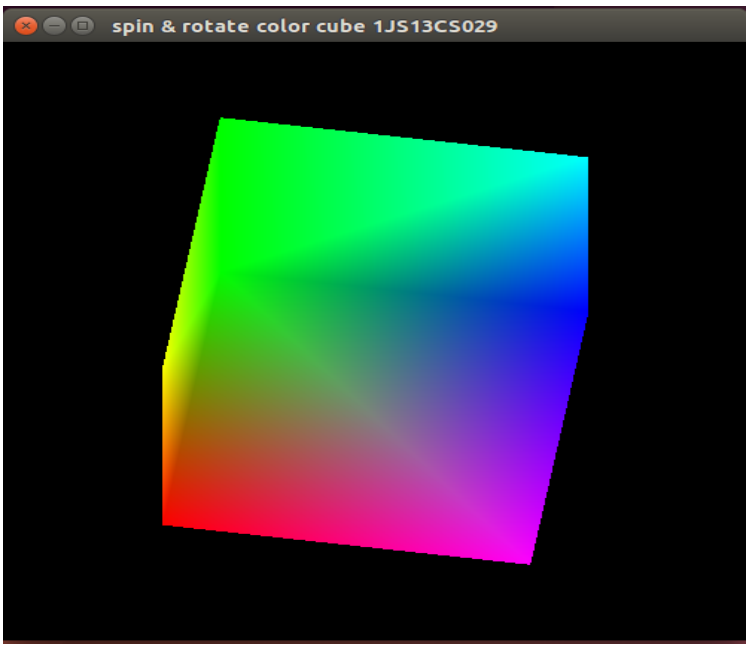
```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if(w<=h)
glOrtho(-2,2,-2*h/w,2*h/w,-10,10);
else
glOrtho(-2*w/h,2*w/h,-2,2,-10,10);
glMatrixMode(GL_MODELVIEW);
}

int main(int argc,char **argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DOUBLE|GLUT_DEPTH);
glutInitWindowSize(500,500);
glutCreateWindow("CUBE");
glutReshapeFunc(myReshape);
glutDisplayFunc(display);
glutIdleFunc(spinCube);
glutMouseFunc(mouse);
glEnable(GL_DEPTH_TEST);
glutMainLoop();
}
```

Output command

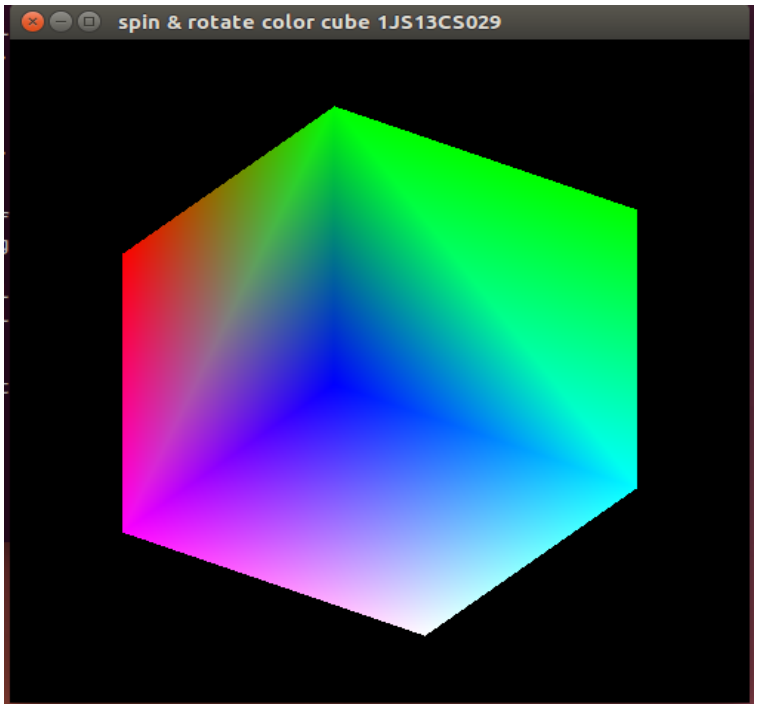
To create file - gedit filename.c

To compile file - gcc filename.c -lGL -lGLU -lglut

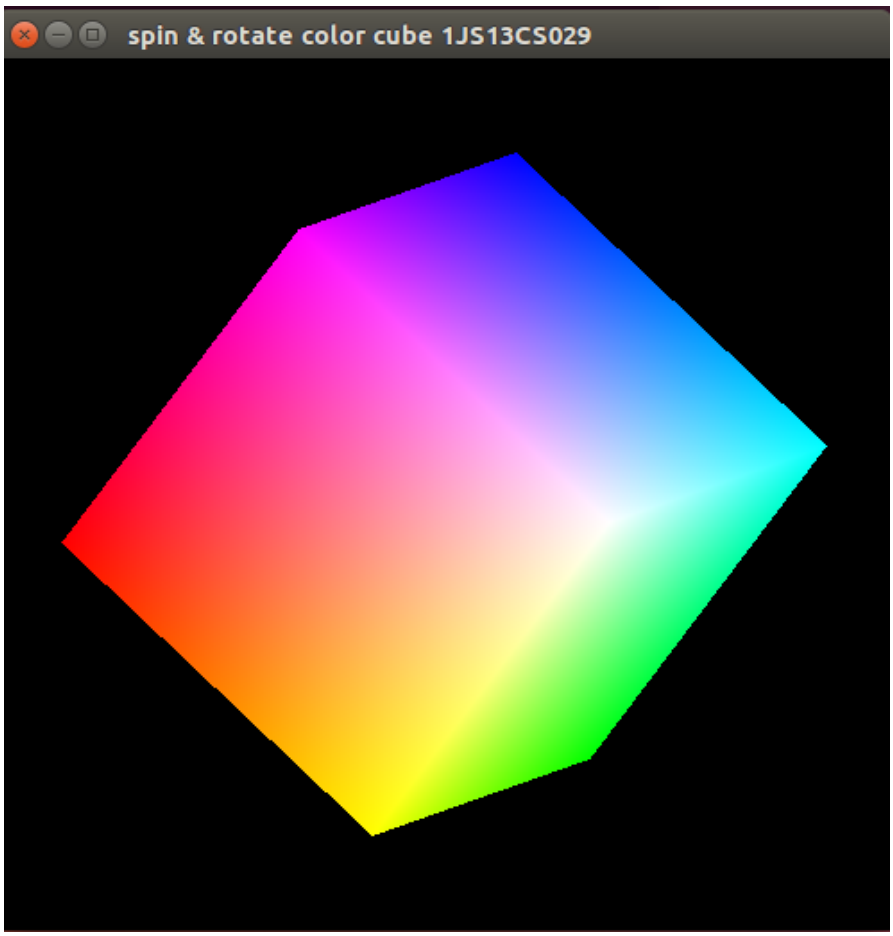
To execute - ./a.out



Rotation of cube in Z-axis (when Right button is pressed)



Rotation of cube in X-axis (when Left button is pressed)



Rotation of cube in Y-axis (when middle button is pressed)

PROGRAM -4

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

```
#include<GL/glut.h>
#include<stdio.h>
#include<stdlib.h>
float v[8][3]={{0,0,0},{1,0,0},{1,1,0},{0,1,0},{0,0,1},{1,0,1},{1,1,1},{0,1,1}};
float v1[3]={0,0,5};
void polygon(int a, int b, int c, int d);
void polygon1();

void display()
{
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glClearColor(1,1,1,1);
glLoadIdentity();
gluLookAt(v1[0],v1[1],v1[2],0,0,0,0,1,0);
polygon1();
glFlush();
}

void polygon1()
{
polygon(0,1,2,3);
polygon(4,5,6,7);
polygon(5,1,2,6);
polygon(4,0,3,7);
polygon(4,5,1,0);
polygon(7,6,2,3);
}

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

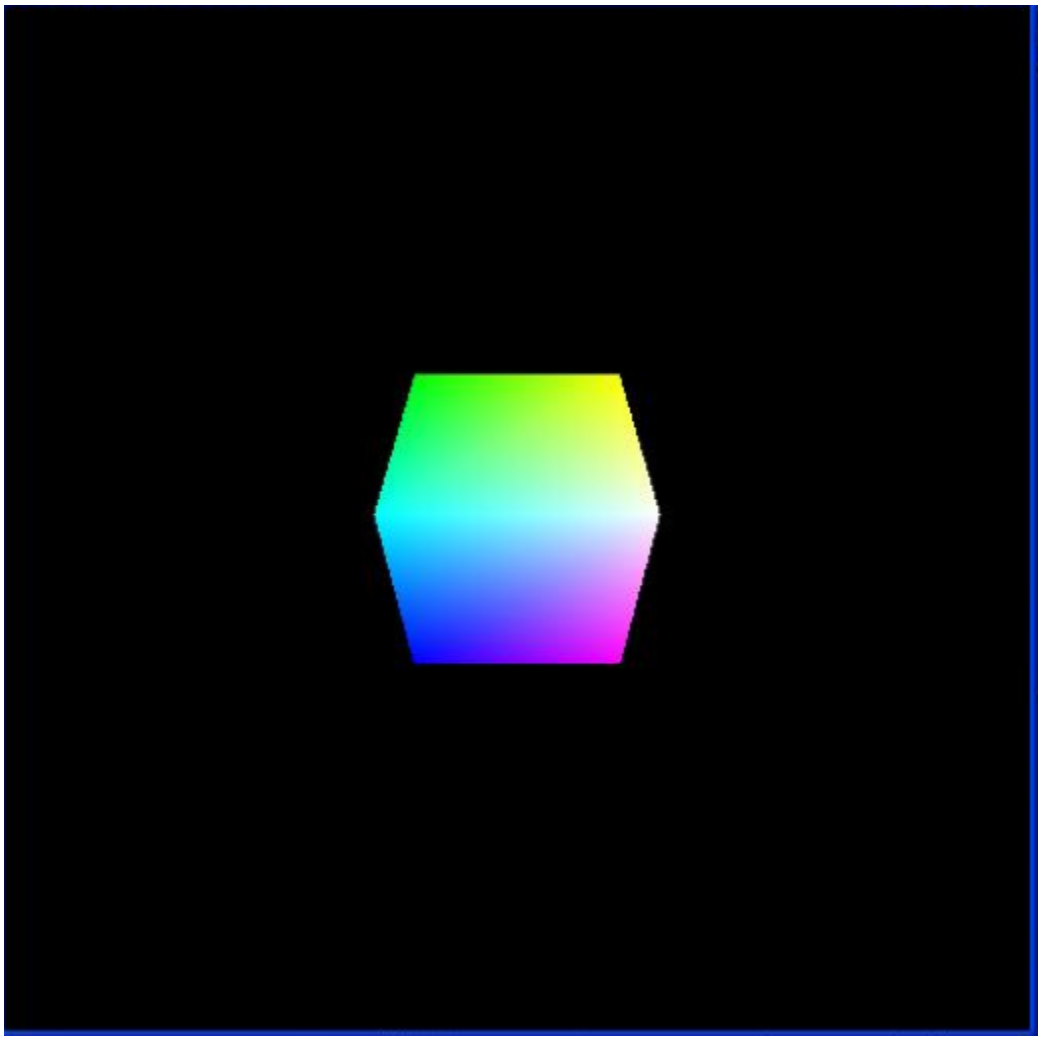
```
glEnd();
}

void key(unsigned char f, int mouseX, int mouseY)
{
if(f=='x')
v1[0]-=0.1;
if(f=='X')
v1[0]+=0.1;
if(f=='y')
v1[1]-=0.1;
if(f=='Y')
v1[1]+=0.1;
if(f=='z')
v1[2]-=0.1;
if(f=='Z')
v1[2]+=0.1;
display();
}

void reshape(int w, int h)
{
glViewport(0,0,w,h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if(w<=h)
glFrustum(-2.0,2.0,-2.0*h/w,2.0*h/w,2.0,20);
else
glFrustum(-2.0*w/h,2.0*w/h,-2.0,2.0,2.0,20);
glMatrixMode(GL_MODELVIEW);
}

int main(int argc, char **argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_DEPTH);
glutInitWindowSize(1500,1500);
glutInitWindowPosition(10,10);
glutCreateWindow("cube presp");
glutDisplayFunc(display);
glutKeyboardFunc(key);
glEnable(GL_DEPTH_TEST);
glutReshapeFunc(reshape);
glutMainLoop();
}
```

Output command

To create file - gedit filename.c

To compile file - gcc filename.c -lGL -lGLU -lglut

To execute - ./a.out



PROGRAM -5

5. Clip a lines using Cohen-Sutherland algorithm.

Cohen-Sutherland Algorithm for clipping the line:

- Step 1 − Assign a region code (outcode) for each endpoints.
- Step 2 − If both endpoints have a region code 0000 then accept this line.
- Step 3 − Else, perform the logical AND operation for both region codes.(If both codes have a 1 in the same bit position) If the result is not 0000, then reject the line. Else you need clipping.
- Step 4 − Choose an endpoint of the line that is outside the window.
- Step 5 − Find the intersection point at the window boundary (based on region code).
- Step 6 − Replace endpoint with the intersection point and update the region code.
- Repeat step 2 until we find a clipped line either trivially accepted or trivially rejected.

Assignment of a region code for each endpoints:

```
RIGHT=8;//1000
TOP=4;//0100
LEFT=2;//0010
```

Outcode =0100: if $y > y_{max}$

Outcode =0001: if $y < y_{min}$

Outcode =1000: if $x > x_{max}$

Outcode =0010: if $x < x_{min}$

Test for trivial acceptance or trivial rejection using bitwise functions:

if outcode0| outcode1= 0000 accept (draw)

else if outcode0 & outcode1≠0000 reject (don't draw)

To find Intersection points of the line with edges of the window:

Equation to find intersection point of 2 lines:

if (outcodeOut & TOP)

```
        x = x0 + (x1 - x0)*(ymax - y0)/(y1 - y0);
        y = ymax;
    else if (outcodeOut & BOTTOM)
            x = x0 + (x1 - x0)*(ymin - y0)/(y1 - y0);
            y = ymin;
    else if (outcodeOut & RIGHT)
            y = y0 + (y1 - y0)*(xmax - x0)/(x1 - x0);
            x = xmax;
    else if (outcodeOut & LEFT)
            y = y0 + (y1 - y0)*(xmin - x0)/(x1 - x0);
            x = xmin;
    if (outcodeOut == outcode0)
                x0 = x; y0 = y;
            outcode0 = CompOutCode(x0, y0);
    else
                x1 = x; y1 = y;
            outcode1 = CompOutCode(x1, y1);
```

Window-to-viewport conversion is performed by the following sequence of transformations:

1. Perform a scaling transformation using a fixed-point position of (xmin, ymin) that scales the window area to the size of the viewport.

Scale factor in x direction =sx = $\frac{\text{Width of viewport}}{\text{Width of window}}$

Scale factor in x direction =sy = $\frac{\text{Height of viewport}}{\text{Height of window}}$

Therefore sx = $\frac{xvmax - xvmin}{xmax - xmin}$ and sy = $\frac{yvmax - yvmin}{ymax - ymin}$

Note: Relative proportions of objects are maintained if the scaling factors are the same (sx = sy). Otherwise, world objects will be stretched or contracted in either the x or y direction when displayed in viewport.

2. Translate the scaled window area to the position of the viewport.

Resultant equations are: xv = xvmin + (x - xmin)*sx

yv = yvmin + (y – ymin)*sy

//program to clip a line using Cohen Sutherland algorithm

```
#include<stdio.h>
#include<GL/glut.h>
float xmin=50,ymin=50,xmax=100,ymax=100;
float xvmin=200,yvmin=200,xvmax=400,yvmax=400;
int right=2,left=1,top=8,bottom=4;
float sx,sy,vx1,vy1,vx2,vy2;
float x1,y1,x2,y2;
int compute(float x,float y)
{
int code=0;
if(y>ymax)
code=top;
else if(y<ymin)
code=bottom;
if(x>xmax)
code=right;
else if(x<xmin)
code=left;
return code;
}

void cohen(float x1,float y1,float x2,float y2)
{
float x,y;
int a=0,d=0,cp,cq;
int code;
cp=compute(x1,y1);
cq=compute(x2,y2);
do
{
if(!(cp | cq))
```

```
{
a=1;
d=1;
}
else if(cp & cq)
d=1;
else
{
code=cp ? cp : cq;
if(code & top)
{
x=x1+(x2-x1)*(ymax-y1)/(y2-y1);
y=ymax;
}
else if(code & bottom)
{
x=x1+(x2-x1)*(ymin-y1)/(y2-y1);
y=ymin;
}
else if(code & right)
{
y=y1+(y2-y1)*(xmax-x1)/(x2-x1);
x=xmax;
}
else
{
y=y1+(y2-y1)*(xmin-x1)/(x2-x1);
x=xmin;
}
if(code==cp)
{
x1=x;
y1=y;
cp=compute(x1,y1);
}
else
{
x2=x;
y2=y;
cq=compute(x2,y2);
}
}
}while(!d);
if(a)
{
```

```
sx=(xvmax-xvmin)/(xmax-xmin);
sy=(yvmax-yvmin)/(ymax-ymin);
vx1=xvmin+(x1-xmin)*sx;
vy1=xvmin+(y1-ymin)*sy;
vx2=xvmin+(x2-xmin)*sx;
vy2=xvmin+(y2-ymin)*sy;
}
}

void display()
{
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1,0,1);
glLineWidth(2);

glBegin(GL_LINES);
glVertex2f(x1,y1);
glVertex2f(x2,y2);
glEnd();

glColor3f(1,0,0);

glBegin(GL_LINE_LOOP);
glVertex2f(xmin,ymin);
glVertex2f(xmax,ymin);
glVertex2f(xmax,ymax);
glVertex2f(xmin,ymax);
glEnd();

cohen(x1,y1,x2,y2);

glColor3f(0,0,1);

glBegin(GL_LINE_LOOP);
glVertex2f(xvmin,yvmin);
glVertex2f(xvmax,yvmin);
glVertex2f(xvmax,yvmax);
glVertex2f(xvmin,yvmax);
glEnd();

glColor3f(0,1,0);
glBegin(GL_LINES);
glVertex2f(vx1,vy1);
glVertex2f(vx2,vy2);
glEnd();
```

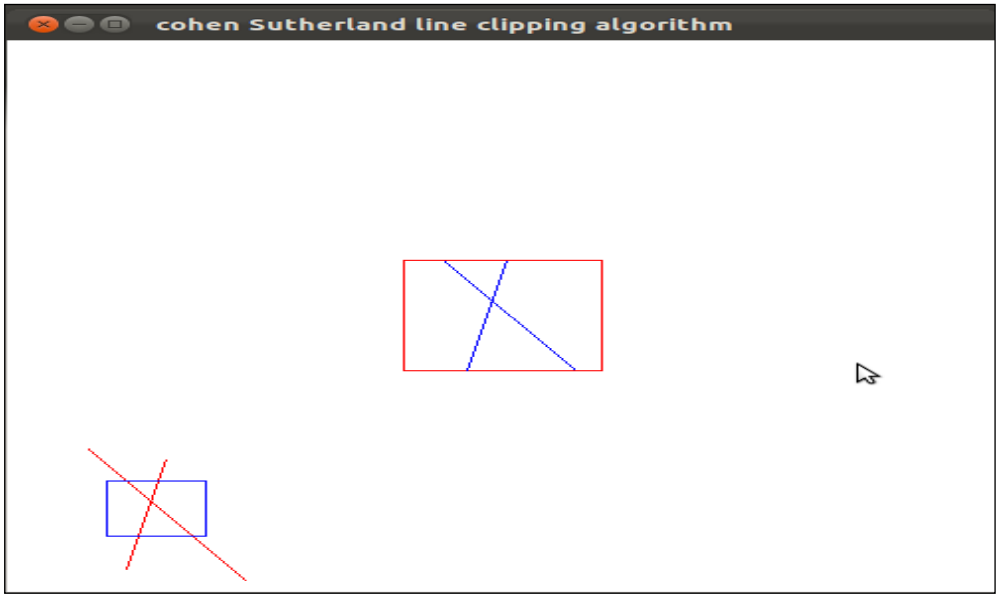
```
glFlush();
}

void minit()
{
glClearColor(1,1,1,1);
gluOrtho2D(0,500,0,500);
}

int main(int argc,char **argv)
{glutInit(&argc, argv);
 printf("enter pts\n");
 scanf("%f%f%f%f",&x1,&y1,&x2,&y2);
 glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
 glutInitWindowSize(1500,1500);
 glutCreateWindow("Cohen Sutherland Line Clipping Algorithm");
 glutDisplayFunc(display);
 minit();
 glutMainLoop();
 return 0;
}
```

Output command

To create file - gedit filename.c

To compile file - gcc filename.c -lGL -lGLU -lglut

To execute - ./a.out



Cohen sutherland line clipping.

PROGRAM -6

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

Description

Light: The identifier of a light. The number of possible lights depends on the implementation, but at least eight lights are supported. They are identified by symbolic names of the form GL_LIGHT*i* where *i* is a value: 0 to GL_MAX_LIGHTS - 1.

glLightfv:- The **glLightfv** function returns light source parameter values.

Syntax:- **void glLightfv(GLenum light, GLenum pname, const GLfloat *params);**

pname : A single-valued light source parameter for *light*. The following symbolic names are accepted. *param* : Specifies the value that parameter *pname* of light source *light* will be set to.

To enable lights from a single source (Light0):

glEnable(GL_LIGHTING);

glEnable(GL_LIGHT0);

Defaults: Light 0: white light (1.0, 1.0, 1.0, 1.0) in RGBA diffuse and specular components.

Other lights: black (0.0, 0.0, 0.0, 1.0)

Position: (0.0, 0.0, 1.0, 0.0) in Homogeneous Coordinates, w=0.0 means ∞ distance,i.e. they represent a direction, not a point.

• Once lighting is enabled, colors assigned by **glColor*( )** are no longer used.

Normals:

• When specifying polygon vertices, we must supply the normal vectors to each vertex.

• To enable automatic normalization, use:

glEnable(GL_NORMALIZE);

The OpenGL light model presumes that the light that reaches your eye from the polygon surface arrives by four different mechanisms:

- AMBIENT - light that comes from all directions equally and is scattered in all directions equally by the polygons in your scene. This isn't quite true of the real world - but it's a good first approximation for light that comes pretty much uniformly from the sky and arrives onto a surface by bouncing off so many other surfaces that it might as well be uniform.

- DIFFUSE - light that comes from a particular point source (like the Sun) and hits surfaces with an intensity that depends on whether they face towards the light or away from it. However, once the light radiates from the surface, it does so equally in all directions. It is diffuse lighting that best defines the shape of 3D objects.

- SPECULAR - as with diffuse lighting, the light comes from a point souce, but with specular lighting, it is reflected more in the manner of a mirror where most of the light bounces off in a particular direction defined by the surface shape. Specular lighting is what produces the shiney highlights and helps us to distinguish between flat, dull surfaces such as plaster and shiney surfaces like polished plastics and metals.

- EMISSION - in this case, the light is actually emitted by the polygon - equally in all directions.

So, there are three light colours for each light - Ambient, Diffuse and Specular (set with glLight) and four for each surface (set with glMaterial). All OpenGL implementations support at least eight light sources - and the glMaterial can be changed at will for each polygon (although there are typically large time penalties for doing that - so we'd like to minimise the number of changes)

The final object colour is the sum of all four light components, each of which is formed by multiplying the glMaterial colour by the glLight colour (modified by the directionality in the case of Diffuse and Specular). Since there is no Emission colour for the glLight, that is added to the final colour without modification.A good set of settings for a light source would be to set the Diffuse and Specular components to the colour of the light source, and the Ambient to the same colour - but at MUCH reduced intensity, 10% to 40% seems reasonable in most cases.

For the glMaterial, it's usual to set the Ambient and Diffuse colours to the natural colour of the object and to put the Specular colour to white. The emission colour is generally black for objects that do not shine by their own light.

In this program a solid cube is transformed into Table legs, table top and walls by using translation and scaling transformations. Teapot is drawn using the built-in function **glutSolidTeapot(size);**

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

Syntax:- **void glPopMatrix(void);**

```
#include<stdio.h>
#include<GL/glut.h>
void wall()
{
glPushMatrix();
glScalef(2,0.05,2);
glutSolidCube(2);
glPopMatrix();

glPushMatrix();
glTranslatef(-2,2,0);
glRotatef(-90,0,0,1);
glScalef(2,0.05,2);
glutSolidCube(2);
glPopMatrix();

glPushMatrix();
glTranslatef(0,2,-2);
glRotatef(90,1,0,0);
glScalef(2,0.05,2);
glutSolidCube(2);
glPopMatrix();
}

void table()
{glPushMatrix();
glTranslatef(0,0.5,0);
glScalef(1,0.05,1);
glutSolidCube(2);
glPopMatrix();

glPushMatrix();
glTranslatef(-0.8,0.2,0.8);
glScalef(0.1,0.25,0.1);
glutSolidCube(2);
glPopMatrix();

glPushMatrix();
glTranslatef(0.8,0.2,0.8);
glScalef(0.1,0.25,0.1);
glutSolidCube(2);
glPopMatrix();

glPushMatrix();
```

```
glTranslatef(0.8,0.2,-0.8);
glScalef(0.1,0.25,0.1);
glutSolidCube(2);
glPopMatrix();

glPushMatrix();
glTranslatef(-0.8,0.2,-0.8);
glScalef(0.1,0.25,0.1);
glutSolidCube(2);
glPopMatrix();
}

void teapot()
{
glPushMatrix();
glTranslatef(0,1.3,0);
glRotatef(45,0,1,0);
glutSolidTeapot(1);
glPopMatrix();
}

void display(void)
{
float amb[]={1,0,0,1};
float pos[]={2,4,1};
glMaterialfv(GL_FRONT,GL_AMBIENT,amb);
glLightfv(GL_LIGHT0,GL_POSITION,pos);
glLightfv(GL_LIGHT0,GL_AMBIENT,amb);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-4,4,-4,4,-10,10);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(2.5,1,2,0,0.5,0,0,1,0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
wall();
table();
teapot();
glFlush();
}

int main(int argc,char **argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_DEPTH|GLUT_RGB);
```

```
glutInitWindowSize(600,600);
glutCreateWindow("TEAPOT");
glutDisplayFunc(display);
glEnable(GL_DEPTH_TEST);
glEnable(GL_SMOOTH);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_NORMALIZE);
glutMainLoop();
}
```

Output command

To create file - gedit filename.c

To compile file - gcc filename.c -lGL -lGLU -lglut

To execute - ./a.out

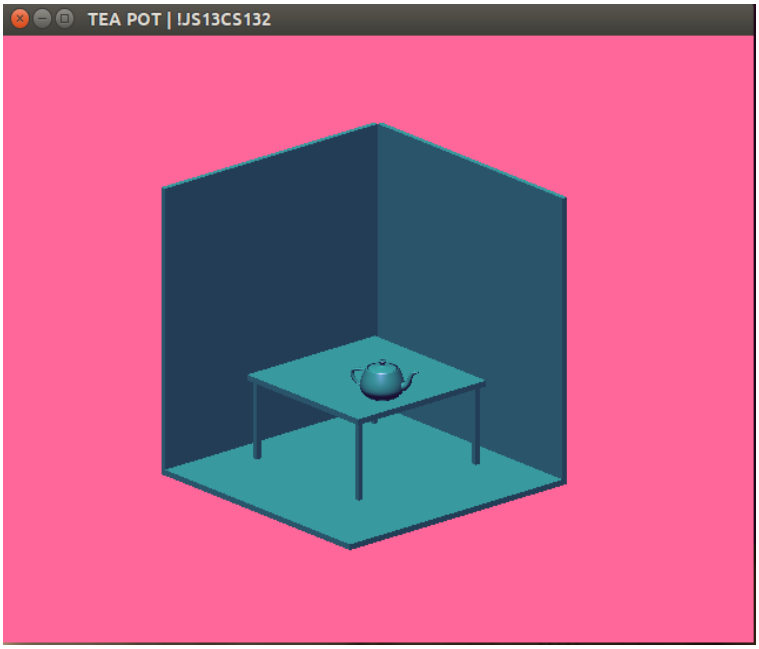


Fig: Teapot on the table

PROGRAM -7

7. Design, develop and implement recursively subdivide a tetrahedron to form 3D sierpinski gasket.

3D Sierpinski gasket (sieve) is a fractal based on a triangle with four equal triangles inscribed in it. The central triangle is removed and each of the other three treated as the original was, and so on, creating an infinite regression in a finite space.

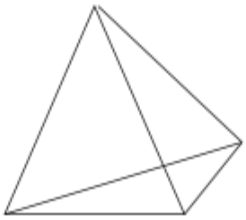


Figure (A)

Figure (B)

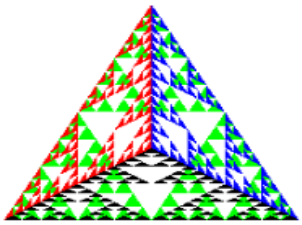


Figure (C)

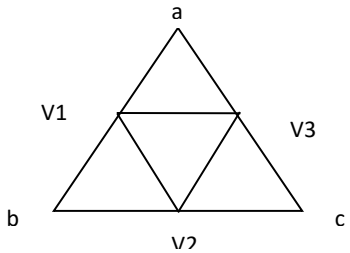
Steps to draw a 3D Sierpinski gasket :

1. Draw a Tetrahedron using initial triangle points - Figure (A)
2. Triangle subdivision using vertex coordinates -*Figure (B)*

   v1 [j] = (a[j] + b[j])/2

   v2 [j] = (b[j] + c[j])/2

   v3 [j] = (c[j] + a[j])/2



3. If m>0 // m Number of recursive steps.

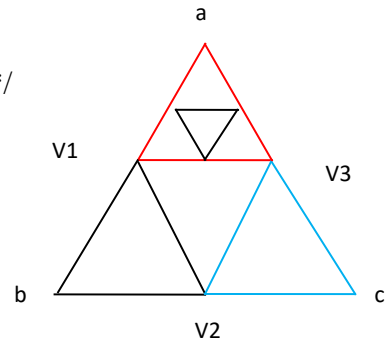
   /*Treat the each new triangle as Original one and re-divide it*/

   divide_triangle(a,v1,v2,m-1);

   divide_triangle(c,v2,v3,m-1);

   divide_triangle(b,v3,v1,m-1);

   else (triangle (a, b, c)); /*Draw triangle at end of recursion */



4. 3D Gasket after 5th iteration- *Figure(C)*

```
#include<stdio.h>
#include<GL/glut.h>
float v[][3]={{-1,-0.5,0},{1,-0.5,0},{0,1,0},{0,0,1}};
int m;
void triangle(float *p, float *q, float *r)
{
glVertex3fv(p);
glVertex3fv(q);
glVertex3fv(r);
}
void tetra(float *a, float *b, float *c, float *d)
{
glColor3f(1,0,0);
triangle(a,b,c);
glColor3f(1,1,0);
triangle(a,b,d);
glColor3f(1,0,1);
triangle(a,d,c);
glColor3f(0,1,1);
triangle(b,c,d);
}
void dt(float *a, float *b, float *c, float *d, int m)
{
float mid[6][3];
int j;
if(m>0)
{
for(j=0;j<3;j++)
{
mid[0][j]=(a[j]+b[j])/2;
mid[1][j]=(a[j]+c[j])/2;
mid[2][j]=(a[j]+d[j])/2;
mid[3][j]=(b[j]+c[j])/2;
mid[4][j]=(b[j]+d[j])/2;
mid[5][j]=(c[j]+d[j])/2;
}
dt(a,mid[0],mid[1],mid[2],m-1);
dt(mid[0],b,mid[3],mid[4],m-1);
dt(mid[1],mid[3],c,mid[5],m-1);
dt(mid[2],mid[4],mid[5],d,m-1);
}
else
tetra(a,b,c,d);
```

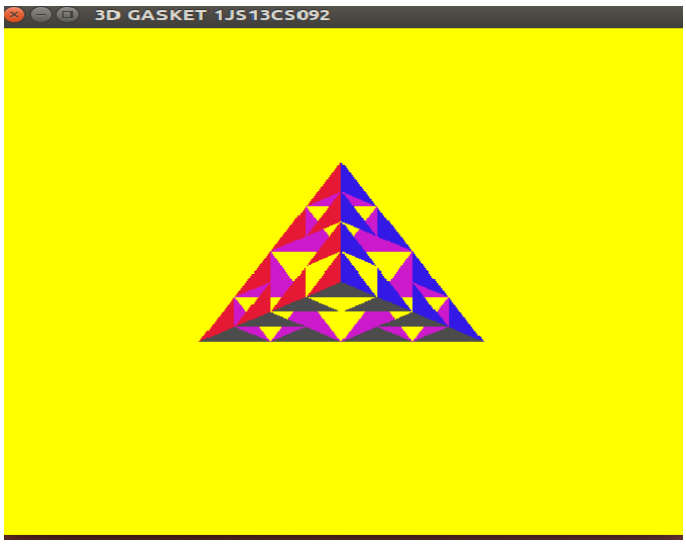
```
}
void display()
{
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glBegin(GL_TRIANGLES);
dt(v[0],v[1],v[2],v[3],m);
glEnd();
glFlush();
}
void init()
{
glClearColor(0,0,0,1);
glOrtho(-2,2,-2,2,-14,10);
}
int main(int argc, char **argv)
{
printf("enter the no. of steps\n");
scanf("%d",&m);
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_DEPTH);
glutInitWindowSize(700,700);
glutCreateWindow("3d Gasket");
init();
glutDisplayFunc(display);
glEnable(GL_DEPTH_TEST);
glutMainLoop();
}
```

Output command

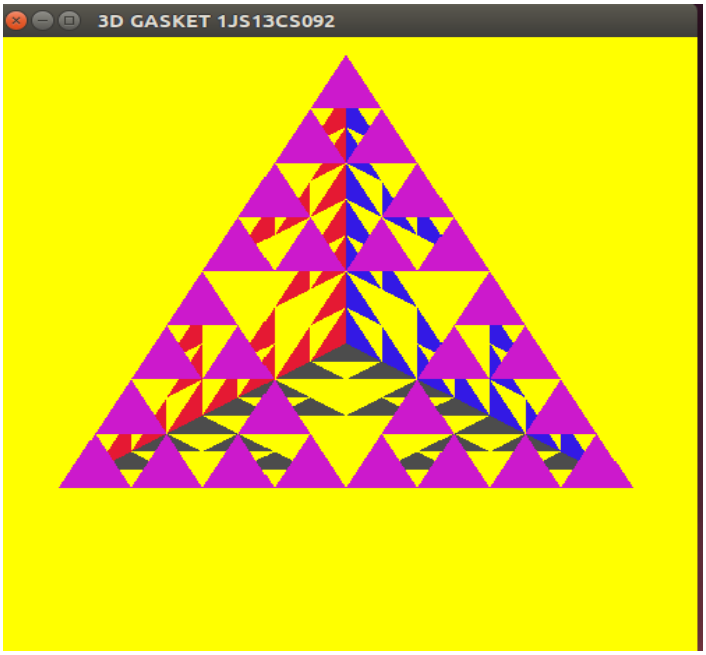
To create file - gedit filename.c

To compile file - gcc filename.c -lGL -lGLU -lglut

To execute - ./a.out



Sierpinski gasket (front view)



Sierpinski gasket (rear view)

PROGRAM -8

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

```
#include<stdio.h>
#include<stdlib.h>
#include<GL/glut.h>
#include<math.h>
#include<iostream>
# define PI 3.1416
GLsizei w=600,h=600;
GLfloat xmin=0.0,xmax=120.0;
GLfloat ymin=0.0,ymax=120.0;
#define wave 1
#define stop 2
#define quit 3
class w3d {
public:
GLfloat x,y,z;
};

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

void compute(GLfloat u,w3d *bp,GLint ncp,w3d *cp,GLint *c)
{
GLint k,n=ncp-1;
GLfloat bbf;
bp->x=bp->y=bp->z=0.00;
for(k=0;k<ncp;k++)
{
bbf=c[k]*pow(u,k)*pow(1-u,n-k);
bp->x+=cp[k].x*bbf;
bp->y+=cp[k].y*bbf;
```

```
bp->z+=cp[k].z*bbf;
}
}

void bezier(w3d *cp,GLint ncp,GLint nbcp)
{
w3d bcp;
GLfloat u;
GLint *c,k;
c=new GLint[ncp];
bino(ncp-1,c);
glBegin(GL_LINE_STRIP);
for(k=0;k<=nbcp;k++)
{
u=(GLfloat)k/(GLfloat)nbcp;
compute(u,&bcp,ncp,cp,c);
glVertex2f(bcp.x,bcp.y);
}
glEnd();
delete [ ] c;
}
static float t=0;
void display()
{
GLint i, ncp=4,nbcp=20;

w3d cp[4]={
{20,100,0},
{30,110,0},
{50,90,0},
 {60,100,0}};

 cp[1].x+=10*sin(t*PI/180.0);
 cp[1].y+=5*sin(t*PI/180.0);
 cp[2].x-=10*sin((t+30)*PI/180.0);
 cp[2].y-=10*sin((t+30)*PI/180.0);
 cp[3].x-=4*sin((t)*PI/180.0);
 cp[3].y+=sin((t-30)*PI/180.0);
 glClear(GL_COLOR_BUFFER_BIT);
 glColor3f(1.0,1.0,1.0);
 glPointSize(5);
 glPushMatrix();
 glLineWidth(5);
 glColor3f(255/255,153/255.0,51/255.0);
 for(i=0;i<8;i++)
```

```
{
glTranslatef(0,-0.8,0);
bezier(cp,ncp,nbcp);
}
glColor3f(1,1,1);
for(i=0;i<8;i++)
{
glTranslatef(0,-0.8,0);
bezier(cp,ncp,nbcp);
}
glColor3f(19/255.0,136/255.0,8/255.0);
for(i=0;i<8;i++)
{
glTranslatef(0,-0.8,0);
bezier(cp,ncp,nbcp);
}
glPopMatrix();
glColor3f(0.7,0.5,0.3);
glLineWidth(5);
glBegin(GL_LINES);
glVertex2f(20,100);
glVertex2f(20,40);
glEnd();
glFlush();
glutPostRedisplay();
glutSwapBuffers();
}

void res(GLint nW,GLint nH)
{
glViewport(0,0,nW,nH);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(xmin,xmax,ymin,ymax);
glMatrixMode(GL_MODELVIEW);
glutPostRedisplay();
}

void animate()
{
t+=5;
glutPostRedisplay();
}

void menu(int item)
```

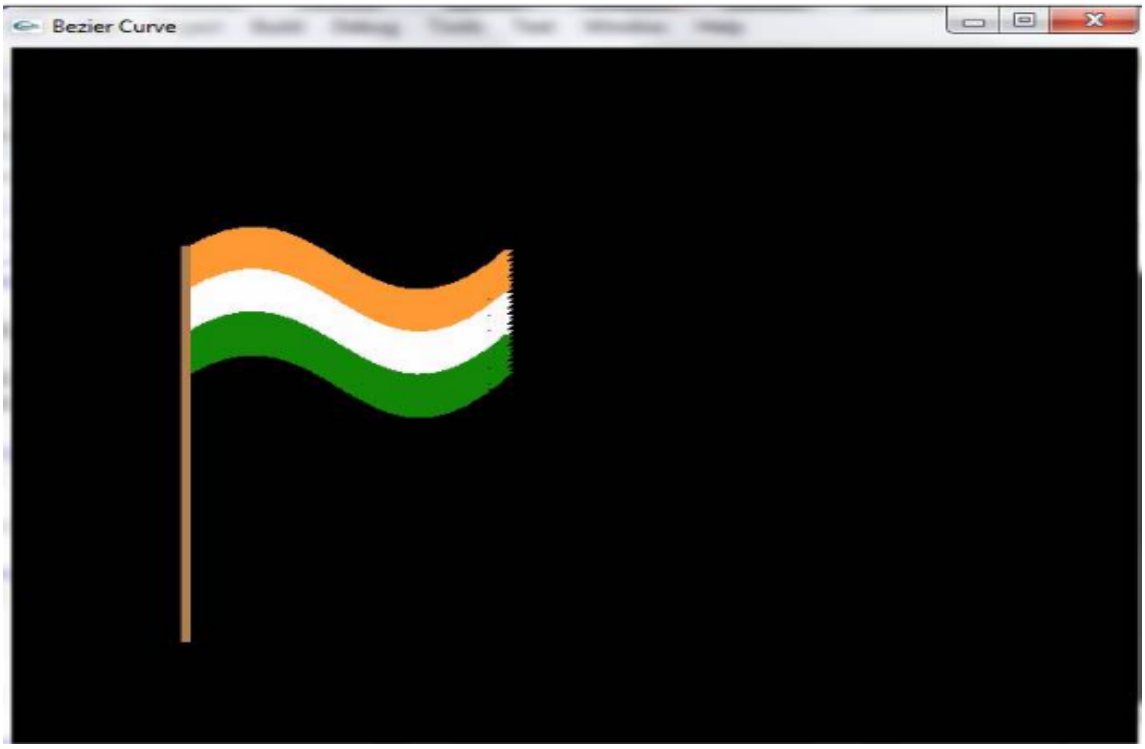
```
{
switch(item)
{
case wave:
glutIdleFunc(animate);
break;
case stop:
glutIdleFunc(NULL);
break;
case quit:
exit(0);
break;
}
}

int main(int argc,char** argv)
{
glutInit(&argc,argv);
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB);
glutInitWindowPosition(50,50);
glutInitWindowSize(w,h);
glutCreateWindow("flag animation");
glutDisplayFunc(display);
glutReshapeFunc(res);
glutCreateMenu(menu);
glutAddMenuEntry("flag waiving",wave);
glutAddMenuEntry("stop waiving",stop);
glutAddMenuEntry("quit",quit);
glutAttachMenu(GLUT_RIGHT_BUTTON);
glutMainLoop();
}
```

Output command

To create file - gedit filename.c

To compile file - gcc filename.c -lGL -lGLU -lglut

To execute - ./a.out



Animating a flag using Bezier Curve algorithm

PROGRAM -9

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

```
#include<GL/glut.h>
#include<stdlib.h>
#include<stdio.h>
GLfloat x1,y1,x2,y2,x3,y3,x4,y4;
int sb;
int n=30;
int r=0,b=0,g=1;

void cm(int id)
{
switch(id)
{
case 2: r=1;
        g=0;
        b=0;
break;

case 3: r=0;
        g=0;
        b=1;
break;

case 4: r=0;
        g=1;
        b=0;
}
glutPostRedisplay();
}

void tm(int id)
{
switch(id)
{
case 1: exit(1);
break;

default: cm(id);
break;
}
}
```

```
void ed(GLfloat x1,GLfloat y1,GLfloat x2,GLfloat y2,int *le,int *re)
{
float t,mx,x;
int i;
if((y2-y1)<0)
{
t=x1;
x1=x2;
x2=t;
t=y1;
y1=y2;
y2=t;
}

if((y2-y1)!=0)
mx=(x2-x1)/(y2-y1);
else
mx=x2-x1;
x=x1;
for(i=y1+1;i<y2;i++)
{
if(x<le[i])
le[i]=x;
if(x>re[i])
re[i]=x;
x+=mx;
}
}

void dp(int x, int y)
{
glColor3f(r,g,b);
glBegin(GL_POINTS);
glVertex2i(x,y);
glEnd();
}

void scf(float x1, float y1, float x2, float y2, float x3, float y3, float x4, float y4)
{
int le[500],re[500];
int i,y;
for(i=0;i<500;i++)
{
le[i]=500;
```

```
re[i]=0;
}
ed(x1,y1,x2,y2,le,re);
ed(x2,y2,x3,y3,le,re);
ed(x3,y3,x4,y4,le,re);
ed(x4,y4,x1,y1,le,re);
for(y=0;y<500;y++)
{
if(le[y]<=re[y])
for(i=le[y]+1;i<re[y];i++)
dp(i,y);
}
}
void display()
{
glClear(GL_COLOR_BUFFER_BIT);
int x1=200,y1=200,x2=100,y2=300,x3=200,y3=400,x4=300,y4=300;
glBegin(GL_LINE_LOOP);
glVertex2i(x1,y1);
glVertex2i(x2,y2);
glVertex2i(x3,y3);
glVertex2i(x4,y4);
glEnd();
scf(x1,y1,x2,y2,x3,y3,x4,y4);
glFlush();
}

void minit()
{
glClearColor(1,1,1,1);
glColor3f(r,g,b);
glPointSize(1);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0,499,0,499);
}

int main(int argc, char **argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
glutInitWindowSize(1500,1500);
glutCreateWindow("SFA");
glutDisplayFunc(display);
minit();
```

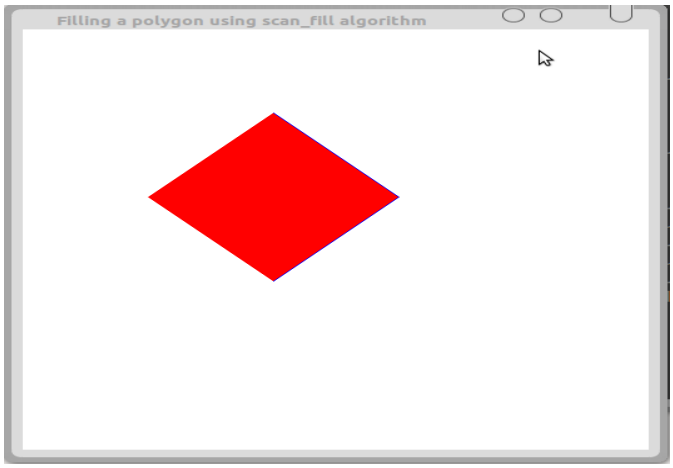
```
sb=glutCreateMenu(cm);
glutAddMenuEntry("RED",2);
glutAddMenuEntry("BLUE",3);
glutAddMenuEntry("GREEN",4);
glutCreateMenu(tm);
glutAddMenuEntry("QUIT",1);
glutAddSubMenu("COLOR",sb);
glutAttachMenu(GLUT_RIGHT_BUTTON);

glutMainLoop();
}
```

Output -

To create file - gedit filename.c

To compile file - gcc filename.c -lGL -lGLU -lglut

To execute - ./a.out



Scanline fill algorithm

Viva Questions and Answers

1. Define Computer graphics?

Computer graphicsare graphicscreated by computersand, more generally, the representationand manipulationof pictorialdataby a computer

2. Define Computer Animation?

Computer animationis the art of creating moving images via the use of computers. It is a subfield of computer graphicsand animation

3. Define Pixel?

Theword pixelis based on a contraction of pix("pictures") and el(for "element"). Pixels are normally arranged in a 2-dimensional grid, and are often represented using dots, squares, or rectangles

4. Define Raster graphics?

Raster Graphics, which is the representation of images as an array of pixels, as it is typically used for the representation of photographic images.

5. Define Rendering?

Renderingis the process of generating an image from a model, by means of computer programs.

6. Define Ray Tracing?

Ray tracingis a technique for generating an imageby tracing the path of lightthrough pixelsin an image plane.

7. Define Projection?

Projectionis a method of mapping points to a plane.

8. Define 3D Projection?

3D projectionis a method of mapping three dimensional points to a two dimensional plane.

9. What is OpenGL?

OpenGL(R) is the software interface for graphics hardware that allows graphics programmers to produce high-quality color images of 3D objects. OpenGL is a rendering only, vendor neutral API providing 2D and 3D graphics functions, including modeling, transformations, color, lighting, smooth shading, as well as advanced features like texture mapping, NURBS, fog, alpha blending and motion blur. OpenGL works in both immediate and retained (display list) graphics modes.OpenGL is window system and operating system independent. OpenGL has been integrated with Windows NT and with the X Window System under UNIX. Also, OpenGL is

network transparent. A defined common extension to the X Window System allows an OpenGL client on one vendor's platform to run across a network to another vendor's OpenGL server.

10. What are the benefits of OpenGL for hardware and software developers?□

Industry standard,□Reliable, portable, Evolving, Scalable, Easy to use.

11. What is the GLUT Toolkit?

GLUT is a portable toolkit which performs window and event operations to support OpenGL rendering

12. How does the camera work in OpenGL?

As far as OpenGL is concerned, there is no camera. More specifically, the camera is always located at the eye space coordinate (0.0, 0.0, 0.0). To give the appearance of moving the camera, your OpenGL application must move the scene with the inverse of the camera transformation by placing it on the MODELVIEW matrix. This is commonly referred to as the viewing transformation. In practice this is mathematically equivalent to a camera transformation but more efficient because model transformations and camera transformations are concatenated to a single matrix. As a result though, certain operations must be performed when the camera and only the camera is on the MODELVIEW matrix. For example to position a light source in worldspace it most be positioned while the viewing transformation and only the viewing transformation is applied to the MODELVIEW matrix.

13. Define Perspective Projection?

Perspective(from Latinperspicere, to see through) in the graphic arts, such as drawing, isan approximate representation, on a flat surface (such as paper), of an image as it is perceived by the eye. The two most characteristic features of perspective are that objects are drawn.

14. Define World CoordinateSystems.

World CoordinateSystems (WCS) are any coordinatesystems that describe the physical coordinateassociated with a data array, such as sky coordinates.

15. What does the glClearColor() do?

- glClearColor() establishes what color the window will be cleared to, and glClear() actually clears the window.
- Once the clearing color is set, the window is cleared to that color whenever glClear() is called.

16. What does the glClearColor() do?

- glClearColor() establishes what color the window will be cleared to, and glClear() actually clears the window.
- Once the clearing color is set, the window is cleared to that color whenever glClear() is called.

17. What is persistence?

The time it takes the emitted light from the screen to decay one tenth of its original intensity is called as persistence.

18. What is Aspect ratio?

The ratio of vertical points to the horizontal points necessary to produce length of lines in both directions of the screen is called the Aspect ratio. Usually the aspect ratio is ¾.

19. Name two techniques for producing colour displays with a CRT?

Beam penetration method, shadow mask method.

20. What is scan conversion?

A major task of the display processor is digitizing a picture definition given in an application program into a set of pixel-intensity values for storage in the frame buffer. This digitization process is called scan conversion.

21. What is rasterization?

The process of determining the appropriate pixels for representing picture or graphics object is known as rasterization

22. What is a Depth buffer?

- The depth buffer stores a depth value for each pixel.
- The depth buffer is also called as the z buffer
- It is usually measured in terms of distance to the eye.
- The depth buffer's behavior can be modified as described in "Depth Test."

23. What is GL_DEPTH_TEST?

- The depth buffer is generally used for hidden-surface elimination. If a new candidate color for that pixel appears, it's drawn only if the corresponding object is closer than the previous object. Therefore only objects that aren't obscured by other items remain.
- Initially, the clearing value for the depth buffer is a value that's as far from the viewpoint as possible, so the depth of any object is nearer than that value comes previous value is overwritten with the new value which is closer to the eye.
- If this is how you want to use the depth buffer, you simply have to enable it by passing GL_DEPTH_TEST to glEnable().