VTU QUESTION PAPER SOLUTION

UNIT -1
INTRODUCTION

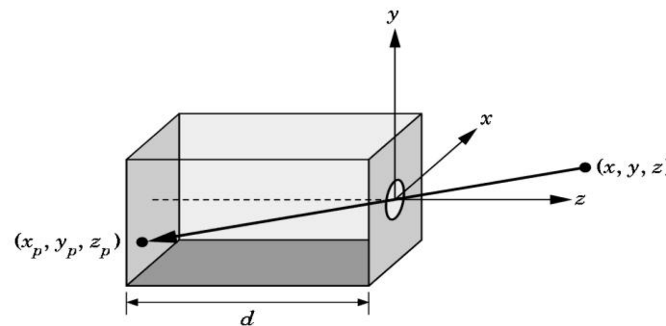
1. Briefly explain any two applications of computer graphics. (June 2012) 4M

Ans: Applications of computer graphics are:

- Display Of Information
- Design
- Simulation & Animation
- User Interfaces

2. Explain the concept of pinhole camera of an imaging system. Also derive the expression for angle of view. (June 2012) 6M

Ans :



Use trigonometry to find projection of point at (x,y,z)

$xp= -x/z/d \quad yp= -y/z/d \quad zp= d$

These are equations of simple perspective

3. Discuss the graphics pipeline architecture, with the help of a functional schematic diagram. (June 2012) 10M

Ans : Graphics Pipeline :

- Process objects one at a time in the order they are generated by the application
- All steps can be implemented in hardware on the graphics card

Vertex Processor

- Much of the work in the pipeline is in converting object representations from one coordinate system to another
  - Object coordinates
  - Camera (eye) coordinates
  - Screen coordinates
- Every change of coordinates is equivalent to a matrix transformation
- Vertex processor also computes vertex colors

Primitive Assembly

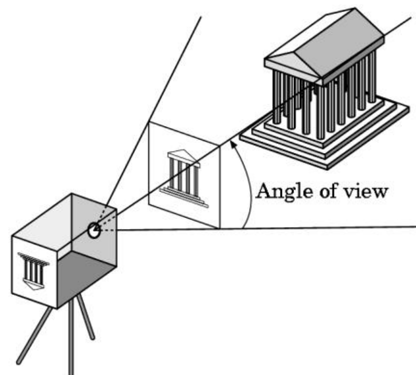
Vertices must be collected into geometric objects before clipping and rasterization can take place

- Line segments
- Polygons
- Curves and surfaces

Clipping

Just as a real camera cannot “see” the whole world, the virtual camera can only see part of the world or object space

- Objects that are not within this volume are said to be *clipped* out of the scene



Rasterization :

- If an object is not clipped out, the appropriate pixels in the frame buffer must be assigned colors
- Rasterizer produces a set of fragments for each object
- Fragments are “potential pixels”
  - Have a location in frame bufffer
  - Color and depth attributes
- Vertex attributes are interpolated over objects by the rasterizer

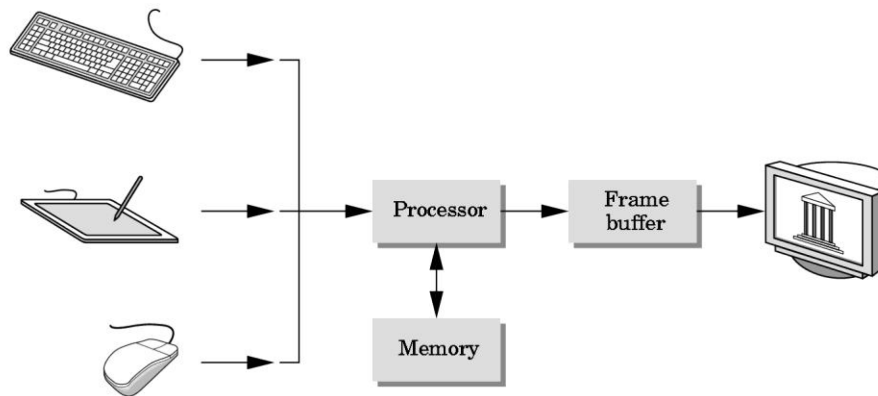
Fragment Processor:

- Fragments are processed to determine the color of the corresponding pixel in the frame buffer
- Colors can be determined by texture mapping or interpolation of vertex colors
- Fragments may be blocked by other fragments closer to the camera

4. With a neat diagram, explain the components of a graphics system. (Dec 2011) 6M

Ans : A Graphics system has 5 main elements :

- Input Devices
- Processor
- Memory
- Frame Buffer
- Output Devices



Pixels and the Frame Buffer

- A picture is produced as an array (raster) of picture elements (pixels).
- These pixels are collectively stored in the Frame Buffer.

Properties of frame buffer:

Resolution – number of pixels in the frame buffer

Depth or Precision – number of bits used for each pixel

E.g.: 1 bit deep frame buffer allows 2 colors

8 bit deep frame buffer allows 256 colors.

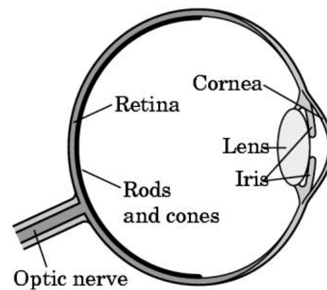
A Frame buffer is implemented either with special types of memory chips or it can be a part of system memory.

In simple systems the CPU does both normal and graphical processing.

Graphics processing - Take specifications of graphical primitives from application program and assign values to the pixels in the frame buffer It is also known as Rasterization or scan conversion.

5. With a neat diagram, explain the human visual system. (Dec 2011) 6M

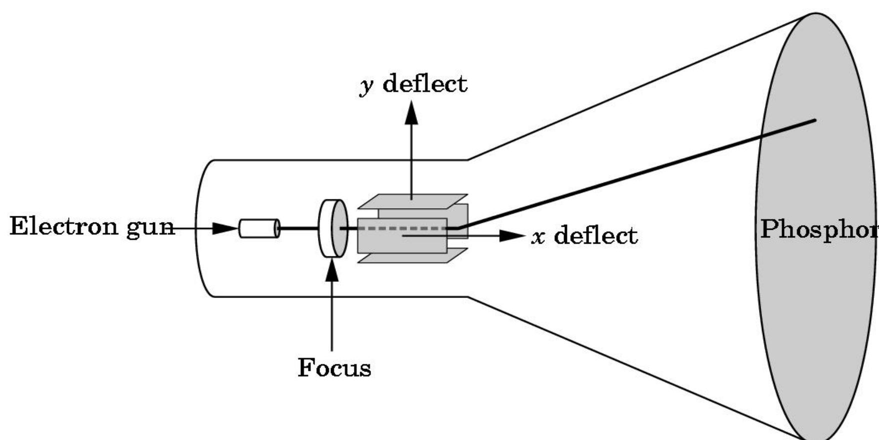
Ans:



- Rods are used for : monochromatic, night vision
- Cones
    - Color sensitive
    - Three types of cones
    - Only three values (the *tristimulus* values) are sent to the brain
- Need only match these three values
    - Need only three *primary* colors

6. Describe the working of an output device with an example. (July2011) 5M

Ans : The most predominant type of display has been the Cathode Ray Tube (CRT).



Various parts of a CRT:

- Electron Gun – emits electron beam which strikes the phosphor coating to emit light.

- Deflection Plates – controls the direction of beam. The output of the computer is converted by digital-to-analog converters o voltages across x & y deflection plates.
- Refresh Rate – In order to view a flicker free image, the image on the screen has to be retraced by the beam at a high rate (modern systems operate at 85Hz)

2 types of refresh:

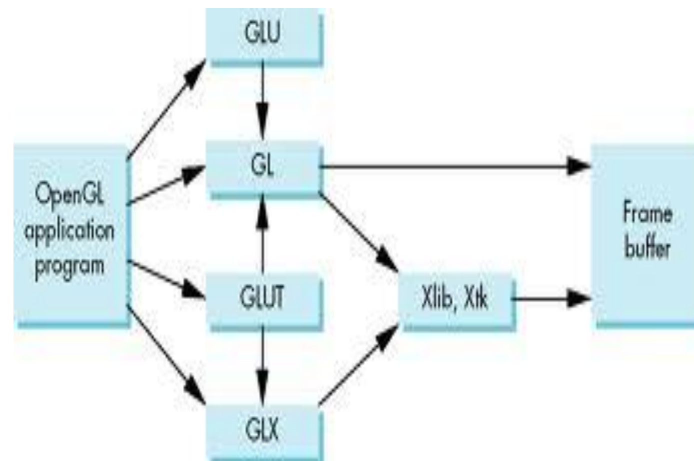
- Noninterlaced display: Pixels are displayed row by row at the refresh rate.
- Interlaced display: Odd rows and even rows are refreshed alternately.

UNIT -2

THE OPENGL

1. With the help of a diagram, describe the open GL interface. (Jun2012) 4M

Ans: OpenGL provides a powerful but primitive set of rendering commands, and all higher-level drawing must be done in terms of these commands.



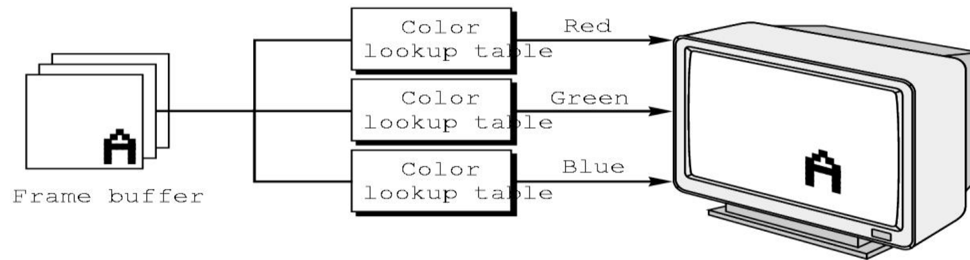
- The OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections, performing polygon tessellation, and rendering surfaces. This library is provided as part of every OpenGL implementation.
- For every window system, there is a library that extends the functionality of that window system to support OpenGL rendering. For machines that use the X Window System, the OpenGL Extension to the X Window System (GLX) is provided as an adjunct to OpenGL. GLX routines use the prefix **glX**. For Microsoft Windows, the WGL routines provide the Windows to OpenGL interface.
- The OpenGL Utility Toolkit (GLUT) is a window system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window system APIs.

2. Write explanatory notes on: i) RGB color model; ii) indexed color model. (Jun2012) 6M

Ans: Colors are indices into tables of RGB values

- Requires less memory
  - indices usually 8 bits
  - not as important now
    - Memory inexpensive

- Need more colors for shading



In indexed mode, colors are stored as indices. If there are k indices then there can be kn-1 colors that could be got by combining red, green and blue. This yields a huge color palette as compared to the normal RGB mode.

3. Write an open GL recursive program for 2D sierpinski gasket with relevant comments. (Jun2012) 10M

Ans:
```
#include "stdafx.h"
#include <GL/glut.h>

typedef float point[3];

/* initial tetrahedron */

point v[] ={{0.0, 0.0, 1.0},
       {0.0, 1.0, -1.0},
       {-1.0, -1.0, -1.0},
       {1.0, -1.0, -1.0}};

static GLfloat theta[] = {0.0,0.0,0.0};

int n;

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

/* display one triangle using a line loop for wire frame */
{
   glBegin(GL_POLYGON);
     glVertex3fv(a);
     glVertex3fv(b);
     glVertex3fv(c);
```

```
   glEnd();
}

void divide_triangle(point a, point b, point c, int m)
{

/* triangle subdivision using vertex numbers
  righthand rule applied to create outward pointing faces */

   point v1, v2, v3;
   int j;
   if(m>0)
         {
      for(j=0; j<3; j++) v1[j]=(a[j]+b[j])/2;
      for(j=0; j<3; j++) v2[j]=(a[j]+c[j])/2;
      for(j=0; j<3; j++) v3[j]=(b[j]+c[j])/2;
      divide_triangle(a, v1, v2, m-1);
      divide_triangle(c, v2, v3, m-1);
      divide_triangle(b, v3, v1, m-1);
   }
   /* draw triangle at end of recursion */
   else(triangle(a,b,c));
}

void tetrahedron( int m)
{

/* Apply triangle subdivision to faces of tetrahedron.Give a
different color to each face of the tetrahedron*/

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

void myReshape(int w, int h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   /* code to maintain the aspect ratio*/
   /* When width becomes less
         than height, adjust the bottom,top parameters to
     maintain the aspect ratio*/
         if (w <= h)
       glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
         2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
   /* When height becomes less
         than width, adjust the left,right parameters to
     maintain the aspect ratio*/
         else
       glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
         2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0,
         -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
   glutPostRedisplay();
}

void main(int argc, char **argv)
{
   printf("enter the no of division : ");
   scanf("%d",&n);
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(640, 480);
   glutCreateWindow("3D Gasket");
   glutReshapeFunc(myReshape);
```

```
    glutDisplayFunc(display);
        glEnable(GL_DEPTH_TEST);
    glClearColor (1.0, 1.0, 1.0, 1.0);
    glutMainLoop();
}
```

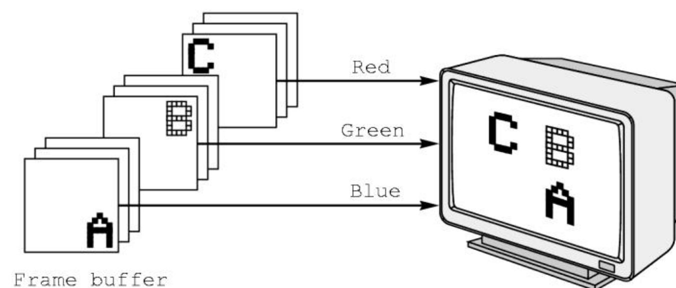
4. With a neat diagram, discuss the color formation. Explain the additive and subtractive colors, indexed color and color solid concept. (Dec2011) 12M

Ans:

- A visible color can be characterized by the function C(λ)
- Tristimulus values – responses of the 3 types of cones to the colors.
- 3 color theory – "If 2 colors produce the same tristimulus values, then they are visually indistinguishable."
- Additive color model – Adding together the primary colors to get the percieved colors. E.g. CRT.
- Subtractive color model – Colored pigments remove color components from light that is striking the surface. Here the primaries are the complimentary colors : cyan, magenta and yellow

RGB color

- Each color component is stored separately in the frame buffer
- Usually 8 bits per component in buffer
- Note in **glColor3f** the color values range from 0.0 (none) to 1.0 (all), whereas in **glColor3ub** the values range from 0 to 255



The color as set by **glColor** becomes part of the state and will be used until changed

- Colors and other attributes are not part of the object but are assigned when the object is rendered

- We can create conceptual *vertex colors* by code such as

  glColor

  glVertex

  glColor

  glVertex

RGBA color system :

- This has 4 arguments – RGB and alpha

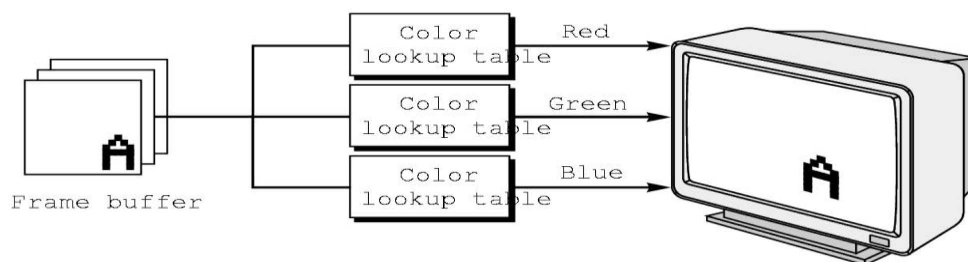
alpha – Opacity.

glClearColor(1.0,1.0,1.0,1.0)

This would render the window white since all components are equal to 1.0, and is opaque as alpha is also set to 1.0

Indexed color

- Colors are indices into tables of RGB values
- Requires less memory
  - indices usually 8 bits
  - not as important now
    - Memory inexpensive
    - Need more colors for shading



5. What are control functions? Explain with examples. (Dec2011) 8M

Ans: Window – A rectangular area of our display.

Modern systems allow many windows to be displayed on the screen (multiwindow environment).

The position of the window is with reference to the origin. The origin (0, 0) is the top left corner of the screen.

glutInit()

allows application to get command line arguments and initializes system.The function is basically used for initializing the glut library and also to initiate a session with the windows system. The function does not take any arguments and should be the first function to be called within the main program.

gluInitDisplayMode() requests properties for the window (the *rendering context*)

RGB color- specified by the argument GLUT_RGB. It specifies that a 3 color mode needs to be used.

Single buffering – GLUT_SINGLE: specifies that the images are static and only a single frame buffer is required to store the pixels

GLUT_DOUBLE: specifies that the images are animations and two frame buffers,front and back are required for rendering a smooth image.

Properties logically ORed together

glutWindowSize in pixels

glutWindowPosition from top-left corner of display

glutCreateWindow create window with a particular title

6. Write a complete open GL program for creating 3D sierpinski gasket by subdivision of a tetrahedron. (July2011) 10M

Ans:
```
#include "stdafx.h"
#include <GL/glut.h>

typedef float point[3];

/* initial tetrahedron */

point v[] ={{0.0, 0.0, 1.0},
        {0.0, 1.0, -1.0},
        {-1.0, -1.0, -1.0},
        {1.0, -1.0, -1.0}};

static GLfloat theta[] = {0.0,0.0,0.0};

int n;

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

```
/* display one triangle using a line loop for wire frame */
{
   glBegin(GL_POLYGON);
     glVertex3fv(a);
     glVertex3fv(b);
     glVertex3fv(c);
   glEnd();
}

void divide_triangle(point a, point b, point c, int m)
{

/* triangle subdivision using vertex numbers
  righthand rule applied to create outward pointing faces */

   point v1, v2, v3;
   int j;
   if(m>0)
         {
      for(j=0; j<3; j++) v1[j]=(a[j]+b[j])/2;
      for(j=0; j<3; j++) v2[j]=(a[j]+c[j])/2;
      for(j=0; j<3; j++) v3[j]=(b[j]+c[j])/2;
      divide_triangle(a, v1, v2, m-1);
      divide_triangle(c, v2, v3, m-1);
      divide_triangle(b, v3, v1, m-1);
   }
   /* draw triangle at end of recursion */
   else(triangle(a,b,c));
}

void tetrahedron( int m)
{

/* Apply triangle subdivision to faces of tetrahedron.Give a
different color to each face of the tetrahedron*/

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

void myReshape(int w, int h)
{
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    /* code to maintain the aspect ratio*/
    /* When width becomes less
        than height, adjust the bottom,top parameters to
      maintain the aspect ratio*/
        if (w <= h)
      glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
        2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
    /* When height becomes less
        than width, adjust the left,right parameters to
      maintain the aspect ratio*/
        else
      glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
        2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0,
        -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glutPostRedisplay();
}

void main(int argc, char **argv)
{
    printf("enter the no of division : ");
```

```
    scanf("%d",&n);
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(640, 480);
    glutCreateWindow("3D Gasket");
    glutReshapeFunc(myReshape);
    glutDisplayFunc(display);
        glEnable(GL_DEPTH_TEST);
    glClearColor (1.0, 1.0, 1.0, 1.0);
    glutMainLoop();
}
```

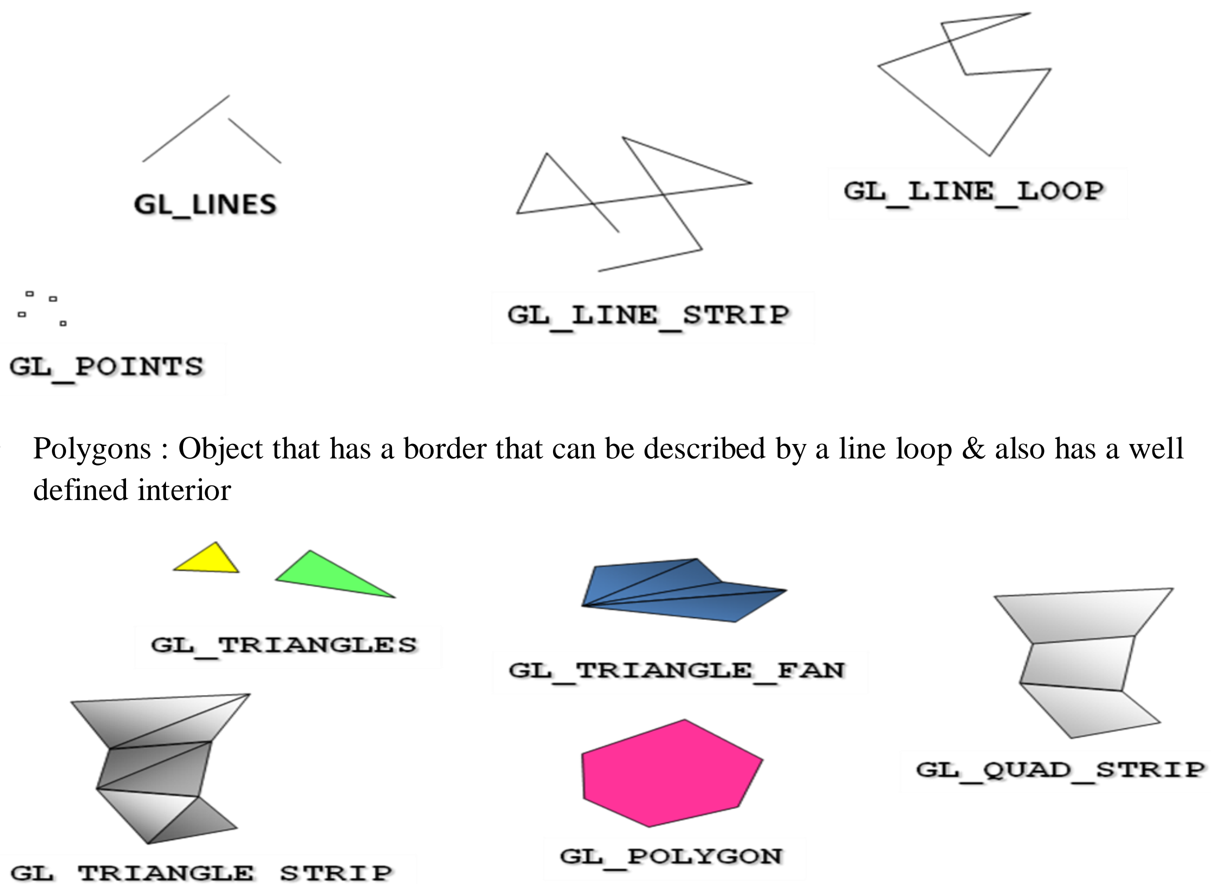
7. Classify the major groups of API functions in open GL. Explain any four of them. (July2011) 6M

Ans:

- Primitive functions: Defines low level objects such as points, line segments, polygons etc.
- Attribute functions : Attributes determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Polygon mode
    - Display as filled
    - Display edges
    - Display vertices
- Viewing functions: Allows us to specify various views by describing the camera's position and orientation.
- Transformation functions: Provides user to carry out transformation of objects like rotation, scaling etc.
- Input functions : Allows us to deal with a diverse set of input devices like keyboard, mouse etc
- Control functions: Enables us to initialize our programs, helps in dealing with any errors during execution of the program.
- Query functions: Helps query information about the properties of the particular implementation.

8. What is an attribute with respect to graphics system? List attributes for lines and polygons. (July2011) 4M

- **Ans:** Attribute functions : Attributes determine the appearance of objects
  - Color (points, lines, polygons)
  - Size and width (points, lines)
  - Polygon mode
    - Display as filled
    - Display edges
    - Display vertices



- Polygons : Object that has a border that can be described by a line loop & also has a well defined interior

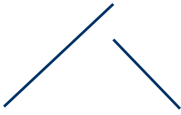
9. List out different open GL primitives, giving examples for each. (Jan2010) 10M

Ans: OpenGL supports 2 types of primitives:

- Geometric primitives (vertices, line segments.) – they pass through the geometric pipeline

- Raster primitives (arrays of pixels) – passes through a separate pipeline to the frame buffer.

Line segments



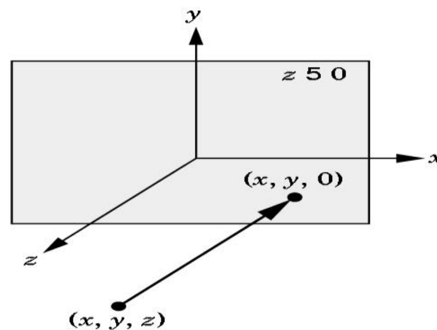
GL_LINES

GL_LINE_STRIP

GL_LINE_LOOP

10. Briefly explain the orthographic viewing with OpenGL functions for 2d and 3d viewing. Indicate the significance of projection plane and viewing point in this. (Jan2010) 10M

Ans: In the default orthographic view, points are projected forward along the z axis onto the plane $z=0$



Transformations and Viewing

- The pipeline architecture depends on multiplying together a number of transformation matrices to achieve the desired image of a primitive.
- Two important matrices :
    - Model-view
    - Projection
- The values of these matrices are part of the state of the system.

In OpenGL, projection is carried out by a projection matrix (transformation)

There is only one set of transformation functions so we must set the matrix mode first

glMatrixMode (GL_PROJECTION)

Transformation functions are incremental so we start with an identity matrix and alter it with a projection matrix that gives the view volume

glLoadIdentity();

glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);

UNIT-3

INPUT AND INTERACTION

1. What are the various classes of logical input devices that are supported by open GL? Explain the functionality of each of these classes. (Jun2012) 8M

Ans : Consider the C and C++ code

- C++: **cin >> x;**
- C: **scanf (“%d”, &x);**

- What is the input device?
  - Can’t tell from the code
  - Could be keyboard, file, output from another program
- The code provides *logical input*
  - A number (an **int**) is returned to the program regardless of the physical device

Graphical Logical Devices

- Graphical input is more varied than input to standard programs which is usually numbers, characters, or bits
- Two older APIs (GKS, PHIGS) defined six types of logical input
  - **Locator**: return a position. Placing the mouse pointer at any location on the screen would return the corresponding x and y coordinates of the location. Mouse acts as a locator device.

  - **Pick**: return ID of an object. When there are several graphical objects on the screen picking one of them and that would occupy the entire screen. This is a pick operation.
    Again the mouse can act as apick device.

  - **Keyboard**: return strings of characters. Typing on the keyboard would keep storing the typed letters into a buffer and on evnets like pressing on enter key, the entire string would be returned to the graphics system.Keyboard itself is the device for this purpose.

  - **Stroke**: return array of positions. Basically used in paint applications where when a paint brush is moved across the editor a stroke is generated. All the locations involved in the stroke are returned as an array. Mouse can act as a stroke device.

  - **Valuator**: return floating point number.

– **Choice**: return one of n items. When there are several items on the screen then selecting one of them is the purpose of this device. It could be selected by a mouse click which returns the id associated with a particular object.

2. **List the various features that a good interactive program should include. (Jun2012) 4M**

Ans: Some of the good features of a interactive graphics program are:

- User friendly GUI
- Having help menus
- Easily understandable
- Providing smooth transitions of images.
- Smooth 3d animations by using z buffer.

3. **Write an open GL program to demonstrate the hierarchical means, to draw a rectangle and to increase or decrease the size of rectangle. (Jun2012) 8M**

Ans: The open GL program is as follows:

```
glutCreateMenu(demo_menu);
glutAddMenuEntry("quit",1);
glutAddMenuEntry("increase square size",2);
glutAddMenuEntry("decrease square size", 3);
GlutAttachMenu(GLUT_RIGHT_BUTTON);
```

- The callback function is:

```
void demo_menu(int id)
{
    switch(id)
    {
        case 1: exit(0);
        break;
        case 2: size=2*size;
        break;
        case 3: if(size>1) size=size/2;
        break;
    }
glutPostRedisplay();
```

```
}

sub_menu = glutCreateMenu(size_menu);
glutAddMenuEntry("increase square size", 2);
glutAddMenuEntry("Decrease square size",3);
glutCreateMenu(top_menu);
glutAddMenuEntry("Quit",1);
glutAddSubMenu("Resize",sub_menu);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

4. Which are the six classes of logical input devices? Explain. (Dec2011) 6M

Ans:

- **Locator**: return a position. Placing the mouse pointer at any location on the screen would return the corresponding x and y coordinates of the location. Mouse acts as a locator device.

- **Pick**: return ID of an object. When there are several graphical objects on the screen picking one of them and that would occupy the entire screen. This is a pick operation.
  Again the mouse can act as apick device.

- **Keyboard**: return strings of characters. Typing on the keyboard would keep storing the typed letters into a buffer and on evnets like pressing on enter key, the entire string would be returned to the graphics system.Keyboard itself is the device for this purpose.

- **Stroke**: return array of positions. Basically used in paint applications where when a paint brush is moved across the editor a stroke is generated. All the locations involved in the stroke are returned as an array. Mouse can act as a stroke device.

- **Valuator**: return floating point number.

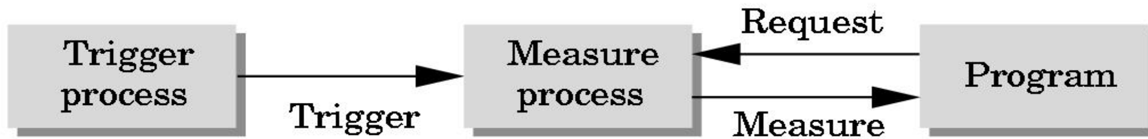
- **Choice**: return one of n items. When there are several items on the screen then selecting one of them is the purpose of this device. It could be selected by a mouse click which returns the id associated with a particula object.

5. Discuss the request mode, sample mode and event modes with the figures wherever required. (Dec2011) 8M

Ans: Request Mode:

- The measure of the device is not returned to program until device is triggered

- Standard in nongraphical applications
- Typical of keyboard input
    - Can erase (backspace), edit, correct until enter (return) key (the trigger) is depressed

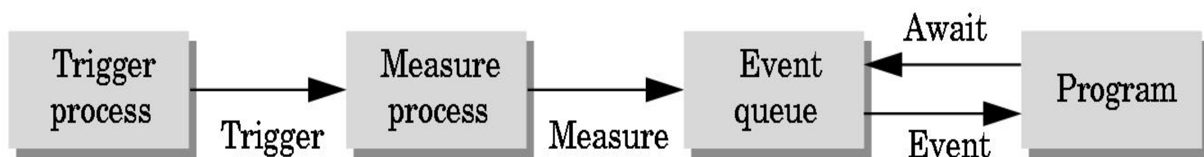


Sample Mode:

- As soon as the function call in the user program is encountered, the measure is returned.
- No trigger is needed
- The user must have positioned the pointing device before the function call, because the measure is extracted immediately from the buffer.
- Both request and sample mode are useful where program guides the user.

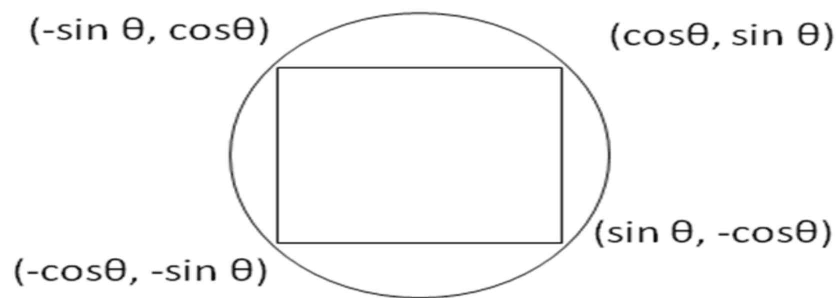
Event Mode:

- Most systems have more than one input device, each of which can be triggered at an arbitrary time by a user
- Each trigger generates an *event* whose measure is put in an *event queue* which can be examined by the user program



6. Write a program to draw a rotating square. (Dec2011) 6M

Ans:



- The points x=cos θ, y=sin θ always lies on a unit circle regardless of the value of θ.
- In order to increase θ by a fixed amount whenever nothing is happening, we use the idle function

```
Void idle()
{
        theta+ =2;
   If (theta >=360.0) theta - = 360.0;
        glutPostRedisplay();
}
```

- In order to turn the rotation feature on and off, we can include a mouse function as follows :

```
Void mouse(int button, int state, int x, int y)
{
        if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        glutIdleFunc(idle);
        if (button == GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
        glutIdleFunc(NULL);
}
```

7. **Suppose that the openGL window is 500 X 50 pixels and the clipping window is a unit square with the origin at the lower left corner. Use simple XOR mode to draw erasable lines. (Jul2011) 10M**

Ans:

```
void MouseMove(int x,int y)
{

if(FLAG == 0){
X = x;
Y = winh - y;

Xn = x;
Yn = winh - y;

FLAG = 1;
}
```

```
else if(FLAG == 1){
glEnable(GL_COLOR_LOGIC_OP);
glLogicOp(GL_XOR);

glBegin(GL_LINES);
glVertex2i(X,Y);
glVertex2i(Xn,Yn);
glEnd();
glFlush();/*Old line erased*/

glBegin(GL_LINES);
glVertex2i(X,Y);
glVertex2i(x, winh - y);
glEnd();
glFlush();

Xn = x;
Yn = winh - y;
}
}
```

8. **What is the functionality of display lists in modeling? Explain with an example. (Jul2011) 5M**

Ans : Display lists help in avoiding redundant code by storing the compiled code in a buffer and then re executing it again and again.

- **For eg:**The most efficient way of defining text is to define the font once, using a display list for each char, and then store the font on the server using these display lists
- A function to draw ASCII characters

```
void OurFont(char c)
{
switch(c)
{
        case 'O' :
        glTranslatef(0.5,0.5,0.0); /* move to the center */
        glBegin(GL_QUAD_STRIP)
                for (i=0;i<12;i++) /* 12 vertices */
                {
                        angle = 3.14159/6.0 * i; /* 30 degrees in radians */
```

```
                    glVertex2f(0.4 * cos(angle)+0.5, 0.4 * sin(angle)+0.5)
                    glVertex2f(0.5 * cos(angle)+0.5, 0.5 * sin(angle)+0.5)
                }
            glEnd();
    break;
    }

    }
```

9. Explain Picking operation in openGL with an example. (Jul2011) 5M

Ans : Identify a user-defined object on the display

In principle, it should be simple because the mouse gives the position and we should be able to determine to which object(s) a position corresponds

Practical difficulties

- Pipeline architecture is feed forward, hard to go from screen back to world
- Complicated by screen being 2D, world is 3D
- How close do we have to come to object to say we selected it?

```
void mouse (int button, int state, int x, int y)
{
        GLUint nameBuffer[SIZE];
        GLint hits;
        GLint viewport[4];
        if (button == GLUT_LEFT_BUTTON && state== GLUT_DOWN)
        {
                /* initialize the name stack */
                glInitNames();
                glPushName(0);
                glSelectBuffer(SIZE, nameBuffer)l

                /* set up viewing for selection mode */

                glGetIntegerv(GL_VIEWPORT, viewport); //gets the current viewport
                glMatrixMode(GL_PROJECTION);
                /* save original viewing matrix */
                glPushMatrix();
                glLoadIdentity();
                /* N X N pick area around cursor */
                gluPickMatrix( (GLdouble) x,(GLdouble)(viewport[3]-y),N,N,viewport);
```

```
                    /* same clipping window as in reshape callback */
                    gluOrtho2D(xmin,xmax,ymin,ymax);

                    draw_objects(GL_SELECT);
                    glMatrixMode(GL_PROJECTION);

                    /* restore viewing matrix */
                    glPopMatrix();
                    glFlush();

                    /* return back to normal render mode */

                    hits = glRenderMode(GL_RENDER);

                    /* process hits from selection mode rendering*/

                    processHits(hits, nameBuff);

                    /* normal render */
                    glutPostRedisplay();
            }
    }
    void draw_objects(GLenum mode)
    {
            if (mode == GL_SELECT)
                    glLoadName(1);
            glColor3f(1.0,0.0,0.0)
            glRectf(-0.5,-0.5,1.0,1.0);

            if (mode == GL_SELECT)
                            glLoadName(2);
            glColor3f(0.0,0.0,1.0)
            glRectf(-1.0,-1.0,0.5,0.5);
    }
void processHits(GLint hits, GLUint buffer[])
    {
            unsigned int i,j;

    }
```

UNIT – 4

GEOMETRIC OBJECTS AND TRANSFORMATIONS-1

1. Explain the complete procedure of converting a world object frame into camera or eye frame, using the model view matrix. (Jun2012) 10M

Ans: World Space

Object space for a particular object gives it no spatial relationship with respect to other objects. The purpose of *world space* is to provide some absolute reference for all the objects in your scene. How a world-space coordinate system is established is arbitrary. For example, you may decide that the origin of world space is the center of your room. Objects in the room are then positioned relative to the center of the room and some notion of scale (Is a unit of distance a foot or a meter?) and some notion of orientation (Does the positive *y*-axis point "up"? Is north in the direction of the positive *x*-axis?).

The Modeling Transform

The way an object, specified in object space, is positioned within world space is by means of a modeling transform. For example, you may need to rotate, translate, and scale the 3D model of a chair so that the chair is placed properly within your room's world-space coordinate system. Two chairs in the same room may use the same 3D chair model but have different modeling transforms, so that each chair exists at a distinct location in the room.

You can mathematically represent all the transforms in this chapter as a 4x4 matrix. Using the properties of matrices, you can combine several translations, rotations, scales, and projections into a single 4x4 matrix by multiplying them together. When you concatenate matrices in this way, the combined matrix also represents the combination of the respective transforms. This turns out to be very powerful, as you will see.

If you multiply the 4x4 matrix representing the modeling transform by the object-space position in homogeneous form (assuming a 1 for the *w* component if there is no explicit *w* component), the result is the same position transformed into world space. This same matrix math principle applies to all subsequent transforms discussed in this chapter.

Eye Space

Ultimately, you want to look at your scene from a particular viewpoint (the "eye"). In the coordinate system known as *eye space*(or *view space*), the eye is located at the origin of the coordinate system. Following the standard convention, you orient the scene so the eye is looking down one direction of the *z*-axis. The "up" direction is typically the positive *y* direction.

Eye space, which is particularly useful for lighting, will be discussed in Chapter 5.

The View Transform

The transform that converts world-space positions to eye-space positions is the *view transform*. Once again, you express the view transform with a 4x4 matrix.

The typical view transform combines a translation that moves the eye position in world space to the origin of eye space and then rotates the eye appropriately. By doing this, the view transform defines the position and orientation of the viewpoint.

The Modelview Matrix

Most lighting and other shading computations involve quantities such as positions and surface normals. In general, these computations tend to be more efficient when performed in either eye space or object space. World space is useful in your application for establishing the overall spatial relationships between objects in a scene, but it is not particularly efficient for lighting and other shading computations.

For this reason, we typically combine the two matrices that represent the modeling and view transforms into a single matrix known as the *modelview matrix*. You can combine the two matrices by simply multiplying the view matrix by the modeling matrix.

2. With respect to modeling discuss vertex arrays. (Jun2012) 5M

Ans :

i) Vertex arrays provide a method for encapsulating the information in data structure such that we can draw polyhedral objects with only few function calls.

There are three steps in using vertex arrays

(i) Enable the functionality of vertex arrays

(ii) Tell openGL, location & format of the array.

(iii) Render the object.

The first two steps are called initialization part and the third step is called display callback.

OpenGL allows many different types of arrays; here we are using two such arrays called color and vertex arrays. The arrays can be enabled as follows.

```
glEnableClientstate (GL_COLOR_ARRAY)
glEnableClientstate (GL_VERTEX_ARRAY).
```

The arrays are same as before. Next, we identify where the arrays are as follows.

```
glVertexPointer (3,GL_FLOAT, 0, Vertices);
glColorPointer (3,GL_FLOAT, 0, COLOR);
```

3. Explain modeling a color cube in detail. (Dec2011) 10M

Ans: We can use the vertex list to define a color cube. We can define a function quad to draw quadrilaterals polygons specified by pointers into the vertex list. The color cube specifies the six faces, taking care to make them all outward facing as follows.

```
GLfloatVertices [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 color [8] [3] = {{0.0, 0.0, 0.0}, {1.0, 0.0, 0.0}, {1.0, 1.0, 0.0}, {0.0, 1.0, 0.0}, {0.0, 0.0,
1.0}, {1.0, 0.0, 1.0}, {1.0, 1.0, 1.0}, {0.0, 1.0, 1.0}};


void quad (int a, int b, int c, int d)
{
        glBegin (GL_QUADS);
                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 ()
        {       quad (0,3,2,1);
                quad (2,3,7,6);
                quad (0, 4,7,3);
                quad (1, 2, 6, 5);
                quad (4, 5, 6, 7);
                quad (0, 1, 5, 4);
```

4. Explain affine transformations. (Dec2011) 10M

Ans: An affine transformation is an important class of linear 2-D geometric transformations which

maps variables (*e.g.* pixel intensity values located at position (x_1, y_1) in an input image) into new variables (*e.g.* (x_2, y_2) in an output image) by applying a linear combination of translation, rotation, scaling and/or shearing (*i.e.* non-uniform scaling in some directions) operations.

The general affine transformation is commonly written in homogeneous coordinates as shown below:

$$\begin{vmatrix} x_2 \\ y_2 \end{vmatrix} = A \times \begin{vmatrix} x_1 \\ y_1 \end{vmatrix} + B$$

By defining only the *B* matrix, this transformation can carry out pure translation:

$$A = \begin{vmatrix} 1 & 0 \\ 0 & 1 \end{vmatrix}, B = \begin{vmatrix} b_1 \\ b_2 \end{vmatrix}$$

Pure rotation uses the *A* matrix and is defined as (for positive angles being clockwise rotations):

$$A = \begin{vmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{vmatrix}, B = \begin{vmatrix} 0 \\ 0 \end{vmatrix}$$

Here, we are working in image coordinates, so the y axis goes downward. Rotation formula can be defined for when the y axis goes upward.

Similarly, pure scaling is:

$$A = \begin{vmatrix} a_{11} & 0 \\ 0 & a_{22} \end{vmatrix}, B = \begin{vmatrix} 0 \\ 0 \end{vmatrix}$$

(Note that several different affine transformations are often combined to produce a resultant transformation. The order in which the transformations occur is significant since a translation followed by a rotation is not necessarily equivalent to the converse.)

Since the general affine transformation is defined by 6 constants, it is possible to define this transformation by specifying the new output image locations (x_2, y_2) of any three input image coordinate (x_1, y_1) pairs. (In practice, many more points are measured and a least squares method is used to find the best fitting transform.)

5. **In a homogenous coordinate system given two frames (v_1, v_2, v_3, P_0) and (u_1, u_2, u_3, Q_0). Let a and b be two vectors defined in two frames respectively. Derive the expression that represents vector b interms of a. (July2011) 10M**

   Ans: Need to know

- The origin (or displacement vector)
- The basis vectors - The direction and distance for +1 movement along each axis

This definition is relative

To plot a point

- Begin at origin
- Travel along the x basis vector [direction] scaled by x coord, then along the y basis vector scaled by the y coord, then finally along the z basis vector scaled by the z coord.

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\begin{matrix} 1 * x & + & 0 * y & + & dx * 1 \\ 0 * x & + & 1 * y & + & dy * 1 \\ 0 * x & + & 0 * y & + & 1 * 1 \end{matrix}$$

UNIT – 5

GEOMETRIC OBJECTS AND TRANSFORMATIONS-II

1. Define and represent the following 2-D transformations in homogenous coordinate system.

   a. Translation

   b. Rotation

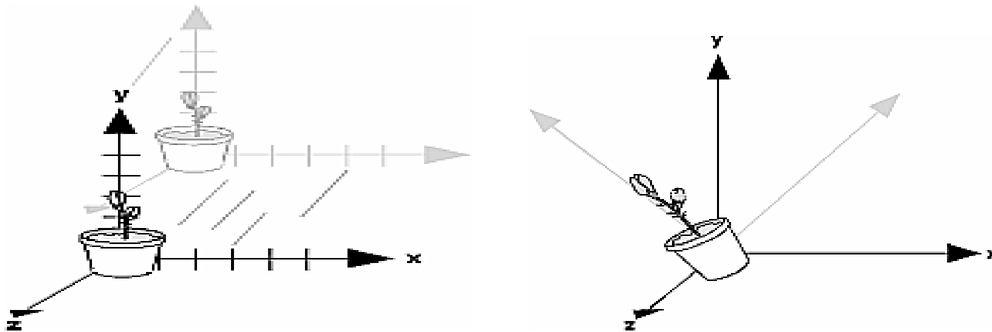
   c. Scaling

   d. Reflection (Jun2012) 12M

Ans: **Translation**

void glTranslate{fd} (TYPE x, TYPE y, TYPE z);

Multiplies the current matrix by a matrix that moves (translates) an object by the given x, y, and z values

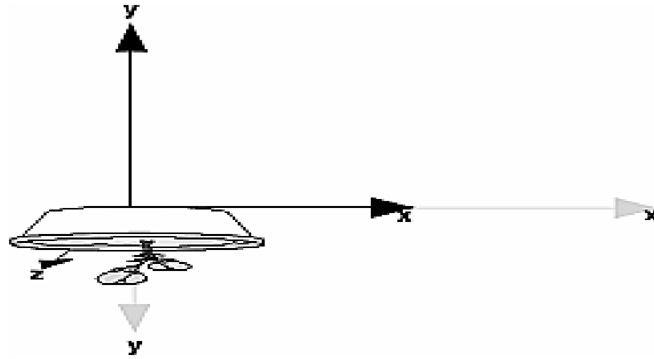


Rotation

- void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);
- Multiplies the current matrix by a matrix that rotates an object 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.

Scaling

- void glScale{fd} (TYPEx, TYPE y, TYPEz);
- Multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axes.



Equations :

- Translation: Pf = T + P

  $x_f = x_o + dx$

  $y_f = y_o + dy$

- Rotation: Pf = R · P

  $x_f = x_o * \cos\alpha - y_o * \sin\alpha$

  $y_f = x_o * \sin\alpha + y_o * \cos\alpha$

- Scale: Pf = S · P

  $x_f = sx * x_o$

  $y_f = sy * y_o$

2. What is concatenation transformation? Explain rotation about a fixed point. (Jun2012) 8M

Ans: Rotate a house about the origin

- Rotate the house about one of its corners
  - translate so that a corner of the house is at the origin
  - rotate the house about the origin
  - translate so that the corner returns to its original position

3. What are quaternions? With an example, explain its mathematical representations. (Dec2011) 10M

Ans: A quaternion is an element of a 4 dimensional vector-space. It's defined as w + xi + yj + zk where i, j and k are imaginary numbers. Alternatively, a quaternion is what you get when you add a scalar and a 3d vector. The math behind quaternions is only slightly harder than the math behind vectors.

```
void Camera::movex(float xmmod)
{
  pos += rotation * Vector3(xmmod, 0.0f, 0.0f);
}

void Camera::movey(float ymmod)
{
  pos.y -= ymmod;
}

void Camera::movez(float zmmod)
{
  pos += rotation * Vector3(0.0f, 0.0f, -zmmod);
}

void Camera::rotatex(float xrmod)
{
  Quaternion nrot(Vector3(1.0f, 0.0f, 0.0f), xrmod * PIOVER180);
  rotation = rotation * nrot;
}

void Camera::rotatey(float yrmod)
{
  Quaternion nrot(Vector3(0.0f, 1.0f, 0.0f), yrmod * PIOVER180);
  rotation = nrot * rotation;
}

void Camera::tick(float seconds)
{
  if (xrot != 0.0f) rotatex(xrot * seconds * rotspeed);
  if (yrot != 0.0f) rotatey(yrot * seconds * rotspeed);

  if (xmov != 0.0f) movex(xmov * seconds * movespeed);
  if (ymov != 0.0f) movey(ymov * seconds * movespeed);
  if (zmov != 0.0f) movez(zmov * seconds * movespeed);
}
```

4. Explain the basic transformations in 3D and represent them in matrix form. (Jun2010) 10M

Ans: Translation:

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scaling:

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Rotation:

$$\begin{bmatrix} Cos\alpha & -sin\alpha & 0 \\ sin\alpha & cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

5. What are the advantages of quaternion? (Jun2010) 2M

Ans: Quaternions have some advantages over other representations of rotations.

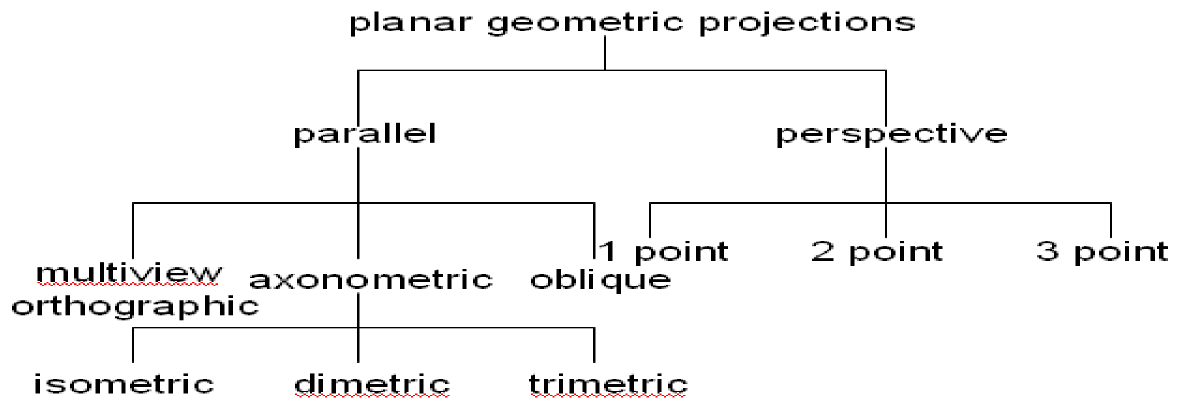
- Quaternions don't suffer from gimbal lock, unlike Euler angles.
- They can be represented as 4 numbers, in contrast to the 9 numbers of a rotations matrix.
- The conversion to and from axis/angle representation is trivial.
- Smooth interpolation between two quaternions is easy (in contrast to axis/angle or rotation matrices).
- After a lot of calculations on quaternions and matrices, rounding errors accumulate, so you have to normalize quaternions and orthogonalize a rotation matrix, but normalizing a quaternion is a lot less troublesome than orthogonalizing a matrix.
- Similar to rotation matrices, you can just multiply 2 quaternions together to receive a quaternion that represents both rotations.

UNIT -6

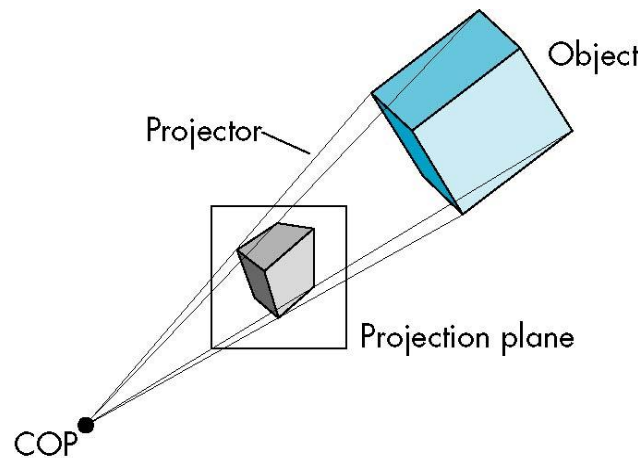
VIEWING

1. **With neat sketches, explain the various types of views that are employed in computer graphics systems. (Jun2012) 10M**

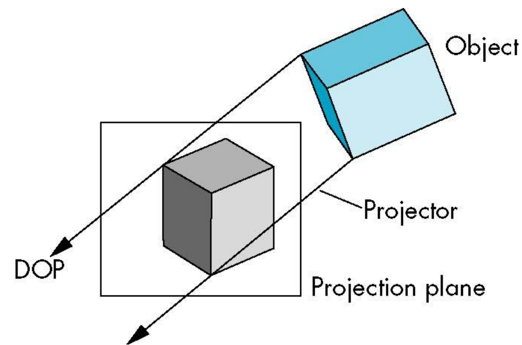
Ans:



Perspective and parallel projections :
Parallel viewing is a limiting case of perspective viewing
Perspective projection has a COP where all the projector lines converge.

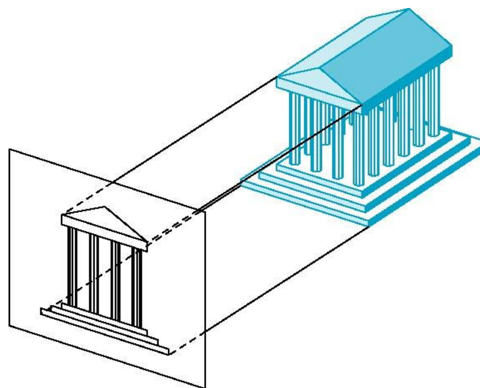


Parallel projection has parallel projectors. Here the viewer is assumed to be present at infinity. So here we have a "Direction of projection (DOP)" instead of center of projection(COP).



Orthographic Projections :

- Projectors are perpendicular to the projection plane.
- Projection plane is kept parallel to one of the principal faces.
- A viewer needs more than 2 views to visualize what an object looks like from its multiview orthographic projection.

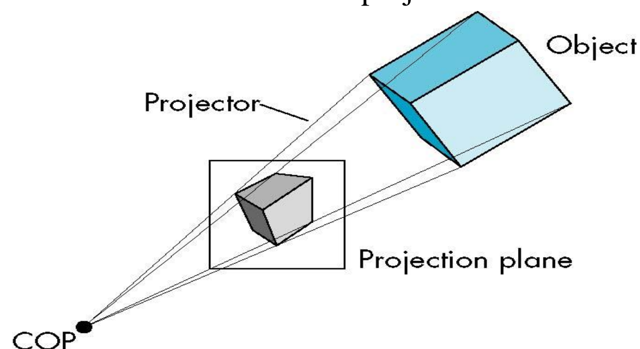


2. Briefly discuss the following along with the functions used for the purpose in open GL

i) Perspective projections

ii) Orthogonal projections (Jun2012) 10M

Ans: i) Perspective projection has a COP where all the projector lines converge.

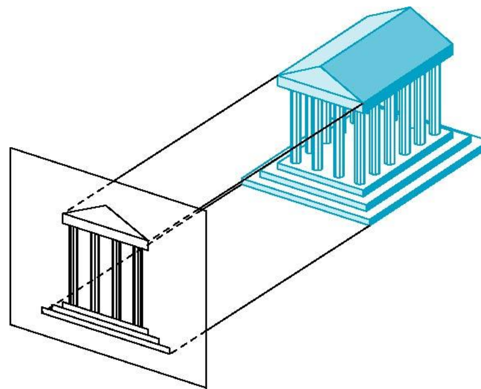


The following functions have to be used to create perspective projections:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(fFOV, fAspect , fNearPlane, fFarPlane);
```

Orthogonal projections

Projectors are perpendicular to the projection plane.
Projection plane is kept parallel to one of the principal faces.
A viewer needs more than 2 views to visualize what an object looks like from its multiview orthographic projection.



```
void glOrtho(   GLdouble    left,
    GLdouble    right,
    GLdouble    bottom,
    GLdouble    top,
    GLdouble    nearVal,
    GLdouble    farVal);
```

Parameters

left, right
    Specify the coordinates for the left and right vertical clipping planes.

bottom, top
    Specify the coordinates for the bottom and top horizontal clipping planes.

nearVal, farVal
    Specify the distances to the nearer and farther depth clipping planes.
    These values are negative if the plane is to be behind the viewer.

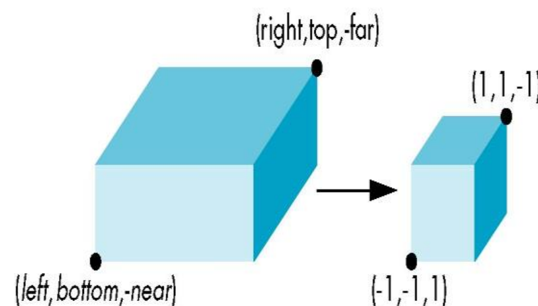
3. Explain the various types of axonometric projections. (Dec2011) 7M

Ans: Projectors are orthogonal to the projection plane , but projection plane can move relative to object.

- Classification by how many angles of a corner of a projected cube are the same
  - none: trimetric
  - two: dimetric
  - three: isometric

4. What is canonical view volume? Explain the mapping of a given view volume to the canonical form. (Dec2011) 7M

Ans: The diagram shows the normalization process



First the view volume specified by the glortho function is mapped to the canonical form
Canonical Form : default view volume centerd at the origin and having sides of length 2.

This involves 2 steps :

– Move center to origin

T(-(left+right)/2, -(bottom+top)/2,(near+far)/2))

– Scale to have sides of length 2

S(2/(left-right),2/(top-bottom),2/(near-far))

The resultant matrix is a product of the above 2 matrices i.e. P = ST =

$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right-left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{2}{near-far} & \frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

5. Derive equations for perspective projection and describe the specifications of a perspective camera view in open GL. (Jun2011) 8M

Ans.: The new coordinate system is specified by a translation and rotation
with respect to the old coordinate system:
v´= R (v - v0) v0 is displacement vector
R is rotation matrix
R may be decomposed into
3 rotations about the
coordinate axes:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

By multiplying the 3 matrices Rx, Ry and Rz, one gets

$$R = \begin{bmatrix} \cos\beta\cos\gamma & \cos\beta\sin\gamma & -\sin\beta \\ \sin\alpha\sin\beta\cos\gamma - \cos\alpha\sin\gamma & \sin\alpha\sin\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\cos\beta \\ \cos\alpha\sin\beta\cos\gamma + \sin\alpha\sin\gamma & \cos\alpha\sin\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\cos\beta \end{bmatrix}$$

For formula manipulations, one tries to avoid the trigonometric functions and takes

$$R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$$

Note that the coefficients of R are constrained:
A rotation matrix is orthonormal: R RT = I (unit matrix)

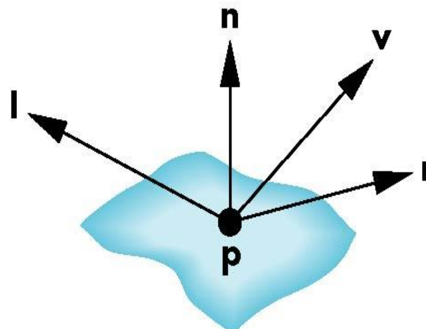
UNIT – 7

LIGHTING AND SHADING

1. Explain phong lighting model. Indicate the advantages and disadvantages. (Jun2012) 10M

Ans: Phong developed a simple model that can be computed rapidly

- It considers three components
  - Diffuse
  - Specular
  - Ambient
- And Uses four vectors
  - To source represented by the vector l
  - To viewer represented by the vector v
  - Normal represented by the vector n
  - Perfect reflector represented by the vector r



We need 9 coefficients to characterize the light source with ambient, diffuse and specular components.The Illumination array for the ith light source is given by the matrix:

$$L_i = \begin{matrix} L_{ira} & L_{iga} & L_{iba} \\ L_{ird} & L_{igd} & L_{ibd} \\ L_{irs} & L_{igs} & L_{ibs} \end{matrix}$$

The intensity for each color source can be computed by adding the ambient,specular and diffuse components.

- E.g. Red intensity that we see from source I:

$I_{ir} = R_{ira}L_{ira} + R_{ird}L_{ird} + R_{irs}L_{irs} = I_{ra}+I_{rd}+I_{rs}$

Since the necessary computations are same for each light source,

$I = I_a+I_d+I_s$

2. What are the different methods available for shading a polygon? Briefly discuss any 2 of them. (Jun2012) 10M

Ans: Polygonal Shading

Flat shading

In case of flat shading there are distinct boundaries after color interpolation

3 vectors needed for shading are: l,n,v .The openGL function to enable flat shading is :

glShadeModel(GL_FLAT)

For a flat polygon,n is constant as the normal **n** is same at all points on the polygon.Also if we assume a distant viewer, the vector **v** is constant and if we consider a distant light source then the vector l is also a constant.Here all the 3 vectors are constant and therefore the shading calculations needs to be done only once for an entire polygon and each point on the polygon is assigned the same shade. This technique is known as Flat shading.

Disadvantage : But if we consider light sources and the viewer near the polygon, then flat shading will show differences in shading and the human eye is very sensitive to slightest of such differences due to the principle of “Lateral Inhibition”

Smooth and Gouraud Shading

Gouraud shading, also called intensity interpolation, provides a way to display smooth-shaded polygons by defining the RGB color components of each polygon vertex. It operates by first interpolating the RGB values between the vertical vertices along each edge. This gives us the RGB components for the left and right edges of each scan line (pixel row). We then display each row of pixels by horizontally interpolating the RGB values between that row's left and right edges. This produces a remarkably smooth-shaded polygon. Fastgraph supports Gouraud shading for direct color virtual buffers, but not for 256-color virtual buffers.

3. Write a program segment using structures to represent meshes of quadrilateral and shade them. (Dec2011) 7M

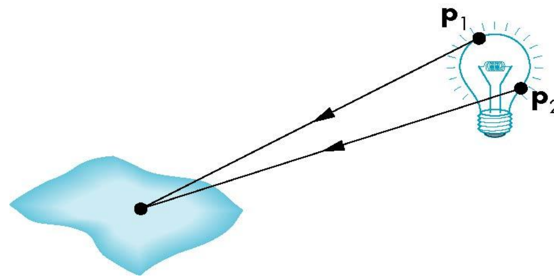
Ans:
```
#include <stdlib.h>
#include <GL/glut.h>

#define maxx 20
#define maxy 25
#define dx 15
#define dy 10

GLfloat x[maxx]={0.0},y[maxy]={0.0};
GLfloat x0=50,y0=50;  // initial values for x, y
GLint i,j;
void init()
```

```
{
   glClearColor(1.0,1.0,1.0,1.0);
        glColor3f(1.0,0.0,0.0);
        glPointSize(5.0);
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0,499.0,0.0,499.0);
        glutPostRedisplay();            // request redisplay
}

void display(void)
{
        /* clear window */
         glClear(GL_COLOR_BUFFER_BIT);
         glColor3f(0.0, 0.0, 1.0);                    // set color to blue
        /* draw rectangles */
         for(i=0;i<maxx;i++)
                 x[i]=x0+i*dx;    // compute x[i]
                for(j=0;j<maxy;j++)
                        y[j]=y0+j*dy;  // compute y[i]

                glColor3f(0.0, 0.0, 1.0);
                for(i=0;i<maxx-1;i++)
                for(j=0;j<maxy-1;j++)
                {
                        glColor3f(0.0, 0.0, 1.0);
                        glBegin(GL_LINE_LOOP);
                                glVertex2f(x[i],y[j]);
                                glVertex2f(x[i],y[j+1]);
                                glVertex2f(x[i+1],y[j+1]);
                                glVertex2f(x[i+1],y[j]);
                        glEnd();
                        glFlush();
                }
                glFlush();
}

void main(int argc, char** argv)
{
        glutInit(&argc, argv); // OpenGL initializations
```

```
        glutInitDisplayMode(GLUT_SINGLE                    |                    GLUT_RGB
        glutInitWindowSize(500, 400); // create a 500x400 window
        glutInitWindowPosition(0, 0); // ...in the upper left
        glutCreateWindow("Rectangular Mesh"); // create the window
        glutDisplayFunc(display); // setup callbacks
        init();
        glutMainLoop(); // start it running
}
```

4. Describe any two types of light sources. (Jul2011) 4M

Ans: General light sources are difficult to work with because we must integrate light coming from all points on the source.



- Point source
  - Model with position and color
  - Distant source = infinite distance away (parallel)
- Spotlight
  - Restrict light from ideal point source
- Ambient light
  - Same amount of light everywhere in scene
  - Can model contribution of many sources and reflecting surfaces

5. How is approximation of a sphere done by recursive subdivision? (Dec2011) 10M

Ans:

```
/* Recursive subdivision of tetrahedron (Chapter 6). Three display
modes: wire frame, constant, and interpolative shading */

/*Program also illustrates defining materials and light sources
in myiit() */

/* mode 0 = wire frame, mode 1 = constant shading,
mode 2 = interpolative shading */

/* Updated October 16, 2000, by Cary Laxer to convert I/O to C++. */

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

typedef double point[3];

/* initial tetrahedron */

point v[]={{0.0, 0.0, 1.0}, {0.0, 0.942809, -0.33333},
          {-0.816497, -0.471405, -0.333333}, {0.816497, -0.471405, -
0.333333}};

static GLfloat theta[] = {0.0,0.0,0.0};

int n;
int mode;

void triangle (point a, point b, point c);
void normal (point p);
void divide_triangle (point a, point b, point c, int m);
void tetrahedron (int m);
void display (void);
void myReshape (int w, int h);
void myinit ();

using namespace std;

void main(int argc, char **argv)
{
    cout << "Enter number of levels of recursion: ";
    cin >> n;
    glutInit (&argc, argv);
    glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("Sphere");
    myinit ();
    glutReshapeFunc (myReshape);
    glutDisplayFunc (display);
    glutMainLoop ();
}

void triangle (point a, point b, point c)
/* display one triangle using a line loop for wire frame, a single
normal for constant shading, or three normals for interpolative shading */
```

```
{
    if (mode==0) glBegin(GL_LINE_LOOP);
    else glBegin(GL_POLYGON);
                if(mode==1 || mode==2) glNormal3dv(a);
                glVertex3dv(a);
                if(mode==2) glNormal3dv(b);
                glVertex3dv(b);
                if(mode==2) glNormal3dv(c);
                glVertex3dv(c);
    glEnd();
}

void normal (point p)
/* normalize a vector */
{
    double d =0.0;
    int i;
    for(i=0; i<3; i++) d+=p[i]*p[i];
    d=sqrt(d);
    if(d>0.0) for(i=0; i<3; i++) p[i]/=d;
}

void divide_triangle (point a, point b, point c, int m)
/* triangle subdivision using vertex numbers
righthand rule applied to create outward pointing faces */
{
    point v1, v2, v3;
    int j;
    if(m>0)
    {
        for(j=0; j<3; j++) v1[j]=a[j]+b[j];
        normal(v1);
        for(j=0; j<3; j++) v2[j]=a[j]+c[j];
        normal(v2);
        for(j=0; j<3; j++) v3[j]=b[j]+c[j];
        normal(v3);
        divide_triangle  (a, v1, v2, m-1);
        divide_triangle (c, v2, v3, m-1);
        divide_triangle (b, v3, v1, m-1);
        divide_triangle (v1, v3, v2, m-1);
    }
    else
                triangle (a,b,c); /* draw triangle at end of recursion */
}

void tetrahedron (int m)
/* Apply triangle subdivision to faces of tetrahedron */
{
    divide_triangle (v[0], v[1], v[2], m);
    divide_triangle (v[3], v[2], v[1], m);
    divide_triangle (v[0], v[3], v[1], m);
    divide_triangle (v[0], v[2], v[3], m);
}

void display (void)
/* Displays all three modes, side by side */
{
```

```
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
        glLoadIdentity ();
    mode=0;
    tetrahedron (n);
    mode=1;
    glTranslated (-2.0, 0.0, 0.0);
    tetrahedron (n);
    mode=2;
    glTranslated (4.0, 0.0, 0.0);
    tetrahedron (n);
    glutSwapBuffers ();
}

void myReshape (int w, int h)
{
    glViewport (0, 0, w, h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    if (w <= h)
        glOrtho (-4.0, 4.0, -4.0 * (GLfloat) h / (GLfloat) w,
            4.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
    else
        glOrtho (-4.0 * (GLfloat) w / (GLfloat) h,
            4.0 * (GLfloat) w / (GLfloat) h, -4.0, 4.0, -10.0, 10.0);
    glMatrixMode (GL_MODELVIEW);
    glutPostRedisplay ();
}
void myinit ()
{
    GLfloat mat_specular[]={0.5, 0.0, 0.0, 1.0};
    GLfloat mat_diffuse[]={1.0, 0.0, 0.0, 1.0};
    GLfloat mat_ambient[]={1.0, 0.0, 0.0, 1.0};
    GLfloat mat_shininess={100.0};
    GLfloat light_ambient[]={0.0, 0.0, 0.0, 1.0};
    GLfloat light_diffuse[]={1.0, 1.0, 1.0, 1.0};
    GLfloat light_specular[]={1.0, 1.0, 1.0, 1.0};

        /* set up ambient, diffuse, and specular components for light 0 */

    glLightfv (GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv (GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv (GL_LIGHT0, GL_SPECULAR, light_specular);

        /* define material proerties for front face of all polygons */

    glMaterialfv (GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv (GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv (GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialf (GL_FRONT, GL_SHININESS, mat_shininess);
    glEnable (GL_SMOOTH); /*enable smooth shading */
    glEnable (GL_LIGHTING); /* enable lighting */
    glEnable (GL_LIGHT0);  /* enable light 0 */
    glEnable (GL_DEPTH_TEST); /* enable z buffer */
    glClearColor (1.0, 1.0, 1.0, 1.0);
    glColor3f (0.0, 0.0, 0.0);
}
```

UNIT -8

IMPLEMENTATION

1. Discuss the Bresenham's rasterization algorithm. How is it advantageous when compared to other existing methods? Describe. (Jun2012) 10M

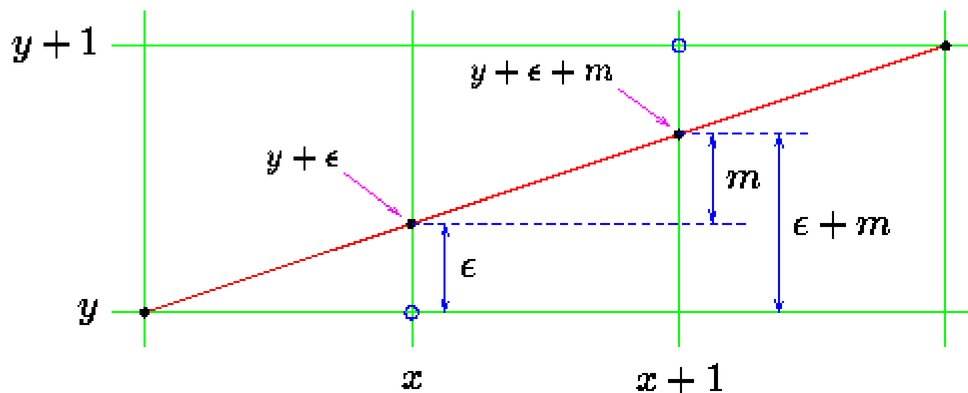
Ans: Consider drawing a line on a raster grid where we restrict the allowable slopes of the line to the range $0 \leq m \leq 1$.

If we further restrict the line-drawing routine so that it always increments *x* as it plots, it becomes clear that, having plotted a point at *(x,y)*, the routine has a severely limited range of options as to where it may put the *next* point on the line:

- It may plot the point *(x+1,y)*, or:
- It may plot the point *(x+1,y+1)*.

So, working in the *first positive octant* of the plane, line drawing becomes a matter of deciding between two possibilities at each step.

We can draw a diagram of the situation which the plotting program finds itself in having plotted *(x,y)*.



In plotting *(x,y)* the line drawing routine will, in general, be making a compromise between what it would like to draw and what the resolution of the screen actually allows it to draw. Usually the plotted point *(x,y)* will be in error, the actual, mathematical point on the line will not be addressable on the pixel grid. So we associate an error, ϵ, with each *y* ordinate, the real value of *y* should be $y + \epsilon$. This error will range from -0.5 to just under +0.5.

In moving from x to $x+1$ we increase the value of the true (mathematical) y-ordinate by an amount equal to the slope of the line, m. We will choose to plot $(x+1,y)$ if the difference between this new value and y is less than 0.5.

$$y + \epsilon + m < y + 0.5$$

Otherwise we will plot $(x+1,y+1)$. It should be clear that by so doing we minimise the total error between the mathematical line segment and what actually gets drawn on the display.

The error resulting from this new point can now be written back into ϵ, this will allow us to repeat the whole process for the next point along the line, at $x+2$.

The new value of error can adopt one of two possible values, depending on what new point is plotted. If $(x+1,y)$ is chosen, the new value of error is given by:

$$\epsilon_{new} \leftarrow (y + \epsilon + m) - y$$

Otherwise it is:

$$\epsilon_{new} \leftarrow (y + \epsilon + m) - (y + 1)$$

This gives an algorithm for a DDA which avoids rounding operations, instead using the error variable ϵ to control plotting:

$\epsilon \leftarrow 0, \quad y \leftarrow y_1$
For $x \leftarrow x_1$ **to** x_2 **do**
  Plot point at (x, y).
  If ($\epsilon + m < 0.5$)
    $\epsilon \leftarrow \epsilon + m$
  Else
    $y \leftarrow y + 1, \quad \epsilon \leftarrow \epsilon + m - 1$
  EndIf
EndFor

This still employs floating point values. Consider, however, what happens if we multiply across both sides of the plotting test by Δx and then by 2:

$$\begin{aligned} \epsilon + m &< 0.5 \\ \epsilon + \Delta y/\Delta x &< 0.5 \\ 2\epsilon\Delta x + 2\Delta y &< \Delta x \end{aligned}$$

All quantities in this inequality are now integral.

Substitute ϵ' for $\epsilon\Delta x$. The test becomes:

$$2(\epsilon' + \Delta y) < \Delta x$$

This gives an *integer-only* test for deciding which point to plot.

The update rules for the error on each step may also be cast into ϵ' form. Consider the floating-point versions of the update rules:

$$\epsilon \leftarrow \epsilon + m$$
$$\epsilon \leftarrow \epsilon + m - 1$$

Multiplying through by Δx yields:

$$\epsilon\Delta x \leftarrow \epsilon\Delta x + \Delta y$$
$$\epsilon\Delta x \leftarrow \epsilon\Delta x + \Delta y - \Delta x$$

which is in ϵ' form.

$$\epsilon' \leftarrow \epsilon' + \Delta y$$
$$\epsilon' \leftarrow \epsilon' + \Delta y - \Delta x$$

Using this new ``error" value, ϵ', with the new test and update equations gives Bresenham's integer-only line drawing algorithm:

$\epsilon' \leftarrow 0, \quad y \leftarrow y_1$
For $x \leftarrow x_1$ **to** x_2 **do**
  Plot point at (x, y).
  If ($2(\epsilon' + \Delta y) < \Delta x$)
    $\epsilon' \leftarrow \epsilon' + \Delta y$
  Else
    $y \leftarrow y + 1, \quad \epsilon' \leftarrow \epsilon' + \Delta y - \Delta x$
  EndIf
EndFor

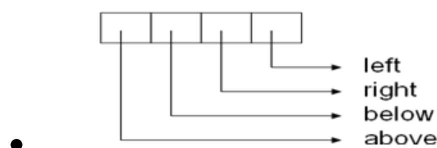
- Integer only - hence efficient (fast).
- Multiplication by 2 can be implemented by left-shift.
- This version limited to slopes in the first octant, $0 \leq m \leq 1$.

2. Explain the cohen-sutherland line clipping algorithm in detail. (Jun2012) 10M

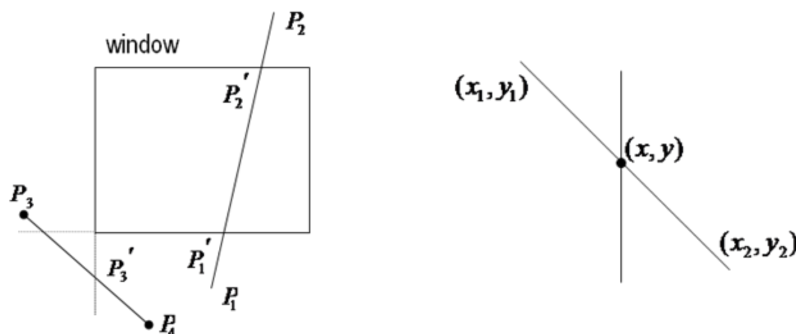
Ans: Basic Idea

- Encode the line endpoints
- Successively divide the line segments so that they are completely contained in the window or completely lies outside the window

1001	1000	1010
0001	0000	0010
0101	0100	0110

- 

Clipping happens as follows :



3. Derive the mathematical formula for bresenham's mid point line algorithm. (Dec2011) 10M

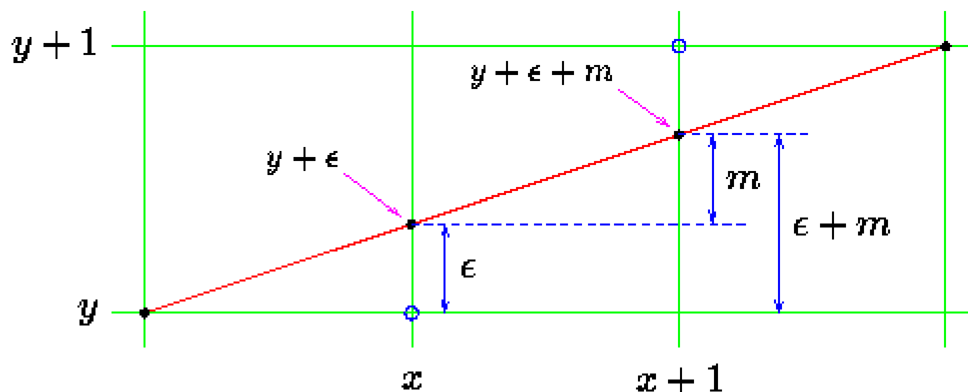
Ans: Consider drawing a line on a raster grid where we restrict the allowable slopes of the line to the range $0 \leq m \leq 1$.

If we further restrict the line-drawing routine so that it always increments *x* as it plots, it becomes clear that, having plotted a point at *(x,y)*, the routine has a severely limited range of options as to where it may put the *next* point on the line:

- It may plot the point *(x+1,y)*, or:
- It may plot the point *(x+1,y+1)*.

So, working in the *first positive octant* of the plane, line drawing becomes a matter of deciding between two possibilities at each step.

We can draw a diagram of the situation which the plotting program finds itself in having plotted *(x,y)*.



In plotting *(x,y)* the line drawing routine will, in general, be making a compromise between what it would like to draw and what the resolution of the screen actually allows it to draw. Usually the plotted point *(x,y)* will be in error, the actual, mathematical point on the line will not be addressable on the pixel grid. So we associate an error, ϵ, with each *y* ordinate, the real value of *y* should be $y+\epsilon$. This error will range from -0.5 to just under +0.5.

In moving from *x* to *x+1* we increase the value of the true (mathematical) y-ordinate by an amount equal to the slope of the line, *m*. We will choose to plot *(x+1,y)* if the difference between this new value and *y* is less than 0.5.

$$y+\epsilon+m < y+0.5$$

Otherwise we will plot *(x+1,y+1)*. It should be clear that by so doing we minimise the total error between the mathematical line segment and what actually gets drawn on the display.

The error resulting from this new point can now be written back into ϵ, this will allow us to repeat the whole process for the next point along the line, at *x+2*.

The new value of error can adopt one of two possible values, depending on what new point is plotted. If *(x+1,y)* is chosen, the new value of error is given by:

$$\epsilon_{new} \leftarrow (y + \epsilon + m) - y$$

Otherwise it is:

$$\epsilon_{new} \leftarrow (y + \epsilon + m) - (y + 1)$$

This gives an algorithm for a DDA which avoids rounding operations, instead using the error variable ϵ to control plotting:

$\epsilon \leftarrow 0, \quad y \leftarrow y_1$
For $x \leftarrow x_1$ **to** x_2 **do**
  Plot point at (x, y).
  If ($\epsilon + m < 0.5$)
    $\epsilon \leftarrow \epsilon + m$
  Else
    $y \leftarrow y + 1, \quad \epsilon \leftarrow \epsilon + m - 1$
  EndIf
EndFor

This still employs floating point values. Consider, however, what happens if we multiply across both sides of the plotting test by Δx and then by 2:

$$\begin{aligned} \epsilon + m &< 0.5 \\ \epsilon + \Delta y/\Delta x &< 0.5 \\ 2\epsilon\Delta x + 2\Delta y &< \Delta x \end{aligned}$$

All quantities in this inequality are now integral.

Substitute ϵ' for $\epsilon\Delta x$. The test becomes:

$$2(\epsilon' + \Delta y) < \Delta x$$

This gives an *integer-only* test for deciding which point to plot.

The update rules for the error on each step may also be cast into ϵ' form. Consider the floating-point versions of the update rules:

$$\epsilon \leftarrow \epsilon + m$$
$$\epsilon \leftarrow \epsilon + m - 1$$

Multiplying through by Δx yields:

$$\epsilon\Delta x \leftarrow \epsilon\Delta x + \Delta y$$
$$\epsilon\Delta x \leftarrow \epsilon\Delta x + \Delta y - \Delta x$$

which is in ϵ' form.

$$\epsilon' \leftarrow \epsilon' + \Delta y$$
$$\epsilon' \leftarrow \epsilon' + \Delta y - \Delta x$$

Using this new ``error" value, ϵ', with the new test and update equations gives Bresenham's integer-only line drawing algorithm:

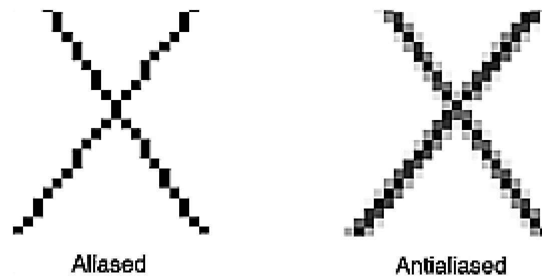
$\epsilon' \leftarrow 0, \quad y \leftarrow y_1$
For $x \leftarrow x_1$ **to** x_2 **do**
  Plot point at (x, y).
  If ($2(\epsilon' + \Delta y) < \Delta x$)
    $\epsilon' \leftarrow \epsilon' + \Delta y$
  Else
    $y \leftarrow y + 1, \quad \epsilon' \leftarrow \epsilon' + \Delta y - \Delta x$
  EndIf
EndFor

- Integer only - hence efficient (fast).
- Multiplication by 2 can be implemented by left-shift.
- This version limited to slopes in the first octant, $0 \leq m \leq 1$.

4. What do you mean by antialiasing? Explain. (Jul2011) 4M

Ans: You might have noticed in some of your OpenGL pictures that lines, especially nearly horizontal or nearly vertical ones, appear jagged. These jaggies appear because the ideal line is approximated by a series of pixels that must lie on the pixel grid. The jaggedness is called aliasing, and this section describes antialiasing techniques to reduce it. Figure 6-2 shows two

intersecting lines, both aliased and antialiased. The pictures have been magnified to show the effect.

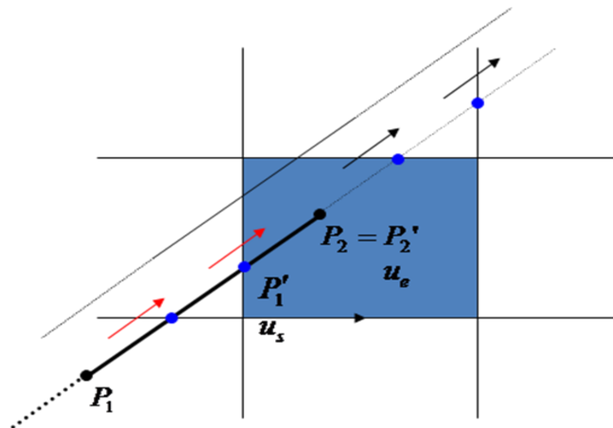


5. Explain Liang Barsky line clipping algorithm in detail. (Jun2010) 10M

Ans:

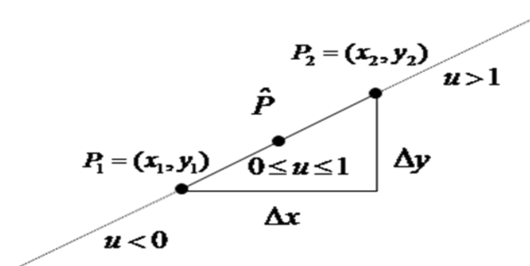
$$\hat{P} = P_1 + u(P_2 - P_1),$$
$$0 \le u \le 1$$

$$P_1' : \max_u \{P_1 \equiv 0, \uparrow\} = \boxed{u_s}$$
$$P_2' : \min_u \{P_2 \equiv 1, \uparrow\} = \boxed{u_e}$$

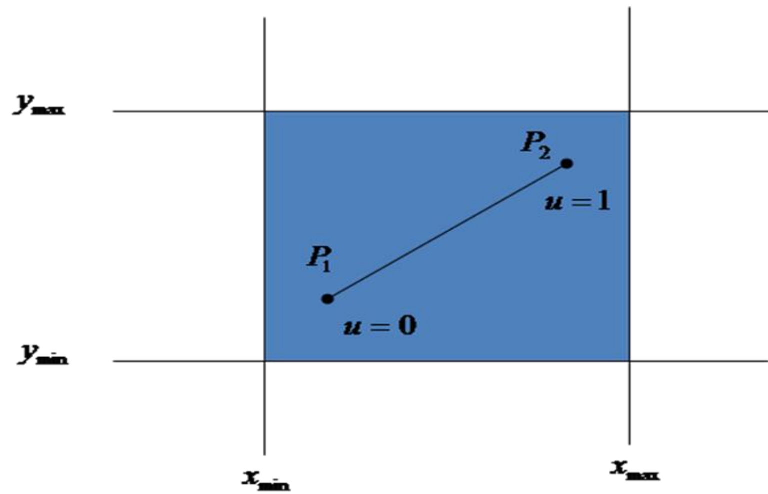


$$\hat{P} = vP_1 + uP_2$$
$$u + v = 1$$

$$\therefore \hat{P} = vP_1 + uP_2$$
$$= (1-u)P_1 + uP_2$$
$$= P_1 + u(P_2 - P_1)$$



$$\hat{P} = \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_1 + u(x_2 - x_1) \\ y_1 + u(y_2 - y_1) \end{pmatrix}$$
$$\therefore x = x_1 + u\Delta x$$
$$y = y_1 + u\Delta y$$



$$x_{min} \le x_1 + u\Delta x \le x_{max}$$
$$y_{min} \le y_1 + u\Delta y \le y_{max}$$
$$\forall 0 \le u \le 1$$