COMPUTER GRAPHICS AND VISUALIZATION.

STUDY GUIDE - UNIT - (1).

A survey of computer graphics [Applications]

→ Use of computer graphics is widespread.

**[Refer: Computer graphics with OpenGL - Heaun & Baker]

→ Graphs and charts: financial, statistical, mathematical, scientific, economical data

→ Computer-Aided Design: computer Aided drafting and design. used is design of buildings, automobiles, aircraft, spacecraft, textiles, appliances etc. Network communication, water supply real time applications.

→ Virtual-Reality Environments: User can interact with objects in three-dimensional scene. Simulation of architectual designs, flight training etc.

→ Data Visualizations: Representing data sets for better interpretation / scientific analysis.

→ Education and Training: Simulators for practical sessions.

→ Computer Art: Drawings, Designing logos, page layout advertising.

→ Entertainment: Animations.

→ Image Processing: Medicine (PET scan, X ray, CT) Ultrasound imaging.

→ Graphical User Interface:

Overview of Graphics systems.

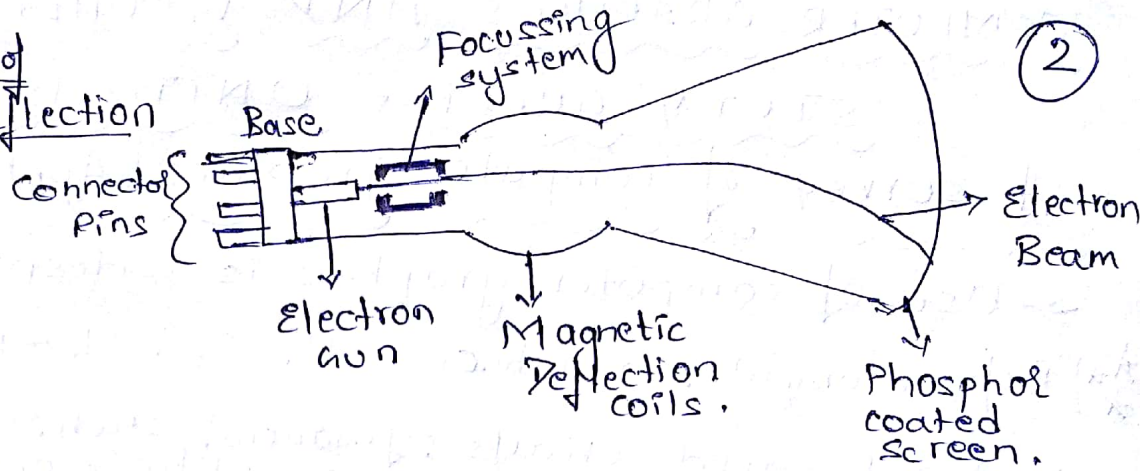
**[Refer: Computer graphics with OpenGl - Heaun & Baker]

→ Video Display Devices:

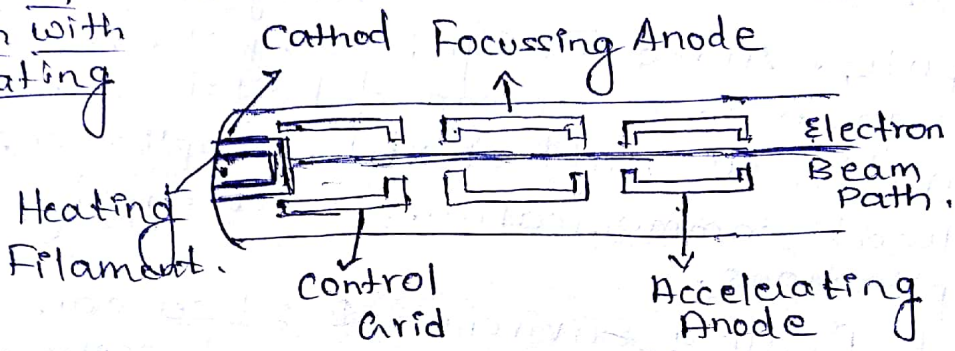
The primary output device in a graphics system is a video monitor. Most of the video monitors is based on the Cathod-Ray-Tube (CRT) design.

✓ Refresh Cathod-Ray Tubes: A beam of electrons emitted by an electron-gun, passes through focusing and deflection systems, that direct the beam towards specified positions on the phosphor-coated screen.

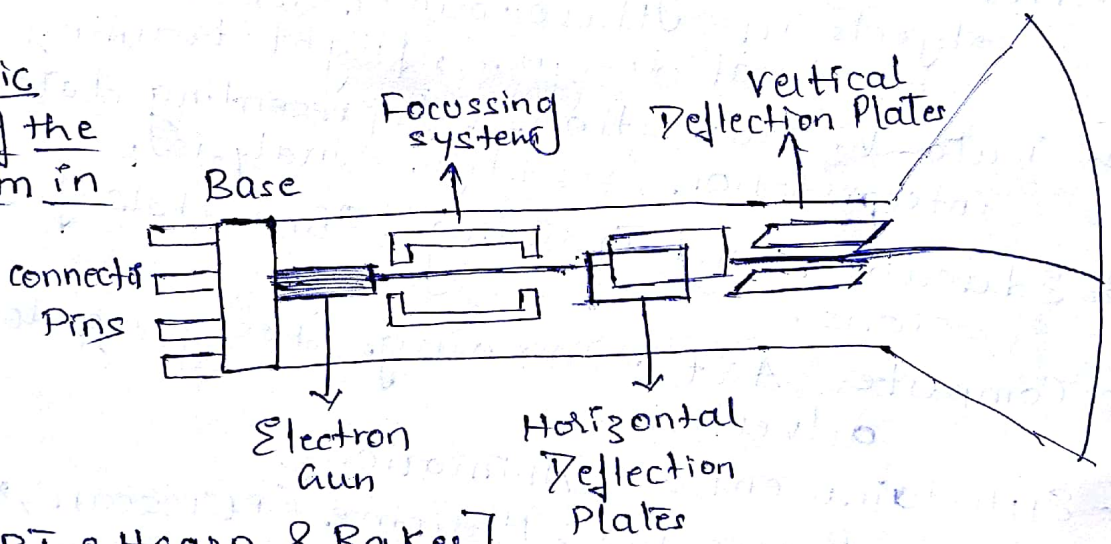
Basic design of a magnetic-deflection CRT



Operation of an electron gun with an accelerating anode



Electrostatic Deflection of the electron beam in a CRT



[Working of CRT : Hearn & Baker]

- The common method to maintain the phospha glow is to redraw the pictue repeatedly by quickly directing the electron beam back over the same screen points. This type of display is called a refresh CRT and the frequency at which the pictue is redrawn on the screen is called the refresh rate.
- Persistence : how long the phospha continues to emit light after the CRT beam is removed.
- The maximum number of points that can be displayed without overlap on a CRT is referred to as resolution.

→ Raster-Scan Displays: In raster-scan display the electron beam is swept across the screen, one row at a time from top to bottom. Each row is referred to as a scan line. As the electron beam moves across a scan line, the beam intensity is turned on and off to create a pattern of illuminated spots. Picture definition is stored in a memory area called the refresh buffer or frame buffer, where the term frame refers to the total screen area. This memory area holds the set of color values for the screen points. These stored color values are then retrieved from the refresh buffer and used to control the intensity of the electron beam as it moves from spot to spot across the screen.

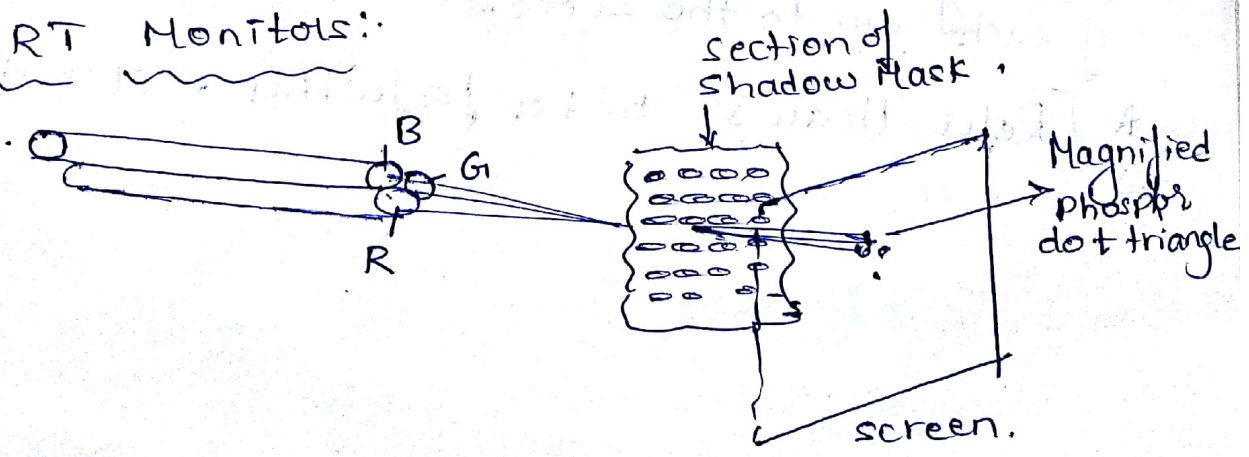
* [Refer Hearn & Baker for further explanation]

** Key concepts: frame buffer, pixel (picture element) or pel, color buffer, resolution, aspect ratio → number of pixel columns divided by no. of scan lines.

→ Random Scan Displays: CRT has the electron beam directed only to those parts of the screen where a picture is to be displayed. Also called vector displays or stroke writing displays or calligraphic displays. Refresh rate on a random scan depends on the number of lines to be displayed on the system. Picture definition is now stored as a set of line drawing commands in an area of memory referred to as the display list, refresh display file, vector file or display program.

[explanation contd... Hearn & Baker]

→ Color CRT Monitors:



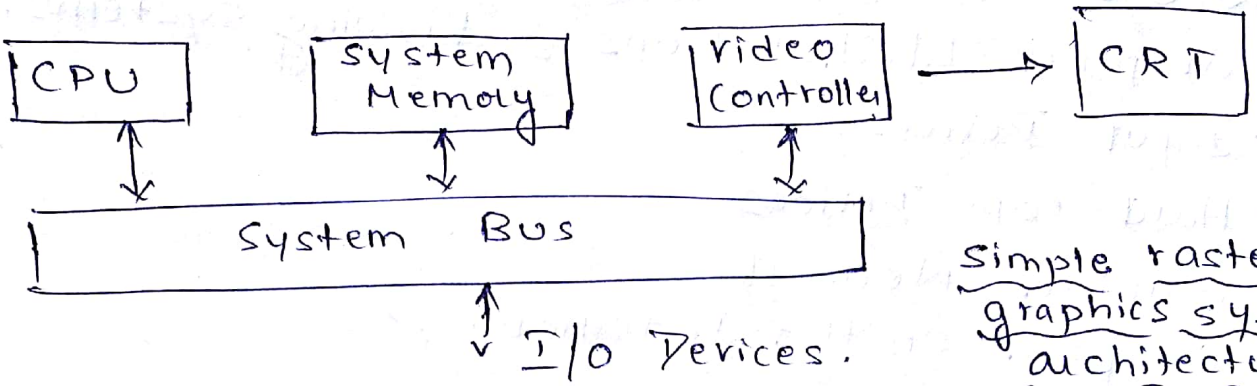
→ Flat Panel Displays: Reduced volume, weight and power requirements compared to a CRT
Two categories: emissive displays and non-emissive displays.
The emissive displays (or emitters) are devices that convert electrical energy into light. ex. Plasma panels, thin film electroluminescent displays and light emitting diodes.
Non-emissive displays use optical effects to convert sunlight or light from some other source into graphics panel. ex:- Liquid crystal device.
* [Working of each type of display refer: Hearn & Baker].

→ Three Dimensional viewing Devices:
Graphics monitors for the display of 3D scenes have been derised using a technique that reflects a CRT image from a vibrating flexible mirror.
As the varifocal mirror vibrates, it changes focal length. These vibrations are synchonized with the display of an object on a CRT so that each point on the object is reflected from the mirror into a spatial position corresponding to the distance of that point from a specified viewing location.

→ Stereoscopic and Virtual Reality systems:
This method does not produce true 3D images. but it does provide a three-dimensional effect by presenting a different view to each eye of an observer so that scenes do appear to have depth. To obtain a stereoscopic projection, we must obtain two views of a scene generated with viewing directions along the lines from the position of each eye to the scene.
* [Refer Hearn & Baker for further explanation]

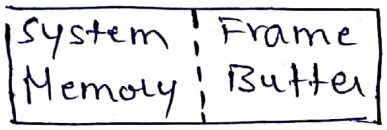
Raster Scan Systems.

(i)



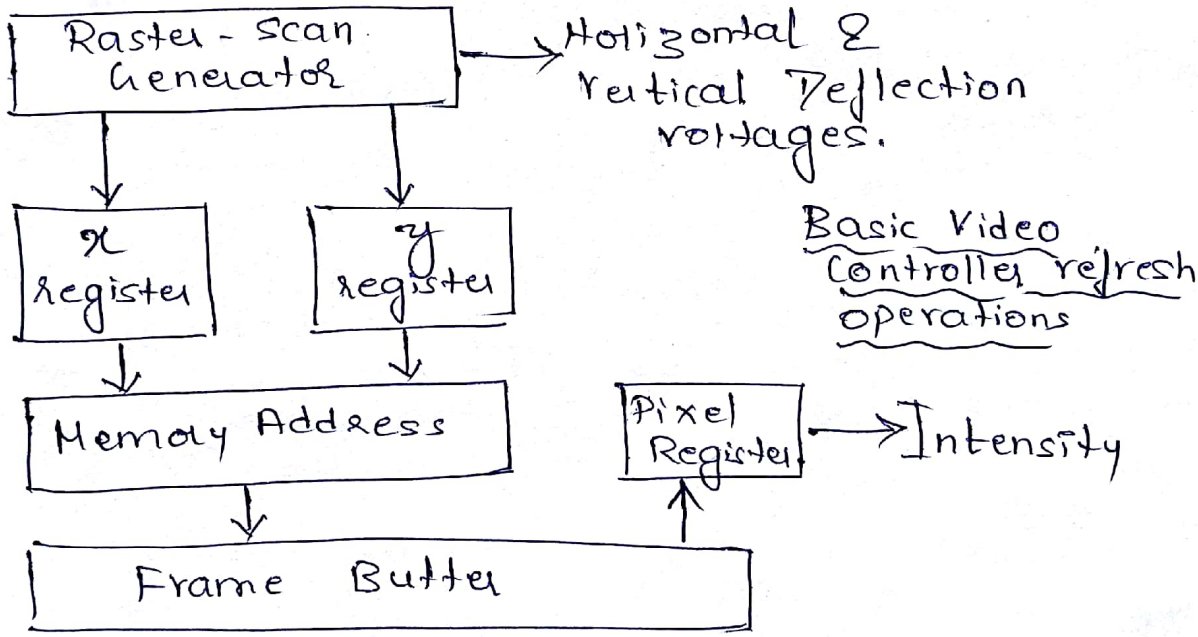
Simple raster-graphics system architecture

(ii)



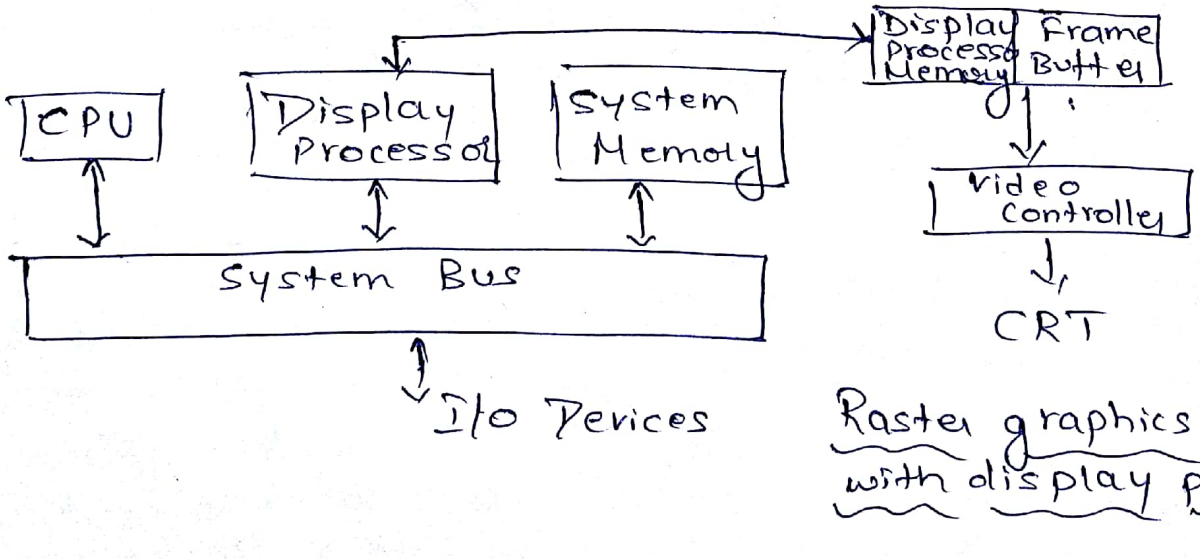
→ Fixed portion of system memory reserved for frame buffer.

(iii)



Basic Video Controller refresh operations

(iv)



Raster graphics system with display processor

★★★ [Explation of all architectures refer Hearn & Baker]

★ Self study. [Learn & Baker]

→ Graphics Workstations & Viewing systems.
→ Input Devices.
→ Hard-copy Devices
→ Graphics Networks
→ Graphics on the Internet.

Graphics Software

→ Special Purpose packages are designed for non-programmers who want to generate pictures graphs or charts in some application area without worrying about graphics procedures that might be needed to produce such displays.
The interface to a special purpose package is typically a set of menus that allows users to communicate with the programs in their own terms.
Ex:: CAD systems, photoshop, coraldraw etc.

→ A general purpose programming package provides a library of graphics functions that can be used in programming languages such as C, C++, Java etc.
Basic functions of a typical graphics library are straight lines, polygons and other objects, setting color values, selecting views of a scene and applying transformations. Ex. OpenGL, VRML (Virtual-Reality Modeling Language)

→ A set of graphics functions is often called a CG API because the library provides a software interface between a programming language and the hardware.

Coordinate Representation:

→ First we need to give the geometric descriptions of the objects that are to be displayed.
These descriptions determine the locations and shapes of the objects Ex:- box requires vertices, sphere requires centre and radius.

→ General graphics package require geometric descriptions to be specified in a standard, right-handed, Cartesian-co-ordinate reference frame..

→ The shapes of individual objects such as trees or furniture should be first defined within a separate co-ordinate reference frame for each object.
These reference frames are called modelling co-ordinates / local co-ordinates or master co-ordinates

→ Once the individual object shapes have been specified we can construct or model a scene by placing the objects into appropriate locations within a scene reference frame called world co-ordinates.

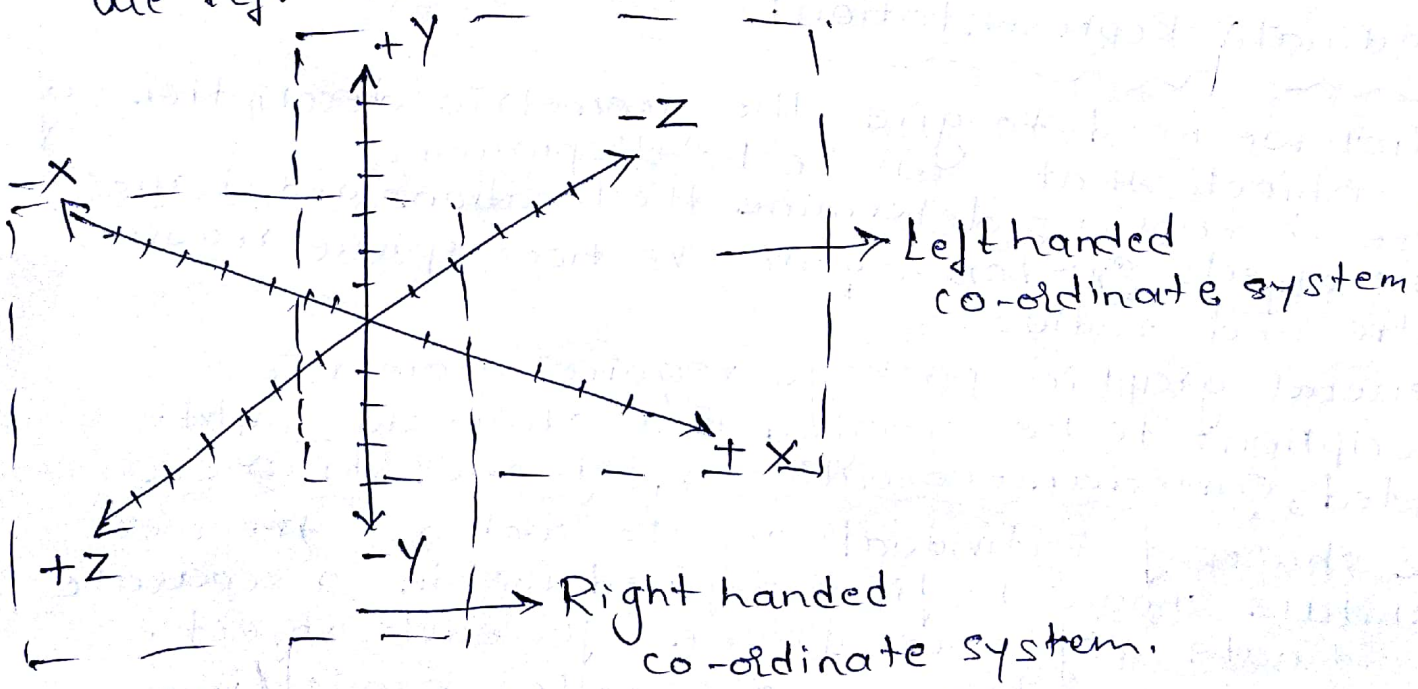
→ After all parts of the scene have been specified the overall world-coordinates description is processed through various routines onto one or more output devices reference frames for display. This process is called the viewing pipeline

→ World co-ordinate positions are first converted to viewing coordinates corresponding to the view we want of a scene (hypothetical camera view)

→ Then object locations are transformed into a two dimensional projection of the screen, which corresponds to what we will see on the output device, the scene is then stored in normalized co-ordinates where the range is -1 to 1 or 0 to 1 depending on the system.

→ The co-ordinate system for display devices are generally called device co-ordinates or screen co-ordinates in this the monitor screen.

Both normalized co-ordinates and screen coordinates are left-handed coordinate reference frame.



Modeling co-ordinates → World co-ordinates → Viewing & Projection co-ordinates → normalized co-ordinates → Device co-ordinates.

$(x_{mc}, y_{mc}, z_{mc}) \rightarrow (x_{wc}, y_{wc}, z_{wc}) \rightarrow$
$(x_{vc}, y_{vc}, z_{vc}) \rightarrow (x_{pc}, y_{pc}, z_{pc}) \rightarrow$
$(x_{nc}, y_{nc}, z_{nc}) \rightarrow (x_{dc}, y_{dc})$

range (0,0) to (x_{max}, y_{max})

Graphics Functions: * [Make a list]

- graphics output primitives.
- transformations
- viewing
- Input functions.
- control operations.

Introduction to openGL [Hardware independent]

Data types:

GLbyte, GLshort, GLint, GLfloat, GLdouble, GLboolean.

symbolic constants:

GL-2D, GL-3D, GL-RGB, GL-POLYGON, GL-EQUAL,

Display Window Management:

```
glutInit(&argc, argv);
glutDisplayFunc(linesegment); // linesegment function is passed to the window.
glutCreateWindow("<name given to the application>");
```

Ex:- `glutCreateWindow("First openGL Window");`

glutMainLoop() ; //activates all display windows created.

glutInitWindowSize (100,100) ; //100 * 100 pixels screen

glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

glClearColor (<R>, <G>, <B>, <alpha_value>)

↓ 0.0 → transpaent
1.0 → opaque

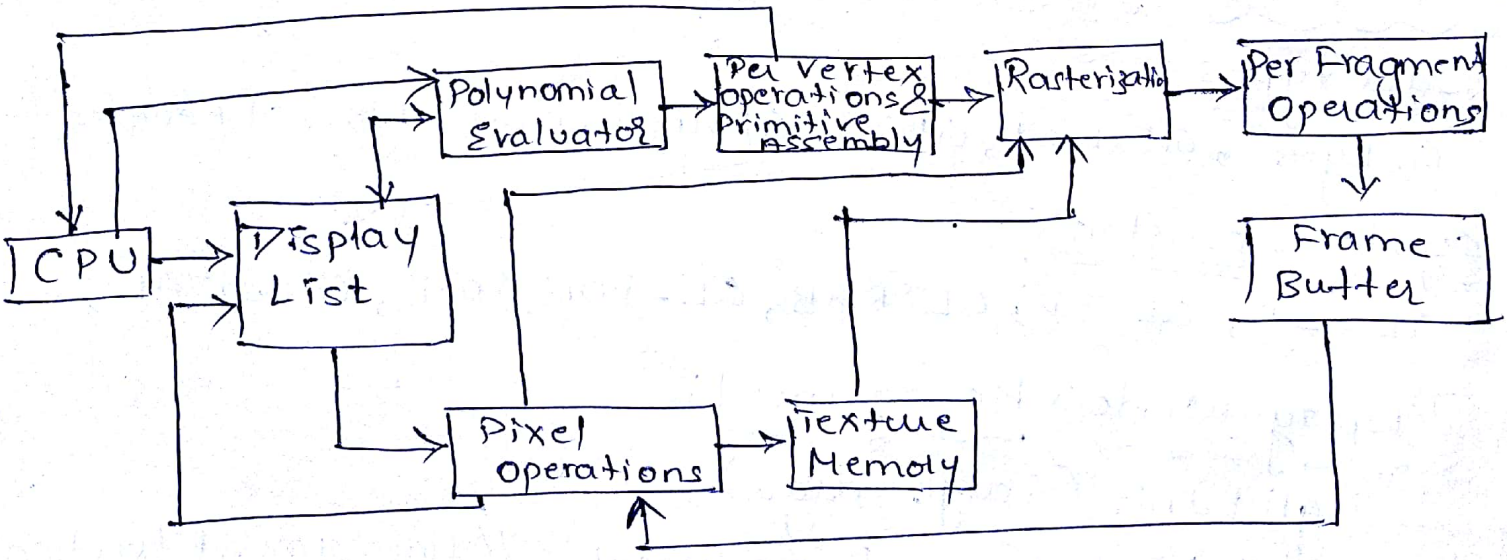
* 1,1,1 → White
* 0,0,0 → Black.

glClear (GL_COLOR_BUFFER_BIT) ; → clears the color screen

glMatrixMode (GL_PROJECTION);
gluOrtho2D(0.0, 200.0, 0.0, 150.0);
glBegin (GL_LINES) ;
    glVertex2i (180,15);
    glVertex2i (10,145);
glEnd();

} draws a line between (180,15) & (10,145)

OpenGL Architecture: [Pipeline]

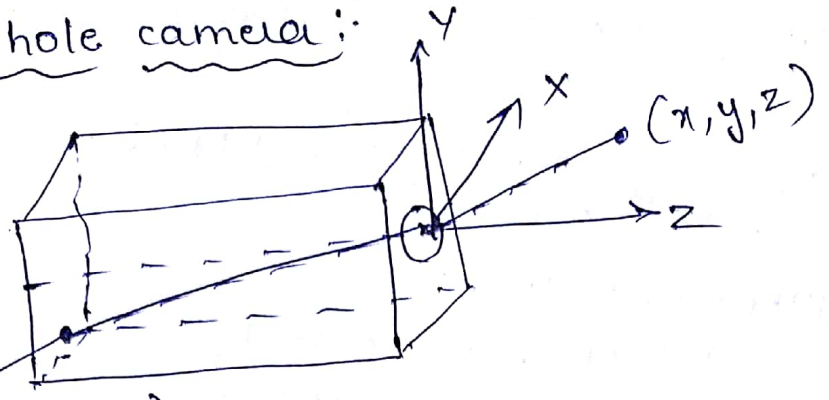


// First Opengl code.

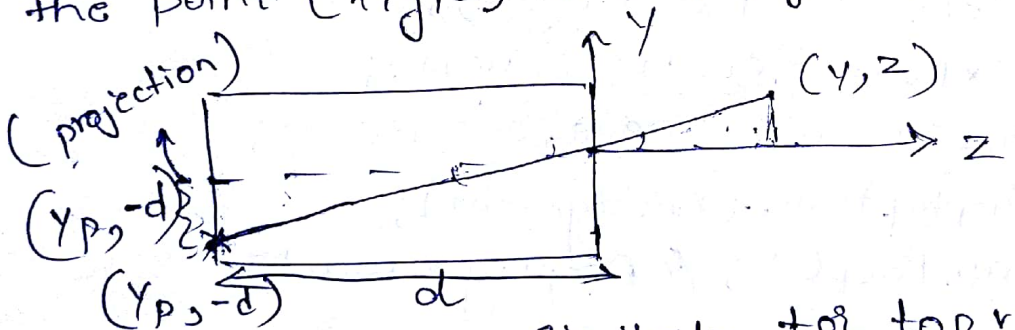
```
void lineSegment()
{
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1.0,0.0,0.0);
  glBegin(GL_LINES);
        glVertex2i(180,15);
        glVertex2i(10,145);
  glEnd();
  glFlush(); // forces execution, buffers emptied.
}
void main(int argc, char **argv)
{ glutInit(&argc, argv);
  glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
  glutInitWindowPosition(50,100);
  glutInitWindowSize(400,300);
  glutCreateWindow("First OpenGL program");
  glClearColor(1.0,1.0,1.0,0.0);
  glMatrixMode(GL_PROJECTION);
  gluOrtho2D(0.0,200.0,0.0,150.0);
  glutDisplayFunc(lineSegment);
  glutMainLoop(); // Display & wait.
}
```

Imaging Systems: [Source: Edward Angel Interactive Computer Graphics.]

→ The Pinhole camera:



A pinhole camera is a box with a small hole in the centre of one side of the box, the film is placed inside the box on the side opposite the pinhole. Initially the pinhole is covered. It is uncovered for a short time to expose the film. Suppose that we orient our camera along the z-axis with the pinhole at the origin of our co-ordinate system. We assume that the hole is so small that only a single ray of light emanating from a point can enter it. The film plane is located at a distance d from the pinhole. A side view allows us to calculate where the image of the point (x,y,z) is on the film plane z = -d.



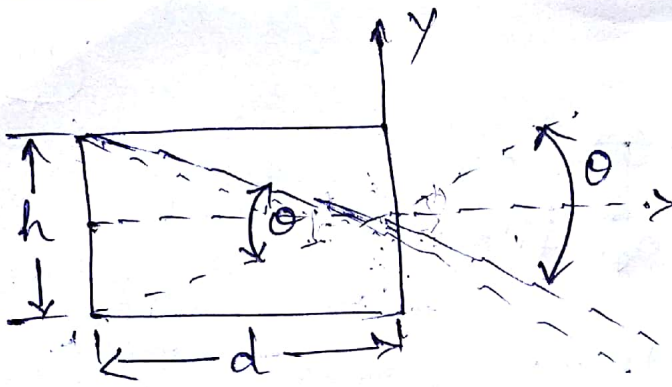
Similarly for top view yields

$$y_p = \frac{-y}{z/d}$$

$$x_p = \frac{x}{z/d}$$

The point $(x_p, y_p, -d)$ is called the projection of the point (x,y,z). All points along the line between (x,y,z) and $(x_p, y_p, -d)$ all project to $(x_p, y_p, -d)$

Field of angle of view of our camera is the angle made by the largest object that our camera can image on its film plane.



$$\theta = 2 \tan^{-1} \frac{h}{2d}$$

If h is the height of the camera, then θ is the angle of view

The ideal pinhole camera has an infinite depth of field. Every point within its field of view is in focus, regardless of how far it is from the camera.

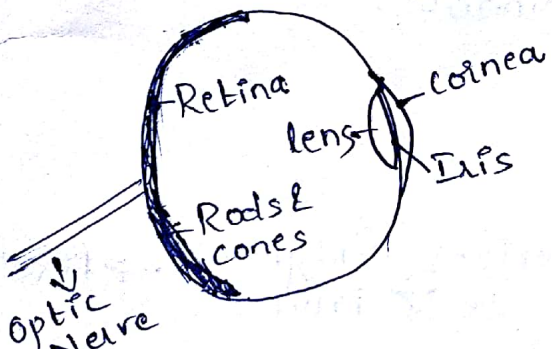
The pinhole camera has two disadvantages:

→ pinhole is small hence admits only a single ray from a point source.

→ camera cannot be adjusted to have a different angle of view.

→ The Human visual system: Our extremely complex visual system has all the components of a physical imaging system, such as a camera or a microscope. Light enters the eye through the lens and cornea, a transparent structure that protects the eye. The lens forms an image on a two dimensional structure called the retina at the back of the eye. The rods and cones are light sensors and are located on the retina. They are excited by electromagnetic energy in the range of 350 to 780 nm

[Refer: Edward Angel. Further reading]



The synthetic camera model:

Optical imaging systems lead directly to the conceptual foundation for modern three-dimensional computer graphics. Idea is to create a computer generated image as being similar to forming an image using an optical system.

Consider the following imaging system. The system has objects and a viewer (a camera)

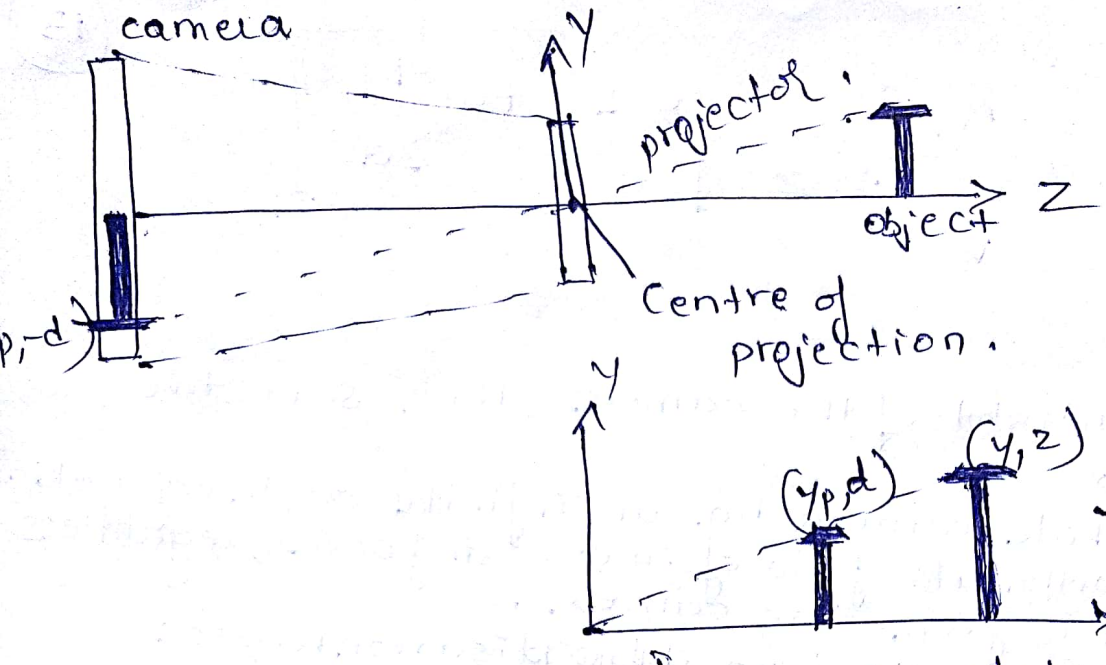


Image moved to front of camera.

Note that the image of the object is flipped relative to the object. Whereas with a real camera, we would simply flip the film to regain the original orientation of the object, with our synthetic camera we can avoid the flipping by a simple trick. We draw another plane in front of the lens and work in three dimensions.

★ Terminology : Projector
Centre of Projection
Projection plane
Clipping rectangle/window.

Graphics Pipeline:

vertices → Vertex Processor → Clipping & primitive assembler → Rasterizer → Fragment Processor → Pixels

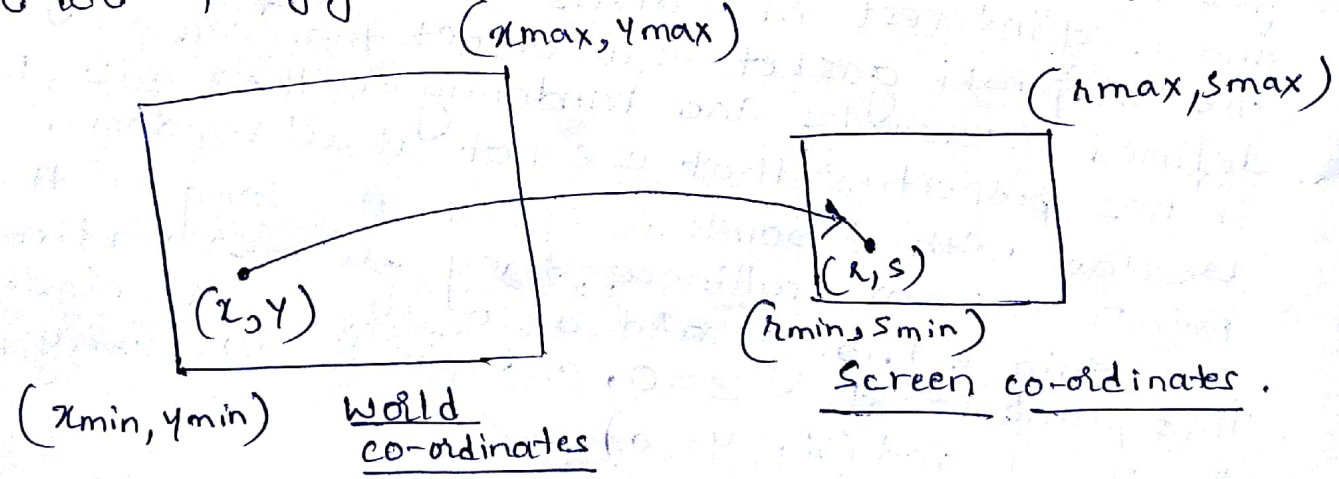
★★ Explain each stage in detail.

Co-ordinate system:

Originally, graphics systems required the user to specify all information, such as vertex locations, directly in units of the display device. If that were true for high level application programs, we would have to talk about points in terms of screen locations in pixels or centimeters from a corner of the display. There are obvious problems with this method, not the least of which is the absurdity of using distances on the computer screen to describe phenomena where the natural unit might be light years or microns. One of the major advances in graphics software systems occured when the graphics systems allowed users to work in any co-ordinate system that they desired.

The user's co-ordinate system became known as the world co-ordinate system or the application model or object co-ordinate system.

Units on the display were first called physical device co-ordinates or just device co-ordinates. For raster devices, such as most CRT displays, we use the term window co-ordinates or screen co-ordinates. Screen co-ordinates are always expressed in some integer type, because the center of any pixel in the frame buffer must be located on a fixed grid or equivalently, because pixels are inherently discrete and we specify their locations using integers.



Co-ordinate reference frames:

To describe a picture, we first decide upon a convenient Cartesian co-ordinate system, called the world-co-ordinate reference frame, which could be either two dimensional or three dimensional. We then describe the objects in our picture by giving their geometric specifications in terms of positions in world co-ordinates.

Screen Co-ordinates:

Locations on a video monitor are referenced in integer screen co-ordinates, which correspond to the pixel positions in the frame buffer. Pixel coordinate values give the scan line number (y value) and the column number (x value). Hardware processes such as screen refreshing, typically address pixel positions with respect to the top left corner of the screen. Scan lines are then referenced from 0 at the top of the screen, to some integer value ymax at the bottom of the screen, and pixel positions along each scan line are numbered from 0 to xmax, left to right.

The 2-D Sierpinski Gasket.

Let us consider drawing a sample Sierpinski gasket - an interesting shape that has a long history and is of interest in areas such as fractal geometry. The Sierpinski gasket is an object that can be defined recursively and randomly; in the limit, however it has properties that are not at all random.

We start with 3 points in space. As long as the points are not collinear, they are the vertices of a unique triangle and also define a unique plane. This plane is at $z=0$, and points are $(x_1, y_1, 0)$ $(x_2, y_2, 0)$ and $(x_3, y_3, 0)$

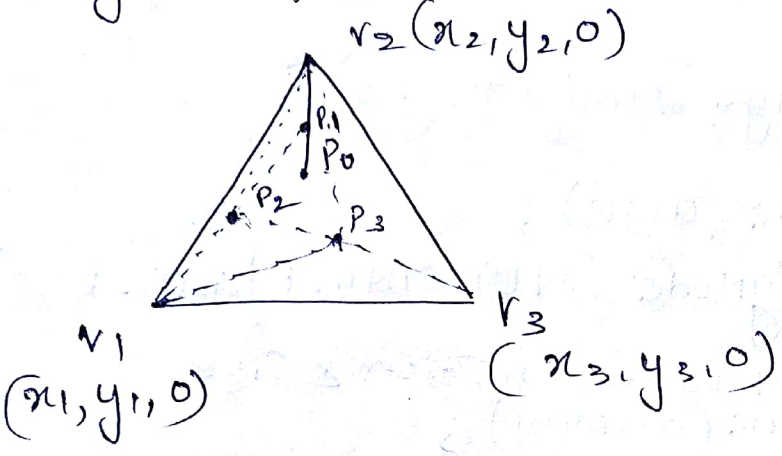
The algorithm for construction of the sierpinski gasket are as follows.

1. → Pick an initial point at random inside the triangle
2. → Select one of the 3 points at random
3. → Find the location halfway between the initial point and the randomly selected vertex

(4) → Display the new point by putting some sort of marker such as a small circle at the corresponding location on the display.

(5) → Replace the point at (x, y, z) with this new point

(6) → goto step (2).



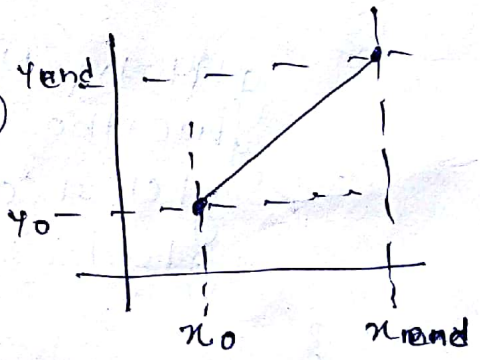

```
#include <GL/glut.h>
void init()
{
    glMatrixMode (GL_PROJECTION);
    gluOrtho2D (0.0, 50, 0.0, 50);
    glClearColor (0, 0, 0, 0) ; // black screen
    glColor3f (1, 1, 1);    //white image.
}
void display ()
{ GLfloat vertices[3][2] = { {0,0} , {25, 50}, {50, 0}};
  int i, j, k ;
  GLfloat p[2] = {7.5, 5.0} ;
  glClear (GL_COLOR_BUFFER_BIT) ;
  glBegin (GL_POINTS) ;
  for(k=0 ; k<5000 ; k++)
  {  j = rand() % 3 ;
     p[0] = (p[0] + vertices[j][0]) / 2 ;
```

```
        p[1] = (p[1] + vertices[j][1])/2;
        glVertex2fv(p);
    }
    glEnd();
    glFlush();
}
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);
    init(); //--> set window position & size.
    glutDisplayFunc(display);
    glutMainLoop();
    return 0;
}
```

UNIT-(2) - OPENGL IMPLEMENTATION.

Line Drawing Algorithms.

Line Equation: $mx + b = y$ — (1)



m → slope of the line

b → y intercept.

$$m = \frac{y_{end} - y_0}{x_{end} - x_0} \quad — (2)$$

$$b = y_0 - m x_0 \quad — (3)$$

For any given x interval δx along a line, we can compute y interval from δy as.

$$\delta y = m \, \delta x \quad — (4)$$

and $\therefore \delta x = \frac{\delta y}{m}$ — (5)

DDA [Digital Differential Analyzer] Algorithm (9)

This is a scan conversion line algorithm based on calculating either dy or δx using eqn. (4) or (5). A line is sampled at unit intervals in one co-ordinate and the corresponding integer values nearest the line path are determined for the other conditions.

Consider a line with positive slope, if slope is less than or equal to 1 ($m <= 1$) we sample at unit x intervals ($\delta x = 1$) and compute successive y values as,

$$y_{K+1} = y_K + m \quad -(6)$$

* where K takes 0 to end value and each y value is rounded to nearest integer corresponding to pixel position in the x column we are processing.

For lines with positive values greater than 1 we reverse x and y as

$$x_{K+1} = x_K + (1/m) \quad -(7) \quad * (\delta y = 1)$$

Equation (6) & (7) process lines from left to right.

If it is reversed then $\delta x = -1$

$$y_{K+1} = y_K - m \quad -(8)$$

&

$$x_{K+1} = x_K - (1/m) \quad -(9) \quad * (\delta y = -1)$$

Similar calculations are carried out using (6) to (9) to determine pixel positions along a line with negative slope. Thus if the absolute value of the slope is less than 1 and the starting point is from left to end point at left we set $\delta x = 1$ and calculate y as (6), when starting endpoint is at right (same slope) we set $\delta x = -1$ and obtain y as (8), for negative slope with absolute value greater than 1 we use $\delta y = -1$ and (9) & we use $\delta y = 1$ and (7)

```
#include <stdlib.h>
#include <math.h>
int round (const float a)
{
    return (int)(a+0.5);
}
void line_DDA (int xo, int yo, int xend, int yend)
{
    int dx = xend - xo;
    int dy = yend - yo;
    int steps, k;
    float xInc, yInc, x=xo, y=yo;
    if (fabs (dx) > fabs (dy))
        steps = fabs (dx);
    else
        steps = fabs (dy);
    xInc = float (dx) / float (steps);
    yInc = float (dy) / float (steps);
    setPixel (round (x), round (y));
    for (k=0; k<steps; k++)
    {
        x += xInc;
        y += yInc;
        setPixel (round (x), round (y));
    }
}
```

★ { Chang m to -ve if plot is right to left with δx or δy = -1 }

Positive slope: [left to right]

case 1 ; $m <= 1$

$\delta x = 1$, $y_{k+1} = y_k + m$.

$\delta x = -1$, $y_{k+1} = y_k - m$ ★

case 2 ; $m > 1$

$\delta y = 1$, $x_{k+1} = x_k + 1/m$

$\delta y = -1$, $x_{k+1} = x_k - (1/m)$ ★

Negative slope [left to right]

case 1 ; $m <= 1$

$\delta x = 1$, $y_{k+1} = y_k + m$

$\delta x = -1$, $y_{k+1} = y_k - m$ ★

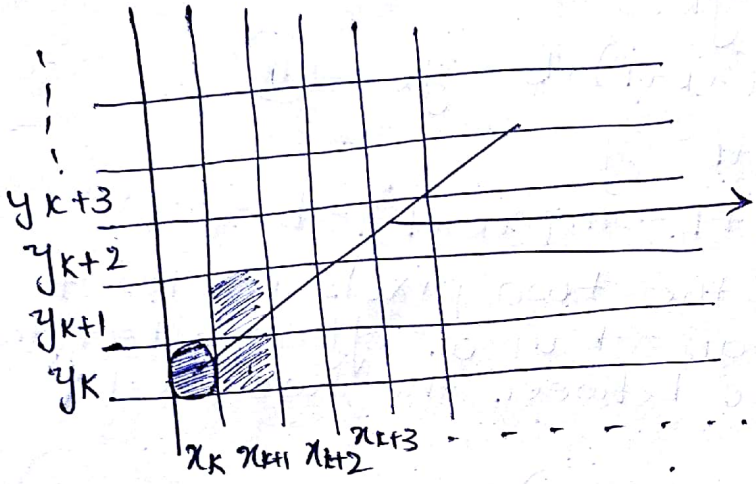
case 2 : $m > 1$, $\delta y = 1$, $x_{k+1} = x_k + 1/m$

$\delta y = -1$, $x_{k+1} = x_k - 1/m$ ★

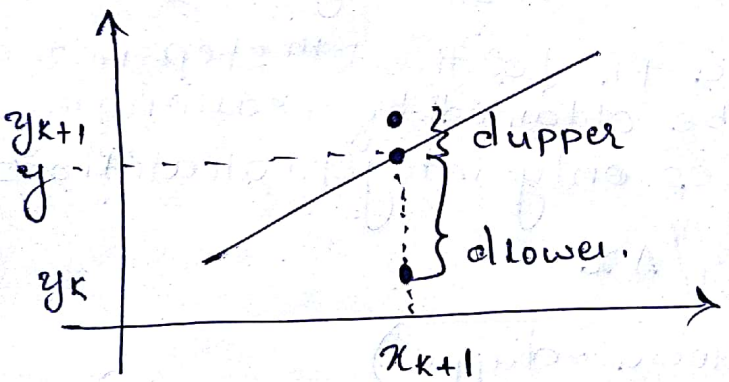
Bresenham's Line Algorithm:

This algorithm is an accuate and efficient raster line-generating algorithm dereloped by Bresenham, that uses only incremental integer calculations. It can also be adapted to display circles and other curves.

★ Draw back of DDA self study



$y = mx + b$.
★ $(m <= 1)$



To illustrate Bresenham's approach, first we consider the scan conversion process for lines with positive slope less than 1.0. Pixel positions along a line path are determined by sampling at unit x intervals, starting from the left endpoint (x_0, y_0) of a given line, we step to each successive column (x) and plot the pixel whose scan-line y value is closest to the line path. Assuming that we have determined that the pixel at (x_k, y_k) is to be displayed, we next need to decide which pixel to plot in column $x_{k+1} = x_k + 1$, the choices are pixel at positions (x_{k+1}, y_k) and (x_{k+1}, y_{k+1}).

At sampling position x_{k+1} we label the vertical pixel separations from the mathematical line path as dlower & dupper.
The y co-ordinate on the mathematical line at pixel column position x_{k+1} is calculated as

$$y = m(x_{k+1}) + b \quad -(10)$$

then

$$\text{dlower} = y - y_k.$$
$$= m(x_{k+1}) + b - y_k \quad -(11)$$

&

$$\text{dupper} = y_{k+1} - y$$
$$= y_{k+1} - m(x_{k+1}) - b \quad -(12)$$

To determine which of the two pixels is closest to the line path, we can set up an efficient test that is based on the difference between the two pixel separations.

$$\text{dlower} - \text{dupper} = 2m(x_{k+1}) - 2y_k + 2b - 1 \quad -(13)$$

A decision parameter p_k for the k^{th} step in the line algorithm can be obtained by rearranging eqn (13) so that it involves only integer calculations.

Substitute $m = \Delta y / \Delta x$

$$p_k = \Delta x(\text{dlower} - \text{dupper})$$
$$= 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + c \quad -(14)$$

The sign of p_k is the same as the sign of dlower - dupper, since $\Delta x > 0$ in our example. Parameter c is constant and has the value $2\Delta y + \Delta x(2b-1)$, which is independent of the pixel position and will be eliminated in the recursive calculations for p_k.
If the plot at y_k is closer to the line path than the pixel at y_{k+1} (dlower < dupper), then the decision parameter p_k is negative, in this case we plot the lower pixel, otherwise we plot the upper pixel.
Co-ordinate changes along the line occur in unit steps in either the x or y directions.

$$\therefore p_{k+1} = 2\Delta y\, x_{k+1} - 2\Delta x\, y_{k+1} + c$$

Subtracting (14) from above equation.

$$p_{k+1} - p_k = 2\Delta y (x_{k+1} - x_k) - 2\Delta x (y_{k+1} - y_k)$$

But $x_{k+1} = x_k + 1$

$$\Rightarrow p_{k+1} = p_k + 2\Delta y - 2\Delta x (y_{k+1} - y_k) \quad (15)$$

where $y_{k+1} - y_k$ is either 0 or 1 depending on the sign of parameter p_k.
This recursive calculation of decision parameters is performed at each integer x position, starting at the left coordinate endpoint of the line.
The first parameter p_0 is evaluated from eq" (14) at the starting pixel position (x_0, y_0) and with m evaluated as $\Delta y / \Delta x$

$$p_0 = 2\Delta y - \Delta x \quad (16)$$

Constants $2\Delta y$ and $2\Delta y - 2\Delta x$ are calculated once for each line to be scan converted.

Algorithm:

1. Input the two line endpoints and store the left endpoint in (x_0, y_0).
2. Set the color for frame buffer position (x_0, y_0) i.e plot the first point.
3. Calculate the constants $\Delta x, \Delta y, 2\Delta y$ and $2\Delta y - 2\Delta x$ and obtain the starting value for the decision parameter as
$$p_0 = 2\Delta y - \Delta x$$
4. At each x_k along the line starting at $k=0$, perform the following test. If $p_k < 0$, the next point to plot is (x_{k+1}, y_k) and
$$p_{k+1} = p_k + 2\Delta y$$
otherwise the next point to plot is $(x_k + 1, y_k + 1)$ and
$$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$
5. Perform step (4) $\Delta x - 1$ times.

Example: Refer Hearn & Baker.

```
#include <stdlib.h>
#include <math.h>
// m < 1
void lineBres (int x0, int y0, int xend, int yend)
{
  int dx = fabs (xend - x0), dy = fabs (yend - y0);
  int p = 2 * dy - dx;
  int twoDy = 2 * dy, twoDyMinusDx = 2 * (dy - dx);
  int x, y;
  if (x0 > xend)
  {
    x = xend;
    y = yend;
    xend = x0;
  }
  else
  {
    x = x0;
    y = y0;
  }
  setPixel (x, y);
  while (x < xend)
  {
    x++;
    if (p < 0)
      p += twoDy;
    else
    {
      y++;
      p += twoDyMinusDx;
    }
    setPixel (x, y);
  }
}
```

OpenGL Point Functions.

Examples: (1)
```
glBegin(GL_POINTS); // draws a point on screen.
    glVertex*();
glEnd();
```
* indicates the suffix codes required for this function

(2)
```
glBegin(GL_POINTS);
    glVertex2i(50,100);
    glVertex2i(75,100);
    glVertex2i(100,200);
glEnd();
```

(3)
```
int p1[] = {50,100};
    p2[] = {75,100};
    p3[] = {100,200};
```
⇒
```
glBegin(GL_POINTS);
    glVertex2iv(p1);
    glVertex2iv(p2);
    glVertex2iv(p3);
glEnd();
```

Similarly we can use glVertex2f(), glVertex2fv(); glVertex3i(), glVertex3iv() etc for corresponding co-ordinate types.

OpenGL Line Functions:

→ GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
etc.

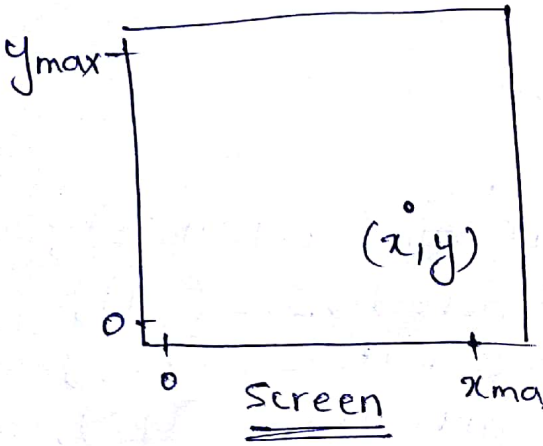
Examples:
```
glBegin(*);
    glVertex_();
    .
    .
    .
    glVertex_();
glEnd();
```

* lines are drawn between adjacent vertices.

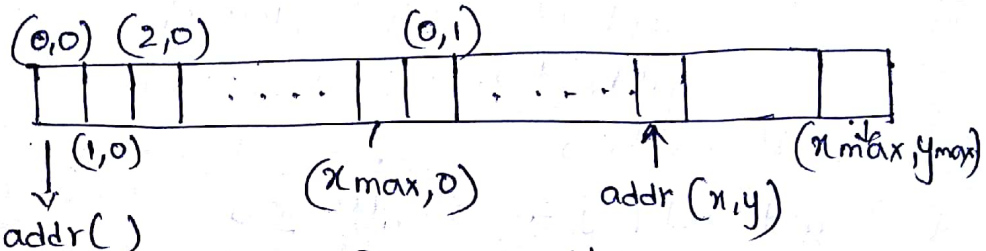
** Use all the above functions and parameters in * programs and view the output.

Setting Frame buffer values:

A final stage in the implementation procedures for line segments and other objects is to set the frame buffer color values. Since scan-conversion algorithms generate pixel positions at successive unit intervals, incremental operations can also be used to access the frame buffer efficiently at each step of the scan conversion process.



Screen



Frame buffer.

★ row-major representation.

Frame buffer bit address for pixel position (x, y) is calculated as

★ $addr(x,y) = addr(0,0) + y(x_{max}+1) + x$

Similarly ★ $addr(x+1, y) = addr(x,y) + 1$

★ $addr(x+1, y+1) = addr(x,y) + x_{max} + 2$

OpenGL Curve Functions.

→ Routines for generating basic curves, such as circles and ellipses are not included as primitive functions in the OpenGL core library

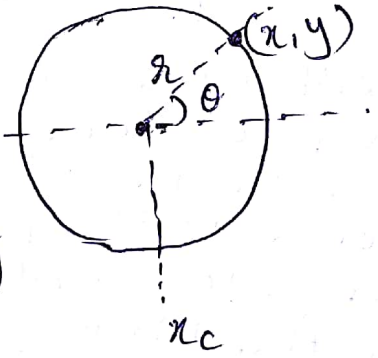
→ First method to use rational B-splines

→ Second method is to approximate a polyline. We need to locate a set of points along the curve path and connect the points with straight-line segments.

→ Third method is to write our own curve generation functions based on simple geometry.

Circle Generating Algorithms:

A circle is defined as the set of points that are all at a given distance r from the centre position (x_c, y_c). For any circle point (x, y), this distance relationship is expressed by Pythagorean theorem in cartesian co-ordinates as



$$(x - x_c)^2 + (y - y_c)^2 = r^2 \quad - (1)$$

$$y = y_c \pm \sqrt{r^2 - (x_c - x)^2} \quad - (2)$$

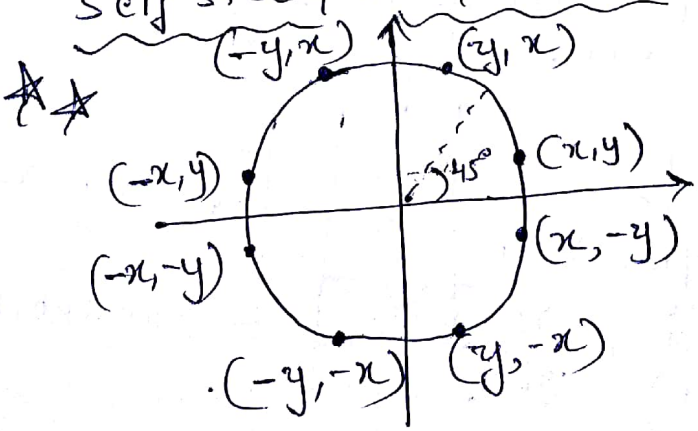
We can use eq" (2) to calculate the position of points on a circle's circumference by stepping along the x-axis in unit steps from $(x_c - r)$ to $(x_c + r)$.

This method involves considerable calculations at each step. Also the spacing between plotted pixel positions is not uniform. We could adjust the spacing by interchanging x and y whenever the absolute value of the slope of the circle is greater than 1.

Another way to eliminate the unequal spacing is to calculate points along the circular boundary using polar co-ordinates r and θ. Expressing the circle equation in parametric polar form yields a pair of expressions.

$$x = x_c + r\cos\theta \quad - (3)$$
$$y = y_c + r\sin\theta$$

Self study: symmetric point calculations.



Midpoint Circle Algorithm.

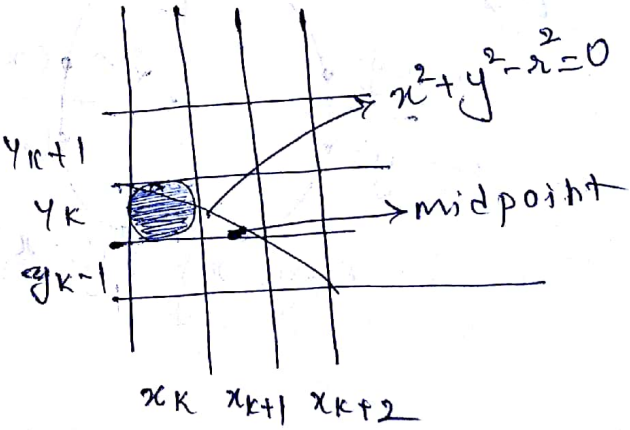
For a given radius r and screen centre position (x_c, y_c) we can first set up our algorithm to calculate pixel positions around a circle path centered at the co-ordinate origin $(0,0)$. Then each calculated (x,y) is moved to its proper screen position by adding x_c to x and y_c to y. Along the circle section from $x=0$ to $x=y$ in the first quadrant, the slope of the curve varies from 0 to -1.0. Therefore we can take unit steps in the positive x direction over this octant and use a decision parameter to determine which of the two possible pixel positions in any column is vertically closer to the circle path. Positions in the other seven octants are then obtained by symmetry

To apply midpoint method; we define the circle function as $f_{circ}(x,y) = x^2 + y^2 - r^2$ — (4)

Any point (x,y) on the boundary of the circle with radius r satisfies the equation $f_{circ}(x,y) = 0$. If the point is in the interior of the circle, the circle function is negative and if the point is outside the circle, the circle function is positive.

$$f_{circ}(x,y) = \begin{cases} <0 & \text{if } (x,y) \text{ is inside circle boundary} \\ =0 & \text{if } (x,y) \text{ is on the circle boundary.} \\ >0 & \text{if } (x,y) \text{ is outside the circle} \end{cases}$$ — (5)

The tests in eq^n (5) are performed for the midpositions between pixels near the circle path at each sampling step. Thus the circle function is the decision parameter in the midpoint algorithm and we can set up incremental calculations for this function as we did in the line algorithm.



As in the figure, at sampling position x_{k+1}, we assume to have plotted a pixel at (x_k, y_k) we next need to determine whether the pixel at position (x_{k+1}, y_k) or (x_{k+1}, y_{k-1}) is closer to the circle.

Our decision parameter is the circle function (4) evaluated at the midpoint between these two pixels.

$$p_k = f_{circ}(x_k + 1, y_k - 1/2)$$
$$= (x_k + 1)^2 + (y_k - 1/2)^2 - r^2 \quad -(6)$$

If $p_k < 0$, this midpoint is (less than 0) inside the circle and the pixel on scan line y_k is closer to the circle boundary, and we select the pixel on scan line $y_k - 1$.

Successive decision parameters are obtained using incremental calculations. we obtain a recursive expression for the next decision parameter by evaluating the circle function at sampling position $x_{k+1} + 1 = x_k + 2$:

$$p_{k+1} = f_{circ}(x_{k+1} + 1, y_{k+1} - 1/2)$$
$$= [(x_{k+1}) + 1]^2 + (y_{k+1} - 1/2)^2 - r^2$$

(or)
$$p_{k+1} = p_k + 2(x_k + 1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1 \quad -(7)$$

where y_{k+1} is either y_k or $y_k - 1$ depending on the sign of p_k.

Increments for obtaining p_{k+1} are either $2x_{k+1} + 1$ (if p_k is negative) or $2x_{k+1} + 1 - 2y_{k+1}$

Evaluation of the terms $2x_{k+1}$ and $2y_{k+1}$ can also be done incrementally as

$$2x_{k+1} = 2x_k + 2$$
$$2y_{k+1} = 2y_k - 2$$

At the start position $(0, r)$ these two terms have the values 0 and $2r$ respectively. Each successive value for the $2x_{k+1}$ term is obtained by adding 2 to the previous value and each successive value for the $2y_{k+1}$ term is obtained by subtracting 2 from the previous value.

The initial decision parameter is obtained by evaluating the circle function at the start position $(x_0, y_0) = (0, r)$:

$$p_0 = f_{circ}(1, r - 1/2)$$
$$= 1 + (r - 1/2)^2 - r^2$$
$$= \frac{5}{4} - r \quad \text{— (8)}.$$

If the radius r is specified as an integer, we can simply round p_0 to

$$p_0 = 1 - r \text{ as all increments are integers.}$$

Algorithm:

(1) Input radius r and circle centre (x_c, y_c), then set the co-ordinates for the first point on the circumference of a circle centered on the origin as $(x_0, y_0) = (0, r)$

(2) Calculate the initial value of the decision parameter as $p_0 = \frac{5}{4} - r$

(3) At each x_k position, stating at $k=0$, perform the following test. If $p_k < 0$ the next point along the circle centered on $(0,0)$ is (x_{k+1}, y_k) and $p_{k+1} = p_k + 2x_{k+1} + 1$.
Otherwise the next point along the circle is $(x_{k+1}, y_k - 1)$ and $p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$
where $2x_{k+1} = 2x_k + 2$ and $2y_{k+1} = 2y_k - 2$

(4) Determine symmetry points in the other seven octants.

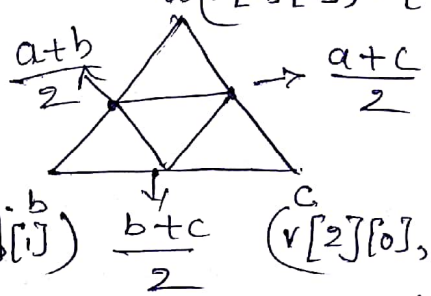
(5) Move each calculated pixel position (x, y) onto the circular path centered at (x_c, y_c) and plot the coordinate values:

$$x = x + x_c, \quad y = y + y_c$$

(6) Repeat steps (3) through (5) until $x >= y$

The output of 2D gasket shows a considerable structure. More the iterations, the randomness in the image would disappear. Examining this structure we see that regardless of how many points we generate, there are no points in the middle. Using this, we can subdivide each of these triangles into four triangles by connecting the mid points of the sides and each middle triangle will contain no points.

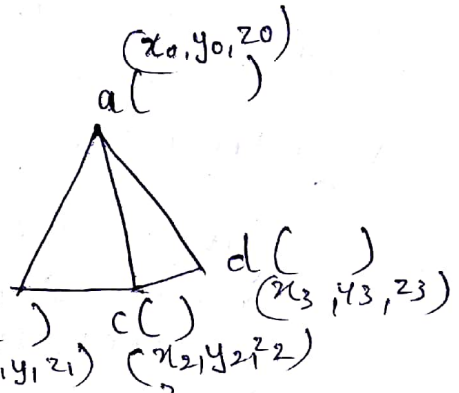
```
/* Triangle initial points */ GLfloat v[3][2];

void triangle (GLfloat *a, GLfloat *b, GLfloat *c)
{
    glVertex2fv (a);
    glVertex2fv (b);
    glVertex2fv (c);
}
```




```
void divide_triangle (GLfloat *a, GLfloat *b, GLfloat *c)
                      int k.
{
  GLfloat ab[2], ac[2], bc[2];
  int j;
  if (k > 0)
  {
   // Midpoints
  for (j=0; j<2; j++)
      { ab[j] = (a[j] + b[j]) / 2;
        ac[j] = (a[j] + c[j]) / 2;
        bc[j] = (b[j] + c[j]) / 2;
      }
  divide_triangle (a, ab, ac, k-1);
  divide_triangle (c, ac, bc, k-1);
  divide_triangle (b, bc, ab, k-1);
  }
  else
        triangle (a, b, c);
}
```

```
void display ()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_TRIANGLES);
        diride_triangle( v[0], v[1], v[2], n);
    glEnd();
    glFlush();
}
```

// write the main() function code.



Three Dimensional gasket.

Use 3D points as:

```
GLfloat vertices[4][3] = { {0.0, 0.0, 0.0},
                           {25.0, 50.0, 10.0},
                           {50.0, 25.0, 25.0},
                           {25.0, 10.0, 25.0} };
```

// short assignment to program 3D gasket with both the methods (random point & mid point)

Hidden Surface Removal.

Consider a tetrahedra [as above] we can only see those faces of the tetrahedra that were in front of all other faces as seen by a viewer.

* [more detail in later classes]
As of now we use Z-buffer algorithm. This is to be enabled and disabled accordingly.

* glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH); ← Z-depth.

&

* glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

Logical Input Devices.

Two major characteristics describe the logical behaviour of an input device.

(i) Measurements that the device returns to the user program

(ii) The time when the device returns those measurements.

→ Six classes.

1) String: A string device is a logical device that provides ASCII strings to the user program. This logical device is usually implemented by means of a physical keyboard.

2) Locator: A locator device provides a position in world co-ordinates to the user program. It is usually implemented by means of a pointing device such as a mouse or a trackball.

3) Pick: A pick device returns the identifier of an object on the display to the user program. It is usually implemented with the same physical device as a locator, but has a separate software interface to the user program (selection in OpenGL)

4) Choice: Choice devices allow the user to select one of a discrete number of options. (widgets in OpenGL)

5) Valuators: Valuators provide analog input to the user program. (slidebars, radioboxes)

6) Stroke: A stroke device returns an array of locations. Although we can think of a stroke device as similar to multiple uses of a locator, it is often implemented such that an action say pushing down a mouse button starts the transfer of data into a specified array and a second action such as releasing the button ends this transfer.

* Input Modes: Self study [Edward Angel]

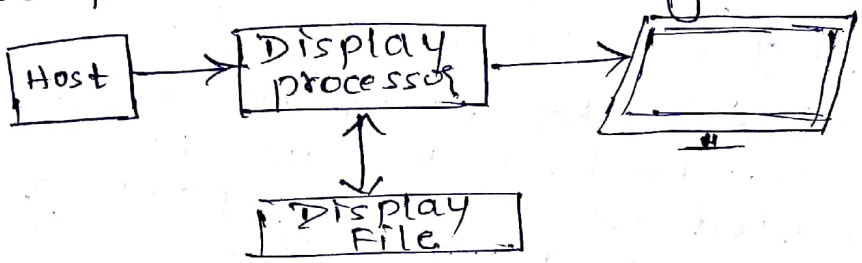
→ Request Mode

→ Sample Mode

→ Event-Mode model.

Display Lists.

OpenGL application programs are clients that use the graphics server.
Display lists illustrate how we can use clients and servers on a network to improve interactive graphics performance.
In earlier days the computer would send out necessary information to redraw the display at a rate sufficient to avoid noticeable flicker.
The solution to this problem was to build a special purpose computer called a display processor.



The display processor had a limited instruction set, most of which was oriented towards drawing primitives on the display. The user program was processed in the host computer, resulting in a compiled list of instructions that was then sent to the display processor, where the instructions were stored in a display memory as a display file, or display list. For a simple non-interactive application once the display list was sent to the display processor, the host was free for other tasks and the display processor would execute its display list repeatedly at a rate sufficient to avoid flicker.
In addition to resolving the bottleneck due to budening the host, the display processor introduced the advantage of special-purpose rendering hardware.
Today the display processor of old has become a graphics server and the application program on the host comp has become a client. The major bottleneck is no longer the rate at which we have to refresh the display, but rather the amount of traffic that passes between the client and server.
Graphical entities can be displayed in two modes.

→ Immediate mode

→ Retained mode.

In the immediate mode, as soon as the program executes a statement that defines a primitive, that primitive is sent to the server for possible display and no memory of it is retained in the system. To redisplay the primitive after clearing of the screen, or in a new position after an interaction, the program must respecify the primitive and then must resend the information through the display process.

In the retained mode, the object is defined once and its description is put on the display list. The display list is stored in the server and redisplayed by a simple function call issued from the client to the server.
The display lists need memory on the server and there is the overhead of creating a display list.

Definition and Execution of Display Lists:

```
glNewList(BOX, GL_COMPILE);   sends info to server but not to display
    glBegin(GL_POLYGON);
        glColor3f(1.0, 0.0, 0.0);
        glVertex2f(-1.0, -1.0);
        glVertex2f(1.0, -1.0);
        glVertex2f(1.0, 1.0);
        glVertex2f(-1.0, 1.0);
    glEnd();
glEndList();
```

#define BOX : Each display list should have an unique identifier.

→ GL_COMPILE_AND_EXECUTE can be used for immediate display.

→ glCallList(BOX) → to draw the box each time we call this function.

Self study: [Edward Angel]

glPushAttrib(); glPushMatrix();
glPopAttrib(); glPopMatrix();
Texts & Display lists, Fonts in GLUT

Programming Event Driven Input:

→ Using the Pointing Device:

- ✓ Move event is generated when the mouse is moved with one of the buttons pressed.
- ✓ If the mouse is moved without a button being held down, this event is called a passive mouse event.
- ✓ After a move event, the position of the mouse - its measure is made available to the application program.
- ✓ A mouse event occurs when one of the mouse buttons is either pressed or released.

Mouse callback function: glutMouseFunc(myMouse);

void myMouse (int button, int state, int x, int y);

Ex:

```
void myMouse (int button, int state, int x, int y)
{
    if (button == GLUT_LEFT_BUTTON &&
        state == GLUT_DOWN)
        exit(0);
}

int main (int argc, char **argv)
{
    /* All required functions as
       before */

    glutMouseFunc (myMouse);
    glutDisplayFunc (myDisplay);
    glutMainLoop();

}
```

→ Window Events:

Most window systems allows a user to resize the window interactively usually by using the mouse to drag a corner of the window to a new location. This event is an example of a window event. When a mouse event occurs we have to decide:

→ Do we redraw all the objects that were in the window before it was resized?

→ What do we do if the aspect ratio of the new window is different from that of the old window?

→ Do we change the sizes or attributes of new primitives if the size of the new window is different from that of the old?

As a solution we can create a reshape event that returns in its measure the height and width of the new window.

```
Void myReshape (GLsizei w, GLsizei h)
{
    // clipping Box adjustment
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluOrtho2D (0.0, (GLdouble) w, 0.0, (GLdouble) h);
    glMatrixMode (GL_MODELVIEW);
    glLoadIdentity ();
    // viewport & clear
    glViewport (0, 0, w, h);
    ww = w;
    wh = h;
}
```

Self study.

```
glutMotionFunc (<action>);
```

→ Keyboard Functions.

```
glutKeyboardFunc(mykey);

void mykey(unsigned char key, int x, int y)
{
    if (key == 'q' || key == 'Q')
        exit();
}
```

→ Display & Idle callbacks.

```
glutDisplayFunc(myDisplay);   // function name.
glutPostRedisplay();  // only once each time the program goes through the event loop.
```

The idle callback is invoked when there is no other events. Its default is the null function pointer. A typical use of the idle callback is to continue to generate graphical primitives through a display function while nothing else is happening.

→ Window Management

To open multiple windows & subwindows.

Ex
```
id = glutCreateWindow("second window");
glutSetWindow(id);
```

Menus

GLUT provides one additional feature, pop-up menus that we can use with the mouse to create sophisticated interactive applications.

→ First define the actions corresponding to each entry in the menu.

→ Link the menu to a particular mouse button.

→ Register a callback function for each menu.

Using menus involves taking a few simple steps
→ define the action corresponding to each entry in the menu.
→ link the menu to a particular mouse button.
→ register a callback function for each menu] repeated

Ex:-
```
glutCreateMenu (demo_menu); // callback demo_menu()
glutAddMenuEntry ("quit", 1);
glutAddMenuEntry ("increase square size", 2);
glutAddMenuEntry ("decrease square size", 3);
glutAttachMenu (GLUT_RIGHT_BUTTON);
```

```
void demo_menu (int id)        // GLint (also can be used)
{
  switch (id)
  {
    case 1 : exit(0);
             break;
    case 2 : size = 2*size;
             break;
    case 3 : if (size>1) size=size/2; break;
  }
  glutPostRedisplay(); // redraws screen from
                       //glutDisplayFunc() callback
}
```

* menu_id = glutCreateMenu (demo_menu);
  A newly created menu becomes the current menu for the current display window.

* glutSetMenu (menu_id); → activates a menu for the current display window.

* glutDestroyMenu (menu_id); → eliminates a menu.

* current_menu_id = glutGetMenu(); → id of current menu.
* glutDetachMenu (mouseButton);
* glutRemoveMenuItem (itemNumber);] self study

Creating GLUT submenus (Hierarchical menus)

```
submenuID = glutCreateMenu (submenuFunc);
    glutAddMenuEntry ("First submenu item", 1);
        .
        .
        .
glutCreateMenu (menu_func);
    glutAddMenuEntry ("First Menu item", 1);
        .
        .
        .
    glutAddSubMenu ("SubMenu Option", submenuID);
```

Ex:-

```
void mainmenu (GLint id)
{
    switch (id)
    {
        case 1 : exit (0)
                 break;
        case 2 : size = size * 2; break.
        case 3 : if (size > 1) size = size / 2;
                 break;
    }   glutPostRedisplay ();
}
void colorsubmenu (GLint coloroption)
{   switch (coloroption)
    {
        case 1 : red = 0; green = 0; blue = 1;
                 break;
        case 2 : red = 0; green = 1; blue = 0;
                 break;
        case 3 : red = 1; green = 1; blue = 1;
                 break;
    }   glutPostRedisplay ();
}
```

OpenGL Point Attribute Functions.

glPointSize(size); → set size of point.
↳ size * size pixel array.

Ex:
```
glColor3f (1, 0, 0);
glBegin (GL_POINTS);
    glVertex2f (50, 100);
    glPointSize (2.0);
    glColor3f (0, 1, 0);
    glVertex2i (75, 100);
    glPointSize (3.0);
    glColor3f (0, 0, 1);
    glVertex2i (100, 200);
glEnd ();
```

OpenGL Line Attribute Functions:

glLineWidth (width);
glLineStipple (repeatFactor, pattern);
- pattern: series of 0's & 1's indicating OFF, ON
- repeatFactor: how many times each bit in pattern is to be displayed.

glShadeModel (GL_SMOOTH); ↳ default
or
GL_FLAT ↳ end point color is used.

★★ Self study - Character Attribute Functions
★ → Bitmap
→ Stroke.

Antialiasing

→ jagged staircase appearance because the sampling process digitizes co-ordinate points on an object to discrete integer pixel positions.

→ This distortion of information due to low frequency display of a high resolution primitive is called aliasing.

→ The counter measure to improve the appearance of displayed primitives is called antialiasing.

Consider a periodic generation of a signal.

*Losing information can be the effects of undersampling To avoid this we have Nyquist sampling frequency which suggests

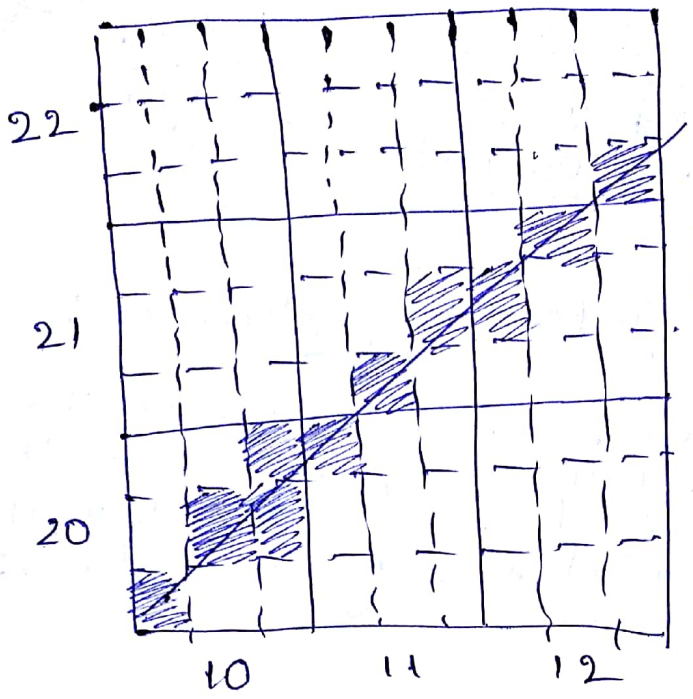
$$f_s = 2 f_{max}$$

Similarly the sampling interval

$$\Delta x_s = \frac{\Delta x_{cycle}}{2} \quad \text{where } \Delta x_{cycle} = 1/f_{max}$$

→ One way to increase sampling rate with raster scan systems is simply to display objects at higher resolution.
But there is a limit to how big the frame buffer can be and also refresh rate should be higher.

(i) supersampling: Also called postfiltering involves computing intensities at subpixel grid positions then combining the results to obtain the pixel intensities. Hence sampling is at high (frequency) resolution and display is with low resolution.
An alternative to supersampling is to determine pixel intensity by calculating the areas of overlap of each pixel with the objects to be displayed.
This is called area sampling or prefiltering..
We can perform supersampling by subdividing each pixel into subpixels and count the number of subpixels that overlap the line path.
[In case of straight line segments].



(★ Further explanation Hearn & Baker)

1	2	1
2	4	2
1	2	1

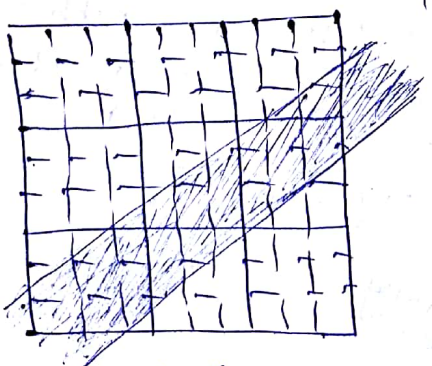
- Mask

(ii) Subpixel weighting Masks.

Supersampling algorithms are often implemented by giving more weight to subpixels near the center of a pixel area since we would expect these subpixels to be more important in determining the overall intensity of a pixel. (★)

(iii) Area sampling straight-Line segments.

We perform area sampling of a straight line by setting pixel intensity proportional to the area of overlap of the pixel with the finite-width line. (★)



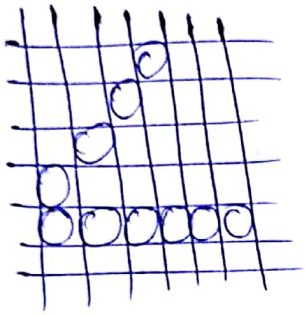
(iv) Filtering techniques: (★)

- Box filter
- Cone filter
- Gaussian filter.

(v) Pixel Phasing: A line display is smoothed with this technique by moving (micropositioning) pixel positions close to the line path.

(A).

(vi) Compensating for line Intensity Differences:



Both lines are plotted with the same number of pixels yet the diagonal line is longer than the horizontal line by a factor $\sqrt{2}$

(A).

(vii) Antialiasing Area Boundaries:

Midpoint line algorithm concept.
Slope m in the range 0 to 1.

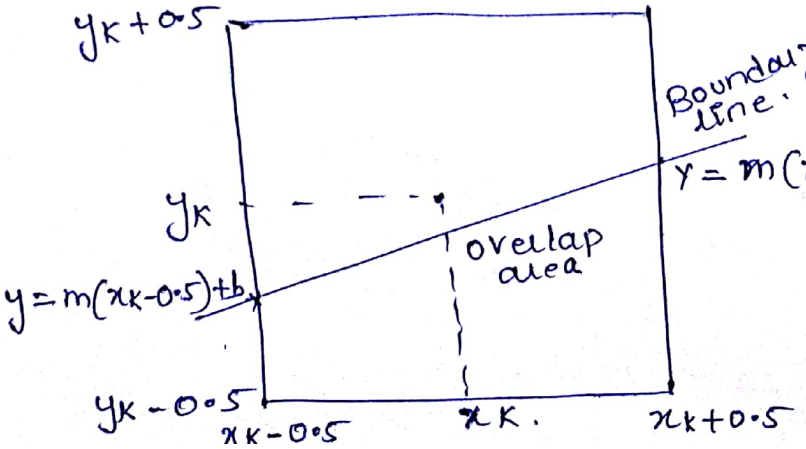
$$y - y_{mid} = [m(x_k+1) + b] - (y_k + 0.5).$$

If this calculation results in negative value, then pixel y_k is closer to the line, if it is positive then y_{k+1} is closer to the line.
We add quantity $1-m$ to produce a positive quantity

$$\therefore p = [m(x_k+1) + b] - (y_k + 0.5) + (1-m)$$

Parameter p also measures the amount of the current pixel that is overlapped by the area.
For pixel at (x_k, y_k), the interior part of the pixel has an area that can be calculated as.

$$\text{area} = m x_k + b - y_k + 0.5$$



UNIT-3 - Study Guide.

GEOMETRIC TRANSFORMATIONS

Basic two dimensional geometric transformations.

- → Translation
- → Rotation
- → Scaling.
- → Reflection
- → Shearing

[** All topics covered in this * unit are from Heaun & Baker * Further reading is suggested.]

Two Dimensional translation

Translation is performed by adding offsets to the original co-ordinates to generate a new co-ordinate position.

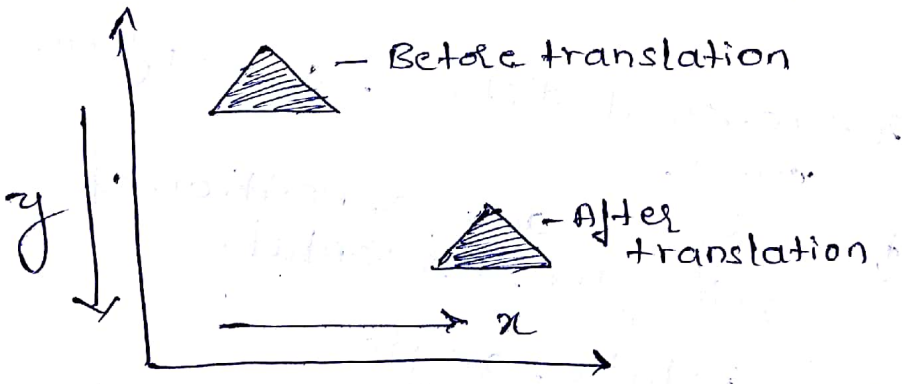
The offset distances are called translation distance. t_x and t_y (for x & y co-ordinates) to obtain new co-ordinate position (x', y')

$$x' = x + t_x \qquad y' = y + t_y.$$

(t_x, t_y) is called the translation vector or shift vector.

$$P = \begin{bmatrix} x \\ y \end{bmatrix} \qquad P' = \begin{bmatrix} x' \\ y' \end{bmatrix} \qquad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$P' = P + T.$$



Two-Dimensional Rotation:

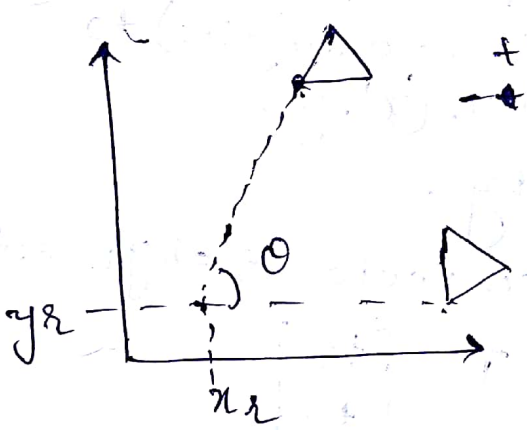
A rotation transformation of an object is generated by specifying a rotation angle and a rotation axis. All points of the object are then transformed to new positions by rotating the points through the specified angle about the rotation axis.

A two-D rotation of an object is obtained by repositioning the object along a circular path in the xy plane.

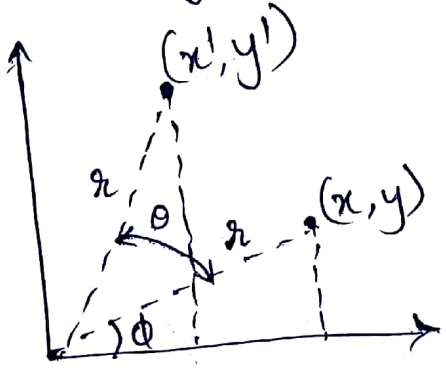
Consider we are rotating around an axis perpendicular to the xy plane. (parallel to the co-ordinate z-axis)

Rotation angle - θ

Rotation point or pivot point (x_r, y_r)



$$x' = r\cos(\phi+\theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$
$$y' = r\sin(\phi+\theta) = r\cos\phi\sin\theta + r\sin\phi\cos\theta$$



r → constant distance of point from origin

ϕ → original angular position of point from horizontal.

θ → rotation angle.

Polar co-ordinates:

$$x = r\cos\phi \quad y = r\sin\phi$$

substituting in above eq^n^s.

$x' = x\cos\theta - y\sin\theta$

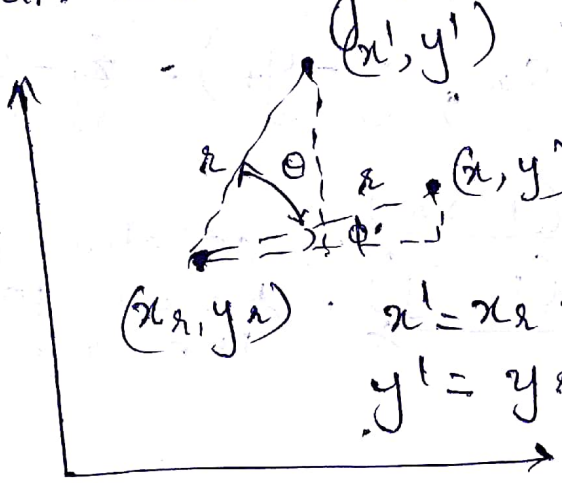
$y' = x\sin\theta + y\cos\theta$

Column vector representation

$P' = R \cdot P$

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \quad P = \begin{bmatrix} x \\ y \end{bmatrix}$$

At an arbitrary point.



$x' = x_r + (x - x_r)\cos\theta - (y - y_r)\sin\theta$

$y' = y_r + (x - x_r)\cos\theta - (y - y_r)\cos\theta$

Two-D scaling.

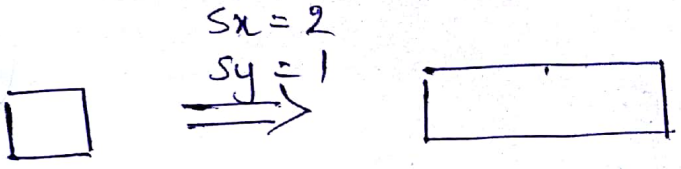
Scaling transformation alters the size of an object.

A simple scaling operation in 2-D is performed by multiplying object positions (x, y) with scaling factors s_x and s_y to produce (x', y').

$x' = x \cdot s_x \quad y' = y \cdot s_y$

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \quad P' = S \cdot P$$

If s_x & s_y are equal it is uniform scaling otherwise called differential scaling.



Scaling factors with absolute values less than one moves objects closer to the co-ordinate origin, while absolute values greater than 1 move co-ordinate positions farther from the origin.

We can control the location of a scaled object by chosing a position called the fixed point that is to remain unchanged after the scaling transformation. (x_f, y_f).

$$(x' - x_f) = (x - x_f) \cdot Sx, \quad (y' - y_f) = (y - y_f) \cdot Sy$$

$$x' = x \cdot Sx + x_f(1 - Sx)$$
$$y' = y \cdot Sy + y_f(1 - Sy)$$

→ constant points for all points in the object.

Matrix Representations and Homogeneous Co-ordinates.

General Matrix form

$$P' = M_1 P + M_2$$

P' and P are column matrices, M_1 is a 2 × 2 array containing multiplicative factors and M_2 is a two-element column matrix.
For translation M_1 is identity matrix
For rotation or scaling M_2 will have the translational terms with the pivot point or scaling fixed point.
The order of transformations are first scaling then co-ordinates are rotated and finally translated.
A better approach is to combine all the transformations so that the final co-ordinate positions are obtained directly without intermediate co-ordinate values.

Homogeneous co-ordinates.

If we use a 3 × 3 matrix representation to combine all transformation then the last column of the matrix can be for all translation terms.
Homogeneous co-ordinates expands 2D representation (x, y) to three element representation (x, y, h) where h is non-zero

$$x = \frac{xh}{h} \qquad y = \frac{yh}{h}$$

hence the orginal homogeneous representation is $(x \cdot h, y \cdot h, h)$
There can be any number of homogenous representations for any non-zero h value.
Conveniently we pick. $h = 1$. $\Rightarrow (x, y, 1)$.

Translation Matrix. (2D)

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$P' = T(tx, ty) \cdot P.$$

Rotation Matrix (2D)

$$P' = R(\theta) \cdot P.$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scaling Matrix (2D)

$$P' = S(sx, sy) \cdot P.$$

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

Inverse translations:

$$T^{-1} = \begin{bmatrix} 1 & 0 & -tx \\ 0 & 1 & -ty \\ 0 & 0 & 1 \end{bmatrix} \qquad S^{-1} = \begin{bmatrix} 1/sx & 0 & 0 \\ 0 & 1/sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2D composite Transformations.

$$P' = M_2 \cdot M_1 \cdot P$$

For translations,

2 successive translations.

$$P' = T_2 \cdot T_1 \cdot P$$

$$P' = T(t_{2x}, t_{2y}) \cdot \{ T(t_{1x}, t_{2y}) \cdot P \}$$

$$= \begin{bmatrix} 1 & 0 & t_{2x} \\ 0 & 1 & t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & t_{1x} \\ 0 & 1 & t_{1y} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{1x}+t_{2x} \\ 0 & 1 & t_{1y}+t_{2y} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\therefore P' = T(t_{1x}+t_{2x}, t_{1y}+t_{2y}) \cdot P.$$

For rotations,

2 successive rotations,

$$P' = R(\theta_2) \cdot \{ R(\theta_1) \cdot P \}$$

$$= R(\theta_2) \cdot R(\theta_1) \cdot P.$$

Prove that $P' = R(\theta_1 + \theta_2) \cdot P$.

For Scaling, calculate the 2 successive scaling operations on P.

General 2D Pivot Point Rotation :-

(1) Translate the object so that the pivot-position is moved to the co-ordinate origin

(2) Rotate the object about the co-ordinate origin,

(3) Translate the object so that the pivot point is returned to its original position.

$$T(x_r, y_r) \cdot R(\theta) \cdot T(-x_r, -y_r) = R(x_r, y_r, \theta)$$

★ Obtain the matrix form.

$$\begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta)+y_r\sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta)-x_r\sin\theta \\ 0 & 0 & 1 \end{bmatrix}$$

General Two-D fixed Point scaling.

(1) Translate the object so that the fixed point coincides with the co-ordinate origin.

(2) Scale the object with respect to the co-ordinate origin

(3) Use the inverse of the translation in step (1) to return the object to its original position.

Prove

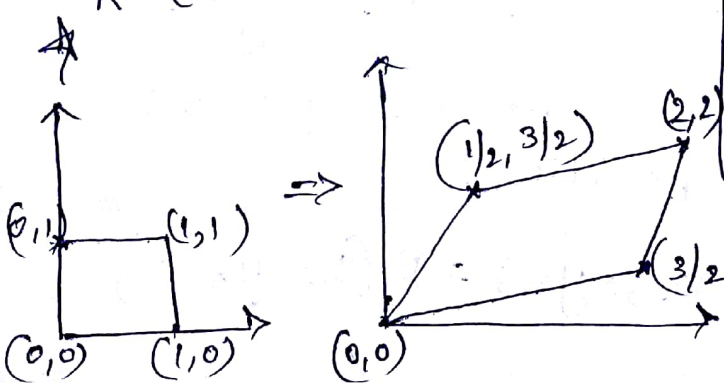
★ $S(x_f, y_f, s_x, s_y) = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}$

General Two Dimensional scaling Directions:

Parameters s_x and s_y scale objects along the x and y directions. We can scale an object in other directions by rotating the object to align the desired scaling directions with the co-ordinate axis before applying the scaling transformation.

$$R^{-1}(\theta) \cdot S(s_1, s_2) \cdot R(\theta) = \begin{bmatrix} s_1\cos^2\theta + \sin^2\theta & (s_2-s_1)\cos\theta\sin\theta & 0 \\ (s_2-s_1)\cos\theta\sin\theta & s_1\sin^2\theta + s_2\cos^2\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

★



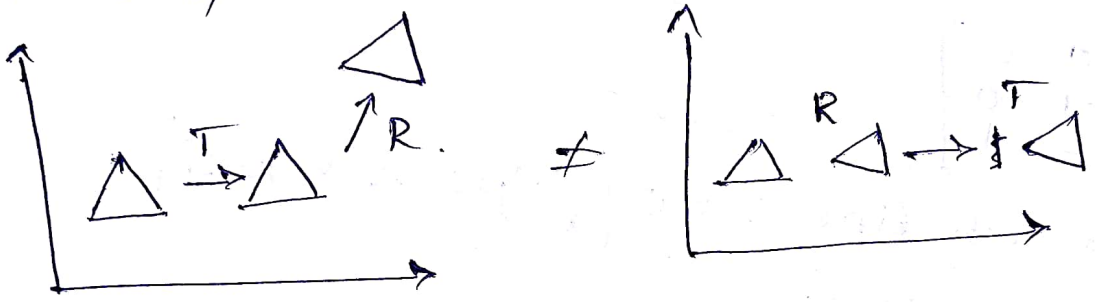
Matrix Concatenation Properties :-

$$M_3 \cdot M_2 \cdot M_1 = (M_3 \cdot M_2) \cdot M_1 = M_3 (M_2 \cdot M_1)$$

→ Associativity works.

Transformation products on the other hand are not commutative.

$$T * R \neq R * T$$



General 2-D Composite Transformations and computational efficiency.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} rsxx & rsxy & trsx \\ rsyx & rsyy & trsy \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$\Rightarrow T(tx, ty) * R(xc, yc, \theta) * S(xc, yc, sx, sy)$$

Two-D Rigid Body Transformation :

Only translation & Rotation transformations applied are called rigid body transformation [Rigid motion].

$$M_R = \begin{bmatrix} r_{xx} & r_{xy} & trx \\ r_{yx} & r_{yy} & try \\ 0 & 0 & 1 \end{bmatrix}$$

★ Self study.

Reflection:-

A transformation that produces a mirror image of an object is called reflection, and for 2D reflection this image is generated relative to an axis of reflection by rotating the object 180° about the reflection axis.

Reflection about the line $y=0$ (x-axis) [flips y value]

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Reflection about line $x=0$ [flips x value.]

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Flip both x & y co-ordinates

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Diagonal reflection about $y=x$ line.

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

* clockwise 45° rotation
* reflection about x axis
* counterclockwise 45° rotation

for $y=-x$ line.

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

* clockwise 45°
* reflection about y axis
* counterclockwise 45°.

Shear.

A transformation that distorts the shape of an object such that the transformed shape appears as if the object were composed of internal layers that had been caused to slide over each other is called a shear.

x-direction shear

$$\begin{bmatrix} 1 & shx & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

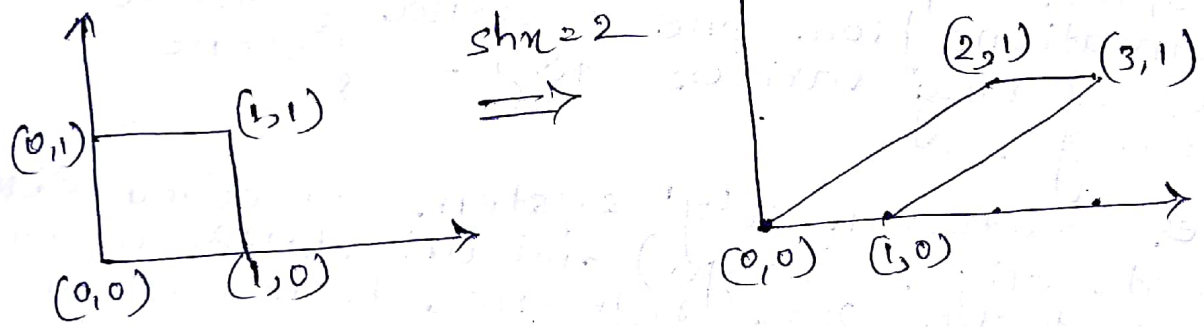
$$\Rightarrow x' = x + shx * y$$
$$y' = y$$

y-direction shear

$$\begin{bmatrix} 1 & 0 & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

$$x' = x$$
$$y' = y + shy * x$$

Shear shx value means, a co-ordinate position (x, y) is then shifted horizontally by an amount proportional to its perpendicular distance (y value) from the x axis.



Relative shear operation x-direction.

$$\begin{bmatrix} 1 & shx & -shx * y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad x' = x + shx(y - y_{ref}), \quad y' = y$$

Relative shear operation y-direction.

$$\begin{bmatrix} 1 & 0 & 0 \\ shy & 1 & -shy * x_{ref} \\ 0 & 0 & 1 \end{bmatrix} \quad x' = x, \quad y' = y + shy(x - x_{ref})$$

* Self study - Raster methods for geometric transformations

Opengl raster transformations:

```
glCopyPixels (xmin, ymin, width, height, GL_COLOR);
glReadPixels (xmin, ymin, width, height, GL_RGB,
              GL_UNSIGNED_BYTE, colorArray);
glDrawPixels (width, height, GL_RGB, GL_UNSIGNED_BYTE,
              colorArray);
glPixelZoom (sx, sy);
```

Transformations between two dimensional co-ordinate systems.

Graphics applications involve co-ordinate transformations from one reference frame to another during various stages of scene processing.

Consider cartesian x'y' system specified with co-ordinate origin (x_0, y_0) and orientation angle θ in a cartesian xy reference frame.

To transform object descriptions from xy co-ordinates to x'y' co-ordinates, we set up a transformation that superimposes the x'y' axes onto the xy axes as follows:

(a) Translate so that the origin (x_0, y_0) of the x'y' system is moved to the origin $(0,0)$ of the xy system.

(b) Rotate the x' axis onto the x axis.

$$T(-x_0, -y_0) = \begin{bmatrix} 1 & 0 & -x_0 \\ 0 & 1 & -y_0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$R(-\theta) = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$ → clockwise rotation

$$M = R(-\theta) \cdot T(-x_0, -y_0)$$

An alternate method for describing the orientation of the x'y' co-ordinate system is to specify a vector V that indicates the direction for the positive y' axis. We can specify vector V as a point in the xy reference frame relative to the origin of the xy system, which we convert to a unit vector.

$$v = \frac{V}{|V|} = (v_x, v_y).$$

we can obtain a unit vector u along the x' axis by applying a 90° clockwise rotation to vector v.

$$u = (v_y, -v_x)$$
$$= (u_x, u_y).$$

$$\therefore R = \begin{bmatrix} u_x & u_y & 0 \\ v_x & v_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Geometric transformations in three Dimensional space

Translation - 3D space.

$$x' = x + t_x, \quad y' = y + t_y, \quad z' = z + t_z.$$

$$P' = T \cdot P$$

$$T = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1. \end{bmatrix}$$

Three Dimensional Rotation (Co-ordinate Axis)

z-axis : $x' = x\cos\theta - y\sin\theta$
$y' = x\sin\theta + y\cos\theta$
$z' = z$.

x-axis : $y' = y\cos\theta - z\sin\theta$
$z' = y\sin\theta + z\cos\theta$
$x' = x$.

y-axis : $z' = z\cos\theta - x\sin\theta$
$x' = z\sin\theta + x\cos\theta$
$y' = y$.

★ Inverse 3D rotation matrix is obtained by just replacing θ by $-\theta$.

$R^{-1} = R^T$.

General 3D Rotation.

A rotation matrix for any axis that does not coincide with a co-ordinate axis can be set up as composite transformation involving combinations of translations and the co-ordinate axis rotations.

If an arbitrary axis is parallel to one of the co-ordinate axis, we can follow the steps.

(i) Translate the object so that the rotation axis coincides with the parallel coordinate axis.
(ii) Perform the specified rotation about that axis.
(iii) Translate the object so that the rotation axis is moved back to its original position.

$$P' = T^{-1} \cdot R_x(\theta) \cdot T \cdot P.$$

For non-parallel axis to co-ordinate axis.

(i) Translate the object so that the rotation axis passes through the co-ordinate origin
(ii) Rotate the object so that the axis of rotation coincides with one of the co-ordinate axes.

(iii) Perform the specified rotation about the selected co-ordinate axis

(iv) Apply inverse rotations to bring the rotation axis back to its original orientation.

(v) Apply the inverse translation to bring the rotation axis back to its original spatial position.

★ Further reading.

Three dimensional scaling:

$$P' = S \cdot P$$

$$S = \begin{bmatrix} sx & 0 & 0 & 0 \\ 0 & sy & 0 & 0 \\ 0 & 0 & sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

with respect to specific point (x_f, y_f, z_f)

$$T(x_f, y_f, z_f) \cdot S(sx, sy, sz) \cdot T(-x_f, -y_f, -z_f) = \begin{bmatrix} sx & 0 & 0 & (1-sx)x_f \\ 0 & sy & 0 & (1-sy)y_f \\ 0 & 0 & sz & (1-sz)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3D composite transformations are same as 2D composite transformations.

3D Reflection.

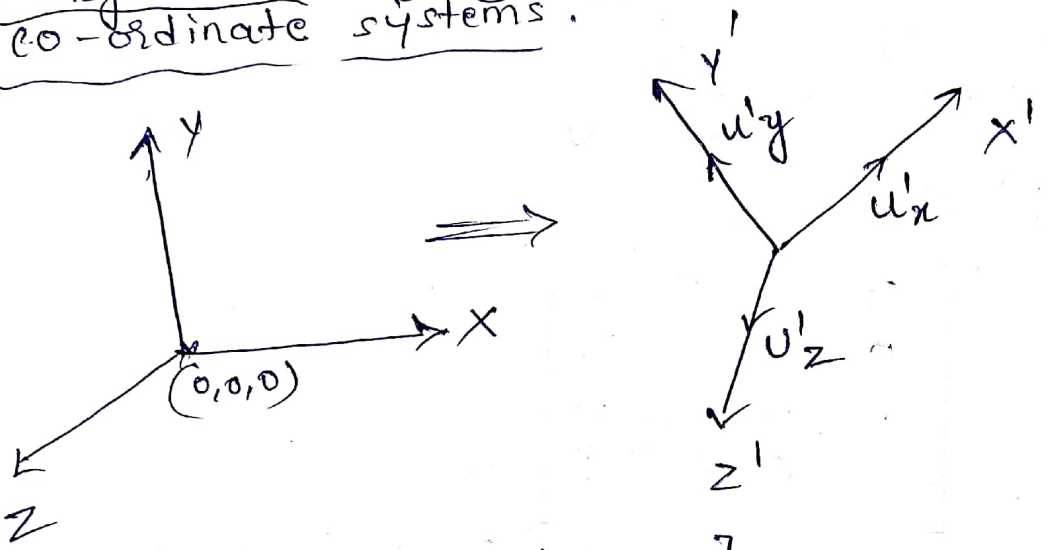
$$M_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

★ Similarly other reflections about x & y co-ordinates.

3D shears.

$$M_{zshear} = \begin{bmatrix} 1 & 0 & sh_{zx} & -sh_{zx} \cdot z_{ref} \\ 0 & 1 & sh_{zy} & -sh_{zy} \cdot z_{ref} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Transformations between three dimensional co-ordinate systems.



$$R = \begin{bmatrix} u'_{x1} & u'_{x2} & u'_{x3} & 0 \\ u'_{y1} & u'_{y2} & u'_{y3} & 0 \\ u'_{z1} & u'_{z2} & u'_{z3} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$M = R * T$.

$T(-x_0, -y_0, -z_0)$.

Affine Transformations.

$$x' = a_{xx}x + a_{xy}y + a_{xz}z + b_x$$
$$y' = a_{yx}x + a_{yy}y + a_{yz}z + b_y$$
$$z' = a_{zx}x + a_{zy}y + a_{zz}z + b_z.$$

Rotation, scaling, reflection, translation & shear are examples of affine transformations

It means after the transformation the object still preserves collinearity and ratio of distances Also parallel lines remain parallel and finite points map to finite points.

OpenGL Geometric transformation functions

glMatrixMode ( ) → to set projection mode.

for matrix mode in transformations we use

glMatrixMode (GL_MODELVIEW);

```
glColor3f (0.0, 0.0, 1.0);
glRecti (50, 100, 200, 150);
glColor3f (1.0, 0.0, 0.0);
glTranslatef (-200.0, -50.0, 0.0);
glRecti (50, 100, 200, 150);
glLoadIdentity ( ) // Reset.
glRotatef (90.0, 0.0, 0.0, 1.0);
                              → z axis.
glRecti (50, 100, 200, 150);
glLoadIdentity ( )
glScalef (-0.5, 1.0, 1.0);
glRecti (50, 100, 200, 150);
```

* glPushMatrix ( ) → Make a copy of identity Matrix
  glPopMatrix ( )

VIEWING

Classical Viewing.

When an architect draws an image of a building, she knows which side she wishes to display and thus where the viewer should be placed in relationship to the building.

In classical viewing, there is an underlying notion of a principal face. The types of objects viewed in real world applications, such as architecture, tend to be composed of a number of planar faces, each of which can be thought of as a principal face.

→ Orthographic projections: Projectors are perpendicular to the projection plane.

→ Axonometric Projections: The projectors are still orthogonal to the projection plane, if the projection plane is placed symmetrically w.r.t the three principal faces that meet at a corner of our rectangular object, then we have isometric view, if the projection plane is placed symmetrically w.r.t two of the principal faces, then the view is dimetric, the general case is a trimetric view.
In isometric view, a line segment's length in the image space is shorter than its length measured in the object space. This foreshortening of distances is the same in the three principal directions.
In the dimetric view however there are two different foreshortening ratios, in trimetric there are three.
Axonometric views are extensively used in architectual and mechanical design.

→ Oblique projections: We obtain oblique projections by allowing the projectors to make an arbitrary angle with the projection plane.

→ Perspective Viewing: All perspective views are characterized by diminution of size. When objects are moved further from the viewer, their images become smaller. This size change gives perspective views their natural appearance, however because the amount by which a line is foreshortened depends on how far the line is from the viewer, we cannot make measurements from the perspective view. Hence the major use of perspective views is in applications such as architecture and animation, where it is important to achive natural looking images.

Classical perspective views are usually known as one point, two point & three point perspectives.

The difference among the three cases are based on how many of the three principal directions in the object are paallel to the projection plane.

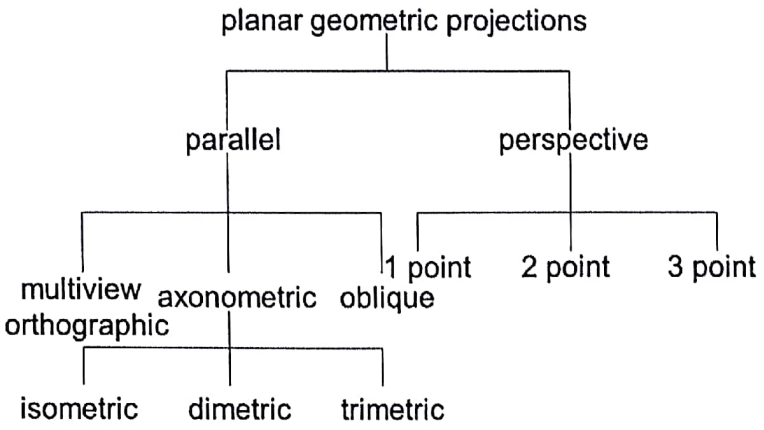
→ 3 point perspective — paallel lines in each of the three principal directions converges to a finite vanishing point.

→ 2 point perspective — only two of the three principal directions converges, one direction will be paallel to the projection plane.

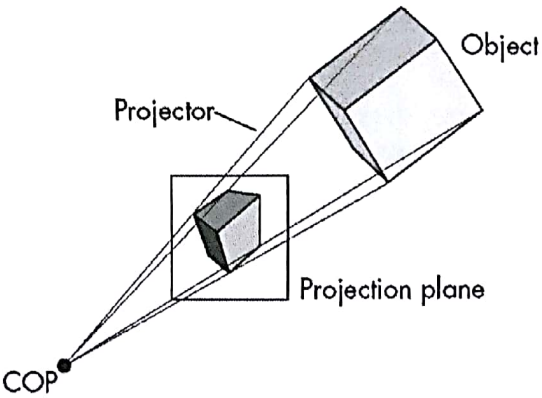
→ 1 point perspective — 2 principal directions are paallel to the projection plane and only one single vanishing point.

Classification of Planar Geometric Projections in Viewing

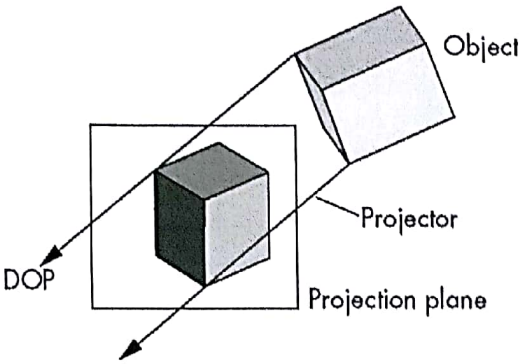
- Viewing requires three basic elements
  - One or more objects.
  - A viewer with a projection surface.
  - Projectors that go from the object(s) to the projection surface.
- Classical views are based on the relationship among these elements



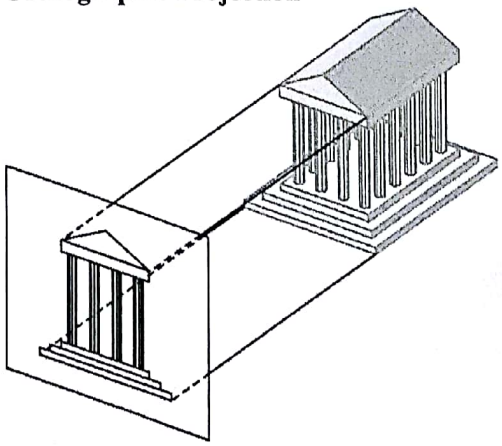
Perspective Projection



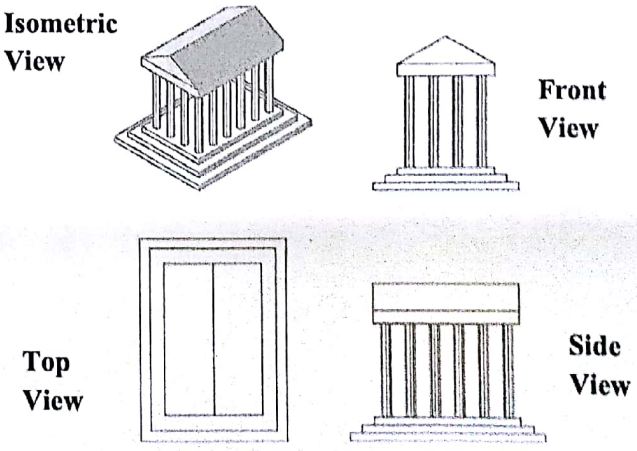
Parallel Projection



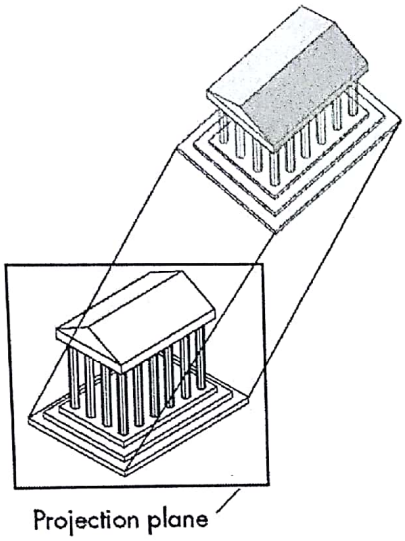
Orthographic Projection



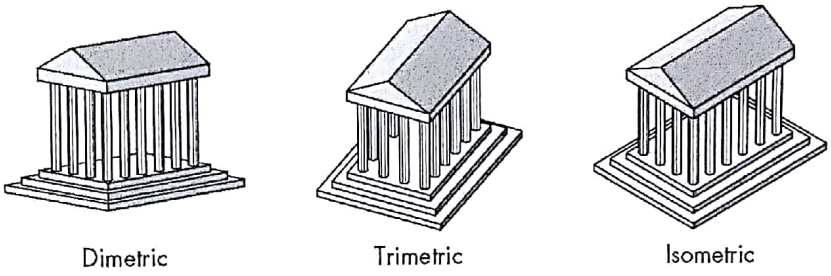
Multi-view Orthographic Projection



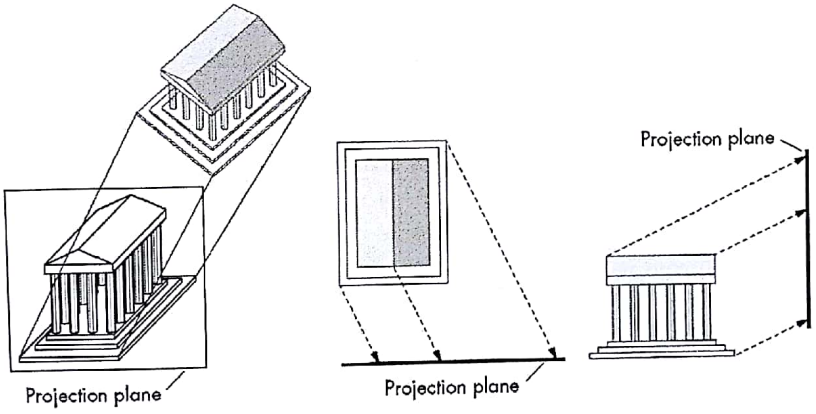
Axonometric Projection



Types of Axonometric projections



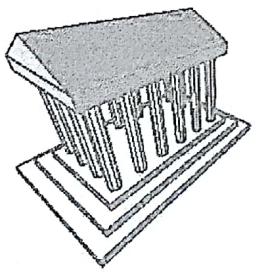
Oblique Projection



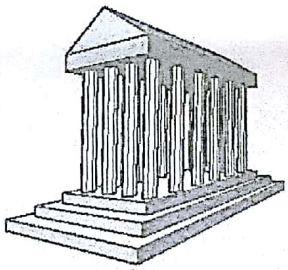
Vanishing Points

- Parallel lines (not parallel to the projection plan) on the object converge at a single point in the projection (the *vanishing point*)
- Drawing simple perspectives by hand uses these vanishing point(s)

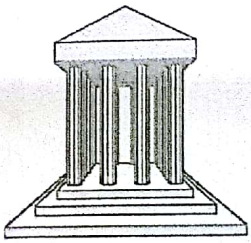
Three Point Perspective



Two Point Perspective

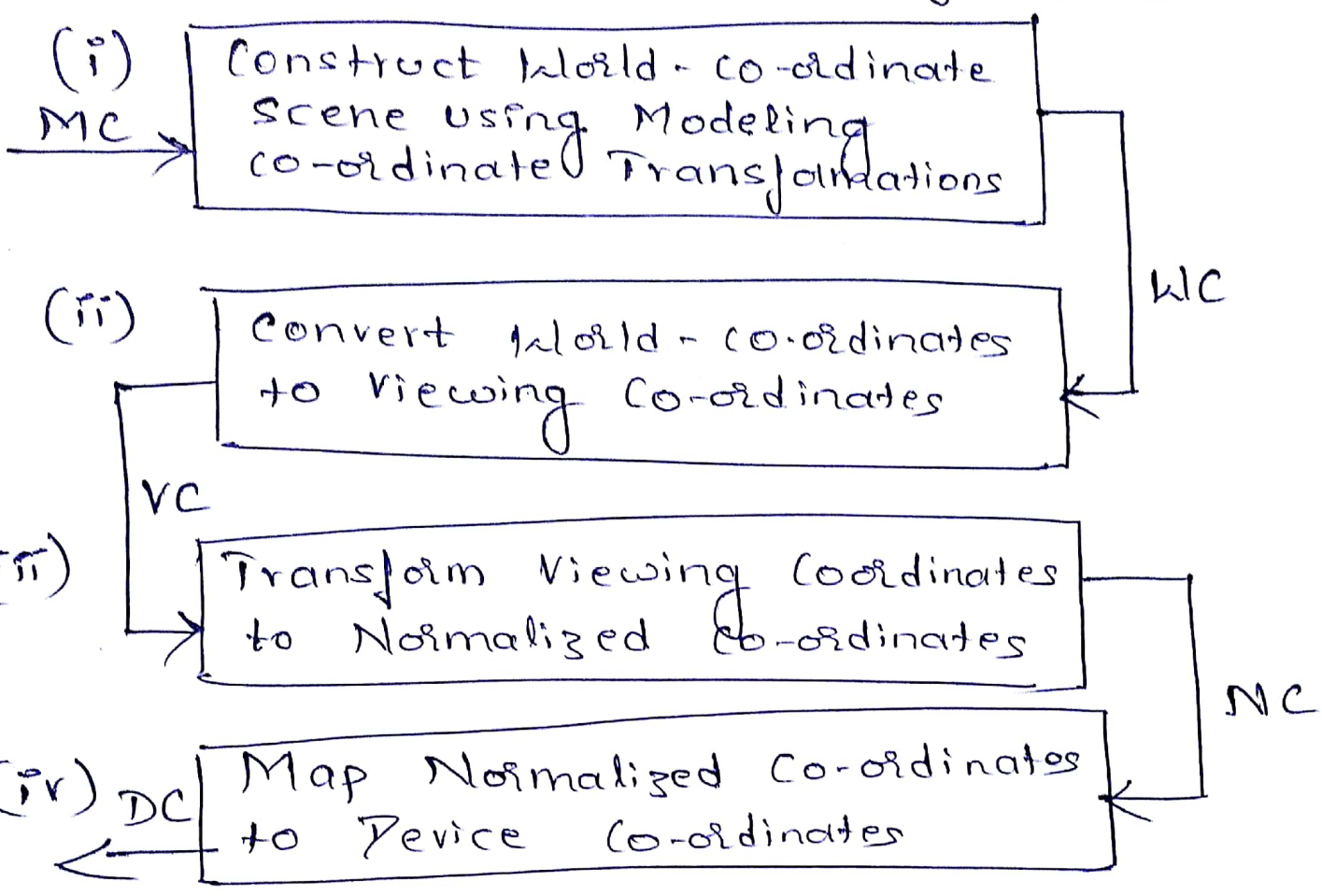


One Point Perspective



Projections	Advantages and Disadvantages
Orthographic Projections	➢ Preserves both distances and angles • Shapes preserved • Can be used for measurements(Building plans, Manuals) ➢ Cannot see what object really looks like because many surfaces hidden from view • Often we add the isometric
Axonometric Projections	➢ Lines are scaled (*foreshortened*) but can find scaling factors ➢ Lines preserved but angles are not • Projection of a circle in a plane not parallel to the projection plane is an ellipse ➢ Can see three principal faces of a box-like object ➢ Some optical illusions possible • Parallel lines appear to diverge ➢ Does not look real because far objects are scaled the same as near objects ➢ Used in CAD applications
Oblique Projections	➢ Can pick the angles to emphasize a particular face • Architecture: plan oblique, elevation oblique ➢ Angles in faces parallel to projection plane are preserved while we can still see "around" side ➢ In physical world, cannot create with simple camera; possible with bellows camera or special lens (architectural)
Perspective Projections	➢ Objects further from viewer are projected smaller than the same sized objects closer to the viewer (*diminution*) • Looks realistic ➢ Equal distances along a line are not projected into equal distances (*non uniform foreshortening*) ➢ Angles preserved only in planes parallel to the projection plane ➢ More difficult to construct by hand than parallel projections (but not more difficult by computer)

The Two Dimensional Viewing Pipeline:



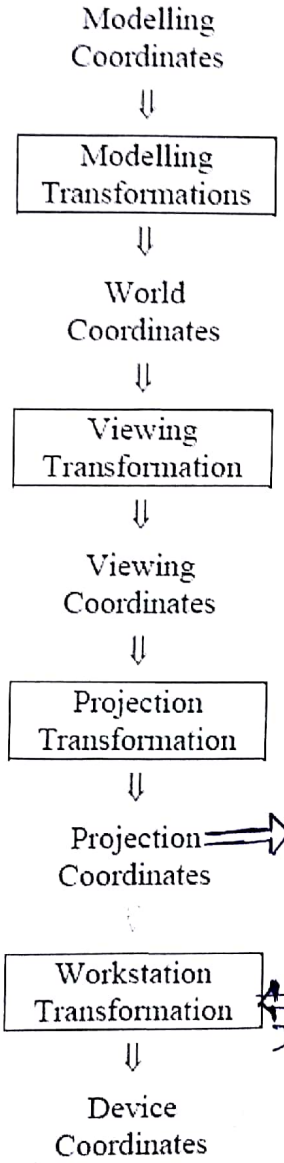
[** Refer Hearn & Baker for complete explanation.]

Three-Dimensional Viewing

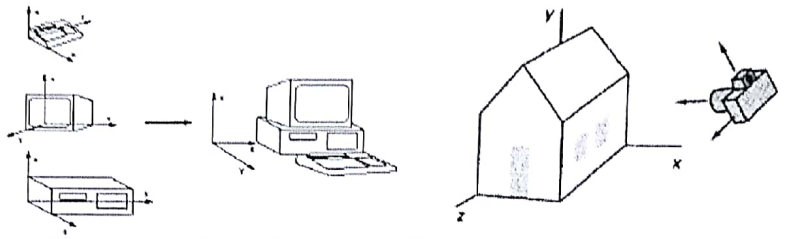
Viewing in 3D involves the following considerations:

- We can view an object from any spatial position, eg.
  - In front of an object,
  - Behind the object,
  - In the middle of a group of objects,
  - Inside an object, etc.
- 3D descriptions of objects must be projected onto the flat viewing surface of the output device.
- The clipping boundaries enclose a volume of space.

Three Dimensional Viewing Pipeline

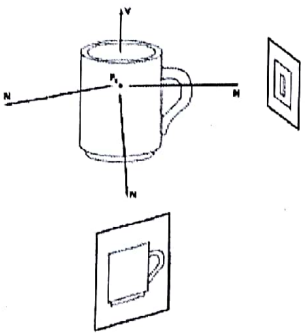


Explanation:



Modelling Transformation and Viewing Transformation can be done by 3D transformations.

The viewing-coordinate system is used in graphics packages as a reference for specifying the observer viewing position and the position of the projection plane.



Projection operations convert the viewing-coordinate description (3D) to coordinate positions on the projection plane (2D). (Usually combined with clipping, visual-surface identification, and surface-rendering)

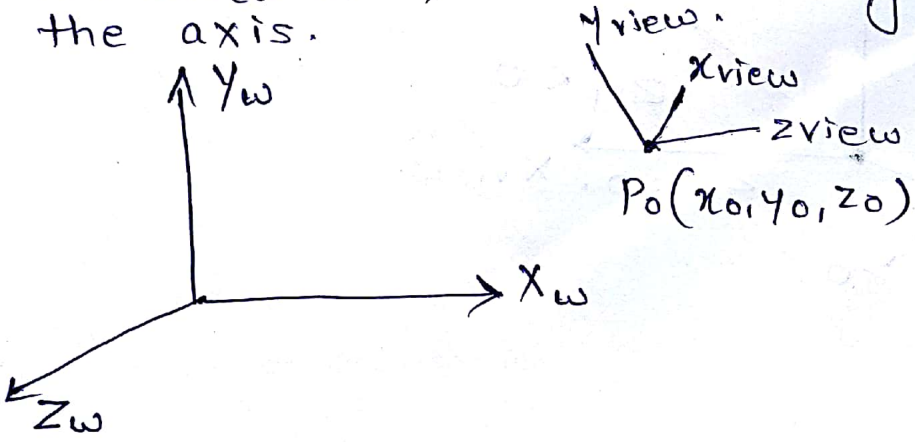
Workstation transformation maps the coordinate positions on the projection plane to the output device.

[* Refer Hearn & Baker for detailed
** explanation]

Three Dimensional Viewing Co-ordinate Parameters

→ Select a world co-ordinate position $P_0 = (x_0, y_0, z_0)$ for the viewing origin, which is called the view point or viewing position. (eye position or camera position)

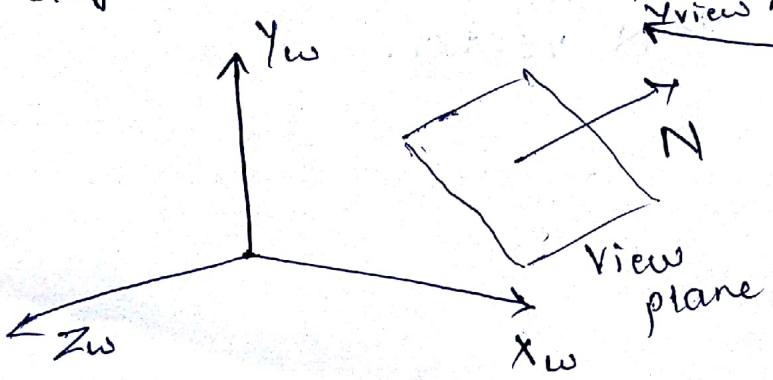
→ We specify a view-up vector V, which defines the yview direction, for 3-D space we also need to assign a direction for one of the remaining two co-ordinate axes. This is typically accomplished with a second vector that defines the zview axis, with the viewing direction along the axis.



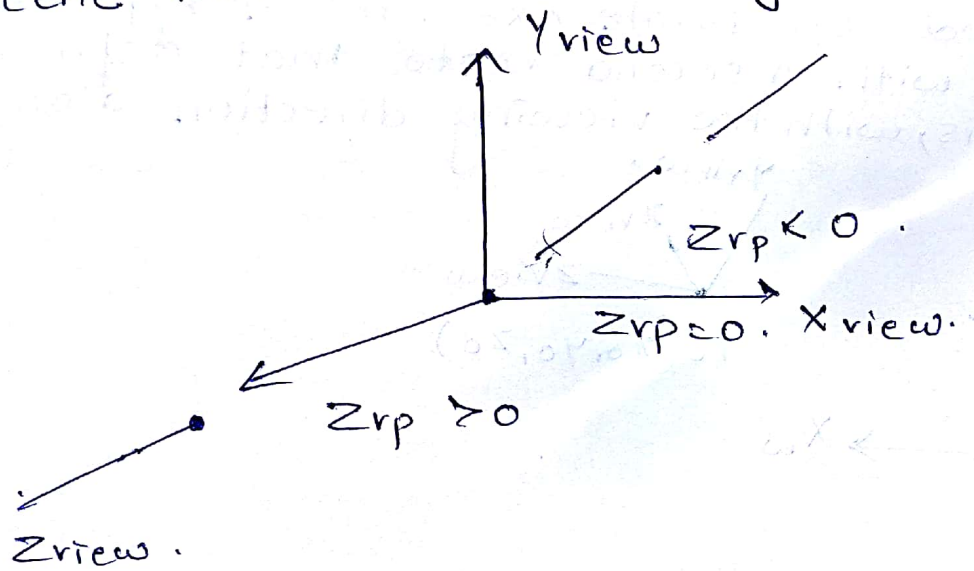
The view plane Normal vector :-

Since the viewing direction is usually along the zview axis, the view plane, also called the projection plane, is normally assumed to be perpendicular to this axis.

Thus the orientation of the view plane, as well as the direction for the positive zview axis can be defined with a view-plane normal vector N.

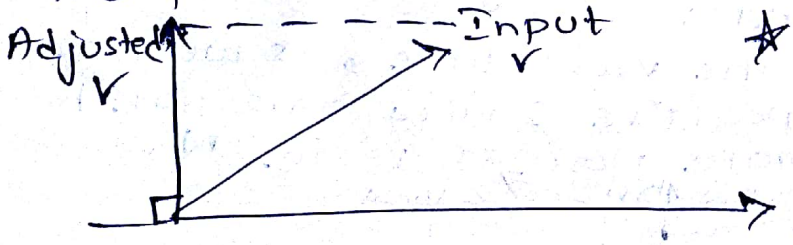


An additional scalar parameter is used to set the position of the view plane at some coordinate value z_{rp} along the z_{view} axis. This parameter value is usually specified as a distance from the viewing origin along the direction of viewing, which is often taken to be in the negative z_{view} direction. Thus, the view plane is always parallel to the $x_{view}y_{view}$ plane and the projection of objects to the view plane corresponds to the view of the scene that will be displayed on the output device.



The View-Up Vector:

Once we have chosen a view-plane normal vector N, we can set the direction for the view up vector V. This vector is used to establish the positive direction for the y_{view} axis.

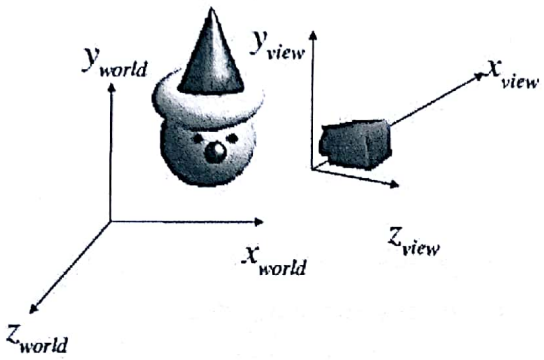


* V is adjusted such that it is projected onto a plane perpendicular to the view-plane normal

3D Viewing-Coordinate Parameters

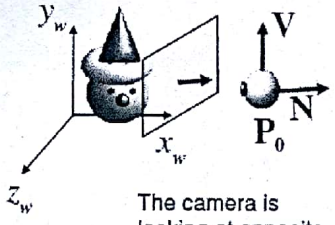
World coordinates to Viewing coordinates:
Viewing transformations

World coordinates



Viewing coordinates:
Viewers (Camera) position and viewing plane.

- How to define the viewing coordinate system (or view reference coordinate system):
  - Position of the viewer: $\mathbf{P}_0$
    - view point or viewing position
  - Orientation of the viewer:
    - View up vector: $\mathbf{V}$
    - Viewing direction: $\mathbf{N}$ (view plane normal)
    - $\mathbf{N}$ and $\mathbf{V}$ should be orthogonal
      - if it is not, $\mathbf{V}$ can be adjusted to be orthogonal to $\mathbf{N}$



The camera is looking at opposite direction of N

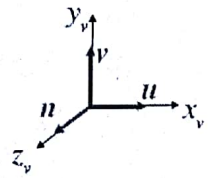
The uvn Viewing-Coordinate Reference Frame

- A set of unit vectors that define the viewing coordinate system is obtained by:

$$n=\frac{N}{|N|}=(n_x,n_y,n_z)$$

$$u=\frac{V\times n}{|V|}=(u_x,u_y,u_z)$$

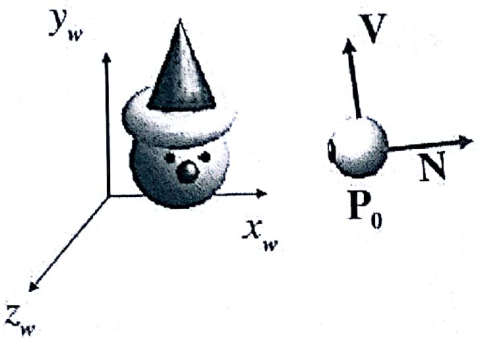
$$v=n\times u=(v_x,v_y,v_z)$$



By changing some of the viewing parameters (e.g., $\mathbf{N}$, viewing position), we can generate 3d viewing effects like rotating around an object or flying over a scene.

Transformation between Coordinate Systems

Given the objects in world coordinates, find the transformation matrix to transform them into viewing coordinate system. n, v, u :unit vectors defining the viewing coordinate system.



World coordinate system can be aligned with the viewing coordinate system in two steps:

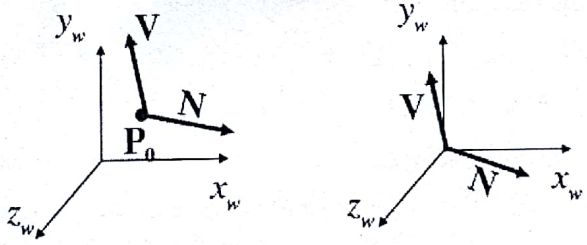
(1) Translate P0 to the origin of the world coordinate system,

(2) Rotate to align the axes.

- Translation: Move view reference point to origin.

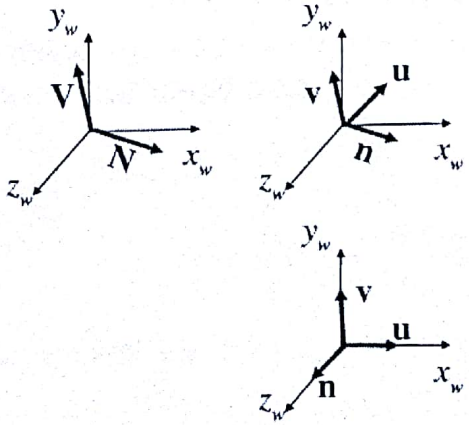
$$P_0=(x_0, y_0, z_0)$$

$$T=\begin{bmatrix} 1 & 0 & 0 & -x_0 \\ 0 & 1 & 0 & -y_0 \\ 0 & 0 & 1 & -z_0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- Rotation: After we find **n**, **v** and **u**, we can use them to define the rotation matrix for aligning the axes.

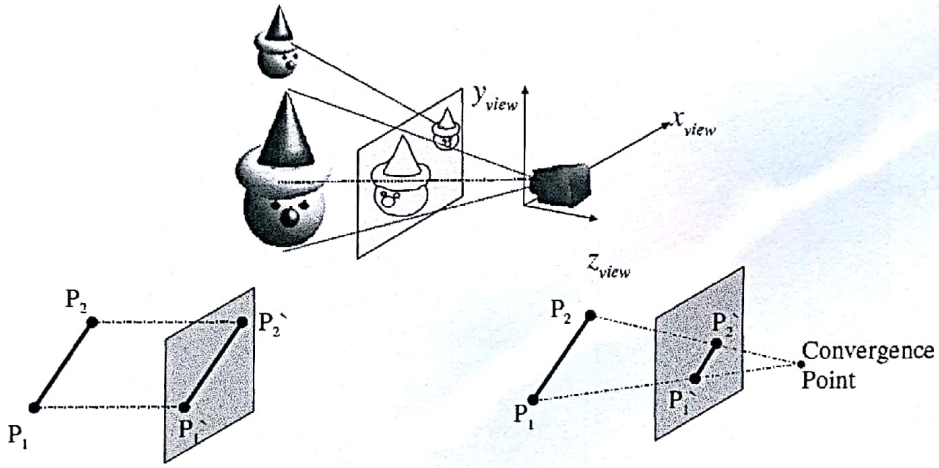
$$R=\begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ n_x & n_y & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



- The transformation matrix from world coordinate to viewing reference frame

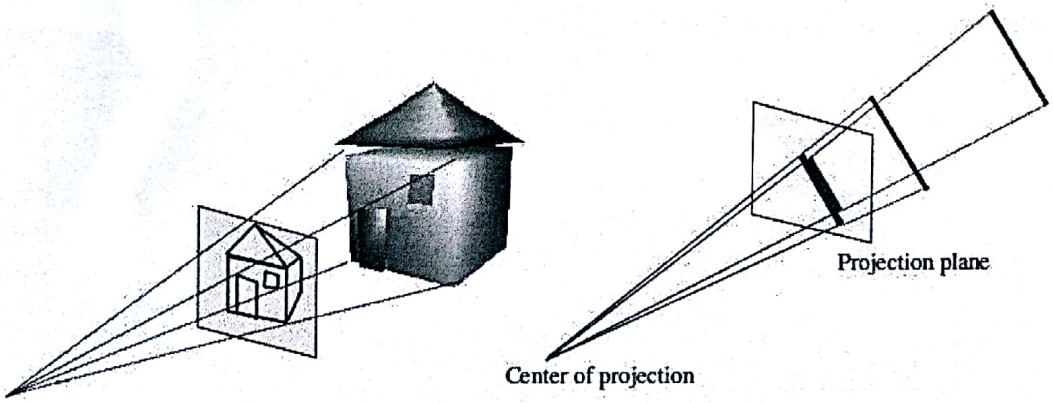
$$M_{WC,VC} = R \cdot T = \begin{bmatrix} u_x & u_y & u_z & -x_0 u_x - y_0 u_y - z_0 u_z \\ v_x & v_y & v_z & -x_0 v_x - y_0 v_y - z_0 v_z \\ n_x & n_y & n_z & -x_0 n_x - y_0 n_y - z_0 n_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Projection: 3D to 2D. Perspective or parallel.

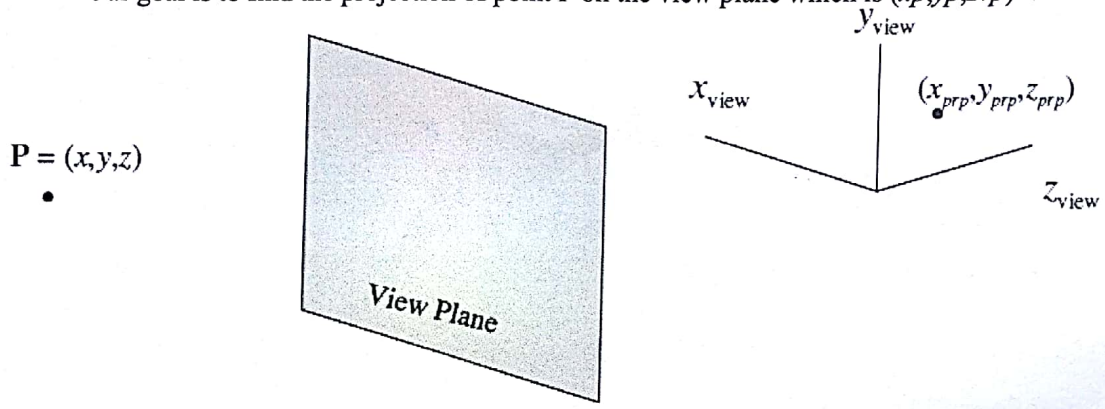


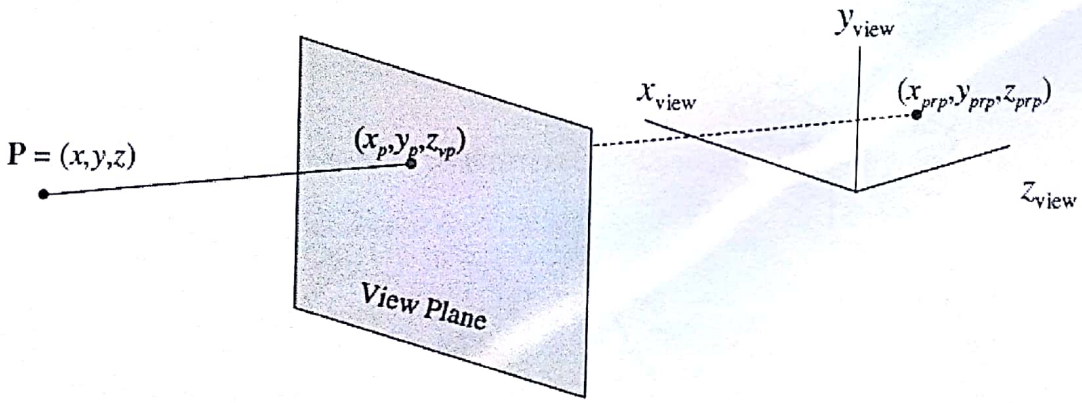
Perspective Projection

- Single point centre of projection (i.e. projection lines converge at a point)
- Shapes are projected smaller as their distances to the view plane increase.
- More realistic (human eye is a perspective projector)
- Depending on number of principal axes intersecting the viewing plane: 1, 2 or 3 vanishing points



- Assume that the projection reference point is selected at an arbitrary position (*xprp,yprp,zprp*) and the view plane is located at a selected position *zvp* on the *zview* axis.
- Our goal is to find the projection of point P on the view plane which is (*xp,yp,zvp*)





- The projected point (*xp,yp,zvp*) can be found using parametric equation of the projection line: We can solve for *u* at *z'* = *zvp* and then plugin *u* to find *xp* and *yp*.

$$x' = x - (x - x_{prp})u$$
$$y' = y - (y - y_{prp})u \qquad 0 \le u \le 1$$
$$z' = z - (z - z_{prp})u$$

- The denominators are functions of *z*, hence we cannot directly derive the matrix form. We will use homogeneous coordinates to do that.

$$u = \frac{z_{vp} - z}{z_{prp} - z}$$

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + x_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right)$$

$$y_p = y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) + y_{prp}\left(\frac{z_{vp} - z}{z_{prp} - z}\right)$$

- **Special cases: When the projection reference point is on the zview axis.**

$$x_{prp} = y_{prp} = 0$$

$$\Rightarrow$$

$$x_p = x\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right) \qquad y_p = y\left(\frac{z_{prp} - z_{vp}}{z_{prp} - z}\right)$$

- **When the projection reference point is at the view coordinate system origin.**

$$x_{prp} = y_{prp} = z_{prp} = 0$$

$$\Rightarrow$$

$$x_p = x\left(\frac{z_{vp}}{z}\right) \qquad y_p = y\left(\frac{z_{vp}}{z}\right)$$

.

- **When the view plane is the uv plane and the projection reference point is on the zview axis**

$$x_{prp} = y_{prp} = z_{vp} = 0$$

$$\Rightarrow$$

$$x_p = x\left(\frac{z_{prp}}{z_{prp} - z}\right) \qquad y_p = y\left(\frac{z_{prp}}{z_{prp} - z}\right)$$

Perspective-Projection Transformation Matrix

- We can use homogeneous coordinates to express the perspective projection equations:
- We compute homogenous coordinates in the perspective projection equations

$$x_p = \frac{x_h}{h}, \qquad y_p = \frac{y_h}{h}$$

h= z_{prp} - z

$$x_h = x(z_{prp} - z_{vp}) + x_{prp}(z_{vp} - z)$$
$$y_h = y(z_{prp} - z_{vp}) + y_{prp}(z_{vp} - z)$$

- **We can set up a perspective projection matrix to convert 3D coordinates to homogenous coordinates, then we can divide the homogeneous coordinates by *h* to obtain the true positions**

$$(1) \qquad P_h = M_{pers} \cdot P$$

$$(2) \qquad P_p = \begin{bmatrix} 1/h & 0 & 0 & 0 \\ 0 & 1/h & 0 & 0 \\ 0 & 0 & 1/h & 0 \\ 0 & 0 & 0 & 1/h \end{bmatrix} \cdot P_h$$

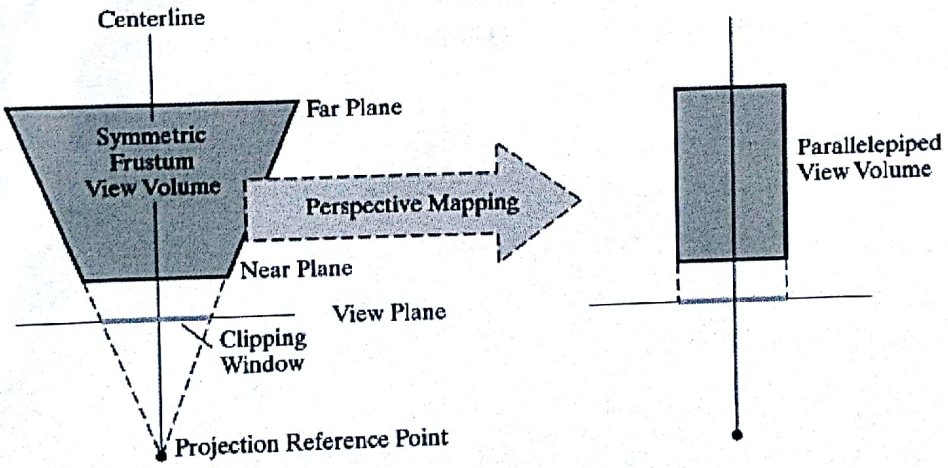
- **Finding Mpers**

We need to be careful when finding *zh*. We need to preserve the depth information, we cannot simply use the coordinate *zvp* for all the projected points. We set up the Mpers matrix so that the z coordinate of a point is converted to a normalized *zp* coordinate.

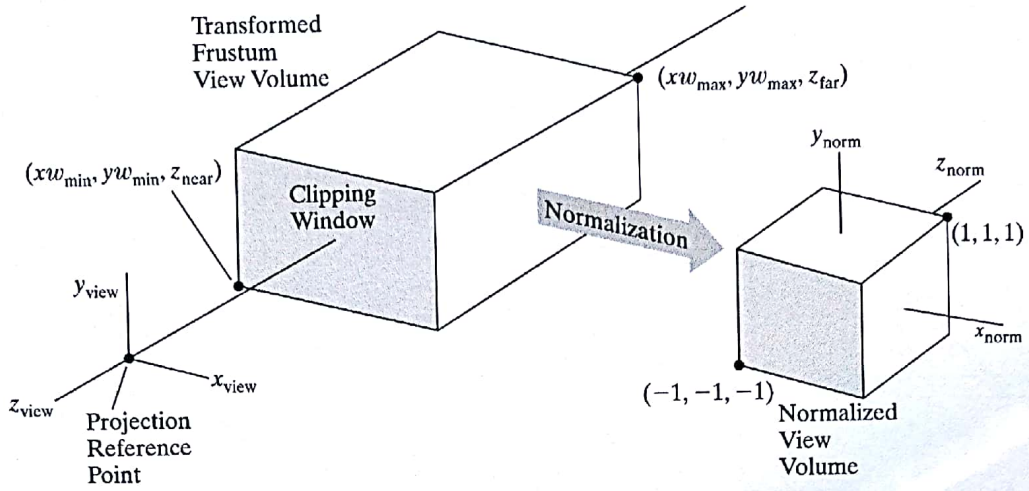
$$M_{pers} = \begin{bmatrix} z_{prp} - z_{vp} & 0 & -x_{prp} & x_{prp} z_{vp} \\ 0 & z_{prp} - z_{vp} & -y_{prp} & y_{prp} z_{vp} \\ 0 & 0 & s_z & t_z \\ 0 & 0 & -1 & z_{prp} \end{bmatrix}$$

s_z and t_z are the scaling and translation parameters for normalizing projected z coordinates.

- The symmetric perspective transformation will map the objects into a parallelpiped view volume



➢ **Normalization :** Normalization is then similar to normalization of an orthographic projection



OpenGL 3D Viewing Functions

➢ The viewing parameters (camera position, viewup vector, viewplane normal) are specified as part of modeling transformations. A matrix is formed and concatenated with the current modelview matrix. So, to set up camera parameters:

```
glMatrixMode (GL_MODELVIEW);
gluLookAt (x0, y0, z0, xref, yref, zref, Vx, Vy, Vz);
```

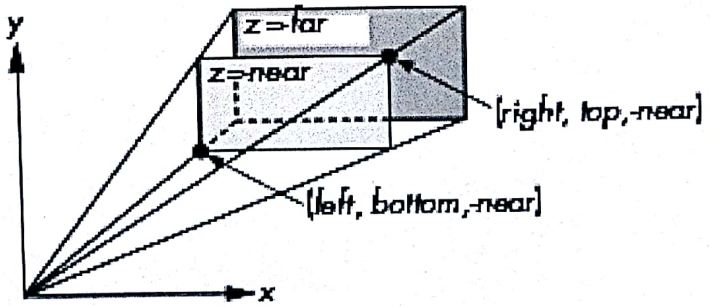
$N = P_0 - P_{ref}$

✓ Viewing direction is along the –zview axis
✓ If we do not invoke the gluLookAt function. The default camera parameters are:

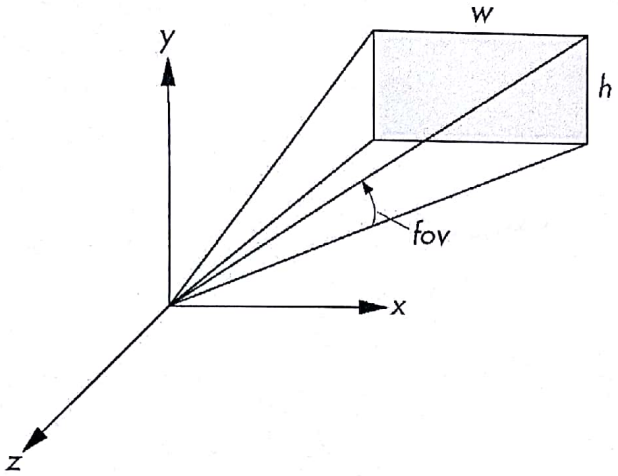
- P0= (0,0,0)
- Pref = (0,0,1)
- V = (0,1,0)

➢ **General Perspective- Projection Function**

glFrustum (xwmin, xwmax, ywmin, ywmax, dnear, dfar)
Projection reference point is the viewing (camera position) and the view plane normal is the zview axis, viewing direction is -zview.
If xwmin != xwmax or ywmin!=ywmax we can obtain an oblique perspective projection frustum. Otherwise, we will have a symmetric perspective projection.



z



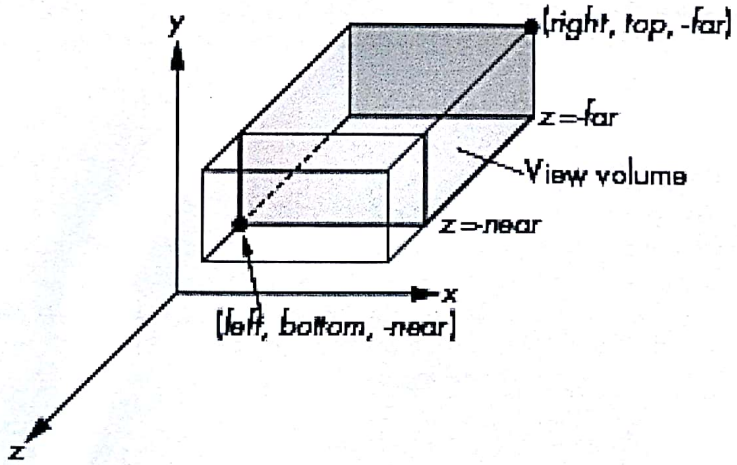
- **OpenGL Symmetric Perspective-Projection Function**

gluPerspective (theta, aspect, dnear, dfar)

The parameters theta and aspect determines the position and the size of the clipping window on the near plane. *theta* is the field of view angle, i.e., the angle between top and bottom clipping planes. *Aspect* is the aspect ratio of the clipping window (i.e., width/height) . *dnear* and *dfar* are assigned positive values with *dnear<dfar*. The actual positions of the near and far planes are *znear* = dnear and *zfar* = dfar

- **OpenGL Orthogonal- Projection Function**

Projection parameters are set in the OpenGL projection mode.



glMatrixMode (GL_PROJECTION);
glOrtho(xwmin,xwmax,ywmin,ywmax,dnear,dfar);

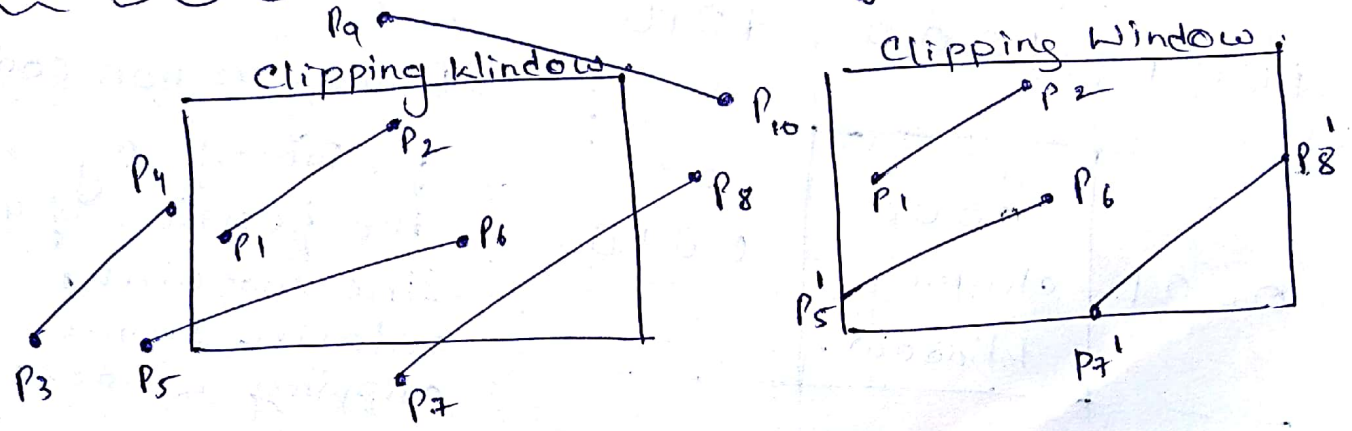
By default we have: glOrtho (1.0,1.0,1.0,1.0,1.0,1.0);

[*** Self Study Orthogonal Projection Hearn & Baker]

UNIT - IV

CLIPPING

Two Dimensional Line Clipping.



Before Clipping

After Clipping

A line clipping algorithm processes each line in a scene through a series of tests and intersections calculations to determine whether the entire line or any part of it is to be saved. The expensive part of a line clipping procedure is in calculating the intersection positions of a line with the window edges.

(i) Determine whether a line is fully inside the clipping window or completely outside.

(ii) If lines are partially inside the clipping windows, intersection calculations are to be performed. (x, y)

$$x = x_0 + u(x_{end} - x_0)$$

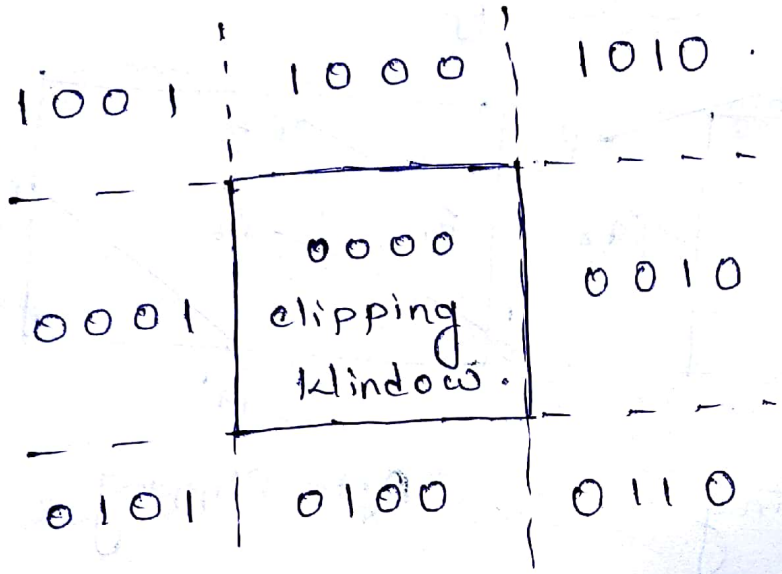
$$y = y_0 + u(y_{end} - y_0) \qquad 0 \le u \le 1.$$

Solving for u, if u value is outside the range from 0 to 1, the line segment does not intersect that window borda line. But if u value is within the range from 0 to 1, part of the line is inside the border.

Cohen - Sutherland line Clipping.

Top.	Bottom	Right	Left
4	3	2	1

→ 4 bit value.



Binary region codes for identifying the position of a line endpoint, relative to the clipping window boundaries.

Bit 1 is set if $x < x_{wmin}$. | Bit 3 is set if $y < y_{wmin}$
Bit 2 is set if $x > x_{wmax}$ | Bit 4 is set if $y > y_{wmax}$

Similarly we can say
Bit 1 is the sign bit of $x - x_{wmin}$,
Bit 2 is the sign bit of $x_{wmax} - x$,
Bit 3 is the sign bit of $y - y_{wmin}$,
Bit 4 is the sign bit of $y_{wmax} - y$.

To determine the boundary intersection for a line segment, we can use the slope intercept form of the line equation.

→ vertical clipping

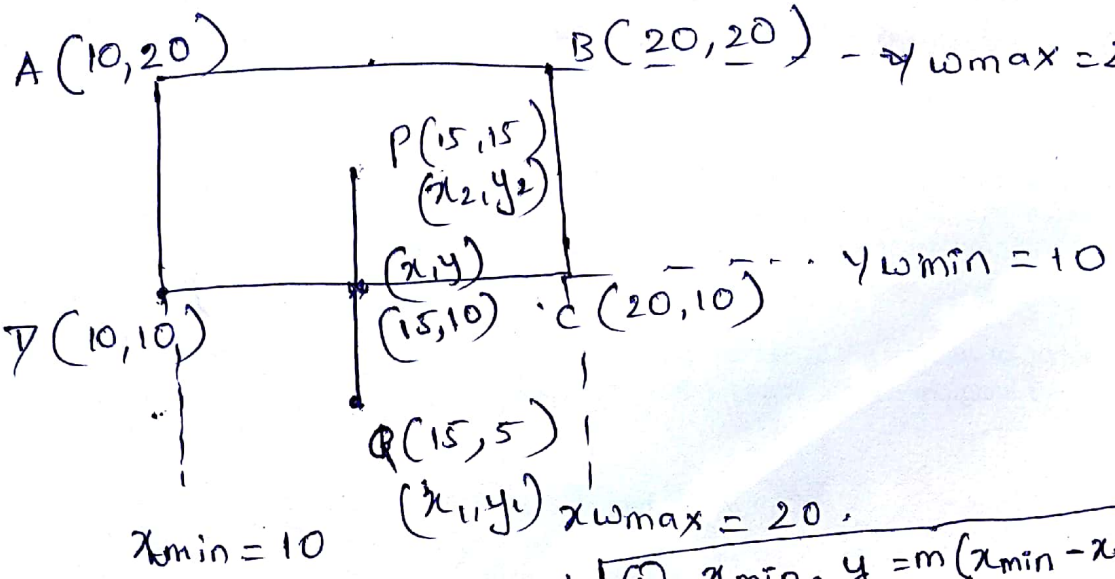
$$y = y_0 + m(x - x_0) \qquad m = \frac{(y_{end} - y_0)}{(x_{end} - x_0)}$$

→ horizontal clipping.

$$x = x_0 + \frac{(y - y_0)}{m}$$

★ set x to x_{wmin} or x_{wmax}
★ set y to y_{wmin} or y_{wmax}.

Example: Window is defined as A(10,20), B(20,20), C(20,10) D(10,10) find the visible portion of line P(15,15) and Q(15,5)



	T	B	R	L
P	0	0	0	0
Q	0	1	0	0

P & Q = 0 0 0 0 → line partially inside window. } clipping.

(1) x_{min}, $y = m(x_{min} - x_1) + y_1$ (L)
(2) x_{max}, $y = m(x_{max} - x_1) + y_1$ (R)
(3) y_{max}, $x = 1/m(y_{max} - y_1) + x_1$ (T)
(4) y_{min}, $x = 1/m(y_{min} - y_1) + x_1$ (B)

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)} = \frac{(15-5)}{0} = \infty.$$

line cuts y_{min}

$$x = x_1 + (y - y_1)/m.$$
$$= 15 + (10-5)/\infty.$$
$$= 15$$

$(x, y) = (15, 10)$ → intersection.

Assignment: A(20, 20), B(90, 20), C(90, 70), D(20, 70) $P_1(10, 30)$ $P_2(80, 90)$. Find the visible portion of the line segment.

Once the codes for each endpoint of a line are determined, the logical **AND** operation of the codes determines if the line is completely outside of the window. If the logical AND of the endpoint codes is **not zero**, the line can be trivially rejected. For example, if an endpoint had a code of 1001 while the other endpoint had a code of 1010, the logical AND would be 1000 which indicates the line segment lies outside of the window. On the other hand, if the endpoints had codes of 1001 and 0110, the logical AND would be 0000, and the line could not be trevally rejected.

The logical **OR** of the endpoint codes determines if the line is completely inside the window. If the logical OR is **zero**, the line can be trivially accepted. For example, if the endpoint codes are 0000 and 0000, the logical OR is 0000 - the line can be trivially accepted. If the endpoint codes are 0000 and 0110, the logical OR is 0110 and the line cannot be trivially accepted.

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

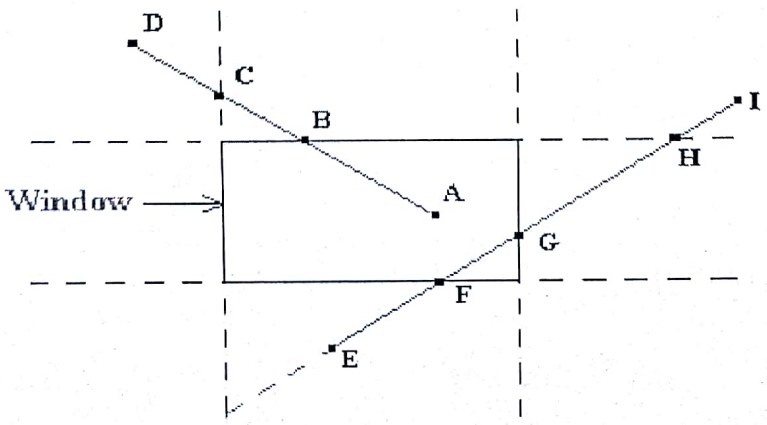
To perform the trivial acceptance and rejection tests, we extend the edges of the window to divide the plane of the window into the nine regions. Each end point of the line segment is then assigned the code of the region in which it lies.

1. Given a line segment with endpoint $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$
2. Compute the 4-bit codes for each endpoint.

If both codes are **0000**,(bitwise OR of the codes yields 0000) line lies completely **inside** the window: pass the endpoints to the draw routine.

If both codes have a 1 in the same bit position (bitwise AND of the codes is **not** 0000), the line lies **outside** the window. It can be trivially rejected.

3. If a line cannot be trivially accepted or rejected, at least one of the two endpoints must lie outside the window and the line segment crosses a window edge. This line must be **clipped** at the window edge before being passed to the drawing routine.
4. Examine one of the endpoints, say $P_1 = (x_1, y_1)$. Read P_1 's 4-bit code in order: **Left-to-Right**, **Bottom-to-Top**.
5. When a set bit (1) is found, compute the **intersection I** of the corresponding window edge with the line from P_1 to P_2. Replace P_1 with **I** and repeat the algorithm.



1. Consider the line segment **AD**.

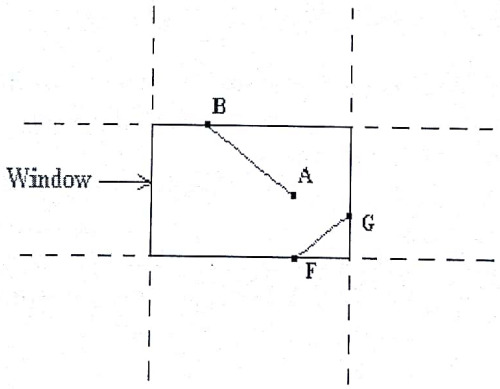
Point **A** has an outcode of **0000** and point **D** has an outcode of **1001**. The logical AND of these outcodes is zero; therefore, the line cannot be trivially rejected. Also, the logical OR of the outcodes is not zero; therefore, the line cannot be trivially accepted. The algorithm then chooses **D** as the outside point (its outcode contains 1's). By our testing order, we first use the top edge to clip **AD** at **B**. The algorithm then recomputes **B**'s outcode as **0000**. With the next iteration of the algorithm, **AB** is tested and is trivially accepted and displayed.

2. Consider the line segment **EI**

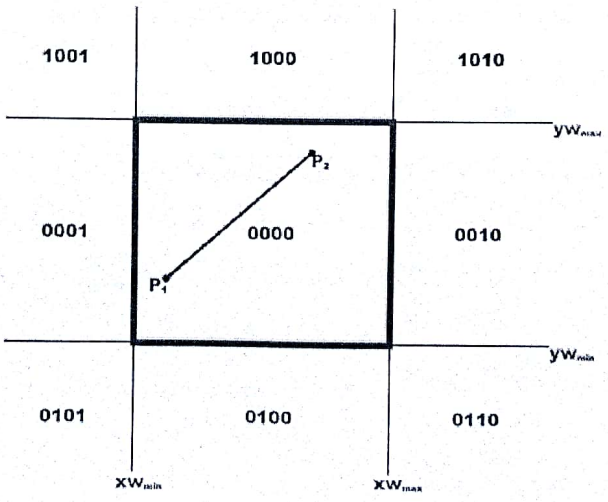
Point **E** has an outcode of **0100**, while point **I**'s outcode is **1010**. The results of the trivial tests show that the line can neither be trivially rejected nor accepted. Point **E** is determined to be an outside point, so the algorithm clips the line against the bottom edge of the window. Now line **EI** has been clipped to be line **FI**. Line **FI** is tested and cannot be trivially accepted or rejected. Point **F** has an outcode of **0000**, so the algorithm chooses point **I** as an outside point since its outcode is**1010**. The line **FI** is clipped against the window's top edge, yielding a new line **FH**. Line **FH** cannot be trivially accepted or rejected. Since **H**'s outcode is **0010**, the next iteration of the algorithm clips against the window's right edge, yielding line **FG**. The next iteration of the algorithm tests **FG** and it is trivially accepted and display.

After Clipping

After clipping the segments **AD** and **EI**, the result is that only the line segment **AB** and **FG** can be seen in the window.



Example 1:

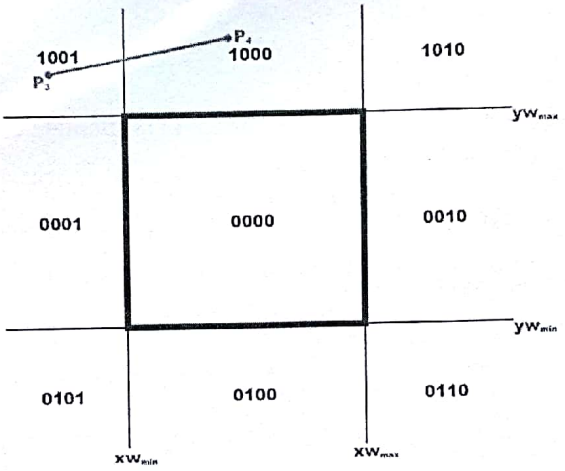


4 bit clip code:
T B R L
T = $y > yw_{max}$
B = $y < yw_{min}$
R = $x > xw_{max}$
L = $x < xw_{min}$
Calculate clip code

$P_1 = 0\ 0\ 0\ 0$
$P_2 = 0\ 0\ 0\ 0$

--- 0 0 0 0
=>accept segment $P_1\ P_2$ (trivial accept since both end points are in the window)

Example 2:

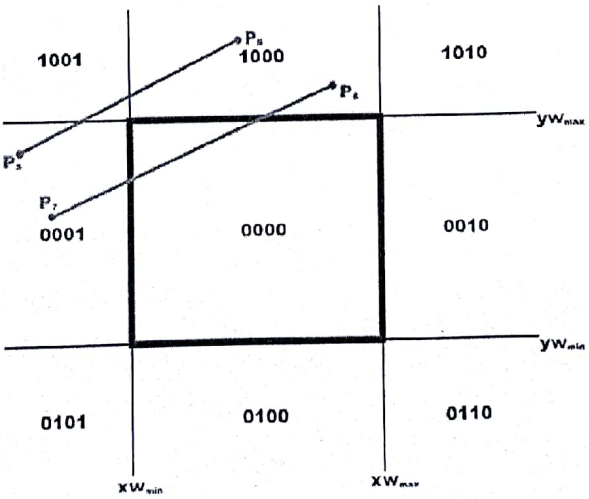


4 bit clip code:
T B R L
T = $y > yw_{max}$
B = $y < yw_{min}$
R = $x > xw_{max}$
L = $x < xw_{min}$
Calculate clip code

$P_3 = 1\ 0\ 0\ 1$
$P_4 = 1\ 0\ 0\ 0$

--- 1 0 0 0
=> reject segment $P_3\ P_4$

Example 3:



Step 1:

4 bit clip code:

T B R L

$T = y > yw_{max}$

$B = y < yw_{min}$

$R = x > xw_{max}$

$L = x < xw_{min}$

$P_5 = 0\ 0\ 0\ 1$ $P_7 = 0\ 0\ 0\ 1$ (1) Calculate clip code

$P_6 = 1\ 0\ 0\ 0$ $P_8 = 1\ 0\ 0\ 0$

--- 0 0 0 0 --- 0 0 0 0

Step 2:

4 bit clip code:

A B R L

$A = y > yw_{max}$

$B = y < yw_{min}$

$R = x > xw_{max}$

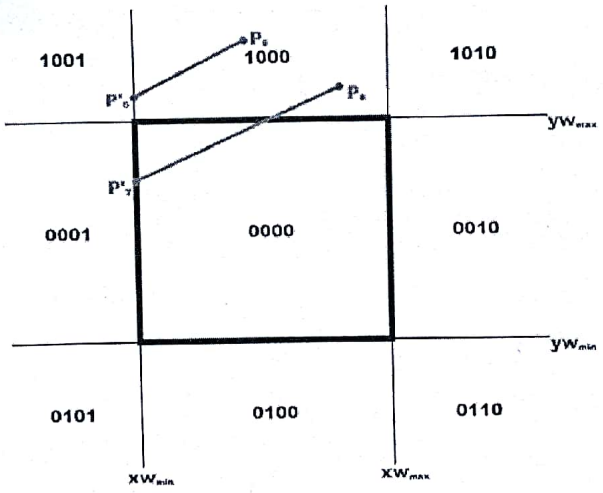
$L = x < xw_{min}$

$P_5 = 0\ 0\ 0\ 1$ $P_7 = 0\ 0\ 0\ 1$ (1) Calculate clip code

$P_6 = 1\ 0\ 0\ 0$ $P_8 = 1\ 0\ 0\ 0$

--- 0 0 0 0 --- 0 0 0 0

$P'_5 = (xw_{min}, y_5 + m(xw_{min} - x_5))$ $P'_7 = (xw_{min}, y_7 + m(xw_{min} - x_7))$



Step 3:

4 bit clip code:

A B R L

$A = y > yw_{max}$

$B = y < yw_{min}$

$R = x > xw_{max}$

$L = x < xw_{min}$

$P_5 = 0\ 0\ 0\ 1$ $P_7 = 0\ 0\ 0\ 1$ (1) Calculate clip code

$P_6 = 1\ 0\ 0\ 0$ $P_8 = 1\ 0\ 0\ 0$

--- 0 0 0 0 --- 0 0 0 0

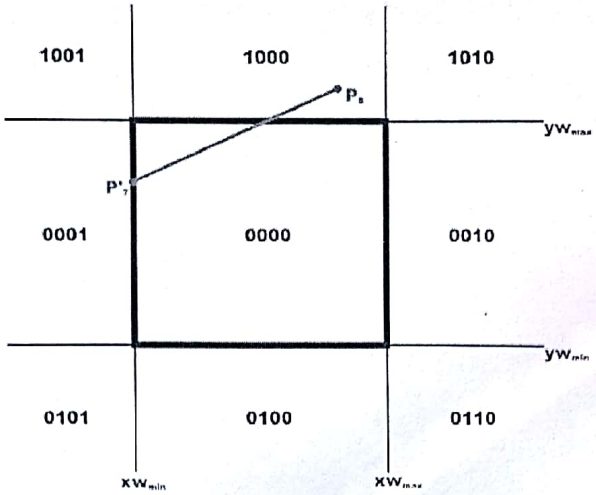
$P'_5 = (xw_{min}, y_5 + m(xw_{min} - x_5))$ $P'_7 = (xw_{min}, y_7 + m(xw_{min} - x_7))$ (2) Clip against left

$P'_5 = 1\ 0\ 0\ 0$ $P'_7 = 0\ 0\ 0\ 0$ (3) Update clip code

$P_6 = 1\ 0\ 0\ 0$ $P_8 = 1\ 0\ 0\ 0$

--- 1 0 0 0 --- 0 0 0 0

=> reject $P'_5\ P_6$



Step 4:

4 bit clip code:

A B R L

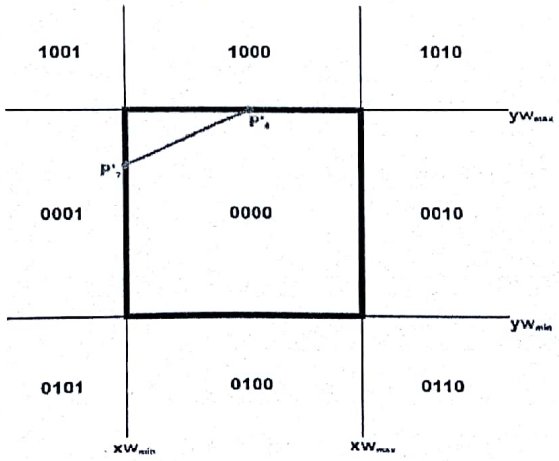
A = $y > yw_{max}$

B = $y < yw_{min}$

R = $x > xw_{max}$

L = $x < xw_{min}$

$P_5 = 0\ 0\ 0\ 1$ $P_6 = 1\ 0\ 0\ 0$	$P_7 = 0\ 0\ 0\ 1$ $P_8 = 1\ 0\ 0\ 0$	(1) Calculate clip code
--- 0 0 0 0	--- 0 0 0 0	
$P'_5 = (xw_{min}, y_5 + m(xw_{min} - x_5))$	$P'_7 = (xw_{min}, y_7 + m(xw_{min} - x_7))$	(2) Clip against left
$P'_5 = 1\ 0\ 0\ 0$ $P_6 = 1\ 0\ 0\ 0$	$P'_7 = 0\ 0\ 0\ 0$ $P_8 = 1\ 0\ 0\ 0$	(3) Update clip code
--- 1 0 0 0 => reject $P'_5\ P_6$	--- 0 0 0 0	
		(4) Clip against right (5) Clip against bottom
	$P'_8 = (x_7 + (yw_{max} - y_7)/m, yw_{max})$	(6) Clip against top
	$P'_7 = 0\ 0\ 0\ 0$ $P'_8 = 0\ 0\ 0\ 0$	
	--- 0 0 0 0 => accept $P'_7\ P'_8$	



Liang - Barsky Line Clipping

* Faster line clipping.

For a line segment with endpoints (x_0, y_0) and (x_{end}, y_{end}) the parametric form will be

$$x = x_0 + u\Delta x$$
$$y = y_0 + u\Delta y \quad 0 \le u \le 1.$$

$$\Delta x = x_{end} - x_0 \quad \Delta y = y_{end} - y_0.$$

$$x_{wmin} \le x_0 + u\Delta x \le x_{wmax}$$
$$y_{wmin} \le y_0 + u\Delta y \le y_{wmax}$$

which can be expressed as

$$up_k \le q_k \quad k = 1, 2, 3, 4.$$

where parameters p and q are defined as.

(L) $p_1 = -\Delta x \quad q_1 = x_0 - x_{wmin}$

(R) $p_2 = \Delta x \quad q_2 = x_{wmax} - x_0$

(B) $p_3 = -\Delta y \quad q_3 = y_0 - y_{wmin}$

(T) $p_4 = \Delta y \quad q_4 = y_{wmax} - y_0.$

Any line that is parallel to one of the clipping window edges has $p_k = 0$ for the value of k corresponding to that boundary where $k = 1, 2, 3, 4$ corresponds to left, right, bottom, top. boundaries. respectively. If for that value of K, we also find $q_k < 0$, then the line is completely outside the boundary and can be eliminated from further consideration. If $q_k >= 0$ the line is inside the parallel clipping border.

When $p_k < 0$, the infinite extension of the line proceeds from the outside to the inside of the infinite extension of this particular clipping window edge. If $p_k > 0$ the line proceeds from the inside to outside.

For a nonzero value of p_k we can calculate the value of u that corresponds to the point where the infinitely extended line intersects the extension of window edge k as

$$u = \frac{q_k}{p_k}.$$

→ Line intersection parameters are initialized to the values $u_1 = 0$ and $u_2 = 1$.

→ For each clipping boundary, the appropriate values for p and q are calculated and used by the following tests.

```
if p<0
{   rk = qk/pk.
    if (r > u2)
        false
    else
        if (r > u1)
            u1 = r
}
else if (p>0)
{   rk = qk/pk
    if (r < u1)
        false
    else if (r < u2)
        u2 = r
}
else if (q < 0)
        false
    else
        true.
```

Liang – Barsky Line Clipping Algorithm

1. Parametric equation of line segment:

$X = X1 + U\,\Delta X$

$Y = Y1 + U\,\Delta Y$

Where, $\Delta X = X2 - X1$ and $\Delta Y = Y2 - Y1$

2. Using these equations Cyrus and Beck developed an algorithm that is generated more efficient than the Cohen Sutherland algorithm.
3. Later Liang and Barsky independently devised an even faster parametric line clipping algorithm.
4. In the Liang-Barsky approach we first the point clipping condition in parametric form:

   $Xmin \le X1 + U\Delta X \le Xmax$

   $Ymin \le Y1 + U\Delta Y \le Ymax$
5. Each of these four inequalities can be expressed as:

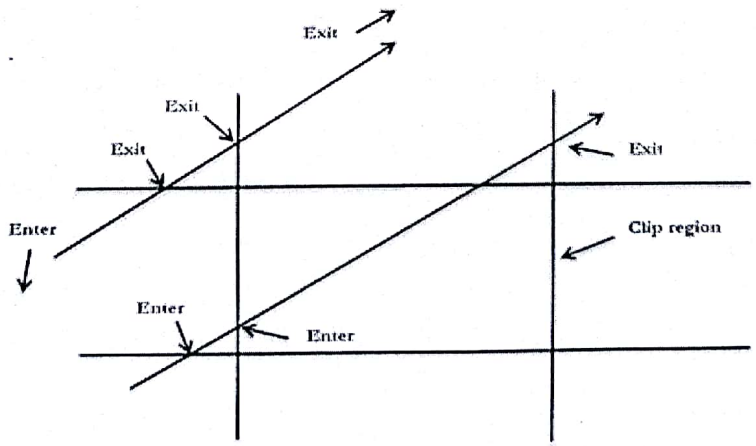
   $\mu pk \le qk$ for $k=1,2,3,4$
6. The parameters p & q are defined as:

   $p1 = -\Delta X$ and $q1 = X1 - Xmin$ (Left Boundary)

   $p2 = \Delta X$ and $q2 = Xmax - x1$ (Right Boundary)

   $P3 = -\Delta Y$ and $q3 = Y1 - Ymin$ (Bottom Boundary)

   $P4 = \Delta Y$ and $q4 = Ymax - y1$ (Top Boundary)
7. If a line is parallel to a view window boundary, the p value for that boundary is zero.
8. If the line is parallel to the X axis, for example then p1 and p2 must be zero.
   - Given $pk = 0$, if $qk < 0$ then line is trivially invisible because it is outside view window.
   - Given $pk = 0$, if $qk > 0$ then the line is inside the corresponding window boundary.
9. When $pk < 0$, as U increase line goes from the outside to inside i.e. entering.
10. When $pk > 0$, line goes from inside to outside i.e. exiting.
11. If there is a segment of line inside the clip region, a sequence of infinite line intersections must go entering, entering, exiting, and exiting as shown



- *f* When $pk < 0$,
  - as u increases - line goes from outside to inside - entering *f*
- When $pk > 0$,
  - line goes from inside to outside - exiting *f*
- When $pk = 0$,
  - line is parallel to an edge *f*
- If there is a segment of the line inside the clip region, a sequence of infinite line intersections must go: entering, entering, exiting, exiting

Example 1:
Given:
(Xmin,Ymin)=(10,10)
(Xmax,Ymax)=(50,50)

P1 (30, 60) and P2 = (60, 25)

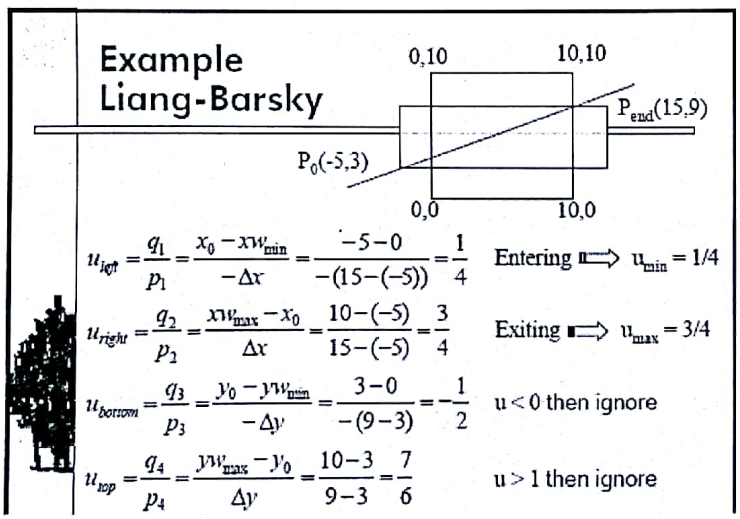
Solution:
Set Umin = 0 and Umax = 1
ULeft = q1 / p1
= X1 – Xmin / - Δ X
= 30 – 10 / - (60 - 30)
= 20 / - 30
= -0.67
URight = q2 / p2
= Xmax – X1/ ΔX
= 50 – 30 / (60 - 30)
= 20 / 30
= 0.67
UBottom= q3 / p3
= Y1 – Ymin / - ΔY
= 60 – 10 / - (25 – 60)
= 50 / 35
= 1.43
UTop = q4 / p4
= Ymax – Y1 / ΔY
= 50 – 60 / (25 - 60)
= -10 / - 35
= 0.29
Since ULeft=−0.57 which is less than Umin. Therefore we ignore it.

Similarly UBottom=1.43 which is greater than Umax. So we ignore it.

URight=Umin = 0.67 (Entering)

UTop=Umax = 0.29 (Exiting)
We have UTop = 0.29 and URight = 0.67
Q – P = (ΔX, ΔY) = (30, -35)
Since Umin>Umax, there is no line segment to draw.

Example 2:



Liang-Barsky Line-Clipping

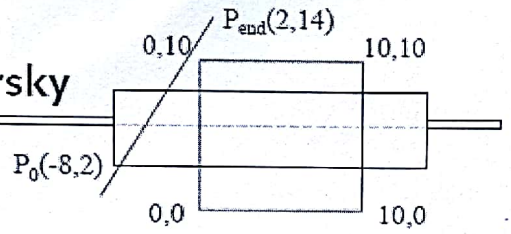
- We have $u_{min} = 1/4$ and $u_{max} = 3/4$

$$P_{end} - P_0 = (15+5, 9-3) = (20, 6)$$
$$\Delta x \quad \Delta y$$

- If $u_{min} < u_{max}$, there is a line segment
  - compute endpoints by substituting u values
- Draw a line from
  (-5+(20)·(1/4), 3+(6)·(1/4))
  to
  (-5+(20)·(3/4), 3+(6)·(3/4))

Example 3:

Example Liang-Barsky



$$u_{left} = \frac{q_1}{p_1} = \frac{x_0 - xw_{min}}{-\Delta x} = \frac{-8-0}{-(2-(-8))} = \frac{4}{5}$$ Entering $\Rightarrow u_{min} = 4/5$

$$u_{right} = \frac{q_2}{p_2} = \frac{xw_{max} - x_0}{\Delta x} = \frac{10-(-8)}{2-(-8)} = \frac{9}{5}$$ $u > 1$ then ignore

$$u_{bottom} = \frac{q_3}{p_3} = \frac{y_0 - yw_{min}}{-\Delta y} = \frac{2-0}{-(14-2)} = -\frac{1}{6}$$ $u < 0$ then ignore

$$u_{top} = \frac{q_4}{p_4} = \frac{yw_{max} - y_0}{\Delta y} = \frac{10-2}{14-2} = \frac{2}{3}$$ Exiting $\Rightarrow u_{max} = 2/3$

Liang-Barsky Line-Clipping

- We have $u_{min} = 4/5$ and $u_{max} = 2/3$

$$P_{end} - P_0 = (2+8, 14-2) = (10, 12)$$

- $u_{min} > u_{max}$,
  there is no line segment do draw