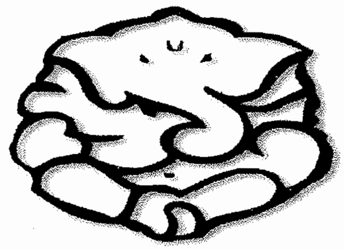


6

COMPUTER GRAPHICS

ASHOK KUMAR K
VIVEKANANDA INSTITUTE OF TECHNOLOGY
MOB: 9742024066
e-MAIL: celestialcluster@gmail.com

S. K. L. N ENTERPRISES
Contact: 9886381393

UNIT 1

INTRODUCTION

Syllabus

* Applications of computer graphics
* A graphics system
* Images: physical and synthetic
* Imaging systems.
* The synthetic camera model
* The programmer's interface.
* Graphics architectures.
* programmable pipelines.
* performance characteristics

* Sierpinski gasket
* programming two-dimensional applications.

7 Hours.

~~APPLICATIONS OF~~

Computer Graphics is concerned with all aspects of producing pictures or impages using a computer.

APPLICATIONS OF COMPUTER GRAPHICS

* Applications of computer graphics are divided into four major areas,

  i.) Display of information
  ii.) Design
  iii.) simulation and animation
  iv.) user interfaces.

Display of information

* Architects, mechanical designers, and draftspeople uses graphics system to generate useful information like floor plans of building etc
* Graphics can be used to develop and manipulate maps to display geographical and celestial information
* workers in the field of statistics uses graphics system (computer plotting packages) for generating plots that aid the viewer in understanding the information in a set of data.
* 3-D data generated by CT (computed Tomography), MRI (magnetic resonance imaging), ultrasound, and positron-emission-Tomography (PET) using computer graphics
* Field of scientific visualization provides graphical tools that helps researchers interpret the vast quantity of data that they generate → working on supercomputer

Design

* Today, the use of interactive graphical tools in computer-aided design (CAD) pervades fields including as architecture, mechanical engg. the design of VLSI circuits, and the creation of characters for animations.
* In all these applications, graphics are used in number of ways (distinct ways)
for ex; in a VLSI design, the graphics provide an interactive interface b/w the user and the design package, usually by means of such tools as menus and icons.

Simulation and Animation

* once graphics systems evolved to be capable of generating sophisticated images in real time, engineers and researchers began to use them as simulators.
One of the most important uses has been in the training of pilots
* the success of flight simulators led to the use of computer graphics for animation in television, motion picture, and advertising industries.
(time varying behaviour of real and simulated objects)

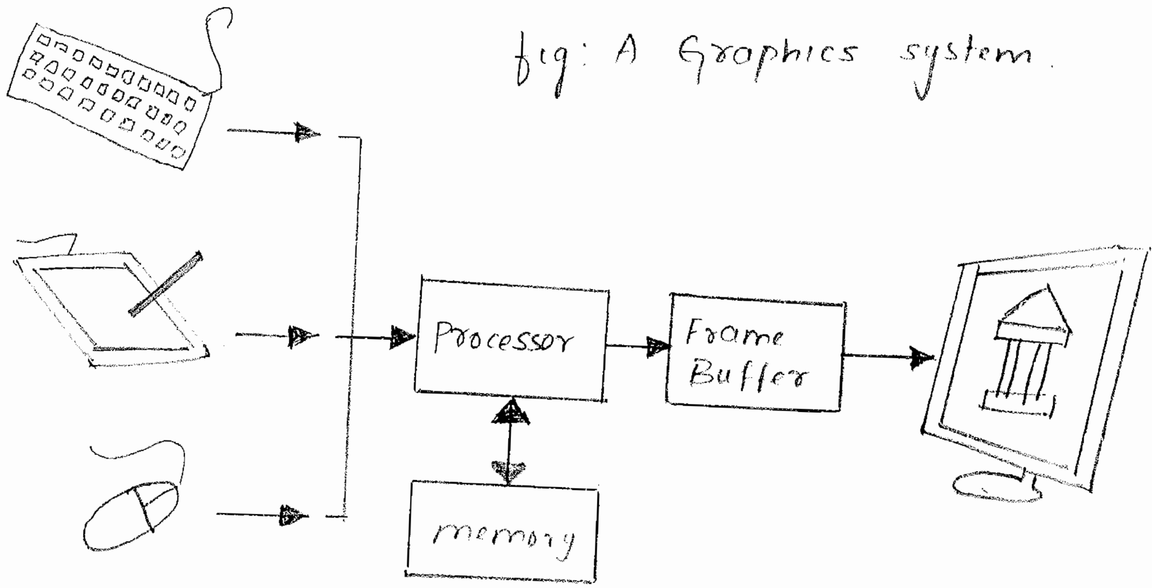
user interfaces.

* Most applications that run on PC and workstations have user interface that rely on desktop window system to manage multiple simultaneous activities.
* These user interface are working with computer graphics.

A GRAPHICS SYSTEM

* There are five major elements in a graphics system.
   1. Input devices
   2. processor
   3. Memory
   4. Frame Buffer
   5. output devices.

fig: A Graphics system.



pixels and the frame buffer (core element of graphic system)

* presently almost all graphics systems are raster based. A picture is produced as an array (the raster) of picture elements, or pixels within the graphics system. pixel is the smallest component of an image.
* Pixels are stored in a part of memory called Frame buffer. Its resolution (no. of pixels in the frame buffer) determines the detail that you can see in the image.
* Depth or precision of the frame buffer is defined as the number of bits that are used for each pixel. It determines the properties such as how many colors can be represented on a given system.

ex: 1-bit deep frame buffer allows only two colors.
  8-bit deep frame buffer allows $2^8 = 256$ colors.

if depth = 24 bits per pixel or more, such a systems are called <u>Full color systems</u> or <u>True color systems</u> or <u>RGB-color systems</u>.

* Frame buffer usually is implemented with special types of memory chips that enable fast redisplay of contents of frame buffer.

* In simple systems, there may be only one processor, the CPU of the system, which must do both the normal processing & graphical processing.
 The main graphical function of a processor is to take specifications of graphical primitives (such as lines, polygons generated by application programs and to assign values to the pixels in the frame buffer that best represent these entities.
 For ex; A triangle is specified by its three vertices, but to display its outline by the three line segments connecting the vertices, the graphic system must generate a set of pixels that appear as line segments to the viewer.

 <u>Defn</u>: The conversion of geometric entities to pixel color and locations in the frame buffer is known as <u>rasterization</u> or <u>scan conversion</u>.
 (or) It is the process of converting graphical primitives (or images) into pixel representation or frame buffer representation

* In earlier graphic system, frame buffer was part of standard m/m that could directly accessed by CPU.
 Today, almost all graphic systems are characterized by special-purpose <u>graphics processing ~~sys~~ units</u> (GPU) which can be either on motherboard of the system or on graphic card
 the frame buffer is accessed through GPU & may be included in the GPU.

<u>Output Devices.</u>

* For many years, the dominant type of display has been the cathode ray Tube (CRT)
* Fig below shows simplified picture of a CRT.

Working :

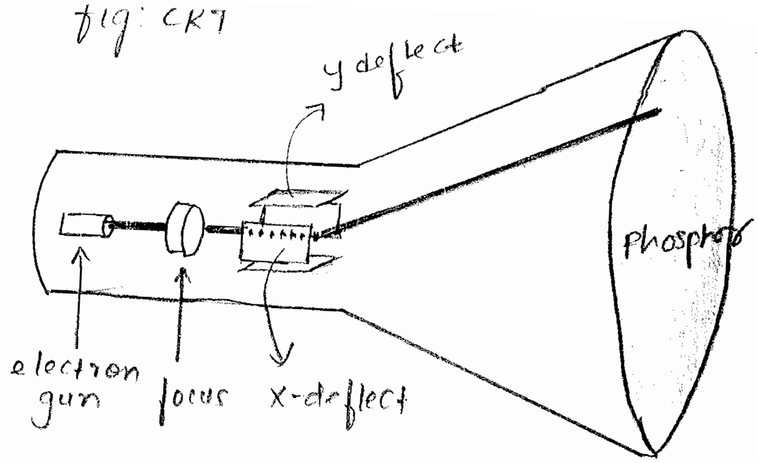


fig: CRT

* when electrons strike the phosphor coating on the tube, light is emitted. The direction of beam is controlled by two pairs of deflection plates.
* The output of computer is converted to voltages across the x and y deflection plates (by digital to analog converters)
* If the voltages steering the beam change at a constant rate, the beam will trace a straight line, visible to a viewer. Such a device is known as random scan, calligraphic, or vector CRT, because the beam can be moved directly from any position to any other position.

Refreshing.

* A typical CRT will emit light for only a short time (~ few milli seconds) after phosphor is excited by electron beam. Therefore, for a human to see a flicker free image, the same path must be retraced or refreshed, by the beam at a sufficiently high rate (refresh rate). In US → 60 cycles per sec or 60 Hz.
  rest of the world → 50Hz.

* In raster system, the graphics system takes pixels from the frame buffer and displays them as points on the surface of the display in one of two fundamental ways:

i.) Non interlaced or progressive display.
Here, pixels are displayed row by row, or scanline by scan line, at the refresh rate.

ii.) Interlaced display.
Here odd rows and even rows are refreshed alternatively
This method is used in Commercial TV.
In an interlaced display operating at 60Hz, the screen in redrawn in its entirety only 30 times per second, although visual system is tricked into ~~the~~ thinking the refresh rate is 60Hz rather than 30Hz.

Shadow-mask CRT (color CRT)

* color CRTs have 3 different colored phosphors (red, green, and blue), arranged in small groups. one common style arranges the phosphors in triangular groups called Triads, each triad consisting of three phosphors, one of each primary.

* Most color CRTs have three electron beams, corresponding to the three types of phosphors. In the shadow-mask CRT, a metal screen with small holes (shadow mask) ensures that an electron beam excites only phosphors of the proper color.

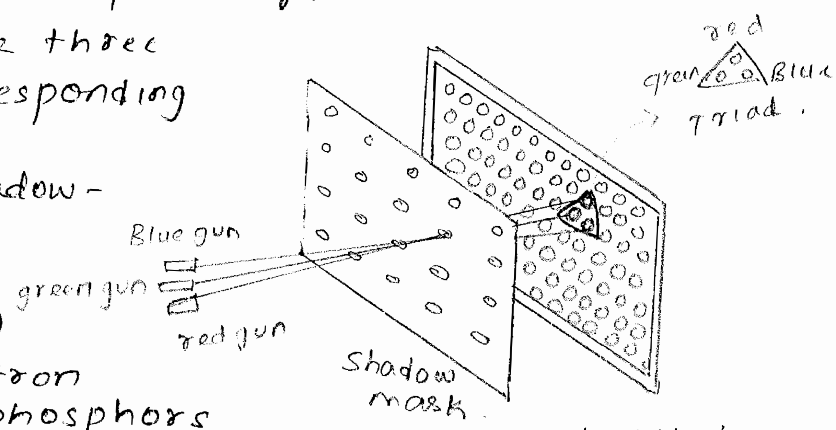
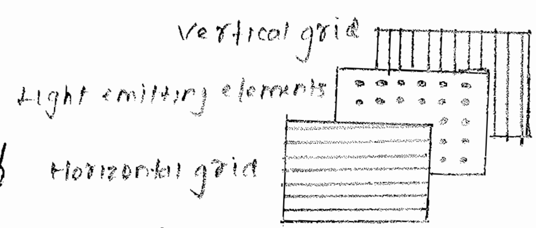


fig: shadow-mask CRT

Flat-screen Technologies.

* Flat panel monitors are are inherently raster.
* There are three technologies available.
  - → Light emitting diodes (LEDs)
  - → Liquid-crystal Display (LCD)
  - → Plasma panels.
* All these uses 2-D grid to address individual light emitting elements.
* Fig. shows a generic flat-panel display.
* By sending electric signals to the proper wire in each grid, the electric field at a location, determined by the intersection of two wires, can be made strong enough to control the corresponding element in the middle plate.



* Middle plate in an LED panel contains LEDs that can be turne on & off by electric signals sent to the grid.
* In an LCD display, electrical field controls the polarization o the liquid crystals in the middle panel, thus turning on & off the light passing through the panel.
* A plasma panel uses the voltage on the grids to energiz gases embedded ~~w~~ b/w the glass panels holding the grid. The energized gas becomes a glowing plasma

Aspect ratio.

* It is the width to height ratio of the frame buffer.
* Until recently, most displays had a 4:3 aspect ratio.
* VGA resolution → 640 × 480 resolution.

ie $\frac{640}{480}$ aspect ratio

* XGA → 1024 × 768.
* HDTV → 16:9.

Input Devices.

* common i/p devices are ; mouse, Joystick, & the data tablet. Each provides positional information to the system, and each usually are equipped with one or more buttons to provide signals to the processor.
* mouse → pointing devices, used for selection of an image, cut, copy, paste etc
* keyboard → used for inserting text based information.

IMAGES: PHYSICAL AND SYNTHETIC

note: Any object is always 3-Dimensional.
Image is always 2-Dimensional.
ie. An object from 3D is converted into an image in 2-D representation.

Objects and viewers

* Two basic entities that are part of any image-formation process are : objects, and viewers.

objects : physical structure (or instance) of an image.

* We form objects by specifying the positions in space of various geometric primitives (eg points, Lines, & polygons)
* In most graphics system, the vertices are sufficient to define most objects. eg: Line can be specified by two vertices.

viewers : one who forms the image of the objects.

* objects exists in real world & it is 3D. viewer sees an object in 2D (image)
* eg for viewer : Human, camera, digitizer.

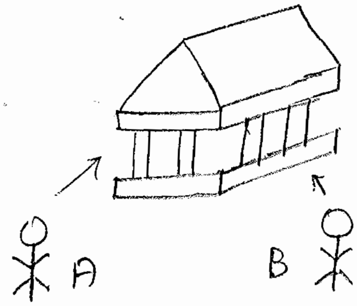
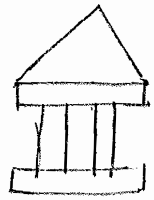
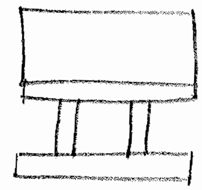


fig (a)



A

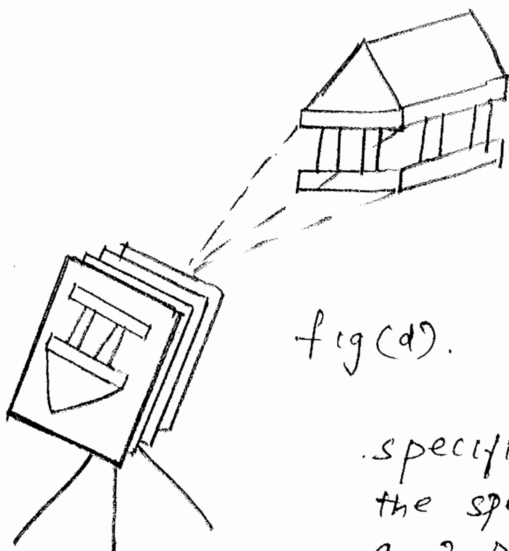
fig (b)



B

fig (c).

* Fig b shows the image seen by ~~per~~ human A. Fig c shows the image seen by human B.
* Fig d shows the image seen by camera.



camera system

fig (d).

Image Formation

It is the process by which the specification of the object is combined with the specification of the viewer to produce a 2-Dimensional image.

Light and images

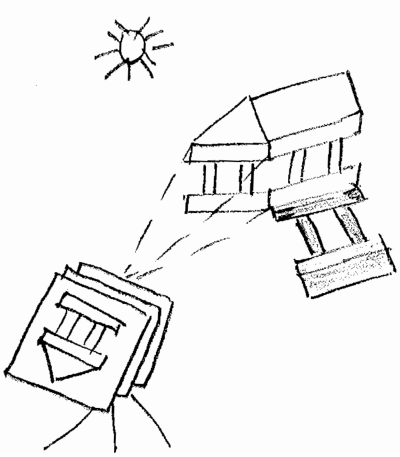
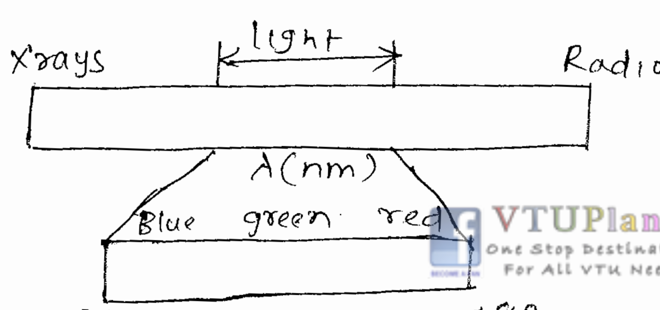


Fig: A camera system with an object and

* If there are no light sources, the objects would be dark, and there would be nothing visible in our object
* Light from the source strikes various surfaces of the object, & portion of reflected light enters the camera through the lens.
* Light is a form of electromagnetic radiation.
* portion of this spectrum is visible ~~for~~ & is called visible spectrum.

* visible spectrum has wavelengths in the range of 350 to 780 nm & is called (visible) Light.



note: Electromagnetic radiations (or energy) travels in the form of waves & measured in the form of wavelength.
the electromagnetic spectrum includes radiowaves, infrared, & a portion is our visual system.

image formation models

* Ray Tracing and photon mapping are image formation techniques that are based on light source, reflection, and luminousity of the object.

IMAGING SYSTEMS

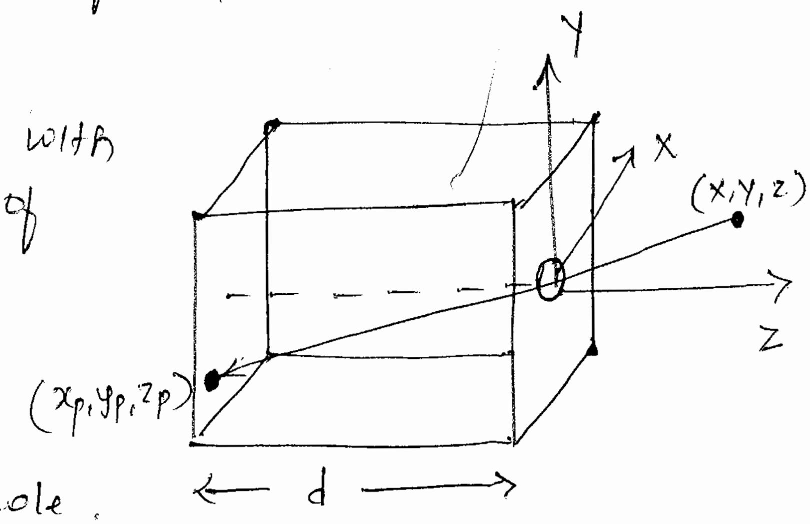
* Two physical imaging systems are
  → pinhole camera
  → Human visual system.

The pinhole camera

IMAGING SYSTEMS.

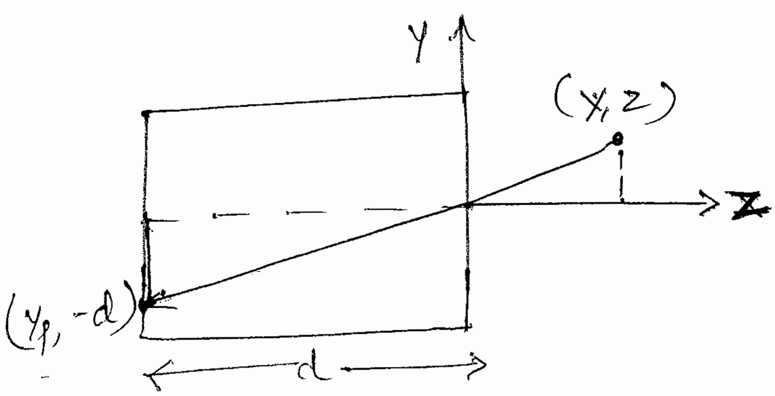
- ↳ pinhole camera
- ↳ Human visual system.

pinhole camera.



* pinhole camera is a box with small hole in the center of one side of the box.
  the film is placed inside the box on the side opposite to pinhole, at a distance d from pinhole.
  (hole will be so small)

* Fig shows sideview of pinhole camera.



To calculate where the image of the point (x,y,z) is on the film plane;
Two triangles are similar as shown above.

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

A similar calculation using top view yields

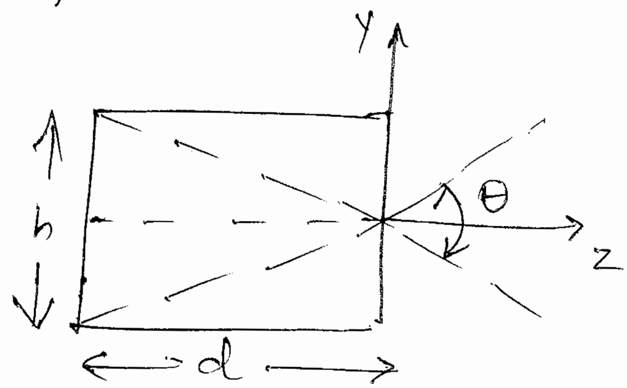
$$\boxed{x_p = \frac{x}{z/d}}$$

The point $(x_p, y_p, -d)$ is called the projection of the point (x, y, z)

The field, or Angle, of view

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

angle of view) $\theta = 2\tan^{-1}\frac{h}{2d}$.



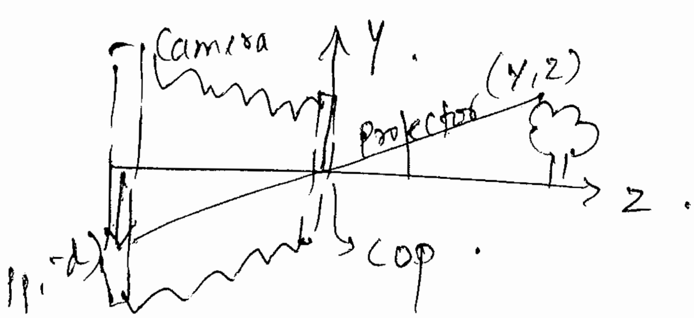
The ideal pinhole camera has an infinite depth of field. ↳ (d).

Disadv of pinhole camera.

1. pinhole is so small that it admits only a single ray from a point source. Almost no light enters the camera.
2. cannot be adjusted to have different angle of view (ie no zoom in or zoom out)

Some definitions.

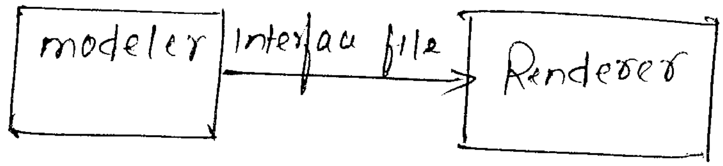
1. projector: we find the image of an point ~~of~~ on the object on the virtual image plane by drawing a line called projector, from the point to the center of lens or centre of projection (COP)



In our synthetic camera,
2. projection plane: virtual image plane that we ~~had~~ have moved in front of the lens is called projection plane.

The modeling - Rendering paradigm.

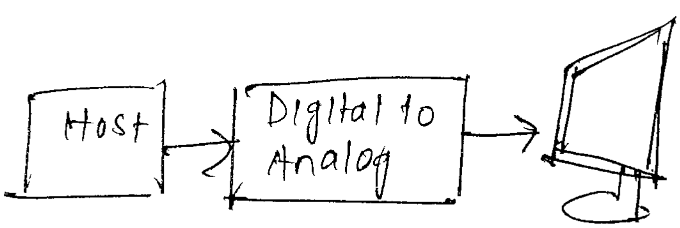
* Image formation is a two-step process (as shown in fig)
  → modelling
  → Rendering



* We might implement the modeler and renderer with different software and hardware.
* modeling involves designing and positioning our objects. & we dont work with detailed images of the objects here.
* Rendering involves adding light sources, material properties, and a variety of other detailed effects to form a production quality image.

GRAPHICS ARCHITECTURES.

Early graphics system: They used gen. purpose comp. with std von newmann architecture. It had single processor that processes single instrⁿ at a time. Display in these systems was based on a calligraphic CRT display.

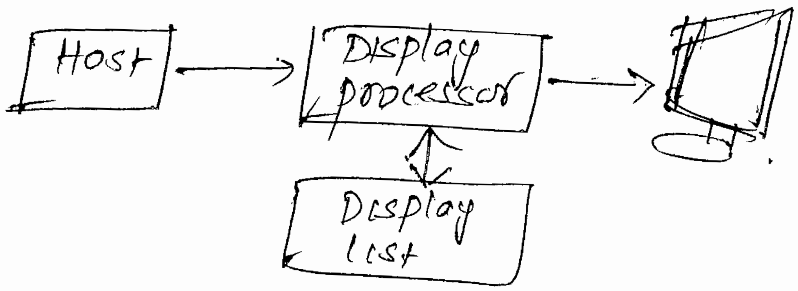


Job of Host comp was to run the applicⁿ prg and to compute end points of the line segments. CRT display includes necessary circuitory to generate lineseg. connecting two points.

Disadv: * Want to do both Job
* Slow ...

Display processors. (random based)

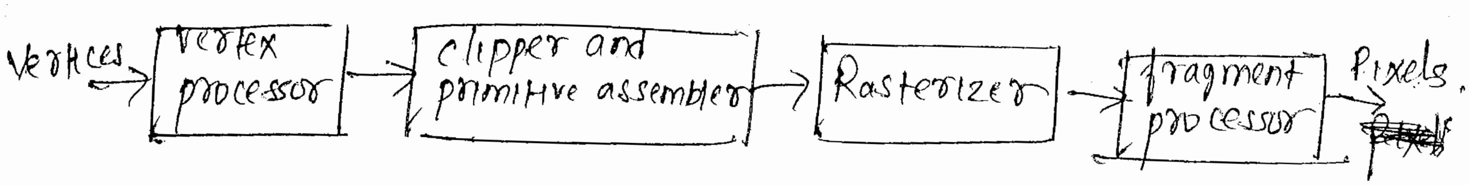
* fig shows Display processor architecture.



* relieves the gen. purpose processor from task of refreshing the display continuously.
* these display processors included instructions to display primitives on the CRT.
* Instructions to generate the image could be assembed once in the host & sent to the display processor, where they were stored in display processors own m/m as display list or display file (random based)
  ↳ Adv.
* Disadv.
  - → complex images can not be displayed (∵ difficult to convert them into instructions).
  - → Transformation is difficult.

Pipleline Architectures. (or) (Graphics pipeline)

* fig shows four major steps in imaging ~~bu~~ process in case of geometric or graphic pipe line.



* 4 major steps in imaging process.
  1. vertex processing.
  2. clipping and primitive assembly.
  3. Rasterization.
  4. Fragment processing.

note: each object comprises a set of geometrical primitives, each primitive comprises a set of vertices.

Vertex processing.

* It carries two functions.
  → coordinate transformations
  → compute color for each vertex.
* we can represent each change of coordinate systems by a ~~vecto~~ matrix. we can represent successive changes in coordinate systems by multiplying, or concatenating the individual matrices into a single matrix.
* Assignment of vertex colors can be simple or complex.

clipping and primitive assembly.

* checks whether the matrix is in the limit of our window and assemblies into appropriate format so that it fits to our window. If the image is too large - it clips it.
* o/p of this stage is a set of primitives whose projections can appear in the image.

Rasterization.

* Here the primitives ~~in the~~ represented in terms of their vertices are processed to generate pixels in the frame buffer.
* o/p of the rasterizer is a set of fragments for each primitive.

Fragment processing.

* Takes the fragments & updates the pixels in the frame buffer to accomodate transformations.

Computer Graphics.

Q: Explain 7 major groups of Graphic functions.

1. primitive functions.
2. Attribute functions.
3. Viewing functions.
4. Transformation functions.
5. Input functions.
6. Control functions.
7. Query functions.

primitive functions.

* They define the low level objects or atomic entities that our system can display.
  Depending on the API, the primitives can include points, line segments, polygons, pixels, text, etc

Attribute functions.

* Gives attributes to our primitives.
  They allow us to perform operations ranging from chosing the color with which we display line seg, to picking a pattern to which to fill the inside of a polygon, to selecting a typeface for the titles on a graph.

Viewing functions (specifies various views)

* It describes the synthetic camera.
  ie camera's position & orientation.
  This process will not only fix the view but also allow us to clip out objects that are too close or too far away.

Transformation functions.

* Allows us to carry out transformations of objects, such as rotation, translation, & scaling.

input functions

* For interactive applications, we need these to deal with the diverse forms of i/p that characterize modern graphic systems.
These functions deal with devices such as k/b, mice & ~~data~~ tablets.

Control functions

* enable us to :
  → communicate with the window system.
  → ~~to~~ initialize our programs.
  → deal with any errors that takes place during the execution of our program.

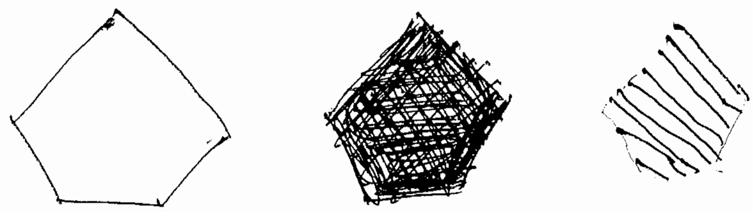
query functions.

* Allow us to retrieve information about o/p device (or system) for eg: how many colors are supported, or size of the display.
* Also allow us to know the camera parameters or values in the frame buffer.

Q: Explain polygon basics & different types of polygons in open GL.

* polygon is an object that has
  → A border that can be described by Line Loop.
  → well defined interior.
* performance of graphics system is characterised by no. of polygons per second that can be rendered.
* we can render ~~a polygon~~ :
  → only its edges
  → its interior with solid color or pattern.

* eg: methods of rendering a polygon

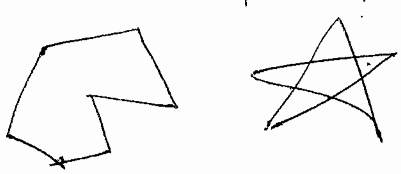


etc ...

* There are 3 properties of a polygon.
  - → simplex
  - → convex
  - → flat. → cost of testing whether a polygon is simple or not is very high.

in 2D, if no two edges of a polygon cross each other, we say its a simple polygon.

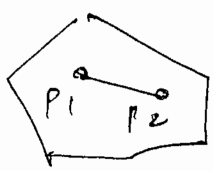
eg



→ non simple

An object is ~~conv~~ convex if all points on the line segment b/w any two points inside the object or on its boundary, are inside the object

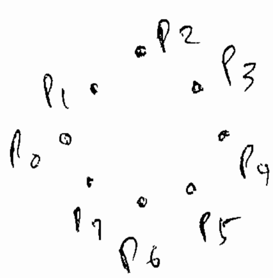
eg

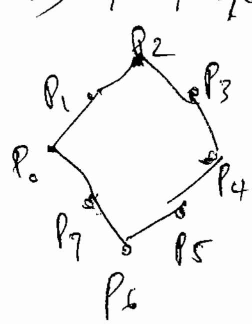


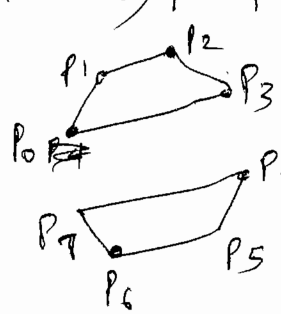
Flat:
↳ It means what is our dimension (2D or 3D)

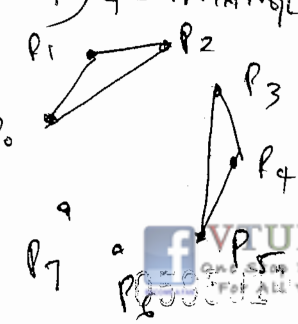
Polygon Types

1.) GL-POINTS 2.) GL-POLYGON 3) GL-QUADS 4.) GL-TRIANGLES

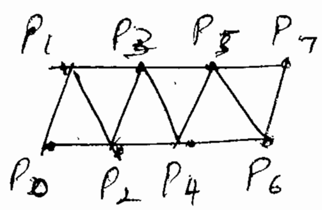


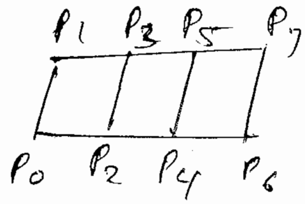


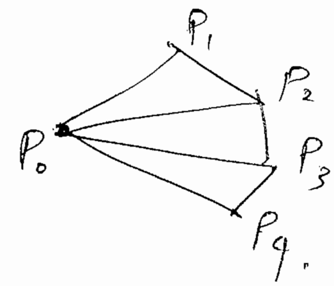




GL_TRIANGLE_STRIP. GL_QUAD_STRIP GL_TRIANGLE_FAN







polygons (GL_POLYGON)

* successive vertices define line segments, & a line segment connets the final vertex to the first.
* Interior is filled according to the state of relevant attributes.
* most graphics systems allow us to fill the polygon with color or pattern or to draw the lines around the edges but not to do both.
* We use the func. glPolygonMode to tell what we want.
* To do both, we have to render it twice, once in each mode.

Triangles & quadrilaterals (GL_TRIANGLES, GL_QUADS).

* These are special cases of polygons.
* Triangle → successive group of 3 vertices
* quadrilateral → — " — 4 — " —
* using these types may lead to a rendering more efficient.

strips & fans (GL_TRIANGLESTRIP, GL_QUAD_STRIP, GL_TRIANGLE_FAN).

* In triangle strip - each additional vertex is combined with the prev. two vertices to define new triangle,
* for quad strip - we combine two new vertices with prev. two vertices
* ~~In~~ triangle fan - Here one point is fixed. next two points determine the first triangle, & subsequent triangles are formed from one new point, the prev point, & the fixed point.

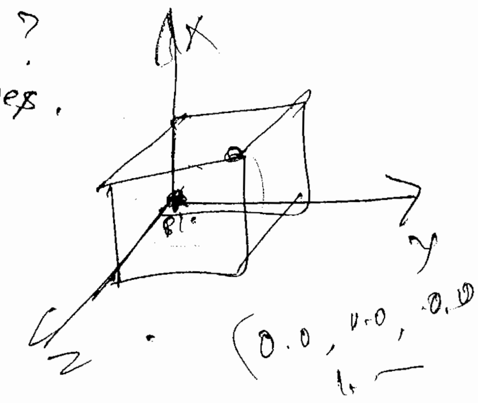
Q: Explain ~~RGB~~ color mode & index color mode.

~~RGB color~~. How color is handled in graphics system ~~from~~ programmer's perspective?

there are two diff approaches.

→ RGB-color model

→ indexed-color model.



RGB-color model.

* Here, there are conceptually separated buffers for red, green, & blue images.
* Each pixels has separate red, green, and blue components that corresponds to locations in memory (refer fig)
* Typically, each pixel might consists of 24 bits (3 bytes) 1 byte for each of red, green & blue. Here we can specify 2^{24} different possible colors (referred as 16 M colors) M denotes 1024^2.

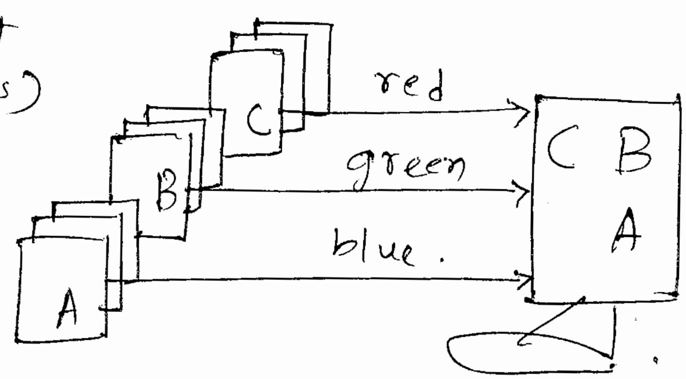


fig: RGB color

* We specify ~~a~~ color components in our API using color cube ~~as color components~~ as a number b/w 0.0 & 1.0 where 1.0 ⇒ max. (saturated), 0.0 ⇒ zero value of that primary.

Ex. to draw in red; we ~~use~~ issue following fun. call.

```
glcolor3f (1.0, 0.0, 0.0);
```

↳ sets current drawing color to red.

~~later,~~

* There is 4-color system also (RGBA). ~~system~~
  4^{th} color (A, or alpha) is also stored in frame buffer as are the RGB values.

  uses: for creating fog effects, combining images.

  $A = 0.0 \Rightarrow$ ~~opacity (opaque)~~ fully transparent
  $A = 1.0 \Rightarrow$ fully opaque.

eg: $\boxed{\text{glClearColor}(1.0, 1.0, 1.0, 1.0)}$

defines an RGB-color clearing color as white & opaque.

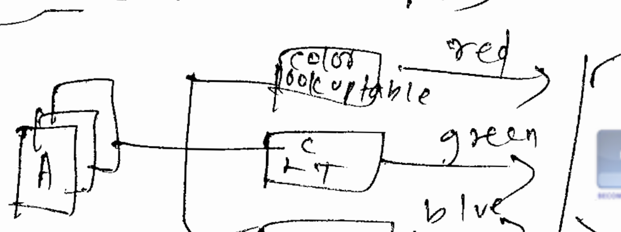
Indexed-color model.

* used for limited-depth pixels.
* Here color is selected by interpreting pixels as indices into a table of colors rather than as color values.
* suppose our frame buffer has k bits per pixel. Each pixel value or index is an integer b/w 0 and $2^k - 1$. suppose that we can display colors with a precision of m bits (ie we can chose from 2^m reds, 2^m greens, and 2^m blues) Hence we can produce any of 2^{3m} colors on the display, but the frame buffer can specify only 2^k of them.

we handle the specification thru' a user defined color look-up table that is of size $2^k \times 3m$ (refer fig)

i/p	Red	Green	Blue
0	0	0	0
1	2^m-1	0	0
:	0	2^m-1	0
:	!	!	!
2^k-1	!	!	!

once user has constructed this table, they can specify a color by its index, which points to appropriate entry in color-lookup table (refer below fig)



UNIT3: INPUT AND INTERACTION

Syllabus.

* Interaction
* Input Devices
* clients and servers
* Display lists
* Display lists and modelling
* programming event-driven input
* Menus
* picking
* A simple CAD program
* Building interactive models
* Animating interactive programs
* Design of interactive programs
* Logic operations.

– 7 Hours.

INTERACTION

* Definition:

Interaction in the field of computer graphics refers to the process of enabling the users to interact with computer displays.

The image change in response to the input from the user.

How it is supported by openGL?

* OpenGL does not support interaction directly. The main reason for this is to make openGL portable (ie to work on all types of systems irrespective of the hardware)

* However openGL provides the GLUT tool kit which supports minimum functionality such as opening of windows, use of keyboards and mouse and creation of pop up menus etc

[We discuss several interacting devices and the variety of ways that we can interact with them]

INPUT DEVICES

* We can think about input devices in two distinct ways

- ~~log~~ physical devices
- logical devices

(or) in computer graphics, ~~we can~~ the input devices (such as mouse, k/b etc) would be assesed from two perspectives.

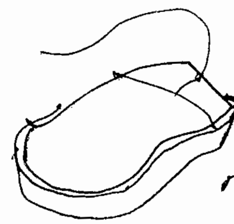
- physical perspective: depending on their physical properties
- logical perspective: The way these devices appear to the application program.

Explanations.

physical input Devices.

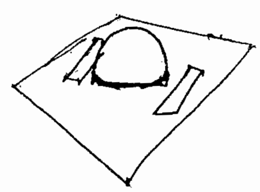
* From the physical perspective, each input device has properties that make it more suitable for certain tasks than for others.
* The two primary input devices are:
  (or) two primary types of physical devices are
  - pointing device.
    - Allows user to indicate a position on a display (ie allows user to return position)
  - Keyboard device
    - Allows user to return ASCII codes.

* Mouse and trackball are two commonly used pointing devices.



mouse

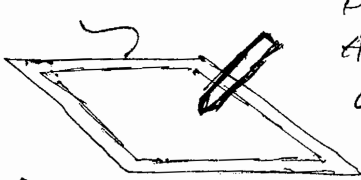
Both are similar in use and in construction.
In both the devices, the motion of the ball is converted into signals and sent to the computer.



Track ball.

o/p of both is two independent values provided by the device, which are considered as positions and converted to a 2D locn in ether screen / world coordinates.
In this mode, these devices are relative positioning devices because changes in the position of the ball yields a position in the user program: Absolute location of the ball (or mouse) is not used by the application program.

*

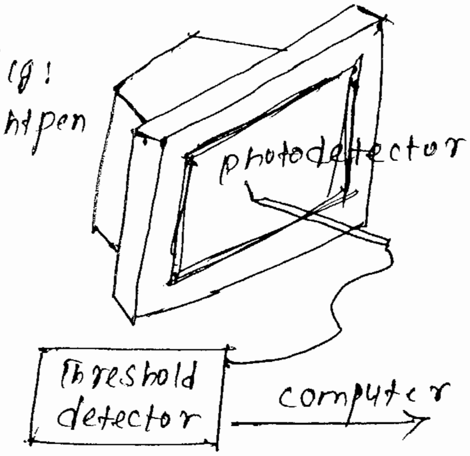


Data tablet

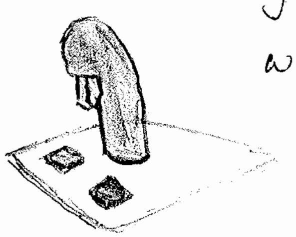
Data tablets provide for absolute positioning.
A typical data tablet has rows & columns of wires embedded under its surface.
The position of the stylus (pen) is determined through electromagnetic interactions b/w signals travelling through the wires and sensors in the st

* Light pen is one of the oldest input device in computer graphics.
It contains a light sensing device.
If the light pen is positioned on the face of the CRT at a location opposite where electron beam strikes the phosphor, the light emitted exceeds a threshold in the photodetector and a signal is sent to the computer.

fig: lightpen



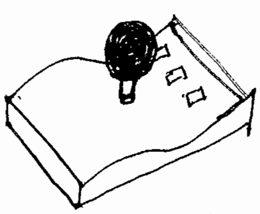
* Joystick is an input device in which the stick would move in orthogonally two directions
  - If the stick is left in the resting position, there is no change in the cursor position.
  - The farther the stick is moved from the resting position, the faster the screen location changes.



Joystick

Advantage of a joystick is that it is designed using mechanical elements such as springs and dampers which offer resistance to the user while pushing it. Such a mechanical feel is suitable for applications such as the flight simulators, game controllers etc.

* For 3D ~~applications~~ graphics, we might prefer to use 3D input devices (eg: spaceball, laser cameras)



space ball.

spaceball looks like joystick with the ball on the end of stick.
But, stick does not move, rather, pressure sensors in the ball measures the forces applied by the user.
It can measure not only 3 direct forces (up-down, front-back, right-left) but also three independent twists. ie the device measures 6 independent values & thus has six degrees of freedom.

Logical Devices.

+ From logical perspective, the input device is assesed by looking at it from inside the application program depending upon the "measurements" that the device returns to the user program and the "time" when the device returns those measurements.

+ From logical perspective, 6 classes of input forms can be identified such as.

- 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.

- Locator: A locator device provides position in world coordinates to the user program. it is usually implemented by means of pointing device (mouse or track ball)

- pick: A pick device returns an identifier of the object on the display to the user program. It is implemented with the same physical device as a locator, but has a separate s/w interface to the user program.

- choice : Choice device allows the user to select one out of a discrete number of options. A various widgets (a graphical interactive device, provided either by window system or tool kit, eg: menus, scroll bars, & graphical buttons) can be used to select one of n alternatives. It can be implemented either using k/b or mouse.

- Valuators: Valuator devices allow the user to provide analog i/p to the user prog. Dials and slide bars can be used for valuator inputs.

- strokes: A stroke device returns an array of locations. (III° to ~~map~~ multiple use of a locator). It is often implemented such that an action (say, pushing down a mouse button) starts the transfer of data into the specified array, and a second action (such as, releasing the button) ends this transfer.

Input modes.

* There are 3 different modes in which an input device provides input to the application program
  - Request mode
  - Sample mode
  - Event mode.

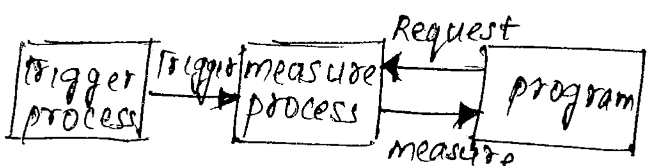
note: The manner by which input devices provide i/p to an application program can be described in terms of two entities: a measure process & a device trigger.
- The measure of a device is what the device returns to the user program.
- The trigger of a device is a physical input on the device ~~which~~ with which the user can signal the computer.

for eg: in k/b : measure - single char./strings of char.
trigger - Return or Enter key

in mouse: measure - position of cursor
trigger - press of a button.

Request mode.

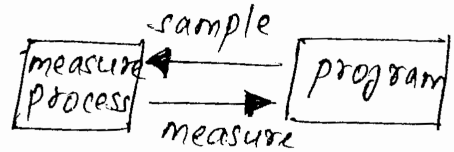
* In this mode, measure of the device is not returned to the program until the device is triggered.



Eg: If a C program requires a string input, a scanf function is used. When the program encounters scanf function (statement) it waits while we type the characters at our terminal (or K/b). All the data that is entered is stored in the keyboard buffer and its contents are given to the program only after the enter key is pressed.
↳ the trigger.

* The relation b/w measure and trigger for request mode is as shown in above figure.

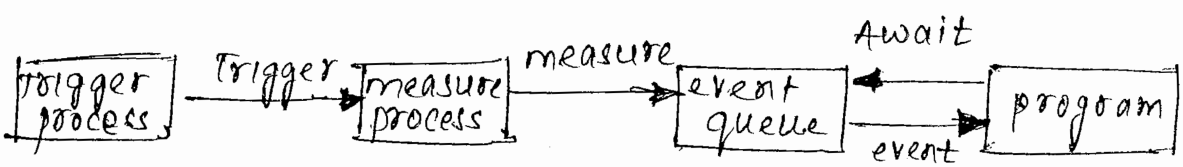
Sample Mode



* Sample mode input is immediate.
* As soon as the function call in the user program is encountered, the measure is returned. Hence no trigger is needed in this mode (refer figure)
* In this mode the user must have positioned the pointing device or entered data using the k/b before the function call, because the measure is extracted immediately from the buffer.

Event mode

* The request and sample mode can be used iff there is a single input device from which input is to be taken. They cannot be used if there are multiple input devices.
* Event mode must be used if there are inputs from a variety of input devices such as joystick, dials, buttons, switches etc



Here two approaches

i.) First Approach

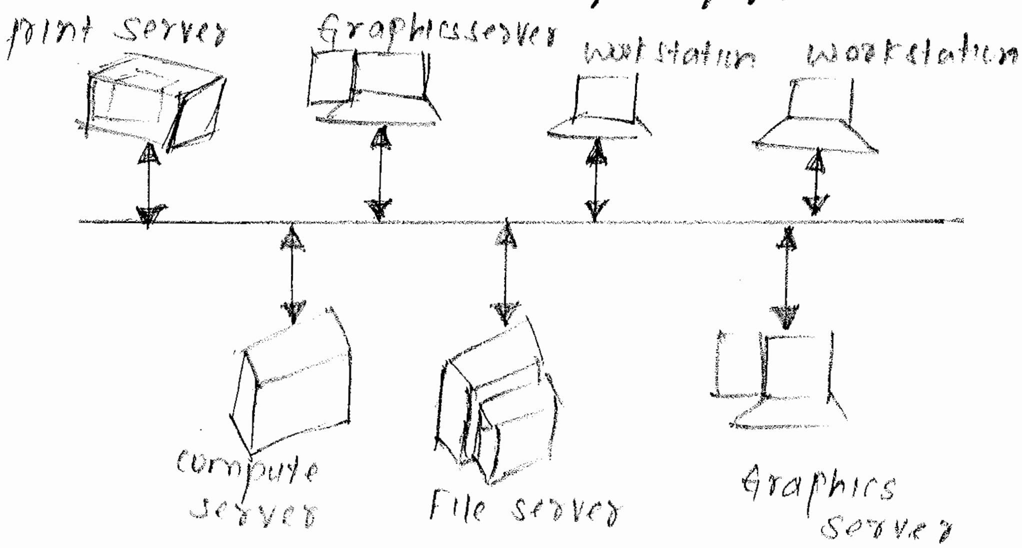
- Each time the device is triggered, an event is generated. The device measure, including the identifier for the device, is placed in an event queue.
- The applicn program can examine the front ~~eq~~ event in the queue (or wait if queue empty) and then decide what ~~to~~ do. (it can discard the front queue & look for next event).

ii.) Second approach.

- A specific callback function is associated with each type of input (event). The OS would regularly poll the event queue and executes the callbacks corresponding to the events in the queue

CLIENTS AND SERVERS

* The openGL application programs are treated as clients, and
the work station (computer) with a display, k/b & pointing device is treated as a graphics server.
Even if we have a single user isolated system, the interaction would be configured as a simple client-server network (refer fig)

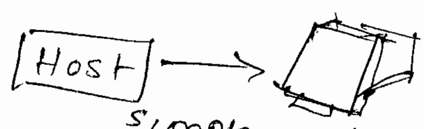


* This is to enable computer graphics to be useful under a variety of real applications.
Today most of the computing is distributed and network based. The building blocks are entities called "servers" that performs tasks for the "clients".
Clients and servers can be distributed over a n/w or can be contained entirely within a single computational unit.

DISPLAY LISTS

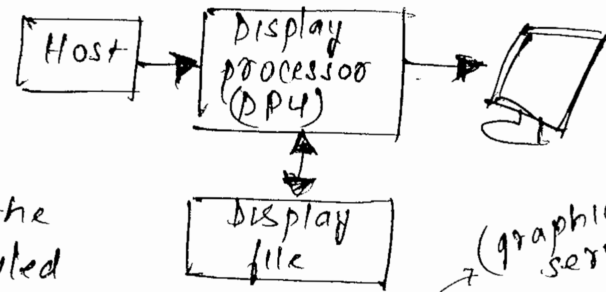
Display processor Architecture.

* original architecture of a graphics system was based on a gen. purpose comp (host) connected to display
* Disadv: comp were slow & expensive.



simple graphics architecture.

* Solution is to build a special purpose ~~process~~ comp called display processor with an org like the one shown below.
  It had limited instruction set (most are oriented towards drawing primitives on the display).



* User prog was processed in the host comp, resulting in compiled list of instructions that was sent to display processor (graphics server), where the instructions were stored in a display m/m as a display file or display list.

* We can send the graphical entities to a display in one of two ways
  - Immediate mode
  - Retained mode.

Immediate mode

- In this operation, as soon as the program executes a statement that defines a primitive, the primitive is sent to the graphics server for possible display and no memory of it is retained in the host.
- Disadv
  To redisplay the primitive after clearing screen, or to display it in a new position on the screen, the host program must resend the information through the display process. This would cause considerable traffic b/w client & server.

Retained mode

- In this mode, we define the object once and place its description (vertices, attributes, primitive types, viewing information etc.) in a display list.
- The display list is stored in the server and redisplayed by a simple function call issued from the client to the serv

Definition:
 Display lists are used to store the description of the objects which are to be displayed. The description would

Definition and execution of display lists.

* ~~Def~~ Display lists are defined in the same way as any geometric primitive is defined. There is a glNewList at the beginning and a glEndList at the end, with the contents in b/w.
* Each display list must have a unique identifier - an integer that is usually macro defined in the C program by means of a #define directive to an appropriate name for the object in the list.
* For ex: following code defines an red box & stores it in display list

```
#define BOX 1
glNewList (BOX, GL_COMPILE);
  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 ();
```

⇒ tells the system to send the list to the server but not to display its contents

* Each time we wish to draw the box, the client must execute a function

```
glCallList (BOX);
```

* If we change the model-view or projection matrices b/w executions of the display list, the box will appear in different places or will no longer appear, as the following code fragment demonstrates

```
glMatrixMode (GL_PROJECTION)
for(i=1 ;i<5; i++)
{  glLoadIdentity();
   gluOrtho2D (-2.0*i, 2.0*i, -2.0*i, 2.0*i);
}  glCallList(BOX);
```

* Each time the glCallList (BOX) is executed, the box is redrawn with a larger clipping rectangle which would not have been possible in immediate mode graphics.
* However, each time the display list is executed, the drawing color is set to red.
Unless the color is set to some other values, primitives defined subsequently in the program also will be colored red.
* A standard and safe procedure to overcome the above problem is to ~~us~~ always push both the attributes and matrices into their respective stacks when we enter a display list, and to pop them when we exit() as shown:

```
glPushAttrib(GL_ALL_ATTRIB_BITS);
glPushMatrix();
```

~~glCallList(BOX);~~ at the beginning of a display list &

```
glPopAttrib();
glPopMatrix();
```

at the end.

Adv.
- Reduced n/w traffic
- Allows much of the overhead in executing commands to be done once and have the results stored in the display list on the graphics server.

Disadv.
- Display lists require m/m on the server.
- There is an overhead involved in creating a display list.

PROGRAMMING EVENT DRIVEN INPUT

using the pointing devices

(How an event driven input can be programmed for a pointing device?)

* Mouse, Trackball, Data tablet etc can be categorized as pointing device.
* There are two types of events associated with the pointing device
  - move event
  - mouse event
* A 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 move event.
  After a move event, the position of the mouse (measure) is made available to the application program.
* A mouse event occurs when one of the mouse buttons is either pressed or released. A button being held down does not generate a mouse event until the button is released.
  The information returned to the program includes
  - The button that generated the event
  - The state of the button after the event (up or down)
  - position of the cursor in window coordinates (with origin in the upper left corner of the window)
* The mouse callback function must be registered in the main() function as shown below

```
glutMouseFunc(MyMouse);
```

* The mouse callback must have the form

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

* The above function must be written by the application programmer acc. to his requirements.
  Within the callback function, the programmer can define the actions that must take place if specified event occurs.

eg: If we want the pressing of the left mouse button to terminate the program, then the myMouse call back function must be designed as follows.

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

{pressing of other buttons results in no response, since no action is defined for them}

Eg 2: program to draw a small Box at each location on the screen where the mouse cursor is located at the time that the left button is pressed. ~~we~~ use the middle button to terminate the program.

```
GLsizei wh=500, ww=500; /* initial window width & height */
GLfloat size=3.0; /* one half of sidelength */

void myInit ()
{
    glViewport (0, 0, ww, hh);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluOrtho2D (0.0, (GLdouble) ww, 0.0 (GLdouble) wh);
    glMatrixMode (GL_MODELVIEW);
    glClearColor (0.0, 0.0, 0.0, 1.0);
    glColor3f (1.0, 0.0, 0.0); /* red squares */
}
```

```
void drawSquare (int x, int y)
{   y = wh-y;
    glBegin (GL_POLYGON);
        glVertex2f (x+size, y+size);
        glVertex2f (x-size, y+size);
        glVertex2f (x-size, y-size);
        glVertex2f (x+size, y-size);
    glEnd();
    glFlush();
}
void myDisplay()
{ glClear (GL_COLOR_BUFFER_BIT);
}
void myMouse(int btn, int state, int x, int y)
{   if(btn==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
        drawsquare(x, y);
    if (btn==GLUT_RIGHT_BUTTON && state== GLUT_DOWN)
        exit(0);
}

int(main int argc, char **argv)
{   glutInit (&argc, argv);
    glutInitWindowSize (ww, wh);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutCreateWindow ("square");
    myInit();
    glutReshapeFunc(myReshape);
    glutMouseFunc (myMouse);
    glutDisplayFunc (myDisplay);
    glutMainLoop();
}
```

Window Events

(How an event driven input can be programmed for an window event?)

* Resizing a window is one of the example for window events.
  Resizing a window is done usually by using a mouse to drag a corner of the window to a new location.
* If such an event occurs, we have to consider three questions
  - Do we ~~draw~~ redraw all the objects that were in the window before it was resized?
  - What do we do if the aspect ratio of new window is different from that of the old window?
  - Do we change the sizes ~~of~~ or attributes of new primitives if the size of the new window is different from that of old?
* The window event must be registered in the main function using glutReshapeFunc (myReshape);

The window event (reshape event) returns its measure, the height, & width of the new window. So myReshape() must be of the form

~~void~~ myReshape (GLsizei w, GLsizei h);

The above program must be written by the applicⁿ programmer acc. to his requirements.

eg:
```
void myReshape(GLsizei w, GLsizei h)
{ glMatrixMode(GL_PROJECTION)
  glLoadIdentity();
  gluOrtho2D (0.0, (GLdouble)w, 0.0, (GLdouble) h);
  glMatrixMode (GL_MODELVIEW);
  glLoadIdentity();
  glViewport(0, 0, w, h);  —> adjust viewport & clear
```
adjust clippin box

WWEW: Learn new window size in

Keyboard Events

(How event driven input can be programmed for a Keyboard device?)

* Keyboard is an input device.
Keyboard events are generated when the mouse (cursor) is in the window and one of the keys is pressed or released

* The keyboard event must be registered in the main function using -

```
glutKeyboardFunc (mykey);
   (or)
glutKeyboardUpFunc (mykey);
```

The above given callback functions are for events generated by pressing a key and for releasing a key respectively.

* When the k/b event occurs, the ASCII code for the key that generated the event and the location of the mouse are returned. So the mykey() must be of the following form

```
mykey (unsigned char key, int x, int y);
```

The above program must be written by the application programmer acc to his requirements.

eg: If we wish to use the k/b to only exit from the program, then it can be done as shown.

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

UNIT 4 :

GEOMETRIC OBJECTS AND TRANSFORMATION - I

Syllabus

* Scalars, ~~points~~, and vectors
* Three dimensional primitives
* Coordinate systems and Frames
* Modeling a colored cube
* Affine Transformations
* Rotation, Translation, and scaling

- 6 Hours.

Ashok Kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

* Minimum set of primitives from which we can obtain (build) more sophisticated objects are –

1. Scalars
2. points
3. Vectors

} Three basic elements.

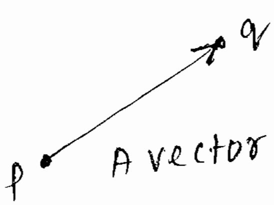
SCALARS, POINTS, AND VECTORS

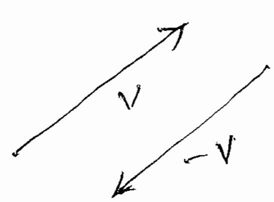
Definitions

* A point is a fundamental geometric object.
  A point is a location in space. Only property a point possess is location. It has neither shape nor a size.

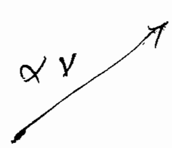
* Scalars are quantities which have magnitude.
  They obey operations of ~~geometry~~ ordinary arithmetic. Hence they obey addition, multiplication, subtraction, division, associativity and commutative rules.
  Scalars alone have no geometric properties.

* A vector allows us to work with directions.
  They have both magnitude and direction.
  A directed line segment is a vector since it has both a magnitude (length) & direction (orientation)

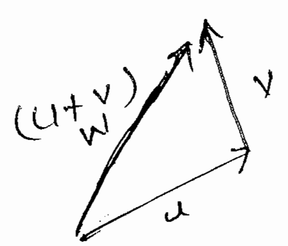




inverse vectors (same magnitude but opposite in direction)



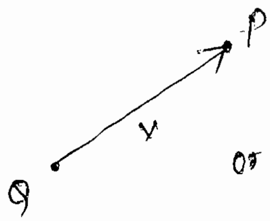
every vector can be multiplied by a scalar.



sum of any two vectors is a vector. (use head-to-tail axiom)

Some more ex:

identical vectors
(same mag & directⁿ)



$v = P - Q$
or $P = v + Q$ } → point - point subtraction yields a vector.
It is equivalent to point - vector addition.

* +) <u>Euclidean space</u> is an extension of a vector space that adds a measure of size and distance and allows us to define such things as the length of a line segment.
* An <u>Affine space</u> is an extension of the vector space that includes additional type of object: the point

operations in Affine space -

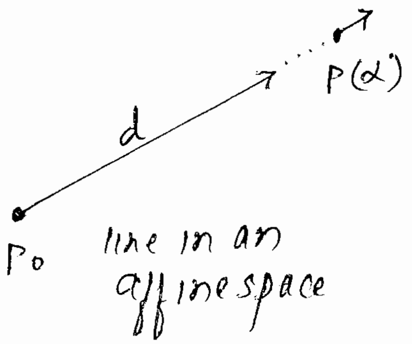
- Vector - Vector Addition
- Scalar - Vector Multiplication
- point - vector addition
- Scalar - Scalar operations.

note: For any point,

$$1 \cdot P = P$$
$$0 \cdot P = \emptyset \text{ (zero vector)}$$

Lines

* Sum of point and a vector ~~or~~ (or subtraction of two points) leads to the notion of a line in an affine space.



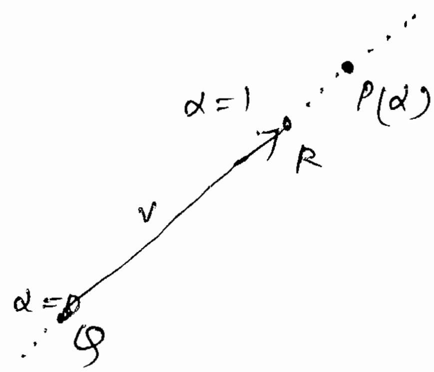
$P(\alpha) = P_0 + \alpha d$

It is the set of all points that passes through P_0 in the direction of a vector d.

Affine sum

$P = Q + \alpha v$ —(1)

describes all the points ~~from~~ on the line from Q in the direction of v. WKT -



$v = R - Q$

therefore (1) ⇒

$P = Q + \alpha(R - Q)$

$= \alpha R + (1-\alpha) Q$.

or $\boxed{P = \alpha_1 R + \alpha_2 Q}$ where $\alpha_1 + \alpha_2 = 1$.

↳ It is called Affine sum of points P & Q.

III^rly Affine sum of points $P_1, P_2 \ldots P_n$ is -

$P = \alpha_1 P_1 + \alpha_2 P_2 + \ldots + \alpha_n P_n$ where $\alpha_1 + \alpha_2 \ldots \alpha_n = 1$.

If $\alpha_i >= 0$ $i = 1, 2 \ldots n$ then it is called the <u>Convex Hull</u> of the set of points.

Dot and Cross product of vectors

Dot product

* Dot product of u & v is written as $u \cdot v$

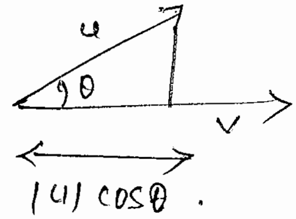
If $u \cdot v = 0$, u and v are said to be orthogonal

The square of magnitude of a vector is given by the dot product.

$$|u|^2 = u \cdot u$$

Cosine of the angle b/w two vectors is given by -

$$\cos\theta = \frac{u \cdot v}{|u||v|}$$



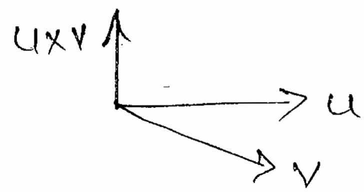
$|u|\cos\theta = \frac{u \cdot v}{|v|}$ is the length of the orthogonal projection of u onto v

cross product

if u & v are two vectors, then the third vector orthogonal to both u & v can be computed as

$n = u \times v$, it is called the cross product of u & v

$$|\sin\theta| = \frac{|u \times v|}{|u||v|}$$



planes

* A plane in an affine space can be defined as a direct extension of the parametric line.

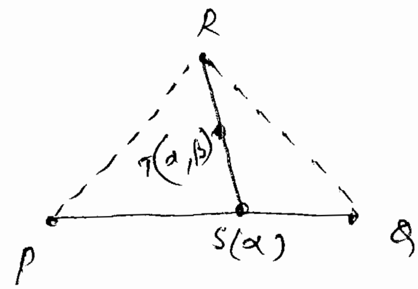
Three points that are not on the same line determines a unique plane.

suppose P, Q, R are such points in an affine space.

line seg. that joins P and Q is the set of points of the form –

$$S(\alpha) = \alpha P + (1-\alpha) Q.$$

$$0 \le \alpha \le 1.$$



Suppose that we take an arbitrary point on this line seg and form a line seg from this point to R. using second parameter β, we can describe points along this line seg as

$$T(\beta) = \beta S + (1-\beta)R \qquad 0 \le \beta \le 1.$$

THREE DIMENSIONAL PRIMITIVES.

* three features characterize 3D objects that fit well with existing graphics h/w & s/w

1. Objects are described by their surfaces and can be thought of as being hallow.

⇒ graphic package reads only 2D primitives to model 3D objects.

2. objects can be specified through a set of vertices in 3D.

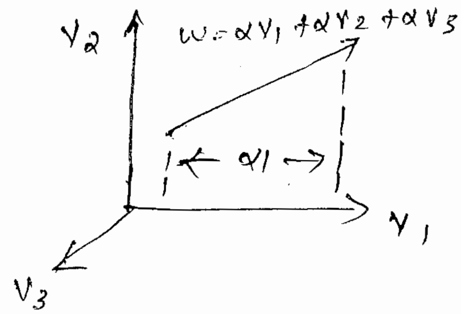
⇒ we can use pipeline architecture to process these vertices at high rates and generate images of object during rasterization

3. Objects are either composed of or can be approximated by flat, convex polygons

COORDINATE SYSTEMS AND FRAMES

* In a 3D vector space, we can represent any vector w uniquely in terms of any 3 linearly independent vectors v_1, v_2, v_3 as -

$$w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$$



The scalars $\alpha_1, \alpha_2, \alpha_3$ are called components of w w.r.t the basis v_1, v_2, v_3

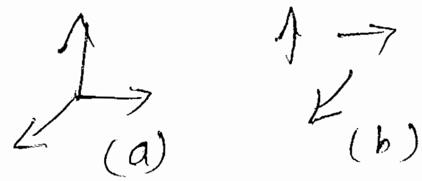
We can write the representation of w w.r.t ~~to~~ this basis as the column matrix -

$$a = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$$

and for basis vectors -

$$v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

note :



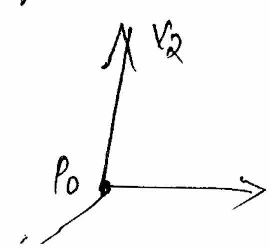
Both are correct since vectors have no fixed location

then,

$$\boxed{w = a^T v} \quad \text{ie} \quad \boxed{w = [\alpha_1\ \alpha_2\ \alpha_3] \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}}$$

Frames

* A coordinate system is not sufficient to represent points.
* If we work in an affine space, we can add a single point, (the origin), to the basis vectors to form a frame.



A point p can be represented in terms of basis vectors as -

$$p = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3$$

$$\boxed{P = P_0 + b^T . v} \quad \dots \quad \begin{bmatrix} \beta_1 \\ \end{bmatrix}$$

ie :

- A frame is determined by (P_0, v_1, v_2, v_3)
- within this frame,
  - Every vector can be written as -

    $w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$
  - Every point can be written as

    $p = P_0 + \beta_1 v_1 + \beta_2 v_2 + \beta_3 v_3$.

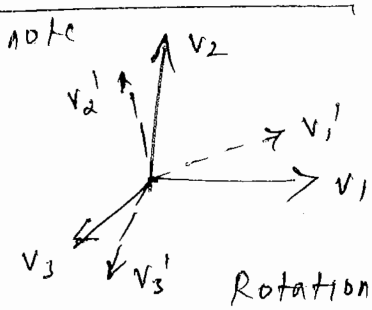
Change of coordinate systems.

+ Consider that $[v_1, v_2, v_3]$ and $[u_1, u_2, u_3]$ are two bases. Each basis vector in the second set can be represented in terms of the first basis vector and vice versa.

Hence there exists 9 scalar components $\{\gamma_{ij}\}$ as -

$$u_1 = \gamma_{11} v_1 + \gamma_{12} v_2 + \gamma_{13} v_3$$
$$u_2 = \gamma_{21} v_1 + \gamma_{22} v_2 + \gamma_{23} v_3$$
$$u_3 = \gamma_{31} v_1 + \gamma_{32} v_2 + \gamma_{33} v_3$$

note



Rotation & scaling of a basis.

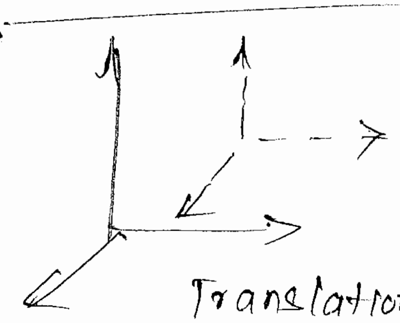
The 3x3 matrix

$$M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} \\ \gamma_{21} & \gamma_{22} & \gamma_{23} \\ \gamma_{31} & \gamma_{23} & \gamma_{33} \end{bmatrix}$$ defines these scalars.

Therefore,

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$$

or $u = Mv$.



Translation of a basis

* The matrix M contains the information to go from a representation of a vector in one basis to its representation in the second basis. The inverse of M gives the matrix representation of the change from $\{u_1, u_2, u_3\}$ to $\{v_1, v_2, v_3\}$

* Consider the vector w that has the representation $\{\alpha_1, \alpha_2, \alpha_3\}$ w.r.t $\{v_1, v_2, v_3\}$

ie $w = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3$

$= a^T v$ where $a = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \end{bmatrix}$ & $v = \begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix}$

Assume that b is the representation of w wrt $\{u_1, u_2, u_3\}$

ie $w = \beta_1 u_1 + \beta_2 u_2 + \beta_3 u_3$

$= b^T u$.

Then, expressing second bases in terms of first basis -

$w = b^T . u$

$= b^T . Mv$

$= a^T v$

Thus, $a = M^T b$. $(\because a^T = b^T M)$

The matrix, $T = (M^T)^{-1}$ takes us from a to b, through the simple matrix eqn -

$$\boxed{b = Ta}$$

Example:

Suppose we have a ~~matrix~~ vector 'w' whose representation is $a = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ along the basis vectors v_1, v_2, v_3

ie $w = v_1 + 2v_2 + 3v_3$.

suppose that we make a new basis from the vectors v_1, v_2, v_3

$$u_1 = v_1$$
$$u_2 = v_1 + v_2$$
$$u_3 = v_1 + v_2 + v_3$$

Then the matrix, M is

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

The above matrix converts the basis v_1, v_2, v_3 to u_1, u_2, u_3.

To do the opposite, the matrix is -

$$T = (M^T)^{-1}$$
$$= \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{bmatrix}^{-1} = \begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix}$$

In the new system, the representation of w is -

$$b = Ta$$
$$= \begin{bmatrix} 0 & -1 & 0 \\ 0 & 1 & -1 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} = \begin{bmatrix} -1 \\ -1 \\ 3 \end{bmatrix}$$

ie $\boxed{w = -u_1 - u_2 + 3u_3}$

Homogeneous Coordinates

* **Homogeneous coordinates** avoid potential confusion between a vector and point by using a 4-D representation for both points and vectors in 3-D.

1. In the frame specified by (v_1, v_2, v_3, P_0), any point P can be written uniquely as -

$$\boxed{P = \alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 + P_0}$$

or $$P = [\alpha_1 \ \alpha_2 \ \alpha_3 \ 1]^T \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}$$

note
$0 \cdot P = 0$
$1 \cdot P = P.$

Thus, P is represented by the column matrix -

$$P = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 1 \end{bmatrix}$$

2. In the same frame, any vector w can be written as.

$$\boxed{w = \delta_1 v_1 + \delta_2 v_2 + \delta_3 v_3}$$

or $$w = [\delta_1 \ \delta_2 \ \delta_3 \ 0]^T \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}$$

Thus, w is represented by the column matrix -

$$w = \begin{bmatrix} \delta_1 \\ \delta_2 \\ \delta_3 \\ 0 \end{bmatrix}$$

change in frame

* If (v_1, v_2, v_3, P_0) and (u_1, u_2, u_3, Q_0) are two frames, then we can express the basis vectors and reference point of second frame in terms of the first as-

$$u_1 = \gamma_{11} v_1 + \gamma_{12} v_2 + \gamma_{13} v_3$$

$$u_2 = \gamma_{21} v_1 + \gamma_{22} v_2 + \gamma_{23} v_3$$

$$u_3 = \gamma_{31} v_1 + \gamma_{32} v_2 + \gamma_{33} v_3$$

$$Q_0 = \gamma_{41} v_1 + \gamma_{42} v_2 + \gamma_{43} v_3 + P_0 .$$

In matrix form -

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} \quad \text{where } M = \begin{bmatrix} \gamma_{11} & \gamma_{12} & \gamma_{13} & 0 \\ \gamma_{21} & \gamma_{22} & \gamma_{23} & 0 \\ \gamma_{31} & \gamma_{32} & \gamma_{33} & 0 \\ \gamma_{41} & \gamma_{42} & \gamma_{43} & 1 \end{bmatrix}$$

M is called the matrix representation of the change of frames.

* We can also use M to compute the changes in the representations directly.

Suppose a & b are homogeneous - coordinate representations either of two points or of two vectors in the two frames. Then -

$$b^T \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ Q_0 \end{bmatrix} = b^T M \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix} = a^T \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ P_0 \end{bmatrix}$$

Hence $\boxed{a = M^T b}$

where $M^T = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Example -

Assume the two frames with basis vectors having the following relations -

$$u_1 = v_1$$
$$u_2 = v_1 + v_2$$
$$u_3 = v_1 + v_2 + v_3$$

the reference point does not change. so -

$$Q_0 = P_0 .$$

The matrix in homogeneous form is -

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

Suppose that in addition to changing the basis vectors, we also want to move the reference point to that point that has the representation $(1, 2, 3, 1)$ in the original system.

then, $Q_0 = v_1 + 2v_2 + 3v_3 + P_0$

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 2 & 3 & 1 \end{bmatrix}$$

$$\therefore T = (M^T)^{-1} = \begin{bmatrix} 1 & -1 & 0 & 1 \\ 0 & 1 & -1 & 1 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* Note that ~~the~~ p is a point $(1,2,3)$ in the original ~~buffe~~ frame. Then the point p' in the new frame is

$$\begin{bmatrix} 1 & -1 & 0 & 1 \\ 0 & 1 & -1 & 1 \\ 0 & 0 & 1 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 \\ 2 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

↳ The origin in new system..

However, A vector $(1,2,3)$ which is represented as

$$a = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 0 \end{bmatrix}$$ in original system is transformed to

$$b = \begin{bmatrix} -1 \\ -1 \\ 3 \\ 0 \end{bmatrix}$$ in the new system.

MODELLING A COLORED CUBE

* A <u>cube</u> is a simple 3D object.

We start by assuming that the vertices of the cube are available through an array of vertices.

For ex -

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

we can use these list of points to specify the faces as -

```
glBegin (GL_POLYGON)
    glVertex3fv (vertices[0]);
    glVertex3fv (vertices [3]);
    glVertex3fv (vertices [2]);
    glVertex3fv (vertices[1]);
glEnd();
```

III'ly the other 5 faces can be defined.

The other five faces are -

(2,3,7,6) , (0,4,7,3) , (1,2,6,5)

(4,5,6,7) , (0,1,5,4)

* A vertex list data structure can be used to represent a cube as shown -

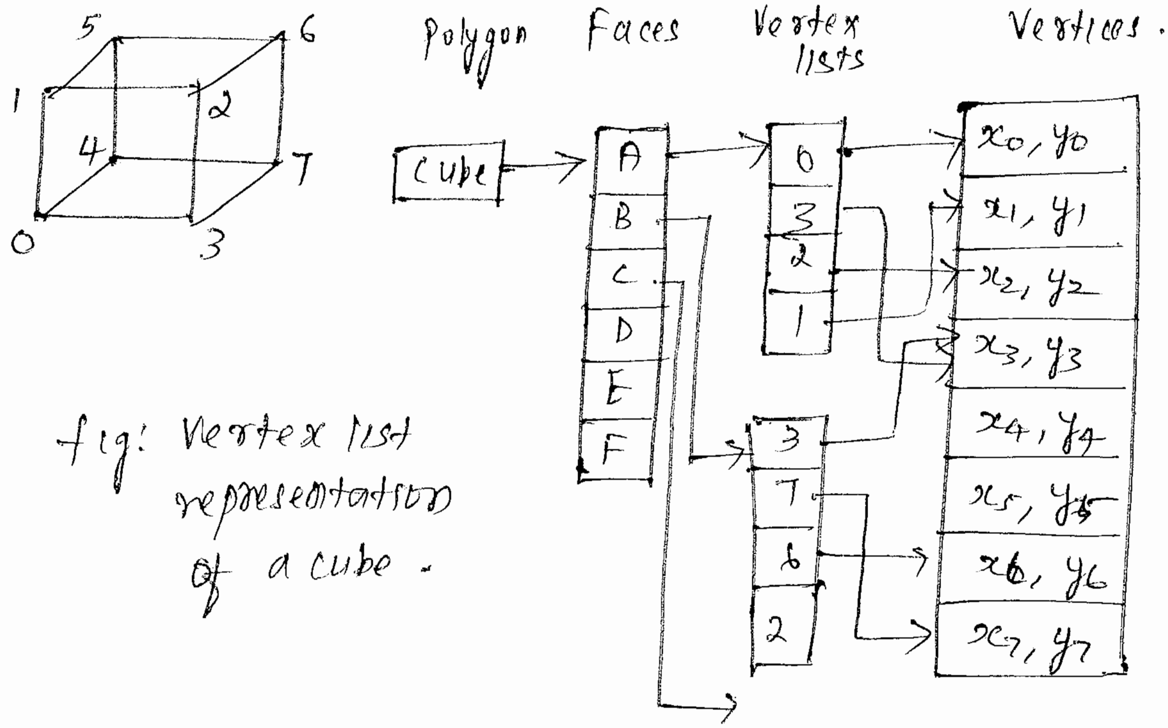


fig: Vertex list representation of a cube.

* A cube is composed of 6 faces. Each face is a quadrilateral which meets other quadrilateral at vertices.
  Each vertex is shared by ~~6~~ 3 faces. Each Edge is shared by 2 faces.

* vertex arrays must first be enabled as -

```
glEnableClientState (GL_VERTEX_ARRAY);
glEnableClientState (GL_COLOR_ARRAY);
```

The openGL must be specified as to where and in what format the vertex arrays are, using -

```
glVertexPointer (3, GL_FLOAT, 0, vertices);
glColorPointer (3, GL_FLOAT, 0, colors);
```

The adv of vertex arrays is that each geometric location appears only once. Instead of being repeated each time, the vertex needs a change only once.

* Bilinear Interpolation is used for coloring.
If c_0, c_1, c_2, & c_3 are colors assigned to the vertices in the application programe. Then the first interpolation is used to interpolate colors along the edges b/w vertices 0 and 1, 2 and 3 creating RGB colors using the parametric equations.

$$c_{01}(\alpha) = (1-\alpha)c_0 + \alpha c_1$$

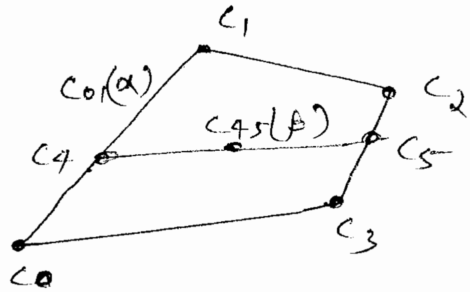


fig: Bilinear interpolation.

* As ~~col~~ α goes from 0 to 1, we generate different colors combination. This process continues until entire cube is colored.

* The code for generating a color cube is -

GLfloat vertices [8][3] = { same as before };

GLfloat colors [8][3] = { same as vertices. };

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

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

AFFINE TRANSFORMATIONS

* A Transformation is a function that takes a point (or vector) and maps that point into another point (or vector)

we can picture such a function by looking at fig below or by writing down the functional form -

$q = T(p)$ for points

$v = R(u)$ for vectors

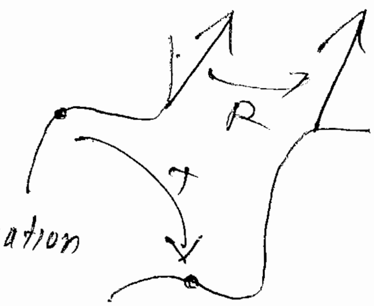


fig: Transformation

* if we use Homogeneous coordinates then we can represent both vectors and points as 4D column matrix as -

$$q = f(p)$$

$$v = f(u)$$

* using homogeneous coordinates, a linear transformation transforms the representation of a given point into another representation as (Lr or vector)

$$v = Cu$$

where

$$C = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ \alpha_{41} & 0 & 0 & 1 \end{bmatrix}$$

12 values of α can ~~are~~ set arbitrarily & said that the transformation has 12 degrees of freedom.

* A point is represented in affine space as

$$p = \begin{bmatrix} \beta_1 \\ \beta_2 \\ \beta_3 \\ 1 \end{bmatrix}$$

A vector is represented as -

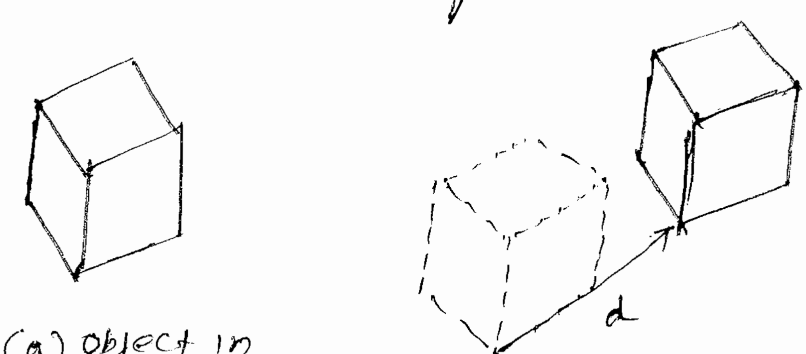
$$u = \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ 0 \end{bmatrix}$$

Hence a point has 12 degrees of freedom where as a vector has only 9.

TRANSLATION, ROTATION, AND SCALING.

Translation

* Translation is an operation that displaces points by a fixed distance in a given direction (refer fig)



(a) object in original position (b) object translated.

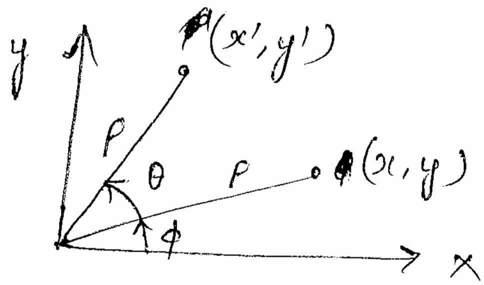
To specify a translation, we need only to specify a displacement vector d, because the transformed points are given by -

$\boxed{p' = p + d}$ for all points p on the object.

Translation has three degrees of freedom because we can specify 3 components of displacement vector arbitrarily.

Rotation

* **Rotation** operation accepts more than one parameters for its specification.
* fig show 2D rotation about origin



A 2D point at (x, y) is rotated about the origin by an angle θ to the position (x', y')

This can be represented as:-

$$x = \rho\cos\phi$$
$$y = \rho\sin\phi$$
$$x' = \rho\cos(\theta + \phi)$$
$$y' = \rho\,\cancel{\cos}\sin(\theta + \phi)$$

expanding, we get

$$x' = \rho\cos\phi\cos\theta - \rho\sin\phi\sin\theta$$
$$= x\cos\theta - y\sin\theta.$$
$$y' = \rho\cos\phi\sin\theta + \rho\sin\phi\cos\theta$$
$$= x\sin\theta + y\cos\theta.$$ where $x = \rho\cos\phi$
$y = \rho\sin\phi$.

In matrix form

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

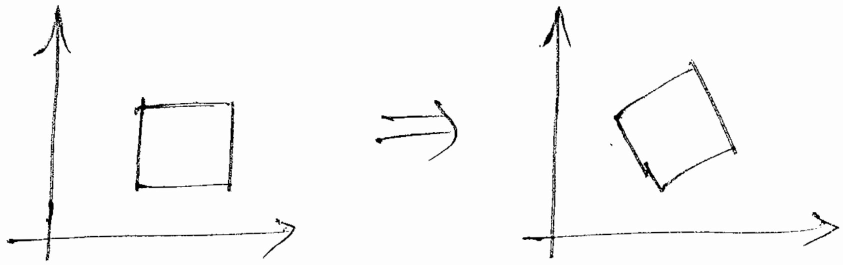


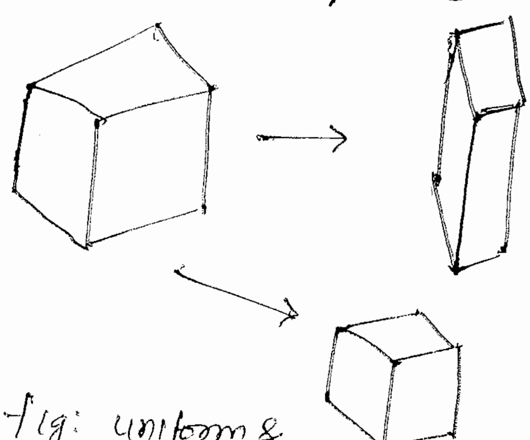
fig: Rotation about a fixed point.

<u>note:</u> Rotation & Translation are known as <u>rigid body transformations</u> since no combination of rotations & translations can alter the shape or volume of an object.

fig: non rigid body Transformation.

<u>Scaling</u> - <has 6 degrees of freedom>

* <u>Scaling</u> is an affine non rigid body transformation by which we can make an object bigger or smaller.
* Fig shows both uniform & non uniform scaling.



* Scaling have a fixed point
* To specify scaling, we should specify the fixed point, direction we wish to scale, & a scale factor (α)

If $\alpha > 1 \Rightarrow$ objects gets bigger

If $\alpha < 1 \Rightarrow$ object gets smaller

-ve value of α gives <u>reflections</u> (ref fig)

fig: uniform & non uniform scaling.

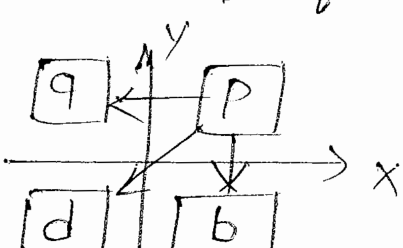
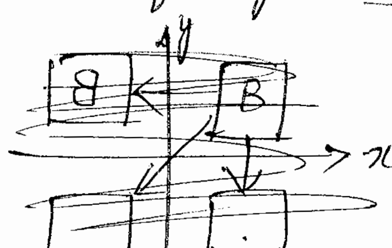


fig Reflection.



UNIT 5 :

GEOMETRIC OBJECTS AND TRANSFORMATIONS - 2

syllabus

* Transformations in homogeneous coordinates.
* Concatenation of Transformations.
* openGL Transformation matrices.
* Interfaces to three dimensional applications.
* Quaternions

- 5 Hours.

Ashok Kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

TRANSFORMATION IN HOMOGENEOUS COORDINATES

* within a frame, each affine Transformation is represented by a 4×4 matrix of the form

$$A = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{14} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{24} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* Here we discuss 4 types of affine Transformation -
  - Translation
  - Rotation
  - Scaling
  - shear.

Translation

* Translation displaces points to new positions defined by displacement vector.
* If we move the point p to p' by displacing by a distance d, then $\boxed{p' = p + d}$
* Homogeneous coordinate forms of p, p', & d are

$$p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad p' = \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} \qquad d = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \\ 0 \end{bmatrix}$$

* These equations can be written component by component as -

$$x' = x + \alpha_x$$

$$y' = y + \alpha_y$$

$$z' = z + \alpha_z$$

* However, we can also get this result using the matrix multiplication -

$$p' = Tp$$

where -

$$T = \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

T is called translation matrix we sometimes write it as $T(\alpha_x, \alpha_y, \alpha_z)$

* we can obtain the inverse of a translation matrix as follows -

$$T^{-1}(\alpha_x, \alpha_y, \alpha_z) = T(-\alpha_x, -\alpha_y, -\alpha_z) = \begin{bmatrix} 1 & 0 & 0 & -\alpha_x \\ 0 & 1 & 0 & -\alpha_y \\ 0 & 0 & 1 & -\alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

* For both scaling and rotation, there is a fixed point that is unchanged by the transformation. (we let the fixed point to be origin here).
* A scaling matrix with a fixed point of the origin allows for independent scaling (increase or decrease size of primitive) along the coordinate axes.
* The three equations are :-

$$x' = \beta_x x$$
$$y' = \beta_y y$$
$$z' = \beta_z z$$

* These three eqns can be combined in homogeneous form as -

$\boxed{p' = Sp}$ where $S = S(\beta_x, \beta_y, \beta_z) = \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \end{bmatrix}$

* We can obtain the inverse of scaling matrix by applying the reciprocals of the scale factors -

$$\boxed{S^{-1}(\beta x, \beta y, \beta z) = S\left(\frac{1}{\beta x}, \frac{1}{\beta y}, \frac{1}{\beta z}\right)}$$

Rotation.

* Here we take the fixed point as origin.
* Rotation enables the programmer to rotate a given point w.r.t the three degrees of freedom.
* Suppose we rotate a point $p(x,y,z)$ about the ~~w.r.t~~ z-axis to get the new point $p'(x',y',z')$. Then the equation for rotation about z axis by an angle θ is given by -

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

or, in matrix form -

$$\boxed{p' = R_z p}$$

where

$$R_z = R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

III'y, eqn for rotation about x-axis is -

$$R_x = R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

III'y, eqn for rotation about y axis is -

$$R_y = R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* A rotation by θ can always be <u>undone</u> by a subsequent rotation by $-\theta$. Hence -

$$R^{-1}(\theta) = R(-\theta)$$

note: $\cos(-\theta) = \cos\theta$

$\sin(-\theta) = -\sin\theta$.

Shear

* Consider a cube centered at origin, aligned with the axes and viewed from +ve z axis. (refer fig)

If we pull the top of the object (cube) to the right and bottom to the left, we say that we <u>shear</u> the object in the x-direction.

<u>note:</u> neither y nor z values are changed by the shear.

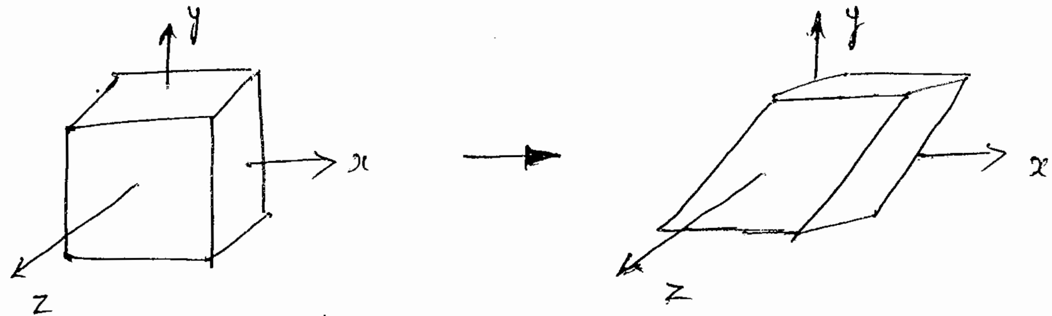


fig: shear.

* Using simple Trigonometry on below fig, we see that each shear is characterised by a single angle θ. The equations for this shear are

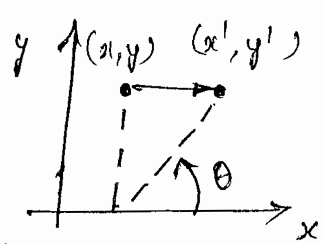


fig: composition of shear matrix.

$$x' = x + y\cot\theta$$

$$y' = y$$

$z' = z$, leading to shearing matrix

$$H_x(\theta) = \begin{bmatrix} 1 & \cot\theta & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* We can obtain inverse by shearing in opposite direction

CONCATENATION OF TRANSFORMATIONS

* It is nothing but multiplication of sequence of basic transformations in order to produce arbitrary transformation.

for ex: If we carry out three ~~consequetive~~ successive transformations on a point p, creating a new point q, then,

$$\boxed{q = CBAp}$$

$A, B, \& C$ can be arbitrary 4×4 matrices.

order is important. Here we carry out A, followed by B, and followed by C.

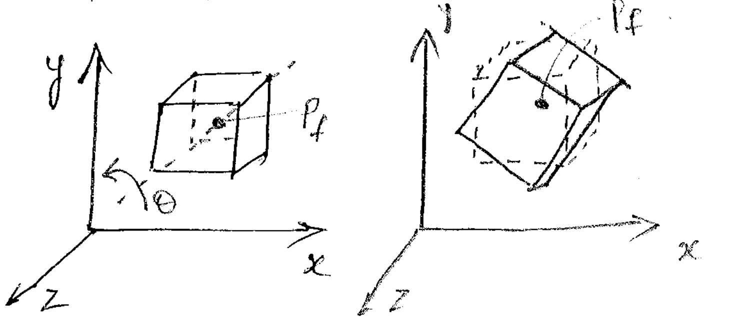
ie $\boxed{q = (C(B(Ap)))}$

$$p \longrightarrow \boxed{A} \longrightarrow \boxed{B} \longrightarrow \boxed{C} \longrightarrow q$$

* Here, we develop matrices for -
  - Rotation about a fixed point
  - General rotation.
  - Instance Transformation.
  - Rotation about an arbitrary axis

Rotation about a fixed point

* Consider a cube with its center at P_f and its sides aligned with axes.
* Assume that the cube is to be rotated about z axis (as shown) about its center P_f.



Rotation of cube about its center.

* It can be done as follows -

Step 1: Move the cube to the origin by applying basic transformation of translate $T(-P_f)$.

Step 2: Rotate w.r.t (or about) z axis by desired angle θ by applying $R_z(\theta)$.

Step 3: Move the object back such that its center is again at P_f by applying ~~reverse~~ translation as $T(+P_f)$

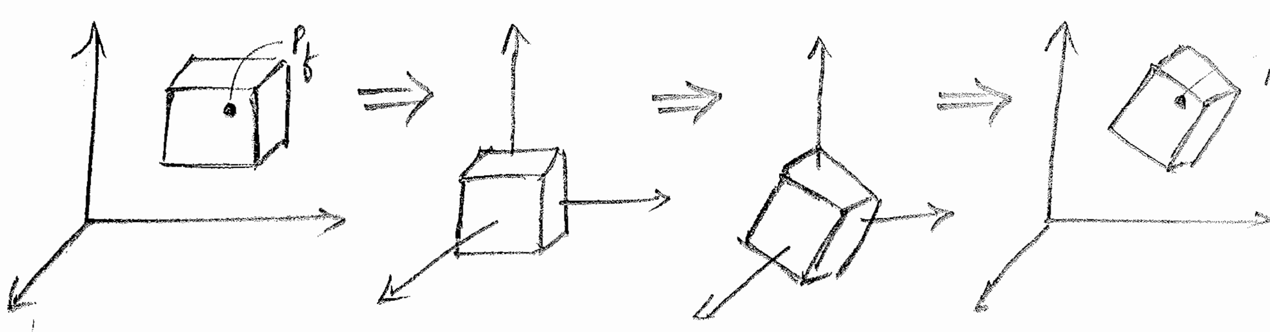
* Concatenating the above transformations, we get

$$\boxed{M = T(P_f)\, R_z(\theta)\, T(-P_f)}$$

If we multiply out the matrices, we find that

$$M = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & x_f - x_f\cos\theta + y_f\sin\theta \\ \sin\theta & \cos\theta & 0 & y_f - x_f\cos\theta - y_f\sin\theta \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* Fig below shows sequence of Transformations.



General Rotation

* We now show that arbitrary rotation about the origin can be composed of three successive rotations about the three axis.

* We can obtain the desired matrix by first doing a rotation about x-axis and then about y axis and then about z-axis.

* However, the order of multiplication does not matter. Therefore, the final rotation matrix M would be –

$$\boxed{M = R_x(\alpha) * R_y(\beta) * R_z(\gamma)}.$$

By selecting appropriate values of α, β, and γ, we can achieve any desired orientation.

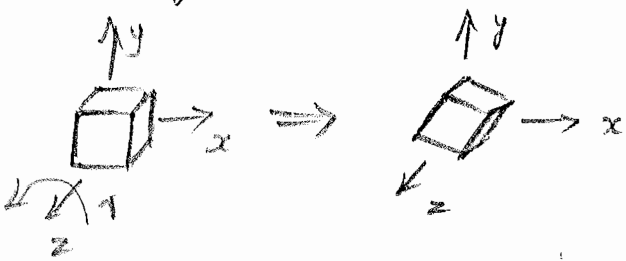


fig: rotation about z-axis.

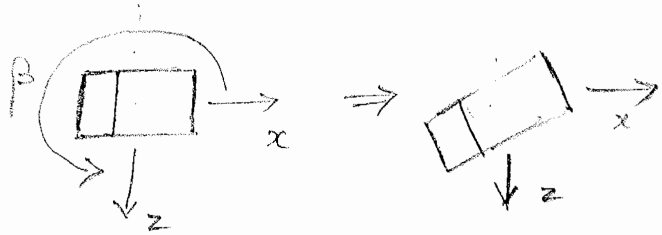


fig: Rotation about y-axis.

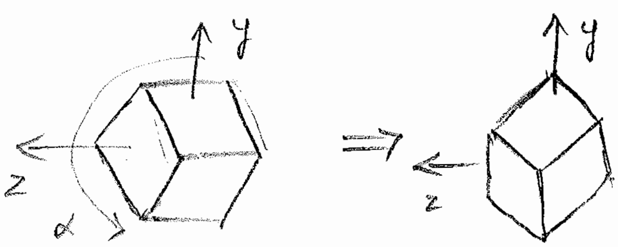


fig: Rotation about x-axis

Instance Transformation.

* Consider a scene composed of many simple objects as shown.

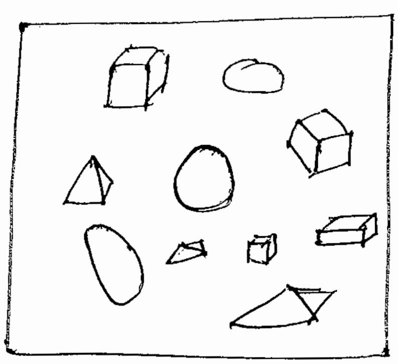


fig: scene of simple objects.

* There are two ways to create the above scene
1. Define each of these objects through its vertex, using glVertex()
2. An alternative is to define each of the object types once at a convenient size, place, & orientation.

   Each occurence of an object in the scene is an "instance" of that object's prototype, and we can obtain the desired size, orientation, and location by applying affine Transform called instance Transformation to the prototype.

* Instance transformation is applied in the order shown in the fig.
* In case of instance transformation, objects are originally defined in their own frames, First they are scaled to the desired size, Then they are oriented suitably using rotation matrix and then it is translated to the desired location.

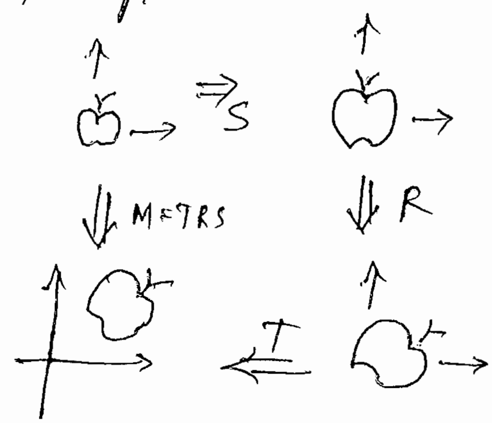


fig: Instance Transformatio

* Therefore, instance Transformation matrix is of the form -

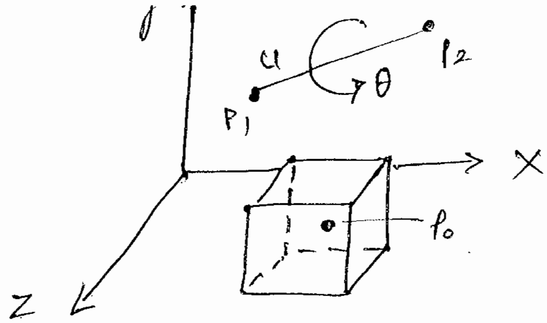
$$\boxed{M = T * R * S}$$

Rotation about an Arbitrary axis

* consider a point P_0 as the center of the cube.
* The vector about which the cube is to be rotated can be specified by providing the points P_1 and P_2. The vector defined by P_1 and P_2 is u.

$$\boxed{u = P_2 - P_1}$$

~~* To~~ γ



* To rotate the cube about the vector u by an angle θ, the steps to be performed are as follows -

Step 1: Translate the fixed point, P_0 to the origin by performing $T(-P_0)$

Step 2 : Rotate the cube about the x-axis by performing $R_x(+\theta_x)$

Step 3: Rotate the cube about the y-axis by performing $R_y(\theta_y)$

Step 4: Rotate the cube about the z-axis by angle θ (known) performing $R_z(\theta)$

~~Step 5: Perform inverse rotation about z axis ie $R_z(-\theta_z)$~~

Step 5: Perform inverse rotation about y axis ie $R_y(-\theta_y)$

Step 6: Perform inverse rotation about x axis ie $R_x(-\theta_x)$.

Step 7: perform inverse Translation of the fixed point P_0 using $T(P_0)$.

* Therefore, the concatenated matrix M is

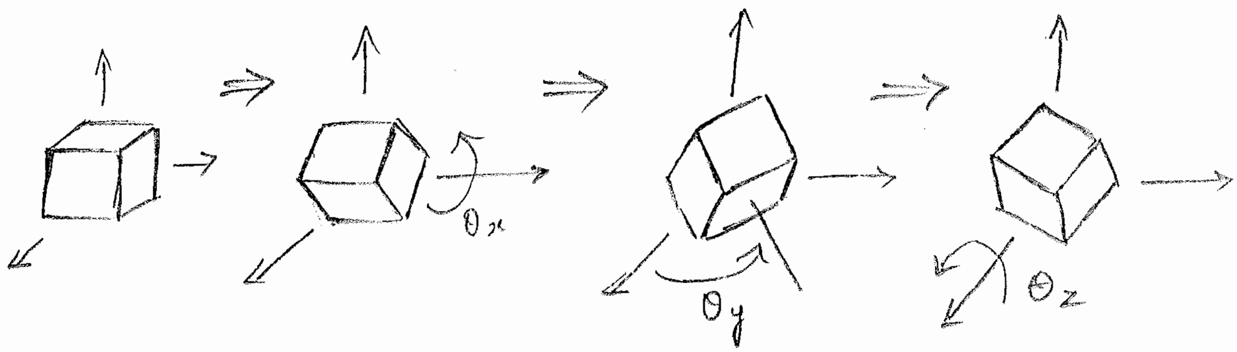
$$\boxed{M = T(P_0)\, R_x(-\theta_x)\, R_y(-\theta_y)\, R_z(\theta)\, R_y(\theta_y)\, R_x(\theta_x)\, T(-P_0)}$$

* Our first and last transformation is the translation. ie $T(-P_0)$ and $T(P_0)$.

In between we perform rotation,

$$R = R_x(-\theta_x)\, R_y(-\theta_y)\, R_z(\theta)\, R_y(\theta_y)\, R_x(\theta_x).$$

This sequence of rotations is shown below.



Determining θ_x and θ_y.

+ We replace 'u' with a unit length vector

$$V = \frac{u}{|u|} = \begin{bmatrix} \alpha_x \\ \alpha_y \\ \alpha_z \end{bmatrix}$$

we have, $\alpha_x^2 + \alpha_y^2 + \alpha_z^2 = 1$, since V is a unit length vecto

* we draw a line segment from origin to $(\alpha_x, \alpha_y, \alpha_z)$. This line seg. has unit length & the orientation of V.
Next we draw perpendiculars from the point $(\alpha_x, \alpha_y, \alpha_z)$ to the co ordinate axes as shown.
The three direction angles - ϕ_x, ϕ_y, ϕ_z are the angles b/w the line seg (or v) and the axes.
The direction cosines are given by

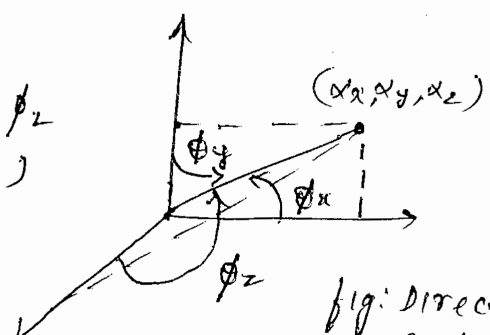


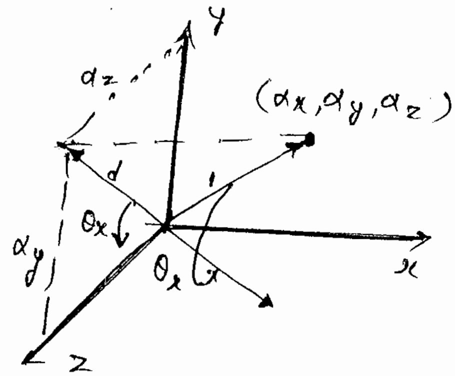
fig: Direction angles.

$\cos\phi_x = \alpha_x$
$\cos\phi_y = \alpha_y$
$\cos\phi_z = \alpha_z$

only two of the direction angles are independent because,

* we now calc θ_x & θ_y using these angles.

Computation of x rotation.

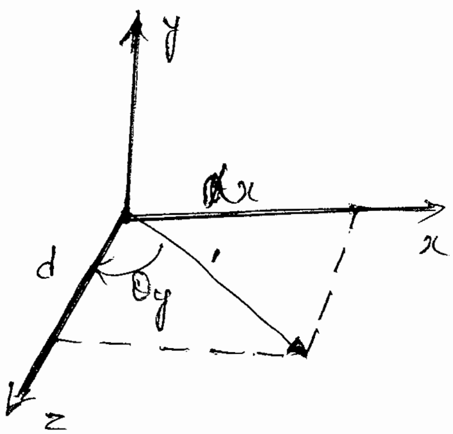


* Consider the fig shown.
- If we ~~see~~ ~~a line~~ look at the projection of the line seg (before rotation) on the plane $x=0$, we will see a line seg of length d on this plane.
The line that we see on the wall is the shadow of line seg from origin to $(\alpha_x, \alpha_y, \alpha_z)$
Note that length of shadow is less than length of line seg.
We can say that the line seg has been foreshortened to $d = \sqrt{\alpha_y^2 + \alpha_z^2}$. We ~~get~~ never need to compute θ_x. rather we need to compute only -

$$R_x(\theta_x) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \alpha_z/d & -\alpha_y/d & 0 \\ 0 & \alpha_y/d & \alpha_z/d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Computation of y rotation.

* It is similar to the above.



$$R_y(\theta_y) = \begin{bmatrix} d & 0 & -\alpha_x & 0 \\ 0 & 1 & 0 & 0 \\ \alpha_x & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

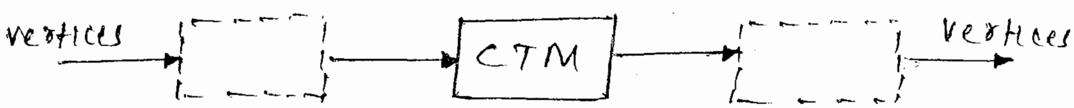
OPENGL TRANSFORMATION MATRICES

* Here we see the implementation of an homogeneous - coordinate transformation package and of that package's interface to the user.

* We use glMatrixMode to select the matrix to which the operations apply.
  In openGL, the model-view matrix normally is an affine - Transformation matrix and has only 12 degrees of freedom.

Current Transformation matrix. (CTM)

* It is the matrix that is applied to any vertex that is defined subsequent to its setting.
  CTM is part of the pipeline. Thus, if p is a vertex specified in the application, then the pipeline produces Cp.



* CTM is an 4×4 matrix, initially set to identity matrix. It can be reinitialized as needed using '$\leftarrow$'

  eg $C \leftarrow I$

* We denote CTM by C. CTM can be altered by a set of functions provided by graphics package.

* The functions that alter C are of two forms
  1. Those that load it with some matrix
  2. Those that modify it by premultiplication or postmultiplication by a matrix.

  openGL uses only post multiplication.

* The three transformations supported in most systems are
  - Translation
  - scaling with fixed point of the origin

* symbolically, we can write these operations in postmultiplication form as -

$C \leftarrow CT$

$C \leftarrow CS$

$C \leftarrow CR$

and in load form as -

$C \leftarrow T$

$C \leftarrow S$

$C \leftarrow R$

* most systems allow us to load the CTM with an arbitrary matrix M -

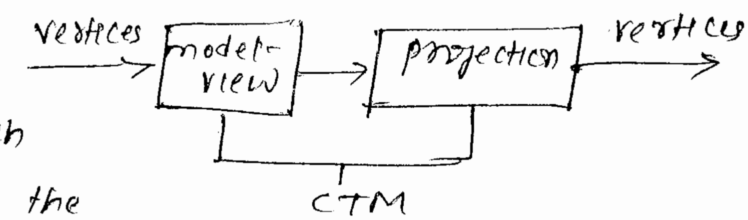
$C \leftarrow M$

or to postmultiply by an arbitrary matrix M -

$C \leftarrow CM$

Rotation, Translation, and scaling

* In openGL, matrix that is applied to all primitives is the product of model-view ~~p~~ matrix and projection matrix. ie CTM is a product of these two matrices. we can manipulate each individually by selecting the desired matrix by glMatrixMode.



→ We can load a matrix with the function

glLoadMatrixf(pointer_to_matrix);

→ or set a matrix to the identity matrix as -

glLoadIdentity();

→ We can alter the selected matrix with

glMultMatrix(pointer_to_matrix);

→ Rotation, Translation, and scaling are provided thro' these functions

glRotatef (angle, vx, vy, vz);

glTranslatef (dx, dy, dy);

glScalef (sx, sy, sz);

Rotation about a fixed point in openGL.

* For a 45° rotation about the line through the origin and the point (1, 2, 3) with a fixed of (4, 5, 6), we have

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (4.0, 5.0, 6.0);
glRotatef (45.0, 1.0, 2.0, 3.0);
glTranslatef (-4.0, -5.0, -6.0);
```

Order of Transformations

* ~~The sequence of~~ Rule in openGL is this:
  "Transformation specified Last is the one applied first"
* The sequence of operations we specified above was

$$C \leftarrow I$$
$$C \leftarrow CT(4.0, 5.0, 6.0)$$
$$C \leftarrow CR(45.0, 1.0, 2.0, 3.0)$$
$$C \leftarrow CT(-4.0, -5.0, -6.0)$$

or

$$C = T(4.0, 5.0, 6.0)\,R(45.0, 1.0, 2.0, 3.0)\,T(-4.0, -5.0, -6.0)$$

Spinning of the cube. // not needed, can be skipped.

* we define following three call back functions -

```
glutDisplayFunc (Display);
glutIdleFunc (spincube);
glutMouseFunc (mouse);
```

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

void mouse(int bt, int st, int x, int y)
{
  if(bt==GLUT_LEFT_BUTTON && st==GLUT_DOWN) axis = 0;
  —"————— _MIDDLE_    ———   "————————————— axis = 1;
  —"——————RIGHT    ———"————————————————— axis = 2;
}

void spincube()
{ theta[axis] += 2.0;
   if (theta[axis] >360.0)   theta[axis] -=360;
   glutPostRedisplay();
}

void mykey( char key, int mousex, int mousey)
{
  if(key=='q' || key=='Q')  exit();
}
```

Loading, pushing, & popping Matrices.

Sometimes if the programmer wants to perform a certain transformation and then return to the same state as before, then he can push the transformation matrix on to stack with glPushMatrix() before multiplication & recover it later with glPopMatrix().

eg

```
glPushMatrix ();
glTranslatef (...);
glRotatef (...);
glScalef (...);
    /* draw objects here */
glPopMatrix ();
```

note:

1. we can load a 4×4 homogeneous coordinate matrix as the current matrix as -

```
glLoadMatrixf (myarray);
```

2. we can also multiply on the right of current matrix by a user defined matrix as -

```
glMultMatrixf (myarray);
```

3. eg for forming myarray -

```
GLfloat m[4][4];
GLfloat myarray[16];
for (i=0; i<3; i++)
    for (j=0; j<3; j++)
        myarray[4*j+i] = m[i][j];
```

INTERFACES TO THREE - DIMENSIONAL APPLICATIONS

* Glut provides for smoother and more interesting interfaces to 3D applications by allowing the user to use keyboard along with mouse to provide interaction.
* Suppose that the user wants to use one mouse button for orienting an object, one for getting closer to or farther from the object, and one for translating the object to left or right.
* We can use the motion call back to achieve all these functions. The call back returns which button has been activated and where the mouse is located. This location of the mouse can be used to control the direction of rotation, translation, and to move in or move out.

A virtual Track Ball

* A virtual Track ball can be created using a mouse and the display device (eg: monitor)
* Adv of virtual device - It creates a frictionless track ball which once started to rotate will continue to rotate until stopped by the user. Thus it supports continuous rotations of objects but will still allow changes in the speed and orientation of the rotation.
* It can be achieved by mapping the position of the ~~mouse~~ trackball to that of a mouse.
* Consider the track ball as shown.
* We assume that the trackball has a radius of 1 unit. We can map a position on its surface to the plane y=0 by doing an orthogonal projection as shown below -

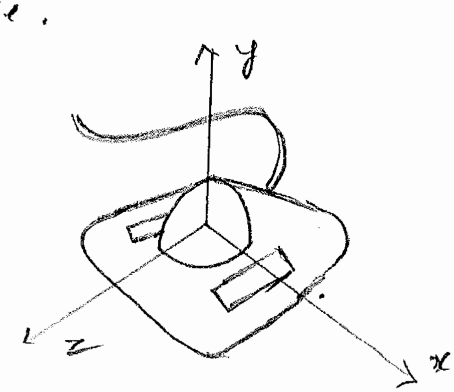
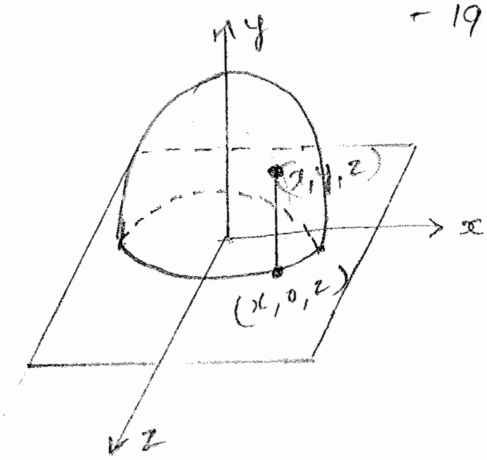
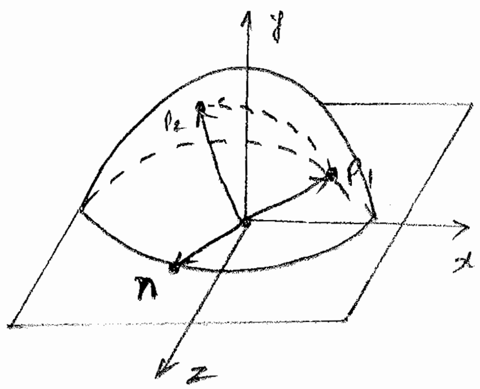


fig: Track ball frame



* The position (x, y, z) on the surface of the ball is mapped to (x, 0, z) on the plane.
* The motion of the track ball that moves from one point p_1 to another point p_2 can be computed by computing the angle θ wrt p_1 and p_2



$$n = p_1 * p_2 .$$

$$\boxed{|\sin\theta| = |n|}$$

If we a tracking the mouse at high rate, then changes in position that we detect will be small. Hence rather than using inverse trigometric func. to find θ, we use the approximation

$$\sin\theta \approx \theta .$$

Incremental Rotation.

* GLUT also provides for smooth incremental rotations.
* Suppose that we are given two orientations of the camera and we want to move smoothly from one orientation to another, then the corresponding code will be -

```
for(i=0; i<imax; i++)
{
    glRotatef(delta_theta, dx, dy, dz);
    draw_object();
}
```

problem of this code - calculation of rotation matrix requires evaluation of sines and cosines of three angles.

* ~~So, we c~~

* So, to overcome this, we compute the rotation matrix once and reuse it through code such as the following -

```
GLfloat m[16];
glRotatef (dx, dy, dz, delta_theta);
glGetFloatv (GL_MODELVIEW_MATRIX, m);
for (i=0; i<max; i++)
  {
     glMultMatrixf (m);
     draw_object ();
  }
```

we could also use <u>small angle approximations</u>

$$\left.\begin{array}{l}\sin\theta \simeq \theta \\ \cos\theta \simeq 1\end{array}\right\} \text{ if } \theta \text{ is very small.}$$

- An arbitrary axis rotation matrix with an angle ψ along z axis, ϕ along y axis, and θ along x axis can be achieved using -

$$R = R_z(\psi)\, R_y(\phi)\, R_x(\theta).$$

$$= \begin{bmatrix} \cos\psi & -\sin\psi & 0 & 0 \\ \sin\psi & \cos\psi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

for small angles of ψ, ϕ, and θ, we get

$$R = \begin{bmatrix} 1 & -\psi & \phi & 0 \\ \psi & 1 & -\theta & 0 \\ -\phi & \theta & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

QUATERNIONS

* **Quaternions** are an extension of complex numbers that provide an alternative method for describing and manipulating rotations.

* There are two methods of performing rotations
  - post multiplication of CTM with rotation matrix
  - performing rotations using quaternions.

Advantage of using second method is that it requires very little computations for rotations when compared to rotation matrices.

* Specifying rotations in 3D space involves two things
  - specifying the direction of rotation which is a vector quantity.
  - specifying the amount of rotation which is a scalar quantity.

* Since a quaternion has both scalar and vector representation it can be used to specify rotation efficiently.

* A point p in space can be represented in the quaternion representation as

$$p = (0, P)$$

consider a quaternion in the polar form -

$$r = \left(\cos\frac{\theta}{2},\ \sin\frac{\theta}{2} * v\right)$$ 'v' has unit length.

inverse quaternion is

$$r^{-1} = \left(\cos\frac{\theta}{2},\ -\sin\frac{\theta}{2} * v\right)$$

* A new point p' after rotation can be obtained using the quaternion product

$$p' = r p r^{-1}$$

* This resultant quaternion has the form $(0, p')$ where

$$p' = \cos^2\frac{\theta}{2}\, p + \sin^2\frac{\theta}{2}(p \cdot v)v + 2\sin\frac{\theta}{2}\cdot\cos\frac{\theta}{2}(v * p)$$
$$- \sin\frac{\theta}{2}(v \times p) \times v$$

* Above result represented by p' is the effect of rotation of point p by θ degrees about the vector v.

* If we ~~cannot~~ count the number of operations required to perform this task using matrix multiplication w.r.t the number of operations required using quaternions, it can be realised that using quaternions results in faster computations.

Ashok Kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

UNIT 6 VIEWING

Syllabus.

* classical and computer viewing
* viewing with a computer.
* position of the camera
* simple projections.
* projections in openGL
* Hidden surface Removal
* Interactive mesh Displays
* parallel projection matrices.
* perspective projection matrices.
* projections and shadows.

– 7 Hours.

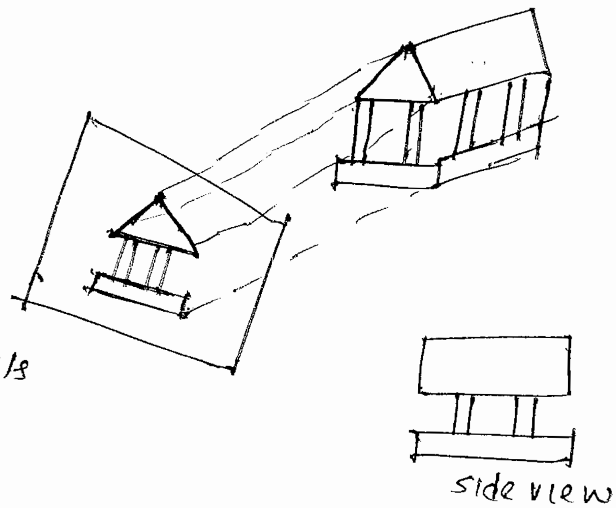
Ashok Kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

CLASSICAL AND COMPUTER VIEWING

* There are four types of classical views (projections)
  1. orthographic projections
  2. Axonometric projections.
  3. oblique projections
  4. perspective projections.

orthographic projection

* In all orthographic views, the projectors are perpendicular to the projection plane.
* Fig shows orthographic projection.

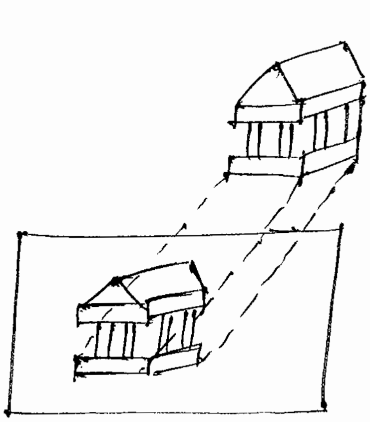


* usually three views are used in an orthographic projection to display the objects
  - front view
  - Top view
  - side view
* Adv : It preserves both distances and angles. Since there is no distortion, it is well suited for working drawings.

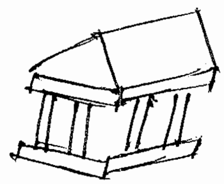
Axonometric projections.

* It is such a projection in which the projector is still orthogonal to the projection plane, but the projection plane can have any orientation wrt the object. using this view, more than one principle faces of the object would be visible.

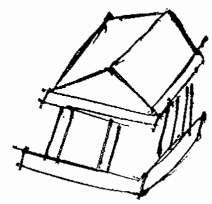
* If the projection plane is placed symmetrically w.r.t the three principal faces that meet at a corner of our rectangular object, then we have an isometric view
* If the projection plane is placed symmetrically wrt two of the principal faces, then the view is dimetric.
* The general case is a trimetric view.



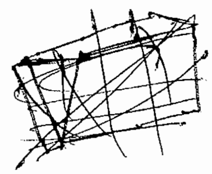
~~orthog~~ Axonometric projection.

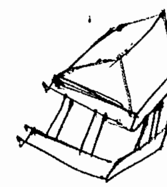


Dimetric



Trimetric.

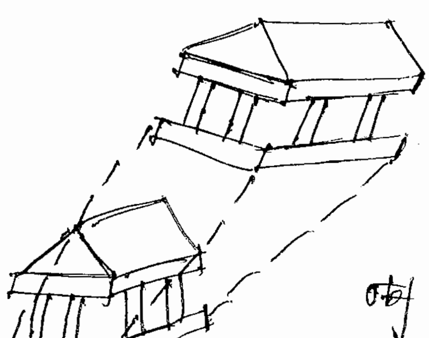




Isometric.

Oblique projections

* It is one of the most general parallel projections.
* This projection is obtained by allowing the projectors to make an arbitrary angle with the projection plane.
* Angles of the objects face that are parallel to the projection plane are preserved.
* oblique views (projection) are most difficult to construct by hand. They are somewhat unnatural.



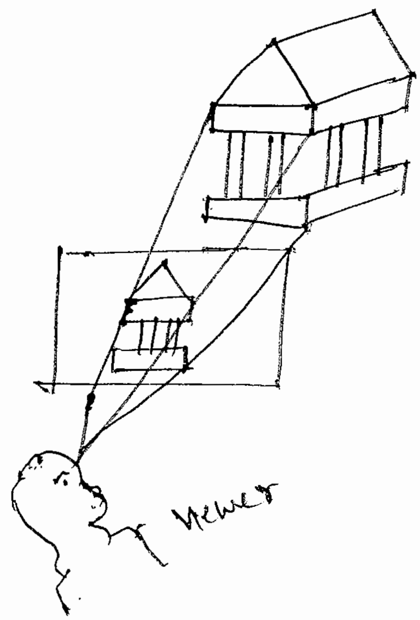
~~obj~~ oblique view.

perspective viewing.

* It is such a projection in which the viewer is located symmetrically wrt the projection plane.
All perspective projectors are characterized by diminution of size.

* when objects are moved farther from the viewer, their images become smaller.
This size change gives perspective projection its natural appearance. Hence it is widely used in architecture and animation

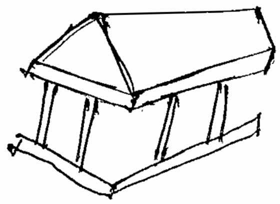
* figure shows perspective projections.



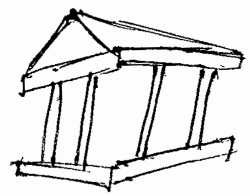
* there are three types of perspective views
- one point perspective
- two point perspective
- three point perspective.

based on how many ~~no.~~ of the three principal directions in the object are parallel to the projection plane.

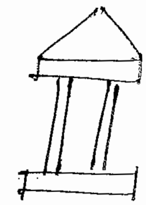
* In most general case, 3-point perspective, the parallel lines in each of the three principal directions converge at one point called - vanishing points. < fig (a) >
* in 2-point perspective, lines in only two of the principal directions converge at vanishing points. < fig(b) >
* In 1-point perspective, two of the principal directions converge at a single vanishing point.



3 point
fig (a)



2 point
fig (b)



1 point
fig (c).

note:

planar projections
- ~~plan~~ parallel
  - oblique
  - orthographic
    - front view
    - side view
    - Top view
    - Axonometric
- perspective
  - 1 point
  - 2 point
  - 3 point

VIEWING WITH A COMPUTER.

* Viewing using computers involve two fundamental operations
1. positioning and orienting the camera which can be achieved by using the model view matrix.
2. Application of the projection transformation. ie parallel projection or perspective projection which can be achieved using projection matrix.

> unit 4

> unit 5

POSITIONING OF THE CAMERA

* camera can be positioned using openGL by modifying the model-view matrix.
* Initially the model view matrix is an identity matrix and hence the camera frame and the object frame would be identical as shown below. fig(a).

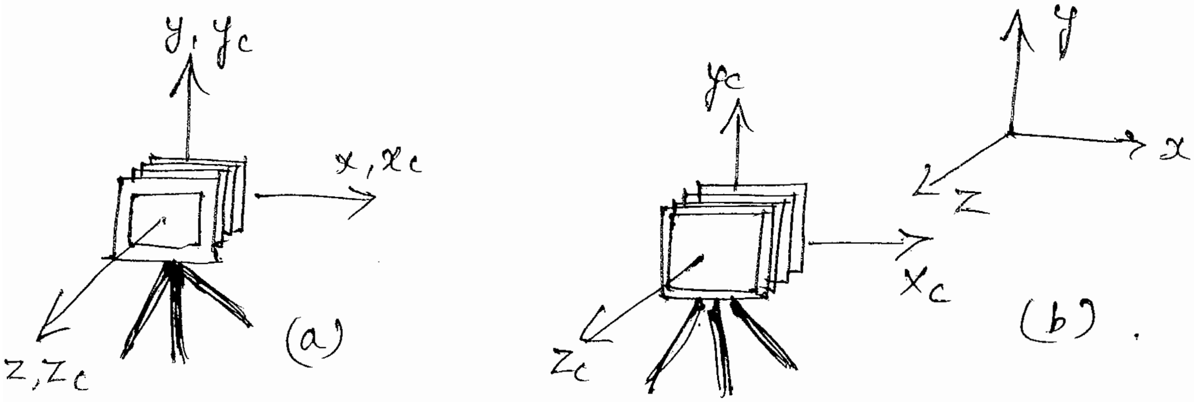


fig: movement of camera and object frames.
(a) initial configuration.
(b) Configuration after change in model-view-matrix

* with fig (a) arrangement, all the objects present in the scene may not be visible and hence we will have to change the model-view matrix suitably to position the camera at the desired location. (refer fig b)

eg: Suppose that the user is interested in viewing at the object from the -ve z axis. Then the camera can be positioned using,

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslate (0.0, 0.0, -d);
```

eg2: Suppose that user is interested in looking at the same object from +ve x axis. Then the camera can be positioned using,

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (0.0, 0.0, -d);
glRotate (-90.0, 0.0, 1.0, 0.0);
```

eg3: To obtain an isometric view of the cube which is centered at the origin and aligned with the axes, we must place the camera anywhere along the line from the origin through the point (1,1,1).
This can be achieved using,

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
glTranslatef (0.0, 0.0, -d);
glRotate (45.0, 0.0, 1.0, 0.0);
glRotate (35.26, 1.0, 0.0, 0.0);
```

* There is an altogether different approach that can be used to position the camera.
This approach is used in PHIGS and GKS-3D (which were one of the earliest graphics packages).
The steps followed are:

1. The camera is assumed to be initially positioned at origin, pointing in the -ve Z direction. Its desired location is referred to as the view reference point (VRP) which can be specified as follows

set_view_reference_point (x, y, z);

2. orientation of the camera can be specified using the view-plane-normal (VPN) using,

```
set_view_plane_normal (nx, ny, nz);
```

3. The updirection from the perspective of the camera can be specified using the view-up (VUP) as shown below,

```
set_view_up ( vup_x, vup_y, vup_z);
```

* Another approach is to use the LookAt function as shown below,

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();
gluLookAt (eyex, eyey, eyez, atx, aty, atz,
           upx, upy, upz);
/* define objects here */
```

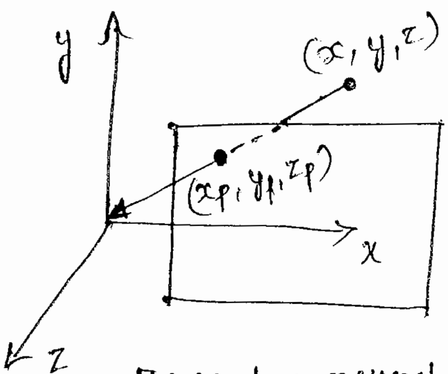
* Another approach is to specify the azimuth and the twist angle to position the camera.

SIMPLE PROJECTIONS.

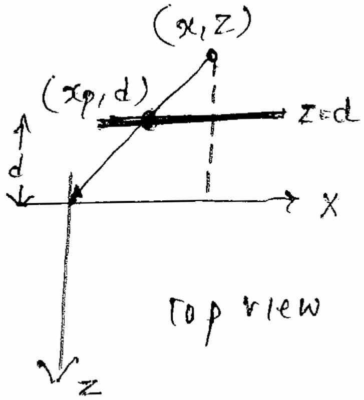
* There are two types of simple projections
  - perspective projections
  - parallel (orthographic) projections.

perspective projection.

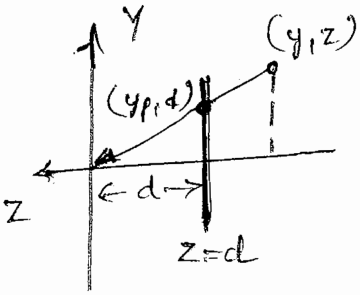
* Suppose that the camera is located at the origin pointing in the negative z-direction.
Assume that the projection plane is in front of the camera.
With the above arrangement, a point in space at the point (x, y, z) is projected along a projector into the point (x_p, y_p, z_p). All projectors pass through the origin as shown.



Three-dimensional view.



Top view



side view.

* From the above, it can be noticed that

$\cancel{d = z_p}$ $\boxed{z_p = d}$

from top view, we see that two similar triangles whose tangents must be same

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

from side view,

$$\frac{y}{z} = \frac{y_p}{d} \Rightarrow \boxed{y_p = \frac{y}{z/d}}$$

These eqns are non linear. The division by z describes non uniform foreshortening.

* The above equations can be obtained in the matrix form as shown below.

Consider the point in space as

$$p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Consider the matrix M as

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

* The matrix M transforms the point p to the point.

$q = M * p$.

$$q = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

By dividing first three terms with the fourth term, we get.

$$q' = \begin{bmatrix} \frac{x}{z/d} \\ \frac{y}{z/d} \\ \frac{z}{z/d} \\ 1 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$

Therefore, $\boxed{x_p = \frac{x}{z/d}}$

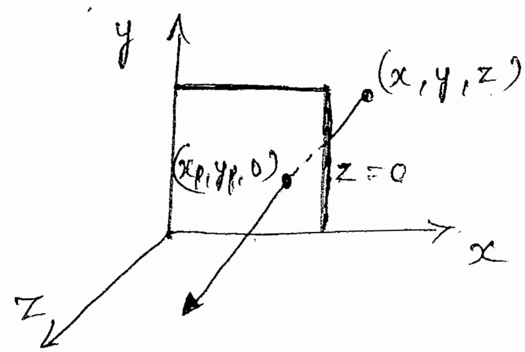
$\boxed{y_p = \frac{y}{z/d}}$

and, $\boxed{z_p = \frac{z}{z/d} = d}$.

Orthogonal Projections (parallel projections)

* Orthogonal projections are such projections in which the cameras have <u>infinite focal length</u>.

  It is as shown below.



* projectors are <u>perpendicular</u> to the view plane
* Above diagram shows a projection plane with $z=0$. As points are projected on to this plane, it can be noticed that they retain ~~re~~ their x and y values i.e diminution does not takes place.

  Therefore, in orthogonal projection,

  $\boxed{x_p = x}$ $\boxed{y_p = y}$ $\boxed{z_p = 0}$

* Above equations can be obtained in matrix mode as shown below.

  Consider the point in space as

$$p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

consider the matrix M such that

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

Then we can obtain orthogonal projection of the point p by multiplying M and p as shown below.

$$q = M * p$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$

Therefore,

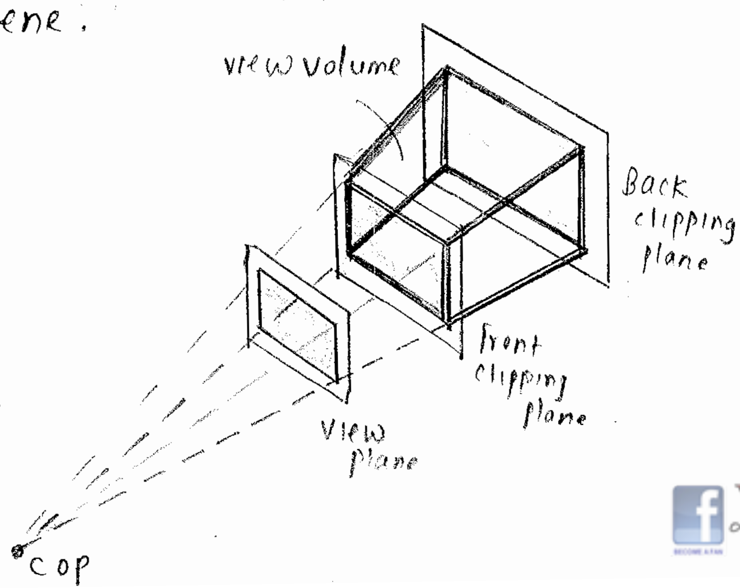
$$\boxed{x_p = x} \quad \boxed{y_p = y} \quad \boxed{z_p = 0}$$

PROJECTIONS IN OPENGL

lets see how view volume are specified in opengl.

* view volume is also referred to as ~~frustrum~~. frustum. objects falling within the view volume are displayed where as the objects falling outside the view volume are clipped out of the scene.

* The view volume is a truncated pyramid with its apex at center of project (COP) as shown in the figure.



* openGL provides two functions for specifying perspective view volume and one function for specifying parallel view volume.

perspective viewing in openGL

* perspective view volume can be specified using -

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glFrustum (left, right, bottom, top, near, far);
```

* This specification creates the following view volume.

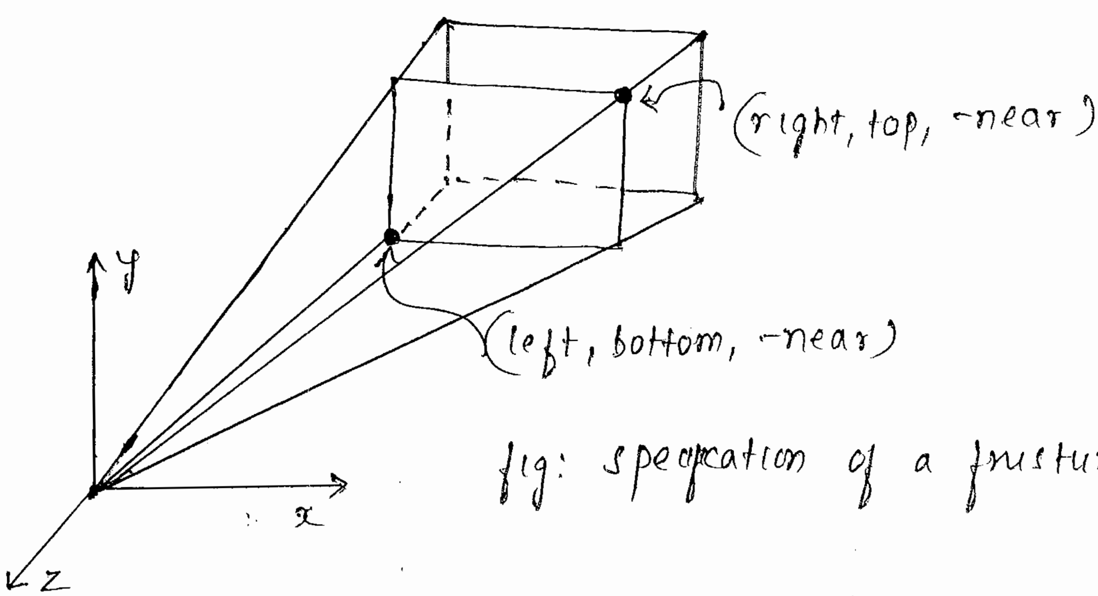


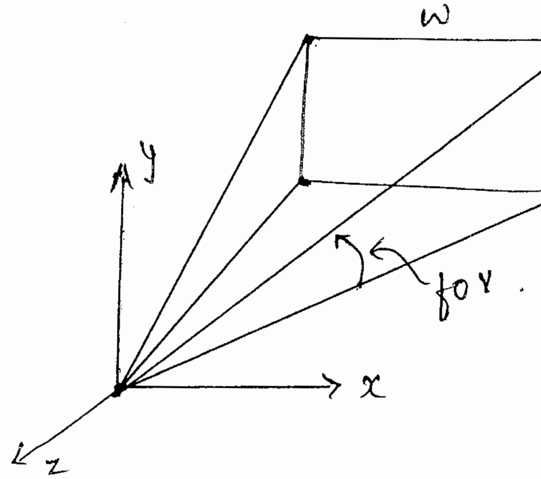
fig: specification of a frustum.

* The perspective view volume can also be specified by prescribing the angle fov, aspect ratio (ratio of width + height near & far parameters as shown below -

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluPerspective (fovy, aspect, near, far);
```

fov - field of view.

* This specification creates the following view volume.

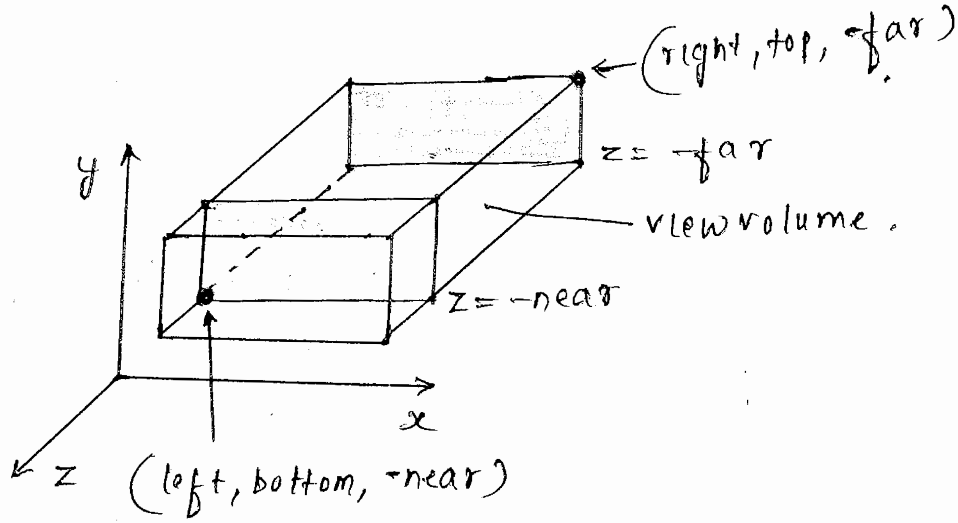


<u>parallel (orthographic) viewing in openGL.</u>

* orthographic viewing volume can be specified in openGL using,

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glOrtho (left, right, Bottom, top, near, far);
```

* This specification creates the following view volume.



PARALLEL PROJECTION MATRICES

lets derive the matrix for orthogonal and oblique projections in openGL.

* Basically there are two types of parallel projections namely -
  - orthogonal projection
  - oblique projection.

Orthogonal projection matrices

* orthogonal projection matrix can be obtained by performing the following steps -

Step 1: creating a view volume equal to the canonical view volume which is a cube defined by the sides $x = \pm 1$, $y = \pm 1$, and $z = \pm 1$

This step can be performed as shown below -

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
glOrtho (-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
```

Step 2:

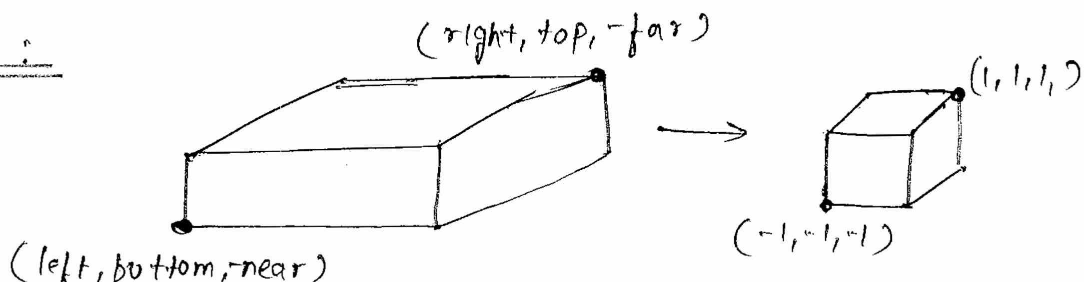


fig: mapping of view volume to the Canonical view volume.

Mapping of the original view volume to the canonical view volume

This step can be performed by first translating the center of the original view volume to the center of the canonical view volume and then scaling the original view volume to the canonical view volume.

Hence the two transformations to be performed are:

Translation $\left(-(right + left)/2,\ -(top + bottom)/2,\ +(far + near)/2\right)$

scaling $\left(2/(right - left),\ 2/(top - bottom),\ 2/(far - near)\right)$

They are concatenated together to form the projection matrix as shown below.

$$P = ST = \begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{left + right}{right - left} \\ 0 & \frac{2}{bottom - top} & 0 & -\frac{Top + Bottom}{Top - Bottom} \\ 0 & 0 & -\frac{2}{far - near} & -\frac{Far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

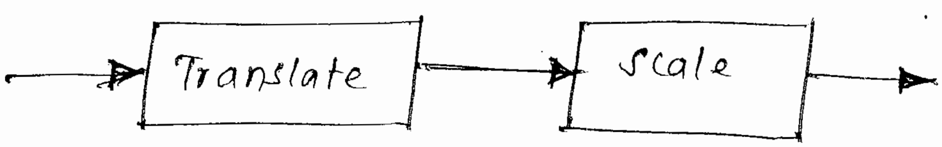


fig: Affine Transformations for normalization.

Oblique Projection Matrix

* Oblique projection Matrix can be obtained by performing the following steps.

step 1: shear of objects by $H(\theta, \phi)$

step 2: Create a view volume equal to canonical view volume

step 3: Mapping the original view volume to canonical view volume by performing Translation and scaling

[step2 and step3 are same as in previous case]

* consider the following oblique clipping volume

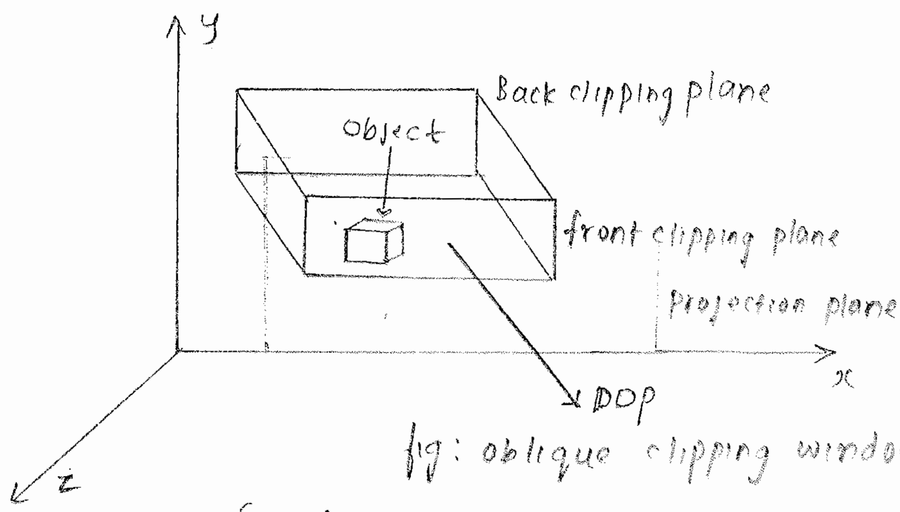


fig: oblique clipping window

fig : oblique projection

(a) Top view

(b) side view

* from the above fig, it can be noticed that -

$$x_p = x + z\cot\theta \longrightarrow \left(\because \text{ from top view, } \tan\theta = \frac{z}{x_p - x}\right).$$

$$y_p = y + z\cot\phi$$

$$z_p = 0$$

* we can write these in terms of a homogeneous coordinate matrix

$$P = \begin{bmatrix} 1 & 0 & \cot\theta & 0 \\ 0 & 1 & \cot\phi & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* This matrix can be expressed as the concatenation (product) of orthographic projection matrix (M_{ortho}) and shear matrix $H(\theta, \phi)$ as shown

$$P = M_{ortho} * H(\theta,\phi) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & \cot\theta & 0 \\ 0 & 1 & \cot\phi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

* However, since scaling and Transformation also has to be performed to make the original view volume equal to canonical view volume, the projection matrix for oblique projection would be -

$$\boxed{P = M_{ortho} * S * T * H}$$

where $ST = \begin{bmatrix} & \\ & \end{bmatrix}$ ← refer prev. case.

PERSPECTIVE PROJECTION MATRICES

lets Derive the matrix for perspective projection in openGL.

+ <u>perspective projection</u> matrix can be obtained by performing following steps -

<u>step1:</u> Distortion (Normalization) of the object

<u>step2:</u> perform orthographic projection.

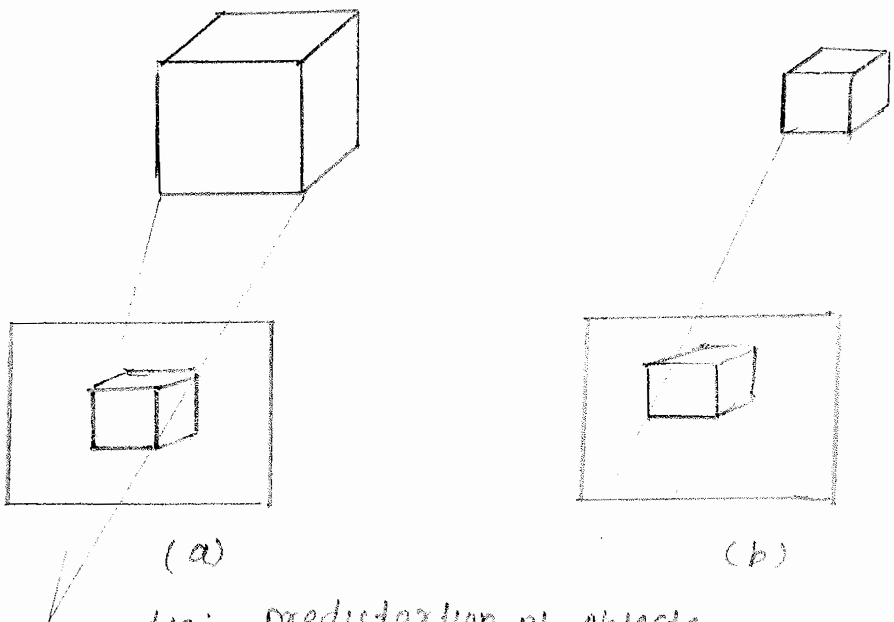


fig: predistortion of objects
(a) perspective view
(b) ~~othr~~ Orthographic projection of distorted object

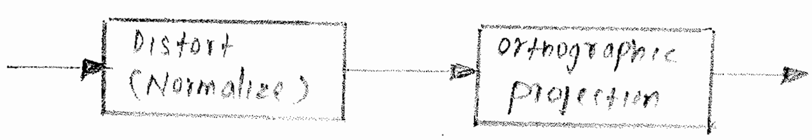


fig: Normalization transformation

* A simple projection matrix for the projection plane at $z = -1$ and the COP at the origin is -

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \rightarrow$$ simple perspective projection matrix.

* consider the matrix -

$$N = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \alpha & \beta \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

which is similar to M but is nonsingular.

* consider the point -

$$p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

* By applying N on p we get

$$q = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$

where, $x' = x$
$y' = y$
$z' = \alpha z + \beta$
$w' = -z$

* After dividing by w', we have the 3-D point -

$$\begin{bmatrix} x'' \\ y'' \\ z'' \\ 1 \end{bmatrix} = \begin{bmatrix} -x/z \\ -y/z \\ -(\alpha + \beta/z) \\ 1 \end{bmatrix}$$

Therefore, we get

$$\boxed{x_p = \frac{-x}{z}} \quad \boxed{y_p = -\frac{y}{z}}$$

* The same result can be obtained by applying an orthographic projection along the z-axis to N.
we get

$$M_{ortho} * N = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

when this orthographic projection is applied on the point $p[x, y, z, 1]$, we get

$$p' = M_{ortho} * N * p = \begin{bmatrix} x \\ y \\ 0 \\ -z \end{bmatrix}$$

Therefore,
we get $\boxed{x_p = \frac{-x}{z}}$ $\boxed{y_p = \frac{-y}{z}}$

Therefore, by applying N directly on the point yields the same result as applying the orthographic projection along z axis on N and then projecting the point

* original view volume can be normalized to perspective canonical view volume by choosing

$$x = \pm \frac{right - left}{-2 * near}$$

$$y = \pm \frac{top - bottom}{-2 * near}$$

$z = -near$

$z = far.$

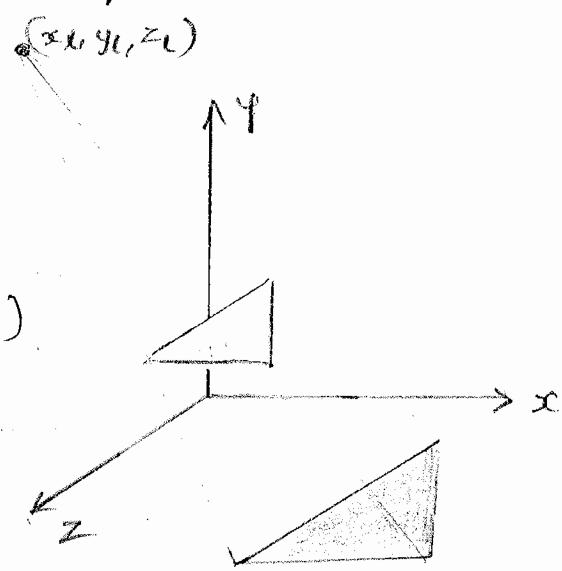
* Therefore, the resulting perspective projection matrix is $\Rightarrow P = N * S * H =$

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

PROJETION AND SHADOWS

lets see how shadows can be created and projected using openGL.

* simple shadows can be created in openGL. Although shadows are not geometric objects, yet they are important components of realistic images and give many visual clues.
* shadows require a light source to be present
  For simplicity, we assume that the shadow falls on ground ie $y=0$.
* A <u>shadow</u> is a flat polygon and is the projection of the original polygon onto the surface (ground). It is as shown below -



* The light source present at (x_L, y_L, z_L) must be bought the origin by performing a Translation $T(-x_L, -y_L, -z_L)$

  Then we have to perform a simple perspective projection through the origin.
  The projection matrix is

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1/y_l & 0 & 0 \end{bmatrix}$$

Then we must translate the light source back to (x_l, y_l, z_l) by $T(x_l, y_l, z_l)$

~~Therefore we have~~ the concatenation of this matrix and the two translation matrices projects the vertex (x, y, z) to

$$\boxed{x_p = x_l - \frac{x - x_l}{(y - y_l)/y_l}} \quad \boxed{y_p = 0} \quad \boxed{z_p = z_l - \frac{z - z_l}{(y - y_l)/y_l}}$$

note: How we got this ?

$$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -x_L \\ 0 & 1 & 0 & -y_L \\ 0 & 0 & 1 & -z_L \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -1/y_L & 0 & 0 \end{bmatrix} * \begin{bmatrix} 1 & 0 & 0 & x_L \\ 0 & 1 & 0 & y_L \\ 0 & 0 & 1 & z_L \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\Rightarrow \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_L - \dfrac{(x-x_L)}{(y-y_L)/y_L} \\ 0 \\ z_L - \dfrac{z-z_L}{(y-y_L)/y_L} \\ 1 \end{bmatrix}$$

* with openGL program, we can alter model-view matrix to form the desired polygon as follows.

```
GLfloat m[16];                    /* shadow projection */
for(i=0; i<m; i++)
    m[i] = 0.0;
m[0] = m[5] = m[10] = 1.0;
m[7] = -1.0/yl

glColor3fv (polygon_color);
glBegin (GL_POLYGON);

    ___
    ___                     /* Draw the polygon
    ___                           normally */.
    ___

glEnd ();
```

```
glMatrixMode (GL_MODELVIEW);
glPushMatrix ();                        /* save state */
glTranslatef (xl, yl, zl);              /* Translate back */
glMultMatrixf (m);                      /* project */
glTranslate (-xl, -yl, -zl);            /* move light to origin */
glColor3fv (shadow_color);
glBegin (GL_POLYGON);
      ---
      ---                               /* Draw the polygon
      ---                                        again */
      ---
glEnd ();
glPopMatrix ();                         /* restore state */
```

Topics left

- Hidden surface removal
- interactive mesh displays

UNIT 8:

IMPLEMENTATION.

Syllabus

* Basic Implementation strategies.
* Major tasks
* clipping
* line-segment clipping
* polygon clipping
* clipping of other primitives
* clipping in three dimensions.
* Rasterization.
* Bresenham's algorithm.
* polygon rasterization.
* Hidden surface removal
* Antialiasing
* Display considerations

– 8 Hours.

Ashok Kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY

BASIC IMPLEMENTATION STRATEGIES

* There are two strategies that are followed
  - object oriented approach
  - image oriented approach.

object oriented approach.

* Here the outer loop is over the objects:-

```
for (each-object)
{ render (object);
}
```

* Vertices are defined by programs and flow through a sequence of modules that transforms them, colors them and determine whether they are visible or not.

* This approach uses a pipeline that contains hardware for each of the tasks. Data flows forward through the system as shown -

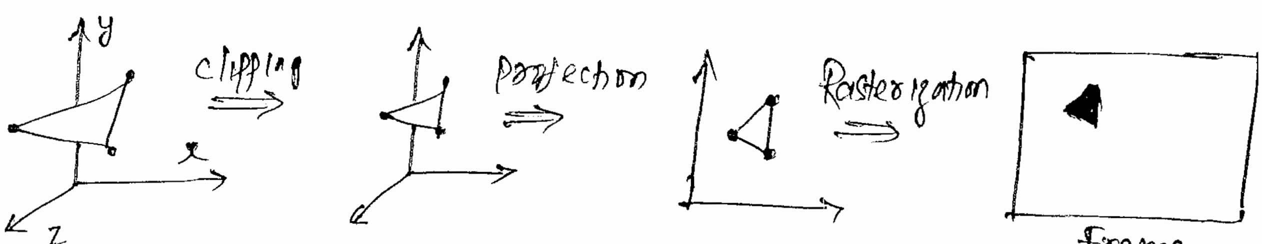


fig: object oriented approach.

Frame buffer.

Adv

1. Availability of geometric parallel pipelined processors makes processing simple since they can process millions of polygons per second.

Disadv

1. They cannot handle most global calculations. Because each geometric primitive is processed independently and in arbitrary order, complex shading effects that involve multiple objects cannot be handled efficiently.

Image oriented approach.

* Here, loop is over pixels. In pseudocode, the outer loop of such a program is of following form -

```
for (each-pixel)
{   assign-a-color (pixel);
}
```

Adv.

we need only limited display m/m at any time. Because results do not vary greatly from pixel to pixel. this coherence can be used to develop incremental forms of algorithm.

Disadv.

Complicated datastructures would be required.

FOUR MAJOR TASKS

* There are four major tasks that any graphics system must perform to render a geometric entity -

1. Modeling
2. Geometry processing
3. Rasterization.
4. Fragment processing.

modelling → Geometry processing → Rasterization → Fragment processing → frame buffer

fig: pipeline implementation.

note: These tasks should be performed in both the approaches.

Modeling

* It refers to processing a set of vertices that specify a primitive (geometric object)
* modelers are black boxes that produce geometric objects and are usually interactive in nature.

Geometric processing

* modeling results in set of vertices that specify a group of geometric objects.
* geometric processing works with these vertices.
* Goal of geometry processor are to determine which geometric objects can appear on the display and to assign shades or colors to the vertices of these objects.
* four processes are required - projection, primitive assembly, clipping, and shading.
* First step in geometry processing is to change representation from ~~for~~ object coordinates to camera / eye coordinates.
* second step is to transform vertices using the projection transformation

Rasterization.

* For line segments, rasterization determines which fragments should be used to approximate a line seg.
* For polygons, rasterization determines which pixels lie inside the polygon.

Fragment processing.

* It refers to the process of hidden surface removal in case of opaque objects and blending of colors of pixels in case of translucent objects.

CLIPPING.

* Clipping is the process of determining which primitives, or parts of primitives fit within the clipping or view volume defined by the application program.
* In general, portions of all primitives that can be displayed lies within -

$$-w \leq x \leq w$$
$$-w \leq y \leq w$$
$$-w \leq z \leq w$$

LINE SEGMENT CLIPPING.

* A clipper decides which primitives or parts of primitives can possibly appear on the display and are passed onto the rasterizer.
* The two main line-segment clipping algorithm are -
   1. Cohen-Sutherland clipping algorithm
   2. Liang-Barsky clipping algorithm

note: primitives that fit within the specified view volume are "accepted" (ie pass through the clipper)
primitives that cannot appear on the display are eliminated, or "rejected", or culled.

Cohen-Sutherland clipping

* The two dimensional clipping problem for line segments is shown in below fig.

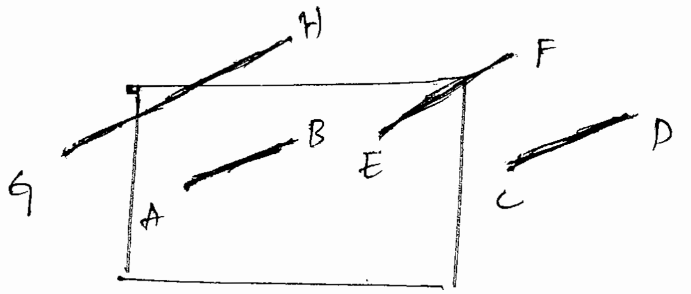


fig: Two-dimensional clipping.

* Algorithm starts by extending the sides of window to infinity, thus breaking up space into the nine regions as shown.

1001	1000	1010	
0001	0000	0010	$y = y_{max}$
0101	0100	0110	$y = y_{min}$
$x = x_{min}$		$x = x_{max}$	

* Each region can be assigned a unique 4-bit binary no. or outcode, $b_0 b_1 b_2 b_3$ as follows -

suppose that (x, y) is a point in the region, then

$$b_0 = \begin{cases} 1 & \text{if } y > y_{max} \\ 0 & \text{otherwise} \end{cases} \qquad b_1 = \begin{cases} 1 & \text{if } y < y_{min} \\ 0 & \text{otherwise} \end{cases}$$

$$b_2 = \begin{cases} 1 & \text{if } x > x_{max} \\ 0 & \text{otherwise} \end{cases} \qquad b_3 = \begin{cases} 1 & \text{if } x < x_{min} \\ 0 & \text{otherwise} \end{cases}$$

* For each end point of a line segment, we first compute the end point's outcode.

(It may require 8 floating point subtractions per line segment)

* consider a line segment whose outcodes are given by o_1 = outcode (x_1, y_1) and o_2 = outcode (x_2, y_2)

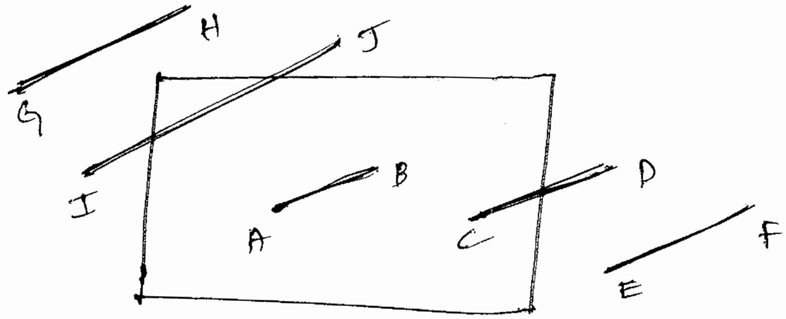


fig: cases of outcodes in cohen-sutherland algorithm.

there are 4 cases -

case 1:

if $(o_1 = o_2 = 0)$ then both end points are inside the clipping window (as in AB) and hence the segment can be sent on to be rasterized.

case 2:

if $(o_1 \neq 0, o_2 = 0$ or vice-versa) then one end point is inside the clipping window & the other is outside (as in CD) the line segment must be shortened.

case 3:

if $(o_1 \& o_2 \neq 0)$ then by taking bitwise AND of the outcodes, we determine whether or not the two end points lie on the same outside side of the window. If so, the line segment can be discarded (see EF in above fig)

case 4:

if $(o_1 \& o_2 = 0)$ then both end points are outside, but they are on the outside of different edges of the window. (see GH & IJ). We cannot tell from just outcodes whether the linesegment can be discarded or ~~not~~ must be shortened. ∴ we intersect with one of the sides of the window & check the outcode of the resulting point.

Adv.

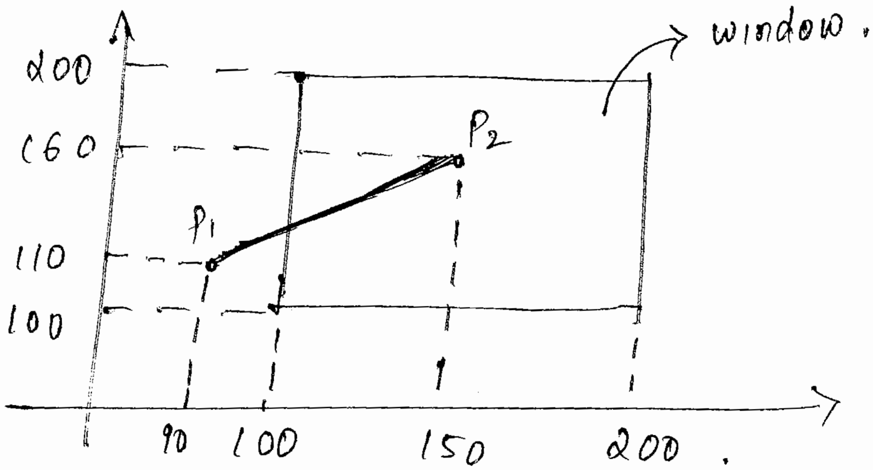
1. Avoids floating point division operations.
2. can be extended to three dimensions.

Disadv.

1. It must be used recursively.
2. It requires clipping window in a rectangular fashion.

Example:

consider below scenario:



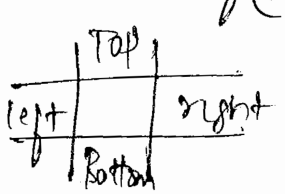
Steps to be followed.

Step 1: Calc. outcodes for both end points.

step 2: if (p1 || p2 == 0) then the line is completely inside the window. ie line is accepted.

step 3: if (p1 && p2 > 0) reject the line (ie o/s)
else if (p1 && p2 == 0) then the line is partial.

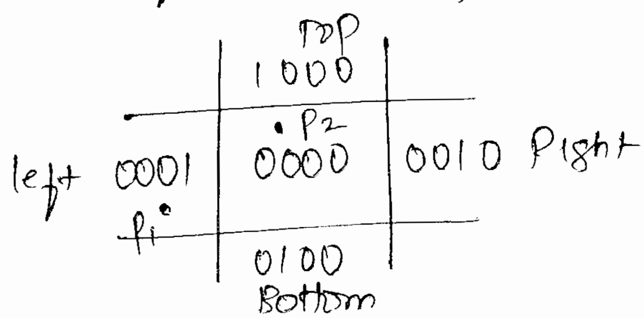
note: if (p1 && top > 0) The point is above the window.
if (p1 && bottom > 0) — " — below — " ——
if (p1 && left > 0) point is less than x_{min}
if (p1 && right > 0) — " — more — " — x_{max}.



Step 1: Calculate the outcodes for both end points -

$P_1 = 0001$

$P_2 = 0000$



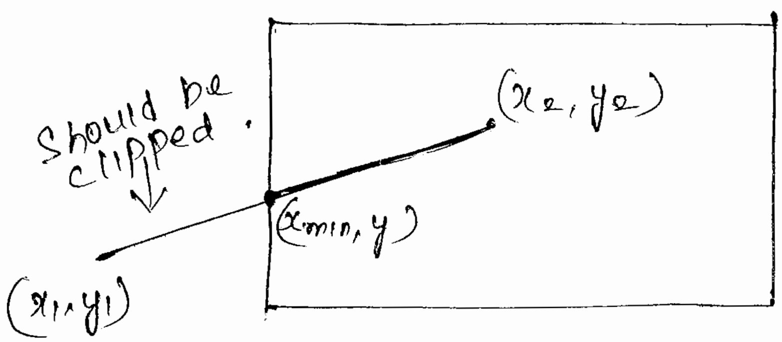
Step 2:

$P_1 \,||\, P_2 = 0001$, which is not equal to 0.

Step 3: $P_1 \,\&\&\, P_2 =$

$$\begin{array}{r} 0001 \\ 0000 \\ \hline 0000 \end{array}$$

∴ the line is partial.



Here, we need to find $y = ??$

~~formula~~:

$p_1 \,\&\&\, top = 0000$

$p_1 \,\&\&\, bottom = 0000$

$p_1 \,\&\&\, left = 0001$ ∴ left portion should be clipped.

formula: $y = y_1 + \left(\dfrac{y_2 - y_1}{x_2 - x_1}\right)(x_{min} - x_1)$, $x = x_{min}$

$= 110 + \dfrac{50}{60} \times 10$

$\Rightarrow \boxed{y = 118.3}$ $\boxed{x = 100}$

note:

* for clipping left portions:

$\cancel{y=}$ $x = x_{min}$

$$y = y_1 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_{min} - x)$$

* for clipping right portions:

$x = x_{max}$.

$$y = y_1 + \left(\frac{y_2 - y_1}{x_2 - x_1}\right)(x_{max} - x_1).$$

* For clipping Top portions:

$y = y_{max}$

$$x = x_1 + \left(\frac{x_2 - x_1}{y_2 - y_1}\right)(y_1 - y_{max}).$$

* For clipping bottom portions:

$y = y_{min}$

$$x = x_1 + \left(\frac{x_2 - x_1}{y_2 - y_1}\right)(y_1 - y_{min})$$

Liang-Barsky clipping. (more efficient)

* makes use of the parametric form of lines.
* Suppose that we have a line segment defined by the two end points $p_1 = [x_1, y_1]^T$ and $p_2 = [x_2, y_2]^T$
* we can write the parametric form of this line segment as -

$$p(\alpha) = (1-\alpha)p_1 + \alpha p_2 \qquad 1 \geq \alpha \geq 0.$$

or as two scalar eqns -

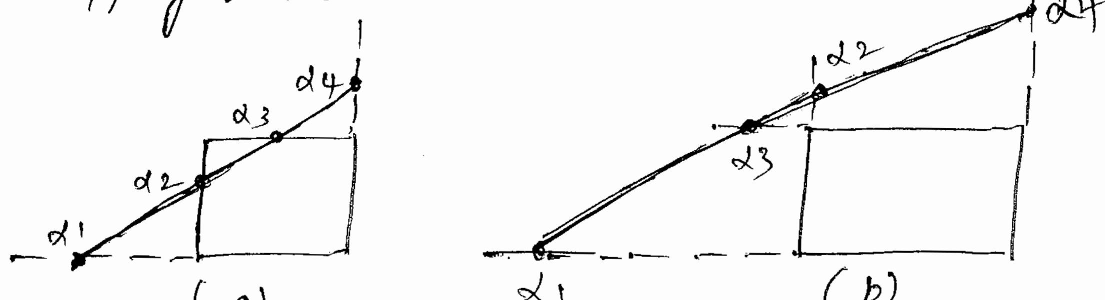
$$x(\alpha) = (1-\alpha)x_1 + \alpha x_2$$
$$y(\alpha) = (1-\alpha)y_1 + \alpha y_2.$$

As the parameter α varies from 0 to 1, we move along the segment from p_1 to p_2

α = -ve $\Rightarrow$ yields points on the line on other side of p_1

$\alpha > 1$ $\Rightarrow$ yields points on the line past p_2, going off to ∞.

* Below fig shows two cases of parametric line & a clipping window



(a) (b)

There are four points where the line intersects the extended sides of the window. These points correspond to the four values of the parameters: $\alpha_1, \alpha_2, \alpha_3,$ & α_4. (order is very imp)

$\alpha_1 \rightarrow$ bottom $\qquad \alpha_3 \rightarrow$ top

$\alpha_2 \rightarrow$ left. $\qquad \alpha_4 \rightarrow$ right.

* In fig (a)

$1 > \alpha_4 > \alpha_3 > \alpha_2 > \alpha_1$

ie line intersects right, top, left, bottom in order

$\Rightarrow$ ~~Accept~~, shorten.

* In fig (b)

$1 > \alpha_4 > \alpha_2 > \alpha_3 > \alpha_1$

ie line intersects right, left, ~~bo~~ top, bottom

$\Rightarrow$ Reject.

note:

To determine the intersection with top of the window, we find the intersection at the value -

$$\alpha = \frac{y_{max} - y_1}{y_2 - y_1} \quad \text{--- (1)}$$

Similar eqns holds for other 3 sides of the window.

(1) can be written as -

$\alpha (y_2 - y_1) = y_{max} - y_1$

$$\boxed{\alpha \Delta y = \Delta y_{max}}$$

POLYGON CLIPPING.

* It is not as simple as line segment clipping.
  - clipping a line segment yields atmost one line segment.
  - clipping a polygon can yield multiple polygons.

  However clipping a convex polygon can yield atmost one other polygon.

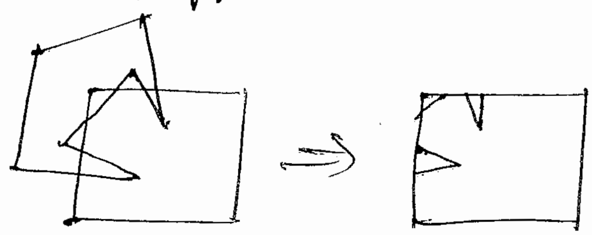
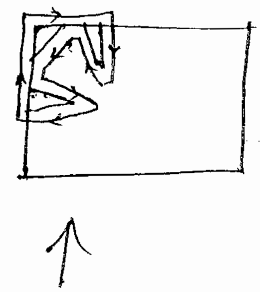


fig: clipping of a concave polygon.



It yields multiple polygons. Sol^n: assume them or create one single polygon as shown above (using traversing)

disadv: still complex.

* Therefore, we replace concave polygons with a set of triangular polygons. This process is called tessellation (ie divide a ~~con~~ given polygon into set of convex polygons)

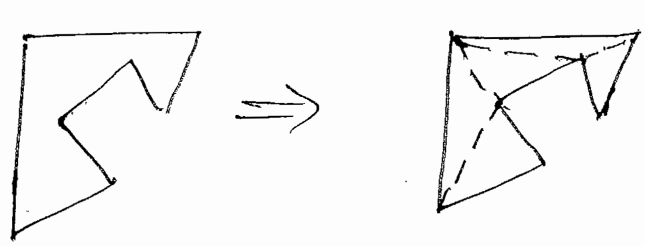
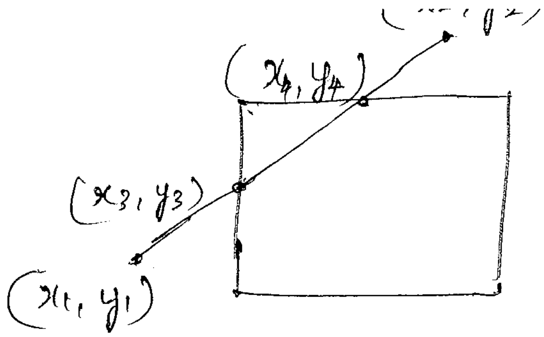


fig: Tessellation of a concave polygon.

* line segment clipping can be considered as (or envisioned as) a black box whose i/p is the pair of vertices from the segment to be tested and clipped, and whose o/p either is a pair of vertices corresponding to the clipped line segment or is nothing if the input line segment lies ~~o/w~~ o/s the window.



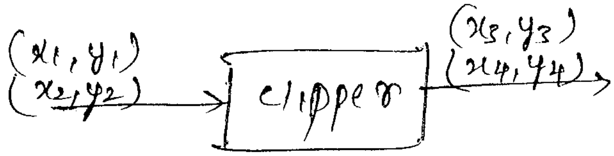
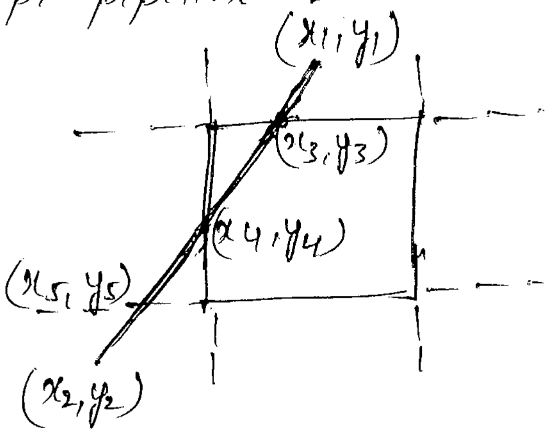
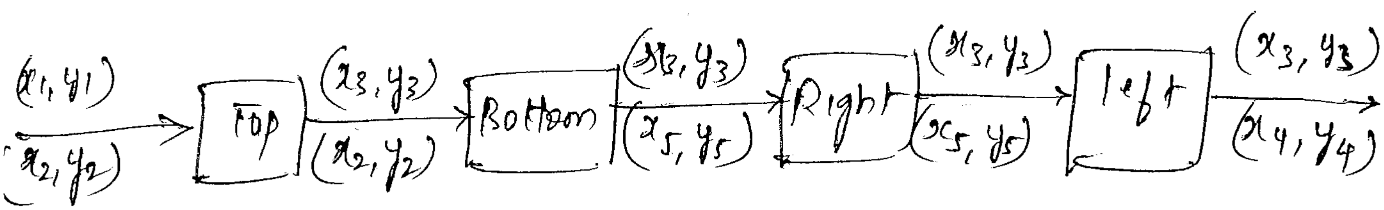


fig: clipper as a black box.

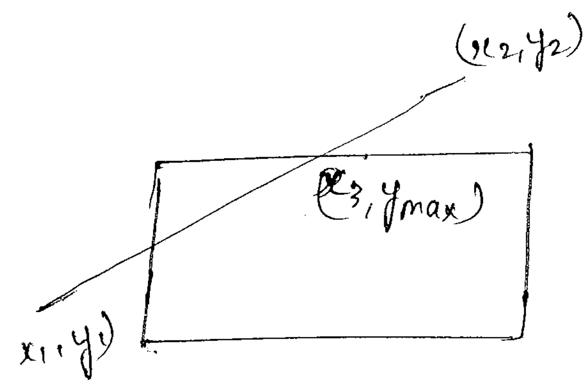
* clipping against each side of window is independent of other sides.

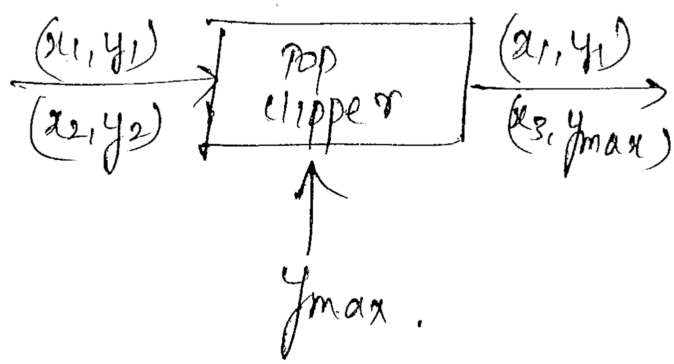
we can use four independent clippers arranged in a ~~pipepl~~ pipeline -





* for eg: top clipper would be like -





* Fig below shows a simple example of the effect of successive clippers on a polygon.

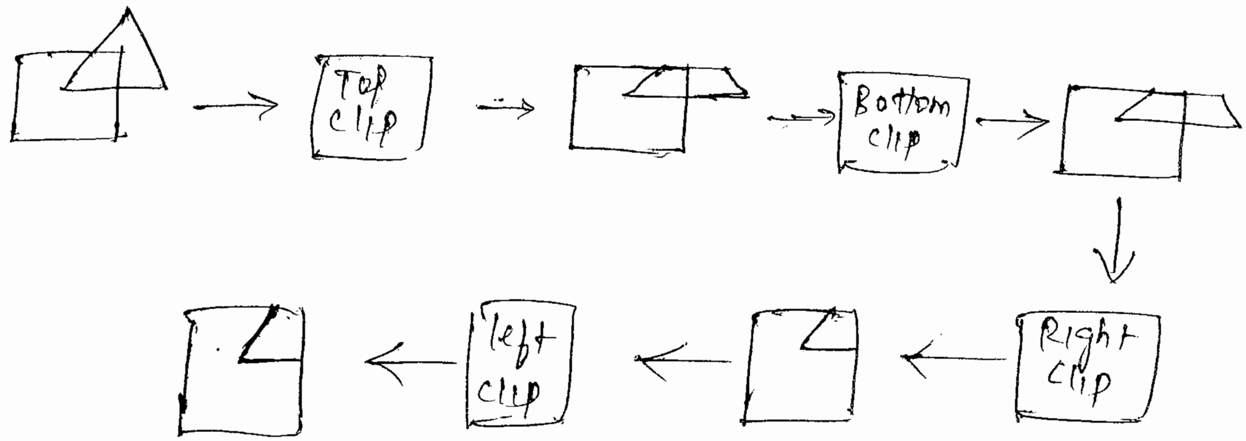


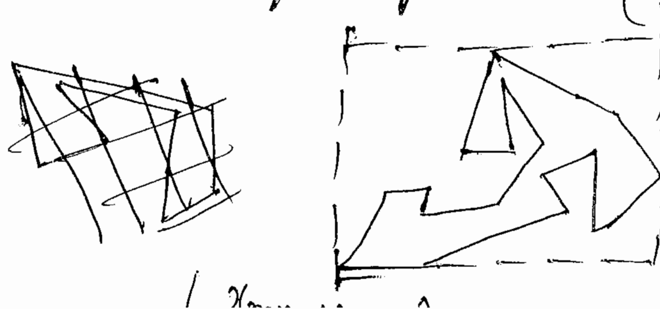
fig: pipe line clipping of polygons.

<u>note</u>: for three dimensional clipping - Add front and back clippers. It results in small increase in latency.

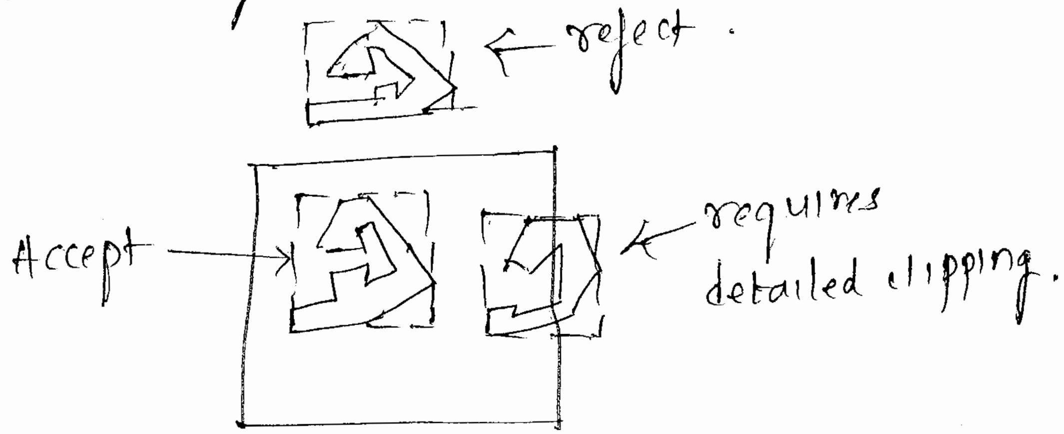
CLIPPING OF OTHER PRIMITIVES.

* Suppose we have an many sided (or complex) polygon, In such a case, Bounding boxes technique can be used for clipping purpose.
* Axis-aligned bounding box or extent of a polygon is the smallest ~~rectangular~~ rectangle (aligned with the window) that contains the polygon.
* Its very simple to compute the bounding box: just find max and min of x & y.

(x_{max}, y_{max})



* we can usually determine accept/reject based only on bounding box.



RASTERIZATION.

* For line segments, rasterization determines which fragments should be used to approximate a line segment. For polygons, rasterization determines which pixels lie inside the polygon.
* ie Rasterization produces a set of fragments. Each fragment has a location in screen coordinates that corresponds to a pixel location in the color buffer.

DDA Algorithm

* simplest scan conversion algorithm for line segments is DDA (Digital Differential Analyzer) algorithm. DDA was an early electromechanical device for digital simulation of differential equations.
* Because a line satisfies the differential equation -

  $\frac{dy}{dx} = m$ where $m \rightarrow$ slope,

  generating a line segment is equivalent to solving a simple differential equation numerically.

* Suppose that we have a line segment defined by the end points (x_1, y_1) and (x_2, y_2). The slope is given by -

$$m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

we assume that $0 \leq m \leq 1$. we can handle other values of m using symmetry.

<u>note:</u> color buffer is an $n \times m$ array of pixels. pixels can be set to a given color by a single function inside the graphics implementation of the following form -

writepixel (int ix, int iy, int value);

value → either index in color-index mode or a pointer to an RGBA color.

* Our algo. is based on writing a pixel for each value of ix in write-pixel as x goes from x_1 to x_2.
If we are on the line seg as shown in below fig, for any change in x equal to Δx, the corresponding change in y must be -

$$\Delta y = m \Delta x$$

As we move from x_1 to x_2, we increase x by 1 each iteration; thus we must increase y by -

$$\Delta y = m.$$

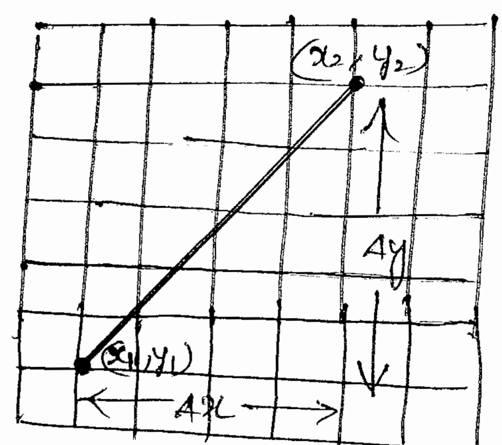


fig: line seg in window coordinates.

* Although x is an integer, y is not, Because 'm' is a floating point number. Therefore we must round it to find the appropriate pixel as shown -

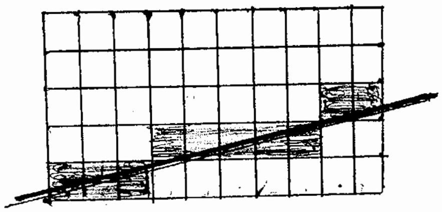


fig: pixels generated by DDA Algorithm.

Disadv: many floating point calculations.

* our algorithm in pseudocode is -

```
for( ix = x1 ; ix <= x2 ; ix++ )
{    y+ = m;
     write_pixel (x, round(y), line_color);
}
```

problem with DDA algorithm

* Reason that we limited the max slope to 1 can be see from the fig shown - ie for large slopes, the separation b/w pixels that are colored may be large, generating unacceptable approximation of line segment. ∴ our algo is of this form - for each x, find the best y.

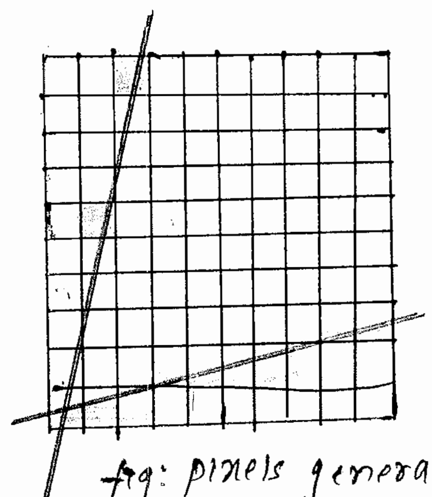
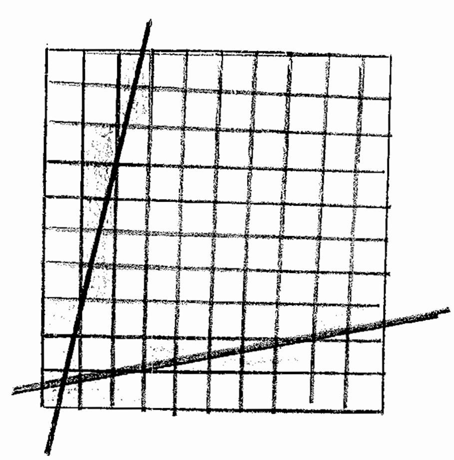


fig: pixels generated by high & low slope lines.

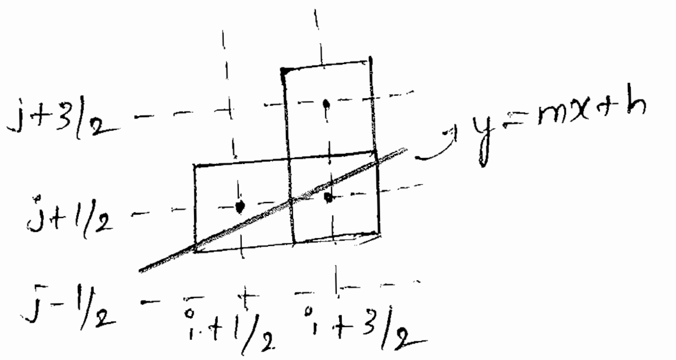
Soln ? :

For $m>1$, we swap the roles of x and y. The algo becomes this - for each y, find the best x. →

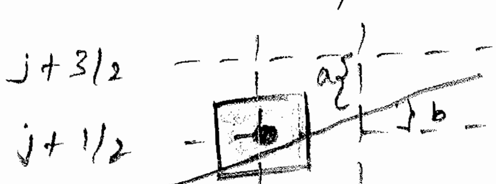


BRESSENHAM'S ALGORITHM.

* DDA Algorithm requires a floating point addition for each pixel generated.
* we can eliminate all floating point calculations through Bresenham's algorithm.
* consider end points of a line segment as (x_1, y_1) & (x_2, y_2) and the slope, $0 \leq m \leq 1$.
* Assume pixel centers are at half integers. If we start at a pixel that has been written, there are only two candidates for the next pixel to be written into the frame buffer.
* Ref fig, suppose that we are somewhere in the middle of the scan conversion of our line segment & have just placed a pixel at, $(i+1/2, j+1/2)$. we KT $y = mx + h$.



* The slope condition indicates that we must set the color of one of only two possible pixels - either the pixel at $(i+3/2, j+1/2)$ or the pixel at $(i+3/2, j+3/2)$. we use decision variable $d = b - a$ for this. where b and a are distances b/w the line and the upper & lower candidate pixels at $x = i + 3/2$ (refer below fig)



* If d is -ve, the line passes closer to the lower pixel, so we choose the pixel at $(i + 3/2, j + 1/2)$.
otherwise, we choose the pixel at $(i + 3/2, j + 3/2)$

Incremental form.

* more efficient if we look at d_k (value of decision variable at $x = k$)

$$d_{k+1} = d_k + \begin{cases} 2\Delta y & \text{if } d_k < 0 \\ 2(\Delta y - \Delta x) & \text{otherwise} \end{cases}$$

* The calculation of each ~~success~~ successive pixel in the color buffer requires only an addition an a sign test.
* this algo is so efficient: it has been incorporated as a single instruction on graphics chips.

CLIPPING IN THREE DIMENSION

* In 3D clipping, we clip against a bounded volume rather than against a bounded region in the plane.

The simplest extension of 2D clipping to 3D clipping is clipping for a right parallelopiped region.

The conditions are -

$$x_{min} \le x \le x_{max}$$
$$y_{min} \le y \le y_{max}$$
$$z_{min} \le z \le z_{max}$$

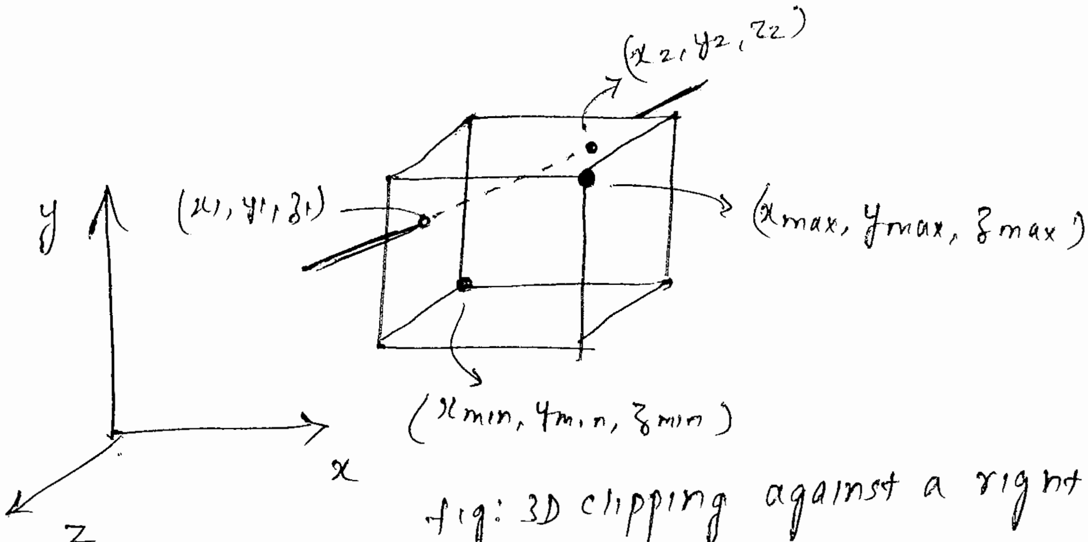
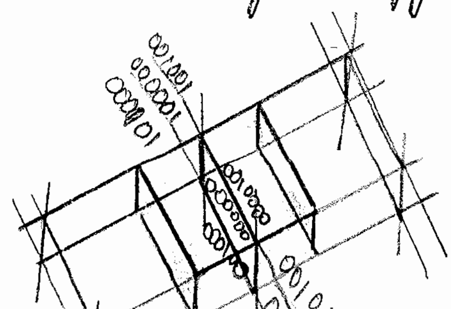


fig: 3D clipping against a right parallelopiped.

* Both cohen sutherland and Liang Barsky algorithms can be extended to clip in 3 Dimensions.

* For cohen sutherland algo, we replace the 4 bit outcodes with a 6 bit outcode. Additional two points are set if the points lie either in front of or behind the clipping volume. Testing strategy is identical ~~to 2D~~ for the 2D & 3D cases.



(refer in text)

* for Liang Barsky algorithm, we add the equation

$$z(\alpha) = (1-\alpha) z_1 + \alpha z_2$$ to the existing eqns.-

$$x(\alpha) = (1-\alpha) x_1 + \alpha x_2$$

$$y(\alpha) = (1-\alpha) y_1 + \alpha y_2$$

we have to consider 6 intersections with the surfaces that forms the clipping volume.
pipeline clippers would add two more modules to clip against the front & back.

HIDDEN SURFACE REMOVAL.

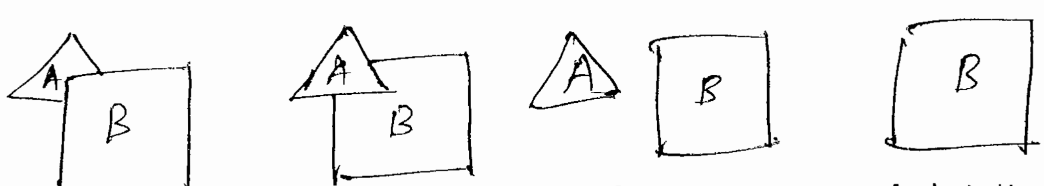
* Hidden surface removal refers to the process of removing the surface which would not be visible to the viewer from the display.

(or) Hidden surface removal (or visible surface determination) is done to discover what part (if any) of object in the view volume is visible to the viewer or is obscured from the viewer by other objects.

object space approach

* Here we consider the objects pair wise, as seen from the center of projection.

eg: consider two such polygons A & B. There are 4 possibilities (refer fig)



B partially obscures A — A partially obscures B — Both A & B are visible — B totally obscures A.

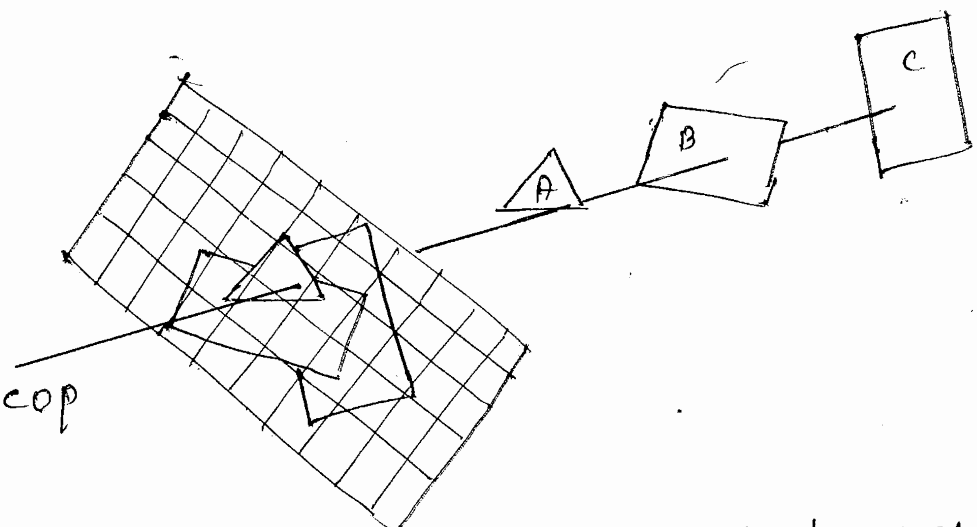
* worst case complexity in this approach is $O(k^2)$

∵ given k polygons, we pick one of the k polygons and compare it pairwise with remaining k-1 polygons

* Ex for Hidden surface Algorithm techniques that uses object space approach are -
    1. painters Algorithm
    2. Depth sort
    3. Backface removal (culling)

image space approach.

* It follows our viewing and ray casting model (ref fig)

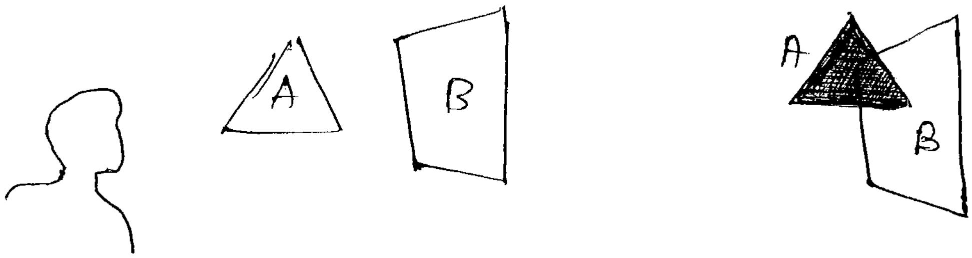


* idea: look at each projector and find closest of k polygons & color the corresponding pixel with appropriate shade.
* For n x m display (frame buffer), we have to carry out this operation ~~nm~~ nmk times, giving O(k) complexity.
* Ex for algorithms that uses this approach -

    1. Z buffer Algorithm
    2. scanline Algorithm.

Painter's Algorithm

* consider the polygons as shown in the fig –



* We can see that the polygon in the front (ie A) partially obscures the other.
* We can render this scene using painters algorithm as follows.

  Render the rear polygon first and then the front polygon, painting over the part of rear polygon not visible to the viewer in this process.

  Both polygons would be rendered completely, with the hidden surface removal being done as a consequence of the Back to front rendering of the polygons..

* Two questions arises
  - How to do the sort ?? → ie which is far & which is nearer.
  - what ~~do~~ to do if polygons overlap??

  soln – Depth sort.

Depth sort.

* First order all the polygons by how far away from the viewer their maximum z-value is.
* suppose that the order is as shown in the fig.

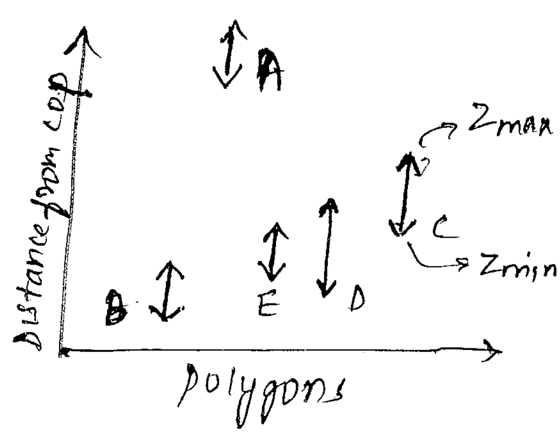
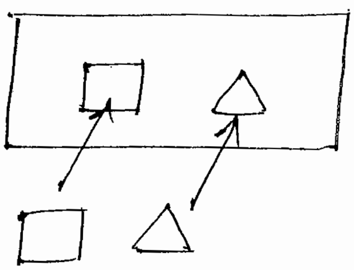


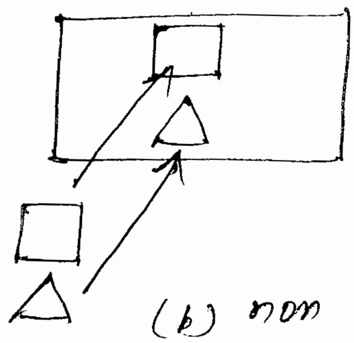
fig: Z-extents of sorted polygons.

* We can see that polygon A is behind all the other polygons (ie not overlapping) and can be painted first. However, the others cannot be painted based solely on z-extent.

* What to do if polygons overlap in z-extent?
Consider a pair of polygons whose z-extents overlap and check their x and y-extents. (refer fig)



(a) non overlapping x-extents



(b) non overlapping y-extents.

* If either x or y extends do not overlap, neither polygon can obscure the other, & they can be painted in either order.

* Even if these tests fail, it may be still possible to find an order in which we can paint the polygons individually.
fig shows such a case. one is fully on one side of the other.

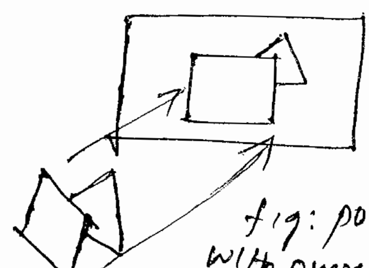
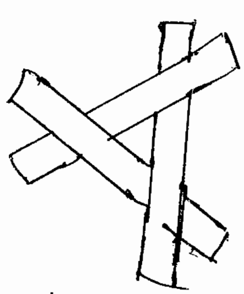
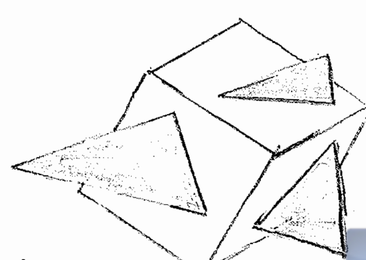


fig: polygon with overlapping extents.

* Two troublesome situation remain →



(a) cyclic overlap.



(b) piercing polygons

Back Face Removal (culling)

* The test for culling a back facing polygon can be derived from below fig

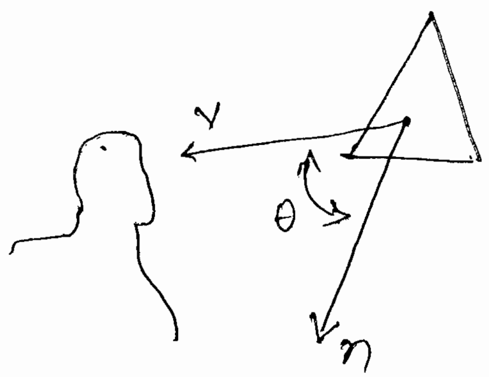


fig: Back face Test.

* If θ is the angle b/w the normal & the viewer, then the polygon is facing forward if and only if

$$-90 \leq \theta \leq 90 \quad \text{or}$$

equivalently, $\cos\theta \geq 0$.

or, $n \cdot v \geq 0$

↳ dot product.

* We can even simplify this test.

in normalized device coordinates,

$$v = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Thus, if the polygon is on the surface,

$ax + by + cz + d = 0$ in normalized device coordinates,

We need only to check the sign of c to determine whether we have a front or back facing polygon.

The Z-buffer Algorithm.

* most widely used.
* suppose that we are in the process of rasterizing one of the two polygons shown in the fig.

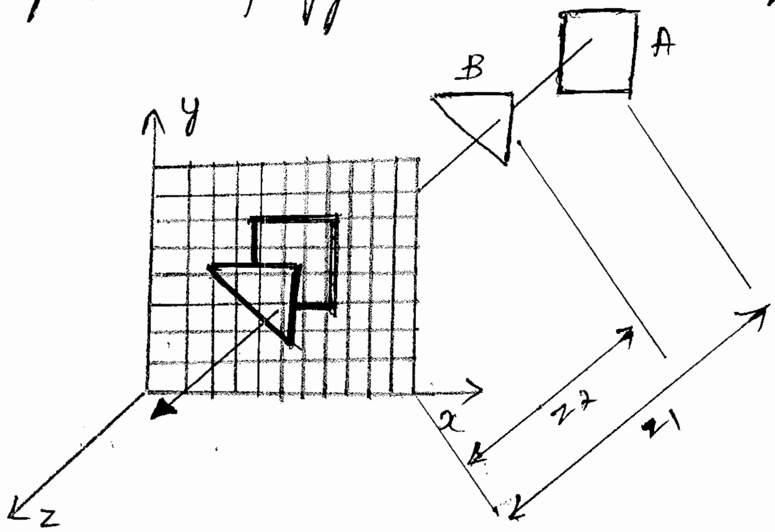


fig: The z-buffer algorithm

* use a buffer, the z-buffer with the same resolution as the frame buffer and with depth consistent with the resolution that we wish to use for ~~deep~~ distance.
  Initially each element in the depth buffer (z buffer) is initialized to a depth corresponding to the max. distance away from the COP.
  Color buffer is initialized to background color.
* Working: we rasterize polygon by polygon using any one of the methods.
  For each fragment on the polygon corresponding to the intersection of the polygon with a ray through a pixel, we compute the depth from COP.
  We compare this depth to the value in the z buffer corresponding to this fragment
  If this depth is greater than the depth in z buffer, then we have already processed a polygon with a corresponding fragment

closer to the viewer, & this fragment is not visible.

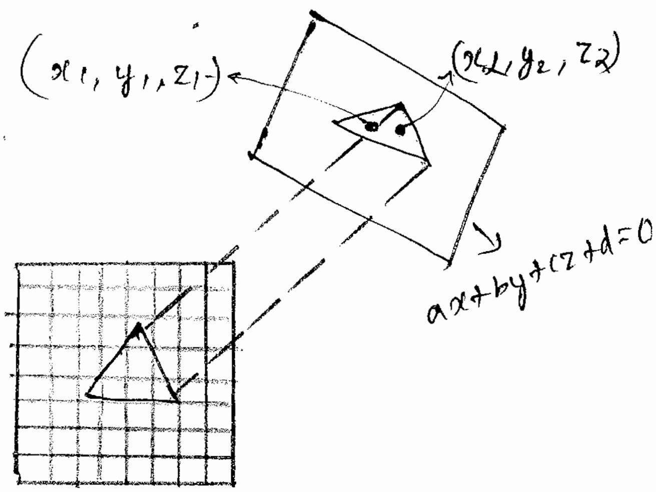
If the depth is less than the depth in z-buffer, then we found a fragment closer to the viewer. we update the depth in the zbuffer & place the shade computed for this fragment at the corresponding location in the color buffer.

Efficiency.

* suppose that we are rasterizing a polygon scanline by scanline the polygon is part of plane (ref fig) that can be represented as -

$$ax+by+cz+d=0.$$

suppose that (x_1, y_1, z_1) and (x_2, y_2, z_2) are two points on the polygon.



then eqn for plane can be written in differential form as -

$$a\Delta x + b\Delta y + c\Delta z = 0$$

where $\Delta x = x_2 - x_1$

$\Delta y = y_2 - y_1$

$\Delta z = z_2 - z_1$.

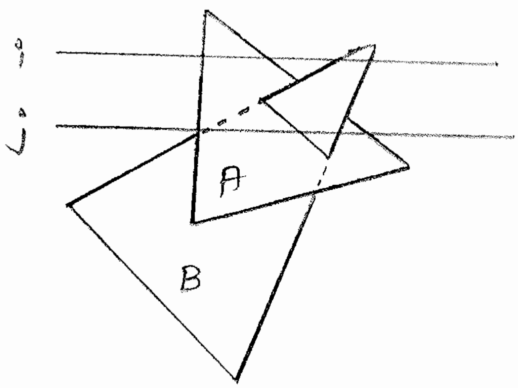
when we raster a polygon scanline by scanline, they $\Delta y = 0$ and we increase x in unit steps, so Δx is constant

$$\therefore \boxed{\Delta z = -\frac{a}{c}\Delta x}$$

This value is constant that needs to be computed only once for each polygon.

Scan-line Algorithm

* can combine ~~sca~~ shading & hidden surface removal ~~us~~ through scanline Algorithm.



scan line i :

no need for depth information, can only be in no or one polygon

scan line j:

need depth information only when in more than one polygon.

Ashok Kumar K
VIVEKANANDA INSTITUTE OF TECHNOLOGY