SYLLABUS

COMPUTER GRAPHICS AND VISUALIZATION

Subject Code: 10CS65	**I.A. Marks : 25**
Hours/Week : 04	**Exam Hours: 03**
Total Hours : 52	**Exam Marks: 100**

PART - A

UNIT - 1

INTRODUCTION: 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. Graphics Programming: The Sierpinski gasket; Programming two-dimensional applications.

7 Hours

UNIT - 2

THE OPENGL: The OpenGL API; Primitives and a6ributes; Color; Viewing; Control functions; The Gasket program; Polygons and recursion; The three-dimensional gasket; Plo8ng implicit functions.

6 Hours

UNIT - 3

INPUT AND INTERACTION: Interaction; Input devices; Clients and servers; Display lists; Display lists and modeling; Programming event-driven input; Menus; Picking; A simple CAD program; Building interactive models; Animating interactive programs; Design of interactive programs; Logic operations.

7 Hours

UNIT - 4

GEOMETRIC OBJECTS AND TRANSFORMATIONS – 1: Scalars, points, and vectors; Three-dimensional primitives; Coordinate systems and frames; Modeling a colored cube; Affine transformations; Rotation, translation and scaling.

6 Hours

PART - B

UNIT - 5

GEOMETRIC OBJECTS AND TRANSFORMATIONS – 2: Transformations in homogeneous coordinates; Concatenation of transformations; OpenGL transformation matrices; Interfaces to three-dimensional applications; Quaternions.

5 Hours

UNIT - 6

VIEWING: Classical and computer viewing; Viewing with a computer; Positioning 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

UNIT - 7

LIGHTING AND SHADING: Light and ma6er; Light sources; The Phong lighting model; Computation of vectors; Polygonal shading; Approximation of a sphere by recursive subdivisions; Light sources in OpenGL; Specification of materials in OpenGL; Shading of the sphere model; Global illumination.

6 Hours

UNIT - 8

IMPLEMENTATION: Basic implementation strategies; The 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

TEXT BOOK:

1. **Interactive Computer Graphics A Top-Down Approach with OpenGL** -Edward Angel, 5th Edition, Addison-Wesley, 2008.

REFERENCE BOOKS:

1. **Computer Graphics Using OpenGL** – F.S. Hill,Jr. 2nd Edition, Pearson 1. Education, 2001.
2. **Computer Graphics** – James D Foley, Andries Van Dam, Steven K Feiner, John F Hughes, Addison-wesley 1997.
3. **Computer Graphics - OpenGL Version** – Donald Hearn and Pauline Baker, 2nd Edition, Pearson Education, 2003.

TABLE OF CONTENTS

PART - A

UNIT - 1 **7 Hours**

INTRODUCTION

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

Graphics Programming:

The Sierpinski gasket

Programming two-dimensional applications

UNIT -1

Graphics Systems and Models

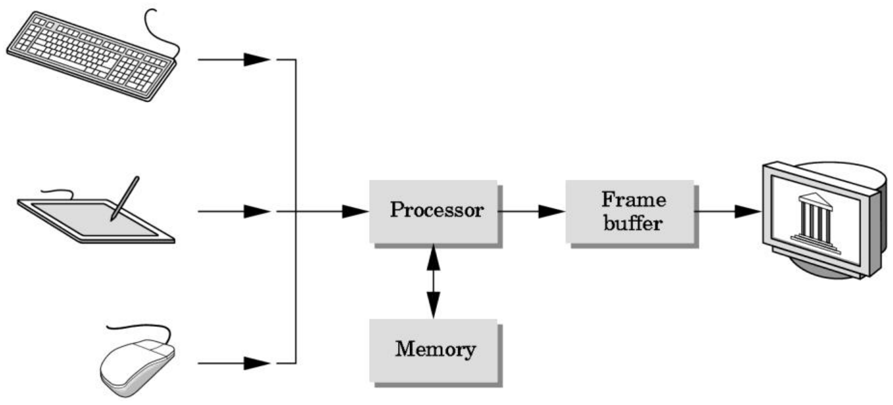
1.1 Applications of computer graphics:

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

1.2 Graphics systems

A Graphics system has 5 main elements:

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



Pixels and the Frame Buffer

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

Properties of frame buffer:

Resolution – number of pixels in the frame buffer

Depth or Precision – number of bits used for each pixel

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

8 bit deep frame buffer allows 256 colors.

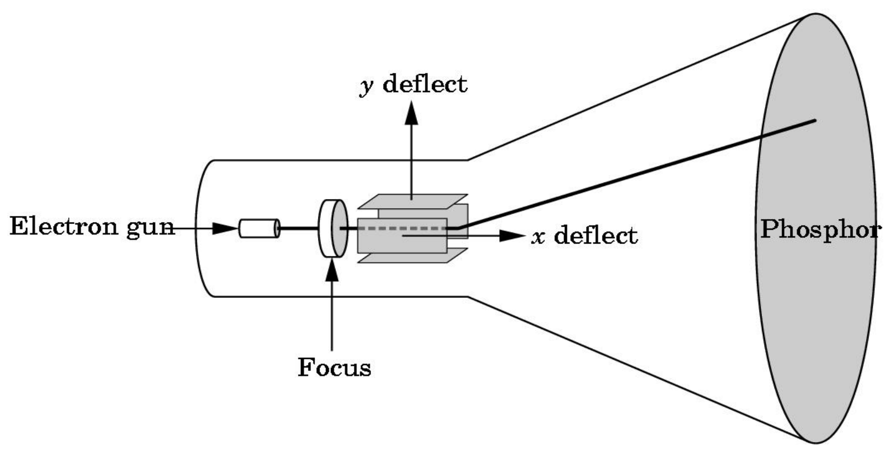
A Frame buffer is implemented either with special types of memory chips or it can be a part of system memory.

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

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

Output Devices

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



Various parts of a CRT :

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

2 types of refresh:

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

1.3 Images: Physical and synthetic

Elements of image formation:

- Objects
- Viewer
- Light source (s)

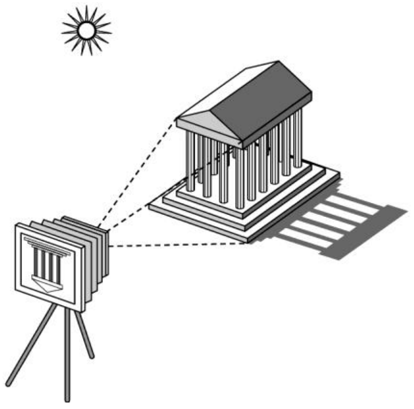
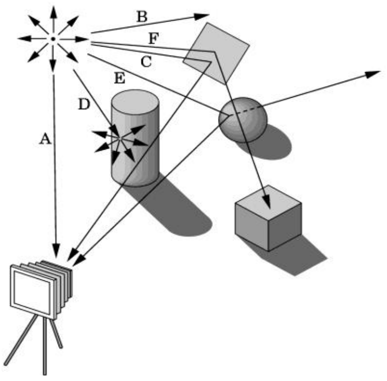


Image formation models

Ray tracing :

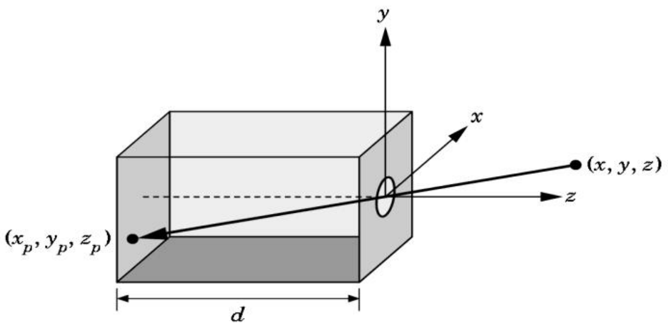
One way to form an image is to follow rays of light from a point source finding which rays enter the lens of the camera. However, each ray of light may have multiple interactions with objects before being absorbed or going to infinity.



1.4 Imaging systems

It is important to study the methods of image formation in the real world so that this could be utilized in image formation in the graphics systems as well.

1. Pinhole camera:

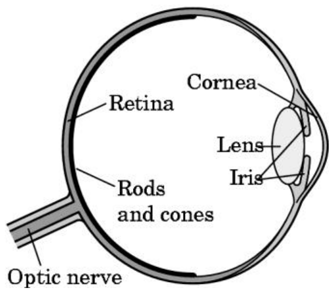


Use trigonometry to find projection of point at (x,y,z)

xp= -x/z/d yp= -y/z/d zp= d

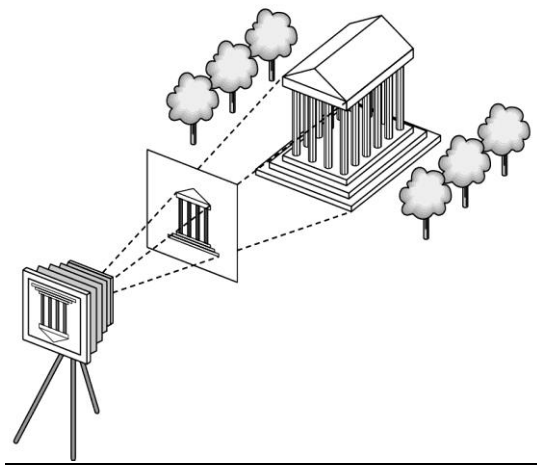
These are equations of simple perspective

2. Human visual system



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

1.5 The Synthetic camera model



The paradigm which looks at creating a computer generated image as being similar to forming an image using an optical system.

Various notions in the model :

Center of Projection

Projector lines

Image plane

Clipping window

- In case of image formation using optical systems, the image is flipped relative to the object.
- In synthetic camera model this is avoided by introducing a plane in front of the lens which is called the image plane.

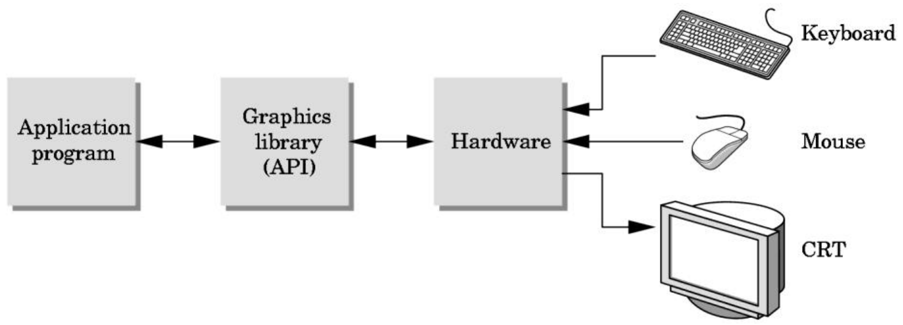
The angle of view of the camera poses a restriction on the part of the object which can be viewed.

This limitation is moved to the front of the camera by placing a Clipping Window in the projection plane.

<u>1.6 Programer's interface :</u>

A user interacts with the graphics system with self-contained packages and input devices. E.g. A paint editor.

This package or interface enables the user to create or modify images without having to write programs. The interface consists of a set of functions (API) that resides in a graphics library



- The application programmer uses the API functions and is shielded from the details of its implementation.

- The device driver is responsible to interpret the output of the API and converting it into a form understood by the particular hardware.

- The pen-plotter model

This is a 2-D system which moves a pen to draw images in 2 orthogonal directions.

E.g. : LOGO language implements this system.

moveto(x,y) – moves pen to (x,y) without tracing a line.

lineto(x,y) – moves pen to (x,y) by tracing a line.

Alternate raster based 2-D model :

Writes pixels directly to frame buffer

E.g. : write_pixel(x,y,color)

In order to obtain images of objects close to the real world, we need 3-D object model.

<u>3-D APIs (OpenGL - basics)</u>

To follow the synthetic camera model discussed earlier, the API should support:

Objects, viewers, light sources, material properties.

OpenGL defines primitives through a list of vertices.

Primitives: simple geometric objects having a simple relation between a list of vertices

Simple prog to draw a triangular polygon :

```
glBegin(GL_POLYGON)
        glVertex3f(0.0, 0.0, 0.0);
        glVertex3f(0.0, 1.0, 0.0);
        glVertex3f(0.0, 0.0, 1.0);
glEnd( );
```

Specifying viewer or camera:

Position - position of the COP

Orientation – rotation of the camera along 3 axes

Focal length – determines the size of image

Film Plane – has a height & width & can be adjusted independent of orientation of lens.

Function call for camera orientation :

gluLookAt(cop_x,cop_y,cop_z,at_x,at_y,at_z,up_x,up_y,up_z);

gluPerspective(field_of_view,aspect_ratio,near,far);

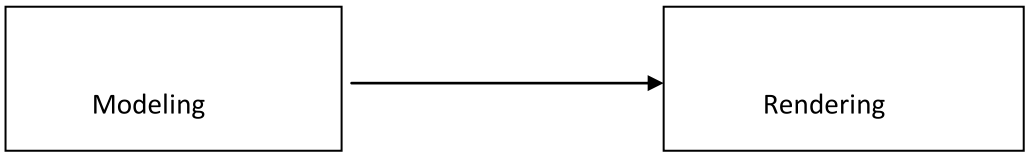
Lights and materials :

- Types of lights
  - Point sources vs distributed sources
  - Spot lights
  - Near and far sources
  - Color properties

- Material properties
  - Absorption: color properties
  - Scattering

Modeling Rendering Paradigm :

Viewing image formation as a 2 step process



E.g. Producing a single frame in an animation:

1st step : Designing and positioning objects

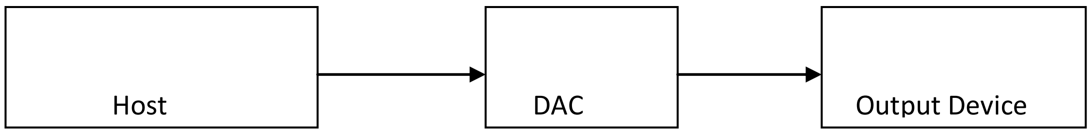
2nd step : Adding effects, light sources and other details

The interface can be a file with the model and additional info for final rendering.

1.7 Graphics Architectures

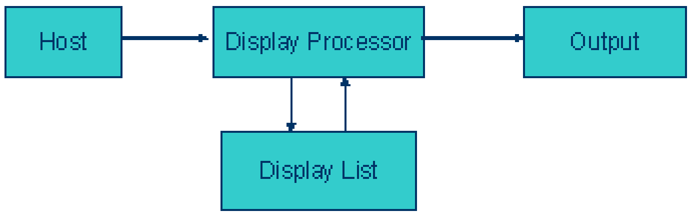
Combination of hardware and software that implements the functionality of the API.

- Early Graphics system :



Here the host system runs the application and generates vertices of the image.

Display processor architecture :



- Relieves the CPU from doing the refreshing action

- Display processor assembles instructions to generate image once & stores it in the Display List. This is executed repeatedly to avoid flicker.
- The whole process is independent of the host system.

1.8 Programmable Pipelines

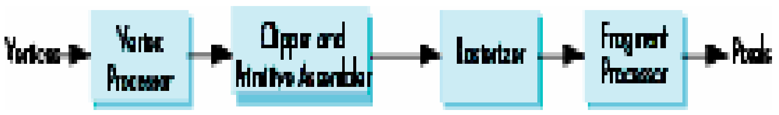
E.g. An arithmetic pipeline

Terminologies :

Latency : time taken from the first stage till the end result is produced.

Throughput : Number of outputs per given time.

Graphics Pipeline :



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

Vertex Processor

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

Primitive Assembly

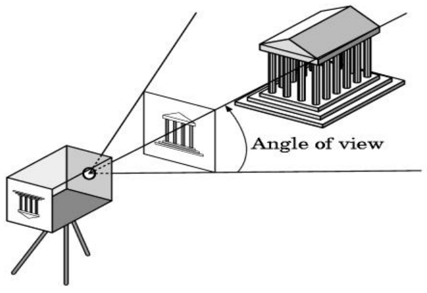
Vertices must be collected into geometric objects before clipping and rasterization can take place

- Line segments
- Polygons
- Curves and surfaces

Clipping

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

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



Rasterization :

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

Fragment Processor :

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

1.9 Graphics Programming

The Sierpinski Gasket :

It is an object that can be defined recursively & randomly

Basic Algorithm :

Start with 3 non-collinear points in space. Let the plane be z=0.

1. Pick an initial point (x,y,z) at random inside the triangle.
2. Select 1 of the 3 vertices in random.
3. Find the location halfway between the initial point & the randomly selected vertex.
4. Display the new point.
5. Replace the point (x,y,z) with the new point
6. Return to step 2.

Assumption : we view the 2-D space or surface as a subset of the 3-D space.

A point can be represented as p=(x,y,z). In the plane z=0, p = (x,y,0).

Vertex function genral form – glVertex*() - * is of the form ntv

n – dimensions (2,3,4)

t – data type (i,f,d)

v – if present, represents a pointer to an array.

Programing 2-D applications :

Definition of basic OpenGL types :

- E.g. – glVertex2i(Glint xi, Glint yi)

  or

```
#define GLfloat float.
GLfloat vertex[3]
glVertex3fv(vertex)
```

E.g. prog :

```
glBegin(GL_LINES);
        glVertex3f(x1,y1,z1);
        glVertex3f(x2,y2,z2);
glEnd();
```

The sierpinski gasket display() function :

```
void display()
{
        GLfloat vertices[3][3] = {{0.0,0.0,0.0},{25.0,50.0,0.0},{50.0,0.0,0.0}};
        /* an arbitrary triangle in the plane z=0 */
        GLfloat p[3] = {7.5,5.0,0.0}; /* initial point inside the triangle */
        int j,k;
        int rand();

        glBegin(GL_POINTS);
        for (k=0;k<5000;k++){
                j=rand()%3;
                p[0] = (p[0] + vertices[j][0])/2; /* compute new location */
                p[1] = (p[1] + vertices[j][1])/2;
                /* display new point */
                glVertex3fv(p);
        }
        glEnd();
        glFlush();
}
```

Coordinate Systems :

- One of the major advances in the graphics systems allows the users to work on any coordinate systems that they desire.
- The user’s coordinate system is known as the “world coordinate system”
- The actual coordinate system on the output device is known as the screen coordinates.
- The graphics system is responsible to map the user’s coordinate to the screen coordinate.

UNIT - 2 **6 Hours**

THE OPENGL

The OpenGL API

Primitives and attributes

Color

Viewing

Control functions

The Gasket program

Polygons and recursion

The three-dimensional gasket

Plotting implicit functions.

UNIT-2

THE OPENGL

2.1 The OpenGL API

- OpenGL is a software interface to graphics hardware.
- This interface consists of about 150 distinct commands that you use to specify the objects and operations needed to produce interactive three-dimensional applications.
- OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms.
- To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, you must work through whatever windowing system controls the particular hardware you're using.

The following list briefly describes the major graphics operations which OpenGL performs to render an image on the screen.

1. Construct shapes from geometric primitives, thereby creating mathematical descriptions of objects.
(OpenGL considers points, lines, polygons, images, and bitmaps to be primitives.)

2. Arrange the objects in three-dimensional space and select the desired vantage point for viewing the composed scene.

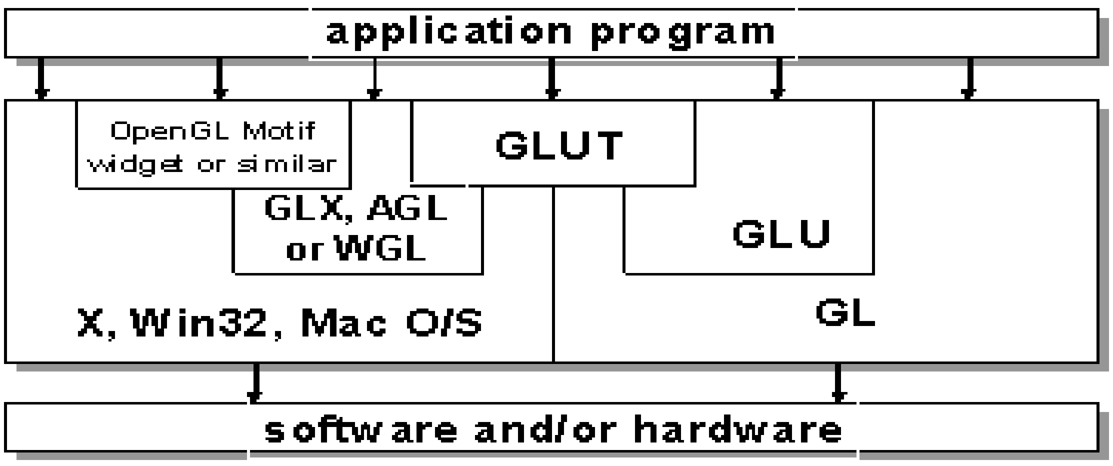
3. Calculate the color of all the objects. The color might be explicitly assigned by the application, determined from specified lighting conditions, obtained by pasting a texture onto the objects, or some combination of these three actions.

4. Convert the mathematical description of objects and their associated color information to pixels on the screen. This process is called *rasterization*.

2.2 OpenGL functions

- Primitive functions : Defines low level objects such as points, line segments, polygons etc.
- Attribute functions : Attributes determine the appearance of objects
  - Color (points, lines, polygons)

- Size and width (points, lines)
- Polygon mode
  - Display as filled
  - Display edges



  - Display vertices

- Viewing functions : Allows us to specify various views by describing the camera's position and orientation.

- Transformation functions : Provides user to carry out transformation of objects like rotation, scaling etc.

- 'Input functions : Allows us to deal with a diverse set of input devices like keyboard, mouse etc

- Control functions : Enables us to initialize our programs, helps in dealing with any errors during execution of the program.

- Query functions : Helps query information about the properties of the particular implementation.

The entire graphics system can be considered as a state machine getting inputs from the application prog.

– inputs may change the state of a machine

– inputs may cause the machine to produce a visible output.

2 types of graphics functions :

– functions defining primitives

– functions that change the state of the machine.

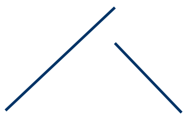
2.3 Primitives and attributes

OpenGL supports 2 types of primitives :

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

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

Line segments



GL_LINES

GL_LINE_STRIP

GL_LINE_LOOP

Polygons :

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

Properties of polygon for it to be rendered correctly :

- Simple – No 2 edges of a polygon cross each other
- Convex – All points on the line segment between any 2 points inside the object, or on its boundary, are inside the object.
- Flat – All the vertices forming the polygon lie in the same plane . E.g. a triangle.

Polygon Issues

- User program can check if above true
  - OpenGL will produce output if these conditions are violated but it may not be what is desired
- Triangles satisfy all conditions

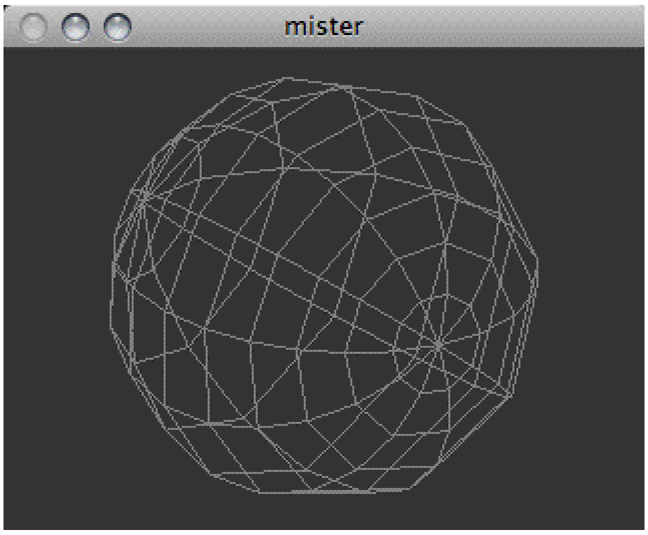
2.4 Approximating a sphere

- Fans and strips allow us to approximate curved surfaces in a simple way.
- E.g. – a unit sphere can be described by the following set of equations :
- $X(\Theta,\Phi)=\sin\Theta\cos\Phi$,
- $Y(\Theta,\Phi)=\cos\Theta\sin\Phi$,
- $Z(\Theta,\Phi)=\sin\Phi$

The sphere shown is constructed using quad strips.

A circle could be approximated using Quad strips.

The poles of the sphere are constructed using triangle fans as can be seen in the diagram



Graphics Text :

A graphics application should also be able to provide textual display.

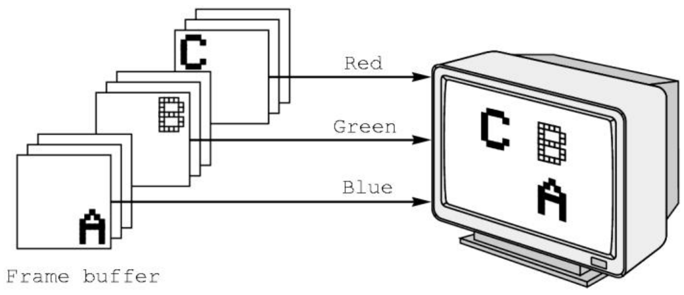
- There are 2 forms of text :
  - Stroke text – Like any other geometric object, vertices are used to define line segments & curves that form the outline of each character.
  - Raster text – Characters are defined as rectangles of bits called **bit blocks.**

bit-block-transfer : the entire block of bits can be moved to the frame buffer using a single function call.

2.5 Color

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

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



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

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

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

glColor

glVertex

glColor

glVertex

RGBA color system :

- This has 4 arguments – RGB and alpha

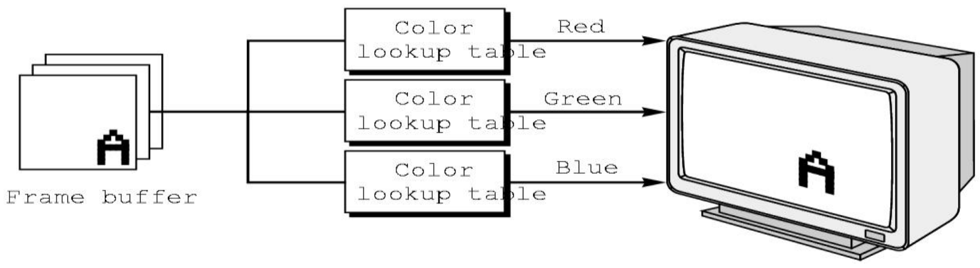
alpha – Opacity.

glClearColor(1.0,1.0,1.0,1.0)

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

Indexed color

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



2.6 Viewing

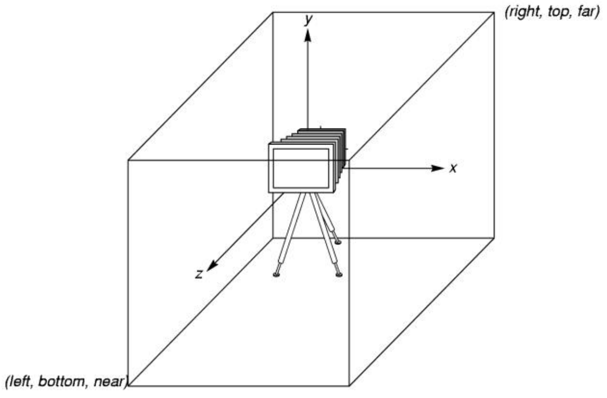
The default viewing conditions in computer image formation are similar to the settings on a basic camera with a fixed lens

The Orthographic view

- Direction of Projection : When image plane is fixed and the camera is moved far from the plane, the projectors become parallel and the COP becomes "direction of projection"

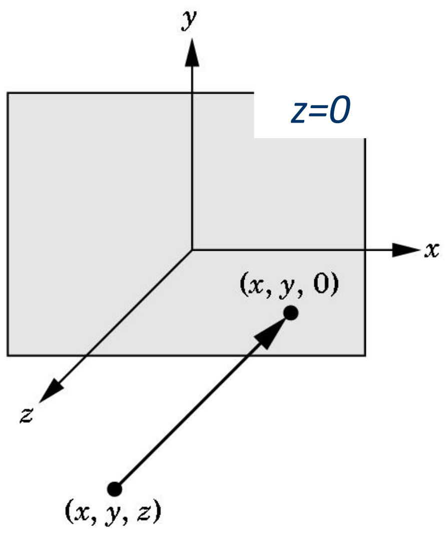
OpenGL Camera

- OpenGL places a camera at the origin in object space pointing in the negative *z* direction
- The default viewing volume is a box centered at the origin with a side of length 2



Orthographic view

In the default orthographic view, points are projected forward along the *z* axis onto theplane



z=0

Transformations and Viewing

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

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

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

glMatrixMode (GL_PROJECTION)

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

glLoadIdentity();

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

<u>2.7 Control Functions (interaction with windows)</u>

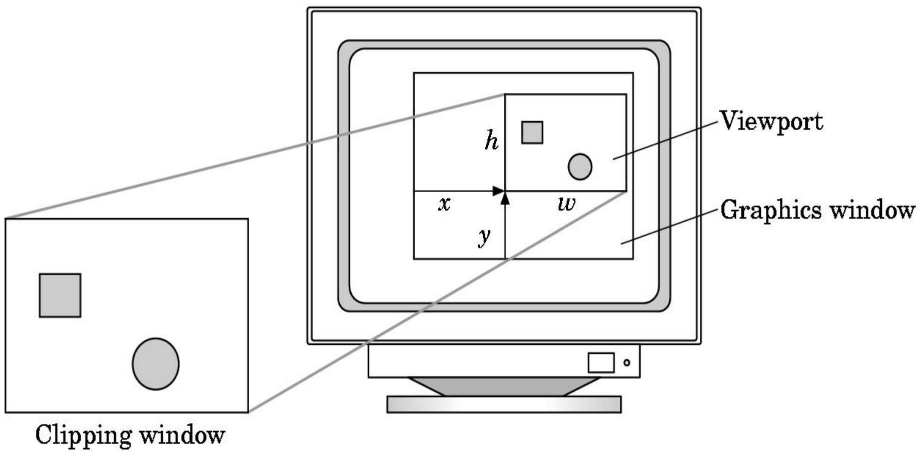
- Window – A rectangular area of our display.
- Modern systems allow many windows to be displayed on the screen (multiwindow environment).
- The position of the window is with reference to the origin. The origin (0,0) is the top left corner of the screen.
- **glutInit** allows application to get command line arguments and initializes system
- **gluInitDisplayMode** requests properties for the window (the *rendering context*)
    - RGB color
    - Single buffering
    - Properties logically ORed together
- **glutWindowSize** in pixels
- **glutWindowPosition** from top-left corner of display
- **glutCreateWindow** create window with a particular title

Aspect ratio and viewports

- Aspect ratio is the ratio of width to height of a particular object.
- We may obtain undesirable output if the aspect ratio of the viewing rectangle (specified by glOrtho), is not same as the aspect ratio of the window (specified by glutInitWindowSize)

Viewport – A rectangular area of the display window, whose height and width can be adjusted to match that of the clipping window, to avoid distortion of the images.

void glViewport(Glint x, Glint y, GLsizei w, GLsizei h) ;



The main, display and myinit functions

- In our application, once the primitive is rendered onto the display and the application program ends, the window may disappear from the display.
- Event processing loop :
- void glutMainLoop();
- Graphics is sent to the screen through a function called **display callback.**
- **void glutDisplayFunc(function name)**
- The function myinit() is used to set the OpenGL state variables dealing with viewing and attributes**.**

<u>Control Functions</u>

- **glutInit**(int **argc*, char ***argv*) initializes GLUT and processes any command line arguments (for X, this would be options like -display and -geometry). **glutInit()** should be called before any other GLUT routine.
- **glutInitDisplayMode**(unsigned int *mode*) specifies whether to use an *RGBA* or color-index color model. You can also specify whether you want a single- or double-buffered window. (If you're working in color-index mode, you'll want to load certain colors into the color map; use **glutSetColor()** to do this.)
- **glutInitDisplayMode**(*GLUT_DOUBLE* | *GLUT_RGB* | *GLUT_DEPTH*).
- If you want a window with double buffering, the RGBA color model, and a depth buffer, you might call
- **glutInitWindowPosition**(int *x*, int *y*) specifies the screen location for the upper-left corner of your window

- **glutInitWindowSize**(int *width*, int *size*) specifies the size, in pixels, of your window.
- int **glutCreateWindow**(char **string*) creates a window with an OpenGL context. It returns a unique identifier for the new window. Be warned: Until **glutMainLoop()** is called.

UNIT - 3 **7 Hours**

INPUT AND INTERACTION

Interaction

Input devices

Clients and servers

Display lists

Display lists and modeling

Programming event-driven input

Menus; Picking

A simple CAD program

Building interactive models

Animating interactive programs

Design of interactive programs

Logic operations.

UNIT - 3

INPUT AND INTERACTION

3.1 Interaction

Project Sketchpad :

Ivan Sutherland (MIT 1963) established the basic interactive paradigm that characterizes interactive computer graphics:

- User sees an *object* on the display
- User points to (*picks*) the object with an input device (light pen, mouse, trackball)
- Object changes (moves, rotates, morphs)
- Repeat

3.2 Input devices

- Devices can be described either by
  - Physical properties
    - Mouse
    - Keyboard
    - Trackball
  - Logical Properties
    - What is returned to program via API
      - **A position**
      - **An object identifier**
- Modes
  - How and when input is obtained
    - Request or event

Incremental (Relative) Devices

- Devices such as the data tablet return a position directly to the operating system
- Devices such as the mouse, trackball, and joy stick return incremental inputs (or velocities) to the operating system
  - Must integrate these inputs to obtain an absolute position
    - Rotation of cylinders in mouse
    - Roll of trackball

        - Difficult to obtain absolute position
        - Can get variable sensitivity

Logical Devices

- Consider the C and C++ code
    - C++: **cin >> x;**
    - C: **scanf ("%d", &x);**
- What is the input device?
    - Can't tell from the code
    - Could be keyboard, file, output from another program
- The code provides *logical input*
    - A number (an **int**) is returned to the program regardless of the physical device

Graphical Logical Devices

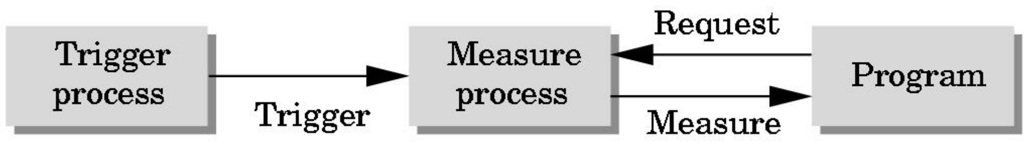
- Graphical input is more varied than input to standard programs which is usually numbers, characters, or bits
- Two older APIs (GKS, PHIGS) defined six types of logical input
    - **Locator**: return a position
    - **Pick**: return ID of an object
    - **Keyboard**: return strings of characters
    - **Stroke**: return array of positions
    - **Valuator**: return floating point number
    - **Choice**: return one of n items

Input Modes

- Input devices contain a *trigger* which can be used to send a signal to the operating system
    - Button on mouse
    - Pressing or releasing a key
- When triggered, input devices return information (their *measure*) to the system
    - Mouse returns position information
    - Keyboard returns ASCII code

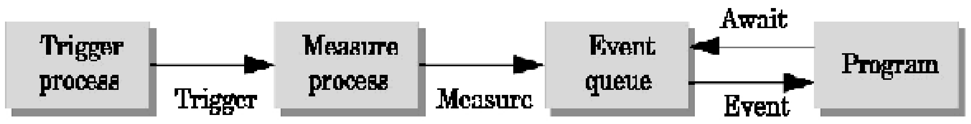
Request Mode

- Input provided to program only when user triggers the device
- Typical of keyboard input
  - Can erase (backspace), edit, correct until enter (return) key (the trigger) is depressed



Event Mode

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



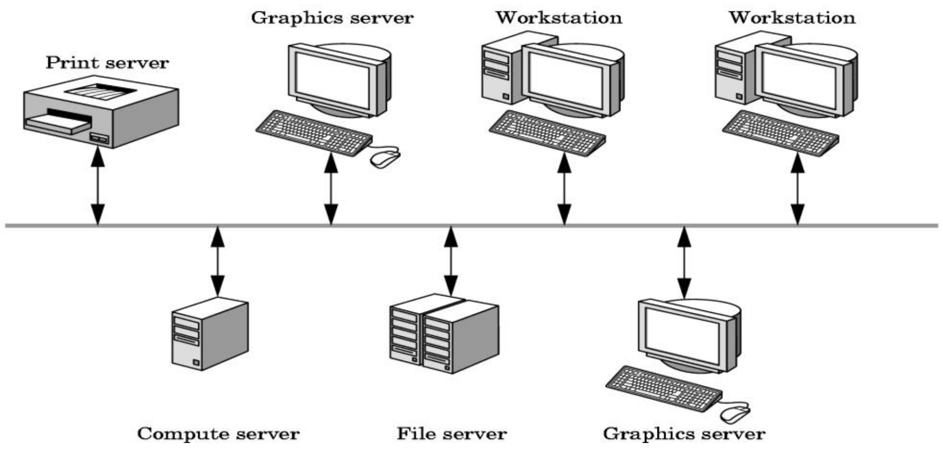
Event Types

- Window: resize, expose, iconify
- Mouse: click one or more buttons
- Motion: move mouse
- Keyboard: press or release a key
- Idle: nonevent
  - Define what should be done if no other event is in queue

3.3 Clients And Servers

- The X Window System introduced a client-server model for a network of workstations
  - **Client**: OpenGL program

– **Graphics Server**: bitmap display with a pointing device and a keyboard



3.4 Display Lists

The Display Processor in modern graphics systems could be considered as a graphics server.

- Retained mode - The host compiles the graphics program and this compiled set is maintained in the server within the display list.
- The redisplay happens by a simple function call issued from the client to the server
- It avoids network clogging
- Avoids executing the commands time and again by the client

- Definition and Execution of display lists:

```
    #define PNT 1
    glNewList(PNT, GL_COMPILE);
    glBegin(GL_POINTS);
            glVertex2f(1.0,1.0);
    glEnd();
glEndList();
```

- GL_COMPILE – Tells the system to send the list to the server but not to display the contents
- GL_COMPILE_AND_EXECUTE – Immediate display of the contents while the list is being constructed.

- Each time the point is to be displayed on the server, the function is executed.
- glCallList(PNT);
  glCallLists function executes multiple lists with a single function call

Text and Display Lists

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

```
void OurFont(char c)
{
switch(c)
{
        case ‘O’ :
        glTranslatef(0.5,0.5,0.0); /* move to the center */
        glBegin(GL_QUAD_STRIP)
                for (i=0;i<12;i++) /* 12 vertices */
                {
                        angle = 3.14159/6.0 * i; /* 30 degrees in radians */
                        glVertex2f(0.4 * cos(angle)+0.5, 0.4 * sin(angle)+0.5)
                        glVertex2f(0.5 * cos(angle)+0.5, 0.5 * sin(angle)+0.5)
                }
        glEnd();
break;
}


}
```

Fonts in GLUT

- GLUT provides a few raster and stroke fonts

- Function call for stroke text :

glutStrokeCharacter(GLUT_STROKE_MONO_ROMAN, int character)

- Function call for bitmap text :

glutBitmapCharacter(GLUT_BITMAP_8_BY_13, int character)

3.5 Display Lists And Modeling

- Building hierarchical models involves incorporating relationships between various parts of a model

```
        #define EYE 1                          glTranslatef(……);
                                                   glCallList(EYE);
            glNewList(EYE);
             /* code to draw eye */
            glEndList();

            # define FACE 2
            glNewList(FACE);
            /* Draw outline */
            glTranslatef(…..)
            glCallList(EYE);
```

3.6 Programing Event Driven Input

- Pointing Devices :

A mouse event occurs when one of the buttons of the mouse is pressed or released

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

The callback from the main function would be :

glutMouseFunc(myMouse);

Window Events

- Most windows system allows user to resize window.
- This is a window event and it poses several problems like
  - Do we redraw all the images
  - The aspect ratio
  - Do we change the size or attributes of the primitives to suit the new window

```
void myReshape(GLsizei w, GLsizei h)
{
        /* first adjust clipping box */
    glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluOrtho2D(0.0,(GLdouble)w, 0.0, (GLdouble)h);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();

        /* adjust viewport */
        glViewport(0,0,w,h);
}
```

Keyboard Events

When a keyboard event occurs, the ASCII code for the key that generated the event and the mouse location are returned.

E.g.

```
   void myKey(unsigned char key, int x, int y)
        {
        if (key=='q' || key=='Q')
        exit(0);
        }
```

Callback : glutKeyboardFunc(myKey);

- GLUT provides the function **glutGetModifiers** function enables us to define functionalities for the meta keys

The Display and Idle callbacks

Interactive and animation programs might contain many calls for the reexecution of the display function.

- glutPostRedisplay() – Calling this function sets a flag inside GLUT's main loop indicating that the display needs to be redrawn.
- At the end of each execution of the main loop, GLUT uses this flag to determine if the display function will be executed.
- The function ensures that the display will be drawn only once each time the program goes through the event loop.
- Idle Callback is invoked when there are no other events to be performed.
- Its typical use is to continuously generate graphical primitives when nothing else is happening.
- Idle callback : glutIdleFunc(function name)

Window Management

- GLUT supports creation of multiple windows
- Id = glutCreateWindow("second window");

- To set a particular window as the current window where the image has to be rendered glutSetWindow(id);

<u>3.7 Menus</u>

- GLUT supports pop-up menus
  - A menu can have submenus
- Three steps
  - Define entries for the menu
  - Define action for each menu item
    - Action carried out if entry selected
  - Attach menu to a mouse button

Defining a simple menu

```
menu_id = glutCreateMenu(mymenu);
glutAddmenuEntry("clear Screen", 1);
gluAddMenuEntry("exit", 2);
glutAttachMenu(GLUT_RIGHT_BUTTON);
```

Menu callback

```
void mymenu(int id)
{
        if(id == 1) glClear();
        if(id == 2) exit(0);
}
```

– Note each menu has an id that is returned when it is created

Add submenus by

glutAddSubMenu(char *submenu_name, submenu id)

3.8 Picking

- Identify a user-defined object on the display
- In principle, it should be simple because the mouse gives the position and we should be able to determine to which object(s) a position corresponds
- Practical difficulties
    - Pipeline architecture is feed forward, hard to go from screen back to world
    - Complicated by screen being 2D, world is 3D
    - How close do we have to come to object to say we selected it?

Rendering Modes

- OpenGL can render in one of three modes selected by glRenderMode(mode)
    - GL_RENDER: normal rendering to the frame buffer (default)
    - GL_FEEDBACK: provides list of primitives rendered but no output to the frame buffer

  - GL_SELECTION: Each primitive in the view volume generates a *hit record* that is placed in a *name stack* which can be examined later
  -

Selection Mode Functions

- glSelectBuffer(GLsizei n, GLuint *buff): specifies name buffer
- glInitNames(): initializes name buffer
- glPushName(GLuint name): push id on name buffer
- glPopName(): pop top of name buffer
- glLoadName(GLuint name): replace top name on buffer

- id is set by application program to identify objects

Using Selection Mode

- Initialize name buffer
- Enter selection mode (using mouse)
- Render scene with user-defined identifiers
- Reenter normal render mode
  - This operation returns number of hits
- Examine contents of name buffer (hit records)
  - Hit records include id and depth information

Selection Mode and Picking

- As we just described it, selection mode won’t work for picking because every primitive in the view volume will generate a hit
- Change the viewing parameters so that only those primitives near the cursor are in the altered view volume
  - Use gluPickMatrix (see text for details)

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

        /* set up viewing for selection mode */

        glGetIntegerv(GL_VIEWPORT, viewport); //gets the current viewport
        glMatrixMode(GL_PROJECTION);

        /* save original viewing matrix */

        glPushMatrix();
        glLoadIdentity();

        /* N X N pick area around cursor */
        gluPickMatrix( (GLdouble) x,(GLdouble)(viewport[3]-y),N,N,viewport);

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

        draw_objects(GL_SELECT);
        glMatrixMode(GL_PROJECTION);

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

        /* return back to normal render mode */

        hits = glRenderMode(GL_RENDER);
```

```
                /* process hits from selection mode rendering*/

                processHits(hits, nameBuff);

                /* normal render */
                glutPostRedisplay();
        }
}

void draw_objects(GLenum mode)
{
        if (mode == GL_SELECT)
                glLoadName(1);
        glColor3f(1.0,0.0,0.0)
        glRectf(-0.5,-0.5,1.0,1.0);

        if (mode == GL_SELECT)
                        glLoadName(2);
        glColor3f(0.0,0.0,1.0)
        glRectf(-1.0,-1.0,0.5,0.5);
}

void processHits(GLint hits, GLUint buffer[])
{
        unsigned int i,j;

}
```

3.9 Building Interactive Programs

- Building blocks : equilateral triangle, square, horizontal and vertical line segments.

```
typedef struct object{
int type;
float x,y;
float color[3];
} object;
```

Define array of 100 objects & index to last object in the list.

```
object table[100];
int last_object;
```

Entering info into the object:

```
table[last_object].type = SQUARE;
table[last_object].x = x0;
table[last_object].y = y0;
table[last_object].color[0] = red;
…..
last_object ++;
```

- To display all the objects, the code looks like this:

```
for (i=0;i<last_object;i++)
{
        switch(table[i].type)
        {
                case 0: break;
                case 1:
                {
                        glColor3fv(table[i].color);
                        triangle(table[i].x,table[i].y);
                        break;
                }
        …..
}
```

- In order to add code for deleting an object, we include some extra information in the object structure:

```
float bb[2][2];

bb[0][0] = x0-1.0;
```

bb[0][1] = y0-1.0;….

3.10 Animating interactive programs

The points x=cos θ, y=sin θ always lies on a unit circle regardless of the value of θ.

- In order to increase θ by a fixed amount whenever nothing is happening, we use the idle function

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

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

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

Double Buffering

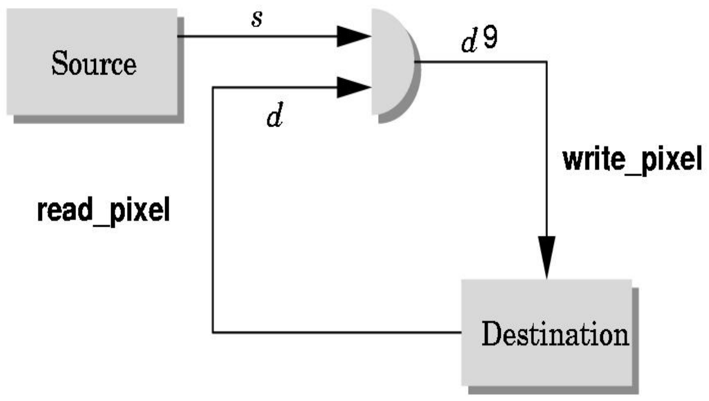
- We have 2 color buffers for our disposal called the Front and the Back buffers.
- Front buffer is the one which is always displayed.
- Back buffer is the one on which we draw
- Function call to swap buffers :
- glutSwapBuffers();
- By default openGl writes on to the back buffer.
- But this can be controlled using

glDrawBuffer(GL_BACK);

glDrawBuffer(FRONT_AND_BACK);

Writing Modes

XOR write



- Usual (default) mode: source replaces destination (d' = s)
    - Cannot write temporary lines this way because we cannot recover what was "under" the line in a fast simple way
- Exclusive OR mode (XOR) (d' = d ⊕ s)
    - x ⊕ y ⊕ x =y
    - Hence, if we use XOR mode to write a line, we can draw it a second time and line is erased!

Rubberbanding

- Switch to XOR write mode
- Draw object
    - For line can use first mouse click to fix one endpoint and then use motion callback to continuously update the second endpoint
    - Each time mouse is moved, redraw line which erases it and then draw line from fixed first position to to new second position
    - At end, switch back to normal drawing mode and draw line
    - Works for other objects: rectangles, circles

XOR in OpenGL

- There are 16 possible logical operations between two bits

- All are supported by OpenGL
    - Must first enable logical operations
        - **glEnable(GL_COLOR_LOGIC_OP)**
    - Choose logical operation
        - **glLogicOp(GL_XOR)**
        - **glLogicOp(GL_COPY)** (default)

UNIT - 4 **6 Hrs**

GEOMETRIC OBJECTS AND TRANSFORMATIONS –I

Scalars

Points, and vectors

Three-dimensional primitives

Coordinate systems and frames

Modelling a colored cube

Affine transformations

Rotation, translation and scaling

UNIT - 4 6 Hrs

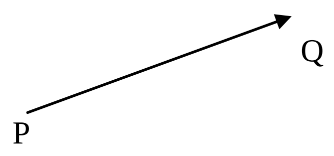
GEOMETRIC OBJECTS AND TRANSFORMATIONS – I

4.1 Scalars, points and vectors

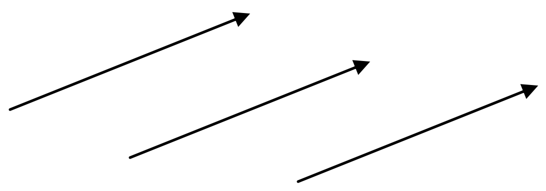
The basic geometric objects and relationship among them can be described using the three fundamental types called scalars, points and vectors.

Geometric Objects.

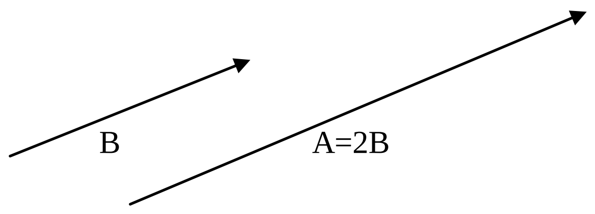
- Points:
  One of the fundamental geometric objects is a point.
  - In 3D geometric system, point is a location in space. Point possesses only the location property, mathematically point neither a size nor a shape.
  - Points are useful in specifying objects but not sufficient.
- Scalars:
  - Scalars are objects that obey a set of rules that are abstraction of the operations of ordinary arithmetic.
  - Thus, addition and multiplication are defined and obey the usual rules such as commutativity and associativity and also every scalar has multiplicative and additive inverses.
- Vector:
  - Another basic object which has both direction and magnitude, however, vector does not have a fixed location in space.
  - Directed line segment shown in figure below connects two points has both direction i.e, orientation and magnitude i.e., its length so it is called as a vector



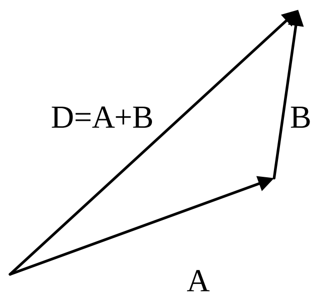
because of vectors does not have fixed position, the directed line segments shown in figure below are identical because they have the same direction and magnitude.



Vector lengths can be altered by the scalar components, so the line segment A shown in figure below is twice t he length of line segment B

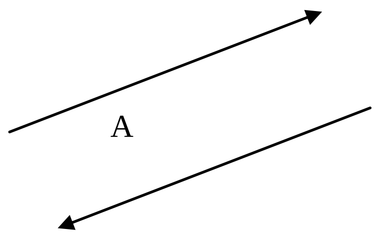


We can also combine directed line segments as shown in figure below by using the head and tail rule



We obtained new vector D from two vectors A and B by connecting head of A to tail of B. Magnitude and direction of vector D is determined from the tail of A to the head of B, we can call D has sum of A and B, so we can write it as D=A+B.

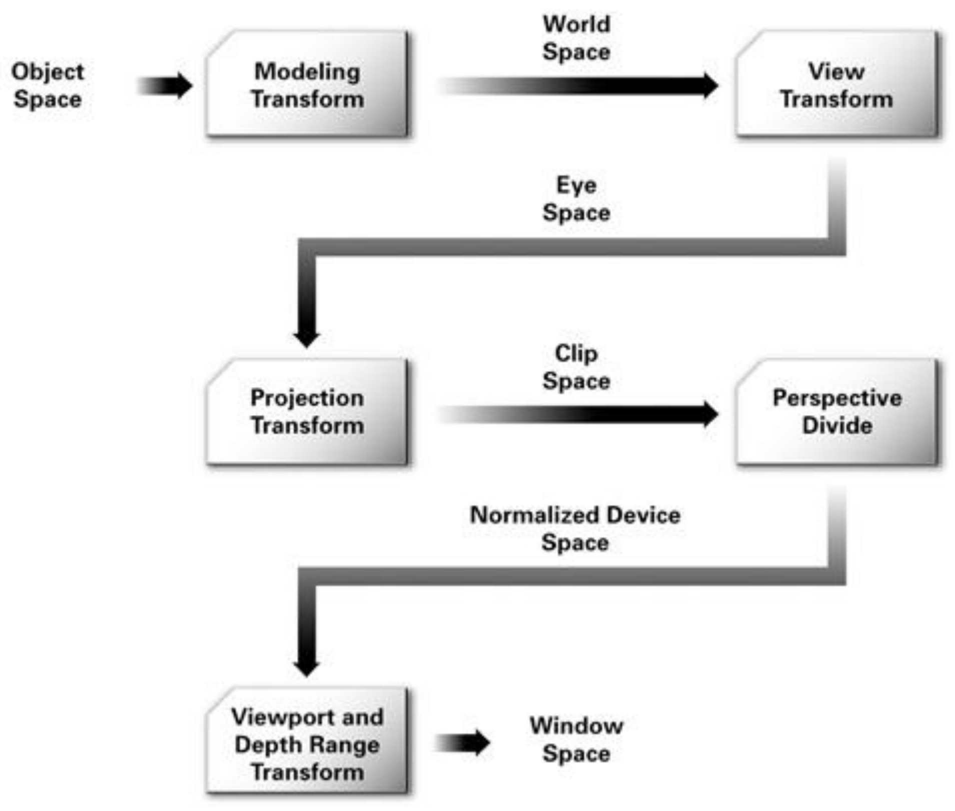
Consider the two directed line segments A and E shown in figure below with the same length but opposite direction. We can define the vector E in terms of A as E =-A, so the vector E is called **inverse vector** of A. The sum of vectors A and E is called **Zero vector**, which is denoted as **0,** that has a zero magnitude and orientation is undefined.



4.2 Coordinate systems and frames

- The purpose of the graphics pipeline is to create images and display them on your screen. The graphics pipeline takes geometric data representing an object or scene (typically in three dimensions) and creates a two-dimensional image from it.
- Your application supplies the geometric data as a collection of vertices that form polygons, lines, and points.
- The resulting image typically represents what an observer or camera would see from a particular vantage point.
- As the geometric data flows through the pipeline, the GPU's vertex processor transforms the constituent vertices into one or more different coordinate systems, each of which serves a particular purpose. Cg vertex programs provide a way for you to program these transformations yourself.

Figure 4-1 illustrates the conventional arrangement of transforms used to process vertex positions. The diagram annotates the transitions between each transform with the coordinate space used for vertex positions as the positions pass from one transform to the next.



4.3 Modeling a colored cube

Modeling the faces

The case is as simple 3D object. There are number of ways to model it. CSG systems regard it as a single primitive. Another way is, case as an object defined by eight vertices. We start modeling the cube by assuming that vertices of the case are available through an array of vertices i.e,

```
        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 also use object oriented form by 3D point type as follows

```
        Typedef GLfloat point3 [3];
```

The vertices of the cube can be defined as follows

Point3 Vertices [8] =

{{-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 the list of points to specify the faces of the cube. For example one face is

```
        glBegin (GL_POLYGON);

                glVertex3fv (vertices [0]);

                glVertex3fv (vertices [3]);

                glVertex3fv (vertices [2]);

                glVertex3fv (vertices [1]);

        glEnd ();
```

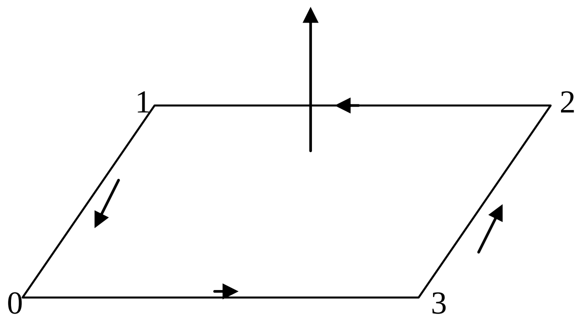
Similarly we can define other five faces of the cube.

Inward and outward pointing faces

When we are defining the 3D polygon, we have to be careful about the order in which we specify the vertices, because each polygon has two sides. Graphics system can display either or both of them. From the camera's perspective, we need to distinguish between the two faces of a polygon. The order in which the vertices are specified provides this information. In the above example we used the order 0,3,2,1 for the first face. The order 1,0,2,3 would be same because the final vertex in polygon definition is always linked back to the first, but the order 0,1,2,3 is different.

We call face outward facing, if the vertices are traversed in a counter clockwise order, when the face is viewed from the outside.

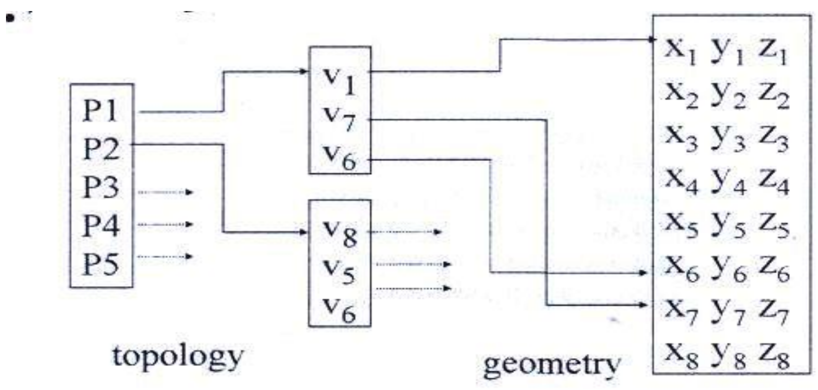
In our example, the order 0,3,2,1 specifies outward face of the cube. Whereas the order 0,1,2,3 specifies the back face of the same polygon.



By specifying front and back carefully, we will can eliminate faces that are not visible. OpenGL can treat inward and outward facing polygons differently.

Data structures for object representation

It is better to use the data structure that separate the geometry from the topology. [The geometry stands for locations of the vertices and topology stands for organization of the vertices and edges]. We use a structure, the vertex list (shown in Fig. below) that is both simple and useful.



The data specifying the location of the vertices contain the geometry and can be stored as a simple list or array, such as in vertices the vertex list. The top level entity is a cube and is composed of six faces. Each face consists of four ordered vertices. Each vertex can be specified indirectly through its index. This can be represented like figure shown above..

The color cube

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

```
GLfloatVertices [8] [3] = {{-1.0, -1.0, -1.0}, {1.0, -1.0, -1.0}, {1.0, 1.0, -1.0}, {-1.0, 1.0, -1.0}
{-1.0, -1.0, 1.0}, {1.0, -1.0, 1.0}, {1.0, 1.0, 1.0}, {-1.0, 1.0, 1.0}}

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

void quad (int a, int b, int c, int d)

{

        glBegin (GL_QUADS);

                glcolor3fv (colors[a]);

                glVertex3fv(vertices[a]);

                glcolor3fv(colors[b]);

                glVertex3fv(vertices[b]);

                glcolor3fv(colors[c]);

                glVertex3fv (vertices[c]);

                glcolor3fv (colors[d]);

                glVertex3fv(vertices[d]);

        glEnd();

}
```

```
Void colorcube ()

{

        quad (0,3,2,1);

        quad (2,3,7,6);

        quad (0, 4,7,3);

        quad (1, 2, 6, 5);

        quad (4, 5, 6, 7);

        quad (0, 1, 5, 4);
```

Vertex arrays

Although we used vertex lists to model the cube, it requires many openGL function calls. For example, the above function make 60 openGL calls: six faces, each of which needs a glBegin, a glEnd, four calls to glColor, and four calls to glVertex. Each of which involves overhead & data transfer. This problem can be solved by using vertex arrays.

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

There are three steps in using vertex arrays

(i) Enable the functionality of vertex arrays

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

(iii) Render the object.

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

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

```
glEnableClientstate (GL_COLOR_ARRAY)

glEnableClientstate (GL_VERTEX_ARRAY).
```

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

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

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

The first three arguments state that the elements are 3D colors and vertices stored as floats & that the elements are contagious in the arrays. Fourth argument is pointer to the array holding the data. Next, provide information in data structure about the relationship between the vertices the faces of the cube by specifying an array that holds the 24 ordered vertex indices for the six faces.

```
GLubytecubeIndices [24] = {0,3,2,1,2,3,7,6,0,4,7,3,1,2,6,5,4,5,6,7,0,1,5,4};
```

Each successive four indices describe a face of the cube, Draw the array through glDrawElements which replaces all glVertex & glcolor calls in the display.

glDrawElements(GLenum type, GLsizei n, GLenum format, void * pointer)

We can draw the cube using two methods

```
1) for (i=0;,i<6;i++)
     glDrawElements(GL_POLYGON,4,GL_UNSIGNED_BYTE, & cubeIndices[4*i]);

2) glDrawElements(GL_QUADS, 24, GL_UNSIGNED_BYTE, cubeIndices);
```

<u>4.4 Affine transformations</u>

An affine transformation is an important class of linear 2-D geometric transformations which maps variables (*e.g.* pixel intensity values located at position (x_1, y_1) in an input image) into new variables (*e.g.* (x_2, y_2) in an output image) by applying a linear combination of translation, rotation, scaling and/or shearing (*i.e.* non-uniform scaling in some directions) operations.

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

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

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

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

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

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

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

Similarly, pure scaling is:

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

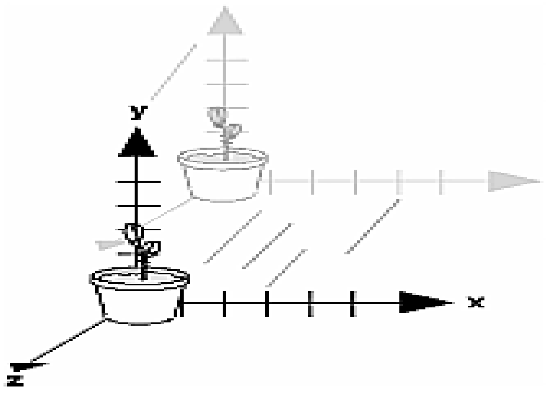
(Note that several different affine transformations are often combined to produce a resultant transformation. The order in which the transformations occur is significant since a translation followed by a rotation is not necessarily equivalent to the converse.)

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

4.5 Rotation, translation and scaling

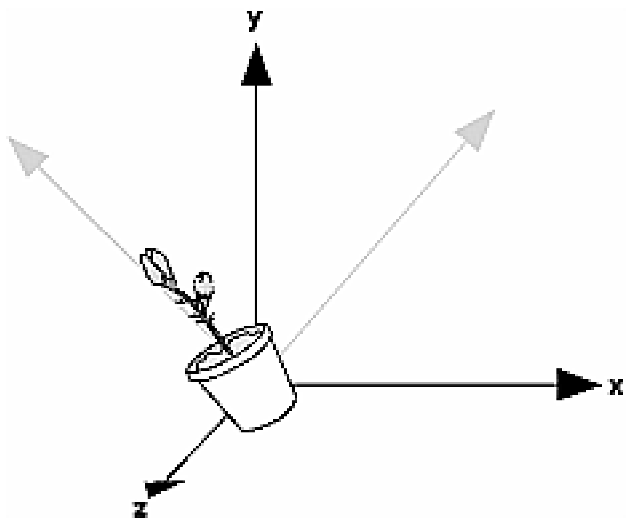
Translation
void glTranslate{fd} (TYPE x, TYPE y, TYPE z);

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



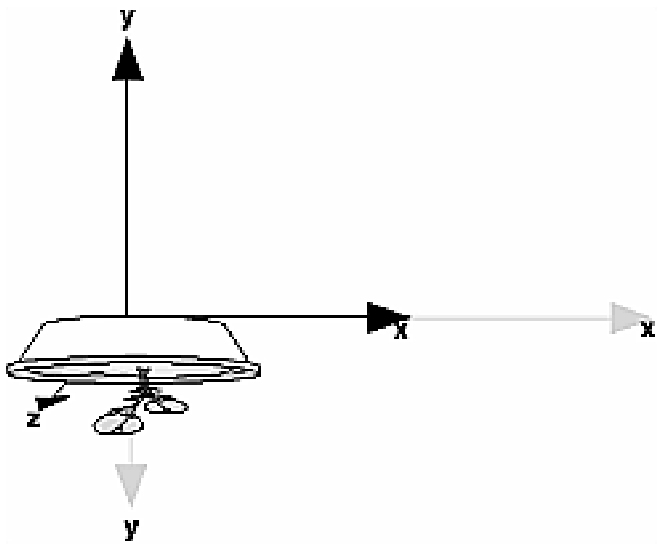
Rotation

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



Scaling

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



Equations :

- Translation: Pf = T + P

$x_f = x_o + dx$

$y_f = y_o + dy$

- Rotation: Pf = R · P

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

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

- Scale: Pf = S · P

$x_f = sx * x_o$

$y_f = sy * y_o$

PART - B

UNIT - 5 **5 Hrs**

GEOMETRIC OBJECTS AND TRANSFORMATIONS – II

Transformations in homogeneous coordinates

Concatenation of transformations

OpenGL transformation matrices

Interfaces to three-dimensional applications

Quaternions.

UNIT - 5 5 Hrs

GEOMETRIC OBJECTS AND TRANSFORMATIONS – II

5.1 Transformations in homogeneous coordinates

More generally, we consider the <*x*, *y*, *z*> position vector to be merely a special case of the four-component <*x*, *y*, *z*, *w*> form. This type of four-component position vector is called a *homogeneous position*. When we express a vector position as an <*x*, *y*, *z*> quantity, we assume that there is an implicit 1 for its *w* component.

Mathematically, the *w* value is the value by which you would divide the *x*, *y*, and *z* components to obtain the conventional 3D (nonhomogeneous) position, as shown in Equation 4-1.

Equation 4-1 Converting Between Nonhomogeneous and Homogeneous Positions

$$\left\langle \frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1 \right\rangle = \langle x, y, z, w \rangle$$

Expressing positions in this homogeneous form has many advantages.

- For one, multiple transformations, including projective transformations required for perspective 3D views, can be combined efficiently into a single 4x4 matrix.
- Also, using homogeneous positions makes it unnecessary to perform expensive intermediate divisions and to create special cases involving perspective views.
- Homogeneous positions are also handy for representing directions and curved surfaces described by rational polynomials.

```
1    0    dx         x
0    1    dy   ·y
0    0     1         1

1 * x        +      0 * y  +      dx * 1
0 * x        +      1 * y  +      dy * 1
0 * x        +      0 * y  +      1 * 1
```

Concatenation of transformations

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

– translate so that the corner returns to its original position

All these operations could be carried out at once by multiplying the corresponding matrices and obtaining one single matrix which would then be multiplied with the projection matrix of the object to obtain the final result.

World Space

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

The Modeling Transform

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

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

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

Figure 4-2 illustrates the effect of several different modeling transformations. The left side of the figure shows a robot modeled in a basic pose with no modeling transformations applied. The right side shows what happens to the robot after you apply a series of modeling transformations to its various body parts. For example, you must rotate and translate the right arm to position it as shown. Further transformations may be required to translate and rotate the newly posed robot into the proper position and orientation in world space.

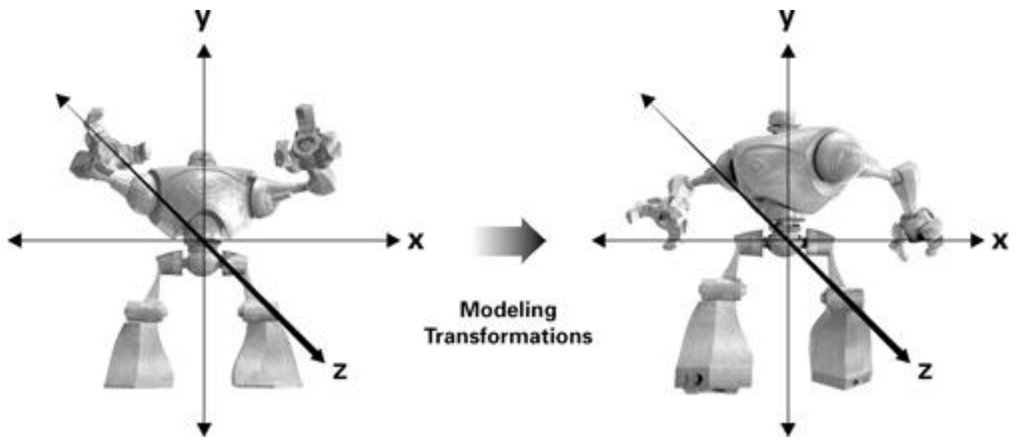


Figure 4-2 The Effect of Modeling Transformations

Eye Space

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

The View Transform

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

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

Figure 4-3 illustrates the view transform. The left side of the figure shows the robot from Figure 4-2 along with the eye, which is positioned at <0, 0, 5> in the world-space coordinate system. The right side shows them in eye space. Observe that eye space positions the origin at the eye. In this example, the view transform translates the robot in order to move it to the correct position in eye space. After the translation, the robot ends up at <0, 0, -5> in eye space, while the eye is at the origin. In this example, eye space and world space share the positive *y*-axis as their "up" direction and the translation is purely in the *z* direction. Otherwise, a rotation might be required as well as a translation.

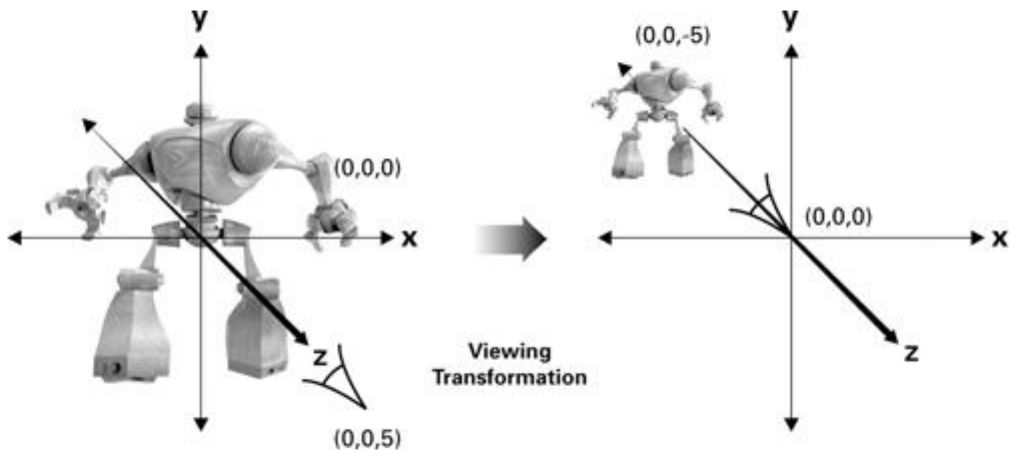


Figure 4-3 The Effect of the Viewing Transformation

The Modelview Matrix

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

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

Clip Space

Once positions are in eye space, the next step is to determine what positions are actually viewable in the image you eventually intend to render. The coordinate system subsequent to eye space is known as *clip space*, and coordinates in this space are called*clip coordinates*.

The vertex position that a Cg vertex program outputs is in clip space. Every vertex program optionally outputs parameters such as texture coordinates and colors, but a vertex program *always* outputs a clip-space position. As you have seen in earlier examples, the **POSITION** semantic is used to indicate that a particular vertex program output is the clip-space position.

The Projection Transform

The transform that converts eye-space coordinates into clip-space coordinates is known as the *projection transform*.

The projection transform defines a *view frustum* that represents the region of eye space where objects are viewable. Only polygons, lines, and points that are within the view frustum are potentially viewable when rasterized into an image. OpenGL and Direct3D have slightly different rules for clip space. In OpenGL, everything that is viewable must be within an axis-aligned cube such that the *x, y,* and *z* components of its clip-space position are less than or equal to its corresponding *w* component. This implies that $-w \leq x \leq w$, $-w \leq y \leq w$, and $-w \leq z \leq w$. Direct3D has the same clipping requirement for *x* and *y*, but the *z*requirement is $0 \leq z \leq w$. These clipping rules assume that the clip-space position is in homogeneous form, because they rely on *w*.

The projection transform provides the mapping to this clip-space axis-aligned cube containing the viewable region of clip space from the viewable region of eye space—otherwise known as the view frustum. You can express this mapping as a 4x4 matrix.

<u>The Projection Matrix</u>

The 4x4 matrix that corresponds to the projection transform is known as the *projection matrix*.

Figure 4-4 illustrates how the projection matrix transforms the robot in eye space from Figure 4-3 into clip space. The entire robot fits into clip space, so the resulting image should picture the robot without any portion of the robot being clipped.

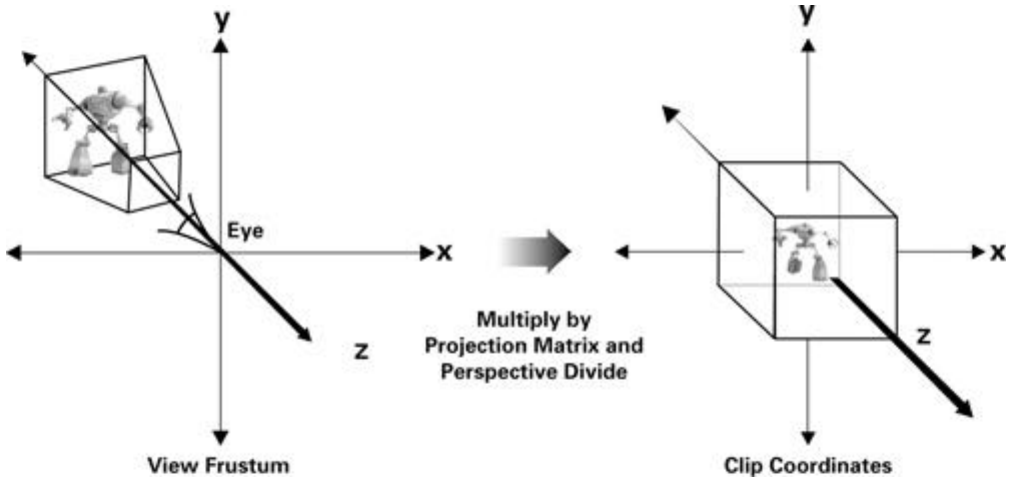


Figure 4-4 The Effect of the Projection Matrix

The clip-space rules are different for OpenGL and Direct3D and are built into the projection matrix for each respective API. As a result, if Cg programmers rely on the appropriate projection matrix for their choice of 3D programming interface, the distinction between the two clip-space definitions is not apparent. Typically, the application is responsible for providing the appropriate projection matrix to Cg programs.

UNIT - 6 **7 hrs**

VIEWING

Classical and computer viewing

Viewing with a computer

Positioning of the camera

Simple projections

Projections in OpenGL

Hidden-surface removal

Interactive mesh displays

Parallel-projection matrices

Perspective-projection matrices

Projections and shadows.

UNIT - 6 **7 hrs**

VIEWING

6.1 Classical Viewing

- 3 basic elements for viewing :
  - One or more objects
  - A viewer with a projection surface
  - Projectors that go from the object(s) to the projection surface

Classical views are based on the relationship among these elements

- Each object is assumed to constructed from flat *principal faces*
  - Buildings, polyhedra, manufactured objects
  - Front, top and side views.

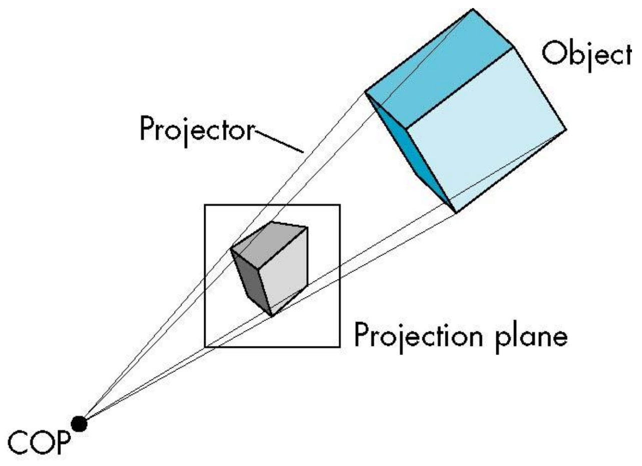
Planar Geometric Projections

- Projections : Standard projections project onto a plane
- Projectors : Lines that either converge at a center of projection or, are parallel (DOP)
- Nonplanar projections are needed for applications such as map construction

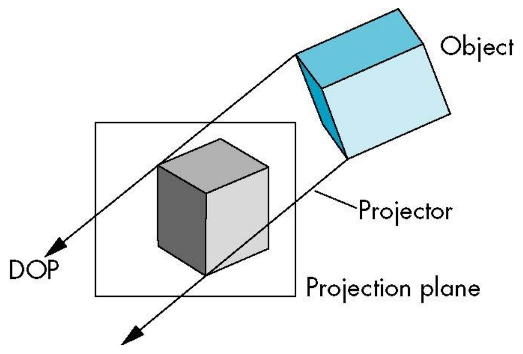
Perspective and parallel projections :

Parallel viewing is a limiting case of perspective viewing

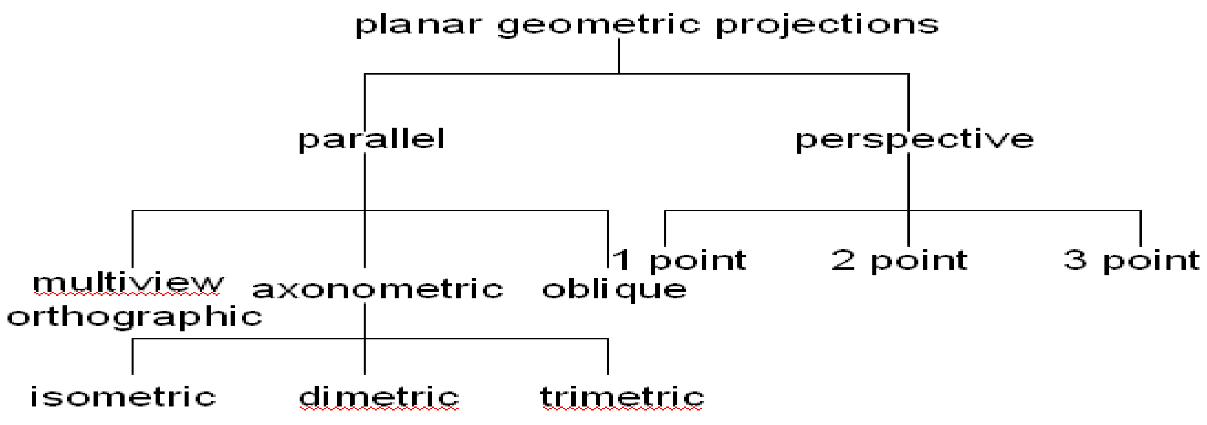
Perspective projection has a COP where all the projector lines converge.



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

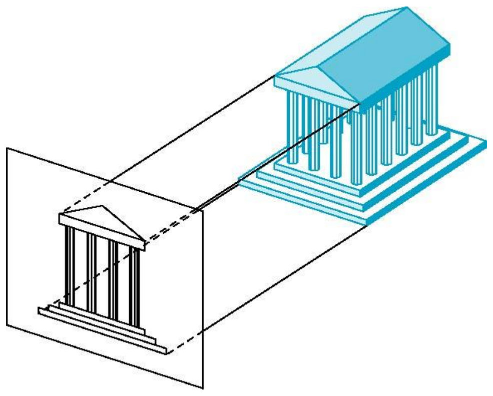


<u>Types Of Planar Geometric Projections :</u>



<u>Orthographic Projections :</u>

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



Advantages and Disadvantages :

- Preserves both distances and angles
    - Shapes preserved
    - Can be used for measurements
        - Building plans
        - Manuals
- Cannot see what object really looks like because many surfaces hidden from view
    - Often we add the isometric

Axonometric Projections

- Projectors are orthogonal to the projection plane , but projection plane can move relative to object.
- Classification by how many angles of a corner of a projected cube are the same
    - none: trimetric
    - two: dimetric
    - three: isometric

Advantages and Disadvantages :

- Lines are scaled (*foreshortened*) but can find scaling factors
- Lines preserved but angles are not
- Projection of a circle in a plane not parallel to the projection plane is an ellipse
- Can see three principal faces of a box-like object
- Some optical illusions possible
- Parallel lines appear to diverge
- Does not look real because far objects are scaled the same as near objects
- Used in CAD applications

Oblique Projections

Arbitrary relationship between projectors and projection plane

Perspective Viewing

Characterized by diminution of size i.e. when the objects move farther from the viewer it appears smaller.

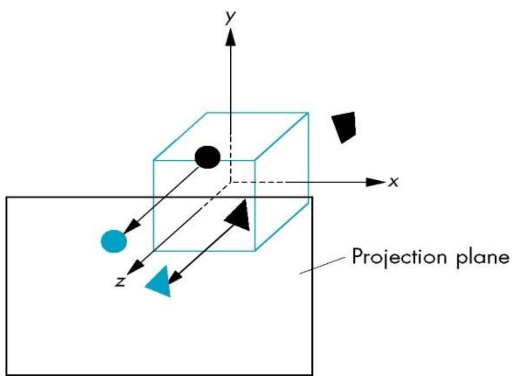
Major use is in architecture and animation.

(imp) Viewing with a Computer

- There are three aspects of the viewing process, all of which are implemented in the pipeline,
  - Positioning the camera
    - Setting the model-view matrix
  - Selecting a lens
    - Setting the projection matrix
  - Clipping
    - Setting the view volume

The OpenGL Camera

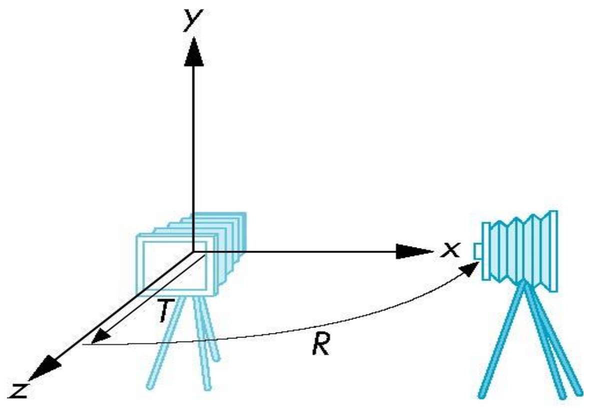
- In OpenGL, initially the object and camera frames are the same
  - Default model-view matrix is an identity
- The camera is located at origin and points in the negative z direction
- OpenGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin
  - Default projection matrix is an identity



Positioning of the Camera.

- If we want to visualize object with both positive and negative z values we can either
  - Move the camera in the positive z direction

        - Translate the camera frame
    - Move the objects in the negative z direction
        - Translate the world frame
- Both of these views are equivalent and are determined by the model-view matrix
    - Want a translation (glTranslatef(0.0,0.0,-d);)
    - d > 0
- We can move the camera to any desired position by a sequence of rotations and translations
- Example: to position the camera to obtain a side view :
    - Rotate the camera
    - Move it away from origin
    - Model-view matrix C = TR



E.g. 1 : Code to view an object present at the origin from a positive x axis

First the camera should be moved away from the object

Secondly the camera should be rotated about the y axis by 90^0.

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

- Consider that we would like to get an isometric view of a cube centered at origin.
- Consider the camera is placed somewhere along the positive z axis.

- An isometric view could be obtained by :
    - Rotate the cube about x-axis till we see the 2 faces symmetrically(45 degrees).
    - Rotate cube about y axis to get the desired isometric view (-35.26 degrees)

- The final model-view matrix is given by
    - M = TRxRy

R = RxRy

Rx is of the form :

$$\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

[NOTE : The default matrix for homogeneous coordinate, right handed, 3D system is given by :

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

Refer chap 4 (transformations)]

- Ry is of the form :

$$\begin{matrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

Translation is a simple addition of the matrices.

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

The openGL function to set camera orientation :

The Look At function : glLookAt(eyex, eyey, eyez, atx, aty, atz, upx, upy, upz)

Where : eyex, eyey, eyez – coordinates of the COP

atx, aty, atz - coordinates of the point at which the camera is pointing at.

upx, upy, upz - The up direction

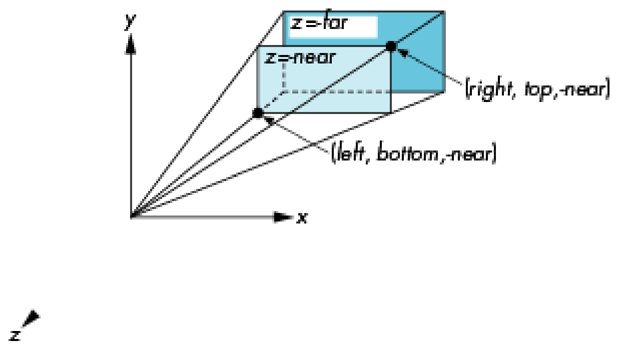
<u>OpenGL Perspective</u>

In case of orthographic projections, the default view volume is a parallelpiped. This is because the projector lines are parallel and they do not converge at any point(theoritically they converge at infinity)

In case of perspective projections, the projector lines converge at the COP. Hence the view volume would be a frustum rather than a parellelpiped.
The frustum (part of a pyramid) as shown in the diagram has a near and a far plane. The objects within this would be within the view volume and visible to the viewer.

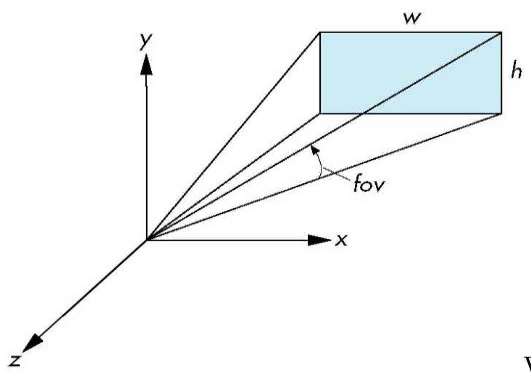
OpenGL provides a function to define the frustum.

glFrustum(left,right,bottom,top,near,far)



<u>Using Field of View</u>

- With **glFrustum** it is often difficult to get the desired view
- **gluPerpective(fov, aspect, near, far)** often provides a better interface



where aspect = w/h

Fov = field of view (The area that the lens would cover is determined by the angle shown in the diagram)

<u>Hidden surface removal</u>

A graphics system passes all the faces of a 3d object down the graphics pipeline to generate the image. But the viewr might not be able to view all these phases. For e.g . all the 6 faces of a cube might not be visible to a viewer.Thus the graphics system must be careful as to which surfaces it has to display.

Hidden surface – removal algortithms are those that remove the surfaces of the image that should not be visible to the viewer.

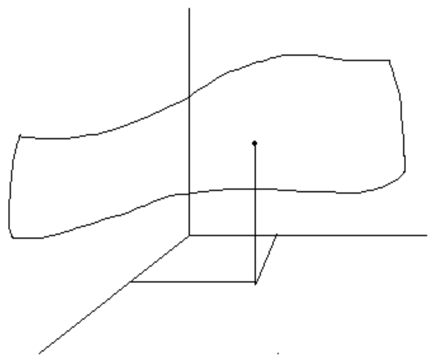
2 types:

- Object Space Algorithm : Orders the surfaces of the objects in such a way that rendering them would provide the correct image.
- Image Space Algorithm : Keeps track of the distance of the point rasterized from the projection plane.
  - The nearest point from the projection plane is what gets rendered.
  - E.g z buffer algorithm.

- Culling : For convex objects like sphere, the parts of the object which are away from the viewer can be eliminated or culled before the rasterizer.
- glEnable(GL_CULL);

Interactive Mesh Displays

A mesh is a set of polygons that share vertices and edges.Used to calculate topographical elevations.



Suppose that the heights are given as a function of y as :

y = f(x,z)

then by taking samples of x and z ,y can be calculated as follows :

$y_{ij} = (x_i,z_j)$

$y_{i+1,j} = x_{i+1},z_j$

$y_{i,j+1} = x_i,z_{j+1}$

$y_{i+1,j+1} = x_{i+1},z_{j+1}$

Normalization

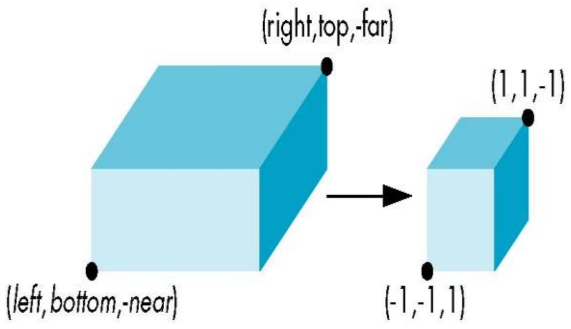
- Rather than derive a different projection matrix for each type of projection, we can convert all projections to orthogonal projections with the default view volume
- This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping
- We stay in four-dimensional homogeneous coordinates through both the modelview and projection transformations
  - o Both these transformations are nonsingular
  - o Default to identity matrices (orthogonal view)
- Normalization lets us clip against simple cube regardless of type of projection

- Delay final projection until end
    - Important for hidden-surface removal to retain depth information as long as possible

Orthogonal Normalization

glOrtho(left,right,bottom,top,near,far)

The diagram shows the normalization process



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

This involves 2 steps :

– Move center to origin

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

– Scale to have sides of length 2

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

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

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

UNIT - 7 **6 Hrs**

LIGHTING AND SHADING

Light and matter

Light sources

The Phong lighting model

Computation of vectors

Polygonal shading

Approximation of a sphere by recursive subdivisions

Light sources in OpenGL

Specification of materials in OpenGL

Shading of the sphere model

Global illumination

UNIT - 7 6 Hrs

LIGHTING AND SHADING

7.1 Light and matter

In order to obtain realistic images, it is important to apply lighting and shading to the images that we create using the graphic packages.

The openGL API provides set of functions to implement lighting, shading and material properties in the programs.

We need lighting because :

- Light-material interactions cause each point to have a different color or shade
- All these following properties affect the way an object looks
  - Light sources
  - Material properties
  - Location of viewer
  - Surface orientation

Types of Materials

- Specular surfaces – These surfaces exhibit high reflectivity. In these surfaces, the angle of incidence is almost equal to the angle of reflection.
- Diffuse surfaces – These are the surfaces which have a matt finish. These types of surfaces scatter light
- Translucent surfaces – These surfaces allow the light falling on them to partially pass through them.

The smoother a surface, the more reflected light is concentrated in the direction a perfect mirror would reflect light. A very rough surface scatters light in all directions.

Rendering Equation

- The infinite scattering and absorption of light can be described by the *rendering equation*. Rendering equation is global and includes
  - Shadows
  - Multiple scattering from object to object

Bidirectional Reflection Distribution function (BRDF)

The reflection, transmission, absorption of light is described by a single func BRDF

It is described by the following :

- Frequency of light

- 2 angles required to describe direction of input vector
- 2 angles required to describe the diurection of output vector.

Light-Material Interaction

- Light that strikes an object is partially absorbed and partially scattered (reflected)
- The amount reflected determines the color and brightness of the object
  - A surface appears red under white light because the red component of the light is reflected and the rest is absorbed
- The reflected light is scattered in a manner that depends on the smoothness and orientation of the surface

Simple Light Sources

These are the different types of light sources:

- Point source : This kind of source can be said as as a distant source or present infinite distance away (parallel)
- Spotlight : This source can be considered as a restrict light from ideal point source. A spotlight origins at a particular point and covers only a specific area in a cone shape.
- Ambient light
  - Same amount of light everywhere in scene
  - Can model contribution of many sources and reflecting surfaces

Any kind of light source will have 3 component colors namely R,G and B

Point source

- Emits light equally in all directions.
- A point source located at p_0 can be characterized by 3 component color function:

$L(p_0) = (L_r(p_0),L_g(p_0),L_b(p_0))$

Intensity of light received at a point p from point source p_0 is

$L(p,p_0) = (1/|p-p_0|^2)L(p_0)$

Ambient light

- The ambient illumination is given by : Ambient illumination = I_a

  And the RGB components are represented by

L_{ar},L_{ag},L_{ab}

Where L_a – scalar representing each component

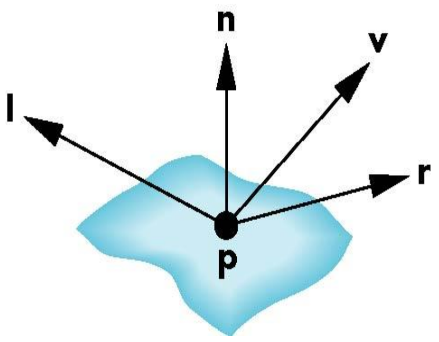
Spotlights

- A spotlight can be characterized by :
  - A cone whose apex is at P_s
  - Pointing in the direction I_s
  - Width determined by an angle θ
- Cosines are convenient functions for lighting calculations

The Phong Lighting Model

Phong developed a simple model that can be computed rapidly

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



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

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

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

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

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

Since the necessary computations are same for each light source,

$I = I_a+I_d+I_s$

Ambient Reflection

The amount of light reflected from an ambient source I_a is given by the ambient reflection coefficient: $R_a = k_a$

Since the ambient reflection co efficient is some positive factor,

$0<=k_a<=1$

Therefore $I_a = k_aL_a$

Diffuse Reflection

A Lambertian Surface has:

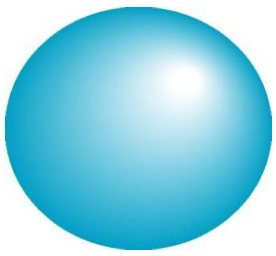
- Perfectly diffuse reflector
- Light scattered equally in all directions

Here the light reflected is proportional to the vertical component of incoming light

- reflected light ~cos q_i
- cos q_i = **l** · **n** if vectors normalized
- There are also three coefficients, k_r, k_b, k_g that show how much of each color component is reflected

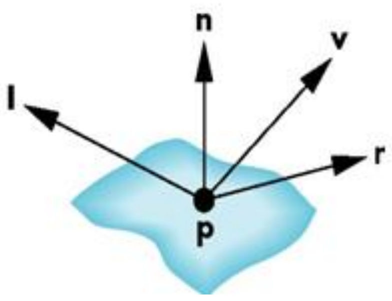
Specular Surfaces

- Most surfaces are neither ideal diffusers nor perfectly specular (ideal reflectors)
- Smooth surfaces show specular highlights due to incoming light being reflected in directions concentrated close to the direction of a perfect reflection . This kind of specular reflection could be observed in mirrors.



Modeling Specular Relections

- Phong proposed using a term that dropped off as the angle between the viewer and the ideal reflection increased



$$I_r \sim k_s I \cos^{\alpha}\phi$$

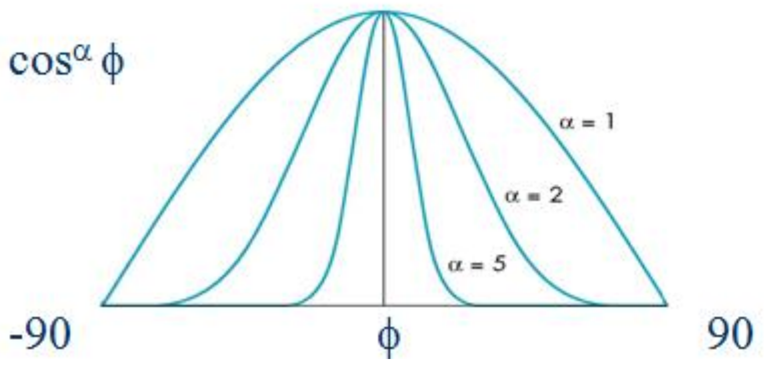
Here I_r is the reflected intensity

K_s = Absorption coefficient

I = Incoming intensity and $\cos^{\alpha}\varphi$

The Shininess Coefficient

Metals are lustrous by nature so they have a higher sineness coefficient. The figure below shows shineness coefficients for different materials:



Values of a between 100 and 200 correspond to metals

Values between 5 and 10 give surface that look like plastic

Computation of Vectors

- Normal vectors :

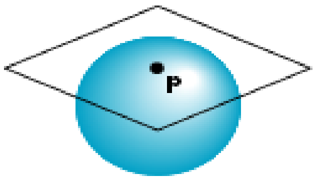
  Given 3 non collinear points (p0,p1,p2) on a plane , the normal can be calculated by

  n =(p2-p0) X (p1-p0)
- If a surface is described implicitly by the function : f(p) = f(x,y,z) =0 and if p & p0 are 2 points close to each other on a smooth surface

Normal to Sphere

- Implicit function f(x,y.z)=0
- Normal given by gradient
- Sphere f(**p**)=**p**·**p**-1

- $n = [\partial f/\partial x, \partial f/\partial y, \partial f/\partial z]^T = \mathbf{p}$



Parametric Form

- For sphere
  $x=x(u,v)=\cos u \sin v$
  $y=y(u,v)=\cos u \cos v$
  $z= z(u,v)=\sin u$

- Tangent plane determined by vectors
  $\partial \mathbf{p}/\partial u = [\partial x/\partial u, \partial y/\partial u, \partial z/\partial u]T$
  $\partial \mathbf{p}/\partial v = [\partial x/\partial v, \partial y/\partial v, \partial z/\partial v]T$
- Normal given by cross product
  $\mathbf{n} = \partial \mathbf{p}/\partial u \times \partial \mathbf{p}/\partial v$

Polygonal Shading

Flat shading

- In case of flat shading there are distinct boundaries after color interpolation 3 vectors needed for shading are: l,n,v .
- The openGL function to enable flat shading is :glShadeModel(GL_FLAT)
- For a flat polygon,n is constant as the normal **n** is same at all points on the polygon.
- Also if we assume a distant viewer, the vector **v** is constant and if we consider a distant light source then the vector l is also a constant.
- Here all the 3 vectors are constant and therefore the shading calculations needs to be done only once for an entire polygon and each point on the polygon is assigned the same shade. This technique is known as Flat shading.
- Disadvantage : But if we consider light sources and the viewer near the polygon, then flat shading will show differences in shading and the human eye is very sensitive to slightest of such differences due to the principle of “Lateral Inhibition”

Light Sources in OpenGL

- Shading calculations are enabled by
    - glEnable(GL_LIGHTING)
    - Once lighting is enabled, glColor() ignored
- Must enable each light source individually
    - glEnable(GL_LIGHTi) i=0,1…..
- For each light source, we can set an RGB for the diffuse, specular, and ambient parts, and the position

```
GLfloat diffuse0[]={1.0, 0.0, 0.0, 1.0};
GLfloat ambient0[]={1.0, 0.0, 0.0, 1.0};
GLfloat specular0[]={1.0, 0.0, 0.0, 1.0};
Glfloat light0_pos[]={1.0, 2.0, 3,0, 1.0};
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightv(GL_LIGHT0, GL_POSITION, light0_pos);
glLightv(GL_LIGHT0, GL_AMBIENT, ambient0);
glLightv(GL_LIGHT0, GL_DIFFUSE, diffuse0);
glLightv(GL_LIGHT0, GL_SPECULAR, specular0);
```

- The source colors are specified in RGBA
- The position is given in homogeneous coordinates
    - If w =1.0, we are specifying a finite location
    - If w =0.0, we are specifying a parallel source with the given direction vector
- The coefficients in the distance terms are by default a=1.0 (constant terms), b=c=0.0 (linear and quadratic terms).

Handling Hidden Surfaces

Lighting and Shading needs to be done properly to the hidden surfaces as well. In order to enable the shading for hidden surfaces, we use glLightModeli(GL_LIGHT_TWO_SIDED,GL_TRUE)

Material Properties

All material properties are specified by :

glMaterialfv(GLenum face, GLenum type, GLfloat *pointer_to_array)

We have seen that each material has a different ambient, diffuse and specular properties.

```
GLfloat ambient[] = {1.0,0.0,0.0,1.0}
GLfloat diffuse[] = {1.0,0.8,0.0,1.0}
GLfloat specular[] = {1.0, 1.0, 1.0,1.0}
```

- Defining shineness and emissive properties

```
glMaterialf(GL_FRONT_AND_BACK,GL_SHINENESS,100.0)
GLfloat emission[] = {0.0,0.3,0.3,1.0};
glMaterialfv(GL_FRONT_AND_BACK,GL_EMISSION,
emission)
```

- Defining Material Structures

```
typedef struct materialStruct
        GLfloat ambient[4];
         GLfloat diffuse[4];
        GLfloat specular[4];
        GLfloat shineness;
materialStruct;
```

Global Illumination

- Ray tracer
  - Considers the ray tracing model to find out the intensity at any point on the object
    - Best suited for specular objects
- Radiosity Renderer
    - Based on the principle that the total light energy is conserved. Involves lot of mathematical calculations
    - Best suited for diffuse objects

UNIT - 8 **8 Hrs**

IMPLEMENTATION

Basic implementation strategies

The 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

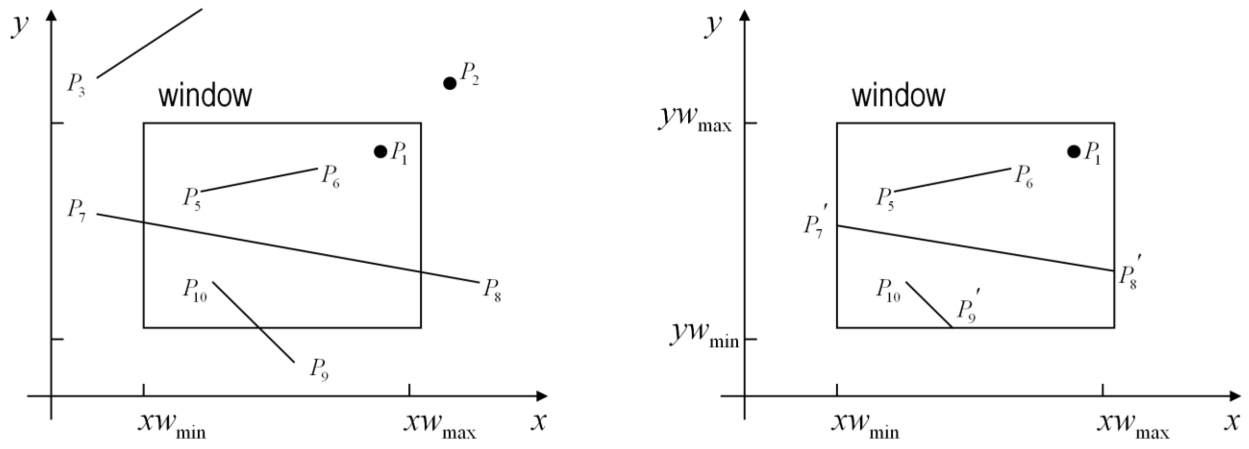
Display considerations.

UNIT - 8 **8 Hrs**

IMPLEMENTATION

8.1 Clipping

Clipping is a process of removing the parts of the image that fall outside the view volume since it would not be a part of the final image seen on the screen.



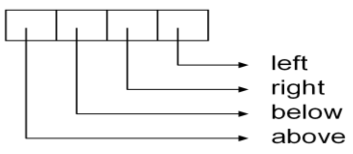
(a) before clipping (b) after clipping

Sutherland and Cohen 2D Clipping Algorithm

Basic Idea

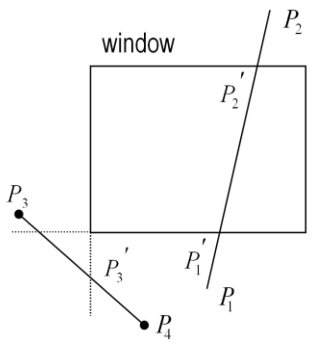
- Encode the line endpoints
- Successively divide the line segments so that they are completely contained in the window or completely lies outside the window

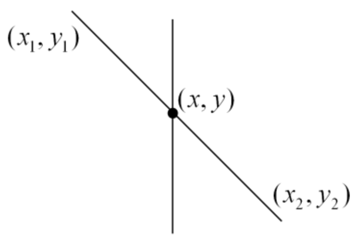
1001	1000	1010
0001	0000	0010
0101	0100	0110



•

Clipping happens as follows :





Bresenham's mid point line algorithm

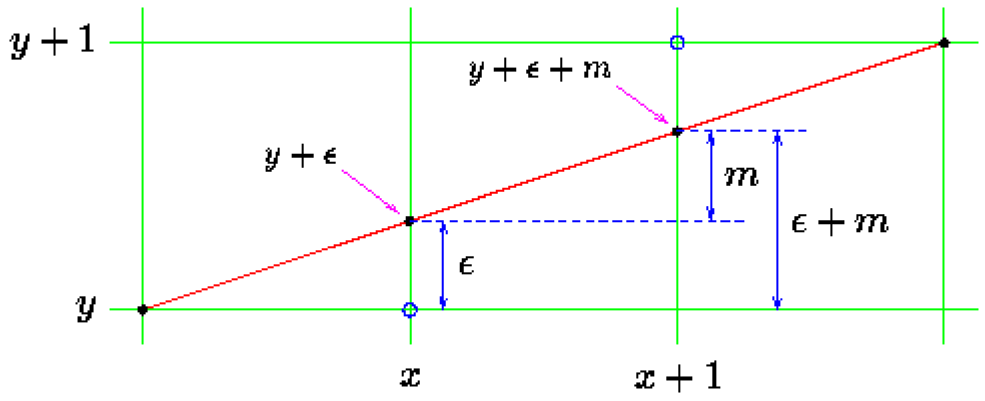
Consider drawing a line on a raster grid where we restrict the allowable slopes of the line to the range $0 \le m \le 1$.

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

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

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

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



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

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

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

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

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

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

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

Otherwise it is:

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

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

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

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

$$\epsilon + m < 0.5$$

$$\epsilon + \Delta y / \Delta x < 0.5$$

$$2\epsilon\Delta x + 2\Delta y < \Delta x$$

All quantities in this inequality are now integral.

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

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

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

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

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

Multiplying through by Δx yields:

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

which is in ϵ' form.

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

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

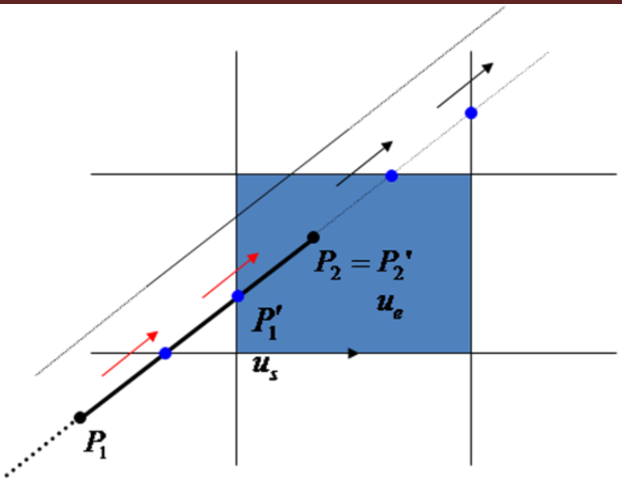
```
$\epsilon' \leftarrow 0, \quad y \leftarrow y_1$
For $x \leftarrow x_1$ to $x_2$ do
    Plot point at $(x, y)$.
    If ( $2(\epsilon' + \Delta y) < \Delta x$ )
        $\epsilon' \leftarrow \epsilon' + \Delta y$
    Else
        $y \leftarrow y + 1, \quad \epsilon' \leftarrow \epsilon' + \Delta y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

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

Liang & Barsky's Algorithm

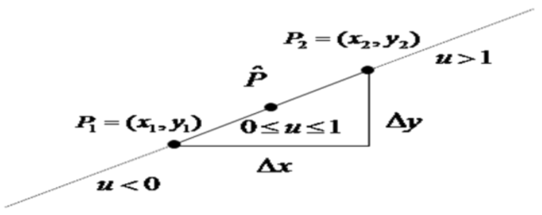
$$\hat{P} = P_1 + u(P_2 - P_1),$$
$$0 \le u \le 1$$

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

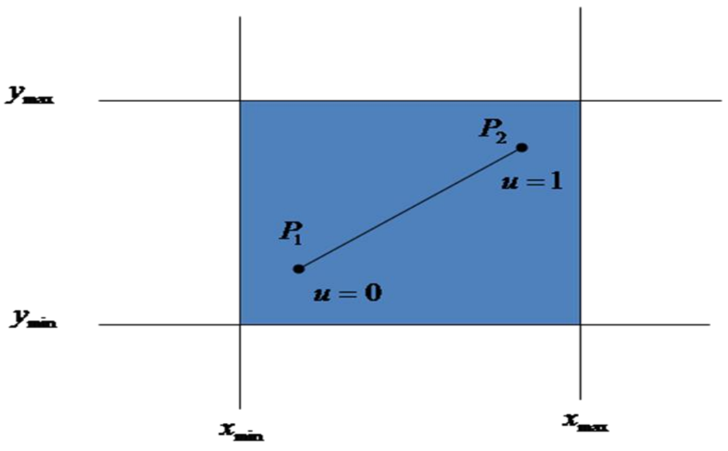


$$\hat{P} = vP_1 + uP_2$$
$$u + v = 1$$

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



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



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

Rasterization

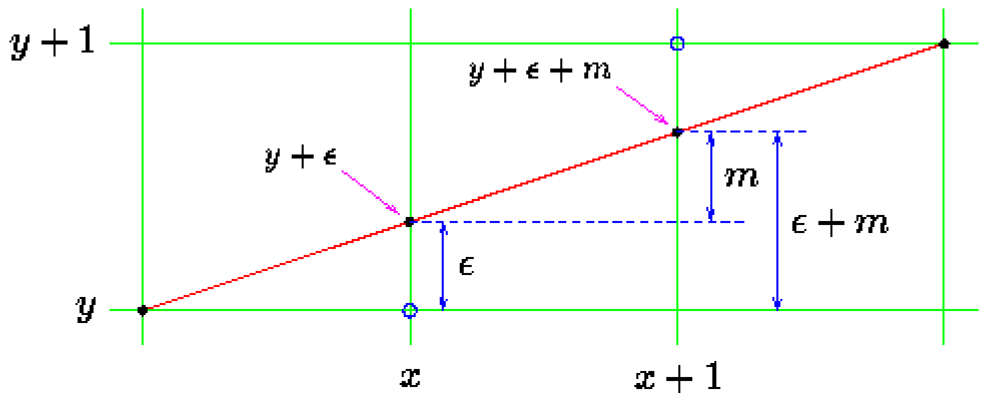
Consider drawing a line on a raster grid where we restrict the allowable slopes of the line to the range $0 \leq m \leq 1$.

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

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

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

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



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

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

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

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

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

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

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

Otherwise it is:

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

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

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

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

$$\epsilon + m < 0.5$$

$$\epsilon + \Delta y / \Delta x < 0.5$$

$$2\epsilon\Delta x + 2\Delta y < \Delta x$$

All quantities in this inequality are now integral.

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

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

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

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

$$\epsilon \leftarrow \epsilon + m$$

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

Multiplying through by Δx yields:

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

which is in ϵ' form.

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

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

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

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

Hidden Surface Removal

Drawing the objects that are closer to the viewing position and eliminating objects which are obscured by other "nearer" objects

Two General Categories of Algorithms:

- Object Space: Compares objects and parts of object to each other to determine which surfaces and lines should be labeled as invisible
Generally used for *hidden line removal*
- Image Space: Visibility is determined point by point at each pixel position on the projection plane Generally used for *hidden surface removal Back face culling* is also a form of hidden surface removal

<u>Painter's Algorithm</u>

The painter’s algorithm is the simplest hidden surface removal algorithm

- Start by drawing the objects which are furthest from the viewpoint
- Continue drawing objects working from far to near
- Draw the closest objects last

The nearer objects will obscure the farther objects

♦ Problems:

- Objects must be drawn in a particular order based upon their distance from the view point
- If the viewing position is changed, the drawing order must be changed

Z Buffering

Commonly used image-space algorithm which uses a *depth* or *Z buffer* to keep track of the distance from the projection plane to each point on the object

- For each pixel position, the surface with the smallest z coordinate is visible
- Depth or Z values are usually normalized to values between zero and one

Z Buffer Algorithm

1. Clear the color buffer to the background color
2. Initialize all xy coordinates in the Z buffer to one
3. For each fragment of each surface, compare depth values to those already stored in the *Z* buffer
    - Calculate the distance from the projection plane for each xy position on the surface
    - If the distance is less than the value currently stored in the *Z* buffer:

      Set the corresponding position in the color buffer to the color of the fragment

      Set the value in the Z buffer to the distance to that object
    - Otherwise: Leave the color and *Z* buffers unchanged

♦ Comments

- *Z*-buffer testing can increase application performance
- Software buffers are much slower than specialized hardware depth buffers
- The number of bitplanes associated with the Z buffer determine its precision or resolution