UNIT I - 2D PRIMITIVES

Output primitives – Line, Circle and Ellipse drawing algorithms - Attributes of output primitives –Two dimensional Geometric transformation - Two dimensional viewing – Line, Polygon, Curve and Text clipping algorithms

Introduction

A picture is completely specified by the set of intensities for the pixel positions in the display. Shapes and colors of the objects can be described internally with pixel arrays into the frame buffer or with the set of the basic geometric – structure such as straight line segments and polygon color areas. To describe structure of basic object is referred as output primitives.

Each output primitive is specified with input co-ordinate data and other information about the way that objects is to be displayed. Additional output primitives that can be used to constant a picture include circles and other conic sections, quadric surfaces, Spline curves and surfaces, polygon floor areas and character string.

Points and Lines

Point plotting is accomplished by converting a single coordinate position furnished by an application program into appropriate operations for the output device. With a CRT monitor, for example, the electron beam is turned on to illuminate the screen phosphor at the selected location

Line drawing is accomplished by calculating intermediate positions along the line path between two specified end points positions. An output device is then directed to fill in these positions between the end points

Pixel positions are referenced according to scan-line number and column number (pixel position across a scan line). Scan lines are numbered consecutively from 0, starting at the bottom of the screen; and pixel columns are numbered from **0,** left to right across each scan line

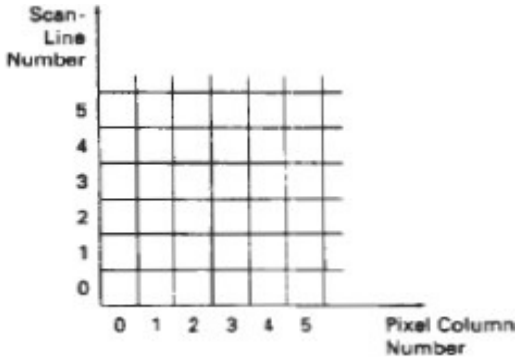


Figure: Pixel Positions reference by scan line number and column number

To load an intensity value into the frame buffer at a position corresponding to column x along scan line y,

setpixel (x, y)

To retrieve the current frame buffer intensity setting for a **specified** location we use a low level function

getpixel (x, y)

Line Drawing Algorithms

- Digital Differential Analyzer (DDA) Algorithm
- Bresenham's Line Algorithm

Digital Differential Analyzer (DDA) Algorithm

The digital differential analyzer (DDA) is a scan-conversion line algorithm based on calculation either Δy or Δx

The lines at unit intervals in one coordinate and determine corresponding integer values nearest the line path for the other coordinate.

A line with positive slop, if the slope is less than or equal to 1, at unit x intervals ($\Delta x=1$) and compute each successive y values as

$$y_{k+1} = y_k + m \qquad (6)$$

Subscript k takes integer values starting from 1 for the first point and increases by 1 until the final endpoint is reached. ***m*** can **be** any real number between 0 and 1 and, the calculated y values must be rounded to the nearest integer

For lines with a positive slope greater than 1 we reverse the roles of x and y, ($\Delta y=1$) and calculate each succeeding x value as

$$x_{k+1} = x_k + (1/m) \qquad (7)$$

Equation (6) and (7) are based on the assumption that lines are to be processed from the left endpoint to the right endpoint.

If this processing is reversed, $\Delta x=-1$ that the starting endpoint is at the right

$$y_{k+1} = y_k - m \qquad (8)$$

When the slope is greater than 1 and $\Delta y = -1$ with

$$x_{k+1} = x_k - 1(1/m) \qquad (9)$$

If the absolute value of the slope is less than 1 and the start endpoint is at the left, we set $\Delta x = 1$ and calculate y values with Eq. (6)

When the start endpoint is at the right (for the same slope), we set $\Delta x = -1$ and obtain y positions from Eq. (8). Similarly, when the absolute value of a negative slope is greater than 1, we use $\Delta y = -1$ and Eq. (9) or we use $\Delta y = 1$ and Eq. (7).

Algorithm

```
#define ROUND(a) ((int)(a+0.5))
void lineDDA (int xa, int ya, int xb, int yb)
{
int dx = xb - xa, dy = yb - ya, steps, k;
float xIncrement, yIncrement, x = xa, y = ya;
if (abs (dx) > abs (dy) steps = abs (dx) ;
else steps = abs dy);
xIncrement = dx / (float) steps;
yIncrement = dy / (float) steps
setpixel (ROUND(x), ROUND(y) ) :
for (k=0; k<steps; k++)
{
x += xIncrement;
y += yIncrement;
setpixel (ROUND(x), ROUND(y));
}
}
```

Algorithm Description:

Step 1 : Accept Input as two endpoint pixel positions
Step 2: Horizontal and vertical differences between the endpoint positions *are* assigned to parameters dx and dy (Calculate dx=xb-xa and dy=yb-ya).
Step 3: The difference with the greater magnitude determines the value of parameter steps.
Step 4 : Starting with pixel position (xa, ya), determine the offset needed at each step to generate the next pixel position along the line path.
Step 5: loop the following process for steps number of times

a. Use a unit of increment or decrement in the x and y direction
b. if xa is less than xb the values of increment in the x and y directions are 1 and m
c. if xa is greater than xb then the decrements -1 and – m are used.

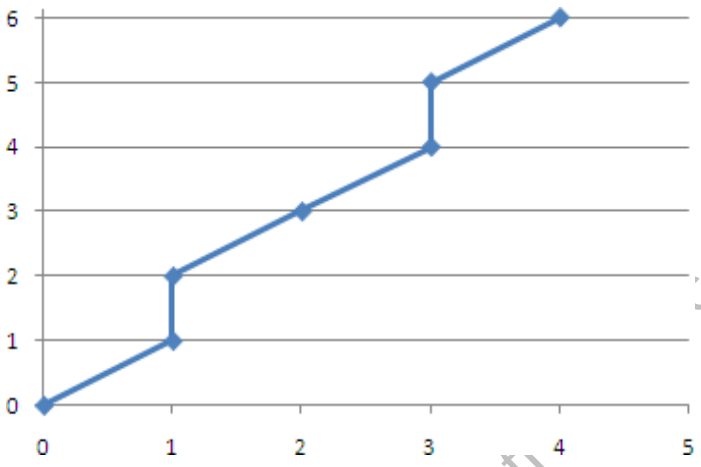
Example : Consider the line from (0,0) to (4,6)

1. xa=0, ya =0 and xb=4 yb=6
2. dx=xb-xa = 4-0 = 4 and dy=yb-ya=6-0= 6
3. x=0 and y=0
4. 4 > 6 (false) so, steps=6
5. Calculate xIncrement = dx/steps = 4 / 6 = 0.66 and yIncrement = dy/steps =6/6=1
6. Setpixel(x,y) = Setpixel(0,0) (Starting Pixel Position)

7. Iterate the calculation for xIncrement and yIncrement for steps(6) number of times
8. Tabulation of the each iteration

k	X	Y	Plotting points (Rounded to Integer)
0	0+0.66=0.66	0+1=1	(1,1)
1	0.66+0.66=1.32	1+1=2	(1,2)
2	1.32+0.66=1.98	2+1=3	(2,3)
3	1.98+0.66=2.64	3+1=4	(3,4)
4	2.64+0.66=3.3	4+1=5	(3,5)
5	3.3+0.66=3.96	5+1=6	(4,6)

Result :



Advantages of DDA Algorithm

1. It is the simplest algorithm
2. It is a is a **faster method** for calculating pixel positions

Disadvantages of DDA Algorithm

1. Floating point arithmetic in DDA algorithm is still time-consuming
2. End point accuracy is poor

Bresenham's Line Algorithm

An accurate and efficient raster line generating algorithm developed by Bresenham that uses only incremental integer calculations.

In addition, Bresenham's line algorithm can be adapted to display circles and other curves. To illustrate Bresenham's approach, we- first consider the scan-conversion process for lines with positive slope less than 1.

Bresenham's line Drawing Algorithm for |m| < 1

1. Input the two line endpoints and store the left end point in (x_0,y_0)
2. load (x_0,y_0) into frame buffer, ie. Plot the first point.
3. Calculate the constants Δx, Δy, $2\Delta y$ and obtain the starting value for the decision parameter as $P_0 = 2\Delta y - \Delta x$
4. At each x_k along the line, starting at k=0 perform the following test
   If $P_k < 0$, the next point to plot is(x_{k+1},y_k) and
   $P_{k+1} = P_k + 2\Delta y$
   otherwise, the next point to plot is (x_{k+1},y_{k+1}) and
   $P_{k+1} = P_k + 2\Delta y - 2\Delta x$

5. Perform step4 Δx times.

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

Bresenham's line Drawing Algorithm for |m| < 1

1. Input the two line endpoints and store the left end point in (x_0,y_0)
2. load (x_0,y_0) into frame buffer, ie. Plot the first point.
3. Calculate the constants Δx, Δy, $2\Delta y$ and obtain the starting value for the decision parameter as $P_0 = 2\Delta y - \Delta x$
4. At each x_k along the line, starting at k=0 perform the following test
   If $P_k < 0$, the next point to plot is(x_{k+1},y_k) and
   $P_{k+1} = P_k + 2\Delta y$
   Otherwise, the next point to plot is (x_{k+1},y_{k+1})
   and $P_{k+1} = P_k + 2\Delta y - 2\Delta x$

5. Perform step4 Δx times.

Implementation of Bresenham Line drawing Algorithm

```
void lineBres (int xa,int ya,int xb, int yb)
{
int dx = abs( xa – xb) , dy = abs (ya - yb);
int p = 2 * dy – dx;
int twoDy = 2 * dy, twoDyDx = 2 *(dy - dx);
int x , y, xEnd;

/* Determine which point to use as start, which as end * /

if (xa > x b )
{
x = xb;
y = yb;
xEnd = xa;
}
 else
{
x = xa;
y = ya;
xEnd = xb;
}
setPixel(x,y);
while(x<xEnd)
{
x++;
if (p<0)
p+=twoDy;
else
{
y++;
p+=twoDyDx;
}
setPixel(x,y);
}
}
```


Example : Consider the line with endpoints (20,10) to (30,18)

The line has the slope m= (18-10)/(30-20)=8/10=0.8

$\Delta x = 10$ $\Delta y=8$

The initial decision parameter has the value

$p_0 = 2\Delta y - \Delta x = 6$

and the increments for calculating successive decision parameters are

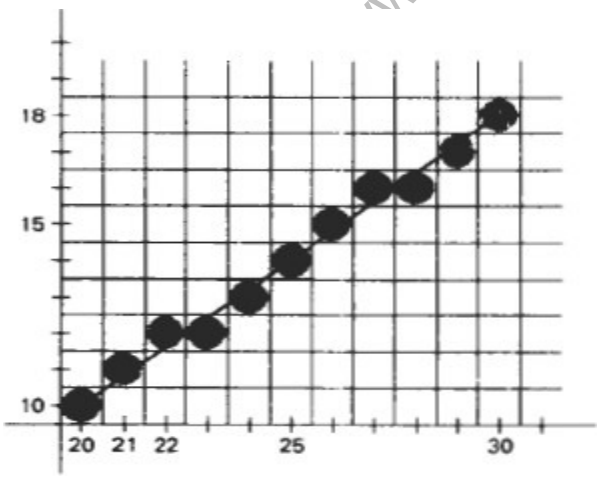
$2\Delta y=16$ $2\Delta y-2\ \Delta x= -4$

We plot the initial point $(x_0,y_0) = (20,10)$ and determine successive pixel positions along the line path from the decision parameter as

Tabulation

k	p_k	(x_k+1, y_K+1)
0	6	(21,11)
1	2	(22,12)
2	-2	(23,12)
3	14	(24,13)
4	10	(25,14)
5	6	(26,15)
6	2	(27,16)
7	-2	(28,16)
8	14	(29,17)
9	10	(30,18)

Result



Advantages

- Algorithm is Fast
- Uses only integer calculations

Disadvantages

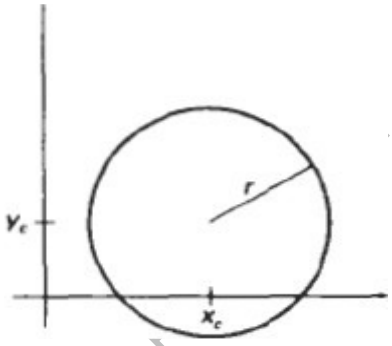
It is meant only for basic line drawing.

Circle-Generating Algorithms

General function is available in a graphics library for displaying various kinds of curves, including circles and ellipses.

Properties of a circle

A circle is defined as a set of points that are all the given distance (x_c,y_c).



This distance relationship is expressed by the Pythagorean Theorem in Cartesian coordinates as

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

Use above equation to calculate the position of points on a circle circumference by stepping along the x axis in unit steps from x_c-r to x_c+r and calculating the corresponding y values at each position as

$$y = y_c + (-)\,(r^2 - (xc - x)^2)^{1/2} \qquad (2)$$

This is not the best method for generating a circle for the following reason

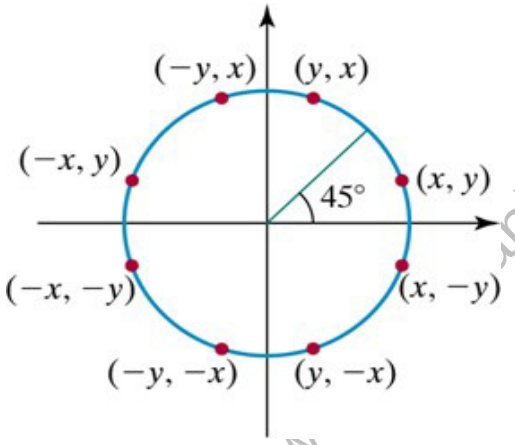
Considerable amount of computation
Spacing between plotted pixels is not uniform

To eliminate the unequal spacing is to calculate points along the circle boundary using polar coordinates r and θ. Expressing the circle equation in parametric polar from yields the pair of equations

$$x = x_c + r\cos\theta \qquad y = y_c + r\sin\theta$$

When a display is generated with these equations using a fixed angular step size, a circle is plotted with equally spaced points along the circumference. To reduce calculations use a large angular separation between points along the circumference and connect the points with straight line segments to approximate the circular path

To generate all pixel positions around a circle by calculating only the points within the sector from x=0 to y=0. the slope of the curve in this octant has an magnitude less than of equal to 1.0. at x=0, the circle slope is 0 and at x=y, the slope is -1.0.



In this approach is to test the halfway position between two pixels to determine if this midpoint is inside or outside the circle boundary. This method is more easily applied to other conics and for an integer circle radius the midpoint approach generates the same pixel positions as the Bresenham circle algorithm.

For a straight line segment the midpoint method is equivalent to the bresenham line algorithm. The error involved in locating pixel positions along any conic section using the midpoint test is limited to one half the pixel separations.

Midpoint circle Algorithm

1. Input radius r and circle center (x_c,y_c) and obtain the first point on the circumference of the circle centered on the origin as

$(x_0,y_0) = (0,r)$

2. Calculate the initial value of the decision parameter as $P_0=(5/4)-r$
3. At each x_k position, starting at k=0, perform the following test. If $P_k < 0$ the next point along the circle centered on (0,0) is (x_{k+1},y_k) and $P_{k+1}=P_k+2x_{k+1}+1$

Otherwise the next point along the circle is (x_{k+1},y_{k-1}) and $P_{k+1}=P_k+2x_{k+1}+1-2\ y_{k+1}$

Where $2x_{k+1}=2x_{k+2}$ and $2y_{k+1}=2y_{k-2}$

4. Determine symmetry points in the other seven octants.
5. Move each calculated pixel position (x,y) onto the circular path centered at (x_c,y_c) and plot the coordinate values.

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

6. Repeat step 3 through 5 until x>=y.

Example : Midpoint Circle Drawing

Given a circle radius r=10

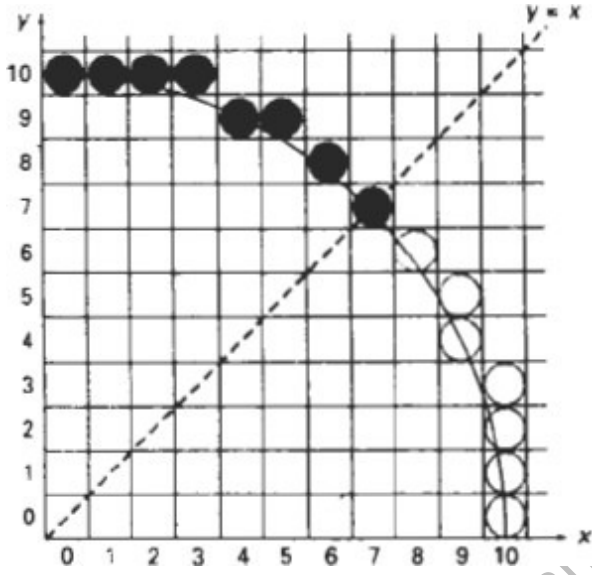
The circle octant in the first quadrant from x=0 to x=y. The initial value of the decision parameter is $P_0=1-r = -9$

For the circle centered on the coordinate origin, the initial point is $(x_0,y_0)=(0,10)$ and initial increment terms for calculating the decision parameters are

$2x_0=0\ ,\ 2y_0=20$

Successive midpoint decision parameter values and the corresponding coordinate positions along the circle path are listed in the following table.

k	p_k	(x_{k+1}, y_{k-1})	$2x_{k+1}$	$2y_{k+1}$
0	-9	(1,10)	2	20
1	-6	(2,10)	4	20
2	-1	(3,10)	6	20
3	6	(4,9)	8	18
4	-3	(5,9)	10	18
5	8	(6,8)	12	16
6	5	(7,7)	14	14



Implementation of Midpoint Circle Algorithm

```
void circleMidpoint (int xCenter, int yCenter, int radius)
{
int x = 0;
int y = radius;
int p = 1 - radius;
void circlePlotPoints (int, int, int, int);
/* Plot first set of points */
circlePlotPoints (xCenter, yCenter, x, y);
while (x < y)
{
x++ ;
if (p < 0)
p +=2*x +1;
else
{
y--;
p +=2* (x - Y) + 1;
}
circlePlotPoints(xCenter, yCenter, x, y)
}
}
void circlePlotPolnts (int xCenter, int yCenter, int x, int y)
{
setpixel (xCenter + x, yCenter + y ) ;
setpixel (xCenter - x. yCenter + y);
setpixel (xCenter + x, yCenter - y);
setpixel (xCenter - x, yCenter - y ) ;
setpixel (xCenter + y, yCenter + x);
setpixel (xCenter - y , yCenter + x);
setpixel (xCenter t y , yCenter - x);
setpixel (xCenter - y , yCenter - x);
}
```


Ellipse-Generating Algorithms

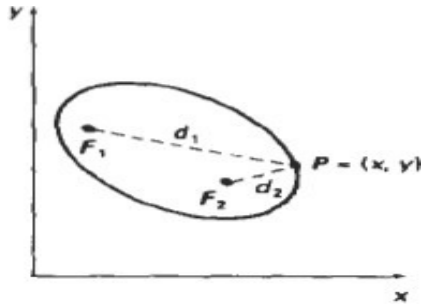
An ellipse is an elongated circle. Therefore, elliptical curves can be generated by modifying circle-drawing procedures to take into account the different dimensions of an ellipse along the major and minor axes.

Properties of ellipses

An ellipse can be given in terms of the distances from any point on the ellipse to two fixed positions called the **foci** of the ellipse. The sum of these two distances is the same values for all points on the ellipse.

If the distances to the two focus positions from any point p=(x,y) on the ellipse are labeled d1 and d2, then the general equation of an ellipse can be stated as

d1+d2=constant



Expressing distances d1 and d2 in terms of the focal coordinates F1=(x_1,y_2) and F2=(x_2,y_2)

$$\mathbf{sqrt((x-x_1)^2+(y-y_1)^2)+sqrt((x-x_2)^2+(y-y_2)^2)=constant}$$

By squaring this equation isolating the remaining radical and squaring again. The general ellipse equation in the form

$$\mathbf{Ax^2+By^2+Cxy+Dx+Ey+F=0}$$

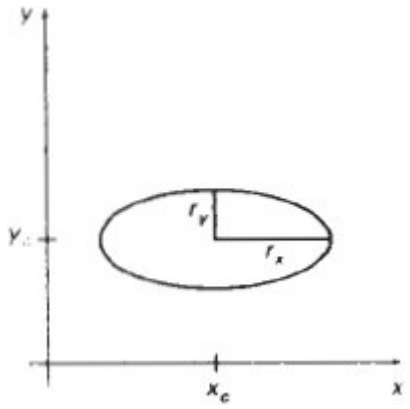
The coefficients A,B,C,D,E, and F are evaluated in terms of the focal coordinates and the dimensions of the major and minor axes of the ellipse.

The major axis is the straight line segment extending from one side of the ellipse to the other through the foci. The minor axis spans the shorter dimension of the ellipse, perpendicularly bisecting the major axis at the halfway position (ellipse center) between the two foci.

An interactive method for specifying an ellipse in an arbitrary orientation is to input the two foci and a point on the ellipse boundary.

Ellipse equations are simplified if the major and minor axes are oriented to align with the coordinate axes. The major and minor axes oriented parallel to the x and y axes parameter r_x for this example labels the semi major axis and parameter r_y labels the semi minor axis

$$((x-x_c)/r_x)^2+((y-y_c)/r_y)^2=1$$



Using polar coordinates r and θ, to describe the ellipse in Standard position with the parametric equations

$x=x_c+r_x\cos\theta$
$y=y_c+r_x\sin\theta$

Angle θ called the eccentric angle of the ellipse is measured around the perimeter of a bounding circle.

We must calculate pixel positions along the elliptical arc throughout one quadrant, and then we obtain positions in the remaining three quadrants by symmetry

Mid point Ellipse Algorithm

1. Input r_x,r_y and ellipse center (x_c,y_c) and obtain the first point on an ellipse centered on the origin as

   $(x_0,y_0) = (0,r_y)$

2. Calculate the initial value of the decision parameter in region 1 as

   $P1_0=r_y^2-r_x^2r_y+(1/4)r_x^2$

3. At each x_k position in region1 starting at k=0 perform the following test. If $P1_k<0$, the next point along the ellipse centered on (0,0) is (x_{k+1}, y_k) and

   $p1_{k+1} = p1_k +2r_y^2x_{k+1} + r_y^2$

$$(x_0,y_0) = (0,r_y)$$

4. Calculate the initial value of the decision parameter in region 1 as
$$P1_0=r_y^2-r_x^2r_y+(1/4)r_x^2$$

5. At each x_k position in region1 starting at k=0 perform the following test. If $P1_k<0$, the next point along the ellipse centered on (0,0) is (x_{k+1}, y_k) and

$$p1_{k+1} = p1_k +2 r_y^2x_{k+1} + r_y^2$$

Otherwise the next point along the ellipse is (x_k+1, y_k-1) and

$$p1_{k+1} = p1_k +2 r_y^2x_{k+1} - 2r_x^2y_{k+1} + r_y^2$$

with

$$2r_y^2x_{k+1} = 2r_y^2x_k + 2r_y^2$$
$$2r_x^2y_{k+1} = 2r_x^2y_k + 2r_x^2$$

And continue until $2ry^2 x>=2rx^2 y$

6. Calculate the initial value of the decision parameter in region 2 using the last point (x_0,y_0) is the last position calculated in region 1.

$$p2_0 = r_y^2(x_0+1/2)^2+r_x^2(y_0-1)^2 - r_x^2r_y^2$$

7. At each position yk in region 2, starting at k=0 perform the following test, If $p2_k>0$ the next point along the ellipse centered on (0,0) is (x_k,y_{k-1}) and

$$p2_{k+1} = p2_k - 2r_x^2y_{k+1}+r_x^2$$

Otherwise the next point along the ellipse is (x_k+1,y_k-1) and

$$p2_{k+1} = p2_k + 2r_y^2x_{k+1} - 2r_x^2y_{k+1} + r_x^2$$

Using the same incremental calculations for x any y as in region 1

8. Determine symmetry points in the other three quadrants.
9. Move each calculate pixel position (x,y) onto the elliptical path centered on (x_c,y_c) and plot the coordinate values
$x=x+x_c, y=y+y_c$
10. Repeat the steps for region1 unit $2r_y^2x>=2r_x^2y$

Example : Mid point ellipse drawing

Input ellipse parameters $r_x=8$ and $r_y=6$ the mid point ellipse algorithm by determining raster position along the ellipse path is the first quadrant. Initial values and increments for the decision parameter calculations are

$2r_y^2 x=0$ (with increment $2r_y^2=72$)

$2r_x^2 y=2r_x^2 r_y$ (with increment $-2r_x^2= -128$)

For region 1 the initial point for the ellipse centered on the origin is (x_0,y_0) = (0,6) and the initial decision parameter value is

$p1_0=r_y^2-r_x^2r_y^2+1/4r_x^2=-332$

Successive midpoint decision parameter values and the pixel positions along the ellipse are listed in the following table.

k	$p1_k$	x_{k+1},y_{k+1}	$2r_y^2x_{k+1}$	$2r_x^2y_{k+1}$
0	-332	(1,6)	72	768
1	-224	(2,6)	144	768
2	-44	(3,6)	216	768
3	208	(4,5)	288	640
4	-108	(5,5)	360	640
5	288	(6,4)	432	512
6	244	(7,3)	504	384

Move out of region 1, $2ry2x > 2rx^2y$.

For a region 2 the initial point is (x_0,y_0)=(7,3) and the initial decision parameter is

$p2_0 = f_{ellipse}(7+1/2,2) = -151$

The remaining positions along the ellipse path in the first quadrant are then calculated as

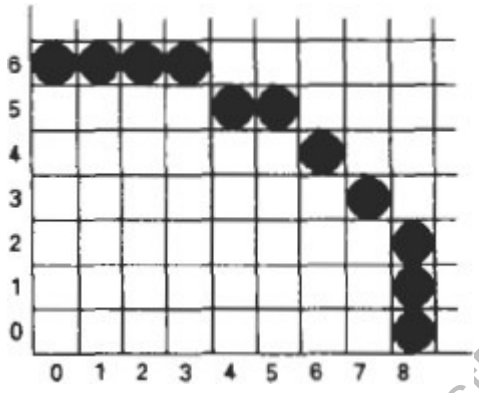
k	$P2_k$	x_{k+1},y_{k+1}	$2r_y^2x_{k+1}$	$2r_x^2y_{k+1}$
0	-151	(8,2)	576	256
1	233	(8,1)	576	128
2	745	(8,0)	-	-

Implementation of Midpoint Ellipse drawing

```
#define Round(a) ((int)(a+0.5))
void ellipseMidpoint (int xCenter, int yCenter, int Rx, int Ry)
{
int Rx2=Rx*Rx;
int Ry2=Ry*Ry;
int twoRx2 = 2*Rx2;
int twoRy2 = 2*Ry2;
int p;
int x = 0;
int y = Ry;
int px = 0;
int py = twoRx2* y;
void ellipsePlotPoints ( int , int , int , int ) ;
/* Plot the first set of points */
ellipsePlotPoints (xcenter, yCenter, x,y ) ;

/ * Region 1 */
p = ROUND(Ry2 - (Rx2* Ry) + (0.25*Rx2));
while (px < py)
{
x++;
px += twoRy2;
i f (p < 0)
p += Ry2 + px;
else
{
y - - ;
py -= twoRx2;
p += Ry2 + px - py;
}
ellipsePlotPoints(xCenter, yCenter,x,y);
}
/* Region 2 */
p = ROUND (Ry2*(x+0.5)*' (x+0.5)+ Rx2*(y- 1 )* (y- 1 ) - Rx2*Ry2);
while (y > 0 )
{
y--;
py -= twoRx2;
i f (p > 0)
p += Rx2 - py;
else
```

```
{
x++;
px+=twoRy2;
p+=Rx2-py+px;
}
ellipsePlotPoints(xCenter, yCenter,x,y);
}
}
void ellipsePlotPoints(int xCenter, int yCenter,int x,int y);
{
setpixel (xCenter + x, yCenter + y);
setpixel (xCenter - x, yCenter + y);
setpixel (xCenter + x, yCenter - y);
setpixel (xCenter- x, yCenter - y);
}
```



Attributes of output primitives

Any parameter that affects the way a primitive is to be displayed is referred to as an attribute parameter. Example attribute parameters are color, size etc. A line drawing function for example could contain parameter to set color, width and other properties.

1. Line Attributes
2. Curve Attributes
3. Color and Grayscale Levels
4. Area Fill Attributes
5. Character Attributes
6. Bundled Attributes

Line Attributes

Basic attributes of a straight line segment are its type, its width, and its color. In some graphics packages, lines can also be displayed using selected pen or brush options

- Line Type
- Line Width
- Pen and Brush Options
- Line Color

Line type

Possible selection of line type attribute includes solid lines, dashed lines and dotted lines. To set line type attributes in a **PHIGS** application program, a user invokes the function

setLinetype (lt)

Where parameter lt is assigned a positive integer value of 1, 2, 3 or 4 to generate lines that are solid, dashed, dash dotted respectively. Other values for line type parameter it could be used to display variations in dot-dash patterns.

Line width

Implementation of line width option depends on the capabilities of the output device to set the line width attributes.

setLinewidthScaleFactor(lw)

Line width parameter lw is assigned a positive number to indicate the relative width of line to be displayed. A value of 1 specifies a standard width line. A user could set lw to a value of 0.5 to plot a line whose width is half that of the standard line. Values greater than 1 produce lines thicker than the standard.

Line Cap

We can adjust the shape of the **line** ends to give them a better appearance by adding line caps.

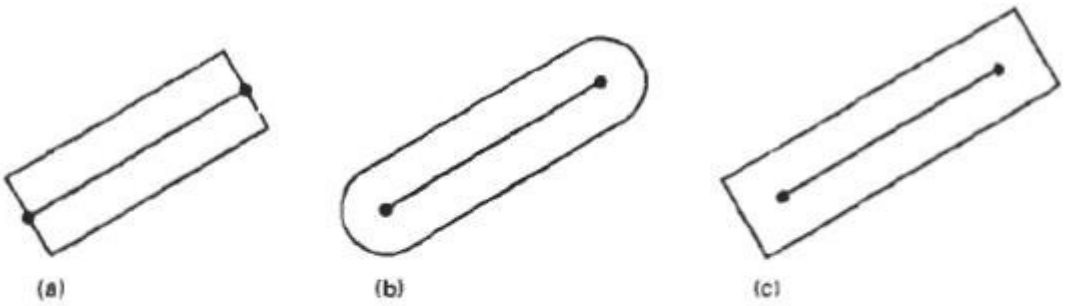
There are three types of line cap. They are

- Butt cap
- Round cap
- Projecting square cap

Butt cap obtained by adjusting the end positions of the component parallel **lines** so that the thick line is displayed with square ends that are perpendicular to the line path.

Round cap obtained by adding a filled semicircle to each butt cap. The circular arcs are centered on the line endpoints and have a diameter equal to the line thickness

Projecting square cap extend the line and add butt caps that are positioned one-half of the line width beyond the specified endpoints.



Thick lines drawn with (a) butt caps, (b) round caps, and (c) projecting square caps.

Color and Grayscale Levels

Various color and intensity-level options can be made available to a user, depending on the capabilities and design objectives of a particular system

In a color raster system, the number of color choices available depends on the amount of storage provided per pixel in the frame buffer

Color-information can be stored in the frame buffer in two ways:

- We can store color codes directly in the frame buffer
- We can put the color codes in a separate table and use pixel values as an index into this table

With the direct storage scheme, whenever a particular color code is specified in an application program, the corresponding binary value is placed in the frame buffer for each-component pixel in the output primitives to be displayed in that color.

A minimum number of colors can be provided in this scheme with 3 bits of storage per

pixel, as shown in Table

THE EIGHT COLOR CODES FOR A THREE-BIT PER PIXEL FRAME BUFFER

Color	Stored Color Values in Frame Buffer			Displayed Color
Code	RED	GREEN	BLUE	
0	0	0	0	Black
1	0	0	1	Blue
2	0	1	0	Green
3	0	1	1	Cyan
4	1	0	0	Red
5	1	0	1	Magenta
6	1	1	0	Yellow
7	1	1	1	White

A user can set color-table entries in a PHIGS applications program with the function

setColourRepresentation (ws, ci, colorptr)

Parameter **ws** identifies the workstation output device; parameter ci specifies the color index, which is the color-table position number **(0** to **255**) and parameter colorptr points to a trio of RGB color values **(r, g, b)** each specified in the range from **0** to 1

Grayscale

With monitors that have no color capability, color functions can be used in an application program to set the shades of gray, or grayscale, for displayed primitives. Numeric values over the range from 0 to 1 can be used to specify grayscale levels, which are then converted to appropriate binary codes for storage in the raster.

Area fill Attributes

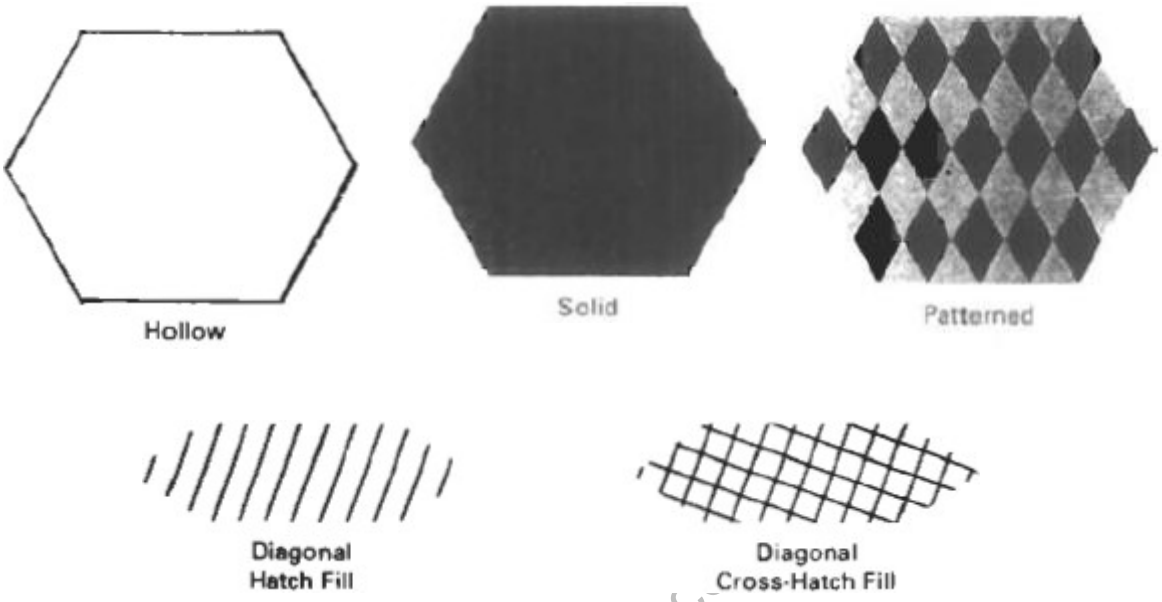
Options for filling a defined region include a choice between a solid color or a pattern fill and choices for particular colors and patterns

Fill Styles

Areas are displayed with three basic fill styles: hollow with a color border, filled with a solid color, or filled with a specified pattern or design. A basic fill style is selected in a **PHIGS** program with the function

setInteriorStyle(fs)

Values for the fill-style parameter fs include hollow, solid, and pattern. Another value for fill style is hatch, which is used to fill an area with selected hatching patterns-parallel lines or crossed lines



The color for a solid interior or for a hollow area outline is chosen with where fill color parameter fc is set to the desired color code

setInteriorColourIndex(fc)

Character Attributes

The appearance of displayed character is controlled by attributes such as font, size, color and orientation. Attributes can be set both for entire character strings (text) and for individual characters defined as marker symbols

Text Attributes

The choice of font or type face is set of characters with a particular design style as courier, Helvetica, times roman, and various symbol groups.

The characters in a selected font also be displayed with styles. (solid, dotted, double) in **bold face** in *italics*, and in outline or shadow styles.

A particular font and associated stvle is selected in a PHIGS program by setting an integer code for the text font parameter tf in the function

setTextFont(tf)

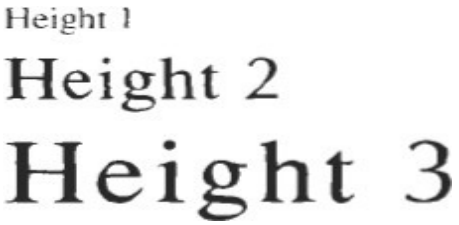
Control of text color (or intensity) is managed from an application program with

setTextColourIndex(tc)

where text color parameter tc specifies an allowable color code.

Text size can be adjusted without changing the width to height ratio of characters with

SetCharacterHeight (ch)



Bundled Attributes

The procedures considered so far each function reference a single attribute that specifies exactly how a primitive is to be displayed these specifications are called individual attributes.

A particular set of attributes values for a primitive on each output device is chosen by specifying appropriate table index. Attributes specified in this manner are called bundled attributes. The choice between a bundled or an unbundled specification is made by setting a switch called the aspect source flag for each of these attributes

setIndividualASF(attributeptr, flagptr)

where parameter attributer ptr points to a list of attributes and parameter flagptr points to the corresponding list of aspect source flags. Each aspect source flag can be assigned a value of individual or bundled

Bundled line attributes

Entries in the bundle table for line attributes on a specified workstation are set with the function

setPolylineRepresentation (ws, li, lt, lw, lc)

Parameter ws is the workstation identifier and line index parameter li defines the bundle table position. Parameter lt, lw, tc are then bundled and assigned values to set the line type, line width, and line color specifications for designated table index.

Example

setPolylineRepresentation(1,3,2,0.5,1)
setPolylineRepresentation (4,3,1,1,7)

A poly line that is assigned a table index value of 3 would be displayed using dashed lines at half thickness in a blue color on work station 1; while on workstation 4, this same index generates solid, standard-sized white lines

SUMMARY OF ATTRIBUTES

Output Primitive Type	*Associated Attributes*	*Attribute-Setting Functions*	*Bundled-Attribute Functions*
Line	Type	`setLinetype`	`setPolylineIndex`
	Width	`setLineWidthScaleFactor`	`setPolylineRepresentation`
	Color	`setPolylineColourIndex`	
Fill Area	Fill Style	`setInteriorStyle`	`setInteriorIndex`
	Fill Color	`setInteriorColorIndex`	`setInteriorRepresentation`
	Pattern	`setInteriorStyleIndex`	
		`setPatternRepresentation`	
		`setPatternSize`	
		`setPatternReferencePoint`	
Text	Font	`setTextFont`	`setTextIndex`
	Color	`setTextColourIndex`	`setTextRepresentation`
	Size	`setCharacterHeight`	
		`setCharacterExpansionFactor`	
	Orientation	`setCharacterUpVector`	
		`setTextPath`	
		`setTextAlignment`	
Marker	Type	`setMarkerType`	`setPolymarkerIndex`
	Size	`setMarkerSizeScaleFactor`	`setPolymarkerRepresentation`
	Color	`setPolymarkerColourIndex`	

Two Dimensional Geometric Transformations

Changes in orientations, size and shape are accomplished with geometric transformations that alter the coordinate description of objects.

Basic transformation

- Translation
  - T(t_x, t_y)
  - Translation distances
- Scale
  - S(s_x,s_y)
  - Scale factors
- Rotation
  - R(θ)
  - Rotation angle

Translation

A translation is applied to an object by representing it along a straight line path from one coordinate location to another adding translation distances, tx, ty to original coordinate position (x,y) to move the point to a new position (x',y') to

x' = x + tx, y' = y + ty

The translation distance point (tx,ty) is called translation vector or shift vector.

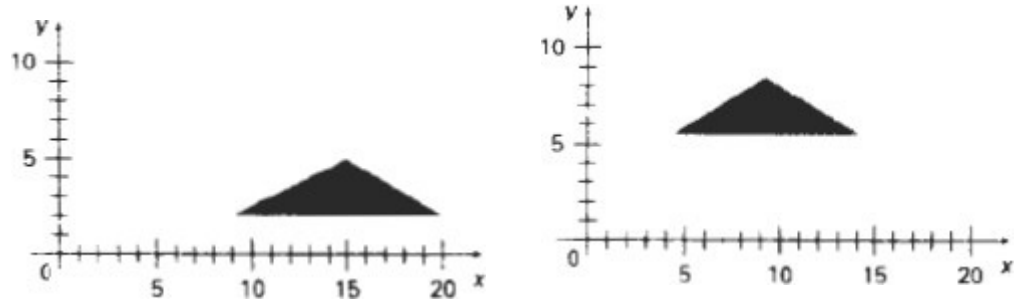
Translation equation can be expressed as single matrix equation by using column vectors to represent the coordinate position and the translation vector as

$$P = (x, y)$$
$$T = (t_x, t_y)$$

$$x' = x + t_x$$
$$y' = y + t_y$$

$$\begin{bmatrix} x^1 \\ y^1 \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$P' = P + T$$

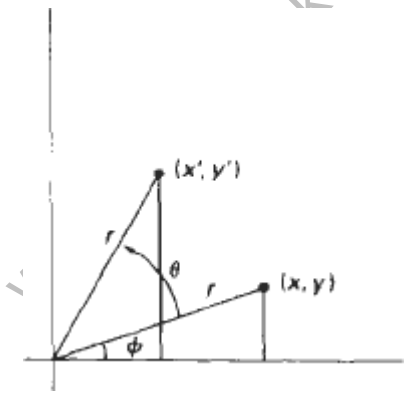


Moving a polygon from one position to another position with the translation vector (-5.5, 3.75)

Rotations:

A two-dimensional rotation is applied to an object by repositioning it along a circular path on xy plane. To generate a rotation, specify a rotation angle θ and the position (x_r,y_r) of the rotation point (pivot point) about which the object is to be rotated.

Positive values for the rotation angle define counter clock wise rotation about pivot point. Negative value of angle rotates objects in clock wise direction. The transformation can also be described as a rotation about a rotation axis perpendicular to xy plane and passes through pivot point



Rotation of a point from position (x,y) to position (x',y') through angle θ relative to coordinateorigin

The transformation equations for rotation of a point position P when the pivot point is at coordinate origin. In figure r is constant distance of the point positions Φ is the original angular of the point from horizontal and θ is the rotation angle.

The transformed coordinates in terms of angle θ and Φ

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

$$y' = r\sin(\theta+\Phi) = r\sin\theta\cos\Phi + r\cos\theta\sin\Phi$$

The original coordinates of the point in polar coordinates

$$x = r\cos\Phi, \quad y = r\sin\Phi$$

the transformation equation for rotating a point at position (x,y) through an angle θ about origin

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

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

Rotation equation

$$P' = R \cdot P$$

Rotation Matrix

$$R = \begin{pmatrix} \cos\theta & \sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$$

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

Note : Positive values for the rotation angle define counterclockwise rotations about the rotation point and negative values rotate objects in the clockwise.

Scaling

A scaling transformation alters the size of an object. This operation can be carried out for polygons by multiplying the coordinate values (x,y) to each vertex by scaling factor Sx & Sy to produce the transformed coordinates (x',y')

x'= x.Sx **y' = y.Sy**

scaling factor Sx scales object in x direction while Sy scales in y direction.

The transformation equation in matrix form

$$\begin{pmatrix} x^1 \\ y^1 \end{pmatrix} = \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix}$$

or

P' = S. P

Where S is 2 by 2 scaling matrix

Turning a square (a) Into a rectangle (b) with scaling factors $s_x = 2$ and $s_y = 1$.

Any positive numeric values are valid for scaling factors sx and sy. Values less than 1 reduce the size of the objects and values greater than 1 produce an enlarged object.
There are two types of Scaling. They are

Uniform scaling
Non Uniform Scaling

To get uniform scaling it is necessary to assign same value for s_x and s_y. Unequal values for s_x and s_y result in a non uniform scaling

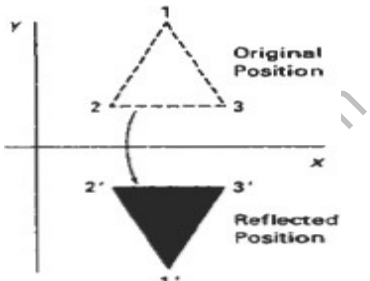
Other Transformations

1. Reflection
2. Shear

Reflection

A reflection is a transformation that produces a mirror image of an object. The mirror image for a two-dimensional reflection is generated relative to an axis of reflection by rotating the object 180° about the reflection axis. We can choose an axis of reflection in the xy plane or perpendicular to the xy plane or coordinate origin

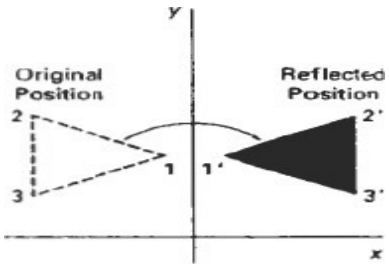
Reflection of an object about the x axis



Reflection the x axis is accomplished with the transformation matrix

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

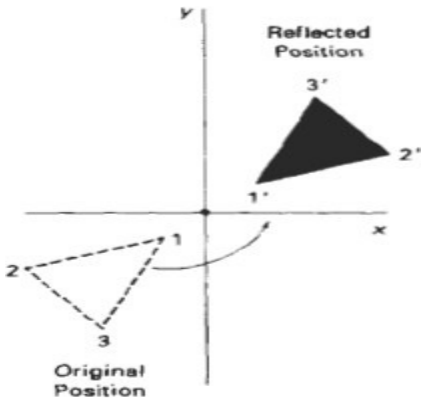
Reflection of an object about the y axis



Reflection the y axis is accomplished with the transformation matrix

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

Reflection of an object about the coordinate origin



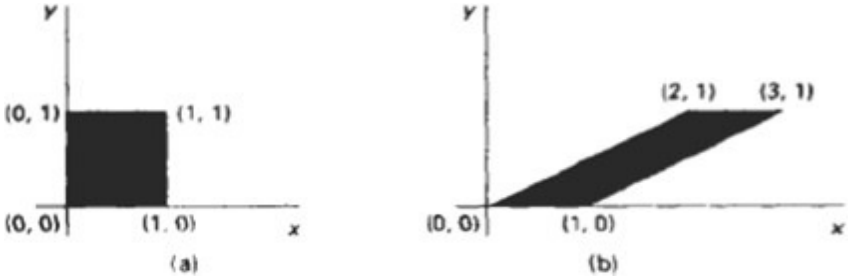
Reflection about origin is accomplished with the transformation matrix

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

Shear

A Transformation that slants the shape of an object is called the shear transformation. Two common shearing transformations are used. One shifts x coordinate values and other shift y coordinate values. However in both the cases only one coordinate (x or y) changes its coordinates and other preserves its values.

The x shear preserves the y coordinates, but changes the x values which cause vertical lines to tilt right or left as shown in figure



The Transformations matrix for x-shear is

$$\begin{pmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

which transforms the coordinates as

$x' = x + sh_x . y$

$y' = y$

Y Shear

The y shear preserves the x coordinates, but changes the y values which cause horizontal lines which slope up or down

The Transformations matrix for y-shear is

$$\begin{pmatrix} 1 & 0 & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Which transforms the coordinates

as $x' = x$

$y' = y + sh_y . x$

The transformation matrix for xy-shear

$$\begin{bmatrix} X^1 \\ Y^1 \\ 1 \end{bmatrix} = \begin{pmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{bmatrix} x \\ Y \\ 1 \end{bmatrix}$$

which transforms the coordinates as

$x' = x + sh_x . y$

$y' = y + sh_y . x$

Shearing Relative to other reference line

We can apply x shear and y shear transformations relative to other reference lines. In x shear transformations we can use y reference line and in y shear we can use x reference line.

X shear with y reference line

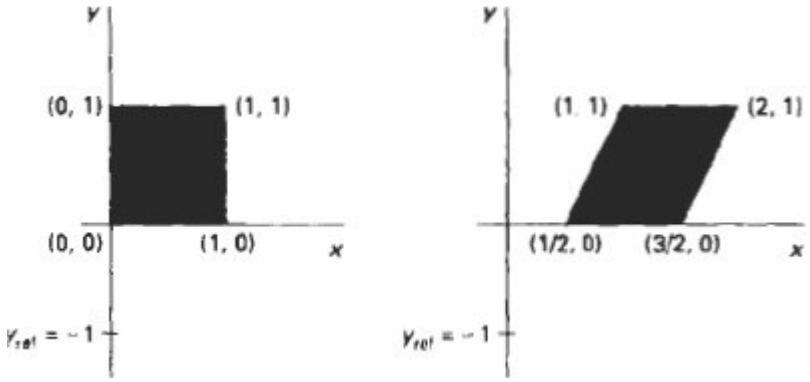
We can generate x-direction shears relative to other reference lines with the transformation matrix

$$\begin{pmatrix} 1 & sh_x & sh_x . y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Which transforms the coordinates as follows

$x' = x + sh_x (y - y_{ref})$

$y' = y$



Y shear with x reference line

We can generate y-direction shears relative to other reference lines with the transformation matrix

$$\begin{bmatrix} 1 & sh_x & -sh_x.y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

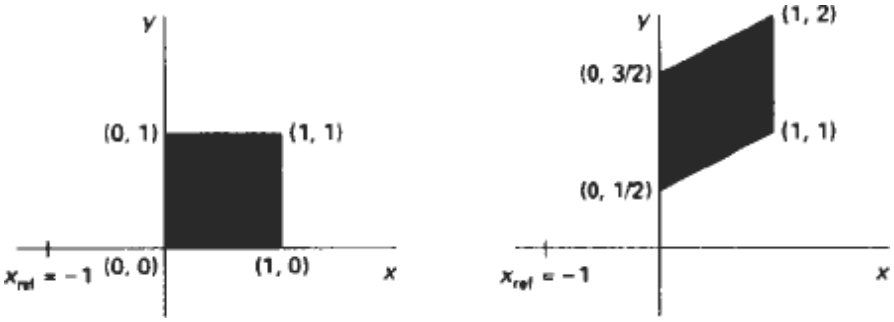
which transforms the coordinates as

$x' = x$

$y' = sh_y (x - x_{ref}) + y$

Example

$Sh_y = ½$ and $x_{ref}=-1$



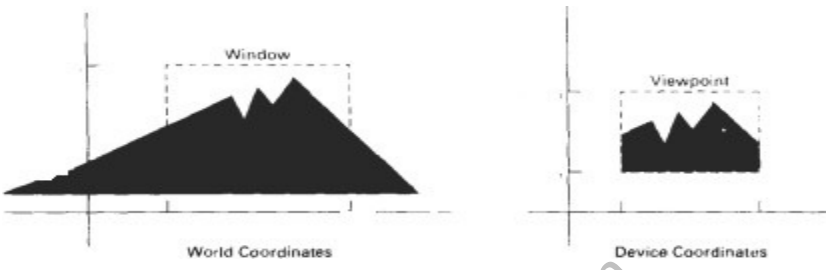
Two dimensional viewing

The viewing pipeline

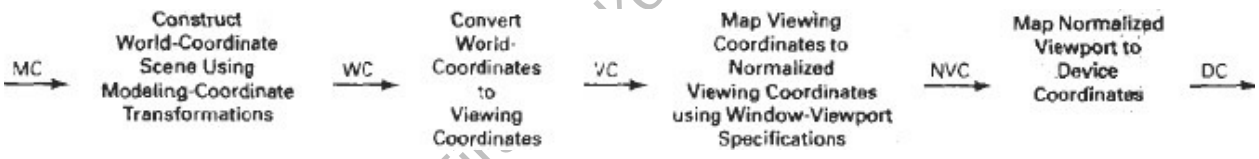
A world coordinate area selected for display is called a window. An area on a display device to which a window mapped is called a view port. The window defines what is to be viewed the view port defines where it is to be displayed.

The mapping of a part of a world coordinate scene to device coordinate is referred to as viewing transformation. The two dimensional viewing transformation is referred to as window to view port transformation of windowing transformation.

A viewing transformation using standard rectangles for the window and viewport



The two dimensional viewing transformation pipeline

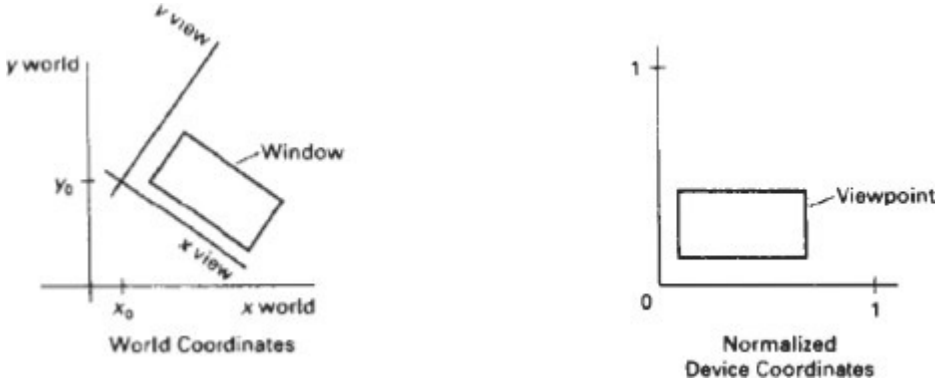


The viewing transformation in several steps as indicated in Fig. First, we construct the scene in world coordinates using the output primitives. Next to obtain a particular orientation for the window, we can set up a two-dimensional viewing- coordinate system in the world coordinate plane, and define a window in the viewing- coordinate system.

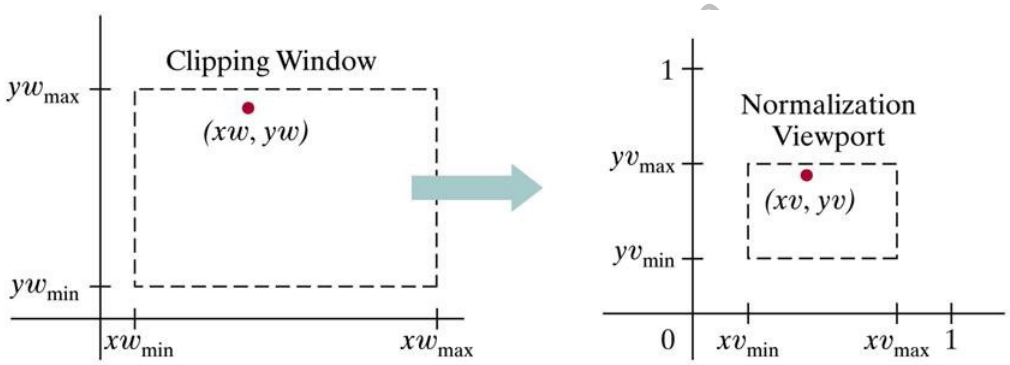
The viewing- coordinate reference frame is used to provide a method for setting up arbitrary orientations for rectangular windows. Once the viewing reference frame is established, we can transform descriptions in world coordinates to viewing coordinates.

We then define a viewport in normalized coordinates (in the range from 0 to 1) and map the viewing-coordinate description of the scene to normalized coordinates.

At the final step all parts of the picture that lie outside the viewport are clipped, and the contents of the viewport are transferred to device coordinates. By changing the position of the viewport, we can view objects at different positions on the display area of an output device.



Window to view port coordinate transformation:



A point (xw, yw) in a world-coordinate clipping window is mapped to viewport coordinates (xv, yv), within a unit square, so that the relative positions of the two points in their respective rectangles are the same.

A point at position (x_w,y_w) in a designated window is mapped to viewport coordinates (x_v,y_v) so that relative positions in the two areas are the same. The figure illustrates the window to view port mapping.

A point at position (x_w,y_w) in the window is mapped into position (x_v,y_v) in the associated view port. To maintain the same relative placement in view port as in window

$$\frac{xv - xv_{min}}{xv_{max} - xv_{min}} = \frac{xw - xw_{min}}{xw_{max} - xw_{min}}$$

$$\frac{yv - yv_{min}}{yv_{max} - yv_{min}} = \frac{yw - yw_{min}}{yw_{max} - yw_{min}}$$

solving these expressions for view port position (x_v,y_v)

$$xv = xv_{min} + (xw - xw_{min})\frac{(xv_{max} - xv_{min})}{xw_{max} - xw_{min}}$$

$$yv = yv_{min} + (yw - yw_{min})\frac{(yv_{max} - yv_{min})}{yw_{max} - yw_{min}}$$

where scaling factors are

$$sx = \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}} \qquad sy = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}}$$

The conversion is performed with the following sequence of transformations.

1. Perform a scaling transformation using point position of (x_w min, y_w min) that scales the window area to the size of view port.
2. Translate the scaled window area to the position of view port. Relative proportions of objects are maintained if scaling factor are the same(Sx=Sy).

Otherwise world objects will be stretched or contracted in either the x or y direction when displayed on output device. For normalized coordinates, object descriptions are mapped to various display devices.

Any number of output devices can be open in particular application and another window view port transformation can be performed for each open output device. This mapping called the work station transformation is accomplished by selecting a window area in normalized apace and a view port are in coordinates of display device.

Two Dimensional viewing functions

Viewing reference system in a PHIGS application program has following function.

evaluateViewOrientationMatrix(x_0,y_0,x_v,y_v,error, viewMatrix)

where x_0,y_0 are coordinate of viewing origin and parameter x_v, y_v are the world coordinate positions for view up vector.An integer error code is generated if the input parameters are in error otherwise the view matrix for world-to-viewing transformation is calculated. Any number of viewing transformation matrices can be defined in an application.

To set up elements of window to view port mapping

evaluateViewMappingMatrix ($x_w$min, $x_w$max, $y_w$min, $y_w$max, $x_v$min, $x_v$max, $y_v$min, $y_v$max, error, viewMappingMatrix)

Here window limits in viewing coordinates are chosen with parameters $x_w$min, $x_w$max, $y_w$min, $y_w$max and the viewport limits are set with normalized coordinate positions $x_v$min, $x_v$max, $y_v$min, $y_v$max.
The combinations of viewing and window view port mapping for various workstations in a viewing table with

setViewRepresentation(ws,viewIndex,viewMatrix,viewMappingMatrix, xclipmin, xclipmax, yclipmin, yclipmax, clipxy)

Where parameter ws designate the output device and parameter view index sets an integer identifier for this window-view port point. The matrices viewMatrix and viewMappingMatrix can be concatenated and referenced by viewIndex.

setViewIndex(viewIndex)

selects a particular set of options from the viewing table.

At the final stage we apply a workstation transformation by selecting a work station window viewport pair.

setWorkstationWindow (ws, xwsWindmin, xwsWindmax,
ywsWindmin, ywsWindmax)

setWorkstationViewport (ws, xwsVPortmin, xwsVPortmax,
ywsVPortmin, ywsVPortmax)

where was gives the workstation number. Window-coordinate extents are specified in the range from 0 to 1 and viewport limits are in integer device coordinates.

Clipping operation

Any procedure that identifies those portions of a picture that are inside or outside of a specified region of space is referred to as **clipping algorithm or clipping**. The region against which an object is to be clipped is called **clip window**.

Algorithm for clipping primitive types:

Line clipping (Straight-line segment)
Curve clipping
Text clipping

.

Line Clipping

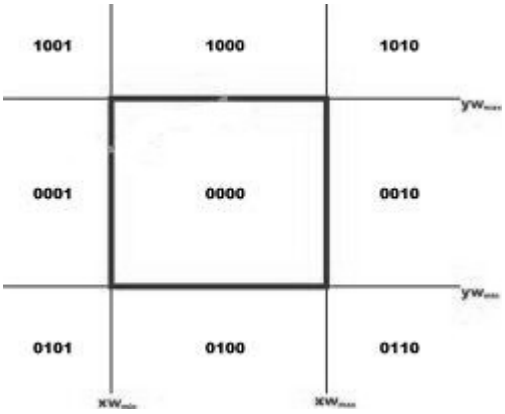
A line clipping procedure involves several parts. First we test a given line segment whether it lies completely inside the clipping window. If it does not we try to determine whether it lies completely outside the window. Finally if we can not identify a line as completely inside or completely outside, we perform intersection calculations with one or more clipping boundaries.

Process lines through "inside-outside" tests by checking the line endpoints. A line with both endpoints inside all clipping boundaries such as line from P1 to P2 is saved. A line with both end points outside any one of the clip boundaries line P3P4 is outside the window.

Cohen-Sutherland Line Clipping

This is one of the oldest and most popular line-clipping procedures. The method speeds up the processing of line segments by performing initial tests that reduce the number of intersections that must be calculated.

Every line endpoint in a picture is assigned a four digit binary code called a **region code** that identifies the location of the point relative to the boundaries of the clipping rectangle.



Binary region codes assigned to line end points according to relative position with respect to the clipping rectangle.

Regions are set up in reference to the boundaries. Each bit position in region code is used to indicate one of four relative coordinate positions of points with respect to clip window: to the left, right, top or bottom. By numbering the bit positions in the region code as 1 through 4 from right to left, the coordinate regions are corrected with bit positions as

bit 1: left

bit 2: right

bit 3: below

bit4: above

A value of 1 in any bit position indicates that the point is in that relative position. Otherwise the bit position is set to 0. If a point is within the clipping rectangle the region code is 0000. A point that is below and to the left of the rectangle has a region code of 0101.

Bit values in the region code are determined by comparing endpoint coordinate values (x,y) to clip boundaries. Bit1 is set to 1 if $x < xw_{min}$.

For programming language in which bit manipulation is possible region-code bit values can be determined with following two steps.

(1) Calculate differences between endpoint coordinates and clipping boundaries.

(2) Use the resultant sign bit of each difference calculation to set the corresponding value

in the region code.

(3) bit 1 is the sign bit of $x - xw_{min}$

bit 2 is the sign bit of $xw_{max} - x$ bit

3 is the sign bit of $y - yw_{min}$ bit 4

is the sign bit of $yw_{max} - y$.

(4) Once we have established region codes for all line endpoints, we can quickly determine which lines are completely inside the clip window and which are clearly outside.

(5) Any lines that are completely contained within the window boundaries have a region code of 0000 for both endpoints, and we accept

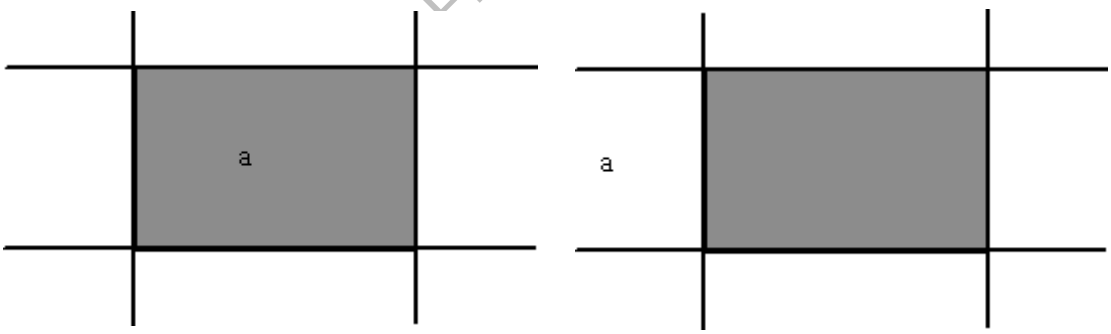
Nicholl-Lee-Nicholl Line clipping

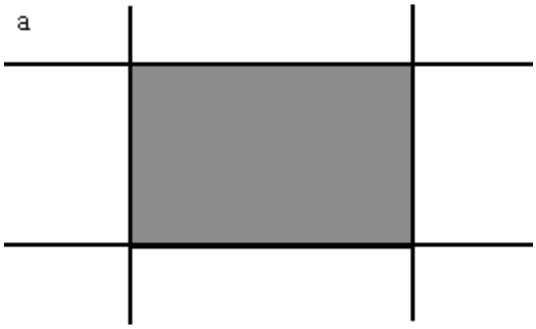
By creating more regions around the clip window, the Nicholl-Lee-Nicholl (or NLN) algorithm avoids multiple clipping of an individual line segment. In the Cohen-Sutherland method, multiple intersections may be calculated.These extra intersection calculations are eliminated in the NLN algorithm by carrying out more region testing before intersection positions are calculated.

Compared to both the Cohen-Sutherland and the Liang-Barsky algorithms, the Nicholl-Lee-Nicholl algorithm performs fewer comparisons and divisions. The trade-off is that the NLN algorithm can only be applied to two-dimensional dipping, whereas both the Liang-Barsky and the Cohen-Sutherland methods are easily extended to three-dimensional scenes.

For a line with endpoints P1 and P2 we first determine the position of point P1, for the nine possible regions relative to the clipping rectangle. Only the three regions shown in Fig. need to be considered. If P1 lies in any one of the other six regions, we can move it to one of the three regions in Fig. using a symmetry transformation. For example, the region directly above the clip window can be transformed to the region left of the clip window using a reflection about the line y = ***-x,*** or we could use a **90 degree** counterclockwise rotation.

Three possible positions for a line endpoint p1(a) in the NLN algorithm





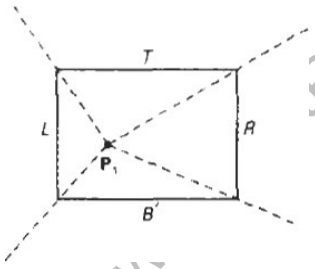
Case 1: p1 inside region

Case 2: p1 across edge

Case 3: p1 across corner

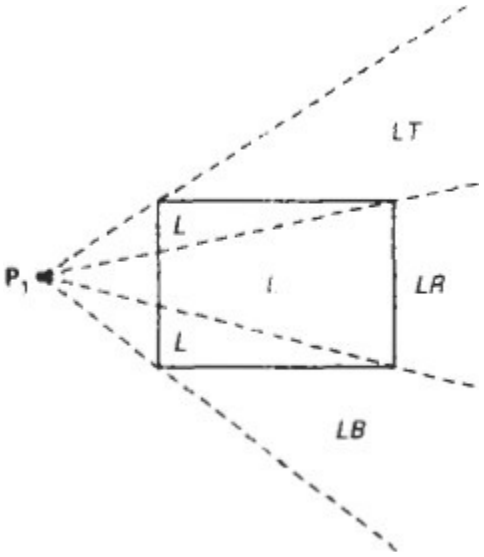
Next, we determine the position of P2 relative to P1**.** To do this, we create some new regions in the plane, depending on the location of P1. Boundaries of the new regions are half-infinite line segments that start at the position of P1 and pass through the window corners. If P1 is inside the clip window and P2 is outside, we set up the four regions shown in Fig

The four clipping regions used in NLN alg when p1 is inside and p2 outside the clip window



The intersection with the appropriate window boundary is then carried out, depending on which one of the four regions (L, T**,** R, or B) contains P2. If both P1 and P2 are inside the clipping rectangle, we simply save the entire line.

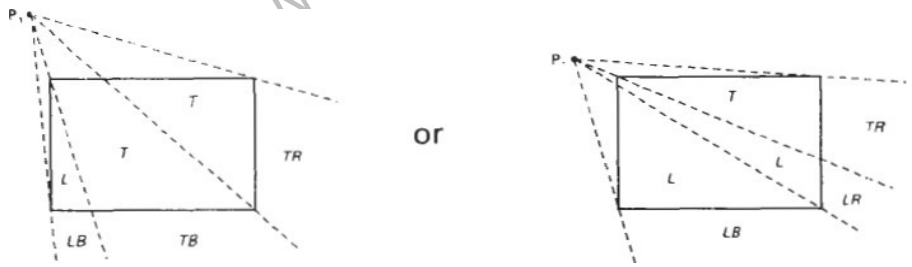
If P1 is in the region to the left of the window, we set up the four regions, ***L, LT, LR,*** and ***LB,*** shown in Fig.



These four regions determine a unique boundary for the line segment. For instance, if P2 is in region L, we clip the line at the left boundary and save the line segment from this intersection point to P2**.** But if P2 is in region **LT,** we save the line segment from the left window boundary to the top boundary. If P2 is not in any of the four regions, L, LT**,** LR, or LB**,** the entire line is clipped.

For the third case, when P1 is to the left and above the clip window, we usethe clipping regions in Fig.

Fig : The two possible sets of clipping regions used in NLN algorithm when P1 is above and to the left of the clip window



In this case, we have the two possibilities shown, depending on the position of P1, relative to the top left corner of the window. If P2**,** is in one of the regions T, L, TR, TB**,** LR, or LB, this determines a unique clip window edge for the intersection calculations. Otherwise, the entire line is rejected.

To determine the region in which P2 is located, we compare the slope of the

line to the slopes of the boundaries of the clip regions. For example, if P1 is left of the clipping rectangle (Fig. a), then P2, is in region LT if

$\text{slope}\overline{P_1P_{TR}} < \text{slope}\overline{P_1P_2} < \text{slope}\overline{P_1P_{TL}}$

or

$$\frac{y_T - y_1}{x_R - x_1} < \frac{y_2 - y_1}{x_2 - x_1} < \frac{y_T - y_1}{x_L - x_1}$$

And we clip the entire line if

$$(y_T - y_1)(x_2 - x_1) < (x_L - x_1)(y_2 - y_1)$$

The coordinate difference and product calculations used in the slope tests are saved and also used in the intersection calculations. From the parametric equations

$$x = x_1 + (x_2 - x_1)u$$

$$y = y_1 + (y_2 - y_1)u$$

an x-intersection position on the left window boundary is $\mathbf{x = x_L}$,, with

$u = (x_L - x_1)/(x_2 - x_1)$ so that the y-intersection position is

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x_L - x_1)$$

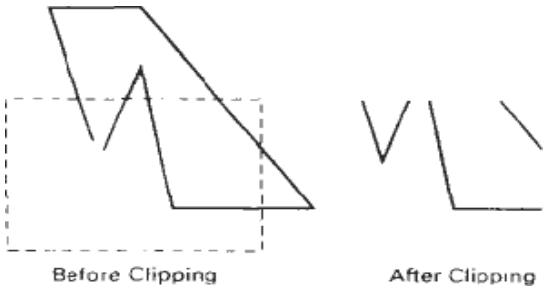
And an intersection position on the top boundary has $y = \mathbf{y_T}$ and $u = (y_T - y_1)/(y_2 - y_1)$ with

$$x = x_1 + \frac{x_2 - x_1}{y_2 - y_1}(y_T - y_1)$$

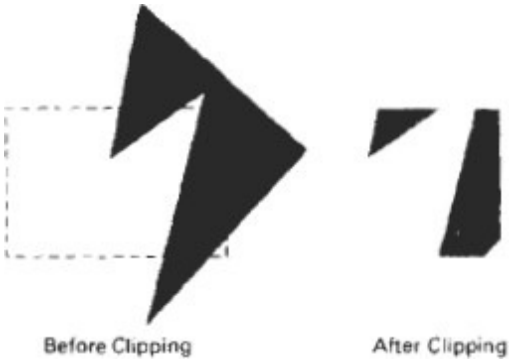
POLYGON CLIPPING

To clip polygons, we need to modify the line-clipping procedures. A polygon boundary processed with a line clipper may be displayed as a series of unconnected line segments (Fig.), depending on the orientation of the polygon to the clipping window.

Display of a polygon processed by a line clipping algorithm



For polygon clipping, we require an algorithm that will generate one or more closed areas that are then scan converted for the appropriate area fill. The output of a polygon clipper should be a sequence of vertices that defines the clipped polygon boundaries.



Sutherland – Hodgeman polygon clipping:

A polygon can be clipped by processing the polygon boundary as a whole against each window edge. This could be accomplished by processing all polygon vertices against each clip rectangle boundary.

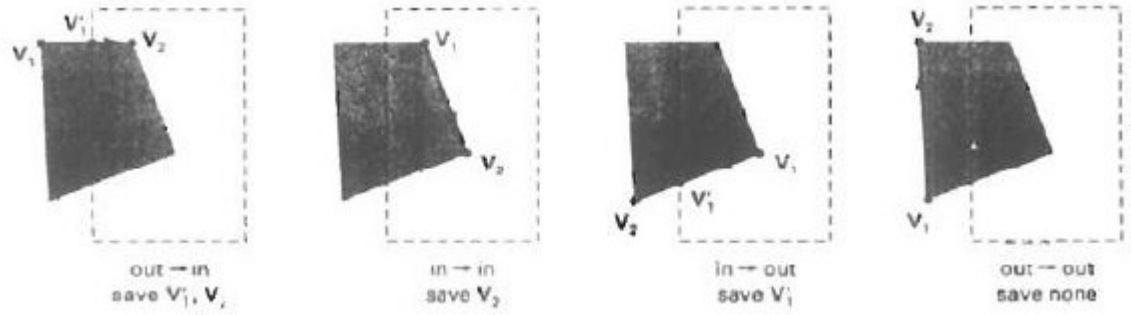
There are four possible cases when processing vertices in sequence around the perimeter of a polygon. As each point of adjacent polygon vertices is passed to a window boundary clipper, make the following tests:

1. If the first vertex is outside the window boundary and second vertex is inside, both the intersection point of the polygon edge with window boundary and second vertex are added to output vertex list.
2. If both input vertices are inside the window boundary, only the second vertex is added to the output vertex list.

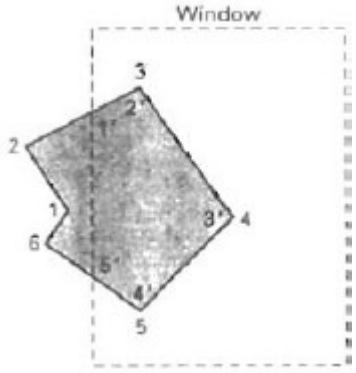
3. If first vertex is inside the window boundary and second vertex is outside only the edge intersection with window boundary is added to output vertex list.
4. If both input vertices are outside the window boundary nothing is added to the output list.

Clipping a polygon against successive window boundaries.

Successive processing of pairs of polygon vertices against the left window boundary



Clipping a polygon against the left boundary of a window, starting with vertex 1. Primed numbers are used to label the points in the output vertex list for this window boundary.

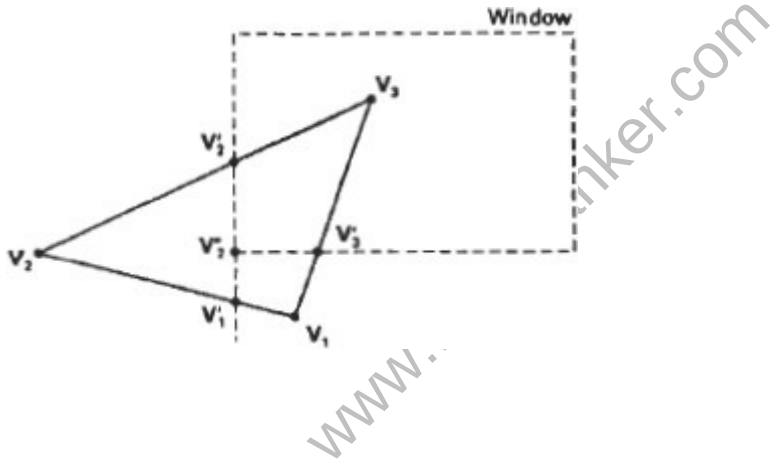


vertices 1 and 2 are found to be on outside of boundary. Moving along vertex 3 which is inside, calculate the intersection and save both the intersection point and vertex 3. Vertex 4 and 5 are determined to be inside and are saved. Vertex 6 is outside so we find and save the intersection point. Using the five saved points we repeat the process for next window boundary.

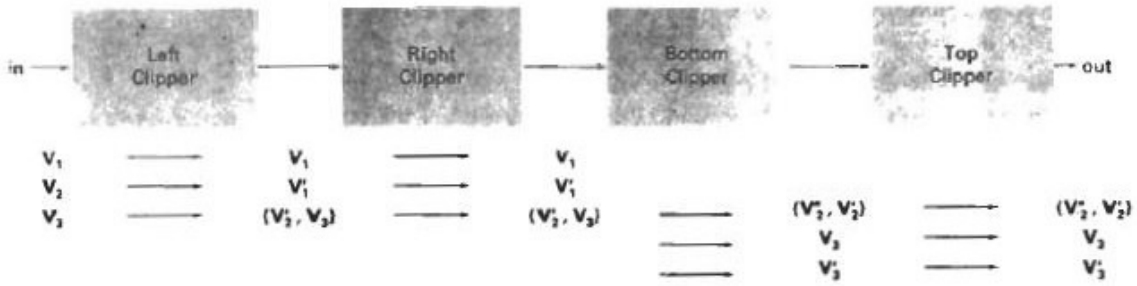
Implementing the algorithm as described requires setting up storage for an output list of vertices as a polygon clipped against each window boundary. We eliminate the intermediate output vertex lists by simply by clipping individual vertices at each step and passing the clipped vertices on to the next boundary clipper.

A point is added to the output vertex list only after it has been determined to be inside or on a window boundary by all boundary clippers. Otherwise the point does not continue in the pipeline.

A polygon overlapping a rectangular clip window



Processing the vertices of the polygon in the above fig. through a boundary clipping pipeline. After all vertices are processed through the pipeline, the vertex list is { v2", v2', v3,v3'}



Implementation of Sutherland-Hodgeman Polygon Clipping

```
typedef enum { Left,Right,Bottom,Top } Edge;
#define N_EDGE 4
#define TRUE 1
#define FALSE 0

int inside(wcPt2 p, Edge b,dcPt wmin,dcPt wmax)
{
switch(b)
{
case Left: if(p.x<wmin.x) return (FALSE); break;
case Right:if(p.x>wmax.x) return (FALSE); break;
case bottom:if(p.y<wmin.y) return (FALSE); break;
case top: if(p.y>wmax.y) return (FALSE); break;
}
return (TRUE);
}
int cross(wcPt2 p1, wcPt2 p2,Edge b,dcPt wmin,dcPt wmax)
{
if(inside(p1,b,wmin,wmax)==inside(p2,b,wmin,wmax))
return (FALSE);
else
return (TRUE);
}
wcPt2 (wcPt2 p1, wcPt2 p2,int b,dcPt wmin,dcPt wmax )
{
wcPt2 iPt;
float m;
if(p1.x!=p2.x)
m=(p1.y-p2.y)/(p1.x-p2.x);
switch(b)
{
case Left:
ipt.x=wmin.x;
```

```
ipt.y=p2.y+(wmin.x-p2.x)*m;
break;
case Right:
ipt.x=wmax.x;
ipt.y=p2.y+(wmax.x-p2.x)*m;
break;
case Bottom:
ipt.y=wmin.y;
if(p1.x!=p2.x)
ipt.x=p2.x+(wmin.y-p2.y)/m;
else
ipt.x=p2.x;
break;
case Top:
ipt.y=wmax.y;
if(p1.x!=p2.x)
ipt.x=p2.x+(wmax.y-p2.y)/m;
else
ipt.x=p2.x;
break;
}
return(ipt);
}
void clippoint(wcPt2 p,Edge b,dcPt wmin,dcPt wmax, wcPt2 *pout,int *cnt, wcPt2
*first[],struct point *s)
{
wcPt2 iPt;
if(!first[b])
first[b]=&p;
else
if(cross(p,s[b],b,wmin,wmax))
{
ipt=intersect(p,s[b],b,wmin,wmax);
if(b<top)
clippoint(ipt,b+1,wmin,wmax,pout,cnt,first,s);
else
{
pout[*cnt]=ipt;
(*cnt)++;
}
}
s[b]=p;
if(inside(p,b,wmin,wmax))
```

```
if(b<top)
clippoint(p,b+1,wmin,wmax,pout,cnt,first,s);
else
{
pout[*cnt]=p;
(*cnt)++;
}
}
void closeclip(dcPt wmin,dcPt wmax, wcPt2 *pout,int *cnt,wcPt2 *first[], wcPt2 *s)
{
wcPt2 iPt;
Edge b;
for(b=left;b<=top;b++)
{
if(cross(s[b],*first[b],b,wmin,wmax))
{
i=intersect(s[b],*first[b],b,wmin,wmax);
if(b<top)
clippoint(i,b+1,wmin,wmax,pout,cnt,first,s);
else
{
pout[*cnt]=i;
(*cnt)++;
}
}
}
}
int clippolygon(dcPt point wmin,dcPt wmax,int n,wcPt2 *pin, wcPt2 *pout)
{
wcPt2 *first[N_EDGE]={0,0,0,0},s[N_EDGE];
int i,cnt=0;
for(i=0;i<n;i++)
clippoint(pin[i],left,wmin,wmax,pout,&cnt,first,s);
closeclip(wmin,wmax,pout,&cnt,first,s);
return(cnt);
}
```

Curve Clipping

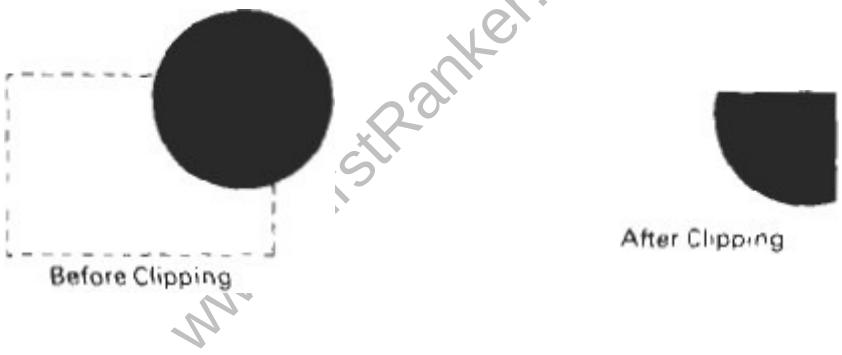
Curve-clipping procedures will involve nonlinear equations, and this requires more processing than for objects with linear boundaries. The bounding rectangle for a circle or other curved object can be used first to test for overlap with a rectangular clip window.

If the bounding rectangle for the object is completely inside the window, we save the object. If the rectangle is determined to be completely outside the window, we discard the object. In either case, there is no further computation necessary.

But if the bounding rectangle test fails, we can look for other computation-saving approaches. For a circle, we can use the coordinate extents of individual quadrants and then octants for preliminary testing before calculating curve-window intersections.

The below figure illustrates circle clipping against a rectangular window. On the first pass, we can clip the bounding rectangle of the object against the bounding rectangle of the clip region. If the two regions overlap, we will need to solve the simultaneous line-curve equations to obtain the clipping intersection points.

Clipping a filled circle

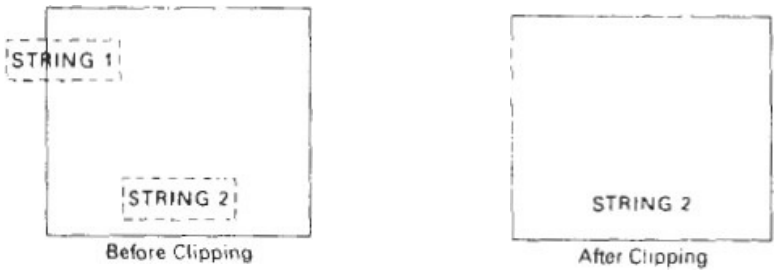


Text clipping

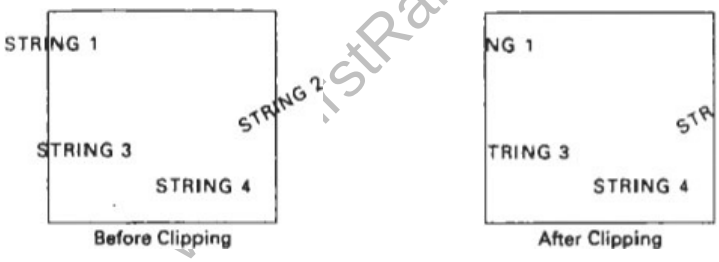
There are several techniques that can be used to provide text clipping in a graphics package. The clipping technique used will depend on the methods used to generate characters and the requirements of a particular application.

The simplest method for processing character strings relative to a window boundary is to use the **all-or-none string-clipping** strategy shown in Fig. . If all of the string is inside a clip window, we keep it. Otherwise, the string is discarded. This procedure is implemented by considering a bounding rectangle around the text pattern. The boundary positions of the rectangle are then compared to the window boundaries, and the string is rejected if there is any overlap. This method produces the fastest text clipping.

Text clipping using a bounding rectangle about the entire string

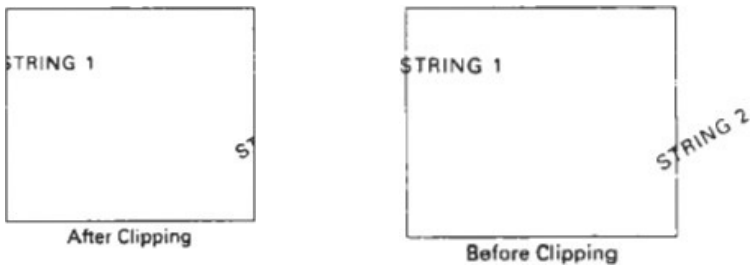


An alternative to rejecting an entire character string that overlaps a window boundary is to use the **all-or-none character-clipping** strategy. Here we discard only those characters that are not completely inside the window .In this case, the boundary limits of individual characters are compared to the window. Any character that either overlaps or is outside a window boundary is clipped.



A final method for handling text clipping is to clip the components of individual characters. We now treat characters in much the same way that we treated lines. If an individual character overlaps a clip window boundary, we clip off the parts of the character that are outside the window.

Text Clipping performed on the components of individual characters



Exterior clipping:

Procedure for clipping a picture to the interior of a region by eliminating everything outside the clipping region. By these procedures the inside region of the picture is saved. To clip a picture to the exterior of a specified region. The picture parts to be saved are those that are outside the region. This is called as exterior clipping.

Objects within a window are clipped to interior of window when other higher priority window overlaps these objects. The objects are also clipped to the exterior of overlapping windows.

UNIT - II THREE-DIMENSIONAL CONCEPTS

Parallel and Perspective projections-Three-Dimensional Object Representations – Polygons, Curved lines,Splines, Quadric Surfaces-Visualization of data sets- Three- Transformations – Three- Dimensional Viewing –Visible surface identification.

Three Dimensional Concepts

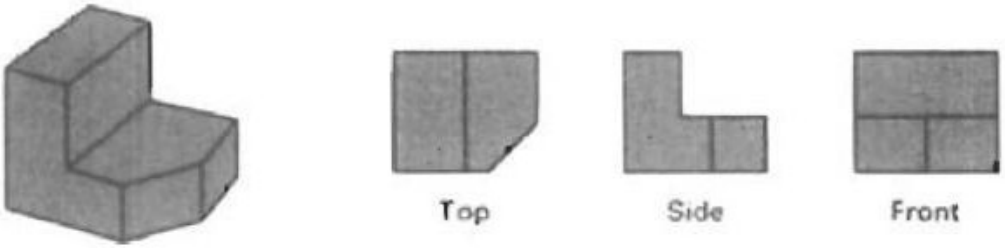
Three Dimensional Display Methods:

- To obtain a display of a three dimensional scene that has been modeled in world coordinates, we must setup a co-ordinate reference for the 'camera'.
- This coordinate reference defines the position and orientation for the plane of the camera film which is the plane we want to use to display a view of the objects in the scene.
- Object descriptions are then transferred to the camera reference coordinates and projected onto the selected display plane.
- The objects can be displayed in wire frame form, or we can apply lighting and surface rendering techniques to shade the visible surfaces.

Parallel Projection:

- Parallel projection is a method for generating a view of a solid object is to project points on the object surface along parallel lines onto the display plane.
- In parallel projection, parallel lines in the world coordinate scene project into parallel lines on the two dimensional display planes.
- This technique is used in engineering and architectural drawings to represent an object with a set of views that maintain relative proportions of the object.

Fig. Three parallel projection views of an object, showing relative proportions from different viewing positions.



Perspective Projection:

- It is a method for generating a view of a three dimensional scene is to project points to the display plane alone converging paths.
- This makes objects further from the viewing position be displayed smaller than objects of the same size that are nearer to the viewing position.
- In a perspective projection, parallel lines in a scene that are not parallel to the display plane are projected into converging lines.
- Scenes displayed using perspective projections appear more realistic, since this is the way that our eyes and a camera lens form images.

Depth Cueing:

- Depth information is important to identify the viewing direction, which is the front and which is the back of displayed object.
- Depth cueing is a method for indicating depth with wire frame displays is to vary the intensity of objects according to their distance from the viewing position.
- Depth cueing is applied by choosing maximum and minimum intensity (or color) values and a range of distance over which the intensities are to vary.

Visible line and surface identification:

- A simplest way to identify the visible line is to highlight the visible lines or to display them in a different color.
- Another method is to display the non visible lines as dashed lines.

Surface Rendering:

- Realism is attained in displays by setting the surface intensity of objects according to the lighting conditions in the scene and surface characteristics.
- Lighting conditions include the intensity and positions of light sources and the background illumination.
- Surface characteristics include degree of transparency and how rough or smooth the surfaces are to be.

Exploded and Cutaway views:

- Exploded and cutaway views of objects can be to show the internal structure and relationship of the objects parts.
- An alternative to exploding an object into its component parts is the cut away view which removes part of the visible surfaces to show internal structure.

Three-dimensional and Stereoscopic Views:

- In Stereoscopic views, three dimensional views can be obtained by reflecting a raster image from a vibrating flexible mirror.
- The vibrations of the mirror are synchronized with the display of the scene on the CRT.
- As the mirror vibrates, the focal length varies so that each point in the scene is projected to a position corresponding to its depth.
- Stereoscopic devices present two views of a scene; one for the left eye and the other for the right eye.
- The two views are generated by selecting viewing positions that corresponds to the two eye positions of a single viewer.
- These two views can be displayed on alternate refresh cycles of a raster monitor, and viewed through glasses that alternately darken first one lens then the other in synchronization with the monitor refresh cycles.

Three Dimensional Graphics Packages

- The 3D package must include methods for mapping scene descriptions onto a flat viewing surface.
- There should be some consideration on how surfaces of solid objects are to be modeled, how visible surfaces can be identified, how transformations of objects are preformed in space, and how to describe the additional spatial properties.
- World coordinate descriptions are extended to 3D, and users are provided with output and input routines accessed with specifications such as
    - Polyline3(n, WcPoints)

- Text3(WcPoint, string)
- Getlocator3(WcPoint)
- Translate3(translateVector, matrix Translate)

Where points and vectors are specified with 3 components and transformation matrices have 4 rows and 4 columns.

Three Dimensional Object Representations

Representation schemes for solid objects are divided into two categories as follows:

1. Boundary Representation (B-reps)

   It describes a three dimensional object as a set of surfaces that separate the object interior from the environment. Examples are polygon facets and spline patches.

2. Space Partitioning representation

   It describes the interior properties, by partitioning the spatial region containing an object into a set of small, nonoverlapping, contiguous solids (usually cubes).

   Eg: Octree Representation

Polygon Surfaces

Polygon surfaces are boundary representations for a 3D graphics object is a set of polygons that enclose the object interior.

Polygon Tables

- The polygon surface is specified with a set of vertex coordinates and associated attribute parameters.
- For each polygon input, the data are placed into tables that are to be used in the subsequent processing.
- Polygon data tables can be organized into two groups: Geometric tables and attribute tables.

Geometric Tables

Contain vertex coordinates and parameters to identify the spatial orientation of the polygon surfaces.

Attribute tables

Contain attribute information for an object such as parameters specifying the degree of transparency of the object and its surface reflectivity and texture characteristics.

A convenient organization for storing geometric data is to create three lists:

1. The Vertex Table

   Coordinate values for each vertex in the object are stored in this table.

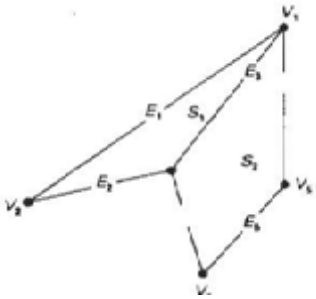
2. The Edge Table

It contains pointers back into the vertex table to identify the vertices for each polygon edge.

3. The Polygon Table

It contains pointers back into the edge table to identify the edges for each polygon.

This is shown in fig



Vertex table	Edge Table	Polygon surface table
$V_1 : X_1, Y_1, Z_1$	$E_1 : V_1, V_2$	$S_1 : E_1, E_2, E_3$
$V_2 : X_2, Y_2, Z_2$	$E_2 : V_2, V_3$	$S_2 : E_3, E_4, E_5, E_6$
$V_3 : X_3, Y_3, Z_3$	$E_3 : V_3, V_1$	
$V_4 : X_4, Y_4, Z_4$	$E_4 : V_3, V_4$	
$V_5 : X_5, Y_5, Z_5$	$E_5 : V_4, V_5$	
	$E6 : V_5, V_1$	

- Listing the geometric data in three tables provides a convenient reference to the individual components (vertices, edges and polygons) of each object.
- The object can be displayed efficiently by using data from the edge table to draw the component lines.
- Extra information can be added to the data tables for faster information extraction. For instance, edge table can be expanded

to include forward points into the polygon table so that common edges between polygons can be identified more rapidly.

$E_1 : V_1, V_2, S_1$

$E_2 : V_2, V_3, S_1$

$E_3 : V_3, V_1, S_1, S_2$

$E_4 : V_3, V_4, S_2$

$E_5 : V_4, V_5, S_2$

$E6 : V_5, V_1, S_2$

- This is useful for the rendering procedure that must vary surface shading smoothly across the edges from one polygon to the next. Similarly, the vertex table can be expanded so that vertices are cross-referenced to corresponding edges.
- Additional geometric information that is stored in the data tables includes the slope for each edge and the coordinate extends for each polygon. As vertices are input, we can calculate edge slopes and we can scan the coordinate values to identify the minimum and maximum x, y and z values for individual polygons.
- The more information included in the data tables will be easier to check for errors.
- Some of the tests that could be performed by a graphics package are:
    1. That every vertex is listed as an endpoint for at least two edges.
    2. That every edge is part of at least one polygon.
    3. That every polygon is closed.
    4. That each polygon has at least one shared edge.
    5. That if the edge table contains pointers to polygons, every edge referenced by a polygon pointer has a reciprocal pointer back to the polygon.

Plane Equations:

- To produce a display of a 3D object, we must process the input data representation for the object through several procedures such as,
    - Transformation of the modeling and world coordinate descriptions to viewing coordinates.
    - Then to device coordinates:
    - Identification of visible surfaces
    - The application of surface-rendering procedures.
- For these processes, we need information about the spatial orientation of the individual surface components of the object. This

information is obtained from the vertex coordinate value and the equations that describe the polygon planes.

The equation for a plane surface is

$$Ax + By + Cz + D = 0 \text{ --------(1)}$$

Where (x, y, z) is any point on the plane, and the coefficients A,B,C and D are constants describing the spatial properties of the plane.

- We can obtain the values of A, B,C and D by solving a set of three plane equations using the coordinate values for three non collinear points in the plane.
- For that, we can select three successive polygon vertices (x_1, y_1, z_1), (x_2, y_2, z_2) and (x_3, y_3, z_3) and solve the following set of simultaneous linear plane equations for the ratios A/D, B/D and C/D.

$$(A/D)x_k + (B/D)y_k + (c/D)z_k = -1, \qquad k=1,2,3 \text{ -------- (2)}$$

- The solution for this set of equations can be obtained in determinant form, using Cramer's rule as

$$A = \begin{vmatrix} 1 & y_1 & z_1 \\ 1 & y_2 & z_2 \\ 1 & y_3 & z_3 \end{vmatrix} \qquad B = \begin{vmatrix} x_1 & 1 & z_1 \\ x_2 & 1 & z_2 \\ x_3 & 1 & z_3 \end{vmatrix}$$

$$C = \begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \qquad D = -\begin{vmatrix} x_1 & y_1 & z_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix} \text{ ---------- (3)}$$

- Expanding the determinants , we can write the calculations for the plane coefficients in the form:

$$A = y_1 (z_2 - z_3) + y_2(z_3 - z_1) + y_3 (z_1 - z_2)$$

$$B = z_1 (x_2 - x_3) + z_2 (x_3 - x_1) + z_3 (x_1 - x_2)$$

$$C = x_1 (y_2 - y_3) + x_2 (y_3 - y_1) + x_3 (y_1 - y_2)$$

$$D = -x_1 (y_2 z_3 - y_3 z_2) - x_2 (y_3 z_1 - y_1 z_3) - x_3 (y_1 z_2 - y_2 z_1) \text{ ------ (4)}$$

- As vertex values and other information are entered into the polygon data structure, values for A, B, C and D are computed for each polygon and stored with the other polygon data.
- Plane equations are used also to identify the position of spatial points relative to the plane surfaces of an object. For any point (x, y, z) hot on a plane with parameters A,B,C,D, we have

$$Ax + By + Cz + D \neq 0$$

- We can identify the point as either inside or outside the plane surface according o the sigh (negative or positive) of Ax + By + Cz + D:

  If Ax + By + Cz + D < 0, the point (x, y, z) is inside the surface.

  If Ax + By + Cz + D > 0, the point (x, y, z) is outside the surface.

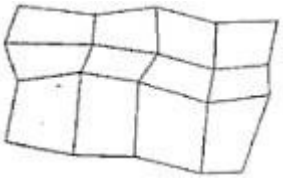
- These inequality tests are valid in a right handed Cartesian system, provided the plane parmeters A,B,C and D were calculated using vertices selected in a counter clockwise order when viewing the surface in an outside-to-inside direction.

Polygon Meshes

- A single plane surface can be specified with a function such as fillArea. But when object surfaces are to be tiled, it is more convenient to specify the surface facets with a mesh function.
- One type of polygon mesh is the triangle strip.A triangle strip formed with 11 triangles connecting 13 vertices.

This function produces n-2 connected triangles given the coordinates for n vertices.

- Another similar function in the quadrilateral mesh, which generates a mesh of (n-1) by (m-1) quadrilaterals, given the coordinates for an n by m array of vertices. Figure shows 20 vertices forming a mesh of 12 quadrilaterals.



Curved Lines and Surfaces

- Displays of three dimensional curved lines and surface can be generated from an input set of mathematical functions defining the objects or from a set of user specified data points for surfaces, a functional description in decorated to produce a polygon-mesh approximation to the surface.

Quadric Surfaces

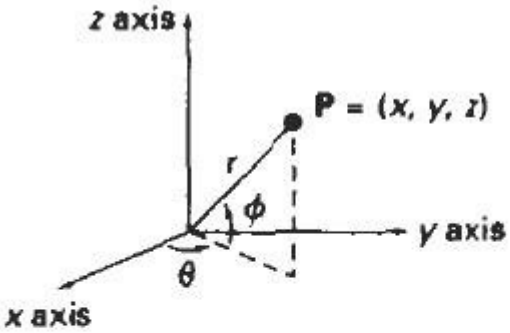
- The quadric surfaces are described with second degree equations (quadratics).
- They include spheres, ellipsoids, tori, parabolids, and hyperboloids.

Sphere

- In Cartesian coordinates, a spherical surface with radius r centered on the coordinates origin is defined as the set of points (x, y, z) that satisfy the equation.

$$x^2 + y^2 + z^2 = r^2 \text{----------(1)}$$

- The spherical surface can be represented in parametric form by using latitude and longitude angles



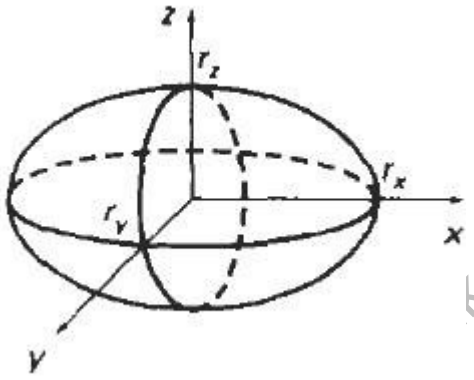
$$x = r\cos\varphi\cos\theta, \quad -\Lambda/2 <= \varphi <= \Lambda/2$$
$$y = r\cos\varphi\sin\theta, \quad -\Lambda <= \varphi <= \Lambda \qquad \text{-------(2)}$$
$$z = r\sin\varphi$$

- The parameter representation in eqn (2) provides a symmetric range for the angular parameter θ and φ.

Ellipsoid

- Ellipsoid surface is an extension of a spherical surface where the radius in three mutually perpendicular directions can have different values The Cartesian representation for points over the surface of an ellipsoid centered on the origin is

$$\left(\frac{x}{r_x}\right)^2 + \left(\frac{y}{r_y}\right)^2 + \left(\frac{z}{r_z}\right)^2 = 1$$



- The parametric representation for the ellipsoid in terms of the latitude angle φ and the longitude angle θ is

$$x = r_x\cos\varphi\cos\theta, \quad -\Lambda/2 <= \varphi <= \Lambda/2$$
$$y = r_y\cos\varphi\sin\theta, \quad -\Lambda <= \varphi <= \Lambda$$
$$z = r_z\sin\varphi$$

Spline Representations

- A Spline is a flexible strip used to produce a smooth curve through a designated set of points.
- Several small weights are distributed along the length of the strip to hold it in position on the drafting table as the curve is drawn.

- The Spline curve refers to any sections curve formed with polynomial sections satisfying specified continuity conditions at the boundary of the pieces.
- A Spline surface can be described with two sets of orthogonal spline curves.
- Splines are used in graphics applications to design curve and surface shapes, to digitize drawings for computer storage, and to

specify animation paths for the objects or the camera in the scene. CAD applications for splines include the design of automobiles bodies, aircraft and spacecraft surfaces, and ship hulls.

Interpolation and Approximation Splines

- Spline curve can be specified by a set of coordinate positions called control points which indicates the general shape of the curve.
- These control points are fitted with piecewise continuous parametric polynomial functions in one of the two ways.
    1. When polynomial sections are fitted so that the curve passes through each control point the resulting curve is said to interpolate the set of control points.

A set of six control points interpolated with piecewise continuous polynomial sections

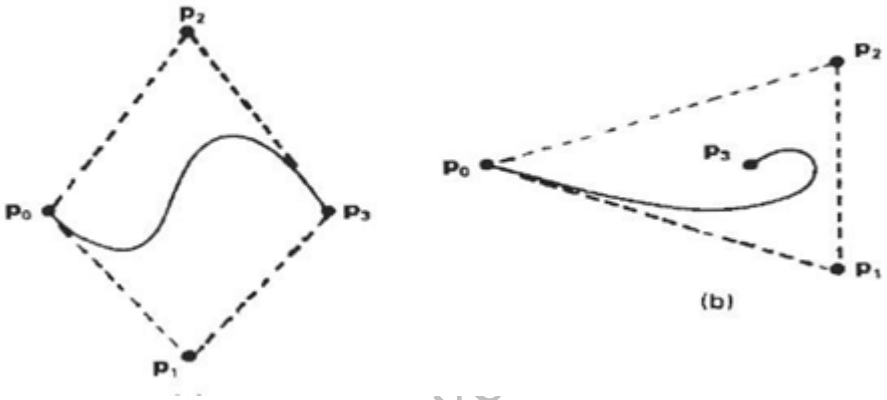
    2. When the polynomials are fitted to the general control point path without necessarily passing through any control points, the resulting curve is said to approximate the set of control points.

A set of six control points approximated with piecewise continuous polynomial sections

- Interpolation curves are used to digitize drawings or to specify animation paths.
- Approximation curves are used as design tools to structure object surfaces.

- A spline curve is designed, modified and manipulated with operations on the control points.The curve can be translated, rotated or scaled with transformation applied to the control points.
- The convex polygon boundary that encloses a set of control points is called the convex hull.
- The shape of the convex hull is to imagine a rubber band stretched around the position of the control points so that each control point is either on the perimeter of the hull or inside it.

Convex hull shapes (dashed lines) for two sets of control points



Parametric Continuity Conditions

For a smooth transition from one section of a piecewise parametric curve to the next various continuity conditions are needed at the connection points.

If each section of a spline in described with a set of parametric coordinate functions or the form

x = x(u), y = y(u), z = z(u), u1<= u <= u2----------(a)

- We set parametric continuity by matching the parametric derivatives of adjoining curve sections at their common boundary.
- Zero order parametric continuity referred to as C^0 continuity, means that the curves meet. (i.e) the values of x,y, and z evaluated at u_2 for the first curve section are equal. Respectively, to the value of x,y, and z evaluated at u_1 for the next curve section.
- First order parametric continuity referred to as C^1 continuity means that the first parametric derivatives of the coordinate functions in equation (a) for two successive curve sections are

equal at their joining point.

- Second order parametric continuity, or C2 continuity means that both the first and second parametric derivatives of the two curve sections are equal at their intersection.

Higher order parametric continuity conditions are defined similarly.

Piecewise construction of a curve by joining two curve segments using different orders of continuity

a) Zero order continuity only

b) First order continuity only

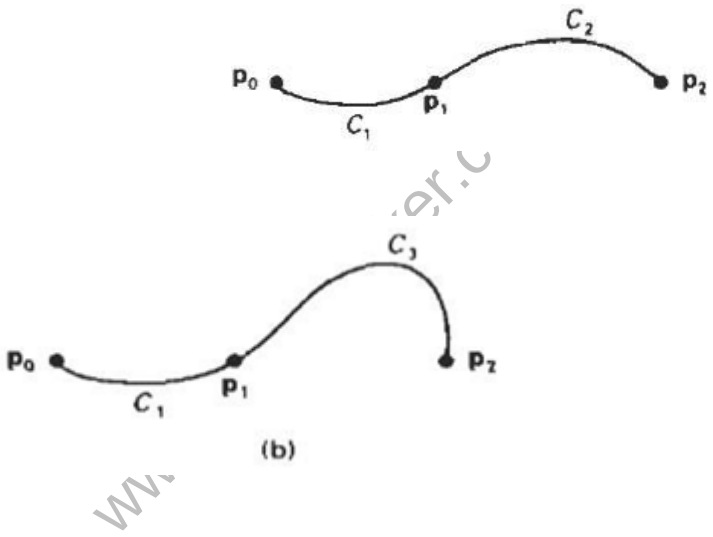
c) Second order continuity only

Geometric Continuity Conditions

- To specify conditions for geometric continuity is an alternate method for joining two successive curve sections.

- The parametric derivatives of the two sections should be proportional to each other at their common boundary instead of equal to each other.
- Zero order Geometric continuity referred as G^0 continuity means that the two curves sections must have the same coordinate position at the boundary point.
- First order Geometric Continuity referred as G^1 continuity means that the parametric first derivatives are proportional at the interaction of two successive sections.
- Second order Geometric continuity referred as G^2 continuity means that both the first and second parametric derivatives of the two curve sections are proportional at their boundary. Here the curvatures of two sections will match at the joining position.

Three control points fitted with two curve sections joined with a) parametric continuity



b) geometric continuity where the tangent vector of curve C_3 at point p1 has a greater magnitude than the tangent vector of curve C1 at p1.

Spline specifications

There are three methods to specify a spline representation:

1. We can state the set of boundary conditions that are imposed on the spline; (or)
2. We can state the matrix that characterizes the spline; (or)
3. We can state the set of blending functions that determine how specified geometric constraints on the curve are combined to calculate positions along the curve path.

To illustrate these three equivalent specifications, suppose we have the following parametric cubic polynomial representation for the x coordinate along the path of a spline section.

$$x(u)=a_xu^3 + a_xu^2 + c_xu + d_x \qquad 0<= u <=1 \text{ ----------(1)}$$

Boundary conditions for this curve might be set on the endpoint coordinates x(0) and x(1) and on the parametric first derivatives at the endpoints x'(0) and x'(1). These boundary conditions are sufficient to determine the values of the four coordinates a_x, b_x, c_x and d_x.

From the boundary conditions we can obtain the matrix that characterizes this spline curve by first rewriting eq(1) as the matrix product

$$x(u) = [u^3 \quad u^2 \quad u \quad 1] \begin{pmatrix} a_x \\ b_x \\ c_x \\ d_x \end{pmatrix} = U.C$$

where U is the row matrix of power of parameter u and C is the coefficient column matrix.

- Using equation (2) we can write the boundary conditions in matrix form and solve for the coefficient matrix C as

$$C = M_{spline} \cdot M_{geom} \text{ ---------- (3)}$$

Where M_{geom} in a four element column matrix containing the geometric constraint values on the spline and M_{spline} in the 4 * 4 matrix that transforms the geometric constraint values to the polynomial coefficients and provides a characterization for the spline curve.

- Matrix M_{geom} contains control point coordinate values and other geometric constraints.
- We can substitute the matrix representation for C into equation (2) to obtain.

$$x(u) = U \cdot M_{spline} \cdot M_{geom} \text{ ---------- (4)}$$

- The matrix M_{spline}, characterizing a spline representation, called the basis matriz is useful for transforming from one spline representation to another.
- Finally we can expand equation (4) to obtain a polynomial representation for coordinate x in terms of the geometric constraint parameters.

$$x(u) = \sum \mathbf{g}_k . BF_k(u)$$

where g_k are the constraint parameters, such as the control point

coordinates and slope of the curve at the control points and $BF_k(u)$ are the polynomial blending functions.

Visualization of Data Sets

- The use of graphical methods as an aid in scientific and engineering analysis is commonly referred to as scientific visualization.
- This involves the visualization of data sets and processes that may be difficult or impossible to analyze without graphical methods. Example medical scanners, satellite and spacecraft scanners
- Visualization techniques are useful for analyzing process that occur over a long period of time or that cannot observed directly. Example quantum mechanical phenomena and special relativity effects produced by objects traveling near the speed of light.
- Scientific visualization is used to visually display , enhance and manipulate information to allow better understanding of the data.
- Similar methods employed by commerce, industry and other nonscientific areas are sometimes referred to as business visualization.
- Data sets are classified according to their spatial distribution (2D or 3D) and according to data type (scalars , vectors , tensors and multivariate data).

Visual Representations for Scalar Fields

- A scalar quantity is one that has a single value. Scalar data sets contain values that may be distributed in time as well as over spatial positions also the values may be functions of other scalar parameters. Examples of physical scalar quantities are energy, density, mass , temperature and water content.
- A common method for visualizing a scalar data set is to use graphs or charts that show the distribution of data values as a function of other parameters such as position and time.
- Pseudo-color methods are also used to distinguish different values in a scalar data set, and color coding techniques can be combined with graph and chart models. To color code a scalar data set we choose a range of colors and map the range of data values to the color range. Color coding a data set can be tricky because some color combinations can lead to misinterpretations of the data.
- Contour plots are used to display isolines (lines of constant scalar value) for a data set distributed over a surface. The isolines are

spaced at some convenient interval to show the range and variation of the data values over the region of space. Contouring methods are applied to a set of data values that is distributed over a regular grid.

A 2D contouring algorithm traces the isolines from cell to cell within the grid by checking the four corners of grid cells to determine which cell edges are crossed by a particular isoline.

The path of an isoline across five grid cells

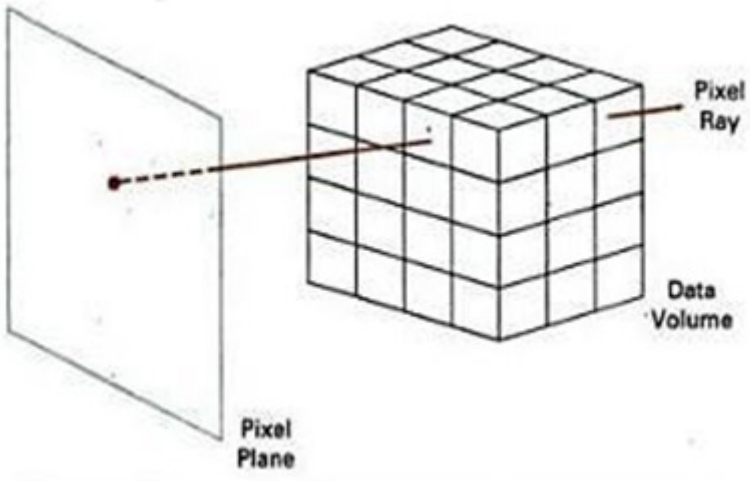
Sometimes isolines are plotted with spline curves but spline fitting can lead to misinterpretation of the data sets. Example two spline isolines could cross or curved isoline paths might not be a true indicator of data trends since data values are known only at the cell corners.

For 3D scalar data fields we can take cross sectional slices and display the 2D data distributions over the slices. Visualization packages provide a slicer routine that allows cross sections to be taken at any angle.

Instead of looking at 2D cross sections we plot one or more isosurfaces which are simply 3D contour plots. When two overlapping isosurfaces are displayed the outer surface is made transparent so that we can view the shape of both isosurfaces.

- Volume rendering which is like an X-ray picture is another method for visualizing a 3D data set. The interior information about a data set is projected to a display screen using the ray- casting method. Along the ray path from each screen pixel

Volume visualization of a regular, Cartesian data grid using ray casting to examine interior data values



. Data values at the grid positions. are averaged so that one value is stored for each voxel of the data space. How the data are encoded for display depends on the application.

Visual representation for Vector fields

- A vector quantity V in three-dimensional space has three scalar values

  (V_x , V_y,V_z,) one for each coordinate direction, and a two-dimensional vector has two components (V_x, V_y,). Another way to describe a vector quantity is by giving its magnitude IV I and its direction as a unit vector u.

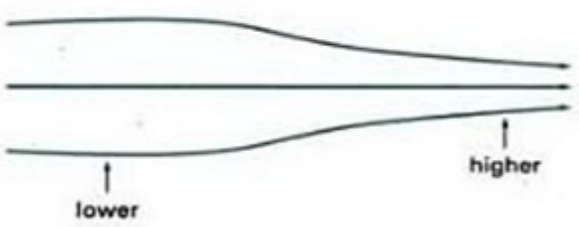
  As with scalars, vector quantities may be functions of position, time, and other parameters. Some examples of physical vector quantities are velocity, acceleration, force, electric fields, magnetic fields, gravitational fields, and electric current.

  One way to visualize a vector field is to plot each data point as a small arrow that shows the magnitude and direction of the vector. This method is most often used with cross-sectional slices, since it can be difficult to see the trends in a three-dimensional region cluttered with overlapping arrows. Magnitudes for the vector values can be shown by varying the lengths of the arrows.

  Vector values are also represented by plotting field lines or streamlines .

  Field lines are commonly used for electric , magnetic and gravitational fields. The magnitude of the vector values is indicated by spacing between field lines, and the direction is the tangent to the field.

Field line representation for a vector data set



Visual Representations for Tensor Fields

A tensor quantity in three-dimensional space has nine components and can be represented with a 3 by 3 matrix. This representation is used for a second-order tensor, and higher-order tensors do occur in some applications.

Some examples of physical, second-order tensors are stress and strain in a material subjected to external forces, conductivity of an electrical conductor, and the metric tensor, which gives the properties of a particular coordinate space.

The stress tensor in Cartesian coordinates, for example, can be represented as

$$\begin{bmatrix} \sigma_x & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_y & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_z \end{bmatrix}$$

Tensor quantities are frequently encountered in anisotropic materials, which have different properties in different directions. The x, xy, and xz elements of the conductivity tensor, for example, describe the contributions of electric field components in the x, y, and *z* directions to the current in the *x* direction.

Usually, physical tensor quantities are symmetric, so that the tensor has only six distinct values. Visualization schemes for representing all six components of a second-order tensor quantity are based on devising shapes that have six parameters.

Instead of trying to visualize all six components of a tensor quantity, we can reduce the tensor to a vector or a scalar. And by applying tensor-contraction operations, we can obtain a scalar representation.

Visual Representations for Multivariate Data Fields

In some applications, at each grid position over some region of space, we may have multiple data values, which can be a mixture of scalar, vector, and even tensor values.

A method for displaying multivariate data fields is to construct graphical objects, sometimes referred to as glyphs, with multiple parts. Each part of a glyph represents a physical quantity. The size and color of each part can be used to display information about scalar magnitudes. To give directional information for a vector field, we can use a wedge, a cone, or some other pointing shape for the glyph part representing the vector.

Three Dimensional Geometric and Modeling Transformations

Geometric transformations and object modeling in three Dimensions are extended from two-dimensional methods by including considerations for the z-coordinate.

Translation

In a three dimensional homogeneous coordinate representation, a point or an object is translated from position P = (x,y,z) to position P' = (x',y',z') with the matrix operation.

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & y_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \quad \text{------1}$$

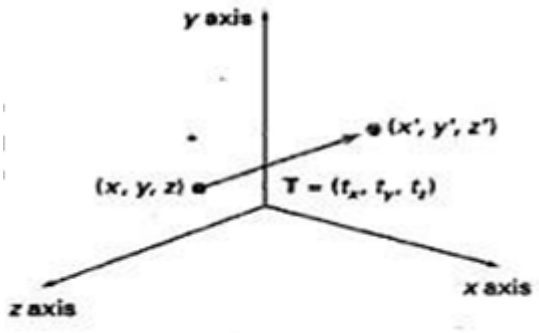
(or) **P' = T.P**----------(2)

Parameters t_x, t_y and t_z specifying translation distances for the coordinate directions x,y and z are assigned any real values.

The matrix representation in equation (1) is equivalent to the three equations

$$x' = x + t_x$$
$$y' = y + t_y$$
$$z' = z + t_z \quad \text{----------(3)}$$

Translating a point with translation vector T = (t_x, t_y, t_z)



Inverse of the translation matrix in equation (1) can be obtained by negating the translation distance t_x, t_y and t_z.

This produces a translation in the opposite direction and the product of a translation matrix and its inverse produces the identity matrix.

Rotation

To generate a rotation transformation for an object an axis of rotation must be designed to rotate the object and the amount of angular rotation is also be specified.

- Positive rotation angles produce counter clockwise rotations about a coordinate axis.

Co-ordinate Axes Rotations

The 2D z axis rotation equations are easily extended to 3D.

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

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

$z' = z$---------- (2)

Parameters θ specifies the rotation angle. In homogeneous coordinate form, the 3D z axis rotation equations are expressed as

$$\begin{pmatrix} x^1 \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \text{-------(3)}$$

which we can write more compactly as

$$\mathbf{P' = R_z\,(\theta)\,.\,P}$$---------- (4)

The below figure illustrates rotation of an object about the z axis.

Transformation equations for rotation about the other two coordinate axes can be obtained with a cyclic permutation of the

coordinate parameters x, y and z in equation (2) i.e., we use the replacements

$$x \rightarrow y \rightarrow z \rightarrow x \text{---------- (5)}$$

Substituting permutations (5) in equation (2), we get the equations for an x-axis rotation

$$\begin{aligned} y' &= y\cos\theta - z\sin\theta \\ z' &= y\sin\theta + z\cos\theta \quad \text{----------(6)} \\ x' &= x \end{aligned}$$

which can be written in the homogeneous coordinate form

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \text{-------(7)}$$

(or) $\mathbf{P' = R_x(\theta).\ P}$ **----------**(8)

Rotation of an object around the x-axis is demonstrated in the below fig

Cyclically permuting coordinates in equation (6) give the transformation equation for a y axis rotation.

$$\begin{aligned} z' &= z\cos\theta - x\sin\theta \\ x' &= z\sin\theta + x\cos\theta \quad \text{----------(9)} \\ y' &= y \end{aligned}$$

The matrix representation for y-axis rotation is

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \text{----------(10)}$$

(or) $\mathbf{P' = R_y(\theta).\ P}$ **----------** (11)

An example of y axis rotation is shown in below figure

- An inverse rotation matrix is formed by replacing the rotation angle θ by –θ.
- Negative values for rotation angles generate rotations in a clockwise direction, so the identity matrix is produces when any rotation matrix is multiplied by its inverse.
- Since only the sine function is affected by the change in sign of the rotation angle, the inverse matrix can also be obtained by interchanging rows and columns. (i.e.,) we can calculate the inverse of any rotation matrix R by evaluating its transpose ($R^{-1} = R^T$).

General Three Dimensional Rotations

- A rotation matrix for any axis that does not coincide with a coordinate axis can be set up as a composite transformation involving combinations of translations and the coordinate axes rotations.
- We obtain the required composite matrix by
  1. Setting up the transformation sequence that moves the selected rotation axis onto one of the coordinate axis.
  2. Then set up the rotation matrix about that coordinate axis for the specified rotation angle.
  3. Obtaining the inverse transformation sequence that returns the rotation axis to its original position.
- In the special case where an object is to be rotated about an axis that is parallel to one of the coordinate axes, we can attain the desired rotation with the following transformation sequence:
  1. Translate the object so that the rotation axis coincides with the parallel coordinate axis.
  2. Perform the specified rotation about the axis.
  3. Translate the object so that the rotation axis is moved back to its original position.
- When an object is to be rotated about an axis that is not parallel to one of the coordinate axes, we need to perform some additional transformations.
- In such case, we need rotations to align the axis with a selected coordinate axis and to bring the axis back to its original orientation.
- Given the specifications for the rotation axis and the rotation angle, we can accomplish the required rotation in five steps:
  1. Translate the object so that the rotation axis passes through the coordinate origin.

2. Rotate the object so that the axis of rotation coincides with one of the coordinate axes.
3. Perform the specified rotation about that coordinate axis.
4. Apply inverse rotations to bring the rotation axis back to its original orientation.
5. Apply the inverse translation to bring the rotation axis back to its original position.

Scaling

- The matrix expression for the scaling transformation of a position P = (x,y,.z) relative to the coordinate origin can be written as

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \text{ --------(11)}$$

(or) **P' = S.P**---------- (12)

where scaling parameters s_x , s_y, and s_z are assigned any position values.

- Explicit expressions for the coordinate transformations for scaling relative to the origin are

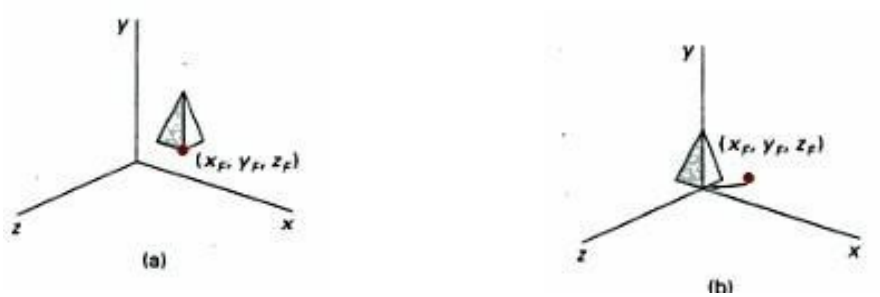
$$x' = x.s_x$$
$$y' = y.s_y \text{----------(13)}$$
$$z' = z.s_z$$

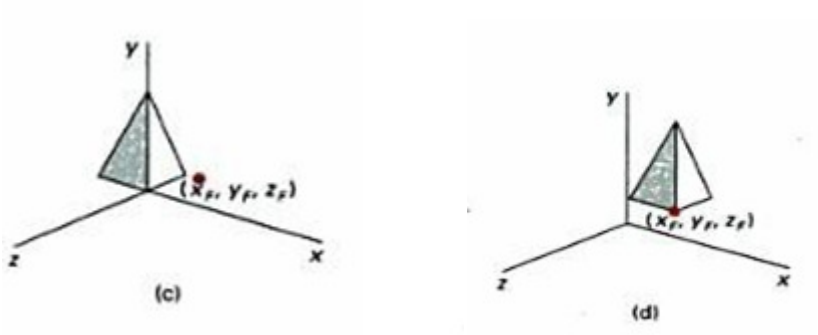
- Scaling an object changes the size of the object and repositions the object relatives to the coordinate origin.
- If the transformation parameters are not equal, relative dimensions in the object are changed.
- The original shape of the object is preserved with a uniform scaling ($s_x = s_y = s_z$).
- Scaling with respect to a selected fixed position (x_f, y_f, z_f) can be represented with the following transformation sequence:

  Translate the fixed point to the origin.

  Scale the object relative to the coordinate origin using Eq.11.

Translate the fixed point back to its original position. This sequence of transformation is shown in the below figure .





- The matrix representation for an arbitrary fixed point scaling can be expressed as the concatenation of the translate-scale-translate transformation are

$T(x_f, y_f, z_f) \cdot S(s_x, s_y, s_z) \cdot T(-x_f, -y_f, -z_f) =$

$$\begin{bmatrix} s_x & 0 & 0 & (1-s_x)x_f \\ 0 & s_y & 0 & (1-s_y)y_f \\ 0 & 0 & s_z & (1-s_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \text{----------(14)}$$

- Inverse scaling matrix m formed by replacing the scaling parameters s_x, s_y and s_z with their reciprocals.
- The inverse matrix generates an opposite scaling transformation, so the concatenation of any scaling matrix and its inverse produces the identity matrix.

Other Transformations

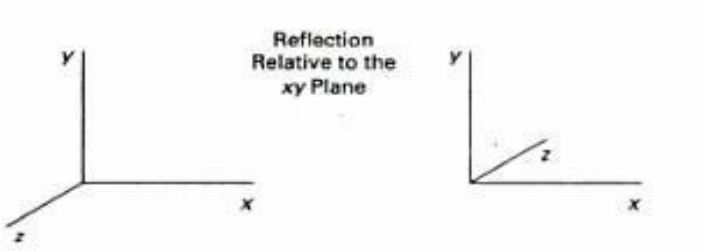
- A 3D reflection can be performed relative to a selected reflection

axis or with respect to a selected reflection plane.

- Reflection relative to a plane are equivalent to 180^0 rotations in 4D space.

- When the reflection plane in a coordinate plane (either x_y, x_z or y_z) then the transformation can be a conversion between left-handed and right-handed systems.

- An example of a reflection that converts coordinate specifications from a right handed system to a left-handed system is shown in the figure



- This transformation changes the sign of z coordinates, leaves the x and y coordinate values unchanged.

- The matrix representation for this reflection of points relative to the xy plane is

$$RF_z = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

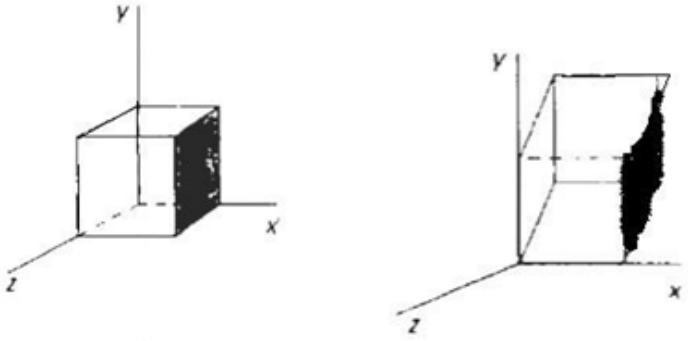
- Reflections about other planes can be obtained as a combination of rotations and coordinate plane reflections.

Shears

- Shearing transformations are used to modify object shapes.
- They are also used in three dimensional viewing for obtaining general projections transformations.
- The following transformation produces a z-axis shear.

Parameters a and b can be assigned any real values.

This transformation matrix is used to alter x and y coordinate Values by an amount that is proportional to the z value, and the z coordinate will be unchanged. Boundaries of planes that are perpendicular to the z axis are shifted by an amount proportional to z the figure shows the effect of shearing matrix on a unit cube for the values a = b = 1.



Three-Dimensional Viewing

In three dimensional graphics applications,

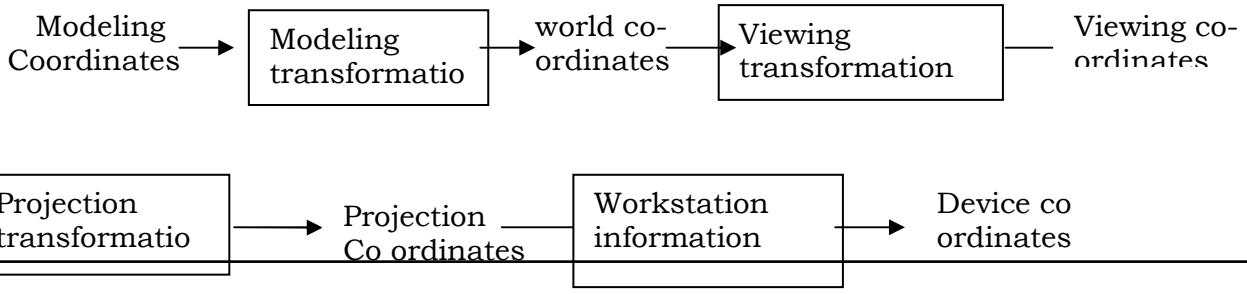
- we can view an object from any spatial position, from the front, from above or from the back.
- We could generate a view of what we could see if we were standing in the middle of a group of objects or inside object, such as a building.

Viewing Pipeline:

In the view of a three dimensional scene, to take a snapshot we need to do the following steps.

1. Positioning the camera at a particular point in space.
2. Deciding the camera orientation (i.e.,) pointing the camera and rotating it around the line of right to set up the direction for the picture.
3. When snap the shutter, the scene is cropped to the size of the 'window' of the camera and light from the visible surfaces is projected into the camera film.

In such a way the below figure shows the three dimensional transformation pipeline, from modeling coordinates to final device coordinate.



s position is

packages as
tion and the

the viewing
positions on
) the output

rom further
sed through
; procedures

converted to
objects onto

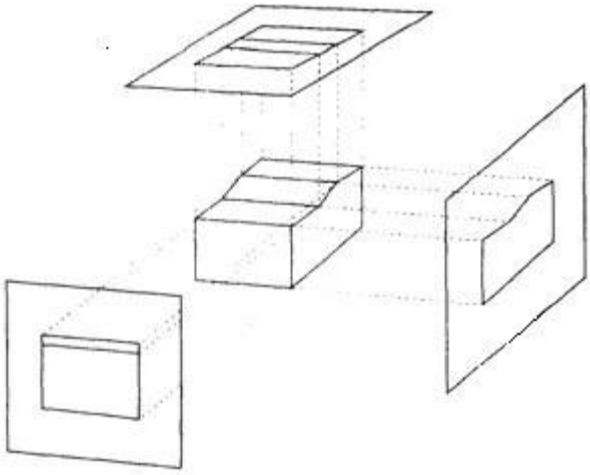
ons are
nes.

Orthographic Projection

- Orthographic projections are used to produce the front, side and top views of an object.
- Front, side and rear orthographic projections of an object are called elevations.
- A top orthographic projection is called a plan view.

- This projection gives the measurement of lengths and angles accurately.

Orthographic projections of an object, displaying plan and elevation views



- The orthographic projection that displays more than one face of an object is called axonometric orthographic projections.
- The most commonly used axonometric projection is the isometric projection.
- It can be generated by aligning the projection plane so that it intersects each coordinate axis in which the object is defined as the same distance from the origin.

UNIT III - GRAPHICS PROGRAMMING

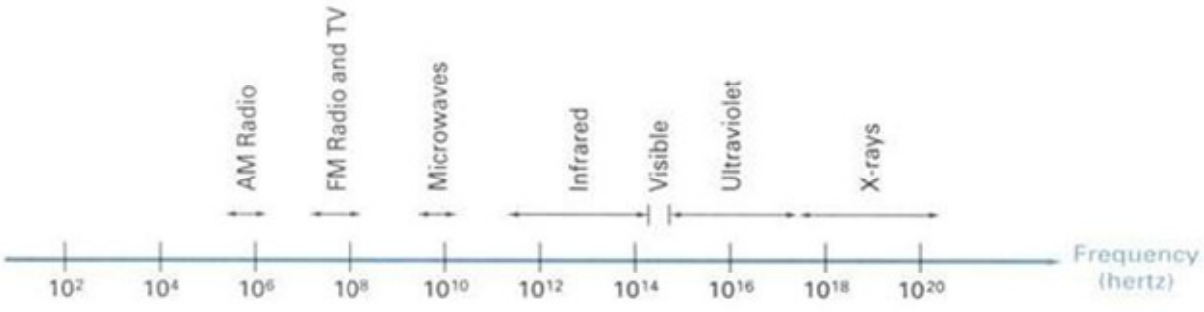
Color Models – RGB, YIQ, CMY, HSV – Animations – General Computer Animation, Raster, Keyframe - Graphics programming using OPENGL – Basic graphics primitives – Drawing three dimensional objects - Drawing three dimensional scenes

Color Models

Color Model is a method for explaining the properties or behavior of color within some particular context. No single color model can explain all aspects of color, so we make use of different models to help describe the different perceived characteristics of color.

Properties of Light

- Light is a narrow frequency band within the electromagnetic system.
- Other frequency bands within this spectrum are called radio waves, micro waves, infrared waves and x-rays. The below fig shows the frequency ranges for some of the electromagnetic bands.



- Each frequency value within the visible band corresponds to a distinct color.
- At the low frequency end is a red color ($4.3*10^4$ Hz) and the highest frequency is a violet color ($7.5 *10^{14}$Hz)
- Spectral colors range from the reds through orange and yellow at the low frequency end to greens, blues and violet at the high end.

- Since light is an electro magnetic wave, the various colors are described in terms of either the frequency for the wavelength λ of the wave.
  The wavelength ad frequency of the monochromatic wave is inversely proportional to each other, with the proportionality constants as the speed of light C where c= λf

- A light source such as the sun or a light bulb emits all frequencies within the visible range to produce white light. When white light is incident upon an object, some frequencies are reflected and some are absorbed by the object. The combination of frequencies present in the reflected light determines what we perceive as the color of the object.

  - If low frequencies are predominant in the reflected light, the object is described as red. In this case, the perceived light has the dominant frequency at the red end of the spectrum. The dominant frequency is also called the hue, or simply the color of the light.

  - Brightness is another property, which in the perceived intensity of the light.

  - Intensity in the radiant energy emitted per limit time, per unit solid angle, and per unit projected area of the source. Radiant energy is related to the luminance of the source.
    - Purity describes how washed out or how pure the color of the light appears.
    - Pastels and Pale colors are described as less pure.

  - The term chromaticity is used to refer collectively to the two properties, purity and dominant frequency.

  - Two different color light sources with suitably chosen intensities can be used to produce a range of other colors.

  - If the 2 color sources combine to produce white light, they are called complementary colors. E.g., Red and Cyan, green and magenta, and blue and yellow.

  - Color models that are used to describe combinations of light in terms of dominant frequency use 3 colors to obtain a wide range of colors, called the color gamut.

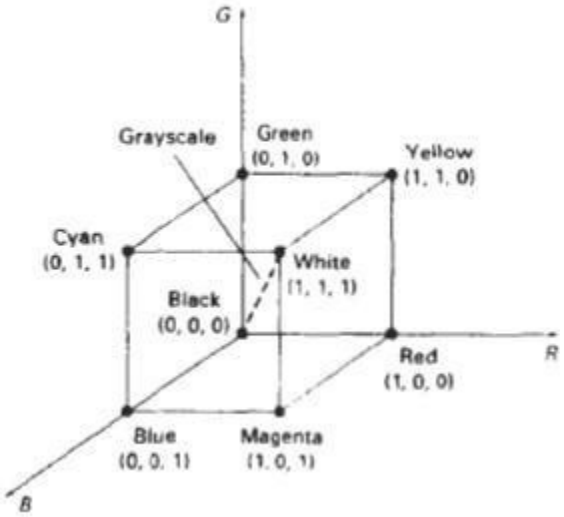
- The 2 or 3 colors used to produce other colors in a color model are called primary colors.

RGB Color Model

- Based on the tristimulus theory of version, our eyes perceive color through the stimulation of three visual pigments in the coneson the retina.

- These visual pigments have a peak sensitivity at wavelengths of about 630 nm (red), 530 nm (green) and 450 nm (blue).

- By comparing intensities in a light source, we perceive the color of the light.
- This is the basis for displaying color output on a video monitor using the 3 color primaries, red, green, and blue referred to as the RGB color model. It is represented in the below figure.



The sign represents black, and the vertex with coordinates (1,1,1) in white.

- Vertices of the cube on the axes represent the primary colors, the remaining vertices represents the complementary color for each of the primary colors.
- The RGB color scheme is an additive model. (i.e.,) Intensities of the primary colors are added to produce other colors.
- Each color point within the bounds of the cube can be represented as the triple (R,G,B) where values for R, G and B are assigned in the range from 0 to1.
- The color Cλ is expressed in RGB component as

$$C\lambda = R\mathbf{R} + G\mathbf{G} + B\mathbf{B}$$

- The magenta vertex is obtained by adding red and blue to produce the triple (1,0,1) and white at (1,1,1) in the sum of the red, green and blue vertices.
- Shades of gray are represented along the main diagonal of the cube from the origin (black) to the white vertex.

YIQ Color Model

- The National Television System Committee (NTSC) color model for forming the composite video signal in the YIQ model.
- In the YIQ color model, luminance (brightness) information in contained in the Y parameter, chromaticity information (hue and purity) is contained into the I and Q

parameters.

- A combination of red, green and blue intensities are chosen for the Y parameter to yield the standard luminosity curve.
- Since Y contains the luminance information, black and white TV monitors use only the Y signal.
- Parameter I contain orange-cyan hue information that provides the flash-tone shading and occupies a bandwidth of 1.5 MHz.
- Parameter Q carries green-magenta hue information in a bandwidth of about 0.6 MHz.
- An RGB signal can be converted to a TV signal using an NTSC encoder which converts RGB values to YIQ values, as follows

$$\begin{matrix} Y \\ I \\ Q \end{matrix} \quad \begin{matrix} 0.299 & 0.587 & 0.144 \\ 0.596 & 0.275 & 0.321 \\ 0.212 & 0.528 & 0.311 \end{matrix} \quad \begin{matrix} R \\ G \\ B \end{matrix}$$

- An NTSC video signal can be converted to an RGB signal using an NTSC encoder which separates the video signal into YIQ components, the converts to RCB values, as follows:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.620 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.108 & 1.705 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

CMY Color Model

- A color model defined with the primary colors cyan, magenta, and yellow (CMY) in useful for describing color output to hard copy devices.
- It is a subtractive color model (i.e.,) cyan can be formed by adding green and blue light. When white light is reflected from cyan-colored ink, the reflected light must have no red component. i.e., red light is absorbed or subtracted by the link.

- Magenta ink subtracts the green component from incident light and yellow subtracts the blue component.

- In CMY model, point (1,1,1) represents black because all components of the incident light are subtracted.
  - The origin represents white light.
  - Equal amounts of each of the primary colors produce grays along the main diagonal of the cube.
  - A combination of cyan and magenta ink produces blue light because the red and green components of the incident light are absorbed.
  - The printing process often used with the CMY model generates a color point with a collection of 4 ink dots; one dot is used for each of the primary colors (cyan, magenta and yellow) and one dot in black.

The conversion from an RGB representation to a CMY representation is expressed as

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Where the white is represented in the RGB system as the unit column vector.

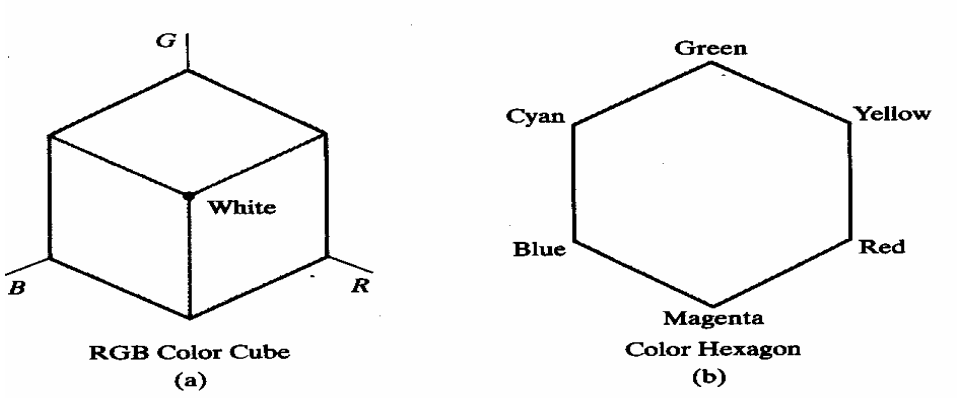
- Similarly the conversion of CMY to RGB representation is expressed as

$$\begin{matrix} R & 1 & C \\ G & 1 & M \\ B & 1 & Y \end{matrix}$$

Where black is represented in the CMY system as the unit column vector.

HSV Color Model

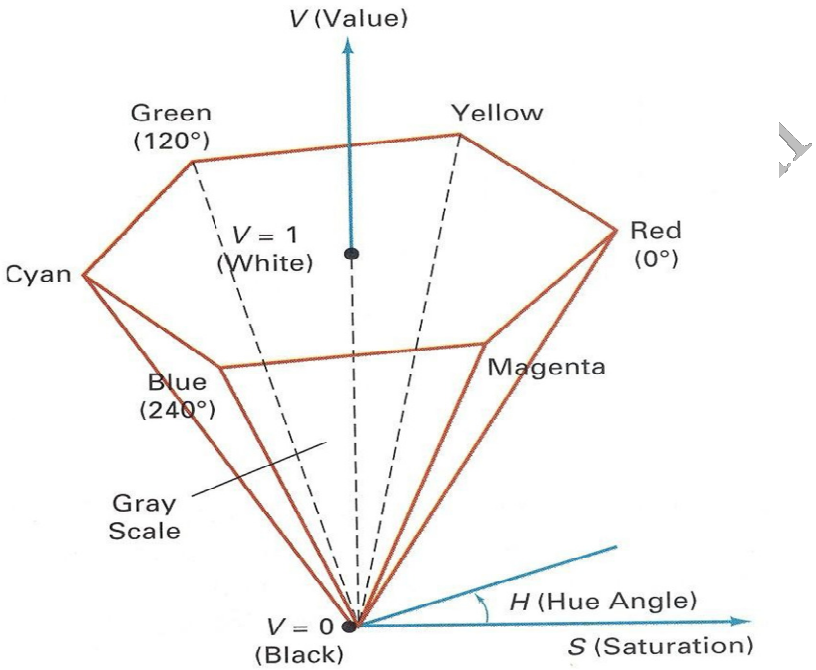
- The HSV model uses color descriptions that have a more interactive appeal to a user.
- Color parameters in this model are hue (H), saturation (S), and value (V).
- The 3D representation of the HSV model is derived from the RGB cube. The outline of the cube has the hexagon shape.



The boundary of the hexagon represents the various hues, and it is used as the top of the HSV hexcone.

- In the hexcone, saturation is measured along a horizontal axis, and value is along a vertical axis through the center of the hexcone.
- Hue is represented as an angle about the vertical axis, ranging from 0^0 at red through 360^0. Vertices of the hexagon are separated by 60^0 intervals.

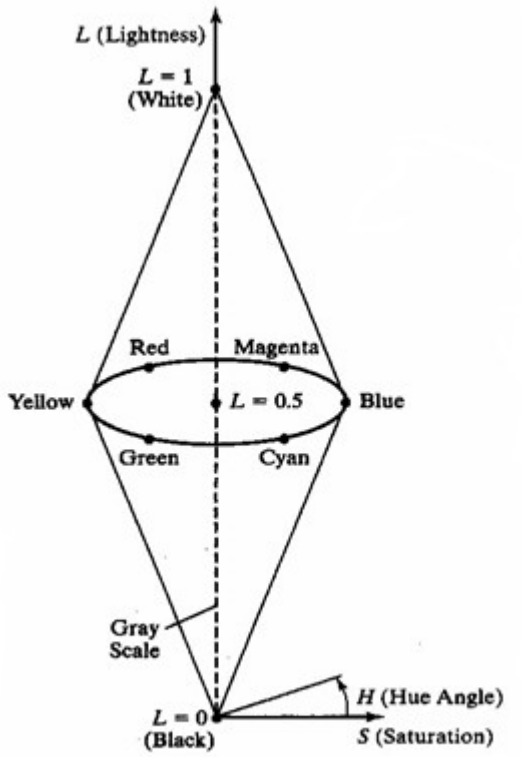
- Yellow is at 60^0, green at 120^0 and cyan opposite red at $H = 180^0$. Complementary colors are 180^0 apart.



- Saturation S varies from 0 to 1. the maximum purity at S = 1, at S = 0.25, the hue is said to be one quarter pure, at S = 0, we have the gray scale.
- Value V varies from 0 at the apex to 1 at the top.
  - the apex representation black.
- At the top of the hexcone, colors have their maximum intensity.
- When V = 1 and S = 1 we have the „pure" hues.
- White is the point at V = 1 and S = 0.

HLS Color Model

- HLS model is based on intuitive color parameters used by Tektronix.
- It has the double cone representation shown in the below figure. The 3 parameters in this model are called Hue (H), lightness (L) and saturation(s).



- Hue specifies an angle about the vertical axis that locates a chosen
  - In this model H = θ^0 corresponds to Blue.
  - The remaining colors are specified around the perimeter of the cone in the same order as in the HSV model.
  - Magenta is at 60^0, Red in at 120^0, and cyan in at H = 180^0.
  - The vertical axis is called lightness (L). At L = 0, we have black, and white is at L = 1 Gray scale in along the L axis and the "purehues" on the L = 0.5 plane.
  - Saturation parameter S specifies relative purity of a color. S varies from 0 to 1 pure hues are those for which S = 1 and L = 0.5

As S decreases, the hues are said to be less pure.

At S= 0, it is said to be gray scale.

Animation

Computer animation refers to any time sequence of visual changes in a scene.

- Computer animations can also be generated by changing camera parameters such as position, orientation and focal length.
- Applications of computer-generated animation are entertainment, advertising, training and education.

Example : Advertising animations often transition one object shape into another.

Frame-by-Frame animation

Each frame of the scene is separately generated and stored. Later, the frames can be recoded on film or they can be consecutively displayed in "real-time playback" mode

Design of Animation Sequences

An animation sequence in designed with the following steps:

- Story board layout
- Object definitions
- Key-frame specifications
- Generation of in-between frames.

Story board

- The story board is an outline of the action.
- It defines the motion sequences as a set of basic events that are to take place.
- Depending on the type of animation to be produced, the story board could consist of a set of rough sketches or a list of the basic ideas for the motion.

Object Definition

- An object definition is given for each participant in the action.

- Objects can be defined in terms of basic shapes such as polygons or splines.
- The associated movements of each object are specified along with the shape.

Key frame

- A key frame is detailed drawing of the scene at a certain time in the animation sequence.
- Within each key frame, each object is positioned according to the time for that frame.
- Some key frames are chosen at extreme positions in the action; others are spaced so that the time interval between keyframes is not too much.

In-betweens

- In betweens are the intermediate frames between the key frames.
- The number of in between needed is determined by the media to be used to display the animation.
- Film requires 24 frames per second and graphics terminals are refreshed at the rate of 30 to 60 frames per seconds.
- Time intervals for the motion are setup so there are from 3 to 5 in-between for each pair of key frames.
- Depending on the speed of the motion, some key frames can be duplicated.
- For a 1 min film sequence with no duplication, 1440 frames are needed.
- Other required tasks are
  - Motion verification
  - Editing
  - Production and synchronization of a sound track.

General Computer Animation Functions

Steps in the development of an animation sequence are,

1. Object manipulation and rendering

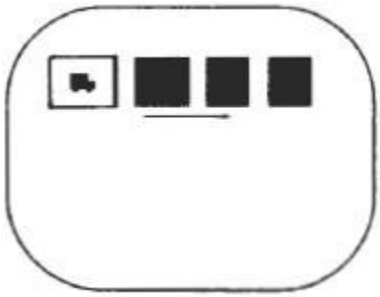
2. Camera motion

3. Generation of in-betweens

- Animation packages such as wave front provide special functions for designing the animation and processing individuals objects.
- Animation packages facilitate to store and manage the object database.
- Object shapes and associated parameter are stored and updated in the database.
- Motion can be generated according to specified constraints using 2D and 3D transformations.
- Standard functions can be applied to identify visible surfaces and apply the rendering algorithms.
- Camera movement functions such as zooming, panning and tilting are used for motion simulation.
- Given the specification for the key frames, the in-betweens can be automatically generated.

Raster Animations

- On raster systems, real-time animation in limited applications can be generated using raster operations.
- Sequence of raster operations can be executed to produce real time animation of either 2D or 3D objects.
- We can animate objects along 2D motion paths using the color-table transformations.
    - Predefine the object as successive positions along the motion path, set the successive blocks of pixel values to color table entries.
    - Set the pixels at the first position of the object to „on“ values, and set the pixels at the other object positions to the background color.
    - The animation is accomplished by changing the color table values so that the object is „on“ at successive positions along the animation path as the preceding position is set to the background intensity.



Computer Animation Languages

- Animation functions include a graphics editor, a key frame generator and standard graphics routines.
- The graphics editor allows designing and modifying object shapes, using spline surfaces, constructive solid geometry methods or other representation schemes.
- Scene description includes the positioning of objects and light sources defining the photometric parameters and setting the camera parameters.
- Action specification involves the layout of motion paths for the objects and camera.
- Keyframe systems are specialized animation languages designed dimply to generate the in-betweens from the user specified keyframes.
- Parameterized systems allow object motion characteristics to be specified as part of the object definitions. The adjustable parameters control such object characteristics as degrees of freedom motion limitations and allowable shape changes.
- Scripting systems allow object specifications and animation sequences to be defined with a user input script. From the script, a library of various objects and motions can be constructed.

Keyframe Systems

- Each set of in-betweens are generated from the specification of two keyframes.
- For complex scenes, we can separate the frames into individual components or objects called cells, an acronym from cartoon animation.

Morphing

- Transformation of object shapes from one form to another is called Morphing.

- Morphing methods can be applied to any motion or transition involving a change in shape. The example is shown in the below figure.

The general preprocessing rules for equalizing keyframes in terms of either the number of vertices to be added to a keyframe.

Suppose we equalize the edge count and parameters L_k and L_{k+1} denote the number of line segments in two consecutive frames. We define,

$L_{max} = \max (L_k, L_{k+1})$

$L_{min} = \min(L_k , L_{k+1})$

$N_e = L_{max} \bmod L_{min}$

$N_s = \text{int} (L_{max}/L_{min})$

- The preprocessing is accomplished by

1. Dividing N_e edges of keyframe_{min} into N_s+1 section.
2. Dividing the remaining lines of keyframe_{min} into N_s sections.

- For example, if $L_k = 15$ and $L_{k+1} = 11$, we divide 4 lines of keyframe_{k+1} into 2 sections each. The remaining lines of keyframe_{k+1} are left infact.

- If the vector counts in equalized parameters V_k and V_{k+1} are used to denote the number of vertices in the two consecutive frames. In this case we define

$V_{max} = \max(V_k,V_{k+1})$, $V_{min} = \min(V_k,V_{k+1})$ and

$N_{ls} = (V_{max} -1) \bmod (V_{min} - 1)$

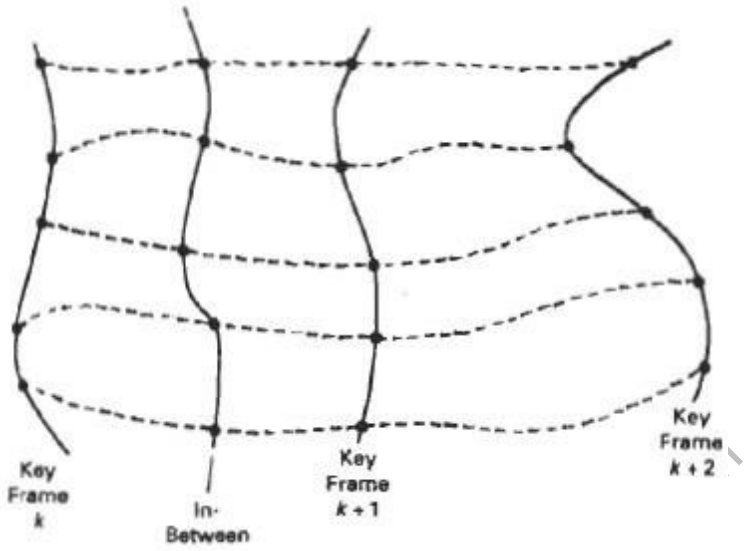
$N_p = \text{int} ((V_{max} - 1)/(V_{min} - 1))$

- Preprocessing using vertex count is performed by

1. Adding N_p points to N_{ls} line section of keyframe_{min}.
2. Adding N_p-1 points to the remaining edges of keyframe_{min}.

Simulating Accelerations

Curve-fitting techniques are often used to specify the animation paths between key frames. Given the vertex positions at the key frames, we can fit the positions with linear or nonlinear paths. Figure illustrates a nonlinear fit of key-frame positions. This determines the trajectories for the in-betweens. To simulate accelerations, we can adjust the time spacing for the in-betweens



For constant speed (zero acceleration), we use equal-interval time spacing for the in-betweens. Suppose we want n in-betweens for key frames at times t1 and t2.
The time interval between key frames is then divided into n + 1 subintervals, yielding an in-between spacing of

$\Delta = t2-t1/n+1$

we can calculate the time for anyin-between as

$tBj = t1+j\ \Delta t, \quad j = 1,2,..... n$

Motion Specification

These are several ways in which the motions of objects can be specified in an animation system.

Direct Motion Specification

- Here the rotation angles and translation vectors are explicitly given.
- Then the geometric transformation matrices are applied to transform coordinate positions.

We can approximate the path of a bouncing ball with a damped, rectified, sine curve

$$y(x) = A / \sin(\omega_x + \theta_0) / e^{-kx}$$

where A is the initial amplitude, ω is the angular frequency, θ_0 is the phase angle and k is the damping constant.

Goal Directed Systems

- We can specify the motions that are to take place in general terms that abstractly describe the actions.
- These systems are called goal directed. Because they determine specific motion parameters given the goals of the animation.
- Eg., To specify an object to „walk" or to „run" to a particular distance.

Kinematics and Dynamics

- With a kinematics description, we specify the animation by motion parameters (position, velocity and acceleration) without reference to the forces that cause the motion.
- For constant velocity (zero acceleration) we designate the motions of rigid bodies in a scene by giving an initial position and velocity vector for each object.
- We can specify accelerations (rate of change of velocity), speed up, slow downs and curved motion paths.
- An alternative approach is to use inverse kinematics; where the initial and final positions of the object are specified at specified times and the motion parameters are computed by the system.

Graphics programming using OPENGL

OpenGL is a software interface that allows you to access the graphics hardware without taking care of the hardware details or which graphics adapter is in the system. OpenGL is a low-level graphics library specification. It makes available to the programmer a small set of geomteric primitives - points, lines, polygons, images, and bitmaps. OpenGL provides a set of commands that allow the specification of geometric objects in two or three dimensions, using the provided primitives, together with commands that control how these objects are rendered (drawn).

Libraries

- OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections and rendering surfaces.
- OpenGL Utility Toolkit (GLUT) is a window-system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window APIs.

Include Files

For all OpenGL applications, you want to include the gl.h header file in every file. Almost all OpenGL applications use GLU, the aforementioned OpenGL Utility Library, which also requires inclusion of the glu.h header file. So almost every OpenGL source file begins with:

```
#include <GL/gl.h>
#include <GL/glu.h>
```

If you are using the OpenGL Utility Toolkit (GLUT) for managing your window manager tasks, you should include:

```
#include <GL/glut.h>
```

The following files must be placed in the proper folder to run a OpenGL Program.

Libraries (place in the lib\ subdirectory of Visual C++)

- opengl32.lib
- glu32.lib
- glut32.lib

Include files (place in the include\GL\ subdirectory of Visual C++)

- gl.h
- glu.h
- glut.h

Dynamically-linked libraries (place in the \Windows\System subdirectory)

- opengl32.dll
- glu32.dll
- glut32.dll

Working with OpenGL

Opening a window for Drawing

The First task in making pictures is to open a screen window for drawing. The following five functions initialize and display the screen window in our program.

1. glutInit(&argc, argv)

The first thing we need to do is call the glutInit() procedure. It should be called before any other GLUT routine because it initializes the GLUT library. The parameters to glutInit() should be the same as those to main(), specifically main(int argc, char** argv) and glutInit(&argc, argv).

2. glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)

The next thing we need to do is call the glutInitDisplayMode() procedure to specify the display mode for a window.

We must first decide whether we want to use an RGBA (GLUT_RGB) or color-index (GLUT_INDEX) color model. The RGBA mode stores its color buffers as red, green, blue, and alpha color components. Color-index mode, in contrast, stores color buffers in indicies. And for special effects, such as shading, lighting, and fog, RGBA mode provides more flexibility. In general, use RGBA mode whenever possible. RGBA mode is the default.

Another decision we need to make when setting up the display mode is whether we want to use single buffering (GLUT_SINGLE) or double buffering (GLUT_DOUBLE). If we aren't using annimation, stick with single buffering, which is the default.

3. glutInitWindowSize(640,480)

We need to create the characteristics of our window. A call to glutInitWindowSize() will be used to specify the size, in pixels, of our inital window. The arguments indicate the height and width (in pixels) of the requested window.

4. glutInitWindowPosition(100,15)

Similarly, glutInitWindowPosition() is used to specify the screen location for the upper-left corner of our initial window. The arguments, x and y, indicate the location of the window relative to the entire display. This function positioned the screen 100 pixels over from the left edge and 150 pixels down from the top.

5. glutCreateWindow("Example")

To create a window, the with the previously set characteristics (display mode, size, location, etc), the programmer uses the glutCreateWindow() command. The command takes a string as a parameter which may appear in the title bar.

6. glutMainLoop()

The window is not actually displayed until the glutMainLoop() is entered. The very last thing is we have to call this function.

Event Driven Programming

The method of associating a call back function with a particular type of event is called as event driven programming. OpenGL provides tools to assist with the event management.

There are four Glut functions available

1. glutDisplayFunc(mydisplay)

The glutDisplayFunc() procedure is the first and most important event callback function. A callback function is one where a programmer-specified routine can be registered to be called in response to a specific type of event. For example, the argument of glutDisplayFunc(mydisplay) is the function that is called whenever GLUT determines that the contents of the window needs to be redisplayed. Therefore, we should put all the routines that you need to draw a scene in this display callback function.

2. glutReshapeFunc(myreshape)

The glutReshapeFunc() is a callback function that specifies the function that is called whenever the window is resized or moved. Typically, the function that is called when needed by the reshape function displays the window to the new size and redefines the viewing characteristics as desired.

3. glutKeyboardFunc(mykeyboard)

GLUT interaction using keyboard inputs is handled. The command glutKeyboardFunc() is used to run the callback function specified and pass as parameters, the ASCII code of the pressed key, and the x and y coordinates of the mouse cursor at the time of the event.

Special keys can also be used as triggers. The key passed to the callback function, in this case, takes one of the following values (defined in glut.h).

Special keys can also be used as triggers. The key passed to the callback function, in this case, takes one of the following values (defined in glut.h).

GLUT_KEY_UP GLUT_KEY_RIGHT GLUT_KEY_DOWN
GLUT_KEY_PAGE_UP GLUT_KEY_PAGE_DOWN GLUT_KEY_HOME
GLUT_KEY_END GLUT_KEY_INSERT

Example : Skeleton for OpenGL Code

```
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(465, 250);
glutInitWindowPosition(100, 150);
glutCreateWindow("My First Example");
glutDisplayFunc(mydisplay);
glutReshapeFunc(myreshape);
glutMouseFunc(mymouse);
glutKeyboardFunc(mykeyboard);
myinit();
glutMainLoop();
return 0;
}
```

Basic graphics primitives

OpenGL Provides tools for drawing all the output primitives such as points, lines, triangles, polygons, quads etc and it is defined by one or more vertices.

To draw such objects in OpenGL we pass it a list of vertices. The list occurs between the two OpenGL function calls glBegin() and glEnd(). The argument of glBegin() determine which object is drawn.

These functions are

```
glBegin(int mode);
glEnd( void );
```

The parameter mode of the function glBegin can be one of the following:

GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
GL_QUADS

glVertex( ) : The main function used to draw objects is named as glVertex. This function defines a point (or a vertex) and it can vary from receiving 2 up to 4 coordinates.

Format of glVertex Command

When we wish to refer the basic command without regard to the specific arguments and datatypes it is specified as

glVertex*();

Example

//the following code plots three dots

```
glBegin(GL_POINTS);
glVertex2i(100, 50);
glVertex2i(100, 130);
glVertex2i(150, 130);
glEnd( );
```

// the following code draws a triangle

```
glBegin(GL_TRIANGLES);
glVertex3f(100.0f, 100.0f, 0.0f);
glVertex3f(150.0f, 100.0f, 0.0f);
glVertex3f(125.0f, 50.0f, 0.0f);
glEnd( );
```

// the following code draw a lines

```
glBegin(GL_LINES);
glVertex3f(100.0f, 100.0f, 0.0f); // origin of the line
glVertex3f(200.0f, 140.0f, 5.0f); // ending point of the line
glEnd( );
```


OpenGl State

OpenGl keeps track of many state variables, such as current size of a point, the current color of a drawing, the current background color, etc.

The value of a state variable remains active until new value is given.

glPointSize() : The size of a point can be set with glPointSize(), which takes one floating point argument

Example : glPointSize(4.0);

glClearColor() : establishes what color the window will be cleared to. The background color is set with glClearColor(red, green, blue, alpha), where alpha specifies a degree of transparency

Example : glClearColor (0.0, 0.0, 0.0, 0.0); //set black background color

glClear() : To clear the entire window to the background color, we use glClear (GL_COLOR_BUFFER_BIT). The argument GL_COLOR_BUFFER_BIT is another constant built into OpenGL

Example : glClear(GL_COLOR_BUFFER_BIT)

glColor3f() : establishes to use for drawing objects. All objects drawn after this point use this color, until it‟s changed with another call to set the color.

Example:

```
glColor3f(0.0, 0.0, 0.0);          //black
glColor3f(1.0, 0.0, 0.0);          //red
glColor3f(0.0, 1.0, 0.0);          //green
glColor3f(1.0, 1.0, 0.0);          //yellow
glColor3f(0.0, 0.0, 1.0);          //blue
glColor3f(1.0, 0.0, 1.0);          //magenta
glColor3f(0.0, 1.0, 1.0);          //cyan
glColor3f(1.0, 1.0, 1.0);          //white
```


gluOrtho2D(): specifies the coordinate system in two dimension
void `gluOrtho2D` (GLdouble *left*, GLdouble *right*, GLdouble *bottom*,GLdouble *top*);

Example : gluOrtho2D(0.0, 640.0, 0.0, 480.0);

glOrtho() : specifies the coordinate system in three dimension

Example : glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);

glFlush() : ensures that the drawing commands are actually executed rather than stored in a buffer awaiting (ie) Force all issued OpenGL commands to be executed

glMatrixMode(GL_PROJECTION) : For orthographic projection

glLoadIdentity() : To load identity matrix

Example : OpenGL Program to draw three dots (2-Dimension)

```
#include "stdafx.h"
#include "gl/glut.h"
#include <gl/gl.h>
void myInit(void)

{
glClearColor (1.0, 1.0, 1.0, 0.0);
glColor3f (0.0, 0.0, 0.0);
glPointSize(4.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}
void Display(void)
{
glClear (GL_COLOR_BUFFER_BIT);
glBegin(GL_POINTS);
glVertex2i(100, 50);
glVertex2i(100, 130);
glVertex2i(150, 130);
glEnd( );
glFlush();
}
int main (int argc, char **argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(640,480);
glutInitWindowPosition(100,150);
glutCreateWindow("Example");
glutDisplayFunc(Display);
myInit();
```

```
glutMainLoop();
return 0;
}
```

Drawing three dimensional objects & Drawing three dimensional scenes

OpenGL has separate transformation matrices for different graphics features

glMatrixMode(GLenum mode), where mode is one of:

- **GL_MODELVIEW** - for manipulating model in scene
- **GL_PROJECTION** - perspective orientation
- **GL_TEXTURE** - texture map orientation

glLoadIdentity(): loads a 4-by-4 identity matrix into the current matrix

glPushMatrix() : push current matrix stack

glPopMatrix() : pop the current matrix stack

glMultMatrix () : multiply the current matrix with the specified matrix

glViewport() : set the viewport

Example : glViewport(0, 0, width, height);

gluPerspective() : function sets up a perspective projection matrix.

Format : gluPerspective(angle, asratio, ZMIN, ZMAX);

Example : gluPerspective(60.0, width/height, 0.1, 100.0);

gluLookAt() - view volume that is centered on a specified eyepoint

Example : gluLookAt(3.0, 2.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

glutSwapBuffers () : glutSwapBuffers swaps the buffers of the current window if double buffered.

Example for drawing three dimension Objects

```
glBegin(GL_QUADS); // Start drawing a quad primitive
glVertex3f(-1.0f, -1.0f, 0.0f); // The bottom left corner
glVertex3f(-1.0f, 1.0f, 0.0f); // The top left corner
glVertex3f(1.0f, 1.0f, 0.0f); // The top right corner
glVertex3f(1.0f, -1.0f, 0.0f); // The bottom right corner
glEnd();
```

// Triangle

```
glBegin( GL_TRIANGLES );
glVertex3f( -0.5f, -0.5f, -10.0 );
glVertex3f( 0.5f, -0.5f, -10.0 );
glVertex3f( 0.0f, 0.5f, -10.0 );
glEnd();
```

// Quads in different colours

```
glBegin(GL_QUADS);
glColor3f(1,0,0); //red
glVertex3f(-0.5, -0.5, 0.0);
glColor3f(0,1,0); //green
glVertex3f(-0.5, 0.5, 0.0);
glColor3f(0,0,1); //blue
glVertex3f(0.5, 0.5, 0.0);
glColor3f(1,1,1); //white
glVertex3f(0.5, -0.5, 0.0);
glEnd();
```

GLUT includes several routines for drawing these three-dimensional objects:

- cone
- icosahedron
- teapot
- cube
- octahedron
- tetrahedron
- dodecahedron
- sphere
- torus

OpenGL Functions for drawing the 3D Objects

```
glutWireCube(double size);
glutSolidCube(double size);
glutWireSphere(double radius, int slices, int stacks);
glutSolidSphere(double radius, int slices, int stacks);
glutWireCone(double radius, double height, int slices, int stacks);
glutSolidCone(double radius, double height, int slices, int stacks);
glutWireTorus(double inner_radius, double outer_radius, int sides, int rings);
glutSolidTorus(double inner_radius, double outer_radius, int sides, int rings);
glutWireTeapot(double size);
glutSolidTeapot(double size);
```

3D Transformation in OpenGL

glTranslate () : multiply the current matrix by a translation matrix

glTranslated(GLdouble x, GLdouble y, GLdouble z);
void glTranslatef(GLfloat x, GLfloat y, GLfloat z);

x, y, z - Specify the x, y, and z coordinates of a translation vector.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after a call to glTranslate are translated.

Use glPushMatrix and glPopMatrix to save and restore the untranslated coordinate system.

glRotate() : multiply the current matrix by a rotation matrix

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

angle : Specifies the angle of rotation, in degrees.
x, y, z : Specify the x, y, and z coordinates of a vector, respectively.

glScale() : multiply the current matrix by a general scaling matrix

voidglScaled(GLdouble x, GLdouble y, GLdouble z);
void glScalef(GLfloat x, GLfloat y, GLfloat z);

x, y, z : Specify scale factors along the x, y, and z axes, respectively.

Example : Transformation of a Polygon

```
#include "stdafx.h"
#include "gl/glut.h"
#include <gl/gl.h>
void Display(void)
{
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
   glLoadIdentity();
   gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   glColor3f(0.0, 1.0, 0.0);
  glBegin(GL_POLYGON);
        glVertex3f( 0.0, 0.0, 0.0);          // V0 ( 0, 0, 0)
        glVertex3f( 1.0f, 0.0, 0.0);         // V1 ( 1, 0, 0)
```

```
                glVertex3f( 1.0f, 1.0f, 0.0);        // V2 ( 1, 1, 0)
                glVertex3f( 0.5f, 1.5f, 0.0);        // V3 (0.5, 1.5, 0)
                glVertex3f( 0.0, 1.0f, 0.0);         // V4
                ( 0, 1, 0) glEnd();
                 glPushMatrix();
                glTranslatef(1.5,
                2.0, 0.0);
                glRotatef(90.0, 0.0, 0.0, 1.0);
                glScalef(0.5, 0.5,
                0.5);
                glBegin(GL_POL
                YGON);
                glVertex3f( 0.0, 0.0, 0.0);          // V0 ( 0, 0, 0)
                glVertex3f( 1.0f, 0.0, 0.0);         // V1 ( 1, 0, 0)
                glVertex3f( 1.0f, 1.0f, 0.0);        // V2 ( 1, 1, 0)
                glVertex3f( 0.5f, 1.5f, 0.0);        // V3 (0.5, 1.5, 0)
                glVertex3f( 0.0, 1.0f, 0.0);         // V4
        ( 0, 1, 0) glEnd();
        glPop
Matrix();
glFlush();
glutSwapBuff
ers();
}
void Init(void)
{
glClearColor(0.0, 0.0, 0.0, 0.0);
}
void Resize(int width, int height)
{
   glViewport(0, 0, width,height);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(60.0, width/height, 0.1,
   1000.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}
int main(int argc, char **argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_DOUBLE |
   GLUT_RGB); glutInitWindowSize(400, 400);
   glutInitWindowPosition(200, 200);
   glutCreateWindow("Polygon in
```

```
    OpenGL"); Init();
    glutDisplayFunc(
    Display);
    glutReshapeFunc(
    Resize);
    glutMainLoop();
    return 0;
}
```

FREQUENTLY ASKED QUESTIONS

Unit I

1. Explain about different line drawing algorithms.
2. a) What is reflection and shear transformation? Discuss with examples.
   b) Discuss about Sutherland Hodgeman polygon clipping algorithm with example.
3. a) Explain about cohen-sutherland line clipping algorithm
   b) Discuss about homogenous coordinates.
4. Discuss about mid-point ellipse algorithm.
5. Derive the decision parameter used in Bresenham's line drawing algorithm
6. Explain the Cohen-Sutherland algorithm for finding the category of a line segment. Show clearly how each category is handled by the algorithm.
7. a) Explain the perspective projection for projecting 3D objects on a 2D view surface.
   b)Describe 3D clipping
8. Using midpoint ellipse algorithm, generate points on the ellipse with center as origin, major axis 8 units and minor axis 6 units.
9. a) Adapt the Liang-Barsky line-clipping algorithm to polygon clipping.
   b) Write a note on viewing functions.
10. a) Write a routine to reflect an object about an arbitrarily selected plane
    b) Write short notes o 3D clipping.
11. Explain 2- dimensional scaling and shear transformations with examples.
12. a) Explain the various approaches followed in different line-clipping algorithms.
    b) What is the principle of Cyrus-Beck algorithm for clipping a polygon?

Unit II

13. a)Differentiate between parallel and perspective projections.
    b) Explain in brief about 3D viewing pipeline.
14. a) Explain the working of the Sutherland-Hodgeman algorithm for polygon clipp with the help of suitable example.
    b) Compare Liang Barsky algorithm with Cohen Sutherland algorithm.
15. a) Derive the perspective projection transformation matrix.
    b) b) Explain in brief about the working process of 3D clipping.

16. a) Show that a rotation about the origin can be done by performing three shearing Transformations
    b) What is the need of homogeneous coordinates? Give the homogeneous coordinates for translation, rotation and scaling.
17. a) Derive the transformation matrix for rotation about an x-axis in 3D.
    b) Compare the orthographic and oblique types of parallel projections and also Explain the various clipping parameters in 3D clipping.

Unit III

18. Explain about color models (RGB, CMY, YIQ, HSV, HSL)?
19 Explain about Animations
20 Explain "Functions in OPENGL" with example programs(line , triangle square etc)

UNIT IV – RENDERING

Introduction to shading models – Flat and smooth shading – Adding texture to faces – Adding shadows of objects – Building a camera ina program – Creating shaded objects – Rendering texture – Drawing shadows.

Introduction to Shading Models

The mechanism of light reflection from an actual surface is very complicated it depends on many factors. Some of these factors are geometric and others are related to the characteristics of the surface.

A shading model dictates how light is scattered or reflected from a surface. The shading models described here focuses on achromatic light. Achromatic light has brightness and no color, it is a shade of gray so it is described by a single value its intensity.

A shading model uses two types of light source to illuminate the objects in a scene : point light sources and ambient light. Incident light interacts with the surface in three different ways:

- Some is absorbed by the surface and is converted to heat.
- Some is reflected from the surface
- Some is transmitted into the interior of the object

If all incident light is absorbed the object appears black and is known as a black body. If all of the incident light is transmitted the object is visible only through the effects of reflection.

Some amount of the reflected light travels in the right direction to reach the eye causing the object to be seen. The amount of light that reaches the eye depends on the orientation of the surface, light and the observer. There are two different types of reflection of incident light

- Diffuse scattering occurs when some of the incident light slightly penetrates the surface and is re-radiated uniformly in all directions. Scattered light interacts strongly with the surface and so its color is usually affected by the nature of the surface material.
- Specular reflections are more mirrorlike and highly directional. Incident light is directly reflected from its outer surface. This makes the surface looks shinny. In the simplest model the reflected light has the same color as the incident light, this makes the material look like plastic. In a more complex model the color of the specular light varies , providing a better approximation to the shininess of metal surfaces.

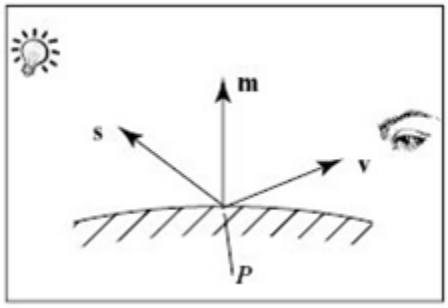
The total light reflected from the surface in a certain direction is the sum of the diffuse component and the specular

component. For each surface point of interest we compute the size of each component that reaches the eye.

Geometric Ingredients For Finding Reflected Light

We need to find three vectors in order to compute the diffuse and specular components. The below fig. shows three principal vectors (s, m and v) required to find the amount of light that reaches the eye from a point P.

Important directions in computing the reflected light



The normal vector , m , to the surface at P.

1. The vector v from P to the viewer"s eye.
2. The vector s from P to the light source.

The angles between these three vectors form the basis of computing light intensities. These angles are normally calculated using world coordinates.

Each face of a mesh object has two sides. If the object is solid , one is inside and the other is outside. The eye can see only the outside and it is this side for which we must compute light contributions.

We shall develop the shading model for a given side of a face. If that side of the face is turned away from the eye there is no light contribution.

How to Compute the Diffuse Component

Suppose that a light falls from a point source onto one side of a face , a fraction of it is re-radiated diffusely in all directions from this side. Some fraction of the re-radiated part reaches the eye, with an intensity denoted by I_d.

An important property assumed for diffuse scattering is that it is independent of the direction from the point P, to the location of the viewer"s eye. This is called omnidirectional scattering , because scattering is uniform in all directions. Therefore I_d is independent of the angle between m and v.

Fig (b) the face is turned partially away from the light source through angle θ. The area subtended is now only cos(θ) , so that the brightness is reduced of S is reduced by this same factor. This relationship between the brightness and surface orientation is called **Lambert's law**.

cos(θ) is the dot product between the normalized versions of s and m. Therefore the strength of the diffuse component:

$$I_d = I_s \rho_d \frac{s.m}{s\ m}$$

I_s is the intensity of the light source and ρ_d is the diffuse reflection coefficient. If the facet is aimed away from the eye this dot product is negative so we need to evaluate I_d to 0. A more precise computation of the diffuse component is :

$$I_d = I_s \rho_d \max \frac{s.m}{|s||m|}, 0$$

The reflection coefficient ρ_d depends on the wavelength of the incident light , the angle θ and various physical properties of the surface. But for simplicity and to reduce computation time, these effects are usually suppressed when rendering images. A reasonable value for ρ_d is chosen for each surface.

Specular Reflection

Real objects do not scatter light uniformly in all directions and so a specular component is added to the shading model. Specular reflection causes highlights which can add reality to a picture when objects are shinny. The behavior of specular light can be explained with Phong model.

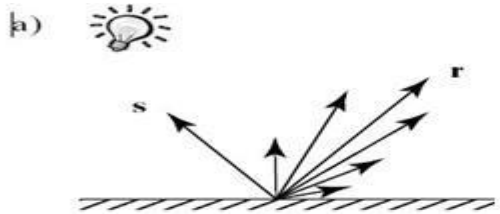
Phong Model

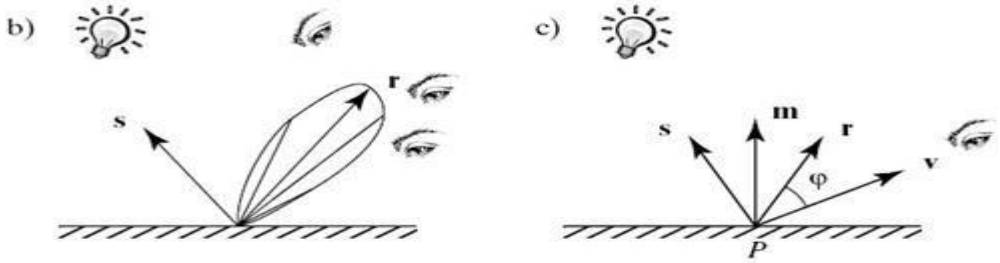
It is easy to apply and the highlights generated by the phong model given an plasticlike appearance , so the phong model is good when the object is made of shinny plastic or glass.

The Phong model is less successful with objects that have a shinny metallic surface.

Fig a) shows a situation where light from a source impinges on a surface and is reflected in different directions.

In this model we discuss the amount of light reflected is greatest in the direction of perfect mirror reflection , r, where the angle of incidence θ equals the angle of reflection. This is the direction in which all light would travel if the surface were a perfect mirror. At the other nearby angles theamount of light reflected diminishes rapidly, Fig





(b) shows this with beam patterns. The distance from P to the beam envelope shows the relative strength of the light scattered in that direction.

Fig(c) shows how to quantify this beam pattern effect . The direction r of perfect reflection depends on both s and the normal vector m to the surface.

The Role of Ambient Light and Exploiting Human Perception

The diffuse and specular components of reflected light are found by simplifying the rules by which physical light reflects from physical surfaces. The dependence of these components on the relative position of the eye , model and light sources greatly improves the reality of a picture.

The simple reflection model does not perfectly renders a scene. An example: shadows are unrealistically deep and harsh, to soften these shadows we add a third light component called ambient light.

With only diffuse and specular reflections, any parts of a surface that are shadowed from the point source receive no light and so are drawn black but in real, the scenes around us are always in some soft nondirectional light. This light arrives by multiple reflections from various objects in the surroundings. But it would be computationally very expensive to model this kind of light.

Ambient Sources and Ambient Reflections

To overcome the problem of totally dark shadows we imagine that a uniform background glow called ambient light exists in the environment. The ambient light source spreads in all directions uniformly.

The source is assigned an intensity I_a. Each face in the model is assigned a value for its ambient reflection coefficient ρ_d, and the term $I_a \rho_a$ is added to the diffuse and specular light that is reaching the eye from each point P on that face. I_a and ρ_a are found experimentally.

Too little ambient light makes shadows appear too deep and harsh., too much makes the picture look washed out and bland.

How to combine Light Contributions

We sum the three light contributions –diffuse, specular and ambient to form the total amount of light I that reaches the eye from point P:

I = ambient + diffuse + specular

$I = I_a \rho_a + I_d \rho_d \times \text{lambert} + I_{sp} \rho_s \times \text{phong}^f$

Where we define the values

$$\text{lambert} = \max\left(0, \frac{s.m}{|s||m|}\right) \text{ and } \text{phong} = \max\left(0, \frac{h.m}{|h||m|}\right)$$

I depends on various source intensities and reflection coefficients and the relative positions of the point P, the eye and the point light source.

$$I_r = I_{ar} \rho_{ar} + I_{dr} \rho_{dr} \times \text{lambert} + I_{spr} \rho_{sr} \times \text{phong}^f$$
$$I_g = I_{ag} \rho_{ag} + I_{dg} \rho_{dg} \times \text{lambert} + I_{spg} \rho_{sg} \times \text{phong}^f$$
$$I_b = I_{ab} \rho_{ab} + I_{db} \rho_{db} \times \text{lambert} + I_{spb} \rho_{sb} \times \text{phong}^f \quad \text{---------- (1)}$$

The above equations are applied three times to compute the red, green and blue components of the reflected light.

The light sources have three types of color : ambient =(I_{ar},I_{ag},I_{ab}) , diffuse=(I_{dr},I_{dg},I_{db}) and specular=(I_{spr},I_{spg},I_{spb}). Usually the diffuse and the specular light colors are the same. The terms lambert and phong^f do not depends on the color component so they need to be calculated once. To do this we need to define nine reflection coefficients:

ambient reflection coefficients: ρ_{ar} , ρ_{ag} and ρ_{ab}
diffuse reflection coefficients: ρ_{dr} , ρ_{dg} and ρ_{db}
specular reflection coefficients: ρ_{sr} , ρ_{sg} and ρ_{sb}

The ambient and diffuse reflection coefficients are based on the color of the surface itself.

The Color of Specular Light

Specular light is mirrorlike , the color of the specular component is same as that of the light source.

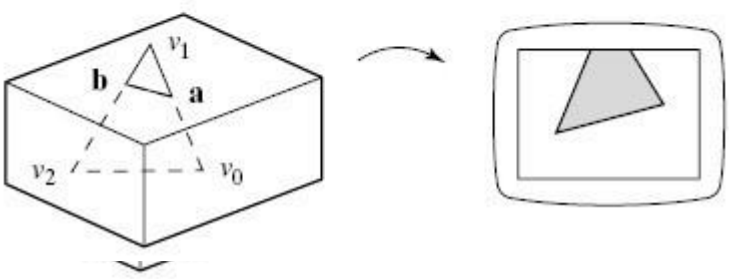
Example: A specular highlight seen on a glossy red apple when illuminated by a yellow light is yellow and not red. This is the same for shiny objects made of plasticlike material.

To create specular highlights for a plastic surface the specular reflection coefficients ρ_{sr} , ρ_{sg} and ρ_{sb} are set to the same value so that the reflection coefficients are gray in nature and do not alter the color of the incident light.

Shading and the Graphics Pipeline

The key idea is that the vertices of a mesh are sent down the pipeline along with their associated vertex normals, and all shading calculations are done on vertices.

The above fig. shows a triangle with vertices v_0,v_1 and v_2 being rendered. Vertex v_i has the normal vector m_i associated with it. These quantities are sent down the pipeline with calls such as :



The call to glNormal3f() sets the "current normal vector" which is applied to all vertices sent using glVertex3f(). The current normal remains current until it is changed with another call to glNormal3f().

The vertices are transformed by the modelview matrix, M so they are then expressed in camera coordinates. The normal vectors are also transformed. Transforming points of a surface by a matrix M causes the normal m at any point to become the normal $M^{-T}m$ on the transformed surface, where M^{-T} is the transpose of the inverse of M.

All quantities after the modelview transformation are expressed in camera coordinates. At this point the shading model equation (1) is applied and a color is attached to each vertex.

The clipping step is performed in homogenous coordinates. This may alter some of the vertices. The below figure shows the case where vertex v_1 of a triangle is clipped off and two new vertices a and b are created. The triangle becomes a quadrilateral. The color at each new vertices must be computed, since it is needed in the actual rendering step.

To Use Light Sources in OpenGL

OpenGL provides a number of functions for setting up and using light sources, as well as for specifying the surface properties of materials.

Create a Light Source

In OpenGL we can define upto eight sources, which are referred through names GL_LIGHT0, GL_LIGHT1 and so on. Each source has properties and must be enabled. Each property has a default value. For example, to create a source located at (3,6,5) in the world coordinates

```
GLfloat myLightPosition[]={3.0 , 6.0,5.0,1.0 };
glLightfv(GL_LIGHT0, GL-POSITION, myLightPosition);
glEnable(GL_LIGHTING);          //enable lighting in general
glEnable(GL_LIGHT0);            //enable source GL_LIGHT0
```

The array myLightPosition[] specifies the location of the light source. This position is passed to glLightfv() along with the name GL_LIGHT0 to attach it to the particular source GL_LIGHT0.

Some sources such as desk lamp are in the scene whereas like the sun are infinitely remote. OpenGL allows us to create both types by using homogenous coordinates to specify light position:(x,y,z,1),

Spotlights

Light sources are point sources by default, meaning that they emit light uniformly in all directions. But OpenGL allows you to make them into spotlights, so they emit light in a restricted set of directions. The fig. shows a spotlight aimed in direction d with a "cutoff angle" of α.

Properties of an OpenGL spotlight

No light is seen at points lying outside the cutoff cone. For vertices such as P, which lie inside the cone, the amount of light reaching P is attenuated by the factor $\cos^{\varepsilon}(\beta)$, where β is the angle between d and a line from the source to P and is the exponent chosen by the user to give the desired falloff of light with angle.

The parameters for a spotlight are set by using glLightf() to set a single value and glLightfv() to set a vector:

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF,45.0); //a cutoff angle 45degree
glLightf(GL_LIGHT0, GL_SPOT_EXPONENT,4.0);          //ε=4.0
GLfloat dir[]={2.0, 1.0, -4.0};          // the spotlight‟s direction
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION,dir);
```

The default values for these parameters are d= (0,0,-1) , α=180 degree and ε=0, which makes a source an omni directional point source.

OpenGL allows three parameters to be set that specify general rules for applying the lighting model. These parameters are passed to variations of the function glLightModel.

The color of global Ambient Light:

The global ambient light is independent of any particular source. To create this light , specify its color with the statements:

```
GLfloat amb[]={ 0.2, 0.3, 0.1, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT,amb);
```

This code sets the ambient source to the color (0.2, 0.3, 0.1). The default value is (0.2, 0.2, 0.2,1.0) so the ambient is always present. Setting the ambient source to a non-zero value makes object in a scene visible even if you have not invoked any of the lighting functions.

Is the Viewpoint local or remote?

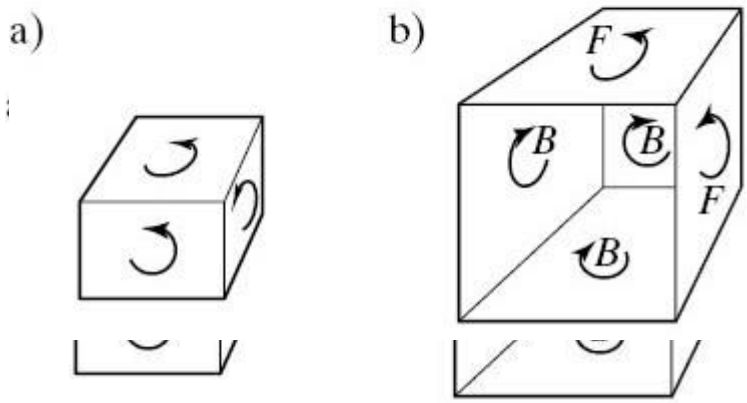
OpenGL computes specular reflection using halfway vector h= s + v. The true directions s and v are different at each vertex. If the light source is directional then s is constant but v varies from vertex to vertex. The rendering speed is increased if v is made constant for all vertices.

As a default OpenGL uses v =(0,0,1),which points along the positive z axis in camera coordinates. The true value of v can be computed by the following statement:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

Are both sides of a Polygon Shaded Properly?

Each polygon faces in a model has two sides, inside and outside surfaces. The vertices of a face are listed in counterclockwise order as seen from outside the object. The camera can see only the outside surface of each face. With hidden surfaces removed, the inside surface of each face is hidden from the eye by some closer face.



In OpenGL the terms "front faces" and "back faces" are used for "inside" and "outside". A face is a front face if its vertices are listed in counterclockwise order as seen by the eye.

The fig.(a) shows a eye viewing a cube which is modeled using the counterclockwise order notion. The arrows indicate the order in which the vertices are passed to OpenGL. For an object that encloses that some space, all faces that are visible to the eye are front faces, and OpenGL draws them with the correct shading. OpenGL also draws back faces but they are hidden by closer front faces.

Fig(b) shows a box with a face removed. Three of the visible faces are back faces. By default, OpenGL does not shade these properly. To do proper shading of back faces we use:

```
glLightModeli (GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
```

When this statement is executed, OpenGL reverses the normal vectors of any back face so that they point towards the viewer, and then it performs shading computations properly. Replacing GL_TRUE with GL_FALSE will turn off this facility.

Moving Light Sources

Lights can be repositioned by suitable uses of glRotated() and glTranslated(). The array position, specified by using

glLightfv(GL_LIGHT0,GL_POSITION,position)

is modified by the modelview matrix that is in effect at the time glLightfv() is called. To modify the position of the light with transformations and independently move the camera as in the following code:

```
void display()
{
   GLfloat position[]={2,1,3,1};        //initial light position
   clear the color and depth buffers
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   glPushMatrix();
            glRotated(….);              //move the light
            glTranslated(…);
            glLightfv(GL_LIGHT0,GL_POSITION,position);
    glPopMatrix();

    gluLookAt(….);                      //set the camera position
    draw the object
    glutSwapBuffers();
}
```

To move the light source with camera we use the following code:

```
GLfloat pos[]={0,0,0,1};
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLightfv(GL_LIGHT0,GL_POSITION,position); //light at (0,0,0)
gluLookAt(….);                      //move the light and the camera
draw the object
```

This code establishes the light to be positoned at the eye and the light moves with the camera.

Working With Material Properties In OpenGL

The effect of a light source can be seen only when light reflects off an object"s surface. OpenGL provides methods for specifying the various reflection coefficients. The coefficients are set with variations of the function glMaterial and they can be specified individually for front and back faces. The code:

```
Glfloat myDiffuse[]={0.8, 0.2, 0.0, 1.0 };
glMaterialfv(GL_FRONT,GL_DIFFUSE,myDiffuse);
```

sets the diffuse reflection coefficients(ρ_{dr} , ρ_{dg} ,ρ_{db}) equal to (0.8, 0.2, 0.0) for all specified front faces. The first parameter of glMaterialfv() can take the following values:

GL_FRONT:Set the reflection coefficient for front faces.
GL_BACK:Set the reflection coefficient for back faces.
GL_FRONT_AND_BACK:Set the reflection coefficient for both front and back faces.

The second parameter can take the following values: GL_AMBIENT: Set the ambient reflection coefficients. GL_DIFFUSE: Set the diffuse reflection coefficients. GL_SPECULAR: Set the specular reflection coefficients.
GL_AMBIENT_AND_DIFFUSE: Set both the ambient and the diffuse reflection coefficients to the same values.
GL_EMISSION: Set the emissive color of the surface.

The emissive color of a face causes it to "glow" in the specified color, independently of any light source.

Shading of Scenes specified by SDL

The scene description language SDL supports the loading of material properties into objects so that they can be shaded properly.

```
light 3 4 5 .8 .8 ! bright white light at (3,4,5)
background 1 1 1 ! white background
globalAmbient .2 .2 .2 ! a dark gray global ambient light
ambient .2 .6 0
diffuse .8 .2 1 ! red material
specular 1 1 1 ! bright specular spots – the color of the source
specularExponent 20 !set the phong exponent
scale 4 4 4 sphere
```

The code above describes a scene containing a sphere with the following material properties:

- ambient reflection coefficients: (ρ_{ar} , ρ_{ag} , ρ_{ab})= (.2, 0.6, 0);
- diffuse reflection coefficients: (ρ_{dr} , ρ_{dg} , ρ_{db})= (0.8,0.2,1.0);
- specular reflection coefficients: (ρ_{sr} , ρ_{sg} , ρ_{sb}) = (1.0,1.0,1.0);
- Phong exponent f = 20.

The light source is given a color of (0.8,0.8,0.8) for both its diffuse and specular component. The global ambient term (I_{ar} , I_{ag} , I_{ab})= (0.2, 0.2, 0.2).

The current material properties are loaded into each object's mtrl field at the time the object is created. When an object is drawn using drawOpenGL(), it first passes its material properties to OpenGL, so that at the moment the object is actually drawn, OpenGL has those properties in its current state.

FLAT SHADING AND SMOOTH SHADING

Different objects require different shading effects. In the modeling process we attached a normal vector to each vertex of each face. If a certain face is to appear as a distinct polygon, we attach the same normal vector to all of its vertices; the normal vector chosen is that indicating the direction normal to the plane of the face. If the face is approximate an underlying surface, we attach to each vertex the normal to the underlying surface at that plane.

The information obtained from the normal vector at each vertex is used to perform different kinds of shading. The main distinction is between a shading method that accentuates the individual polygons (flat shading) and a method that blends the faces to de-emphasize the edges between them (smooth shading).

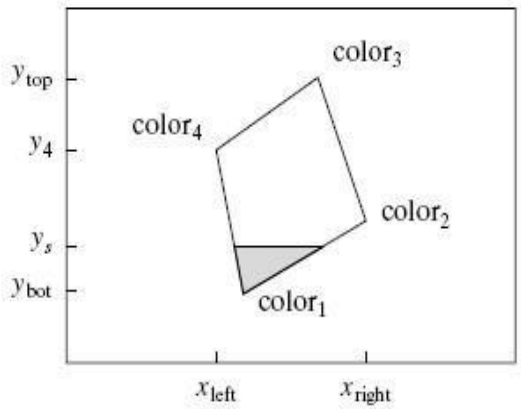
In both kinds of shading, the vertices are passed down the graphics pipeline, shading calculations are performed to attach a color to each vertex and the vertices are converted to screen coordinates and the face is "painted" pixel by pixel with the appropriate color.

Painting a Face

A face is colored using a polygon fill routine. A polygon routine is sometimes called as a tiler because it moves over a polygon pixel by pixel, coloring each pixel. The pixels in a polygon are visited in a regular order usually from bottom to top of the polygon and from left to right.

Polygons intersect are convex. A tiler designed to fill only convex polygons can be very efficient because at each scan line there is unbroken run of pixels that lie inside the polygon. OpenGL uses this property and always fills convex polygons correctly whereas nonconvex polygons are not filled correctly.

A convex quadrilateral whose face is filled with color



The screen coordinates of each vertex is noted. The lowest and highest points on the face are y_{bott} and y_{top}. The tiler first fills in the row at y= y_{bott} , then at y_{bott} + 1, etc. At each scan line y_s, there is a leftmost pixel x_{left} and a rightmost pixel x_{right}. The toler moves from x_{left} to x_{right}, placing the desired color in each pixel. The tiler is implemented as a simple double loop:

```
for (int y= ybott ; y<= ytop; y++) // for each scan line
 {
      find xleft and xright
      for( int x= xleft ; x<= xright; x++) // fill across the scan line
      {
         find the color c for this pixel
         put c into the pixel at (x,y)
      }
 }
```

The main difference between flat and smooth shading is the manner in which the color c is determined in each pixel.

Flat Shading

When a face is flat, like a roof and the light sources are distant , the diffuse light component varies little over different points on the roof. In such cases we use the same color for every pixel covered by the face.

OpenGL offers a rendering mode in which the entire face is drawn with the same color. In this mode, although a color is passed down the pipeline as part of each vertex of the face, the painting algorithm uses only one color value. So the command find the color c for this pixel is not inside the loops, but appears before the loop, setting c to the color of one of the vertices.

Flat shading is invoked in OpenGL using the command glShadeModel(GL_FLAT);

When objects are rendered using flat shading. The individual faces are clearly visible on both sides. Edges between faces actually appear more pronounced than they would on an actual physical object due to a phenomenon in the eye known as lateral inhibition. When there is a discontinuity across an object the eye manufactures a Mach Band at the discontinuity and a vivid edge is seen.

Specular highlights are rendered poorly with flat shading because the entire face is filled with a color that was computed at only one vertex.

Smooth Shading

Smooth shading attempts to de-emphasize edges between faces by computing colors at more points on each face. The two types of smooth shading

- Gouraud shading
- Phong shading

Gouraud Shading

Gouraud shading computes a different value of c for each pixel. For the scan line y_s in the fig. , it finds the color at the leftmost pixel, $color_{left}$, by linear interpolation of the colors at the top and bottom of the left edge of the polygon. For the same scan line the color at the top is $color_4$, and that at the bottom is $color_1$, so $color_{left}$ will be calculated as

$color_{left} = lerp(color_1, color_4, f)$, ----------(1)

where the fraction

$$f = \frac{y_5 - y_{bott}}{y_4 - y_{bott}}$$

varies between 0 and 1 as y_s varies from y_{bott} to y_4. The eq(1) involves three calculations since each color quantity has a red, green and blue component.

$Color_{right}$ is found by interpolating the colors at the top and bottom of the right edge. The tiler then fills across the scan line , linearly interpolating between $color_{left}$ and $color_{right}$ to obtain the color at pixel x:

C(x) = lerp

To increase the efficiency of the fill, this color is computed incrementally at each pixel . that is there is a constant difference between c(x+1) and c(x) so that

C(x+1)=c(x)+

The incremented is calculated only once outside of the inner most loop. The code:

```
for ( int y= ybott; y<=ytop ; y++)          //for each scan line
{
   find xleft and xright
   find colorleft and colorright
   colorinc=( colorright - colorleft) / (xright - xleft);
   for(int x= xleft, c=colorleft; x<=xright; x++, c+=colorinc)
   put c into the pixel at (x,y)
}
```

Computationally Gouraud shading is more expensive than flat shading. Gouraud shading is established in OpenGL using the function:

glShadeModel(GL_SMOOTH);

When a sphere and a bucky ball are rendered using Gouraud shading, the bucky ball looks the same as it was rendered with flat shading because the same color is associated with each vertex of a face. But the sphere looks smoother, as there are no abrupt jumps in color between the neighboring faces and the edges of the faces are gone , replaced by a smoothly varying colors across the object.

Continuity of color across a polygonal edge

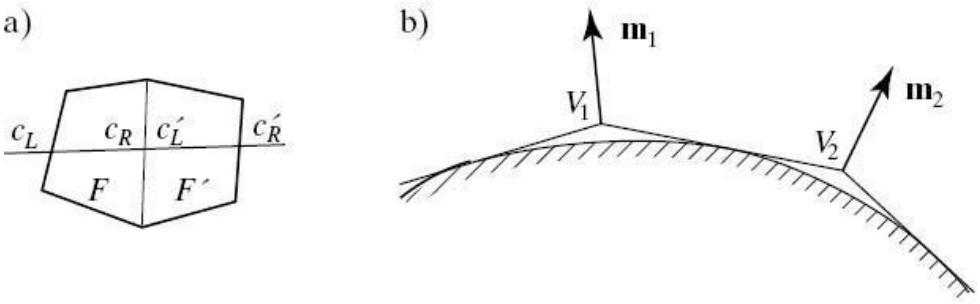


Fig.(a) shows two faces F and F‟ that share an edge. In rendering F, the colors C_L and C_R are used and in rendering F‟, the colors C'_L and C'_R are used. But since C_R equals C'_L, there is no abrupt change in color at the edge along the scan line.

Fig.(b) shows how Gouraud shading reveals the underlying surface. The polygonal surface is shown in cross section with vertices V_1 and V_2. The imaginary smooth surface is also represented. Properly computed vertex normals m_1,m_2 point perpendicularly to this imaginary surface so that the normal for correct shading will be used at each vertex and the color there by found will be correct. The color is then made to vary smoothly between the vertices.

Gouraud shading does not picture highlights well because colors are found by interpolation. Therefore in Gouraud shading the specular component of intensity is suppressed.

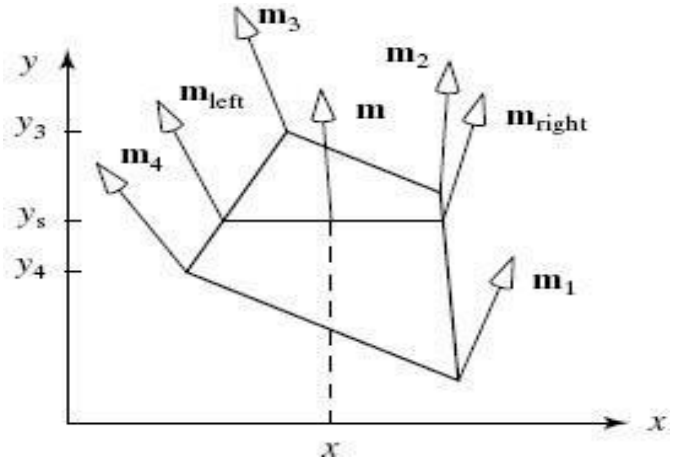
Phong Shading

Highlights are better reproduced using Phong Shading. Greater realism can be achieved with regard to highlights on shiny objects by a better approximation of the normal vector to the face at each pixel this type of shading is called as Phong Shading

When computing Phong Shading we find the normal vector at each point on the face of the object and we apply the shading model there to fig the color we compute the normal vector at each pixel by interpolating the normal vectors at the vertices of the polygon.

The fig shows a projected face with the normal vectors m1, m2, m3 and m4 indicated at the four vertices.

Interpolating normals



For the scan line y_s, the vectors m_{left} and m_{right} are found by linear interpolation

This interpolated vector must be normalized to unit length before it is used in the shading formula once m_{left} and m_{right} are known they are interpolated to form a normal vector at each x along the scan line that vector is used in the shading calculation to form the color at the pixel.

In Phong Shading the direction of the normal vector varies smoothly from point to point and more closely approximates that of an underlying smooth surface the production of specular highlights are good and more realistic renderings produced.

Drawbacks of Phong Shading

- Relatively slow in speed
- More computation is required per pixel

Note: OpenGL does not support Phong Shading

Adding texture to faces

The realism of an image is greatly enhanced by adding surface texture to various faces of a mesh object.

The basic technique begins with some texture function, texture(s,t) in texture space , which has two parameters s and t. The function texture(s,t) produces a color or intensity value for each value of s and t between 0(dark)and 1(light). The two common sources of textures are

- Bitmap Textures
- Procedural Textures

Bitmap Textures

Textures are formed from bitmap representations of images, such as digitized photo. Such a representation consists of an array txtr[c][r] of color values. If the array has C columns and R rows, the indices c and r vary from 0 to C-1 and R-1 resp.,. The function texture(s,t) accesses samples in the array as in the code:

```
Color3 texture (float s, float t)
{
    return txtr[ (int) (s * C)][(int) (t * R)];
}
```

Where Color3 holds an RGB triple.

Example: If R=400 and C=600, then the texture (0.261, 0.783) evaluates to txtr[156][313]. Note that a variation in s from 0 to 1 encompasses 600 pixels, the variation in t encompasses 400 pixels. To avoid distortion during rendering , this texture must be mapped onto a rectangle with aspect ratio 6/4.

Procedural Textures

Textures are defined by a mathematical function or procedure. For example a spherical shape could be generated by a function:

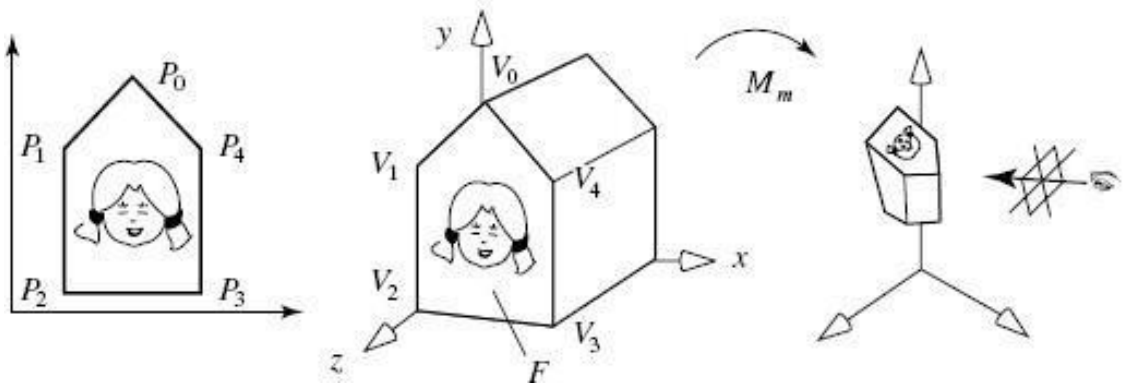
```
float fakesphere( float s, float t)
{
  float r= sqrt((s-0.5) * (s-0.5)+ (t-0.5) * (t-0.5));
  if (r < 0.3) return 1-r/0.3;        //sphere intensity
  else return 0.2;                    //dark background
}
```

This function varies from 1(white) at the center to 0 (black) at the edges of the sphere.

Painting the Textures onto a Flat Surface

Texture space is flat so it is simple to paste texture on a flat surface.

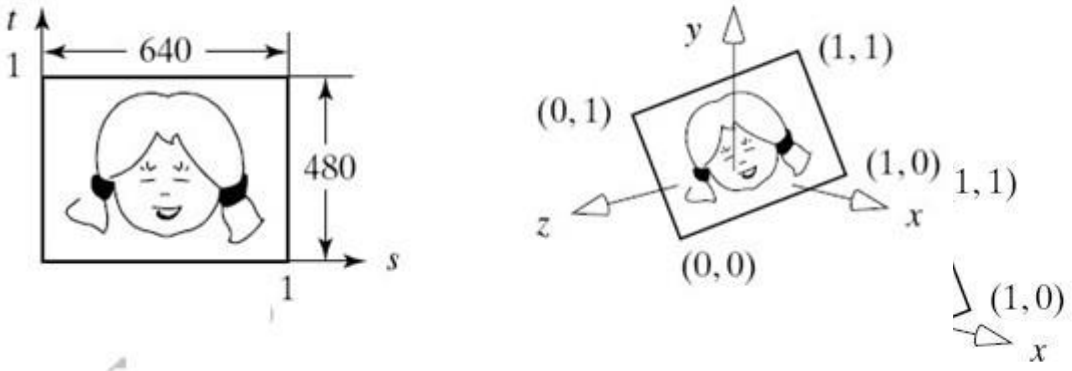
Mapping texture onto a planar polygon

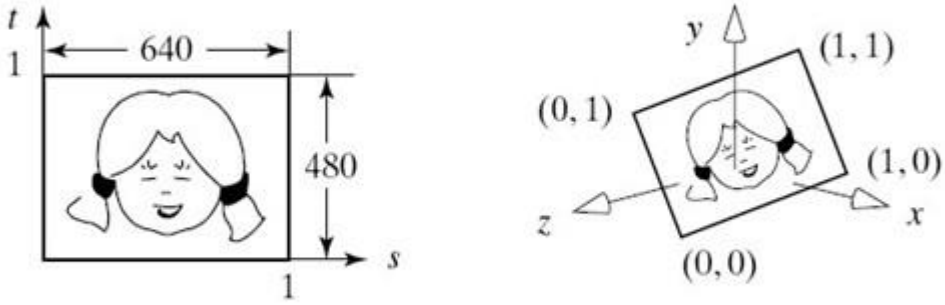


The fig. shows a texture image mapped to a portion of a planar polygon,F. We need to specify how to associate points on the texture with points on F.

In OpenGL we use the function glTexCoord2f() to associate a point in texture space $P_i=(s_i,t_i)$ with each vertex V_i of the face. the function glTexCoord2f(s,t)sets the current texture coordinate to (s,y). All calls to glVertex3f() is called after a call to glTexCoord2f(), so each vertex gets a new pair of texture coordinates.

Example to define a quadrilateral face and to position a texture on it, we send OpenGL four texture coordinates and four 3D points, as follows:





Mapping a Square to a Rectangle

The fig. shows the a case where the four corners of the texture square are associated with the four corners of a rectangle. In this example, the texture is a 640-by-480 pixel bit map and it is pasted onto a rectangle with aspect ratio 640/480, so it appears without distortion.

Adding Texture Coordinates to Mesh Objects

A mesh objects has three lists

- The vertex list
- The normal vector list
- The face list

We need to add texture coordinate to this list, which stores the coordinates (s_i, t_i) to be associated with various vertices. We can add an array of elements of the type

class TxtrCoord(public : float s,t;);

to hold all of the coordinate pairs of the mesh. The two important techniques to treat texture for an object are:

1. The mesh object consists of a small number of flat faces, and a different texture is to be applied to each. Each face has only a sigle normal vector, but its own list of texture coordinates. So the following data are associated with each face:
   - the number of vertices in the face.
   - the index of normal vector to the face.
   - a list of indices of the vertices.
   - a list of indices of the texture coordinates.
2. The mesh represents a smooth underlying object and a single texture is to wrapped around it. Each vertex has associated with it a specific normal vector and a particular texture coordinate pair. A single index into the vertex, normal vector and texture lists is used for each vertex. The data associated with the face are:
   - The number of vertices in the face
   - list of indices of the vertices.

Rendering the Texture

Rendering texture in a face F is similar to Gouraud Shading. It proceeds across the face pixel by pixel. For each pixel it must determine the corresponding texture coordinates (s,t), access the texture and set the pixel to the proper texture color. Finding the coordinated (s,t) should be done carefully.

Painting the Texture by Modulating the Reflection Coefficient

The color of an object is the color of its diffuse light component. Therefore we can make the texture appear to be painted onto the surface by varying the diffuse reflection coefficient. The texture function modulates the value of the reflection coefficient from point to point. We replace eq(1) with

$I = \text{texture}(s,t)\,[I_a \rho_a + I_d \rho_d \times \text{lambert}\,] + I_{sp} \rho_s \times \text{phong}^f$

For appropriate values of s and t. Phong specular reflections are the color of the source and not the object so highlights do not depend on the texture. OpenGL does this type of texturing using

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
          GL_MODULATE);
```

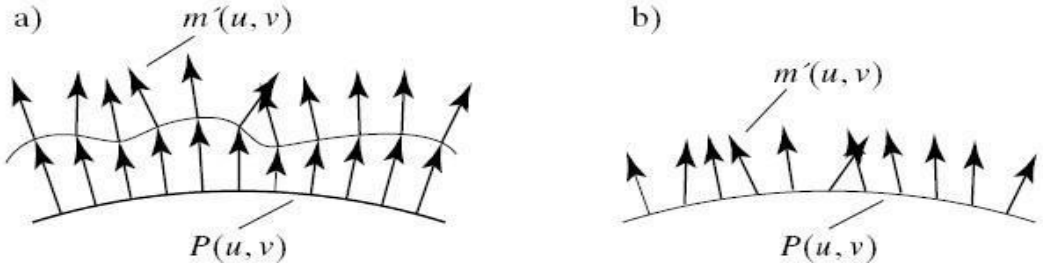
Simulating Roughness by Bump Mapping

Bump mapping is a technique developed by Blinn, to give a surface a wrinkled or dimpled appearance without struggling to model each dimple itself. One problem associated with applying bump mapping to a surface like a teapot is that since the model does not contain the dimples , the object‟s outline caused by a shadow does not show dimples and it is smooth along each face.

The goal is to make a scalar function texture(s,t) disturb the normal vector at each spot in a controlled fashion. This disturbance should depend only on the shape of the surface and the texture.

On the nature of bump mapping

The fig. shows in cross section how bump mapping works.



Suppose the surface is represented parametrically by the function P(u,v) and has unit normal vector m(u,v). Suppose further that the 3D point at(u*,v*) corresponds to texture at (u*,v*).

Blinn‟s method simulates perturbing the position of the true surface in the direction of the normal vector by an amount proportional to the texture (u*,v*);that is

P‟(u*,V*) = P(u*,v*)+texture(u*,v*)m(u*,v*).

Figure(a) shows how this techniques adds wrinkles to the surface. The disturbed surface has a new normal vector m‟(u*,v*)at each point. The idea is to use this disturbed normal as if it were “attached” to the original undisturbed surface at each point, as shown in figure (b). Blinn has demonstrated that a good approximation to m‟(u*,v*) is given by

m‟(u*,v*) =m(u*,v*)+d(u*,v*)

Where the perturbation vector d is given by

$d(u^*,v^*) = (m \times p_v)\ texture_u - (m \times p_u)\ texture_v$.

In which $texture_u$, and $texture_v$ are partial derivatives of the texture function with respect to u and v respectively. Further p_u and p_v are partial derivative of P(u,v) with respect to u and v, respectively. all functions are evaluated at(u*,V*).Note that the perturbation function depends only on the partial derivatives of the texture(),not on texture()itself.

Reflection Mapping

This technique is used to improve the realism of pictures , particularly animations. The basic idea is to see reflections in an object that suggest the world surrounding that object.

The two types of reflection mapping are

- Chrome mapping

  A rough and blurry image that suggests the surrounding environment is reflected in the object as you would see in an object coated with chrome.
- Environment mapping

  A recognizable image of the surrounding environment is seen reflected in the object. Valuable visual clues are got from such reflections particularly when the object is moving.

ADDING SHADOWS OF OBJECTS

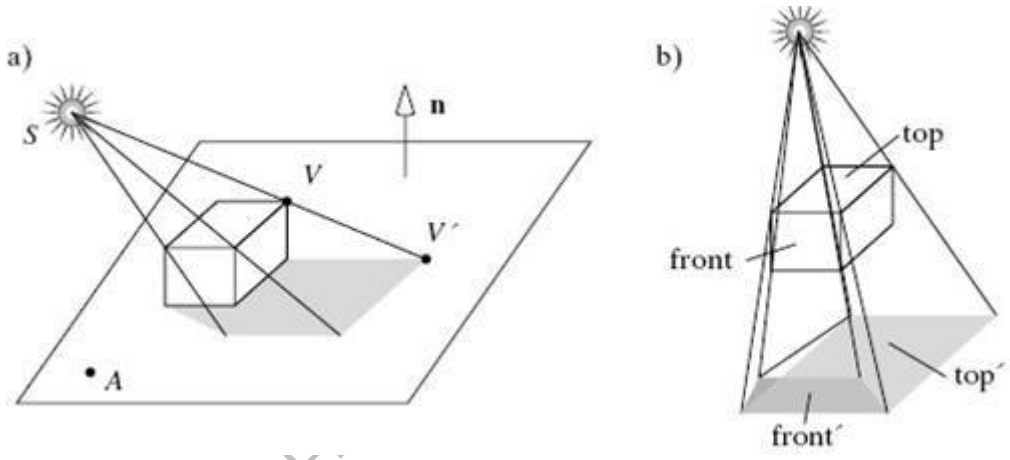
Shadows make an image more realistic. The way one object casts a shadow on another object gives important visual clues as to how the two objects are positioned with respect to each other. Shadows conveys lot of information as such, you are getting a second look at the object from the view point of the light source. There are two methods for computing shadows:

- Shadows as Texture
- Creating shadows with the use of a shadow buffer

Shadows as Texture

The technique of "painting" shadows as a texture works for shadows that are cast onto a flat surface by a point light source. The problem is to compute the shape of the shadow that is cast.

Computing the shape of a shadow



Fig(a) shows a box casting a shadow onto the floor. The shape of the shadow is determined by the projections of each of the faces of the box onto the plane of the floor, using the light source as the center of projection.

Fig(b) shows the superposed projections of two of the faces. The top faces projects to top" and the front face to front".

This provides the key to drawing the shadow. After drawing the plane by the use of ambient, diffuse and specular light contributions, draw the six projections of the box‟s faces on the plane, using only the ambient light. This technique will draw the shadow in the right shape and color. Finally draw the box.

Building the "Projected" Face

To make the new face F‟ produced by F, we project each of the vertices of F onto the plane. Suppose that the plane passes through point A and has a normal vector n. Consider projecting vertex V, producing V‟. V‟ is the point where the ray from source at S through V hits the plane, thispointis

$$= S + (V - S)\frac{n.(A - S)}{n.(V - S)}$$

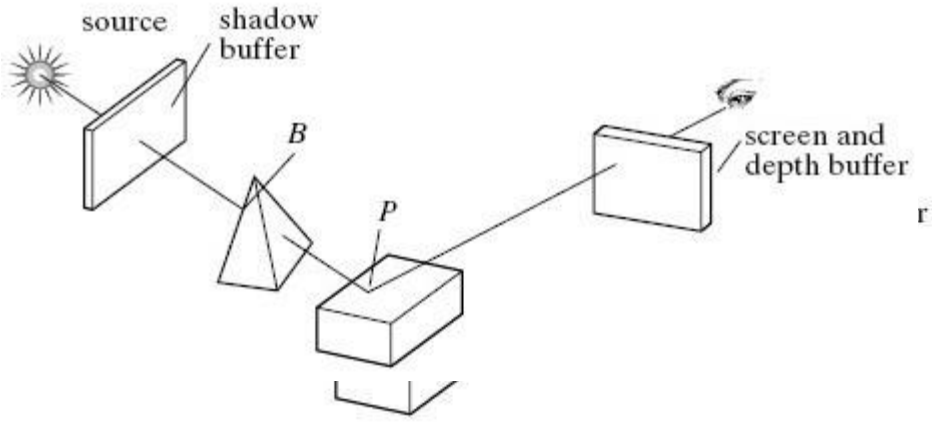
Creating Shadows with the use of a Shadow buffer

This method uses a variant of the depth buffer that performs the removal of hidden surfaces. An auxiliary second depth buffer called a shadow buffer is used for each light source. This requires lot of memory.

This method is based on the principle that any points in a scene that are hidden from the light source must be in shadow. If no object lies between a point and the light source, the point is not in shadow.

The shadow buffer contains a depth picture of the scene from the point of view of the light source. Each of the elements of the buffer records the distance from the source to the closest object in the associated direction. Rendering is done in two stages:

1) Loading the shadow buffer



The shadow buffer is initialized with 1.0 in each element, the largest pseudo depth possible. Then through a camera positioned at the light source, each of the scenes is rasterized but only the pseudo depth of the point on the face is tested. Each element of the shadow buffer keeps track of the smallest pseudo depth seen so far.

Using the shadow buffer. The fig. shows a scene being viewed by the usual eye camera and a source camera located at the light source. Suppose that point P is on the ray from the source through the shadow buffer pixel d[i][j] and that point B on the pyramid is also on this ray. If the pyramid is present d[i][j] contains the pseudo depth to B; if the pyramid happens to be absent d[i][j] contains the pseudo depth to P.

The shadow buffer calculation is independent of the eye position, so in an animation in which only the eye moves, the shadow buffer is loaded only once. The shadow buffer must be recalculated whenever the objects move relative to the light source.

2) Rendering the scene

Each face in the scene is rendered using the eye camera. Suppose the eye camera sees point P through pixel p[c][r]. When rendering p[c][r], we need to find

- The pseudo depth D from the source to p
- The index location [i][j] in the shadow buffer that is to be tested and
- The value d[i][j] stored in the shadow buffer

If d[i][j] is less than D, the point P is in the shadow and p[c][r] is set using only ambient light. Otherwise P is not in shadow and p[c][r] is set using ambient, diffuse and specular light.

BUILDING A CAMERA IN A PROGRAM

To have a finite control over camera movements, we create and manipulate our own camera in a program. After each change to this camera is made, the camera tells OpenGL what the new camera is.

We create a Camera class that does all things a camera does. In a program we create a Camera object called cam, and adjust it with functions such as the following:

```
cam.set(eye, look, up);        // initialize the camera
cam.slide(-1, 0, -2);    //slide the camera forward and to the left
cam.roll(30);            // roll it through 30 degree
cam.yaw(20);             // yaw it through 20 degree
```

The Camera class definition:

```
class Camera {
      private:
        Point3 eye;
        Vector3 u, v, n;
        double viewAngle, aspect, nearDist, farDist; //view volume shape
        void setModelViewMatrix();   //tell OpenGL where the camera is
public:
        Camera();                    //default constructor
        void set(Point3 eye, Point3 look, Vector3 up);   //like gluLookAt()
        void roll(float, angle);     //roll it
        void pitch(float, angle);    // increase the pitch
        void yaw(float, angle);      //yaw it
        void slide(float delU, float delV, float delN);     //slide it
        void setShape(float vAng, float asp, float nearD, float farD);
};
```

The Camera class definition contains fields for eye and the directions u, v and n. Point3 and Vector3 are the basic data types. It also has fields that describe the shape of the view volume: viewAngle, aspect, nearDistandfarDist.

The utility routine setModelViewMatrix() communicates the modelview matrix to OpenGL. It is used only by member functions of the class and needs to be called after each change is made to the camera"s position. The matrix

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

The utility routines set() and setModelViewMatrix()

```
void Camera :: setModelViewMatrix(void)
{   //load modelview matrix with existing camera values
        float m[16];
        Vector3 eVec(eye.x, eye.y, eye.z);    //a vector version of eye
        m[0]= u.x ; m[4]= u.y ; m[8]=   u.z ;   m[12]= -eVec.dot(u);
        m[1]= v.x  ; m[5]= v.y ; m[9]=   v.z ;   m[13]= -eVec.dot(v);
        m[2]= n.x ; m[6]= n.y ; m[10]= y.z ;   m[14]= -eVec.dot(n);
        m[3]= 0    ; m[7]= 0    ; m[11]= 0   ;   m[15]= 1.0            ;
        glMatrixMode(GL_MODELVIEW);
        glLoadMatrixf(m);              //load OpenGL"s modelview matrix
}
void Camera :: set (Point3 eye, Point3 look, Vector3 up)
{ // Create a modelview matrix and send it to OpenGL
        eye.set(Eye);                                  // store the given eye position
        n.set(eye.x – look.x, eye.y – look.y, eye.z – look.z);      // make n
        u.set(up.cross(n));                            //make u= up X n
        n.normalize();                                 // make them unit length
        u.normalize();
        v.set(n.cross(u));                             // make v= n X u
        setModelViewMatrix();                          // tell OpenGL
}
```

The method set() acts like gluLookAt(): It uses the values of eye, look and up to compute u, v and n according to equation:

n= eye – look,
u = up X n
and
v = n X u. It places this information in the camera"s fields and Communicates it to OpenGL.

The routine setShape() is simple. It puts the four argument values into the appropriate camera fields and then calls

gluPerspective(viewangle, aspect, nearDist, farDist)

along with

glMatrixMode(GL_PROJECTION)

and

glLoadIdentity()

to set the projection matrix.

The central camera functions are slide(), roll(), yaw() and pitch(), which makes relative changes to the camera"s position and orientation.

Flying the camera

The user flies the camera through a scene interactively by pressing keys or clicking the mouse. For instance,

- pressing u will slide the camera up some amount
- pressing y will yaw the camera to the left
- pressing f will slide the camera forward

The user can see different views of the scene and then changes the camera to a better view and produce a picture. Or the user can fly around a scene taking different snapshots. If the snapshots are stored and then played back, an animation is produced of the camera flying around the scene.

There are six degrees of freedom for adjusting a camera: It can be slid in three dimensions and it can be rotated about any of three coordinate axes.

Sliding the Camera

Sliding the camera means to move it along one of its own axes that is, in the u, v and n direction without rotating it. Since the camera is looking along the negative n axis, movement along n is forward or back. Movement along u is left or right and along v is up or down.

To move the camera a distance D along its u axis, set eye to eye + Du. For convenience ,we can combine the three possible slides in a single function:

slide(delU, delV, delN)

slides the camera amount delU along u, delV along v and delN along n.
The code is as follows:

```
void Camera : : slide(float delU, float delV, float delN)
{
        eye.x += delU * u.x + delV * v.x + delN * n.x;
        eye.y += delU * u.y + delV * v.y + delN * n.y;
        eye.z += delU * u.z + delV * v.z + delN * n.z;
        setModelViewMatrix();
```

UNIT V FRACTALS

Fractals and Self similarity – Peano curves – Creating image by iterated functions –Mandelbrot sets – Julia Sets – Random Fractals – Overview of Ray Tracing –Intersecting rays with other primitives – Adding Surface texture – Reflections and Transparency – Boolean operations on Objects

Computers are good at repetition. In addition, the high precision with which modern computers can do calculations allows an algorithm to take closer look at an object, to get greater levels of details.

Computer graphics can produce pictures of things that do not even exist in nature or perhaps could never exist. We will study the inherent finiteness of any computer generated picture. It has finite resolution and finite size, and it must be made in finite amount of time. The pictures we make can only be approximations, and the observer of such a picture uses it just as a hint of what the underlying object really looks like.

FRACTALS AND SELF-SIMILARITY

Many of the curves and pictures have a particularly important property called self-similar. This means that they appear the same at every scale: No matter how much one enlarges a picture of the curve, it has the same level of detail.

Some curves are exactly self-similar, whereby if a region is enlarged the enlargement looks exactly like the original.

Other curves are statistically self-similar, such that the wiggles and irregularities in the curve are the same “on the average”, no matter how many times the picture is enlarged. Example: Coastline.

Successive Refinement of Curves

A complex curve can be fashioned recursively by repeatedly “refining” a simple curve. The simplest example is the Koch curve, discovered in1904 by the Swedish mathematician Helge von Koch. The curve produces an infinitely long line within a region of finite area.

Successive generations of the Koch curve are denoted K_0, K_1, K_2….The zeroth generation shape K_0 is a horizontal line of length unity.

Two generations of the Koch curve

To create K_1 , divide the line K_0 into three equal parts and replace the middle section with a triangular bump having sides of length 1/3. The total length of the line is 4/3. The second order curve K_2, is formed by building a bump on each of the four line segments of K_1.To form K_{n+1} from K_n:

Subdivide each segment of K_n into three equal parts and replace the middle part with a bump in the shape of an equilateral triangle.

In this process each segment is increased in length by a factor of 4/3, so the total length of the curve is 4/3 larger than that of the

previous generation. Thus K_i has total length of $(4/3)^i$, which increases as i increases. As i tends to infinity, the length of the curve becomes infinite.

The first few generations of the Koch snowflake

The Koch snowflake of the above figure is formed out of three Koch curves joined together. The perimeter of the ith generations shape S_i is three times length of a Koch curve and so is $3(4/3)^i$, which grows forever as i increases. But the area inside the Koch snowflake grows quite slowly. So the edge of the Koch snowflake gets rougher and rougher and longer and longer, but the area remains bounded. Koch snowflake s_3, s_4 and s_5

The Koch curve K_n is self-similar in the following ways: Place a small window about some portion of K_n, and observe its ragged shape. Choose a window a billion times smaller and observe its shape. If n is very large, the curve appears to be have same shape and roughness. Even if the portion is enlarged another billion times, the shape would be the same.

We call n the order of the curve K_n, and we say the order –n Koch curve consists of four versions of the order (n-1) Koch curve.To draw K_2 we draw a smaller version of K_1 , then turn left $60°$, draw K_1 again, turn right $120°$, draw K_1 a third time. For snowflake this routine is performed just three times, with a $120°$ turn in between.

The recursive method for drawing any order Koch curve is given in the following pseudocode:

To draw K_n:

```
if ( n equals 0 ) Draw a straight line;
else {
        Draw Kn-1;
        Turn left 60°;

        Draw Kn-1;
        Turn right 120 ;

        Draw Kn-1;
        Turn left 60 ;
        Draw Kn-1;
      }
```

Drawing a Koch Curve

```
Void drawKoch (double dir, double len, int n)
{
// Koch to order n the line of length len
// from CP in the direction dir

        double dirRad= 0.0174533 * dir;      // in radians
        if (n ==0)
           lineRel(len * cos(dirRad), len * sin(dirRad));
        else {
          n--;              //reduce the order
          len /=3;          //and the length
          drawKoch(dir, len, n);
          dir +=60;
          drawKoch(dir, len, n);
          dir -=120;
          drawKoch(dir, len, n);
          dir +=60;
          drawKoch(dir, len, n);
         }
  }
```

The routine drawKoch() draws K_n on the basis of a parent line of length len that extends from the current position in the direction dir. To keep track of the direction of each child generation, the parameter dir is passed to subsequent calls of Koch().

Creating An Image By Means of Iterative Function Systems

Another way to approach infinity is to apply a transformation to a picture again and again and examine the results. This technique also provides an another method to create fractal shapes. that it draws gray scale and color images of objects. The image is viewed as a collection of pixels and at each iteration the transformed point lands in one of the pixels. A counter is kept for each pixel and at the completion of the game the number of times each pixel has been visited is converted into a color according to some mapping.

5.3.4 Finding the IFS; Fractal Image Compression

Dramatic levels of image compression provide strong motivation for finding an IFS whose attractor is the given image. A image contains million bytes of data, but it takes only hundreds or thousands of bytes to store the coefficients of the affine maps in the IFS.

Fractal Image Compression and regeneration The original image is processed to create the list of affine maps, resulting in a greatly compressed representation of the image.

In the decompression phase the list of affine maps is used and an algorithm such as the Chaos Game reconstructs the image. This compression scheme is lossy, that is the image I′ that is generated by the

game during decompression is not a perfect replica of the original image I.

THE MANDELBROT SET

Graphics provides a powerful tool for studying a fascinating collection of sets that are the most complicated objects in mathematics.

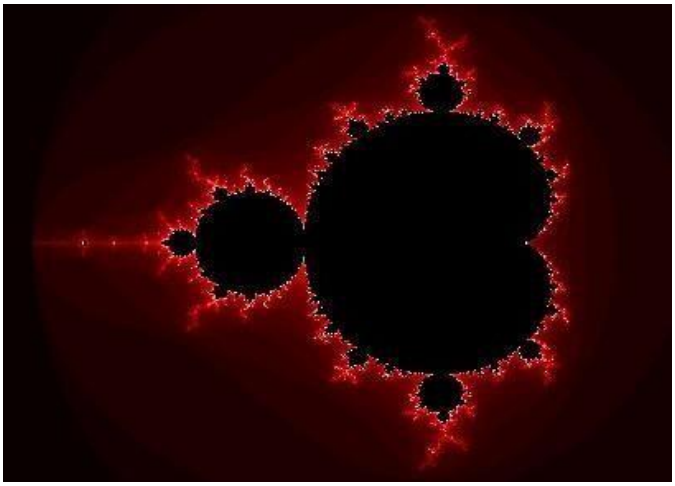
Julia and Mandelbrot sets arise from a branch of analysis known as iteration theory, which asks what happens when one iterates a function endlessly. Mandelbrot used computer graphics to perform experiments.

Mandelbrot Sets and Iterated Function Systems

A view of the Mandelbrot set is shown in the below figure. It is the black inner portion, which appears to consist of a cardoid along with a number of wartlike circles glued to it.

Its border is complicated and this complexity can be explored by zooming in on a portion of the border and computing a close up view. Each point in the figure is shaded or colored according to the outcome of an experiment run on an IFS.

The Mandelbrot set



The Iterated function systems for Julia and Mandelbrot sets

The IFS uses the simple function

$$f(z) = z^2 + c \quad \text{----------(1)}$$

where c is some constant. The system produces each output by squaring its input and adding c. We assume that the process begins with the starting value s, so the system generates the sequence of values or orbit

$$d_1 = (s)^2 + c$$
$$d_2 = ((s)^2 + c)^2 + c$$
$$d_3 = (((s)^2 + c)^2 + c)^2 + c$$
$$d_4 = ((((s)^2 + c)^2 + c)^2 + c)^2 + c \quad \text{---------- (2)}$$

The orbit depends on two ingredients

- the starting point s
- the given value of c

Given two values of s and c how do points d_k along the orbit behaves as k gets larger and larger? Specifically, does the orbit remain finite or explode. Orbits that remain finite lie in their corresponding Julia or Mandelbrot set, whereas those that explode lie outside the set.

When s and c are chosen to be complex numbers , complex arithmetic is used each time the function is applied. The Mandelbrot and Julia sets live in the complex plane – plane of complex numbers.

The IFS works well with both complex and real numbers. Both s and c are complex numbers and at each iteration we square the previous result and add c. Squaring a complex number z = x + yi yields the new complex number:

$$(x + yi)^2 = (x^2 - y^2) + (2xy)^i \quad \text{----------} \quad (3)$$

having real part equal to $x^2 - y^2$ and imaginary part equal to 2xy.

Some Notes on the Fixed Points of the System

It is useful to examine the fixed points of the system f(.) =(.)2 + c . The behavior of the orbits depends on these fixed points that is those complex numbers z that map into themselves, so that $z^2 + c = z$. This gives us the quadratic equation $z^2 - z + c = 0$ and the fixed points of the system are the two solutions of this equation, given by

$$p_+, p_- = \frac{1}{2} \pm \sqrt{\frac{1}{4} - c} \quad \text{----------} \quad (4)$$

If an orbit reaches a fixed point, p its gets trapped there forever. The fixed point can be characterized as attracting or repelling. If an orbit flies close to a fixed point p, the next point along the orbit will be forced

- closer to p if p is an attracting fixed point
- farther away from p if p is a repelling a fixed point.

If an orbit gets close to an attracting fixed point, it is sucked into the point. In contrast, a repelling fixed point keeps the orbit away from it.

Defining the Mandelbrot Set

The Mandelbrot set considers different values of c, always using the starting point s =0. For each value of c, the set reports on the nature of the orbit of 0, whose first few values are as follows:

orbit of 0: 0, c, c^2+c, $(c^2+c)^2+c$, $((c^2+c)^2+c)^2+c$,……..

For each complex number c, either the orbit is finite so that how far along the orbit one goes, the values remain finite or the orbit explodes that is the values get larger without limit. The Mandelbrot set denoted by M, contains just those values of c that result in finite orbits:

- The point c is in M if 0 has a finite orbit.
- The point c is not in M if the orbit of 0 explodes.

Definition:

The Mandelbrot set M is the set of all complex numbers c that produce a finite orbit of 0.

If c is chosen outside of M, the resulting orbit explodes. If c is chosen just beyond the border of M, the orbit usually thrashes around the plane and goes to infinity.

If the value of c is chosen inside M, the orbit can do a variety of things. For some c's it goes immediately to a fixed point or spirals into such a point.

Computing whether Point c is in the Mandelbrot Set

A routine is needed to determine whether or not a given complex number c lies in M. With a starting point of s=0, the routine must examine the size of the numbers d_k along the orbit. As k increases the value of d_k either explodes(c is not in M) or does not explode(c is in M).

At each iteration, the current d_k resides in the pair (dx,dy) which is squared using eq(3) and then added to (cx,cy) to form the next d value. The value $|d_k|^2$ is kept in fsq and compared with 4. The dwell() function plays a key role in drawing the Mandelbrot set.

Drawing the Mandelbrot Set

To display M on a raster graphics device. To do this we set up a correspondence between each pixel on the display and a value of c, and the dwell for that c value is found. A color is assigned to the pixel, depending on whether the dwell is finite or has reached its limit.

The simplest picture of the Mandelbrot set just assign black to points inside M and white to those outside. But pictures are more appealing to the eye if a range of color is associated with points outside M. Such points all have dwells less than the maximum and we assign different colors to them on the basis of dwell size.

The user specifies how large the desired image is to be on the screen that is

- the number of rows, rows
- the number of columns, cols

JULIA SETS

Like the Mandelbrot set, Julia sets are extremely complicated sets of points in the complex plane. There is a different Julia set, denoted J_c for each value of c. A closely related variation is the filled-in Julia set, denoted by K_c, which is easier to define.

The Filled-In Julia Set K_c

In the IFS we set c to some fixed chosen value and examine what happens for different starting point s. We ask how the orbit of starting point s behaves. Either it explodes or it doesn't. If it is finite , we say the starting point s is in K_c, otherwise s lies outside of K_c.

Definition:

The filled-in Julia set at c, K_c, is the set of all starting points whose orbits are finite.

When studying K_c, one chooses a single value for c and considers different starting points. K_c should be always symmetrical about the origin, since the orbits of s and –s become identical after one iteration.

Drawing Filled-in Julia Sets

A starting point s is in K_c, depending on whether its orbit is finite or explodes, the process of drawing a filled-in Julia set is almost similar to Mandelbrot set. We choose a window in the complex plane and associate pixels with points in the window. The pixels correspond to different values of the starting point s. A single value of c is chosen and then the orbit for each pixel position is examined to see if it explodes and if so, how quickly does it explodes.

Pseudocode for drawing a region of the Filled-in Julia set

```
for(j=0; j<rows; j++)
   for(i=0; i<cols; i++)
      {
         find the corresponding s value in equation (5)
         estimate the dwell of the orbit
         find Color determined by estimated dwell
         setPixel( j , k, Color);
      }
```

The dwell() must be passed to the starting point s as well as c. Making a high-resolution image of a K_c requires a great deal of computer time, since a complex calculation is associated with every pixel.

Notes on Fixed Points and Basins of Attraction

If an orbit starts close enough to an attracting fixed point, it is sucked into that point. If it starts too far away, it explodes. The set of points that are sucked in forms a so called basin of attraction for the fixed point p. The set is the filled-in Julia set K_c. The fixed point which lies inside the circle $|z| = ½$ is the attracting point.

All points inside K_c, have orbits that explode. All points inside K_c, have orbits that spiral or plunge into the attracting fixed point. If the starting point is inside K_c, then all of the points on the orbit must also be inside K_c and they produce a finite orbit. The repelling fixed point is on the boundary of K_c.

K_c for Two Simple Cases

The set K_c is simple for two values of c:

1. c=0: Starting at any point s, the orbit is simply s, s2,s4,.....,s2k,..., so the orbit spirals into 0 if $|s|<1$ and explodes if $|s|>1$. Thus K_0 is the set of all complex numbers lying inside the unit circle, the circle of radius 1 centered at the origin.
2. c = -2: in this case it turns out that the filled-in Julia set consists of all points lying on the real axis between -2 and 2.

For all other values of c, the set K_c, is complex. It has been shown that each K_c is one of the two types:

- K_c is connected or
- K_c is a Cantor set

A theoretical result is that K_c is connected for precisely those values of c that lie in the Mandelbrot set.

The Julia Set J_c

Julia Set J_c is for any given value of c; it is the boundary of K_c. K_c is the set of all starting points that have finite orbits and every point outside K_c has an exploding orbit. We say that the points just along the boundary of K_c and "on the fence". Inside the boundary all orbits remain finite; just outside it, all orbits goes to infinity.

Preimages and Fixed Points

If the process started instead at f(s), the image of s, then the two orbits would be:

s, f(s), $f^2(s)$, $f^3(s)$,.... (orbit of s)

or

f(s), $f^2(s)$, $f^3(s)$, $f^4(s)$,.... (orbit of f(s))

which have the same value forever. If the orbit of s is finite, then so is the orbit of its image f(s). All of the points in the orbit , if considered as starting points on their own, have orbits with thew same behavior: They all are finite or they all explode.

Any starting point whose orbit passes through s has the same behavior as the orbit that start at s: The two orbits are identical forever. The point "just before" s in the sequence is called the preimage of s and is the inverse of the function f(.) = $(.)^2$ + c. The inverse of f(.) is $\pm\sqrt{z-c}$, so we have

two preimages of z are given by $z \quad c$---------- (6)

To check that equation (6) is correct, note that if either preimage is passed through $(.)^2$ + c, the result is z. The test is illustrated in figure(a) where the orbit of s is shown in black dots and the two preimages of s are marked. The two orbits of these preimages "join up" with that of s.

Each of these preimages has two preimages and each if these has two, so there is a huge collection of orbits that join up with the orbit of s, and thereafter committed to the same path. The tree of preimages of s is illustrated in fig(B): s has two parent preimages, 4 grandparents, etc. Going back k generations we find that there are 2^k preimages. The Julia set J_c can be characterized in many ways that are more precise than simply saying it is the "boundary of" K_c. One such characterization that suggests an algorithm for drawing J_c is the following:

The collection of all preimages of any point in J_c is dense in J_c.

Starting with any point z in J_c, we simply compute its two parent preimages, their four grandparent preimages, their eight great-grandparent ones, etc. So we draw a dot at each such preimage, and the display fills in with a picture of the Julia set. To say that these dots are dense in J_c means that for every point in J_c, there is some preimage that is close by.

Drawing the Julia set J_c

To draw J_c we need to find a point and place a dot at all of the point's preimages. Therea re two problems with this method:

1. Finding a point in J_c
2. Keeping track of all the preimages

An approach known as the backward-iteration method overcomes these obstacles and produces good result. The idea is simple: Choose some point z in the complex plane. The point may or may not be in J_c. Now iterate in backward direction: at each iteration choose one of the two square roots randomly, to produce a new z value. The following pseudo code is illustrative:

```
do {
    if ( coin flip is heads z= ±√(z − c) );
    else z = −√(z − c) ;
        draw dot at z;
  } while (not bored);
```

The idea is that for any reasonable starting point iterating backwards a few times will produce a z that is in J_c. It is as if the backward orbit is sucked into the Julia set. Once it is in the Julia set, all subsequent iterations are there, so point after point builds up inside J_c, and a picture emerges.

RANDOM FRACTALS

Fractal is the term associated with randomly generated curves and surfaces that exhibit a degree of self-similarity. These curves are used to provide "naturalistic" shapes for representing objects such as coastlines, rugged mountains, grass and fire.

Fractalizing a Segment

The simplest random fractal is formed by recursively roughening or fractalizing a line segment. At each step, each line segment is replaced with a "random elbow".

The figure shows this process applied to the line segment S having endpoints A and B. S is replaced by the two segments from A to C and from C to B. For a fractal curve, point C is randomly chosen along the perpendicular bisector L of S. The elbow lies randomly on one or the other side of the "parent" segment AB.Fractalizing with a random elbow

Steps in the fractalization process. Three stages are required in the fractalization of a segment. In the first stage, the midpoint of AB is perturbed to form point C. In the second stage , each of the two segment has its midpoints perturbed to points D and E. In the third and final stage, the new points F…..I are added.

To perform fractalization in a program

Line L passes through the midpoint M of segment S and is perpendicular to it. Any point C along L has the parametric form:

$C(t) = M + (B-A)^{\perp} t$ ---------- (7)

for some values of t, where the midpoint M= (A+B)/2.

The distance of C from M is |B-A||t|, which is proportional to both t and the length of S. So to produce a point C on the random elbow, we let t be computed randomly. If t is positive, the elbow lies to one side of AB; if t is negative it lies to the other side.

For most fractal curves, t is modeled as a Gaussian random variable with a zero mean and some standard deviation. Using a mean of zero causes, with equal probability, the elbow to lie above or below the parent segment.

Fractalizing a Line segment

```
void fract(Point2 A, Point2 B, double stdDev)
// generate a fractal curve from A to B
      double xDiff = A.x – B.x, yDiff= A.y –B.y;
      Point2 C;
      if(xDiff * XDiff + YDiff * yDiff < minLenSq)
            cvs.lintTo(B.x, B.y);
      else
      {
            stdDev *=factor;                        //scale stdDev by factor
            double t=0;
            // make a gaussian variate t lying between 0 and 12.0
            for(int i=0; I, 12; i++)
                t+= rand()/32768.0;
            t= (t-6) * stdDev;                  //shift the mean to 0 and sc
            C.x = 0.5 *(A.x +B.x) – t * (B.y – A.y);
            C.y = 0.5 *(A.y +B.y) – t * (B.x – A.x);
            fract(A, C, stdDev);
            fract(C, B, stdDev);
      }
```

The routine fract() generates curves that approximate actual fractals. The routine recursively replaces each segment in a random elbow with a smaller random elbow. The stopping criteria used is: When the length of the segment is small enough, the segment is drawn using cvs.lineTo(), where cvs is a Canvas object. The variable t is made to be approximately Gaussian in its distribution by summing together 12 uniformly distributed random values lying between 0 and 1. The result has a mean value of 6 and a variance of 1. The mean value is then shifted to 0 and the variance is scaled as necessary.

The depth of recursion in fract() is controlled by the length of the line segment. Controlling the Spectral Density of the Fractal Curve

The fractal curve generated using the above code has a “power spectral density” given by

$$S(f)= 1/f^{\beta}$$

Where β the power of the noise process is the parameter the user can set to control the jaggedness of the fractal noise. When β is 2, the process is known as Brownian motion and when β is 1, the process is called "1/f noise". 1/f noise is self similar and is shown to be a good model for physical process such as clouds. The fractal dimension of such processes is:

$$D = \frac{5-\beta}{2}$$

In the routine fract(), the scaling factor factor by which the standard deviation is scaled at each level based on the exponent β of the fractal curve. Values larger than 2 leads to smoother curves and values smaller than 2 leads to more jagged curves. The value of factor is given by:

$$\text{Factor} = 2^{(1-\beta/2)}$$

The factor decreases as β increases.

Drawing a fractal curve (pseudocode)

```
double MinLenSq, factor;                    //global variables
void drawFractal (Point2 A, Point2 B)
{
    double beta, StdDev;
    User inputs beta, MinLenSq and the the initial StdDev
    factor = pow(2.0, (1.0 – beta)/ 2.0);
    cvs.moveTo(A);
    fract(A, B, StdDev);
}
```

In this routine factor is computed using the C++ library function pow(…).

One of the features of fractal curves generated by pseudorandom –number generation is that they are repeatable. All that is required is to use the same seed each time the curve is fractalized. A complicated shape can be fractalized and can be stored in the database by storing only

- the polypoint that describes the original line segments
- the values of minLenSq and stdDev and the seed.
- An extract replica of the fractalized curve can be regenerated at any time using these informations.

Unit - VI

Overview of Ray Tracing Intersecting rays with other primitives ◆ Adding Surface texture ◆ Reflections andTransparency ◆ Boolean operations on Objects.

Intersecting rays with other primitives:

Mathematical preliminaries

Coordinate systems

To deal easily with the various primitive objects, you need to be able to work in all three of the standard 3D coordinate systems: rectangular (*x*,*y*,*z*), spherical polar (r, θ, ϕ), and cylindrical polar (r, θ, z). To convert from one to another you use the following formulae.
Spherical polar to rectangular

$$x = r \cos\phi \cos\theta \quad (1)$$

$$y = r \cos\phi \sin\theta \quad (2)$$

$$z = r \sin\phi \quad (3)$$

Cylindrical polar to rectangular

$$x = r \cos\theta \quad (4)$$

$$y = r \sin\theta \quad (5)$$

$$z = z \quad (6)$$

Rectangular to spherical polar[1]

$$r = \sqrt{x^2 + y^2 + z^2} \quad (7)$$

$$\theta = \tan^{-1} y/x \quad (8)$$

$$\phi = \tan^{-1}\left(z/\sqrt{x^2 + y^2}\right) \quad (9)$$

Rectangular to cylindrical polar

$$r = \sqrt{x^2 + y^2} \quad (10)$$

$$\theta = \tan^{-1} y/x \tag{11}$$

$$z = z \tag{12}$$

Vector algebra

It is helpful to remember your vector arithmetic. A 3D vector is represented thus:

$$\mathbf{P} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{13}$$

$$\mathbf{P} = (x, y, z)$$

For ease of writing such definitions in text we may say , where we understand that this ordered triple is equivalent to the vector.

Equations for the primitives

Sphere

The unit sphere, centred at the origin, has the implicit equation:

$$x^2+y^2+z^2=1 \tag{25}$$

In spherical polar coordinates it is even simpler:

$$r=1 \tag{26}$$

In vector arthmetic, it becomes:

$$\mathbf{P} \cdot \mathbf{P} = 1 \tag{27}$$

To find the intersection between this sphere and an arbitrary ray, substitute the ray equation (Equation 24) in the sphere equation (Equation 25):

$$(x_E+tx_D)^2+(y_E+ty_D)^2+(z_E+tz_D)^2=1 \tag{28}$$

$$\Rightarrow \quad t^2(x_D^2+y_D^2+z_D^2)+t(2x_Ex_D+2y_Ey_D+2z_Ez_D) + (x_E^2+y_E^2+z_E^2-1)=0 \tag{29}$$

$$\Rightarrow \quad at^2+bt+c=0 \tag{30}$$

$$\Rightarrow \quad t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} \tag{31}$$

where $a=x_D^2+y_D^2+z_D^2$, $b=2x_Ex_D+2y_Ey_D+2z_Ez_D$, and $c=x_E^2+y_E^2+z_E^2-1$. This gives zero, one, or two real values for t. If there are zero real values then there is no intersection between the ray and the sphere. If there are either one or two real values then chose the smallest, non-negative value, as the intersection point. If there is no non-negative value, then the line (of which the ray is a part) *does* intersect the sphere, but the intersection point is not on the part of the line which consistutes the ray. In this case there is again no intersection point between the ray and the sphere.

An alternative formulation is to use the vector versions of the equations (Equations 23 and 27):

$$(\mathbf{E}+t\mathbf{D})\cdot(\mathbf{E}+t\mathbf{D})=1 \tag{32}$$

$$\Rightarrow \quad t^2(\mathbf{D}\cdot\mathbf{D})+t(2\mathbf{E}\cdot\mathbf{D})+(\mathbf{E}\cdot\mathbf{E}-1)=0 \tag{33}$$

$$\Rightarrow \quad at^2+bt+c=0 \tag{34}$$

$$\Rightarrow \quad t=\frac{-b\pm\sqrt{b^2-4ac}}{2a} \tag{35}$$

Where $a=\mathbf{D}\cdot\mathbf{D}$, $b=2\mathbf{E}\cdot\mathbf{D}$, and $c=\mathbf{E}\cdot\mathbf{E}-1$. In other words, exactly the same result, expressed in a more compact way. *Graphics Gems I* (p. 388) decribes yet another way of arriving at the same result.

Cylinder

The *infinite* unit cylinder aligned along the z-axis is defined as:

$$x^2+y^2=1 \tag{36}$$

In cylindrical polar coordinates it is just:

$$r=1 \tag{37}$$

To intersect a ray with this, substitute Equation 24 in Equation 36.

$$(x_E+tx_D)^2+(y_E+ty_D)^2=1 \tag{38}$$

$$\Rightarrow \quad t^2(x_D^2+y_D^2)+t(2x_Ex_D+2y_Ey_D)+(x_E^2+y_E^2-1)=0 \tag{39}$$

$$\Rightarrow \quad at^2+bt+c=0 \tag{40}$$

$$\Rightarrow \quad t=\frac{-b\pm\sqrt{b^2-4ac}}{2a} \tag{41}$$

where $a=x_D^2+y_D^2$, $b=2x_Ex_D+2y_Ey_D$, and $c=x_E^2+y_E^2-1$.

The *finite* open-ended unit cylinder aligned along the *z*-axis is defined as:

$$x^2 + y^2 = 1, z_{\min} < z < z_{\max} \tag{42}$$

The only difference between this and Equation 36 being the restriction on *z*. In cylindrical polar coordinates this is obviously:

$$r = 1, z_{\min} < z < z_{\max} \tag{43}$$

To handle this finite length cylinder, solve Equation 41 above. This gives, at most, two values of *t*. Call these t_1 and t_2. Calculate z_1 and $z_2$using Equation 24 ($z_1 = z_E + t_1z_D$ and $z_2 = z_E + t_2z_D$) and then check $z_{\min} < z_1 < z_{\max}$ and $z_{\min} < z_2 < z_{\max}$. Whichever intersection point passes this test and, if both pass the test, has the smallest non-negative value of *t*, is the closest intersection point of the ray with the open-ended finite cylinder.

If we wish the finite length cylinder to be closed we must formulate an intersection calculation between the ray and the cylinder's end caps. The end caps have the formulae:

$$z = z_{\min}, \quad x^2 + y^2 \leq 1 \tag{44}$$

$$z = z_{\max}, \quad x^2 + y^2 \leq 1 \tag{45}$$

Once you have calculated the solutions to Equation 41 you will either know that there are no intersections with the infinite cylinder or you will know that there are one or two real intersection points (t_1 and t_2). The previous paragraph explained how to ascertain whether these correspond to points on the finite length open-ended cylinder. Now, if z_1 and z_2 lie either side of $z_{\min}$ we know that the ray intersects the $z_{\min}$ end cap, and can calculate the intersection point as:

$$t_3 = \frac{z_{\min} - z_E}{z_D} \tag{46}$$

A similar equation holds for the $z_{\max}$ end cap. Note that the ray may intersect both end caps, for example when $z_1 < z_{\min}$ and $z_2 > z_{\max}$.

Cone

The *infinite* double cone[2] aligned along the *z*-axis is defined as:

$$x^2+y^2=z^2 \tag{47}$$

In cylindrical polar coordinates it is:

$$r^2=z^2 \tag{48}$$

To intersect a ray with this, substitute Equation 24 in Equation 47.

$$(x_E+tx_D)^2+(y_E+ty_D)^2=(z_E+tz_D)^2 \tag{49}$$

$$\Rightarrow \quad t^2(x_D^2+y_D^2-z_D^2)+t(2x_Ex_D+2y_Ey_D-2z_Ez_D) +(x_E^2+y_E^2-z_E^2)=0 \tag{50}$$

$$\Rightarrow \quad at^2+bt+c=0 \tag{51}$$

$$\Rightarrow \quad t=\frac{-b\pm\sqrt{b^2-4ac}}{2a} \tag{52}$$

where $a=x_D^2+y_D^2-z_D^2$, $b=2x_Ex_D+2y_Ey_D-2z_Ez_D$, and $c=x_E^2+y_E^2-z_E^2$.

The *finite* open-ended cone aligned along the z-axis is defined as:

$$x^2+y^2=z^2, z_{\min}<z<z_{\max} \tag{53}$$

The only difference between this and Equation 47 being the restriction on z. Note that if $z_{\min}$ and $z_{\max}$ are both positive or both negative then you get a single cone with its top truncated. If either $z_{\min}$ or $z_{\max}$ is zero you get a single cone with its apex at the origin.

To handle this finite length cone you proceed as for the finite length cylinder, with the obvious simple modifications.

Texture mapping is a method for defining high frequency detail, surface texture, or color information on a computer-generated graphic or 3D model. Its application to 3D graphics was pioneered by Edwin Catmull in 1974.

Texture mapping originally referred to a method (now more accurately called diffuse mapping) that simply wrapped and mapped pixels from a texture to a 3D surface. In recent decades the advent of multi-pass rendering and complex mapping such as height mapping, bump mapping, normal mapping, displacement mapping, reflection mapping, specular mapping, mipmaps, occlusion mapping, and many other variations on the technique (controlled by a materials system) have made it possible to simulate near-photorealism in real time by vastly reducing the number of polygons and lighting calculations needed to construct a realistic and functional 3D scene.

A **texture map**[5][6] is an image applied (mapped) to the surface of a shape or polygon.[7] This may be a bitmap image or a procedural texture. They may be stored in common image file formats, referenced by 3d model formats or material definitions, and assembled into resource bundles.

They may have 1-3 dimensions, although 2 dimensions are most common for visible surfaces. For use with modern hardware, texture map data may be stored in swizzled or tiled orderings to improve cache coherency. Rendering APIs typically manage texture map resources (which may be located in device memory) as buffers or surfaces, and may allow 'render to texture' for additional effects such as post processing, environment mapping.

They usually contain RGB color data (either stored as direct color, compressed formats, or indexed color), and sometimes an additional channel for alpha blending (RGBA) especially for billboards and *decal* overlay textures. It is possible to use the alpha channel (which may be convenient to store in formats parsed by hardware) for other uses such as specularity. Multiple texture maps (or channels) may be combined for control over specularity, normals, displacement, or subsurface scattering e.g. for skin rendering. Multiple texture images may be combined in **texture atlases** or **array textures** to reduce state changes for modern hardware. (They may be considered a modern evolution of tile map graphics). Modern hardware often supports cube map textures with multiple faces for environment mapping.

Texture application

This process is akin to applying patterned paper to a plain white box. Every vertex in a polygon is assigned a texture coordinate (which in the 2d case is also known as a UV coordinates). This may be done through explicit assignment of vertex attributes, manually edited in a 3D modelling package through UV unwrapping tools. It is also possible to associate a procedural transformation from 3d space to texture space with the material. This might be accomplished via planar projection or, alternatively, cylindrical or spherical mapping. More complex mappings may consider the distance along a surface to minimize distortion. These coordinates are interpolated across the faces of polygons to sample the texture map during rendering. Textures may be **repeated** or **mirrored** to extend a finite rectangular bitmap over a larger area, or they may have a one-to-one unique "injective" mapping from every piece of a surface (which is important for render mapping and light mapping, also known as baking)

Texture space

Texture mapping maps from the model surface (or screen space during rasterization) into **texture space**; in this space, the texture map is visible in its undistorted form. UV unwrapping tools typically provide a view in texture space for manual editing of texture coordinates. Some rendering techniques such as subsurface scattering may be performed approximately by texture-space operations.

Multitexturing

Multitexturing is the use of more than one texture at a time on a polygon.[8] For instance, a light map texture may be used to light a surface as an alternative to recalculating that lighting every time the surface is rendered. **Microtextures** or **detail textures** are used to add higher frequency details, and **dirt maps** may add weathering and variation; this can greatly reduce the apparent periodicity of repeating textures. Modern graphics may use in excess of 10 layers for greater fidelity which are

combined using shaders. Another multitexture technique is bump mapping, which allows a texture to directly control the facing direction of a surface for the purposes of its lighting calculations; it can give a very good appearance of a complex surface (such as tree bark or rough concrete) that takes on lighting detail in addition to the usual detailed coloring. Bump mapping has become popular in recent video games, as graphics hardware has become powerful enough to accommodate it in real-time.[9]

Texture Filtering

The way that samples (e.g. when viewed as pixels on the screen) are calculated from the texels (texture pixels) is governed by texture filtering. The cheapest method is to use the nearest-neighbour interpolation, but bilinear interpolation or trilinear interpolation between mipmaps are two commonly used alternatives which reduce aliasing or jaggies. In the event of a texture coordinate being outside the texture, it is either clamped or wrapped. Anisotropic filtering better eliminates directional artefacts when viewing textures from oblique viewing angles.

Baking

As an optimization, it is possible to render detail from a high resolution model or expensive process (such as global illumination) into a surface texture (possibly on a low resolution model). This is also known as **render mapping**. This technique is most commonly used for lightmapping but may also be used to generate normal maps and displacement maps. Some video games (e.g. Messiah) have used this technique. The original Quake software engine used on-the-fly baking to combine light maps and colour texture-maps ("surface caching"). Baking can be used as a form of level of detail generation, where a complex scene with many different elements and materials may be approximated by a single element with a single texture which is then algorithmically reduced for lower rendering cost and fewer drawcalls. It is also used to take high detail models from 3D sculpting software and point cloud scanning and approximate them with meshes more suitable for realtime rendering.

Reflection in computer graphics is used to emulate reflective objects like mirrors and shiny surfaces. Reflection is accomplished in a ray trace renderer by following a ray from the eye to the mirror and then calculating where it bounces from, and continuing the process until no surface is found, or a non-reflective surface is found. Reflection on a shiny surface like wood or tile can add to the photorealistic effects of a 3D rendering.

- **Polished** - A polished reflection is an undisturbed reflection, like a mirror or chrome.
- **Blurry** - A blurry reflection means that tiny random bumps on the surface of the material cause the reflection to be blurry.
- **Metallic** - A reflection is metallic if the highlights and reflections retain the color of the reflective object.
- **Glossy** - This term can be misused. Sometimes, it is a setting which is the opposite of blurry (e.g. when "glossiness" has a low value, the reflection is blurry). However, some people use the term "glossy reflection" as a synonym for "blurred reflection". Glossy used in this context means that the reflection is actually blurred.

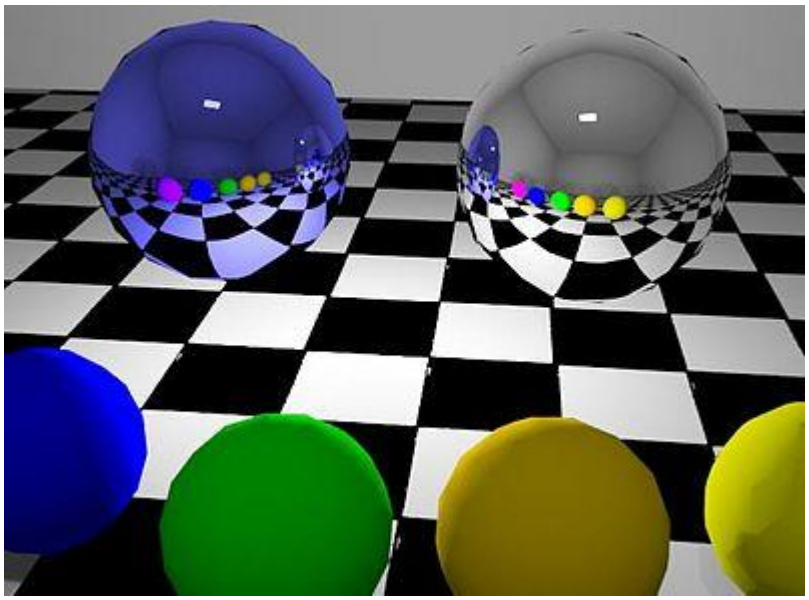
Examples

Polished or mirror reflection

Mirror on wall rendered with 100% reflection.

Mirrors are usually almost 100% reflective

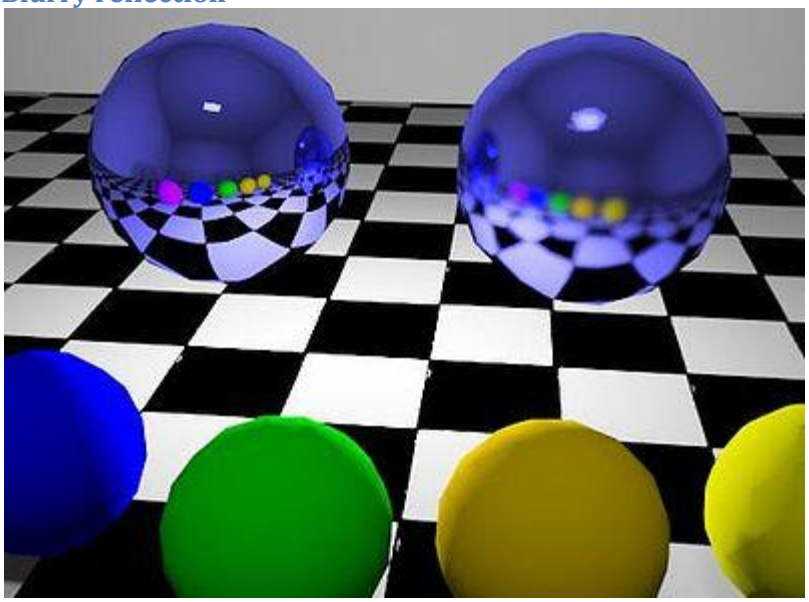
Metallic reflection



The large sphere on the left is blue with its reflection marked as metallic. The large sphere on the right is the same color but does not have the metallic property selected.

Normal (nonmetallic) objects reflect light and colors in the original color of the object being reflected. Metallic objects reflect lights and colors altered by the color of the metallic object itself.

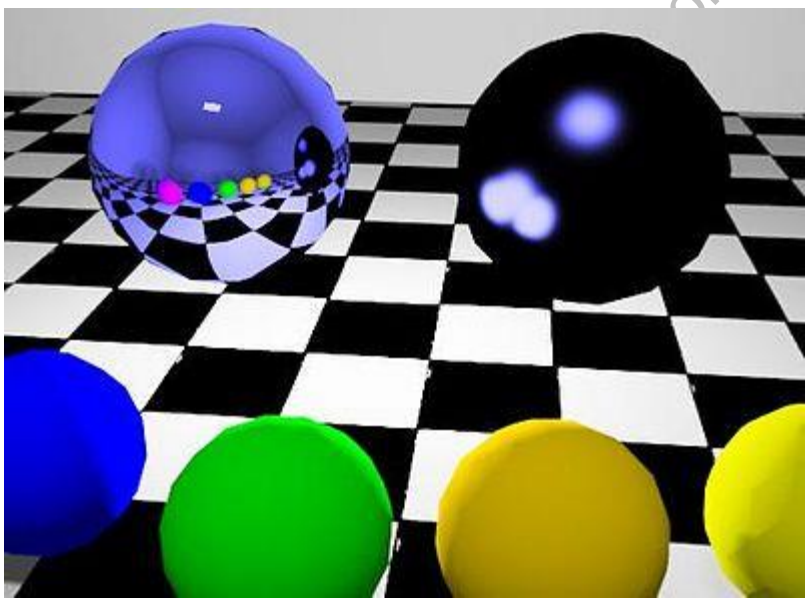
Blurry reflection



The large sphere on the left has sharpness set to 100%. The sphere on the right has sharpness set to 50% which creates a blurry reflection.

Many materials are imperfect reflectors, where the reflections are blurred to various degrees due to surface roughness that scatters the rays of the reflections.

Glossy reflection



The sphere on the left has normal, metallic reflection. The sphere on the right has the same parameters, except that the reflection is marked as "glossy".

Transparency

It is possible in a number of graphics file formats. The term **transparency** is used in various ways by different people, but at its simplest there is "full transparency" i.e. something that is completely invisible. Only part of a graphic should be fully transparent, or there would be

nothing to see. More complex is "partial transparency" or "translucency" where the effect is achieved that a graphic is partially transparent in the same way as colored glass. Since ultimately a printed page or computer or television screen can only be one color at a point, partial transparency is always simulated at some level by mixing colors. There are many different ways to mix colors, so in some cases transparency is ambiguous. In addition, transparency is often an "extra" for a graphics format, and some graphics programs will ignore the transparency.

Transparent pixels

This image has binary transparency (some pixels fully transparent, other pixels fully opaque). It can be transparent against any background because it is monochrome.

One color entry in a single GIF or PNG image's palette can be defined as "transparent" rather than an actual color. This means that when the decoder encounters a pixel with this value, it is rendered in the background color of the part of the screen where the image is placed, also if this varies pixel-by-pixel as in the case of a background image.

Applications include:

- an image that is not rectangular can be filled to the required rectangle using transparent surroundings; the image can even have holes (e.g. be ring-shaped)
- in a run of text, a special symbol for which an image is used because it is not available in the character set, can be given a transparent background, resulting in a matching background.

The transparent color should be chosen carefully, to avoid items that just happen to be the same color vanishing. Even this limited form of transparency has patchy implementation, though most popular web browsers are capable of displaying transparent GIF images. This support often does not extend to printing, especially to printing devices (such as PostScript) which do not include support for transparency in the device or driver. Outside the world of web browsers, support is fairly hit-or-miss for transparent GIF files. Transparency by clipping path

An alternative approach to full transparency is to use a Clipping path. A clipping path is simply a shape or outline, that is used in conjunction with the other graphics. Everything inside the path is visible, and everything outside the path is invisible. The path is inherently vector, but can potentially be used to mask both vector and bitmap data. The main usage of clipping paths is in PostScript files.

Compositing calculations

While some transparency specifications are vague, others may give mathematical details of how two colors are to be composited. This gives a fairly simple example of how compositing calculations can work, can produce the expected results, and can also produce surprises.

In this example, two grayscale colors are to be composited. Grayscale values are considered to be numbers between 0.0 (white) and 1.0 (black). *To emphasize: this is only one possible rule for transparency. If working with transparency, check the rules in use for your situation.*

The color at a point, where color G1 and G2 are to be combined, is `( G1 + G2 ) / 2`. Some consequences of this are:

- Where the colors are equal, the result is the same color because `( G1 + G1 ) /2 = G1`.
- Where one color (G1) is white (0.0), the result is `G2 / 2`. This will always be less than any nonzero value of G2, so the result is whiter than G2. (This is easily reversed for the case where G2 is white).
- Where one color (G1) is black (1.0), the result is `( G2 + 1 ) / 2`. This will always be more than G2, so the result is more black than G2.
- The formula is commutative since `( G1 + G2 ) / 2 = ( G2 + G1 ) / 2`. This means it does not matter which order two graphics are mixed i.e. which of the two is on the top and which is on the bottom.
- The formula is *not* associative since

```
(( G1 + G2 ) / 2 + G3 ) / 2 = G1 / 4 + G2 / 4 + G3 / 2
(G1 + (G2 + G3) / 2) / 2 = G1 / 2 + G2 / 4 + G3 / 4
```

This is important as it means that when combining three or more objects with this rule for transparency, the final color depends very much on the order of doing the calculations.

> Although the formula is simple, it may not be ideal. Human perception of brightness is not linear - we do not necessarily consider that a gray value of 0.5 is halfway between black and white. Such details may not matter when transparency is used only to soften edges, but in more complex designs this may be significant. Most people working seriously with transparency will need to see the results and may fiddle with the colors or (where possible) the algorithm to arrive at the results they need. Transparency in PostScript

The PostScript language has limited support for full (not partial) transparency, depending on the PostScript level. Partial transparency is available with the pdf mark extension,[1] available on many PostScript implementations.

Level 1

Level 1 PostScript offers transparency via two methods:

- A one-bit (monochrome) image can be treated as a mask. In this case the 1-bits can be painted any single color, while the 0-bits are not painted at all. This technique cannot be generalised to more than one color, or to vector shapes.
- Clipping paths can be defined. These restrict what part of all subsequent graphics can be seen. This can be used for any kind of graphic, however in level 1, the maximum number of nodes in a path was often limited to 1500, so complex paths (e.g. cutting around the hair in a photograph of a person's head) often failed.

Level 2

Level 2 PostScript adds no specific transparency features. However, by the use of patterns, arbitrary graphics can be painted through masks defined by any vector or text operations. This is, however, complex to implement. In addition, this too often reached implementation limits, and few if any application programs ever offered this technique.

Level 3

Level 3 PostScript adds further transparency option for any raster image. A transparent color, or range of colors, can be applied; or a separate 1-bit mask can be used to provide an alpha channel.

Boolean operations on polygons

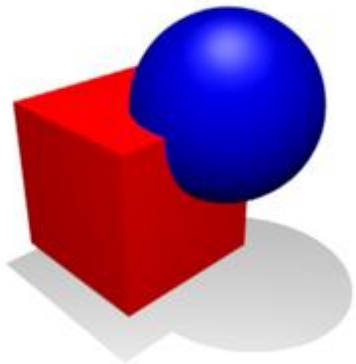
Boolean operations on polygons are a set of Boolean operations (AND, OR, NOT, XOR, ...) operating on one or more sets of polygons in computer graphics. These sets of operations are widely used in computer graphics, CAD, and in EDA (in integrated circuit physical design and verification software)

Constructive solid geometry (CSG) (formerly called computational binary solid geometry) is a technique used in solid modeling. Constructive solid geometry allows a modeler to create a complex surface or object by using Boolean operators to combine simpler objects.[1] Often CSG presents a model or surface that appears visually complex, but is actually little more than cleverly combined or recombined objects. In 3D computer graphics and CAD, CSG is often used in procedural modeling. CSG can also be performed on polygonal meshes, and may or may not be procedural and/or parametric. Contrast CSG with polygon mesh modeling and box modeling.

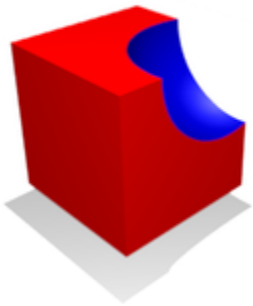
The simplest solid objects used for the representation are called primitives. Typically they are the objects of simple shape: cuboids, cylinders, prisms, pyramids, spheres, cones.[1] The set of allowable primitives is limited by each software package. Some software packages allow CSG on curved objects while other packages do not.

It is said that an object is constructed from primitives by means of allowable operations, which are typically Boolean operations on sets: union, intersection and difference, as well as geometric transformations of those sets.[1]

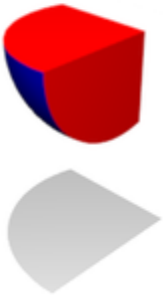
A primitive can typically be described by a procedure which accepts some number of parameters; for example, a sphere may be described by the coordinates of its center point, along with a radius value. These primitives can be combined into compound objects using operations like these:



Union
Merger of two objects into one



Difference
Subtraction of one object from another



Constructive solid geometry has a number of practical uses. It is used in cases where simple geometric objects are desire or where mathematical accuracy is important.[3] Nearly all engineering CAD packages use CSG (where it may be useful for representing tool cuts, and features where parts must fit together). The Quake engine and Unreal engine both use this system, as does Hammer (the native Source engine level editor), and Torque Game Engine/Torque Game Engine Advanced. CSG is popular because a modeler can use a set of relatively simple objects to create very complicated geometry.[2] When CSG is procedural or parametric, the user can revise their complex geometry by changing the position of objects or by changing the Boolean operation used to combine those objects. One of the advantages of CSG is that it can easily assure that objects are "solid" or water-tight if all of the primitive shapes are water-tight.[4] This can be important for some manufacturing or engineering computation applications. By comparison, when creating geometry based upon boundary representations, additional topological data is required, or consistency checks must be performed to assure that the given boundary description specifies a valid solid object. A convenient property of CSG shapes is that it is easy to classify arbitrary points as being either inside or outside the shape created by CSG. The point is simply classified against all the underlying primitives and the resulting Boolean expression is evaluated.[5] This is a desirable quality for some applications such as ray tracing.