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 to 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

Digital devices display a straight line segment by plotting discrete points between the two end points. Discrete coordinate positions along the line path are calculated from the equation of the line. For a raster video display, the line color (intensity) is then loaded into the frame buffer at the corresponding pixel coordinates. Reading from the frame buffer, the video controller then plots “the screen pixels”.

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

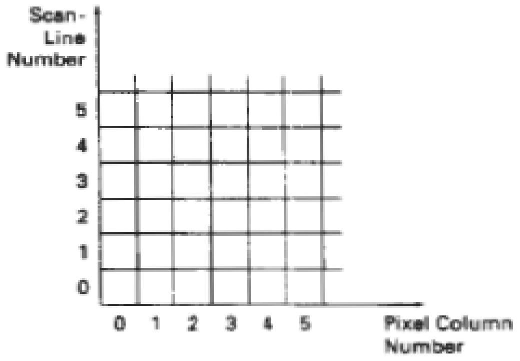


Figure : Pixel Postions reference by scan line number and column number

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
- Parallel Line Algorithm

The Cartesian ***slope-intercept equation*** for a straight line is

$$y = m \,.\, x + b \qquad (1)$$

Where m as slope of the line and b as the y intercept

Given that the two endpoints of a line segment are specified at positions (x_1,y_1) and (x_2,y_2) as in figure we can determine the values for the slope m and y intercept b with the following calculations

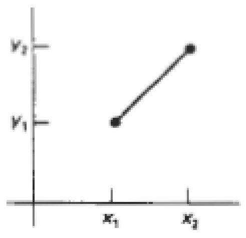


Figure : Line Path between endpoint positions (x_1,y_1) and (x_2,y_2)

$$m = \Delta y / \Delta x = y_2 - y1 / x_2 - x1 \qquad (2)$$

$$b = y1 - m \,.\, x1 \qquad (3)$$

For any given x interval Δx along a line, we can compute the corresponding y interval Δ y

$$\Delta y = m\, \Delta x \qquad (4)$$

We can obtain the x interval Δx corresponding to a specified Δy as

$$\Delta x = \Delta y/m \qquad (5)$$

For lines with slope magnitudes $|m| < 1$, Δx can be set proportional to a small horizontal deflection voltage and the corresponding vertical deflection is then set proportional to Δy as calculated from Eq (4).

For lines whose slopes have magnitudes $|m| > 1$, Δy can be set proportional to a small vertical deflection voltage with the corresponding horizontal deflection voltage set proportional to Δx, calculated from Eq (5)

For lines with $m = 1$, $\Delta x = \Delta y$ and the horizontal and vertical deflections voltage are equal.

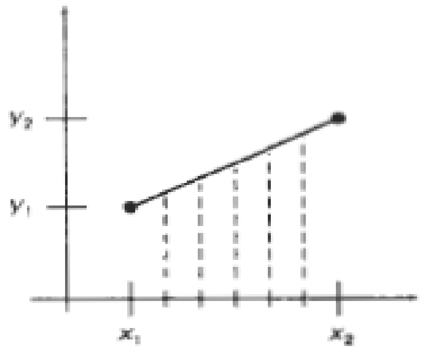


Figure : Straight line Segment with five sampling positions along the x axis between x1 and x2

Digital Differential Analyzer (DDA) Algortihm

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

The line 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.

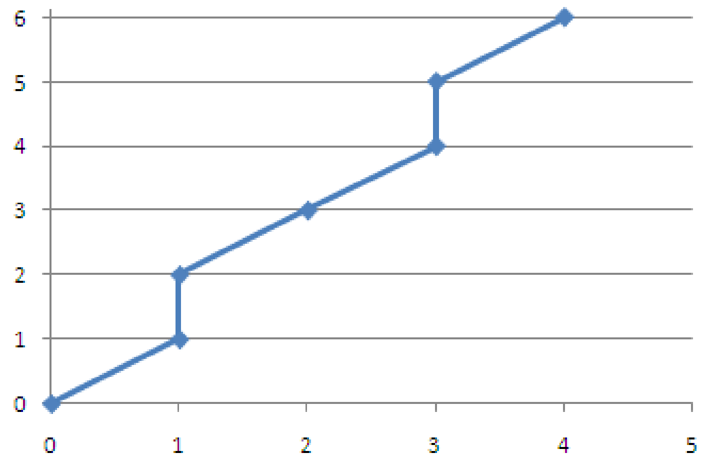
Example : Consider the line from (0,0) to (4,6)

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.

Pixel positions along a line path are then determined by sampling at unit x intervals. Starting from the left endpoint (x_0,y_0) of a given line, we step to each successive column (x position) and plot the pixel whose scan-line y value is closest to the line path.

To determine the pixel (x_k,y_k) is to be displayed, next to decide which pixel to plot the column $x_{k+1}=x_{k+1}.(x_{k+1},y_k)$ and $.(x_{k+1},y_{k+1})$. At sampling position x_{k+1}, we label vertical pixel separations from the mathematical line path as d_1 and d_2. The y coordinate on the mathematical line at pixel column position x_{k+1} is calculated as

$$y = m(x_{k+1})+b \qquad (1)$$

Then

$$\begin{aligned} d_1 &= y-y_k \\ &= m(x_{k+1})+b-y_k \\ d_2 &= (y_{k+1})-y \\ &= y_{k+1}-m(x_{k+1})-b \end{aligned}$$

To determine which of the two pixel is closest to the line path, efficient test that is based on the difference between the two pixel separations

$$d_1- d_2 = 2m(x_{k+1})-2y_k+2b-1 \qquad (2)$$

A decision parameter P_k for the k^{th} step in the line algorithm can be obtained by rearranging equation (2). By substituting $m=\Delta y/\Delta x$ where Δx and Δy are the vertical and horizontal separations of the endpoint positions and defining the decision parameter as

$$\begin{aligned} p_k &= \Delta x\ (d_1- d_2) \\ &= 2\Delta y\ x_k.-2\Delta x.\ y_k + c \qquad (3) \end{aligned}$$

The sign of p_k is the same as the sign of $d_1- d_2$,since $\Delta x>0$

Parameter C is constant and has the value $2\Delta y + \Delta x(2b-1)$ which is independent of the pixel position and will be eliminated in the recursive calculations for $P_{k.}$

If the pixel at y_k is “closer” to the line path than the pixel at y_{k+1} $(d_1< d_2)$ than decision parameter P_k is negative. In this case, plot the lower pixel, otherwise plot the upper pixel. Coordinate changes along the line occur in unit steps in either the x or y directions.

To obtain the values of successive decision parameters using incremental integer calculations. At steps k+1, the decision parameter is evaluated from equation (3) as

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

Subtracting the equation (3) from the preceding equation

$$P_{k+1} - P_k = 2\Delta y\,(x_{k+1} - x_k) - 2\Delta x(y_{k+1} - y_k)$$

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

$$P_{k+1} = P_k + 2\Delta y - 2\Delta x(y_{k+1} - y_k) \qquad (4)$$

Where the term $y_{k+1} - y_k$ is either 0 or 1 depending on the sign of parameter P_k

This recursive calculation of decision parameter is performed at each integer x position, starting at the left coordinate endpoint of the line.

The first parameter P_0 is evaluated from equation at the starting pixel position (x_0, y_0) and with m evaluated as $\Delta y/\Delta x$

$$P_0 = 2\Delta y - \Delta x \qquad (5)$$

Bresenham's line drawing for a line with a positive slope less than 1 in the following outline of the algorithm.

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 $\Delta x$, $\Delta y$, $2\Delta y$ 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 $\Delta x$ 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

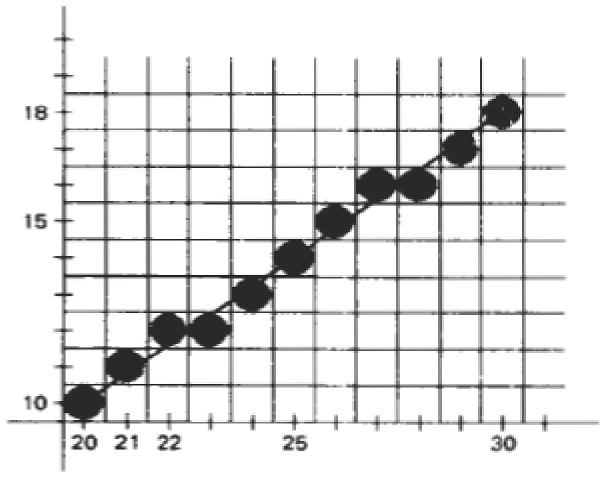
$2\Delta y=16$ $2\Delta y-2\Delta x= -4$

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.

Line Function

The two dimension line function is **Polyline(n,wcPoints)** where **n** is assigned an integer value equal to the number of coordinate positions to be input and **wcPoints** is the array of input world-coordinate values for line segment endpoints.

polyline function is used to define a set of n – 1 connected straight line segments

To display a single straight-line segment we have to set n=2 and list the x and y values of the two endpoint coordinates in wcPoints.

Example : following statements generate 2 connected line segments with endpoints at (50, 100), (150, 250), and (250, 100)

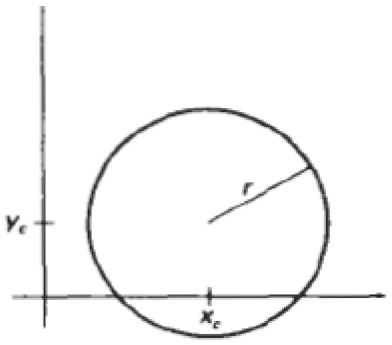
```
typedef struct myPt{int x, y;};
myPt wcPoints[3];
wcPoints[0] .x = 50; wcPoints[0] .y = 100;
wcPoints[1] .x = 150; wcPoints[1].y = 50;
wcPoints[2].x = 250; wcPoints[2] .y = 100;
polyline ( 3 , wcpoints);
```

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

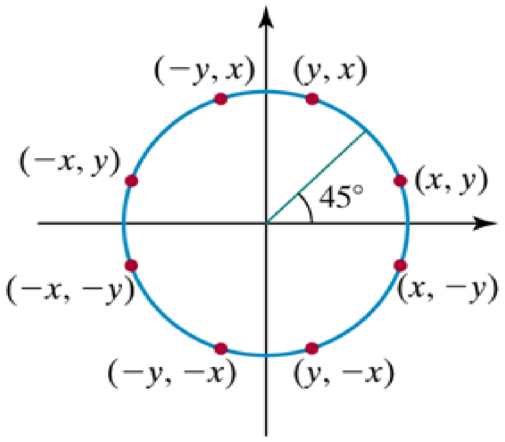
$$x = x_c + r\cos\theta \qquad y = y_c + r\sin\theta$$

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.

Set the angular step size at 1/r. This plots pixel positions that are approximately one unit apart. The shape of the circle is similar in each quadrant. To determine the curve positions in the first quadrant, to generate he circle section in the second quadrant of the xy plane by nothing that the two circle sections are symmetric with respect to the y axis and circle section in the third and fourth quadrants can be obtained from sections in the first and second quadrants by considering symmetry between octants.

Circle sections in adjacent octants within one quadrant are symmetric with respect to the 45^0 line dividing the two octants. Where a point at position (x, y) on a one-eight circle sector is mapped into the seven circle points in the other octants of the xy plane.

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.



Bresenham's line algorithm for raster displays is adapted to circle generation by setting up decision parameters for finding the closest pixel to the circumference at each sampling step. Square root evaluations would be required to computer pixel siatances from a circular path.

Bresenham's circle algorithm avoids these square root calculations by comparing the squares of the pixel separation distances. It is possible to perform a direct distance comparison without a squaring operation.

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:

In the raster line algorithm at unit intervals and determine the closest pixel position to the specified circle path at each step for a given radius r and screen center position (x_c,y_c) set up our algorithm to calculate pixel positions around a circle path centered at the coordinate position by adding x_c to x and y_c to y.

To apply the midpoint method we define a circle function as

$$f_{circle}(x,y) = x^2+y^2-r^2$$

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

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

The tests in the above eqn are performed for the midposition sbteween pixels near the circle path at each sampling step. The circle function is the decision parameter in the midpoint algorithm.

Midpoint between candidate pixels at sampling position x_{k+1} along a circular path. Fig -1 shows the midpoint between the two candidate pixels at sampling position x_{k+1}. To plot the pixel at (x_k,y_k) next need to determine whether the pixel at position (x_{k+1},y_k) or the one at position (x_{k+1},y_{k-1}) is circular to the circle.

Our decision parameter is the circle function evaluated at the midpoint between these two pixels

$$P_k= f_{circle}(x_{k+1},y_k-1/2)$$
$$=(x_{k+1})^2+(y_k-1/2)^2-r^2$$

If $P_k <0$, this midpoint is inside the circle and the pixel on scan line y_k is closer to the circle boundary. Otherwise the mid position is outside or on the circle boundary and select the pixel on scan line $y_k -1$.

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

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

or

$$P_{k+1}=P_k+2(x_{k+1})+(y^2_{k+1}-y^2_{k})-(y_{k+1}-y_k)+1$$

Where y_{k+1} is either y_k or y_{k-1} depending on the sign of P_k .

Increments for obtaining P_{k+1} are either $2x_{k+1}+1$ (if P_k is negative) or $2x_{k+1}+1-2\,y_{k+1}$.

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

$$2x_{k+1}=2x_{k+2}$$
$$2\,y_{k+1}=2\,y_{k-2}$$

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

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

$$P_0= f_{circle}\,(1,r-1/2)$$
$$=1+(r-1/2)^2-r^2$$

or

$$P_0=(5/4)-r$$

If the radius r is specified as an integer

$$P_0=1-r\text{(for r an integer)}$$

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

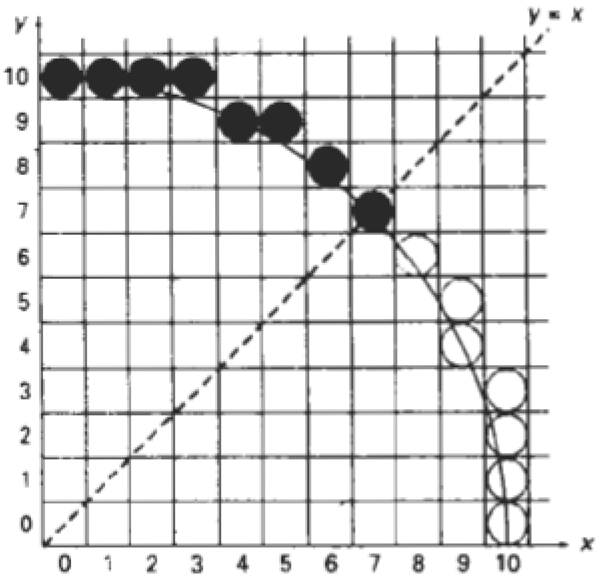
The circle octant in the first quadrant from x=0 to x=y. The initial value of the decision parameter is $P_0=1-r = -9$

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

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

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

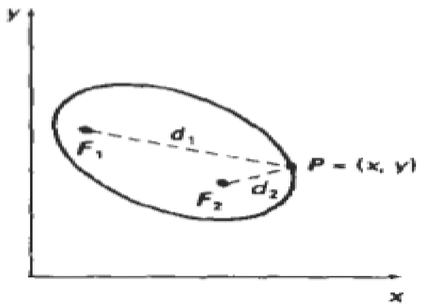
An ellipse is an elongated circle. Therefore, elliptical curves can be generated by modifying circle-drawing procedures to take into account the different dimensions of an ellipse along the major and minor axes.

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

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

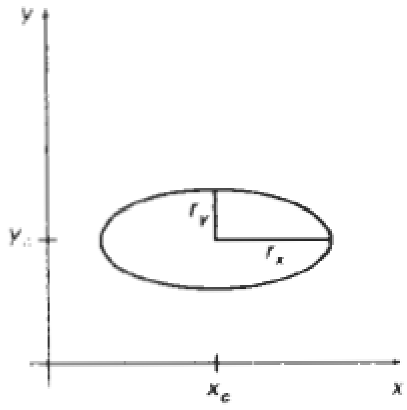
The coefficients A,B,C,D,E, and F are evaluated in terms of the focal coordinates and the dimensions of the major and minor axes of the ellipse.

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

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

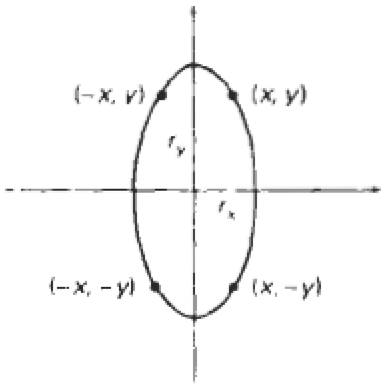


Using polar coordinates r and θ, to describe the ellipse in Standard position with the parametric equations

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

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

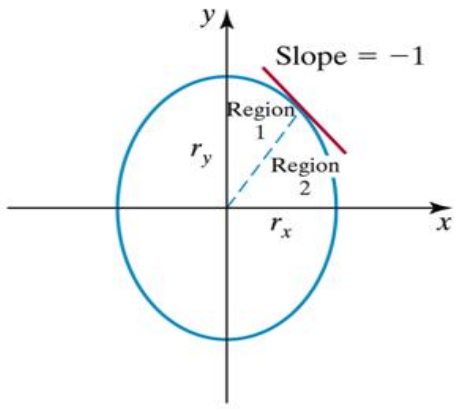
We must calculate pixel positions along the elliptical arc throughout one quadrant, and then we obtain positions in the remaining three quadrants by symmetry



Midpoint ellipse Algorithm

The midpoint ellipse method is applied throughout the first quadrant in two parts.

The below figure show the division of the first quadrant according to the slope of an ellipse with rx<ry.



In the x direction where the slope of the curve has a magnitude less than 1 and unit steps in the y direction where the slope has a magnitude greater than 1.

Region 1 and 2 can be processed in various ways

1. Start at position $(0,r_y)$ and step clockwise along the elliptical path in the first quadrant shifting from unit steps in x to unit steps in y when the slope becomes less than -1
2. Start at $(r_x,0)$ and select points in a counter clockwise order.
   2.1 Shifting from unit steps in y to unit steps in x when the slope becomes greater than -1.0
   2.2 Using parallel processors calculate pixel positions in the two regions simultaneously
3. Start at (0,ry)
   step along the ellipse path in clockwise order throughout the first quadrant ellipse function $(x_c,y_c)=(0,0)$

$$f_{ellipse}(x,y)=ry^2x^2+rx^2y^2-rx^2\,ry^2$$

which has the following properties:

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

Thus, the ellipse function $f_{ellipse}(x,y)$ serves as the decision parameter in the midpoint algorithm.

Starting at (0,ry):

Unit steps in the x direction until to reach the boundary between region 1 and region 2. Then switch to unit steps in the y direction over the remainder of the curve in the first quadrant.

At each step to test the value of the slope of the curve. The ellipse slope is calculated

$$dy/dx= -(2ry^2x/2rx^2y)$$

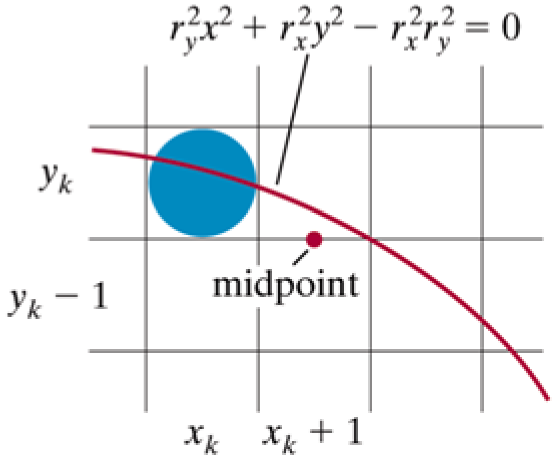
At the boundary between region 1 and region 2

$$dy/dx = -1.0 \text{ and } 2ry^2x=2rx^2y$$

to more out of region 1 whenever

$$2ry^2x>=2rx^2y$$

The following figure shows the midpoint between two candidate pixels at sampling position x_{k+1} in the first region.



To determine the next position along the ellipse path by evaluating the decision parameter at this mid point

$$P1_k = f_{ellipse} (x_{k+1},y_k-1/2)$$
$$= ry^2 (x_{k+1})^2 + rx^2 (y_k-1/2)^2 – rx^2 ry^2$$

if $P1_k$ <0, the midpoint is inside the ellipse and the pixel on scan line y_k is closer to the ellipse boundary. Otherwise the midpoint is outside or on the ellipse boundary and select the pixel on scan line y_{k-1}

At the next sampling position ($x_{k+1}+1=x_k+2$) the decision parameter for region 1 is calculated as

$p1_{k+1} = f_{ellipse}(x_{k+1} +1, y_{k+1} -½)$

$=r_y^2[(x_k +1) + 1]^2 + r_x^2 (y_{k+1} -½)^2 - r_x^2 r_y^2$

Or

$p1_{k+1} = p1_k +2 r_y^2(x_k +1) + r_y^2 + r_x^2 [(y_{k+1} -½)^2 - (y_k -½)^2]$

Where y_{k+1} is y_k or y_{k-1} depending on the sign of $P1_k$.

Decision parameters are incremented by the following amounts

increment = $\{ 2 r_y^2(x_k +1) + r_y^2$ if $p1_k <0 \}$

$\{ 2 r_y^2(x_k +1) + r_y^2 - 2r_x^2 y_{k+1}$ if $p1_k \geq 0 \}$

Increments for the decision parameters can be calculated using only addition and subtraction as in the circle algorithm.

The terms $2ry^2$ x and $2rx^2$ y can be obtained incrementally. At the initial position $(0,r_y)$ these two terms evaluate to

$$2 r_y^2 x = 0$$

$$2r_x^2 y =2r_x^2 r_y$$

x and y are incremented updated values are obtained by adding $2ry^2$to the current value of the increment term and subtracting $2rx^2$ from the current value of the increment term. The updated increment values are compared at each step and more from region 1 to region 2. when the condition 4 is satisfied.

In region 1 the initial value of the decision parameter is obtained by evaluating the ellipse function at the start position

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

region 2 at unit intervals in the negative y direction and the midpoint is now taken between horizontal pixels at each step for this region the decision parameter is evaluated as

$p1_0 = f_{ellipse}(1,r_y -½)$

$= r_y^2 + r_x^2 (r_y -½)^2 - r_x^2 r_y^2$

Or

$p1_0 = r_y^2 - r_x^2 r_y + ¼ r_x^2$

over region 2, we sample at unit steps in the negative y direction and the midpoint is now taken between horizontal pixels at each step. For this region, the decision parameter is evaluated as

$p2_k = f_{ellipse}(x_k + ½ , y_k - 1)$

$= r_y^2 (x_k + ½)^2 + r_x^2 (y_k - 1)^2 - r_x^2 r_y^2$

1. If $P2_k > 0$, the mid point position is outside the ellipse boundary, and select the pixel at x_k.
2. If $P2_k <= 0$, the mid point is inside the ellipse boundary and select pixel position x_{k+1}.

To determine the relationship between successive decision parameters in region 2 evaluate the ellipse function at the sampling step : $y_{k+1} -1= y_{k-2}$.

$P2_{k+1} = f_{ellipse}(x_{k+1} + ½, y_{k+1} -1)$

$= r_y^2 (x_k + ½)^2 + r_x^2 [(y_{k+1} -1) -1]^2 - r_x^2 r_y^2$

or

$p2_{k+1} = p2_k -2 r_x^2 (y_k -1) + r_x^2 + r_y^2 [(x_{k+1} + ½)^2 - (x_k + ½)^2]$

With x_{k+1}set either to $x_k$or x_{k+1}, depending on the sign of P2k. when we enter region 2, the initial position (x_0,y_0) is taken as the last position. Selected in region 1 and the initial decision parameter in region 2 is then

$p2_0 = f_{ellipse}(x_0 + ½ , y_0 - 1)$

$= r_y^2 (x_0 + ½)^2 + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2$

To simplify the calculation of P20, select pixel positions in counter clock wise order starting at $(r_x,0)$. Unit steps would then be taken in the positive y direction up to the last position selected in region 1.

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

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

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

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

with

$2 r_y^2x_k +1 = 2 r_y^2x_k + 2r_y^2$
$2 r_x^2y_k +1 = 2 r_x^2y_k + 2r_x^2$

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

4. 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.

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

5. 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.

6. Determine symmetry points in the other three quadrants.
7. Move each calculate pixel position (x,y) onto the elliptical path centered on (x_c,y_c) and plot the coordinate values

   $x=x+x_c$, $y=y+y_c$
8. 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

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

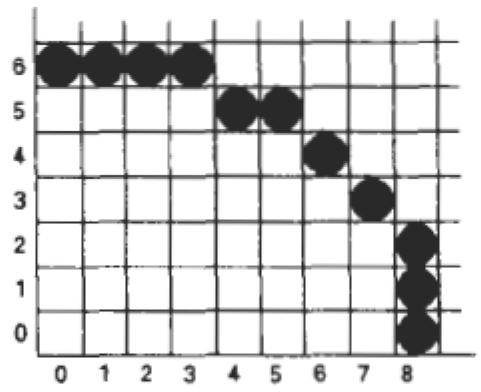
The remaining positions along the ellipse path in the first quadrant are then calculated as

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- l )* (y- l ) - 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.

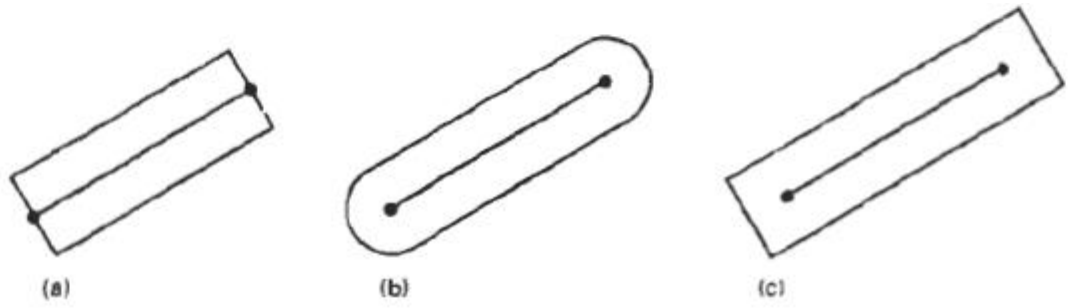
There are three types of line cap. They are

- 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.

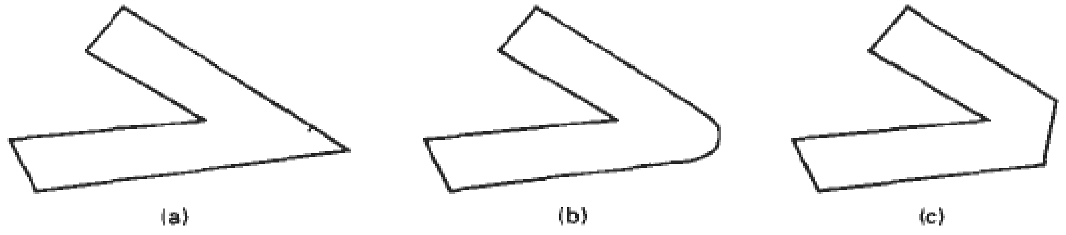


Thick lines drawn with (a) butt caps, (b) round caps, and (c) projecting square caps.

Three possible methods for smoothly joining two line segments

- Mitter Join
- Round Join
- Bevel Join

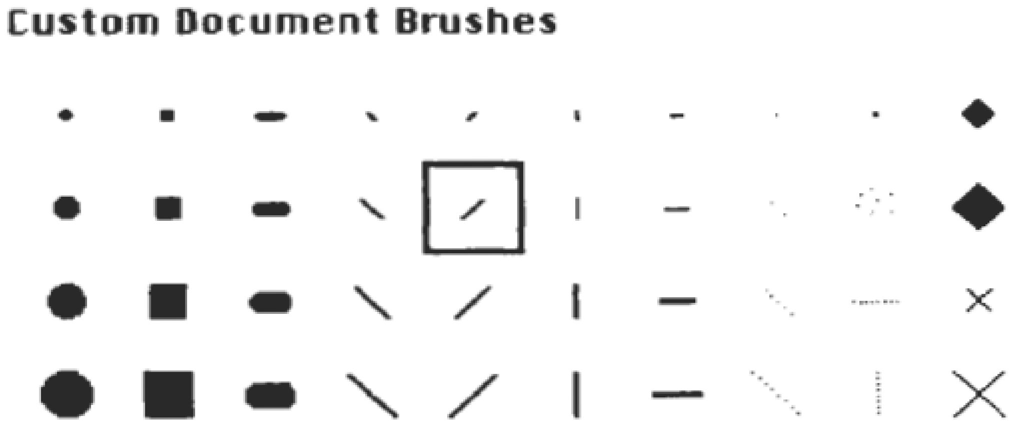
1. A **miter join** accomplished by extending the outer boundaries of each of the two lines until they meet.
2. A **round join** is produced by capping the connection between the two segments with a circular boundary whose diameter is equal to the width.
3. A **bevel join** is generated by displaying the line segment with but caps and filling in tri angular gap where the segments meet



Thick line segments connected with (a) miter join, (b) round join, and (c) bevel join.

Pen and Brush Options

With some packages, lines can be displayed with pen or brush selections. Options in this category include shape, size, and pattern. Some possible pen or brush shapes are given in Figure



Line color

A poly line routine displays a line in the current color by setting this color value in the frame buffer at pixel locations along the line path using the set pixel procedure.
We set the line color value in **PHlGS** with the function

setPolylineColourIndex (lc)

Nonnegative integer values, corresponding to allowed color choices, are assigned to the line color parameter lc

Example : Various line attribute commands in an applications program is given by the following sequence of statements

```
setLinetype(2);
setLinewidthScaleFactor(2);
setPolylineColourIndex (5);
polyline(n1,wc  points1);
setPolylineColorIindex(6);
poly line (n2, wc points2);
```

This program segment would display two figures, drawn with double-wide dashed lines. The first is displayed in a color corresponding to code 5, and the second in color 6.

Curve attributes

Parameters for curve attribute are same as those for line segments. Curves displayed with varying colors, widths, dot –dash patterns and available pen or brush options

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

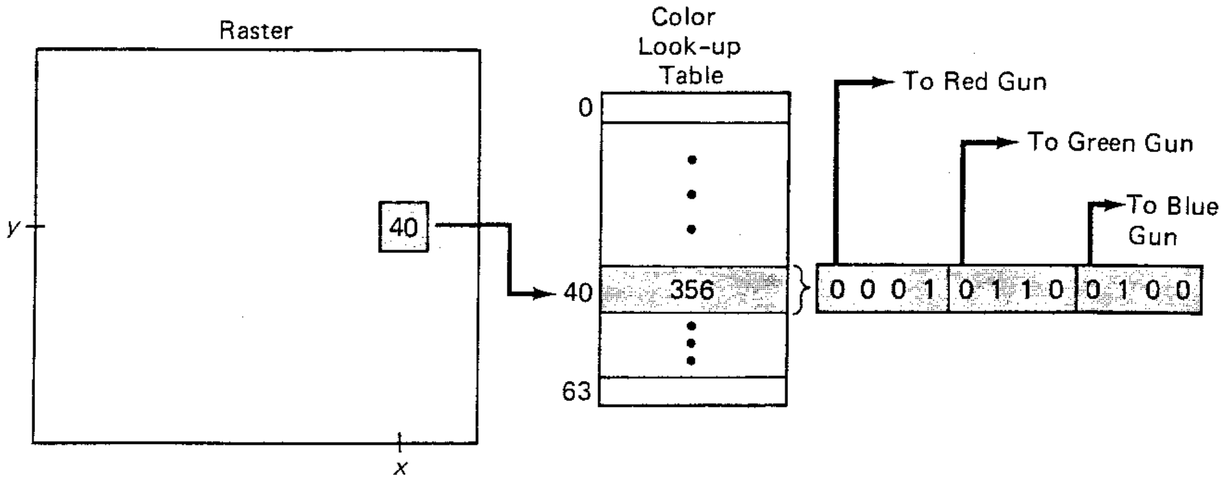
With the direct storage scheme, whenever a particular color code is specified in an application program, the corresponding binary value is placed in the frame buffer for each-component pixel in the output primitives to be displayed in that color.

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

Color tables(Color Lookup Tables) are an alternate means for providing extended color capabilities to a user without requiring large frame buffers



3 bits - 8 choice of color
6 bits – 64 choice of color
8 bits – 256 choice of color

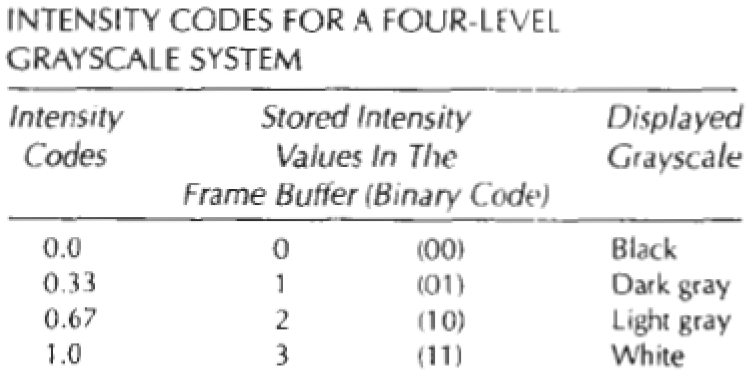
A user can set color-table entries in a PHIGS applications program with the function

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.



INTENSITY CODES FOR A FOUR-LEVEL GRAYSCALE SYSTEM

Intensity Codes	Stored Intensity Values In The Frame Buffer (Binary Code)		Displayed Grayscale
0.0	0	(00)	Black
0.33	1	(01)	Dark gray
0.67	2	(10)	Light gray
1.0	3	(11)	White

Intensity = 0.5[min(r,g,b)+max(r,g,b)]

Area fill Attributes

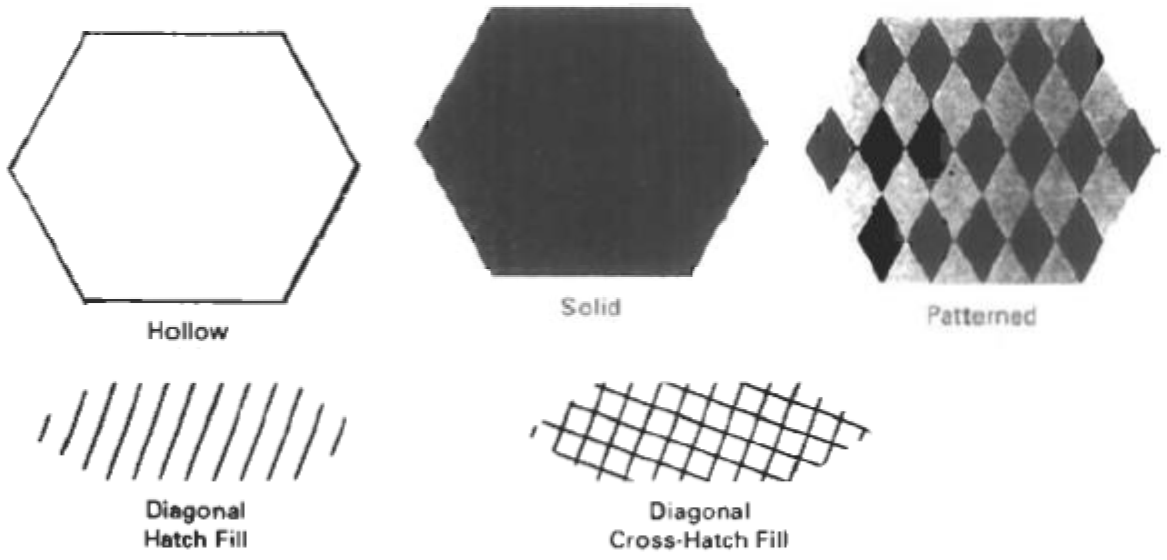
Options for filling a defined region include a choice between a solid color or a pattern fill and choices for particular colors and patterns

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 f**s** 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)

Pattern Fill

We select fill patterns with setInteriorStyleIndex (pi) where pattern index parameter pi specifies a table position

For example, the following set of statements would fill the area defined in the fillArea command with the second pattern type stored in the pattern table:

```
SetInteriorStyle( pattern)
SetInteriorStyleIndex(2);
Fill area (n, points)
```

Index (pi)	*Pattern (cp)*
1	$\begin{bmatrix} 4 & 0 \\ 0 & 4 \end{bmatrix}$
2	$\begin{bmatrix} 2 & 1 & 2 \\ 1 & 2 & 1 \\ 2 & 1 & 2 \end{bmatrix}$

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)

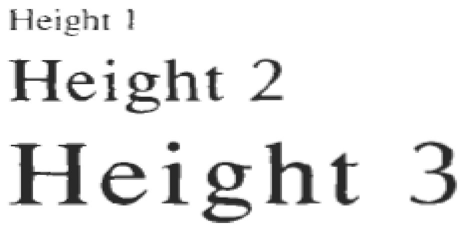
Control of text color (or intensity) is managed from an application program with

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)



Parameter ch is assigned a real value greater than 0 to set the coordinate height of capital letters

The width only of text can be set with function.

SetCharacterExpansionFactor(cw)

Where the character width parameter cw is set to a positive real value that scales the body width of character

width 0.5

width 1.0

width 2.0

Spacing between characters is controlled separately with

setCharacterSpacing(cs)

where the character-spacing parameter **cs** can he assigned any real value

Spacing 0.0

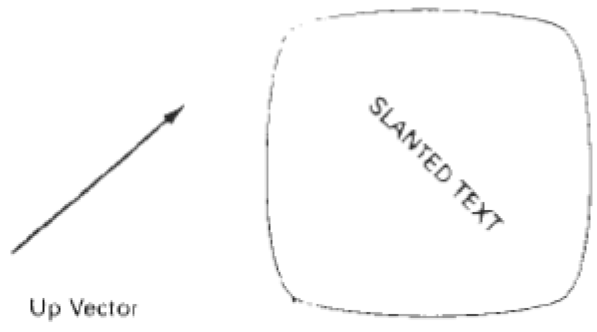
S p a c i n g 0 . 5

S p a c i n g   1 . 0

The orientation for a displayed character string is set according to the direction of the character up vector

setCharacterUpVector(upvect)

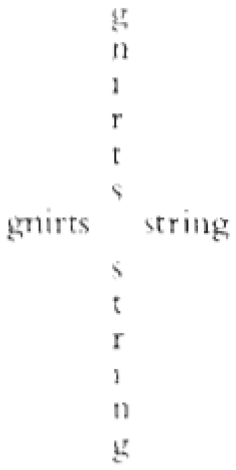
Parameter **upvect** in this function is assigned two values that specify the ***x*** and y vector components. For example, with **upvect** = (1, 1), the direction of the up vector is **45^{o}** and text would be displayed as shown in Figure.



To arrange character strings vertically or horizontally

setTextPath (tp)

Where the text path parameter tp can be assigned the value: right, left, up, or down



Another handy attribute for character strings is alignment. This attribute specifies how text is to be positioned with respect to the $tart coordinates. Alignment attributes are set with

setTextAlignment (h,v)

where parameters h and v control horizontal and vertical alignment. Horizontal alignment is set by assigning h a value of left, center, or right. Vertical alignment is set by assigning v a value of top, cap, half, base or bottom.

A precision specification for text display is given with

setTextPrecision (tpr)

tpr is assigned one of values string, char or stroke.

Marker Attributes

A marker symbol is a single character that can he displayed in different colors and in different sizes. Marker attributes are implemented by procedures that load the chosen character into the raster at the defined positions with the specified color and size. We select a particular character to be the marker symbol with

setMarkerType(mt)

where marker type parameter mt is set to an integer code. Typical codes for marker type are the integers 1 through 5, specifying, respectively, a dot (.) a vertical cross (+), an asterisk (*), a circle (o), and a diagonal cross (X).

We set the marker size with

setMarkerSizeScaleFactor(ms)

with parameter marker size ms assigned a positive number. This scaling parameter is applied to the nominal size for the particular marker symbol chosen. Values greater than 1 produce character enlargement; values less than 1 reduce the marker size.

Marker color is specified with

setPolymarkerColourIndex(mc)

A selected color code parameter mc is stored in the current attribute list and used to display subsequently specified marker primitives

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

Bundle area fill Attributes

Table entries for bundled area-fill attributes are set with

setInteriorRepresentation (ws, fi, fs, pi, fc)

Which defines the attributes list corresponding to fill index fi on workstation ws. Parameter fs, pi and fc are assigned values for the fill style pattern index and fill color.

Bundled Text Attributes

setTextRepresentation (ws, ti, tf, tp, te, ts, tc)

bundles values for text font, precision expansion factor size an color in a table position for work station ws that is specified by value assigned to text index parameter ti.

Bundled marker Attributes

setPolymarkerRepresentation (ws, mi, mt, ms, mc)

That defines marker type marker scale factor marker color for index mi on workstation ws.

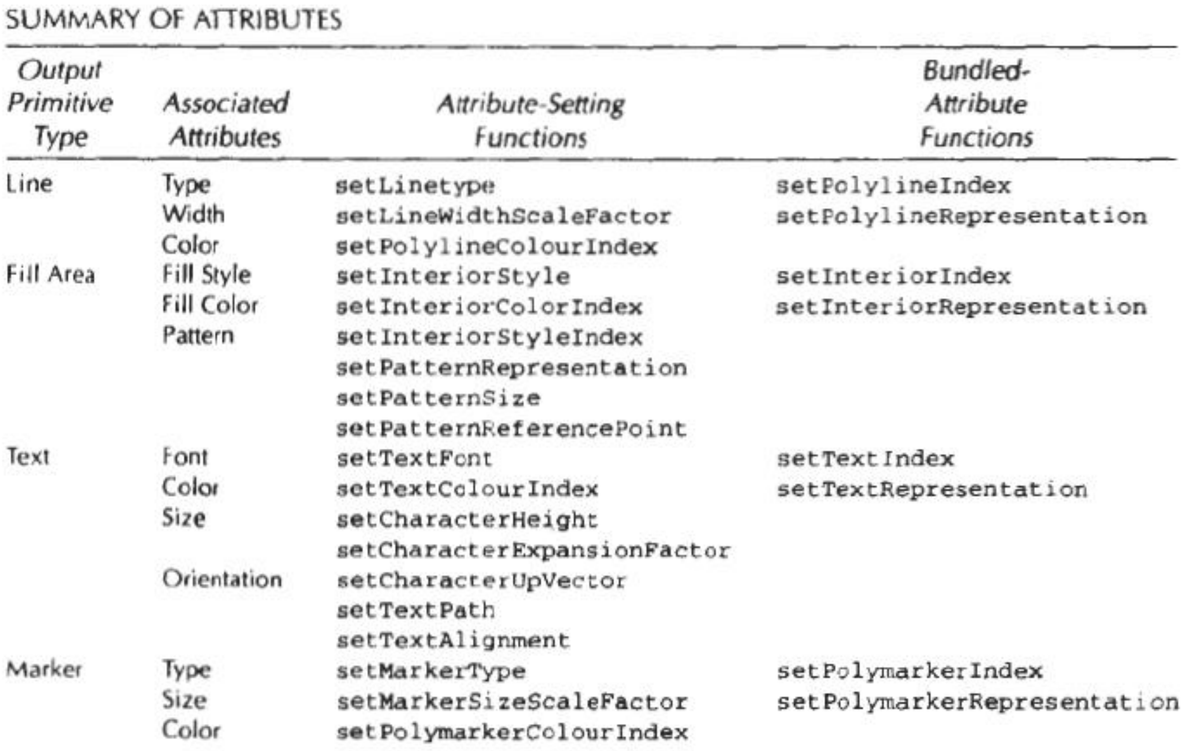
Inquiry functions

Current settings for attributes and other parameters as workstations types and status in the system lists can be retrieved with inquiry functions.

inquirePolylineIndex (lastli) and

inquireInteriorcColourIndex (lastfc)

Copy the current values for line index and fill color into parameter lastli and lastfc.



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

$$\mathbf{x' = x + tx, \quad y' = y + ty}$$

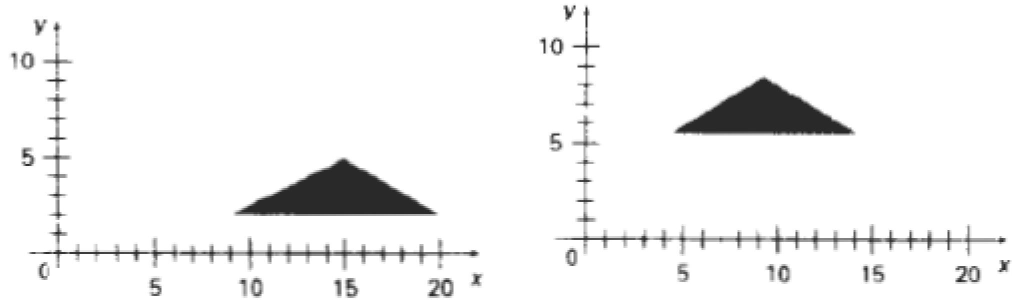
The translation distance point (tx,ty) is called translation vector or shift vector.

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

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

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

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

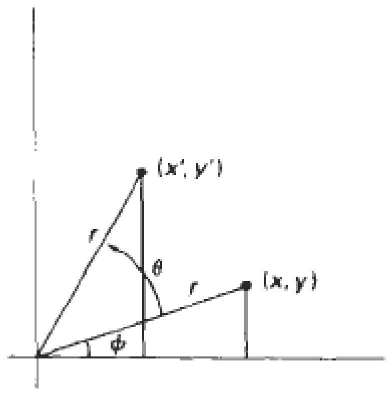


Moving a polygon from one position to another position with the translation vector (-5.5, 3.75)

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 rotate 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 coordinate origin

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

$$R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

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

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.

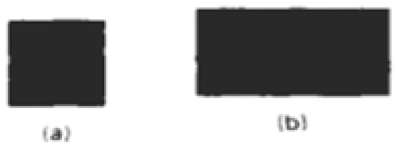
The transformation equation in matrix form

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

or

P' = S. P

Where S is 2 by 2 scaling matrix



Turning a square (a) Into a rectangle (b) with scaling factors sx = 2 and sy= 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 sx and sy. Unequal values for sx and sy result in a non uniform scaling.

Matrix Representation and homogeneous Coordinates

Many graphics applications involve sequences of geometric transformations. An animation, for example, might require an object to be translated and rotated at each increment of the motion. In order to combine sequence of transformations we have to eliminate the matrix addition. To achieve this we have represent matrix as 3 X 3 instead of 2 X 2 introducing an additional dummy coordinate h. Here points are specified by three numbers instead of two. This coordinate system is called as Homogeneous coordinate system and it allows to express transformation equation as matrix multiplication

Cartesian coordinate position (x,y) is represented as homogeneous coordinate triple(x,y,h)

- Represent coordinates as (x,y,h)
- Actual coordinates drawn will be (x/h,y/h)

For Translation

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$P' = T(t_x, t_y) \bullet P$$

For Scaling

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

$$P' = S(s_x, s_y) \bullet P$$

For rotation

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

$$P' = R(\theta) \bullet P$$

Composite Transformations

A composite transformation is a sequence of transformations; one followed by the other. we can set up a matrix for any sequence of transformations as a **composite transformation matrix** by calculating the matrix product of the individual transformations

Translation

If two successive translation vectors (tx1,ty1) and (tx2,ty2) are applied to a coordinate position P, the final transformed location P' is calculated as

P'=T(tx2,ty2).{T(tx1,ty1).P}

={T(tx2,ty2).T(tx1,ty1)}.P

Where P and P' are represented as homogeneous-coordinate column vectors.

$$\begin{bmatrix} 1 & 0 & tx2 \\ 0 & 1 & ty2 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} 1 & 0 & tx1 \\ 0 & 1 & ty1 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & tx1 + tx2 \\ 0 & 1 & ty1 + ty2 \\ 0 & 0 & 1 \end{bmatrix}$$

Or

T(tx2,ty2).T(tx1,ty1) = T(tx1+tx2,ty1+ty2)

Which demonstrated the two successive translations are additive.

Rotations

Two successive rotations applied to point P produce the transformed position

P'=R(θ2).{R(θ1).P}={R(θ2).R(θ1)}.P

By multiplying the two rotation matrices, we can verify that two successive rotation are additive

R(θ2).R(θ1) = R(θ1+ θ2)

So that the final rotated coordinates can be calculated with the composite rotation matrix as

P' = R(θ1+ θ2).P

$$\begin{bmatrix} \cos\theta 2 & -\sin\theta 2 & 0 \\ \sin\theta 2 & \cos\theta 2 & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} \cos\theta 1 & -\sin\theta 1 & 0 \\ \sin\theta 1 & \cos\theta 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta 2+\theta 1) & -\sin(\theta 2+\theta 1) & 0 \\ \sin(\theta 2+\theta 1) & \cos(\theta 2+\theta 1) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

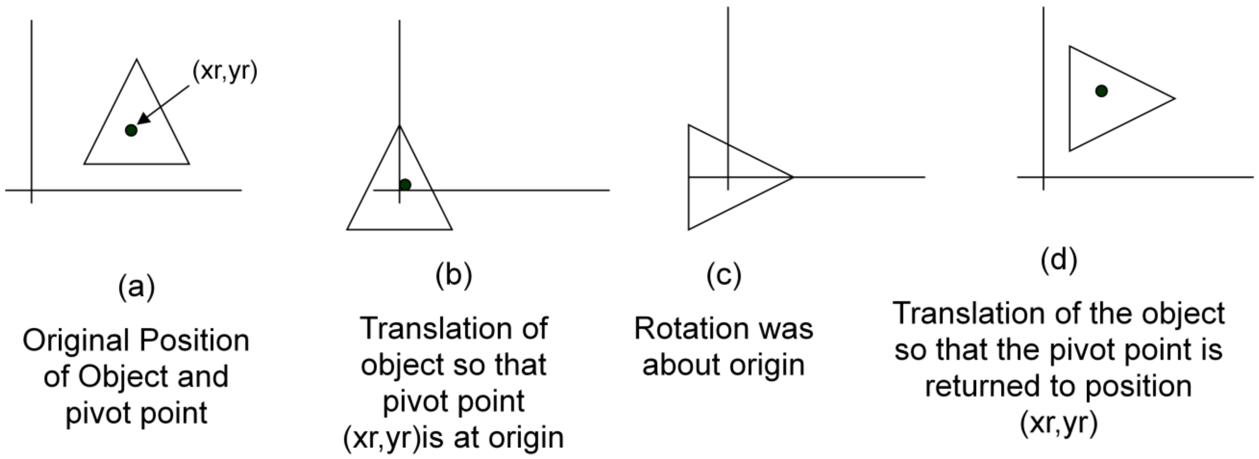
Scaling

Concatenating transformation matrices for two successive scaling operations produces the following composite scaling matrix

$$\begin{bmatrix} sx2 & 0 & 0 \\ 0 & sy2 & 0 \\ 0 & 0 & 1 \end{bmatrix} . \begin{bmatrix} sx1 & 0 & 0 \\ 0 & sy1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} sx2.sx1 & 0 & 0 \\ 0 & sy2.sy1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

General Pivot-point Rotation

1. Translate the object so that pivot-position is moved to the coordinate origin
2. Rotate the object about the coordinate origin
Translate the object so that the pivot point is returned to its original position



(xr,yr)
(a)
Original Position of Object and pivot point
(b)
Translation of object so that pivot point (xr,yr)is at origin
(c)
Rotation was about origin
(d)
Translation of the object so that the pivot point is returned to position (xr,yr)

The composite transformation matrix for this sequence is obtain with the concatenation

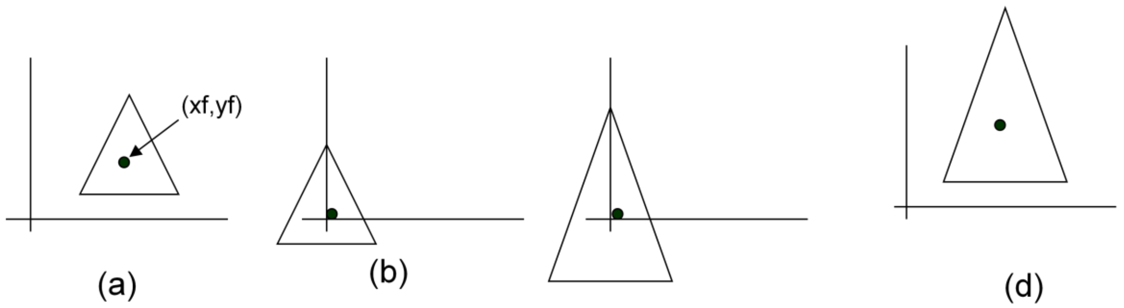
$$\begin{bmatrix} 1 & 0 & xr \\ 0 & 1 & yr \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -xr \\ 0 & 1 & -yr \\ 0 & 0 & 1 \end{bmatrix}$$

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

Which can also be expressed as T(xr,yr).R(θ).T(-xr,-yr) = R(xr,yr,θ)

General fixed point scaling

Translate object so that the fixed point coincides with the coordinate origin
Scale the object with respect to the coordinate origin
Use the inverse translation of step 1 to return the object to its original position



(a) Original Position of Object and Fixed point

(b) Translation of object so that fixed point (xf,yf)is at origin

(c) scaling was about origin

(d) Translation of the object so that the Fixed point is returned to position (xf,yf)

Concatenating the matrices for these three operations produces the required scaling matix

$$\begin{bmatrix} 1 & 0 & xf \\ 0 & 1 & yf \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -xf \\ 0 & 1 & -yf \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} sx & 0 & xf(1- sx) \\ 0 & sy & yf(1- sy) \\ 0 & 0 & 1 \end{bmatrix}$$

Can also be expressed as T(xf,yf).S(sx,sy).T(-xf,-yf) = S(xf, yf, sx, sy)

Note : Transformations can be combined by matrix multiplication

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \left(\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\Theta & -\sin\Theta & 0 \\ \sin\Theta & \cos\Theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \right) \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

Implementation of composite transformations

```
#include <math.h>
#include <graphics.h>
typedef  float Matrix3x3 [3][3];
Matrix3x3 thematrix;

void matrix3x3SetIdentity (Matrix3x3 m)
{
int i,j;
for (i=0; i<3; i++)
for (j=0: j<3; j++ )
m[il[j] = (i == j);
}
```

```
/ * Multiplies  matrix a times b, putting result in b */
void matrix3x3PreMultiply (Matrix3x3 a. Matrix3x3 b)
{
int r,c:
Matrix3x3 tmp:
for (r = 0; r < 3: r++)
for (c = 0; c < 3; c++)
tmp[r][c] =a[r][0]*b[0][c]+ a[r][1]*b[l][c] + a[r][2]*b[2][c]:
for (r = 0: r < 3: r++)
for Ic = 0; c < 3: c++)
b[r][c]=- tmp[r][c]:
}

void translate2 (int tx, int ty)
{
Matrix3x3 m:
rnatrix3x3SetIdentity (m) :
m[0][2] = tx;
m[1][2] = ty:
matrix3x3PreMultiply (m, theMatrix);
}
vold scale2 (float sx. float sy, wcPt2 refpt)
(
Matrix3x3 m.
matrix3x3SetIdentity (m);
m[0] [0] = sx;
m[0][2] = (1 - sx)* refpt.x;
m[l][l] = sy;
m[10][2] = (1 - sy)* refpt.y;
matrix3x3PreMultiply (m, theMatrix);
}

void rotate2 (float a, wcPt2 refPt)
{
Matrix3x3 m;
matrix3x3SetIdentity (m):
a = pToRadians (a);
m[0][0]= cosf (a);
m[0][1] = -sinf (a) ;
m[0] [2] = refPt.x * (1 - cosf (a)) + refPt.y sinf (a);
m[1] [0] = sinf (a);
m[l][l] = cosf (a];
m[l] [2] = refPt.y * (1 - cosf (a) - refPt.x * sinf ( a ) ;
matrix3x3PreMultiply (m, theMatrix);
}

void transformPoints2 (int npts, wcPt2 *pts)
{
int k:
float tmp ;
for (k = 0; k< npts: k++)
{
tmp = theMatrix[0][0]* pts[k] .x * theMatrix[0][1] * pts[k].y+ theMatrix[0][2];
pts[k].y = theMatrix[1][0]* pts[k] .x * theMatrix[1][1] * pts[k].y+ theMatrix[1][2];
pts[k].x =tmp;
}
}

void main (int argc, char  **argv)
{
wcPt2 pts[3]= { 50.0, 50.0, 150.0, 50.0, 100.0, 150.0};
wcPt2 refPt ={100.0. 100.0};
long windowID = openGraphics (*argv,200, 350);
setbackground (WHITE) ;
setcolor (BLUE);
pFillArea(3, pts):
matrix3x3SetIdentity(theMatrix);
scale2 (0.5, 0.5, refPt):
rotate2 (90.0, refPt);
translate2 (0, 150);
transformpoints2 ( 3 , pts)
pFillArea(3.pts);
sleep (10);
closeGraphics (windowID);

}
```

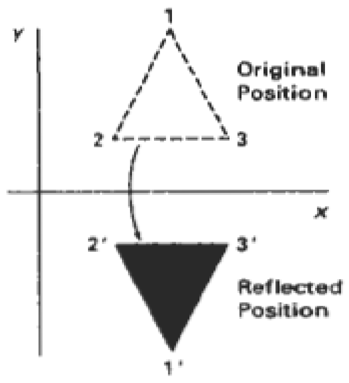
Other Transformations

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^o about the reflection axis. We can choose an axis of reflection in the xy plane or perpendicular to the xy plane or coordinate origin

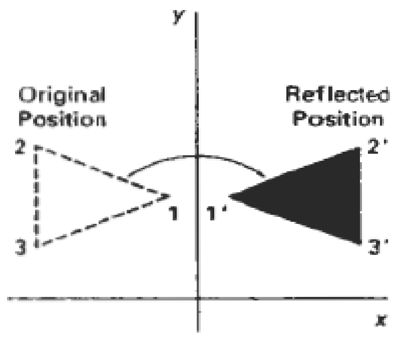
Reflection of an object about the x axis



Reflection the x axis is accomplished with the transformation matrix

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

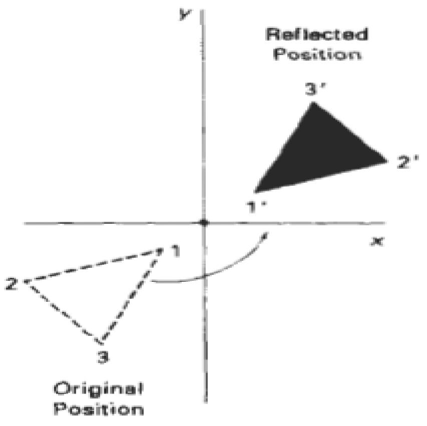
Reflection of an object about the y axis



Reflection the y axis is accomplished with the transformation matrix

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

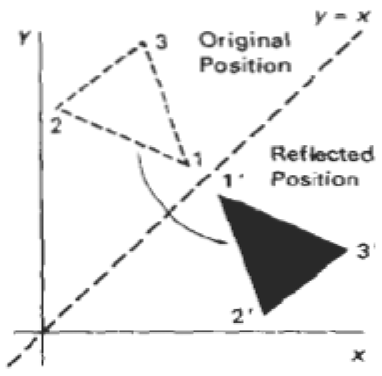
Reflection of an object about the coordinate origin



Reflection about origin is accomplished with the transformation matrix

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

Reflection axis as the diagonal line y = x



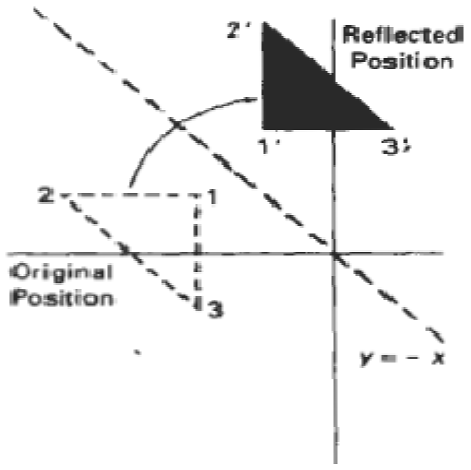
To obtain transformation matrix for reflection about diagonal y=x the transformation sequence is

1. Clock wise rotation by 45^0
2. Reflection about x axis
3. counter clock wise by 45^0

Reflection about the diagonal line y=x is accomplished with the transformation matrix

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

Reflection axis as the diagonal line y = -x



To obtain transformation matrix for reflection about diagonal y=-x the transformation sequence is

1. Clock wise rotation by 45^0
2. Reflection about y axis
3. counter clock wise by 45^0

Reflection about the diagonal line y=-x is accomplished with the transformation matrix

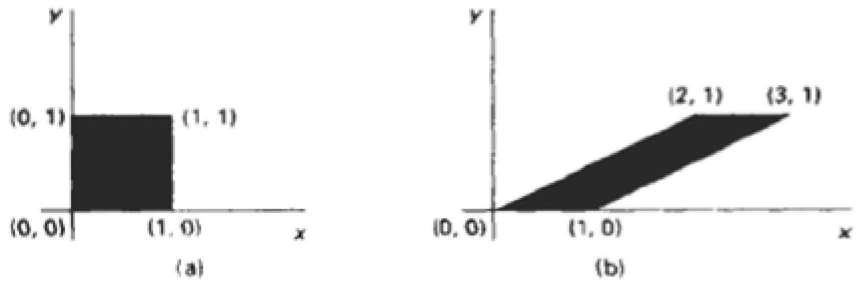
$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

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.

X- Shear

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

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

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

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

which transforms the coordinates as

x' =x

y' = y+ sh_y.x

XY-Shear

The transformation matrix for xy-shear

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & sh_x & 0 \\ sh_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

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

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

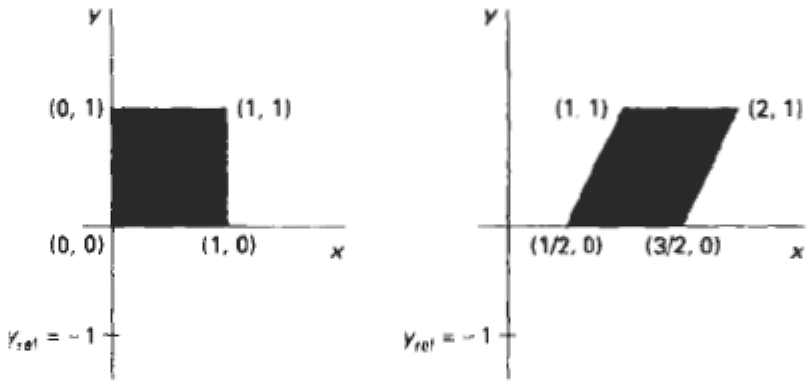
which transforms the coordinates as

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

y' = y

Example

$Sh_x = ½$ and y_{ref}=-1



Y shear with x reference line

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

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

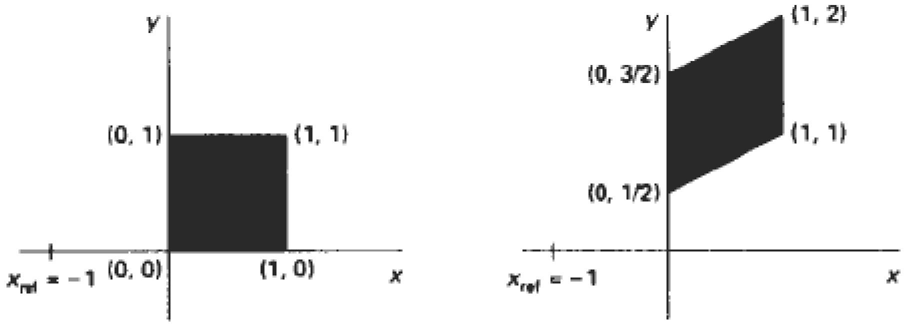
which transforms the coordinates as

x' =x

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

Example

Sh_y = ½ and x_{ref}=-1



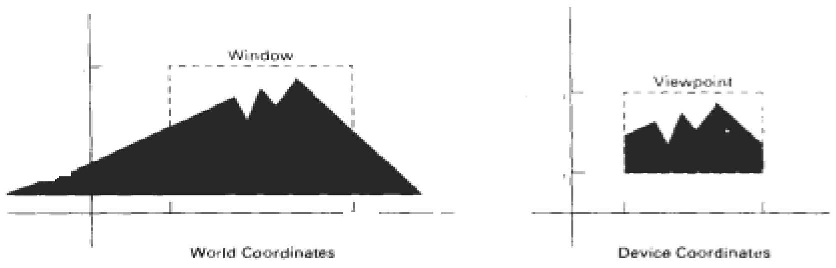
Two dimensional viewing

The viewing pipeline

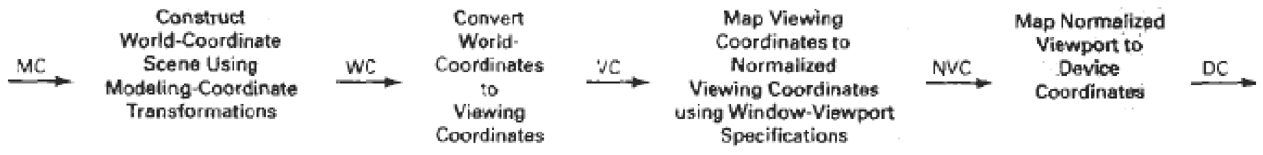
A world coordinate area selected for display is called a window. An area on a display device to which a window is mapped is called a view port. The window defines what is to be viewed the view port defines where it is to be displayed.

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

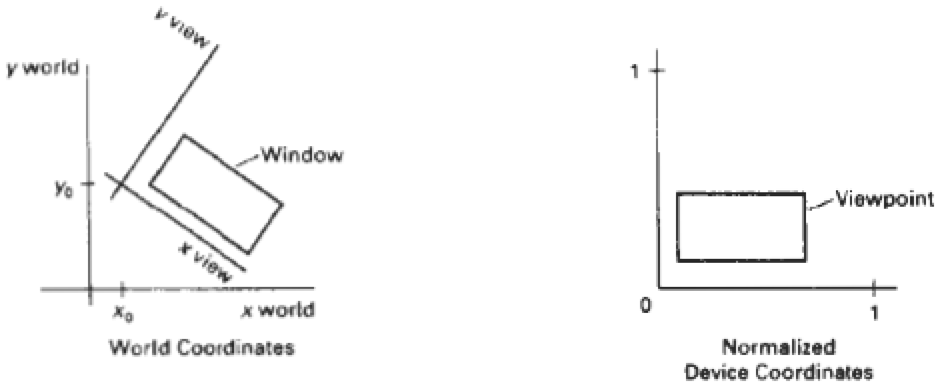

MC
Construct World-Coordinate Scene Using Modeling-Coordinate Transformations
WC
Convert World-Coordinates to Viewing Coordinates
VC
Map Viewing Coordinates to Normalized Viewing Coordinates using Window-Viewport Specifications
NVC
Map Normalized Viewport to Device Coordinates
DC

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.

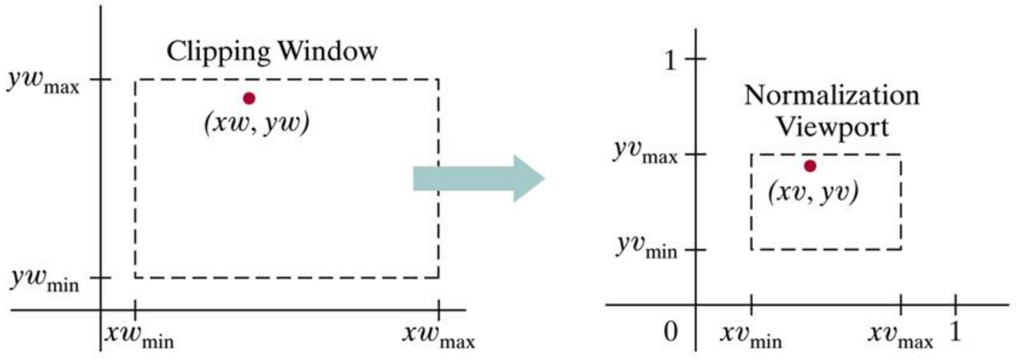
The viewing- coordinate reference frame is used to provide a method for setting up arbitrary orientations for rectangular windows. Once the viewing reference frame is established, we can transform descriptions in world coordinates to viewing coordinates.

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)

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

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

where scaling factors are

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

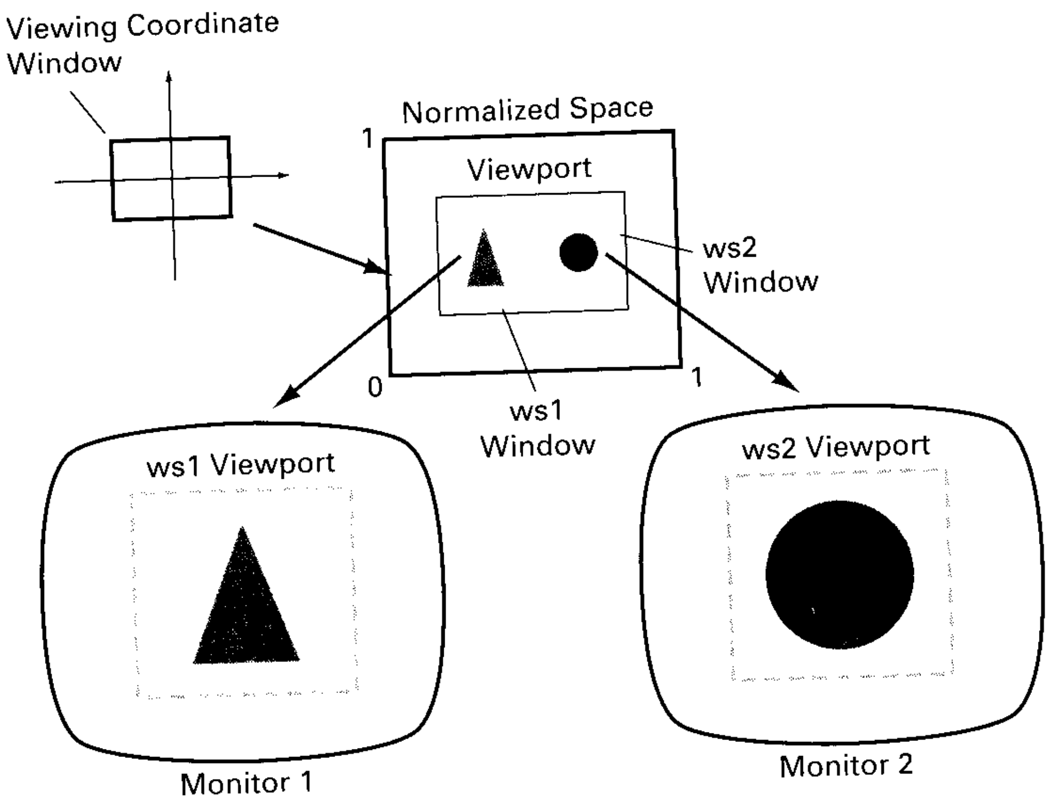
The conversion is performed with the following sequence of transformations.

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.

Mapping selected parts of a scene in normalized coordinate to different video monitors with work station transformation.



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 designates 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:

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

Line and polygon clipping routines are standard components of graphics packages.

Point Clipping

Clip window is a rectangle in standard position. A point P=(x,y) for display, if following inequalities are satisfied:

$$xw_{min} <= x <= xw_{max}$$

$$yw_{min} <= y <= yw_{max}$$

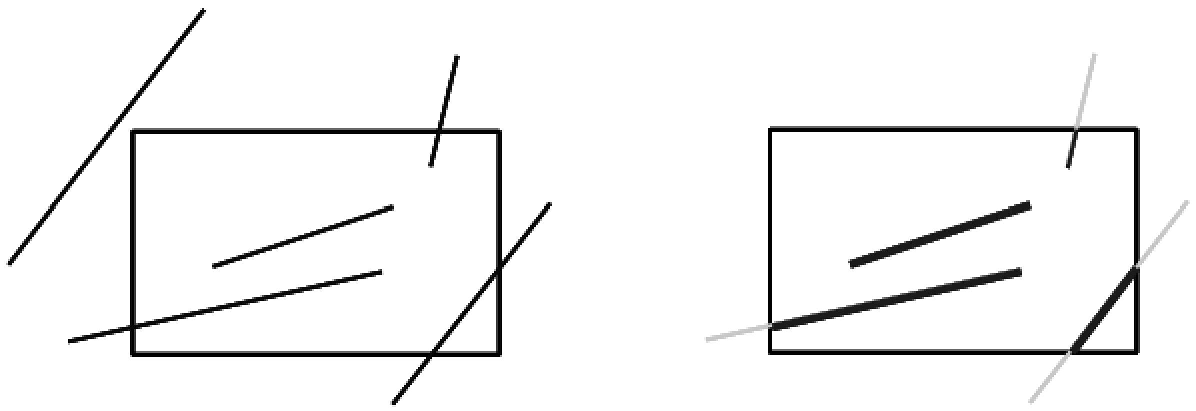
where the edges of the clip window **(xw_{min},xw_{max},yw_{min},yw_{max})** can be either the world-coordinate window boundaries or viewport boundaries. If any one of these four inequalities is not satisfied, the point is clipped (not saved for display).

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 point outside any one of the clip boundaries line P3P4 is outside the window.

Line clipping against a rectangular clip window



All other lines cross one or more clipping boundaries. For a line segment with end points (x_1,y_1) and (x_2,y_2) one or both end points outside clipping rectangle, the parametric representation

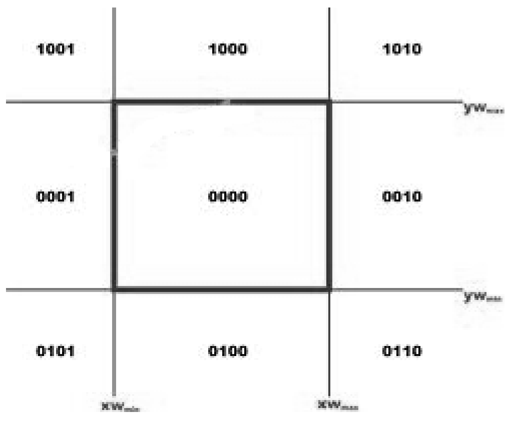
$$x = x_1 + u(x_2 - x_1),$$
$$y = y_1 + u(y_2 - y_1), \quad 0 \le u \le 1$$

could be used to determine values of u for an intersection with the clipping boundary coordinates. If the value of u for an intersection with a rectangle boundary edge is outside the range of 0 to 1, the line does not enter the interior of the window at that boundary. If the value of u is within the range from 0 to 1, the line segment does indeed cross into the clipping area. This method can be applied to each clipping boundary edge in to determined whether any part of line segment is to displayed.

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.

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 $yw_{max} - y$.

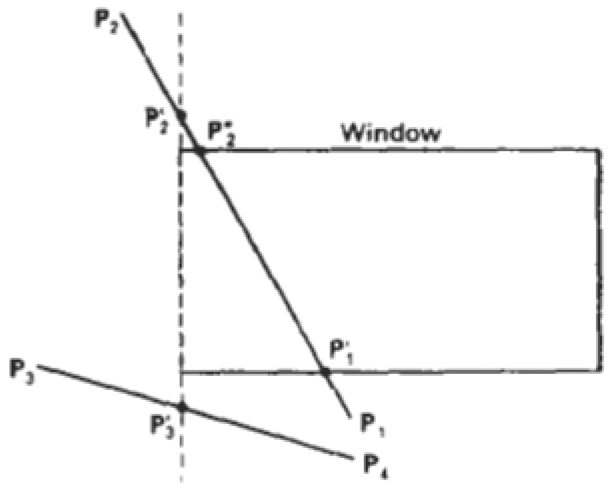
Once we have established region codes for all line endpoints, we can quickly determine which lines are completely inside the clip window and which are clearly outside.

Any lines that are completely contained within the window boundaries have a region code of 0000 for both endpoints, and we accept these lines. Any lines that have a 1 in the same bit position in the region codes for each endpoint are completely outside the clipping rectangle, and we reject these lines.

We would discard the line that has a region code of 1001 for one endpoint and a code of 0101 for the other endpoint. Both endpoints of this line are left of the clipping rectangle, as indicated by the 1 in the first bit position of each region code.

A method that can be used to test lines for total clipping is to perform the logical and operation with both region codes. If the result is not 0000,the line is completely outside the clipping region.

Lines that cannot be identified as completely inside or completely outside a clip window by these tests are checked for intersection with window boundaries.



Line extending from one coordinates region to another may pass through the clip window, or they may intersect clipping boundaries without entering window.

Cohen-Sutherland line clipping starting with bottom endpoint left, right , bottom and top boundaries in turn and find that this point is below the clipping rectangle.

Starting with the bottom endpoint of the line from $\mathbf{P_1}$ to $\mathbf{P_2}$**,** we check P_1 against the left, right, and bottom boundaries in turn and find that this point is below the clipping rectangle. We then find the intersection point P_1' with the bottom boundary and discard the line section from P_1 to P_1'.

The line now has been reduced to the section from P_1' to P_2**,**Since P_2**,** is outside the clip window, we check this endpoint against the boundaries and find that it is to the left of the window. Intersection point P_2’ is calculated, but this point is above the window. So the final intersection calculation yields P_2”**,** and the line from P_1’ to P_2”is saved. This completes processing for this line, so we save this part and go on to the next line.

Point P_3 in the next line is to the left of the clipping rectangle, so we determine the intersection P_3’**,** and eliminate the line section from P_3 to P_3'. By checking region codes for the line section from P_3'to P_4 we find that the remainder of the line is below the clip window and can be discarded also.

Intersection points with a clipping boundary can be calculated using the slope-intercept form of the line equation. For a line with endpoint coordinates (x_1,y_1) and (x_2,y_2) and the y coordinate of the intersection point with a vertical boundary can be obtained with the calculation.

$$y = y_1 + m (x-x_1)$$

where x value is set either to xw_{min} or to xw_{max} and slope of line is calculated as

$$m = (y_2 - y_1) / (x_2 - x_1)$$

the intersection with a horizontal boundary the x coordinate can be calculated as

$$x = x_1 + (y - y_1) / m$$

with y set to either to yw_{min} or to yw_{max}.

Implementation of Cohen-sutherland Line Clipping

```
#define Round(a)     ((int)(a+0.5))

#define LEFT_EDGE         0x1
#define RIGHT_EDGE        0x2
#define BOTTOM_EDGE  0x4
#define TOP_EDGE          0x8
#define TRUE 1
#define FALSE 0

#define INSIDE(a) (!a)
#define REJECT(a,b) (a&b)
#define ACCEPT(a,b) (!(a|b))

unsigned char encode(wcPt2 pt, dcPt winmin, dcPt winmax)
{
```

```
unsigned char code=0x00;
if(pt.x<winmin.x)
code=code|LEFT_EDGE;
if(pt.x>winmax.x)
code=code|RIGHT_EDGE;
if(pt.y<winmin.y)
code=code|BOTTOM_EDGE;
if(pt.y>winmax.y)
code=code|TOP_EDGE;
return(code);
}
void swappts(wcPt2 *p1,wcPt2 *p2)
{
wcPt2 temp;
tmp=*p1;
*p1=*p2;
*p2=tmp;
}
void swapcodes(unsigned char *c1,unsigned char *c2)
{
unsigned char tmp;
tmp=*c1;
*c1=*c2;
*c2=tmp;
}
void clipline(dcPt winmin, dcPt winmax, wcPt2 p1,ecPt2 point p2)
{
unsigned char code1,code2;
int done=FALSE, draw=FALSE;
float m;
while(!done)
{
code1=encode(p1,winmin,winmax);
code2=encode(p2,winmin,winmax);
if(ACCEPT(code1,code2))
{
done=TRUE;
draw=TRUE;
}
else if(REJECT(code1,code2))
done=TRUE;
else
{
```

```
if(INSIDE(code1))
{
swappts(&p1,&p2);
swapcodes(&code1,&code2);
}
if(p2.x!=p1.x)
m=(p2.y-p1.y)/(p2.x-p1.x);
if(code1 &LEFT_EDGE)
{
p1.y+=(winmin.x-p1.x)*m;
p1.x=winmin.x;
}
else if(code1 &RIGHT_EDGE)
{
p1.y+=(winmax.x-p1.x)*m;
p1.x=winmax.x;
}
else if(code1 &BOTTOM_EDGE)
{
if(p2.x!=p1.x)
p1.x+=(winmin.y-p1.y)/m;
p1.y=winmin.y;
}
else if(code1 &TOP_EDGE)
{
if(p2.x!=p1.x)
p1.x+=(winmax.y-p1.y)/m;
p1.y=winmax.y;
}
}
}
if(draw)
lineDDA(ROUND(p1.x),ROUND(p1.y),ROUND(p2.x),ROUND(p2.y));
}
```

Liang – Barsky line Clipping:

Based on analysis of parametric equation of a line segment, faster line clippers have been developed, which can be written in the form :

$$x = x_1 + u\,\Delta x$$
$$y = y_1 + u\,\Delta y \qquad 0<=u<=1$$

where $\Delta x = (x_2 - x_1)$ and $\Delta y = (y_2 - y_1)$

In the Liang-Barsky approach we first the point clipping condition in parametric form :

$xw_{min} <= x_1 + u\,\Delta x <=. xw_{max}$

$yw_{min} <= y_1 + u\,\Delta y <= yw_{max}$

Each of these four inequalities can be expressed as

$\mu p_k <= q_k$. k=1,2,3,4

the parameters p & q are defined as

$p_1 = -\Delta x$ $q_1 = x_1 - xw_{min}$
$p_2 = \Delta x$ $q_2 = xw_{max} - x_1$
$P_3 = -\Delta y$ $q_3 = y_1 - yw_{min}$
$P_4 = \Delta y$ $q_4 = yw_{max} - y_1$

Any line that is parallel to one of the clipping boundaries have $p_k=0$ for values of k corresponding to boundary k=1,2,3,4 correspond to left, right, bottom and top boundaries. For values of k, find $q_k<0$, the line is completely out side the boundary.

If $q_k >=0$, the line is inside the parallel clipping boundary.

When $p_k<0$ the infinite extension of line proceeds from outside to inside of the infinite extension of this clipping boundary.

If $p_k>0$, the line proceeds from inside to outside, for non zero value of p_k calculate the value of u, that corresponds to the point where the infinitely extended line intersect the extension of boundary k as

$u = qk / pk$

For each line, calculate values for parameters $u_1$and u_2 that define the part of line that lies within the clip rectangle. The value of u1 is determined by looking at the rectangle edges for which the line proceeds from outside to the inside (p<0).

For these edges we calculate

$r_k = qk / pk$

The value of u1 is taken as largest of set consisting of 0 and various values of r. The value of u2 is determined by examining the boundaries for which lines proceeds from inside to outside (P>0).

A value of $r_k$is calculated for each of these boundaries and value of u2 is the minimum of the set consisting of 1 and the calculated r values.

If u1>u2, the line is completely outside the clip window and it can be rejected.

Line intersection parameters are initialized to values u1=0 and u2=1. for each clipping boundary, the appropriate values for P and q are calculated and used by function

Cliptest to determine whether the line can be rejected or whether the intersection parameter can be adjusted.

When p<0, the parameter r is used to update u1.

When p>0, the parameter r is used to update u2.

If updating u1 or u2 results in u1>u2 reject the line, when p=0 and q<0, discard the line, it is parallel to and outside the boundary.If the line has not been rejected after all four value of p and q have been tested , the end points of clipped lines are determined from values of u1 and u2.

The Liang-Barsky algorithm is more efficient than the Cohen-Sutherland algorithm since intersections calculations are reduced. Each update of parameters u1 and u2 require only one division and window intersections of these lines are computed only once.

Cohen-Sutherland algorithm, can repeatedly calculate intersections along a line path, even through line may be completely outside the clip window. Each intersection calculations require both a division and a multiplication.

Implementation of Liang-Barsky Line Clipping

```
#define Round(a)     ((int)(a+0.5))
int clipTest (float p, float q, gfloat *u1, float *u2)
{
float r;
int retval=TRUE;
if (p<0.0)
{
```

```
r=q/p
 if (r>*u2)
 retVal=FALSE;
 else
if (r>*u1)
 *u1=r;
}
else
if (p>0.0)
{
r=q/p
if (r<*u1)
retVal=FALSE;
else
if (r<*u2)
*u2=r;
}
else
if )q<0.0)
retVal=FALSE
return(retVal);

void clipLine (dcPt winMin, dcPt winMax, wcPt2 p1, wcpt2 p2)
{
float u1=0.0, u2=1.0, dx=p2.x-p1.x,dy;
if (clipTest (-dx, p1.x-winMin.x, &u1, &u2))
if (clipTest (dx, winMax.x-p1.x, &u1, &u2))
{
dy=p2.y-p1.y;
if (clipTest (-dy, p1.y-winMin.y, &u1, &u2))
if (clipTest (dy, winMax.y-p1.y, &u1, &u2))
{
if (u1<1.0)
{
p2.x=p1.x+u2*dx;
p2.y=p1.y+u2*dy;
}
if (u1>0.0)
{
p1.x=p1.x+u1*dx;
p1.y=p1.y+u1*dy;
}
lineDDA(ROUND(p1.x),ROUND(p1.y),ROUND(p2.x),ROUND(p2.y));
```

}
}
}

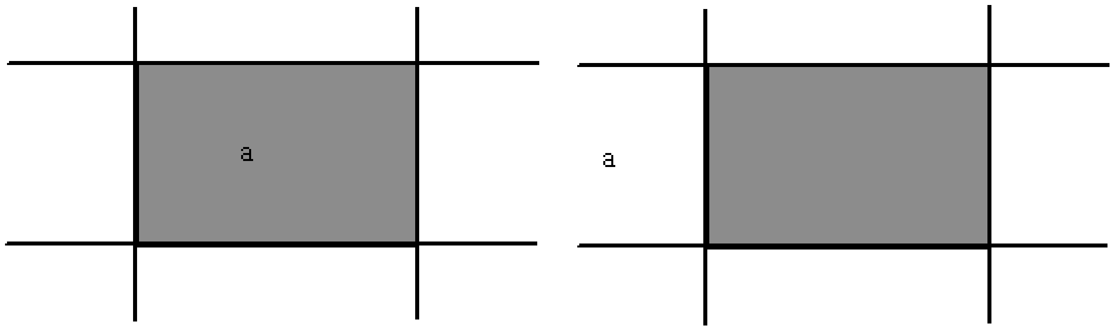
Nicholl-Lee-Nicholl Line clipping

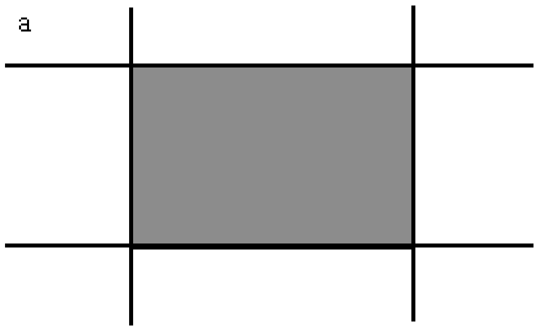
By creating more regions around the clip window, the Nicholl-Lee-Nicholl (or NLN) algorithm avoids multiple clipping of an individual line segment. In the Cohen-Sutherland method, multiple intersections may be calculated.These extra intersection calculations are eliminated in the NLN algorithm by carrying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





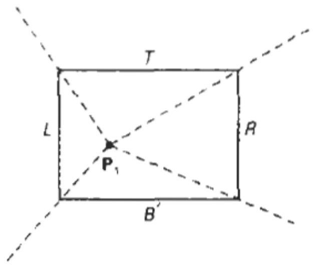
Case 1: p1 inside region

Case 2: p1 across edge

Case 3: p1 across corner

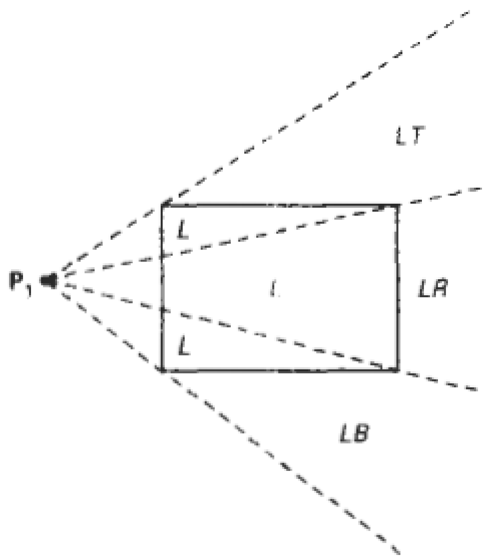
Next, we determine the position of P2 relative to P1**.** To do this, we create some new regions in the plane, depending on the location of P1. Boundaries of the new regions are half-infinite line segments that start at the position of P1 and pass through the window corners. If P1 is inside the clip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

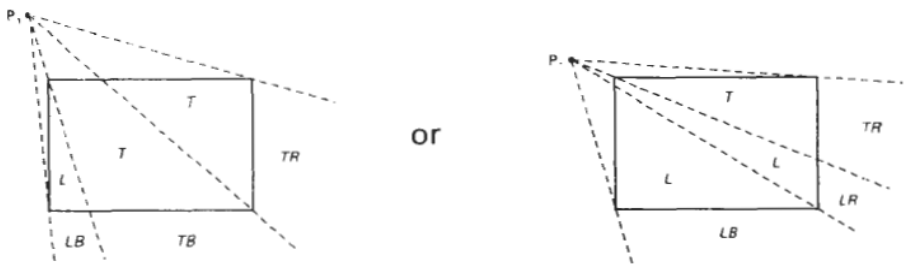
If P1 is in the region to the left of the window, we set up the four regions, ***L, LT, LR,*** and ***LB,*** shown in Fig.



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

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

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 $\mathbf{x} = \mathbf{x_L}$**,,** with

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

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

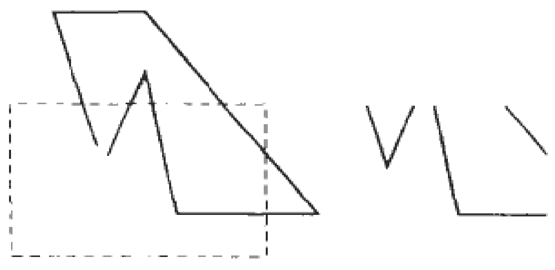
And an intersection position on the top boundary has $y = \mathbf{y_T}$ and $u = (y_T - y_1)/(y_2 - y_1)$ with

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

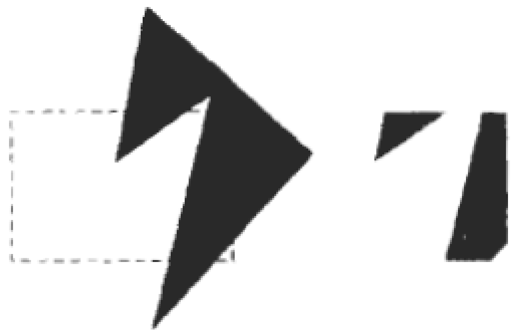
POLYGON CLIPPING

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:

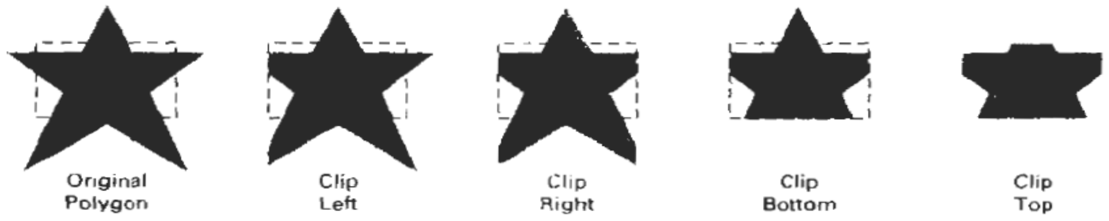
A polygon can be clipped by processing the polygon boundary as a whole against each window edge. This could be accomplished by processing all polygon vertices against each clip rectangle boundary.

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:

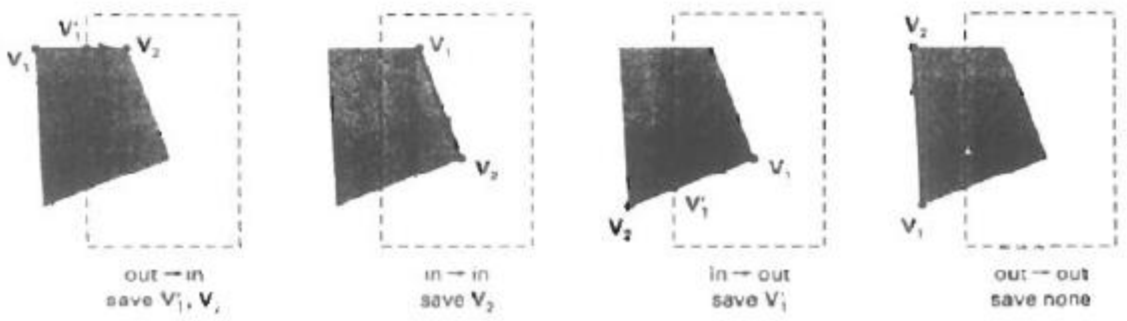
1. If the first vertex is outside the window boundary and second vertex is inside, both the intersection point of the polygon edge with window boundary and second vertex are added to output vertex list.
2. If both input vertices are inside the window boundary, only the second vertex is added to the output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

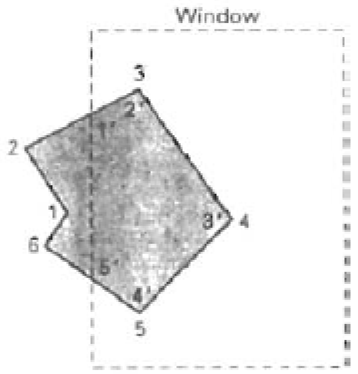
Clipping a polygon against successive window boundaries.



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.

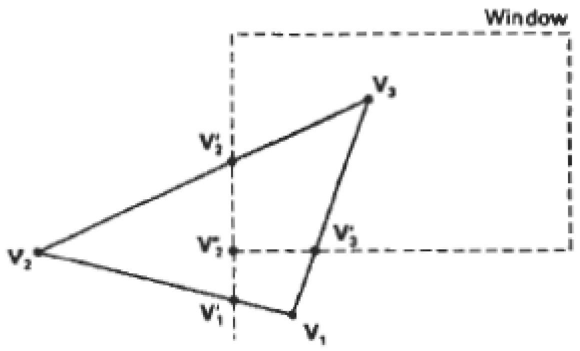


vertices 1 and 2 are found to be on outside of boundary. Moving along vertex 3 which is inside, calculate the intersection and save both the intersection point and vertex 3. Vertex 4 and 5 are determined to be inside and are saved. Vertex 6 is outside so we find and save the intersection point. Using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

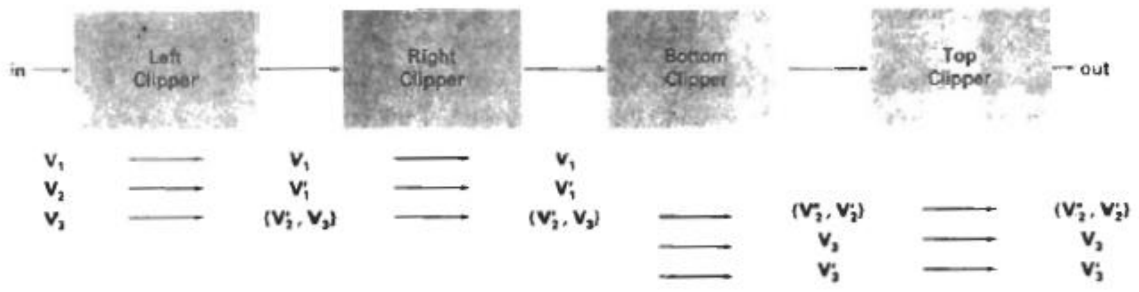
Implementing the algorithm as described requires setting up storage for an output list of vertices as a polygon clipped against each window boundary. We eliminate the intermediate output vertex lists by simply by clipping individual vertices at each step and passing the clipped vertices on to the next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

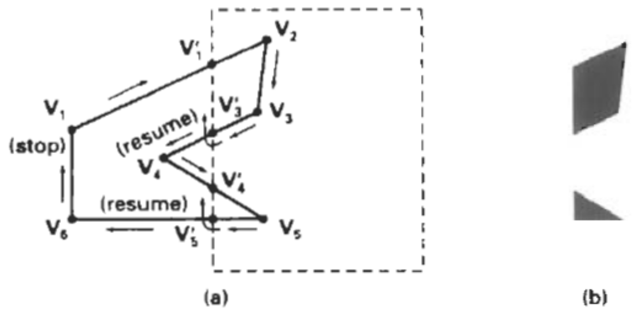
Weiler- Atherton Polygon Clipping

This clipping procedure was developed as a method for identifying visible surfaces, and so it can be applied with arbitrary polygon-clipping regions.

The basic idea in this algorithm is that instead of always proceeding around the polygon edges as vertices are processed, we sometimes want to follow the window boundaries. Which path we follow depends on the polygon-processing direction (clockwise or counterclockwise) and whether the pair of polygon vertices currently being processed represents an outside-to-inside pair or an inside- to-outside pair. For clockwise processing of polygon vertices, we use the following rules:

- For an outside-to-inside pair of vertices, follow the polygon boundary.
- For an inside-to-outside pair of vertices,. follow the window boundary in a clockwise direction.

In the below Fig. the processing direction in the Weiler-Atherton algorithm and the resulting clipped polygon is shown for a rectangular clipping window.



An improvement on the Weiler-Atherton algorithm is the Weiler algorithm, which applies constructive solid geometry ideas to clip an arbitrary polygon against any polygon clipping region.

Curve Clipping

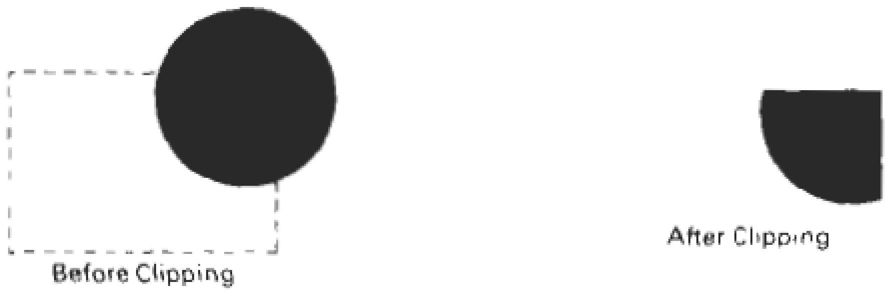
Curve-clipping procedures will involve nonlinear equations, and this requires more processing than for objects with linear boundaries. The bounding rectangle for a circle or other curved object can be used first to test for overlap with a rectangular clip window.

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

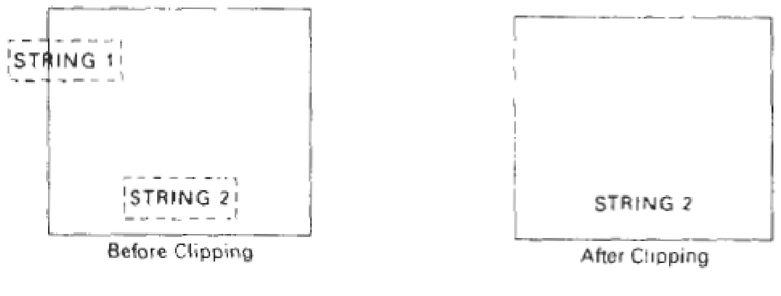


Text clipping

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.

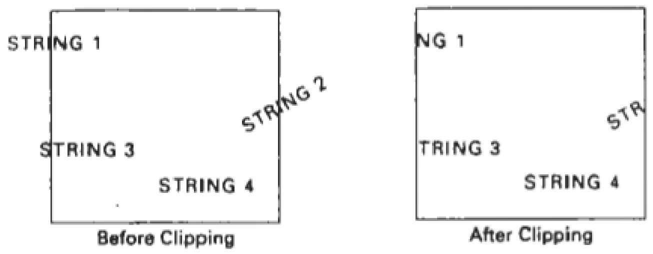
The simplest method for processing character strings relative to a window boundary is to use the **all-or-none string-clipping** strategy shown in Fig. . If all of the string is inside a clip window, we keep it. Otherwise, the string is discarded. This procedure is implemented by considering a bounding 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



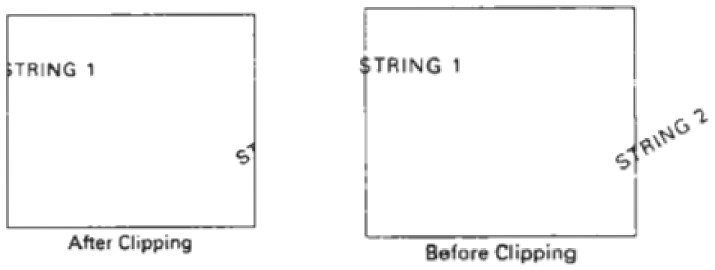
An alternative to rejecting an entire character string that overlaps a window boundary is to use the **all-or-none character-clipping** strategy. Here we discard only those characters that are not completely inside the window .In this case, the boundary limits of individual characters are compared to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

Text clipping using a bounding rectangle about individual characters.



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 overlap 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.

2.1 Three Dimensional Concepts

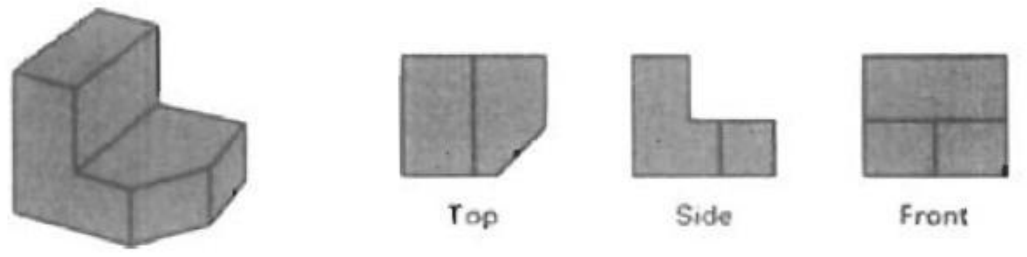
2.1.1 Three Dimensional Display Methods:

- 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.
- The appearance of the solid object can be reconstructed from the major views.

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:

- Surface rendering method is used to generate a degree of realism in a displayed scene.

- 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.

2.1.2 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)

- Fillarea3(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.

2.2 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

2.2.1 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.

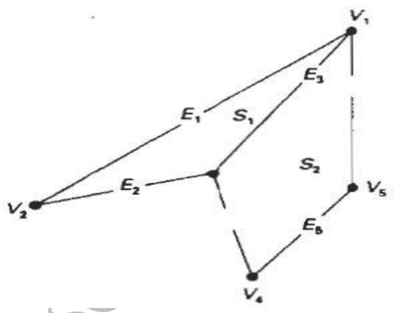
2. The Edge Table

   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

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

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.

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

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

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

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

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

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

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

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

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

- 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:

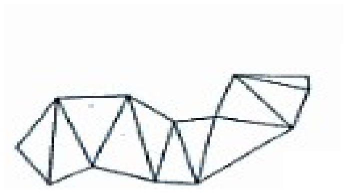
  If $Ax + By + Cz + D < 0$, the point (x, y, z) is inside the surface.

  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.

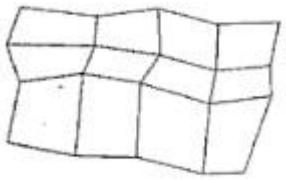
Polygon Meshes

- 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.



2.2.2 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.

- When functions are specified, a package can project the defining equations for a curve to the display plane and plot pixel positions along the path of the projected function.
- For surfaces, a functional description in decorated to produce a polygon-mesh approximation to the surface.

2.2.3 Quadric Surfaces

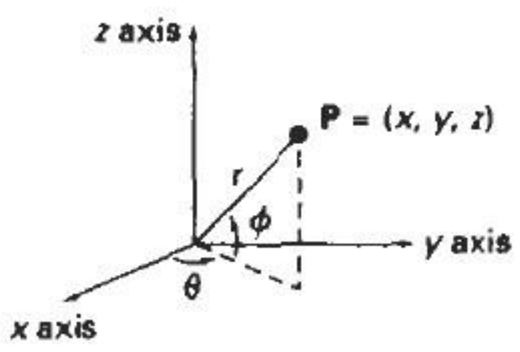
- The quadric surfaces are described with second degree equations (quadratics).
- They include spheres, ellipsoids, tori, parabolids, and hyperboloids.

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.

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

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



$$x = r\cos\varphi\cos\theta, \quad -\text{ʌ}/2 <= \varphi <= \text{ʌ}/2$$

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

$$z = r\sin\varphi$$

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

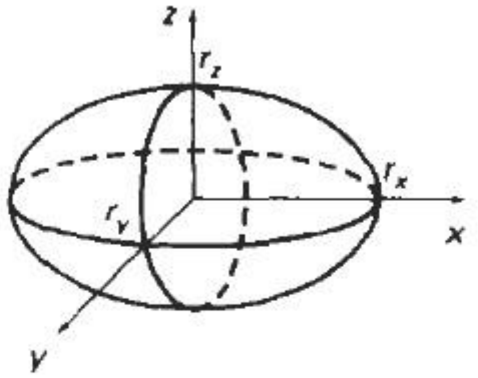
Ellipsoid

- 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



$x = r_x \cos\varphi \cos\theta, \quad -\pi/2 <= \varphi <= \pi/2$

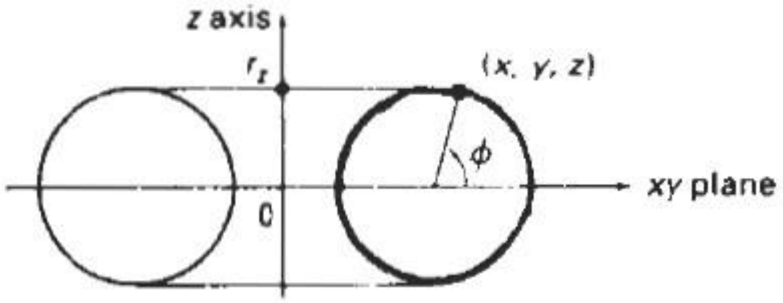
$y = r_y \cos\varphi \sin\theta, \quad -\pi <= \varphi <= \pi$

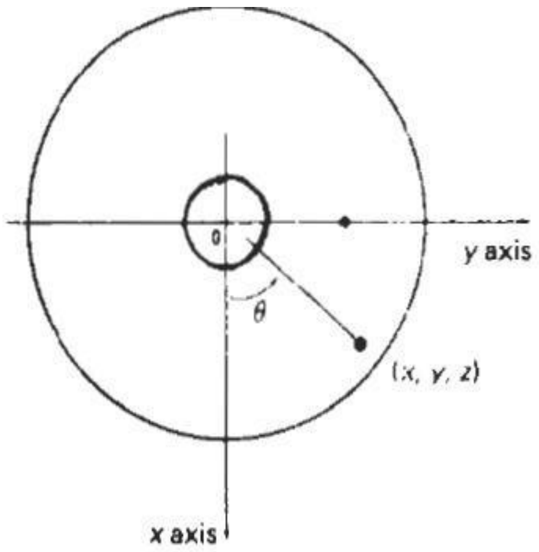
$z = r_z \sin\varphi$

Torus

- Torus is a doughnut shaped object.
- It can be generated by rotating a circle or other conic about a specified axis.

A torus with a circular cross section centered on the coordinate origin





- The Cartesian representation for points over the surface of a torus can be written in the form

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

where r in any given offset value.

- Parametric representation for a torus is similar to those for an ellipse, except that angle φ extends over 360^o.
- Using latitude and longitude angles φ and θ, we can describe the torus surface as the set of points that satisfy.

$$x = r_x (r + \cos\varphi) \cos\theta, \qquad -\Lambda <= \varphi <= \Lambda$$

$$y = r_y(r + \cos\varphi)\sin\theta, \qquad -\Lambda <= \varphi <= \Lambda$$

$$z = r_z \sin\varphi$$

2.2.4 Spline Representations

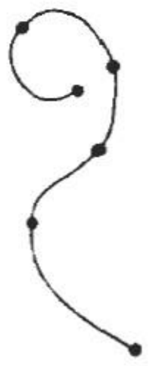
- A Spline is a flexible strip used to produce a smooth curve through a designated set of points.
- Several small weights are distributed along the length of the strip to hold it in position on the drafting table as the curve is drawn.
- The Spline curve refers to any sections curve formed with polynomial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

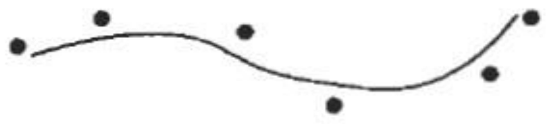
- Spline curve can be specified by a set of coordinate positions called control points which indicates the general shape of the curve.
- These control points are fitted with piecewise continuous parametric polynomial functions in one of the two ways.
  1. When polynomial sections are fitted so that the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

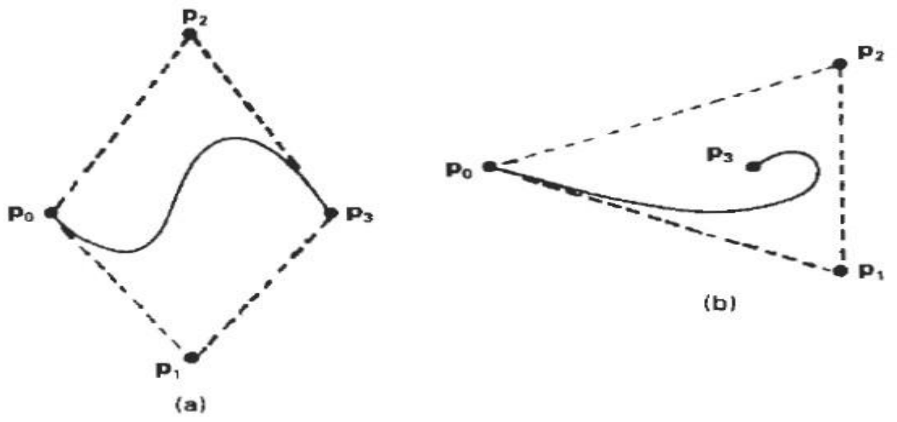
     A set of six control points approximated with piecewise continuous polynomial sections

- 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

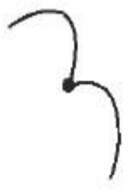
$$x = x(u), y = y(u), z = z(u), u1<= u <= u2 \quad\quad -----(a)$$

- 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.

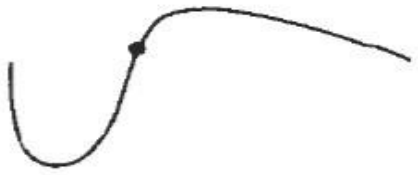
- Higher order parametric continuity conditions are defined similarly.

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

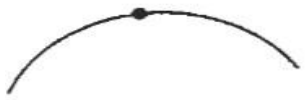
a)Zero order continuity only



b)First order continuity only



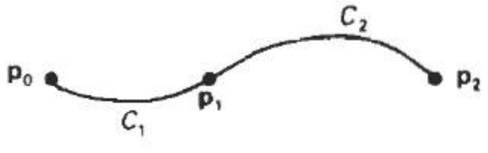
c) Second order continuity only



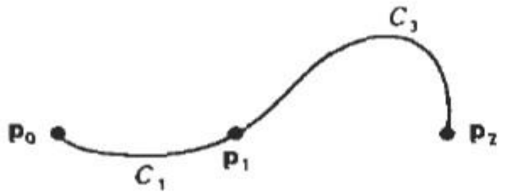
Geometric Continuity Conditions

- 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.



(b)

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.

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

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

$$x(u) = [u^3 \quad u^2 \quad u^1 \quad 1] \begin{bmatrix} a_x \\ b_x \\ c_x \\ d_x \end{bmatrix} \qquad \text{-------(2)}$$

$$= U.C$$

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

$$C = M_{spline} \,.\, M_{geom} \qquad \text{-----(3)}$$

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.

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

- 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.

2.3 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

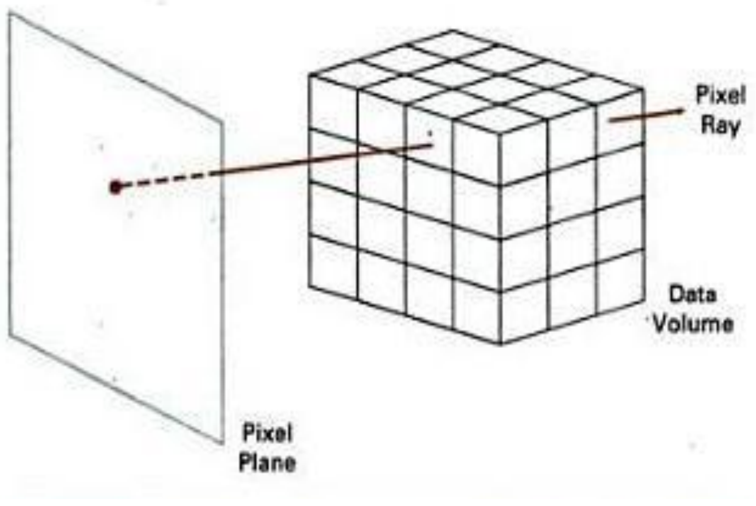
Sometimes isolines are plotted with spline curves but spline fitting can lead to misinterpretation of the data sets. Example two spline isolines could cross or curved isoline paths might not be a true indicator of data trends since data values are known only at the cell corners.

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.

For this volume visualization, a color-coded plot of the distance to the maximum voxel value along each pixel ray was displayed.

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.

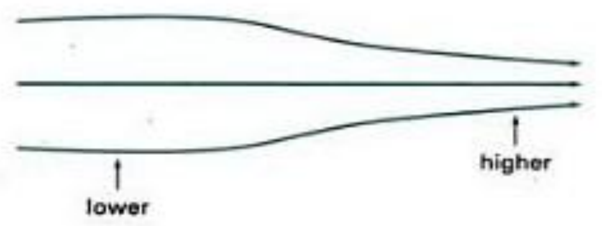
As with scalars, vector quantities may be functions of position, time, and other parameters. Some examples of physical vector quantities are velocity, acceleration, force, electric fields, magnetic fields, gravitational fields, and electric current.

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* diretions 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.

2.4 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.

2.4.1 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.

$$\begin{array}{c} x' \\ y' \\ z' \\ 1 \end{array} = \begin{array}{cccc} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & y_z \\ 0 & 0 & 0 & 1 \end{array} \begin{array}{c} x \\ y \\ z \\ 1 \end{array} \quad \text{--------} (1)$$

(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.

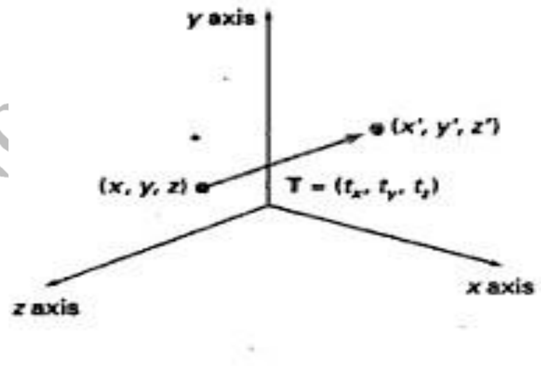
The matrix representation in equation (1) is equivalent to the three equations

$x' = x + t_x$

$y' = y + t_y$

$z' = z + t_z$ ------------------------------(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.

2.4.2 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.

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

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

$$z' = z \qquad \text{--------------------------(2)}$$

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

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

which we can write more compactly as

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

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

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

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

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

which can be written in the homogeneous coordinate form

$$\begin{array}{c} x' \\ y' \\ z' \\ 1 \end{array} = \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{array} \quad \begin{array}{c} x \\ y \\ z \\ 1 \end{array} \qquad \text{-------(7)}$$

(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.

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

The matrix representation for y-axis rotation is

$$x' \qquad \cos\theta \quad 0 \quad \sin\theta \quad 0 \qquad x$$

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

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

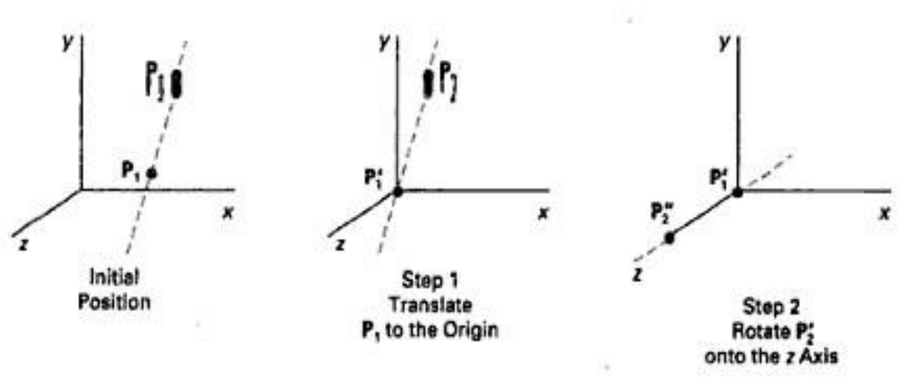
General Three Dimensional Rotations

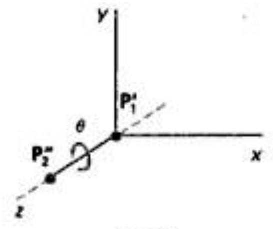
- A rotation matrix for any axis that does not coincide with a coordinate axis can be set up as a composite transformation involving combinations of translations and the coordinate axes rotations.
- We obtain the required composite matrix by
    1. Setting up the transformation sequence that moves the 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.

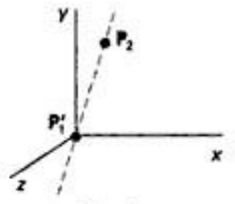
- In the special case where an object is to be rotated about an axis that is parallel to one of the coordinate axes, we can attain the desired rotation with the following transformation sequence:
  1. Translate the object so that the rotation axis coincides with the parallel coordinate axis.
  2. Perform 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.

Five transformation steps

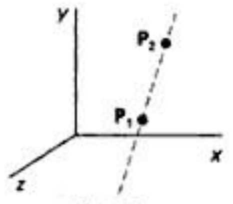

y
P_2
P_1
x
z
Initial Position
y
P_2
P_1'
x
z
Step 1 Translate P_1 to the Origin
y
P_1'
P_2''
x
z
Step 2 Rotate P_2' onto the z Axis



Step 3
Rotate the
Object Around the
z Axis



Step 4
Rotate the Axis
to the Original
Orientation



Step 5
Translate the
Rotation Axis
to the Original
Position

2.4.3 Scaling

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

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

(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

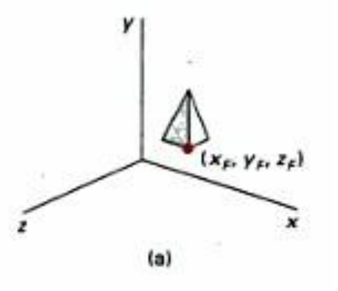
$$x' = x.s_x$$
$$y' = y.s_y \quad \text{----------(13)}$$
$$z' = z.s_z$$

- 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:

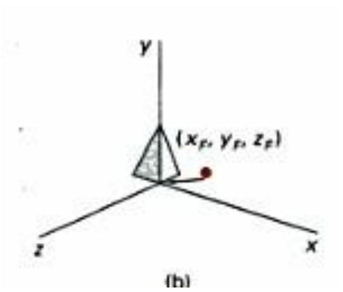
  1. Translate the fixed point to the origin.

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

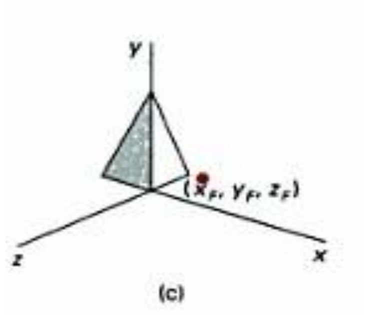
3. Translate the fixed point back to its original position. This sequence of transformation is shown in the below figure .



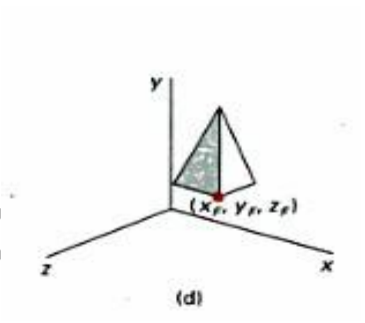
(a)



(b)



(c)



(d)

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

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

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

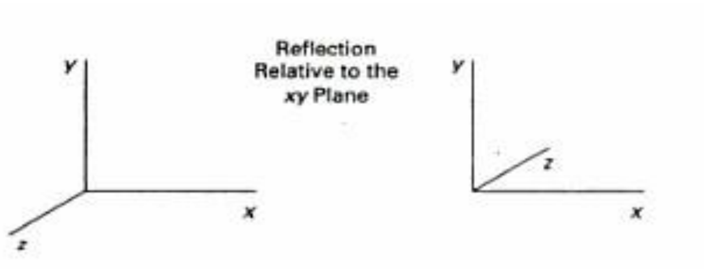
- Inverse scaling matrix m formed by replacing the scaling parameters s_x, s_y and s_z with their reciprocals.
- The inverse matrix generates an opposite scaling transformation, so the concatenation of any scaling matrix and its inverse produces the identity matrix.

2.4.4 Other Transformations

Reflections

- A 3D reflection can be performed relative to a selected reflection axis or with respect to a selected reflection plane.
- Reflection relative to a given axis are equivalent to 180^0 rotations about the axis.

- 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

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

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

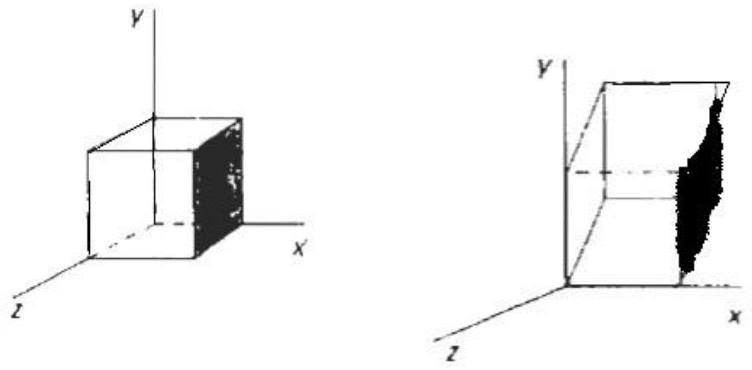
Shears

- 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.

$$SH_z = \begin{matrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

- 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.



2.4.5 Composite Transformation

- Composite three dimensional transformations can be formed by multiplying the matrix representation for the individual operations in the transformation sequence.
- This concatenation is carried out from right to left, where the right most matrixes is the first transformation to be applied to an object and the left most matrix is the last transformation.
- A sequence of basic, three-dimensional geometric transformations is combined to produce a single composite transformation which can be applied to the coordinate definition of an object.

2.4.6Three Dimensional Transformation Functions

Some of the basic 3D transformation functions are:

translate (translateVector, matrixTranslate)

rotateX(thetaX, xMatrixRotate)

rotateY(thetaY, yMatrixRotate)

rotateZ(thetaZ, zMatrixRotate)

scale3 (scaleVector, matrixScale)

- Each of these functions produces a 4 by 4 transformation matrix that can be used to transform coordinate positions expressed as homogeneous column vectors.
- Parameter translate Vector is a pointer to list of translation distances tx, ty, and tz.

- Parameter scale vector specifies the three scaling parameters sx, sy and sz.
- Rotate and scale matrices transform objects with respect to the coordinate origin.
- Composite transformation can be constructed with the following functions:

  composeMatrix3

  buildTransformationMatrix3

  composeTransformationMatrix3

- The order of the transformation sequence for the buildTransformationMarix3 and composeTransfomationMarix3 functions, is the same as in 2 dimensions:

  1. scale
  2. rotate
  3. translate

- Once a transformation matrix is specified, the matrix can be applied to specified points with

  transformPoint3 (inPoint, matrix, outpoint)

- The transformations for hierarchical construction can be set using structures with the function

  setLocalTransformation3 (matrix, type)

where parameter matrix specifies the elements of a 4 by 4 transformation matrix and parameter type can be assigned one of the values of:

Preconcatenate,

Postconcatenate, or replace.

2.4.7 Modeling and Coordinate Transformations

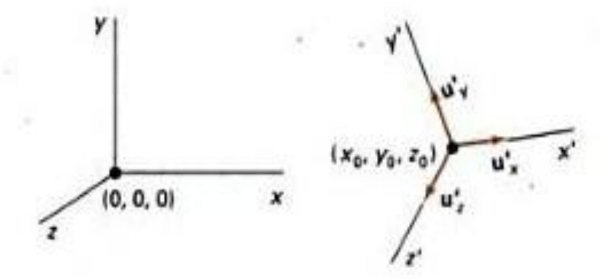
- In modeling, objects are described in a local (modeling) coordinate reference frame, then the objects are repositioned into a world coordinate scene.
- For instance, tables, chairs and other furniture, each defined in a local coordinate system, can be placed into the description of a room defined in another reference frame, by transforming the furniture coordinates to room coordinates. Then the room might be transformed into a larger scene constructed in world coordinate.
- Three dimensional objects and scenes are constructed using structure operations.
- Object description is transformed from modeling coordinate to world coordinate or to another system in the hierarchy.

- Coordinate descriptions of objects are transferred from one system to another system with the same procedures used to obtain two dimensional coordinate transformations.
- Transformation matrix has to be set up to bring the two coordinate systems into alignment:
    - First, a translation is set up to bring the new coordinate origin to the position of the other coordinate origin.
    - Then a sequence of rotations are made to the corresponding coordinate axes.
    - If different scales are used in the two coordinate systems, a scaling transformation may also be necessary to compensate for the differences in coordinate intervals.
- If a second coordinate system is defined with origin (x_0, y_0, z_0) and axis vectors as shown in the figure relative to an existing Cartesian reference frame, then first construct the translation matrix $T(-x_0, -y_0, -z_0)$, then we can use the unit axis vectors to form the coordinate rotation matrix

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

which transforms unit vectors u'_x, u'_y and u'_z onto the x, y and z axes respectively.

Transformation of an object description from one coordinate system to another.



- The complete coordinate-transformation sequence is given by the composite matrix R .T.
- This matrix correctly transforms coordinate descriptions from one Cartesian system to another even if one system is left-handed and the other is right handed.

2.5Three-Dimensional Viewing

In three dimensional graphics applications,

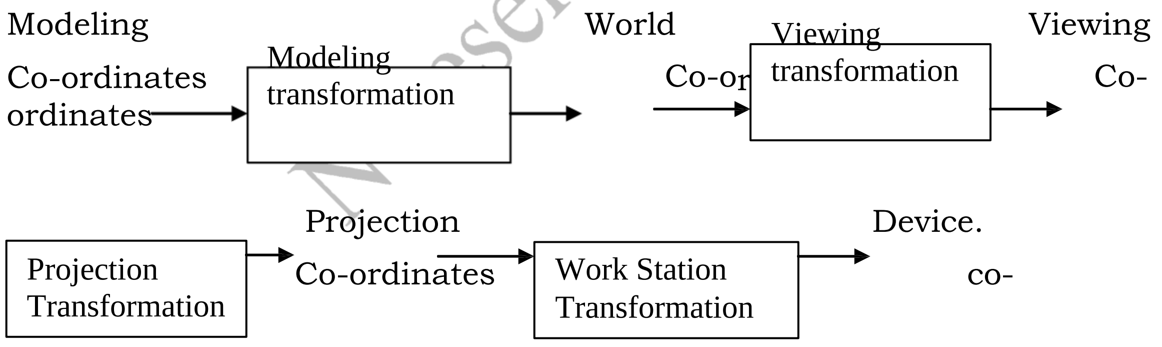
- we can view an object from any spatial position, from the front, from above or from the back.
- We could generate a view of what we could see if we were standing in the middle of a group of objects or inside object, such as a building.

2.5.1Viewing 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.



Processing Steps

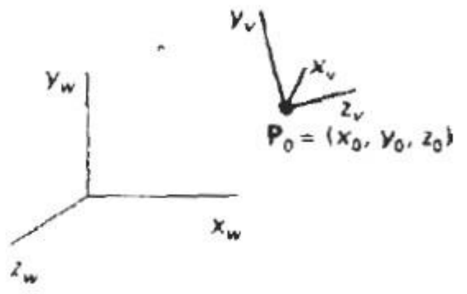
1. Once the scene has been modeled, world coordinates position is converted to viewing coordinates.
2. The viewing coordinates system is used in graphics packages as a reference for specifying the observer viewing position and the position of the projection plane.
3. Projection operations are performed to convert the viewing coordinate description of the scene to coordinate positions on the projection plane, which will then be mapped to the output device.

4. Objects outside the viewing limits are clipped from further consideration, and the remaining objects are processed through visible surface identification and surface rendering procedures to produce the display within the device viewport.

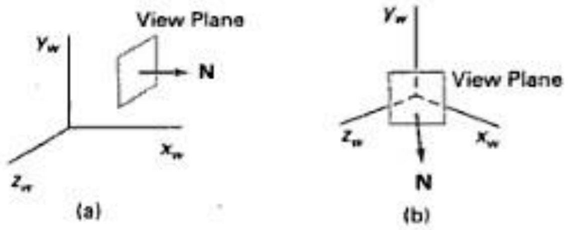
2.5.2Viewing Coordinates

Specifying the view plane

- The view for a scene is chosen by establishing the viewing coordinate system, also called the view reference coordinate system.

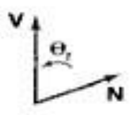


- A viewplane or projection plane is set-up perpendicular to the viewing Z_v axis.
- World coordinate positions in the scene are transformed to viewing coordinates, then viewing coordinates are projected to the view plane.
- The view reference point is a world coordinate position, which is the origin of the viewing coordinate system. It is chosen to be close to or on the surface of some object in a scene.
- Then we select the positive direction for the viewing Z_v axis, and the orientation of the view plane by specifying the view plane normal vector, N. Here the world coordinate position establishes the direction for N relative either to the world origin or to the viewing coordinate origin.



- Then we select the up direction for the view by specifying a vector V called the view-up vector. This vector is used to establish the positive direction for the y_v axis.

Specifying the view –up vector with a twist angle θ_t

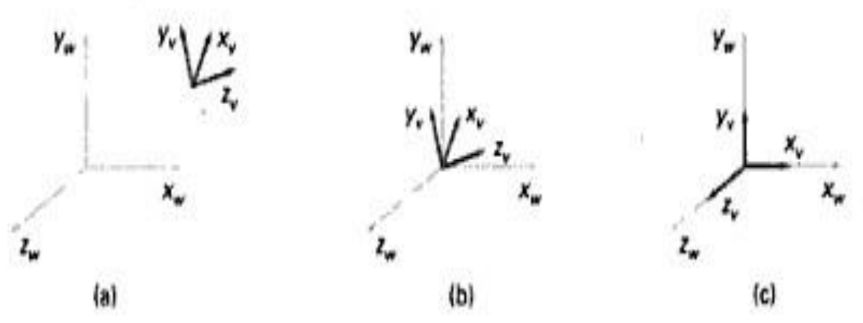


Transformation from world to viewing coordinates

- Before object descriptions can be projected to the view plane, they must be transferred to viewing coordinate. This transformation sequence is,
  1. Translate the view reference point to the origin of the world coordinate system.
  2. Apply rotations to align the x_v, y_v and z_v axes with the world x_w,y_w and z_w axes respectively.
- If the view reference point is specified at world position(x_0,y_0,z_0) this point is translated to the world origin with the matrix transformation.

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

- The rotation sequence can require up to 3 coordinate axis rotations depending on the direction chosen for N. Aligning a viewing system with the world coordinate axes using a sequence of translate – rotate transformations



- Another method for generation the rotation transformation matrix is to calculate unit uvn vectors and form the composite rotation matrix directly.

  Given vectors N and V, these unit vectors are calculated as

$$n = N / (|N|) = (n_1, n_2, n_3)$$

$$u = (V*N) / (|V*N|) = (u_1, u_2, u_3)$$

$$v = n*u = (v_1, v_2, v_3)$$

- This method automatically adjusts the direction for v, so that v is perpendicular to n.

- The composite rotation matrix for the viewing transformation is

$$R = \begin{matrix} u_1 & u_2 & u_3 & 0 \\ v_1 & v_2 & v_3 & 0 \\ n_1 & n_2 & n_3 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

which transforms u into the world x_w axis, v onto the y_w axis and n onto the z_w axis.

- The complete world-to-viewing transformation matrix is obtained as the matrix product. $M_{wc, vc} = R.T$

This transformation is applied to coordinate descriptions of objects in the scene transfer them to the viewing reference frame.

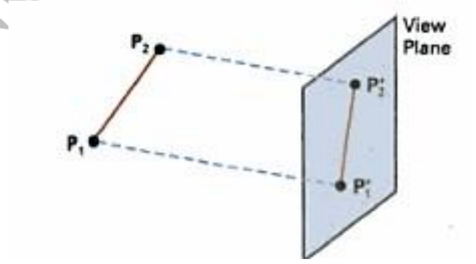
2.5 Projections

- Once world coordinate descriptions of the objects are converted to viewing coordinates, we can project the 3 dimensional objects onto the two dimensional view planes.

- There are two basic types of projection.

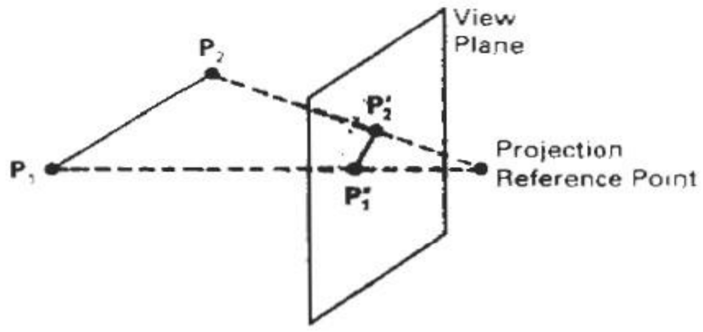
1. Parallel Projection - Here the coordinate positions are transformed to the view plane along parallel lines.

Parallel projection of an object to the view plane



2. Perspective Projection – Here, object positions are transformed to the view plane along lines that converge to a point called the projection reference point.

Perspective projection of an object to the view plane



Parallel Projections

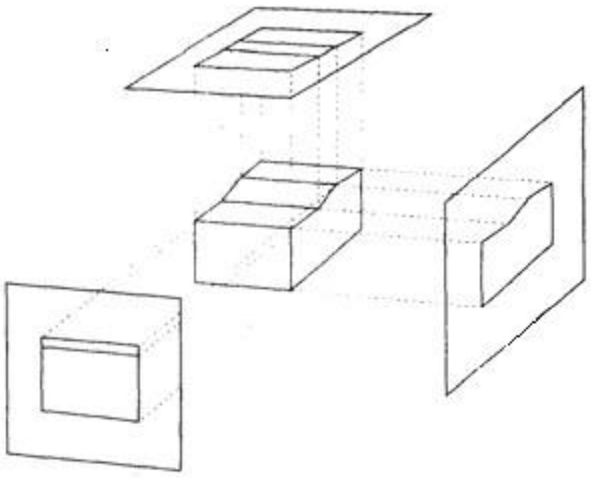
- Parallel projections are specified with a projection vector that defines the direction for the projection lines.
- When the projection in perpendicular to the view plane, it is said to be an Orthographic parallel projection, otherwise it said to be an Oblique parallel projection.

Orientation of the projection vector V_p to produce an orthographic projection (a) and an oblique projection (b)

Orthographic Projection

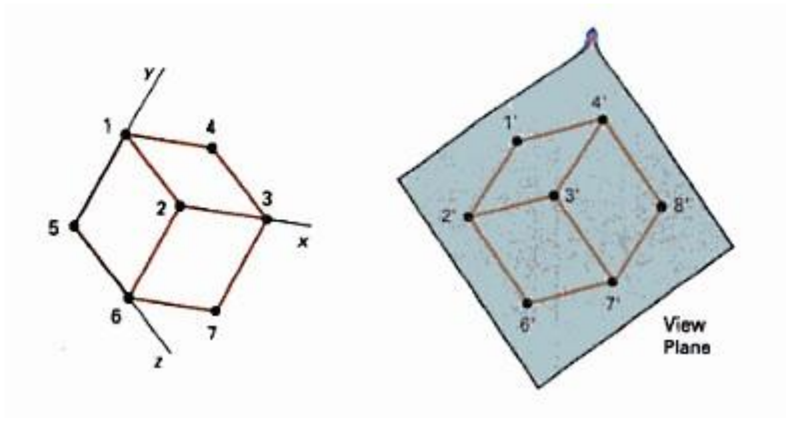
- Orthographic projections are used to produce the front, side and top views of an object.
- Front, side and rear orthographic projections of an object are called elevations.
- A top orthographic projection is called a plan view.
- This projection gives the measurement of lengths and angles accurately.

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.

Isometric projection for a cube



- Transformation equations for an orthographic parallel projection are straight forward.

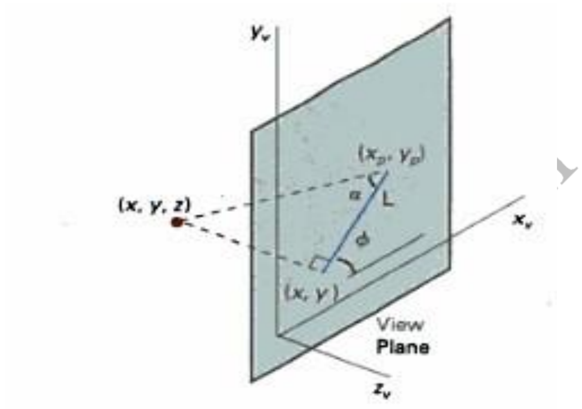
- If the view plane is placed at position z_{vp} along the z_v axis then any point (x,y,z) in viewing coordinates is transformed to projection coordinates as

$$x_p = x, \quad y_p = y$$

where the original *z* coordinates value is kept for the depth information needed in depth cueing and visible surface determination procedures.

Oblique Projection

- An oblique projection in obtained by projecting points along parallel lines that are not perpendicular to the projection plane.
- The below figure α and φ are two angles.



- Point (x,y,z) is projected to position (x_p,y_p) on the view plane.
- The oblique projection line form (x,y,z) to (x_p,y_p) makes an angle α with the line on the projection plane that joins (x_p,y_p) and (x,y).
- This line of length L in at an angle φ with the horizontal direction in the projection plane.
- The projection coordinates are expressed in terms of x,y, L and φ as

$$x_p = x + L\cos\varphi \qquad \text{- - - -(1)}$$
$$y_p = y + L\sin\varphi$$

- Length L depends on the angle α and the z coordinate of the point to be projected:

$$\tan\alpha = z / L$$

thus,

$$L = z / \tan\alpha$$
$$= z L_1$$

where L_1 is the inverse of tanα, which is also the value of L when z = 1.

- The oblique projection equation (1) can be written as

$$x_p = x + z(L_1\cos\varphi)$$
$$y_p = y + z(L_1\sin\varphi)$$

- The transformation matrix for producing any parallel projection onto the $x_v y_v$ plane is

$$M_{parallel} = \begin{array}{cccc} 1 & 0 & L_1\cos\varphi & 0 \\ 0 & 1 & L_1\sin\varphi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{array}$$

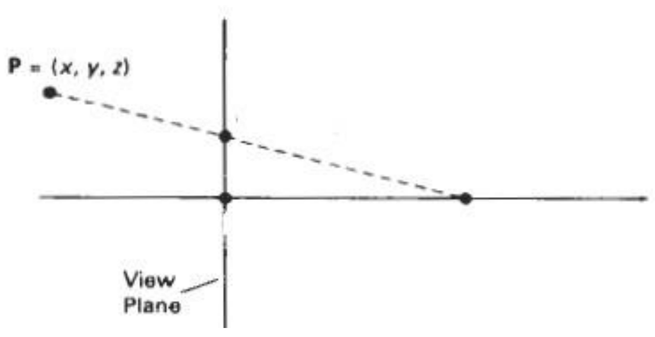
- An orthographic projection is obtained when $L_1 = 0$ (which occurs at a projection angle α of 90^0)
- Oblique projections are generated with non zero values for L_1.

Perspective Projections

- To obtain perspective projection of a 3D object, we transform points along projection lines that meet at the projection reference point.
- If the projection reference point is set at position z_{prp} along the zv axis and the view plane is placed at z_{vp} as in fig , we can write equations describing coordinate positions along this perspective projection line in parametric form as

$$x' = x - x_u$$
$$y' = y - y_u$$
$$z' = z - (z - z_{prp})\,u$$

Perspective projection of a point P with coordinates (x,y,z). to position (x_p, y_p,z_{vp}) on the view plane.



- Parameter u takes values from 0 to 1 and coordinate position (x', y',z') represents any point along the projection line.

- When u = 0, the point is at P = (x,y,z).
- At the other end of the line, u = 1 and the projection reference point coordinates $(0,0,z_{prp})$
- On the view plane $z` = z_{vp}$ and $z`$ can be solved for parameter u at this position along the projection line:
- Substituting this value of u into the equations for x` and y`, we obtain the perspective transformation equations.

$$x_p = x((z_{prp} - z_{vp}) / (z_{prp} - z)) = x(d_p/(z_{prp} - z))$$
$$y_p = y((z_{prp} - z_{vp}) / (z_{prp} - z)) = y(d_p / (z_{prp} - z)) \text{ --------------(2)}$$

where $d_p = z_{prp} - z_{vp}$ is the distance of the view plane from the projection reference point.

- Using a 3D homogeneous coordinate representation we can write the perspective projection transformation (2) in matrix form as

$$\begin{array}{c} x_h \\ y_h \\ z_h \\ h \end{array} = \begin{array}{cccc} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -(z_{vp}/d_p) & z_{vp}(z_{prp}/d_p) \\ 0 & 0 & -1/d_p & z_{prp}/d_p \end{array} \begin{array}{c} x \\ y \\ z \\ 1 \end{array} \text{ --------(3)}$$

- In this representation, the homogeneous factor is

$$h = (z_{prp}-z)/d_p \text{ --------------(4)}$$

and the projection coordinates on the view plane are calculated from eq (2)the homogeneous coordinates as

$$x_p = x_h / h$$
$$y_p = y_h / h \text{ ---------------------(5)}$$

where the original z coordinate value retains in projection coordinates for depth processing.

2.6 CLIPPING

- An algorithm for three-dimensional clipping identifies and saves all surface segments within the view volume for display on the output device. All parts of objects that are outside the view volume are discarded.
- Instead of clipping against straight-line window boundaries, we now clip objects against the boundary planes of the view volume.
- To clip a line segment against the view volume, we would need to test the relative position of the line using the view volume's boundary plane equations. By substituting the line endpoint coordinates into the plane equation of each boundary in turn, we

could determine whether the endpoint is inside or outside that boundary.

- An endpoint (x, y, *z)* of a line segment is outside a boundary plane if $Ax + By + Cz + D > 0$, where *A, B* , C, and *D* are the plane parameters for that boundary.
- Similarly, the point is inside the boundary if $Ax + By + Cz + D < 0$. Lines with both endpoints outside a boundary plane are discarded, and those with both endpoints inside all boundary planes are saved.
- The intersection of a line with a boundary is found using the line equations along with the plane equation.
- Intersection coordinates (x_1, y_1, z_1) are values that are on the line and that satisfy the plane equation $Ax_1, + By_1 + Cz_1 + D = 0$.
- To clip a polygon surface, we can clip the individual polygon edges. First, we could test the coordinate extents against each boundary of the view volume to determine whether the object is completely inside or completely outside that boundary. If the coordinate extents of the object are inside all boundaries, we save it. If the coordinate extents are outside all boundaries, we discard it. Otherwise, we need to apply the intersection calculations.

Viewport Clipping

- Lines and polygon surfaces in a scene can be clipped against the viewport boundaries with procedures similar to those used for two dimensions, except that objects are now processed against clipping planes instead of clipping edges.
- The two-dimensional concept of region codes can be extended to three dimensions by considering positions in front and in back of the three-dimensional viewport, as well as positions that are left, right, below, or above the volume. For three dimensionalpoints, we need to expand the region code to six bits. Each point in the description of a scene is then assigned a six-bit region code that identifies the relative position of the point with respect to the viewport.
- For a line endpoint at position (x, y, z), we assign the bit positions in the region code from right to left as

  bit 1 = 1, if $x < xv_{min}$(left)

  bit 2 = 1, if $x > xv_{max}$(right)

  bit 3 = 1, if $y < yv_{min}$(below)

  bit 4 = 1, if $y > yv_{max}$(above)

  bit 5 = 1, if $z < zv_{min}$(front)

  bit 6 = 1, if $z > zv_{max}$(back)

- For example, a region code of 101000 identifies a point as above and behind the viewport, and the region code 000000 indicates a point within the volume.
- A line segment can immediately identified as completely within the viewport if both endpoints have a region code of 000000. If either endpoint of a line segment does not have a region code of 000000, we perform the logical and operation on the two endpoint codes. The result of this and operation will be nonzero for any line segment that has both endpoints in one of the six outside regions.
- As in two-dimensional line clipping, we use the calculated intersection of a line with a viewport plane to determine how much of the line can be thrown away.
- The two-dimensional parametric clipping methods of Cyrus-Beck or Liang-Barsky can be extended to three-dimensional scenes. For a line segment with endpoints $P_1 = (x_1, y_1, z_1,)$ and $P_2 = (x_2, y_2, z_2)$, we can write the parametric line equations as

$$x = x_1 + (x_2 - x_1)u \qquad 0<=u<=1$$

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

$$z = z_1 + (z_2 - z_1)u \qquad \text{-------------}(1)$$

- Coordinates (x, y, z) represent any point on the line between the two endpoints.
- At u = 0, we have the point PI, and u = 1 puts us at P2.
- To find the intersection of a line with a plane of the viewport, we substitute the coordinate value for that plane into the appropriate parametric expression of Eq.1 and solve for u. For instance, suppose we are testing a line against the zv_{min}, plane of the viewport. Then

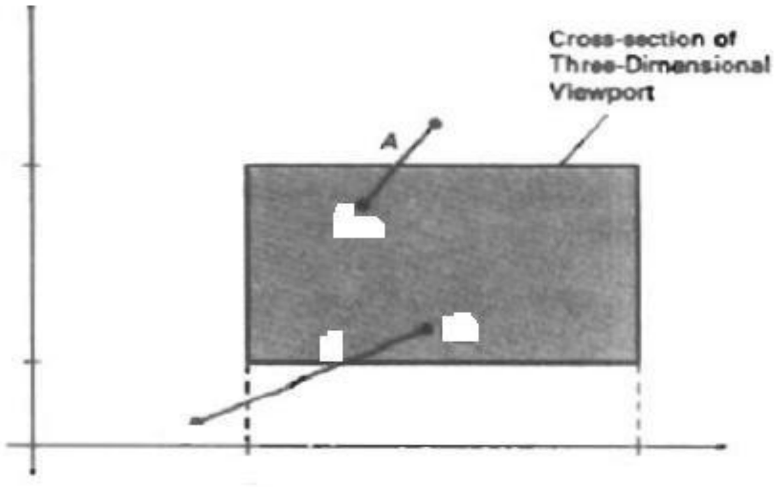
$$u = \frac{zv_{min} - z_1}{z_2 - z_1} \qquad \text{---------------------------}(2)$$

- When the calculated value for u is not in the range from 0 to 1, the line segment does not intersect the plane under consideration at any point between endpoints P1 and P2 (line A in fig).
- If the calculated value for *u* in Eq.2 is in the interval from 0 to 1, we calculate the intersection's x and y coordinates as

$$x_1 = x_1 + (x_2 - x_1)\left(\frac{zv_{min} - z_1}{z_2 - z_1}\right)$$

$$y_1 = y_1 + (y_2 - y_1)\left(\frac{zv_{min} - z_1}{z_2 - z_1}\right)$$

- If either x_1 or y_1 is not in the range of the boundaries of the viewport, then this line intersects the front plane beyond the boundaries of the volume (line *B* in Fig.)



2.7 Three Dimensional Viewing Functions

1. With parameters specified in world coordinates, elements of the matrix for transforming world coordinate descriptions to the viewing reference frame are calculated using the function.

   EvaluateViewOrientationMatrix3(x_0,y_0,z_0,xN,yN,zN,xV,yV,zV,error,viewMatrix)

   - This function creates the viewMatrix from input coordinates defining the viewing system.
   - Parameters x_0,y_0,z_0 specify the sign of the viewing system.
   - World coordinate vector (xN, yN, zN) defines the normal to the view plane and the direction of the positive z_v viewing axis.
   - The world coordinates (xV, yV, zV) gives the elements of the view up vector.
   - An integer error code is generated in parameter error if input values are not specified correctly.

2. The matrix proj matrix for transforming viewing coordinates to normalized projection coordinates is created with the function.

   EvaluateViewMappingMatrix3 (xwmin,xwmax,ywmin,ywmax,xvmin,xvmax,yvmin,yvmax,zvmin,zvmax, projType,xprojRef,yprojRef,zprojRef,zview,zback,zfront,error,projMatrix)

- Window limits on the view plane are given in viewing coordinates with parameters xwmin, xwmax, ywmin and ywmax.
- Limits of the 3D view port within the unit cube are set with normalized coordinates xvmin, xvmax, yvmin, yvmax, zvmin and zvmax.
- Parameter projType is used to choose the projection type either parallel or perspective.
- Coordinate position (xprojRef, yprojRdf, zprojRef) sets the projection reference point. This point is used as the center of projection if projType is set to perspective; otherwise, this point and the center of the viewplane window define the parallel projection vector.
- The position of the viewplane along the viewing z_v axis is set with parameter z view.
- Positions along the viewing z_v axis for the front and back planes of the view volume are given with parameters z front and z back.
- The error parameter returns an integer error code indicating erroneous input data.

2.8VISIBLE SURFACE IDENTIFICATION

A major consideration in the generation of realistic graphics displays is identifying those parts of a scene that are visible from a chosen viewing position.

2.8.1 Classification of Visible Surface Detection Algorithms

These are classified into two types based on whether they deal with object definitions directly or with their projected images

1. Object space methods: compares objects and parts of objects to each other within the scene definition to determine which surfaces as a whole we should label as visible.
2. Image space methods: visibility is decided point by point at each pixel position on the projection plane. Most Visible Surface Detection Algorithms use image space methods.

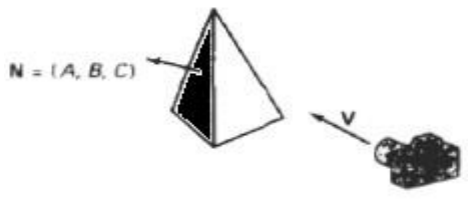
2.8.2 Back Face Detection

A point (x, y,z) is "inside" a polygon surface with plane parameters A, B, C, and D if

$$A_x + B_y + C_z + D < 0 \qquad \text{----------------}(1\ )$$

When an inside point is along the line of sight to the surface, the polygon must be a back face .

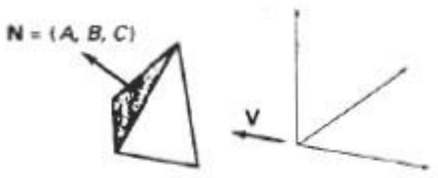
We can simplify this test by considering the normal vector N to a polygon surface, which has Cartesian components *(A, B, C).* In general, if V is a vector in the viewing direction from the eye position, as shown in Fig.,



then this polygon is a back face if V . N > 0

Furthermore, if object descriptions have been converted to projection coordinates and our viewing direction is parallel to the viewing z_v. axis, then V = (0, 0, V_z) and V . N = V_zC so that we only need to consider the sign of C, the ; component of the normal vector N.

In a right-handed viewing system with viewing direction along the negative z_v axis in the below Fig. the polygon is a back face if C < 0.



Thus, in general, we can label any polygon as a back face if its normal vector has a z component value

$C <= 0$

By examining parameter C for the different planes defining an object, we can immediately identify all the back faces.

2.8.3 Depth Buffer Method

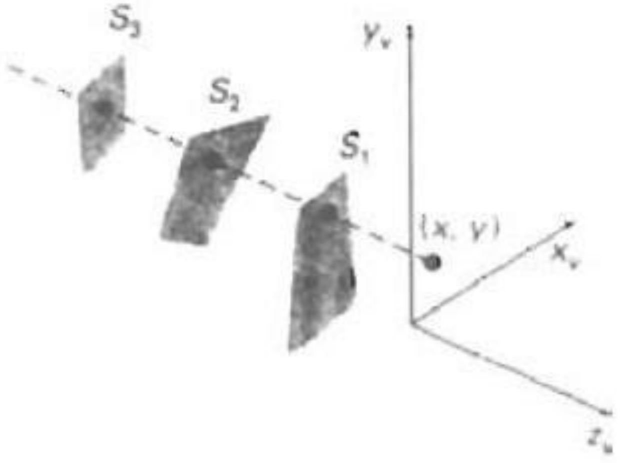
A commonly used image-space approach to detecting visible surfaces is the depth-buffer method, which compares surface depths at each pixel position on the projection plane. This procedure is also referred to as the z-buffer method.

Each surface of a scene is processed separately, one point at a time across the surface. The method is usually applied to scenes containing only polygon surfaces, because depth values can be computed very quickly and the method is easy to implement. But the mcthod can be applied to nonplanar surfaces.

With object descriptions converted to projection coordinates, each (x, y, z) position on a polygon surface corresponds to the orthographic projection point (x, y) on the view plane.

Therefore, for each pixel position (x, y) on the view plane, object depths can be compared by comparing z values. The figure shows

three surfaces at varying distances along the orthographic projection line from position (x,y) in a view plane taken as the (x_v,y_v) plane. Surface S_1, is closest at this position, so its surface intensity value at (x, y) is saved.



We can implement the depth-buffer algorithm in normalized coordinates, so that *z* values range from 0 at the back clipping plane to Z_{max} at the front clipping plane.

Two buffer areas are required.A depth buffer is used to store depth values for each (x, y) position as surfaces are processed, and the refresh buffer stores the intensity values for each position.

Initially,all positions in the depth buffer are set to 0 (minimum depth), and the refresh buffer is initialized to the background intensity.

We summarize the steps of a depth-buffer algorithm as follows:

1. Initialize the depth buffer and refresh buffer so that for all buffer positions (x, y),

$$\text{depth}(x, y)=0, \qquad \text{refresh}(x, y)=I_{backgnd}$$

2. For each position on each polygon surface, compare depth values to previously stored values in the depth buffer to determine visibility.

- Calculate the depth z for each (x, y) position on the polygon.
- If z > depth(x, y), then set

  depth (x, y)=z , refresh(x,y)= I_{surf}(x, y)

where $I_{backgnd}$ is the value for the background intensity, and I_{surf}(x, y) is the projected intensity value for the surface at pixel position (x,y). After all surfaces have been processed, the depth buffer contains depth values for the visible surfaces and the refresh buffer contains the corresponding intensity values for those surfaces.

Depth values for a surface position (x, y) are calculated from the plane equation for each surface:

$$z = \frac{-Ax - By - D}{C} \qquad \text{-----------------------------(1)}$$

For any scan line adjacent horizontal positions across the line differ by1, and a vertical y value on an adjacent scan line differs by 1. If the depth of position(x, y) has been determined to be z, then the depth z' of the next position (x +1, y) along the scan line is obtained from Eq. (1) as

$$z' = \frac{-A(x+1) - By - D}{C} \qquad \text{-----------------------(2)}$$

Or $$z' = z - \frac{A}{C} \qquad \text{-----------------------(3)}$$

On each scan line, we start by calculating the depth on a left edge of the polygon that intersects that scan line in the below fig. Depth values at each successive position across the scan line are then calculated by Eq. (3).

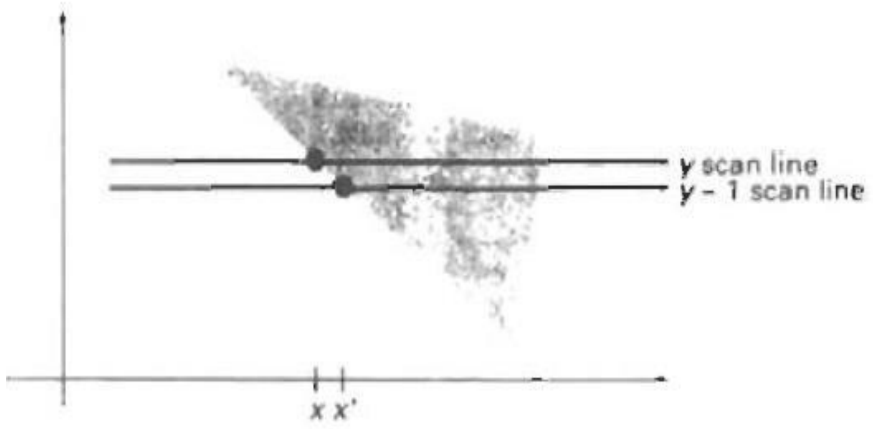
Scan lines intersecting a polygon surface

We first determine the y-coordinate extents of each polygon, and process the surface from the topmost scan line to the bottom scan line. Starting at a top vertex, we can recursively calculate x positions down a left edge of the polygon as x' = x - l/m, where m is the slope of the edge.

Depth values down the edge are then obtained recursively as

$$z' = z + \frac{A/m + B}{C} \qquad \text{-----------------------(4)}$$

Intersection positions on successive scan lines along a left polygon edge



If we are processing down a vertical edge, the slope is infinite and the recursive calculations reduce to

$$z' = z + \frac{B}{C} \qquad \text{-----------------------(5)}$$

An alternate approach is to use a midpoint method or Bresenham-type algorithm for determining x values on left edges for each scan line. Also the method can be applied to curved surfaces by determining depth and intensity values at each surface projection point.

For polygon surfaces, the depth-buffer method is very easy to implement, and it requires no sorting of the surfaces in a scene. But it does require the availability of a second buffer in addition to the refresh buffer.

2.8.4 A- BUFFER METHOD

An extension of the ideas in the depth-buffer method is the A-buffer method. The A buffer method represents an antialiased, area-averaged, accumulation-buffer method developed by Lucasfilm for implementation in the surface-rendering system called REYES (an acronym for "Renders Everything You Ever Saw").

A drawback of the depth-buffer method is that it can only find one visible surface at each pixel position. The A-buffer method expands the depth buffer so that each position in the buffer can reference a linked list of surfaces.

Thus, more than one surface intensity can be taken into consideration at each pixel position, and object edges can be antialiased.

Each position in the A-buffer has two fields:

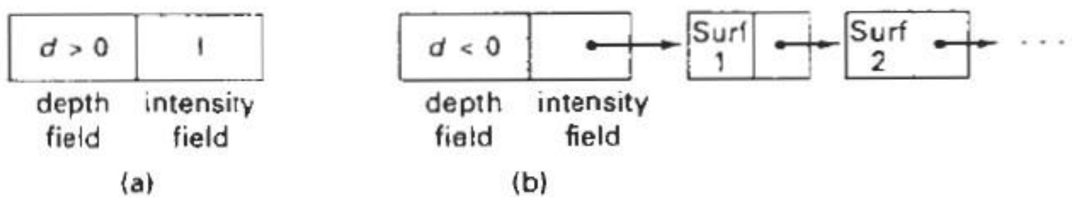
1)depth field - stores a positive or negative real number

2)intensity field - stores surface-intensity information or a pointer value.

If the depth field is positive, the number stored at that position is the depth of a single surface overlapping the corresponding pixel area. The intensity field then stores the RCB components of the surface color at that point and the percent of pixel coverage, as illustrated in Fig.A

If the depth field is negative, this indicates multiple-surface contributions to the pixel intensity. The intensity field then stores a pointer to a linked Iist of surface data, as in Fig. B.

Organization of an A-buffer pixel position (A) single surface overlap of the corresponding pixel area (B) multiple surface overlap



Data for each surface in the linked list includes

- RGB intensity components
- opacity parameter (percent of transparency)
- depth
- percent of area coverage
- surface identifier
- other surface-rendering parameters
- pointer to next surface

.

2.8.5 SCAN-LINE METHOD

This image-space method for removing hidden surfaces is an extension of the scan-line algorithm for filling polygon interiors. As each scan line is processed, all polygon surfaces intersecting that line are examined to determine which are visible. Across each scan line, depth calculations are made for each overlapping surface to determine which is nearest to the view plane. When the visible surface has been determined, the intensity value for that position is entered into the refresh buffer.

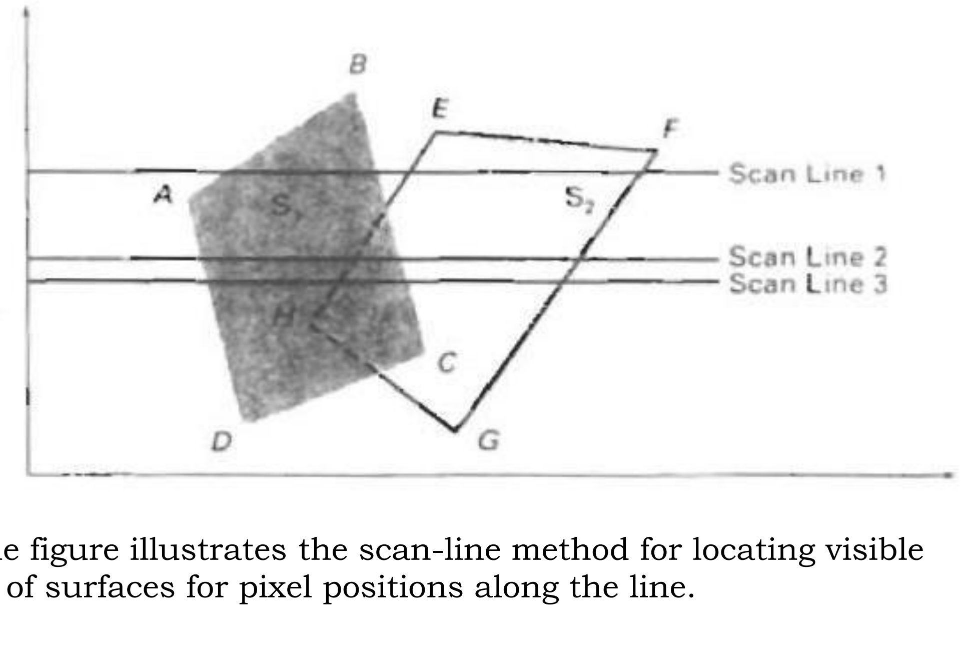
We assume that tables are set up for the various surfaces, which include both an edge table and a polygon table. The edge table contains coordinate endpoints for each line in-the scene, the inverse slope of each line, and pointers into the polygon table to identify the surfaces bounded by each line.

The polygon table contains coefficients of the plane equation for each surface, intensity information for the surfaces, and possibly pointers into the edge table.

To facilitate the search for surfaces crossing a given scan line, we can set up an active list of edges from information in the edge table. This active list will contain only edges that cross the current scan line, sorted in order of increasing x.

In addition, we define a flag for each surface that is set on or off to indicate whether a position along a scan line is inside or outside of the surface. Scan lines are processed from left to right. At the leftmost boundary of a surface, the surface flag is turned on; and at the rightmost boundary, it is turned off.

Scan lines crossing the projection of two surfaces S_1 and S_2 in the view plane. Dashed lines indicate the boundaries of hidden surfaces



The figure illustrates the scan-line method for locating visible portions of surfaces for pixel positions along the line.

The active list for scan line 1 contains information from the edge table for edges AB, BC, *EH,* and *FG.* For positions along this scan line between edges AB and BC, only the flag for surface S_1 is on.

Therefore no depth calculations are necessary, and intensity information for surface S_1, is entered from the polygon table into the refresh buffer.

Similarly, between edges EH and FG, only the flag for surface S2 is on. No other positions along scan line 1 intersect surfaces, so the intensity values in the other areas are set to the background intensity.

For scan lines 2 and 3 , the active edge list contains edges AD,

EH, BC, and FG. Along scan line 2 from edge AD to edge EH, only the flag for surface S_1, is on. But between edges EH and BC, the flags for both surfaces are on.

In this interval, depth calculations must be made using the plane coefficients for the two surfaces. For this example, the depth of surface S_1 is assumed to be less than that of S_2, so intensities for surface S_1, are loaded into the refresh buffer until boundary BC is encountered. Then the flag for surface S_1 goes off, and intensities for surface S_2 are stored until edge FG is passed.

Any number of overlapping polygon surfaces can be processed with this scan-line method. Flags for the surfaces are set to indicate whether a position is inside or outside, and depth calculations are performed when surfaces overlap.

2.8.6 Depth Sorting Method

Using both image-space and object-space operations, the depth-sorting method performs the following basic functions:

1. Surfaces are sorted in order of decreasing depth.

2. Surfaces are scan converted in order, starting with the surface of greatest depth.

Sorting operations are carried out in both image and object space, and the scan conversion of the polygon surfaces is performed in image space.

This method for solving the hidden-surface problem is often referred to as the painter's algorithm. In creating an oil painting, an artist first paints the background colors. Next, the most distant objects are added, then the nearer objects, and so forth. At the final step, the foreground objects are painted on the canvas over the background and other objects that have been painted on the canvas. Each layer of paint covers up the previous layer.

Using a similar technique, we first sort surfaces according to their distance from the view plane. The intensity values for the farthest surface are then entered into the refresh buffer. Taking each succeeding surface in turn we "paint" the surface intensities onto the frame buffer over the intensities of the previously processed surfaces.

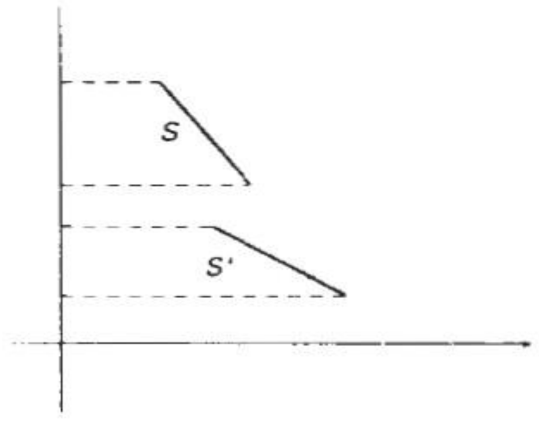
Painting polygon surfaces onto the frame buffer according to depth is carried out in several steps. Assuming we are viewing along the-*z* direction,

1.surfaces are ordered on the first pass according to the smallest *z* value on each surface.

2.Surfaces with the greatest depth is then compared to the other surfaces in the list to determine whether there are any overlaps in depth. If no depth overlaps occur, S is scan converted. Figure shows two surfaces that overlap in the xy plane but have no depth overlap.

3.This process is then repeated for the next surface in the list. As long as no overlaps occur, each surface is processed in depth order until all have been scan converted.

4. If a depth overlap is detected at any point in the list, we need to make some additional comparisons to determine whether any of the surfaces should be reordered.Two surfaces with no depth overlap



We make the following tests for each surface that overlaps with S. If any one of these tests is true, no reordering is necessary for that surface. The tests are listed in order of increasing difficulty.

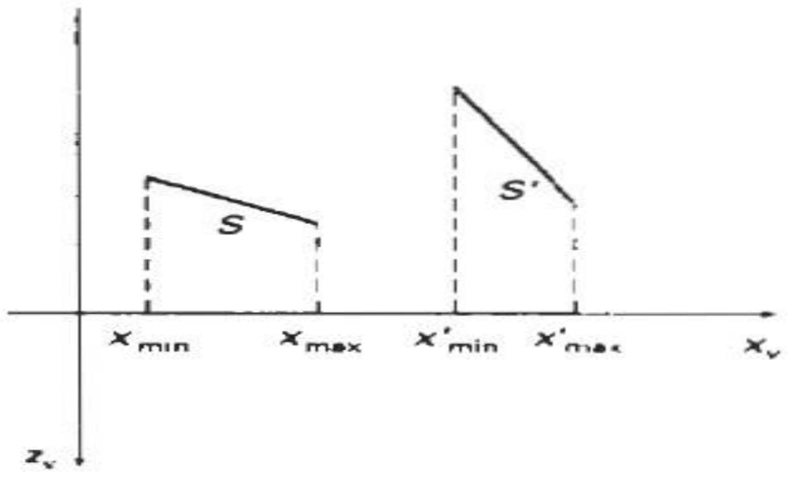
1. The bounding rectangles in the xy plane for the two surfaces do not overlap

2. Surface S is completely behind the overlapping surface relative to the viewing position.

3. The overlapping surface is completelv in front of S relative to the viewing position.

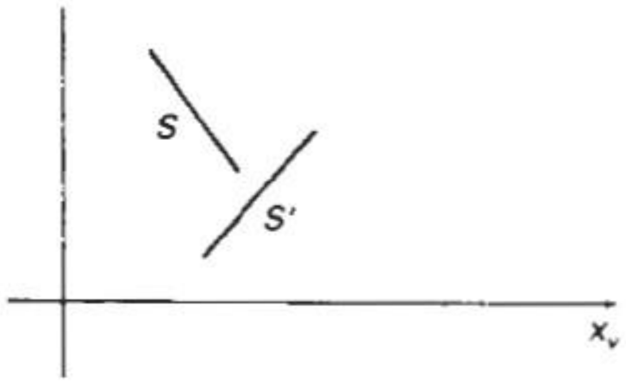
4. The projections of the two surfaces onto the view plane do not overlap.

Test 1 is performed in two parts. We first check for overlap in the x direction,then we check for overlap in the y direction. If either of these directions show no overlap, the two planes cannot obscure one other. An example of two surfaces that overlap in the *z* direction but not in the *x* direction is shown in Fig.



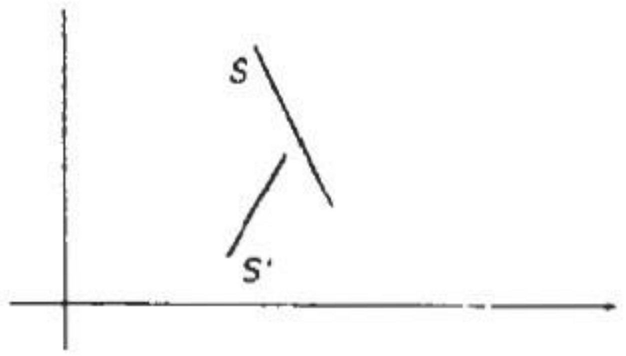
We can perform tests 2 and 3 with an "inside-outside" polygon test. That is,we substitute the coordinates for all vertices of S into the plane equation for the overlapping surface and check the sign of the result. If the plane equations are setup so that the outside of the surface is toward the viewing position, then *S* is behind S' if all vertices of *S* are "inside" S'

Surface S is completely behind (inside) the overlapping surface S'



Similarly, S' is completely in front of S if all vertices of S are "outside" of *S'*. Figure shows an overlapping surface S' that is completely in front of S, but surface S is not completely inside S'.

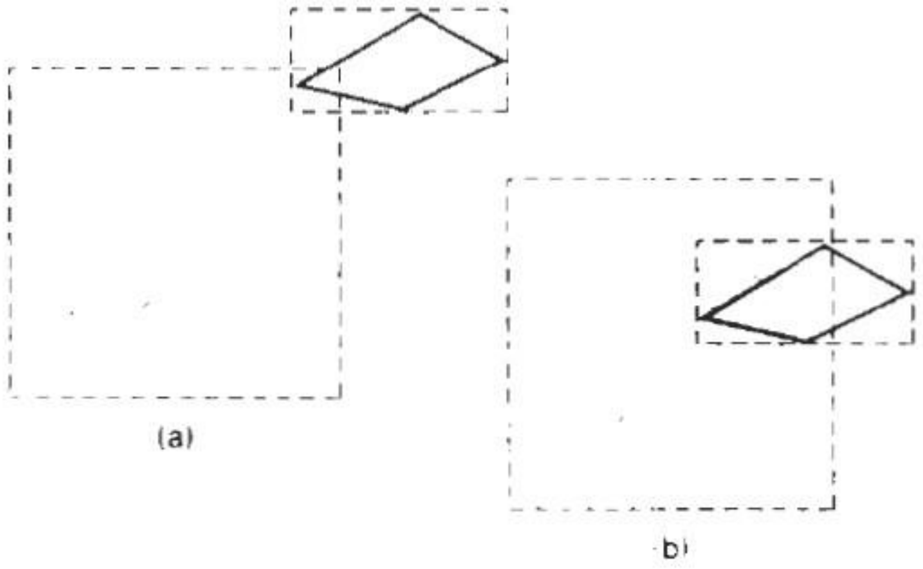
Overlapping surface S' is completely in front(outside) of surface **S but s is not completely behind S'**



If tests 1 through 3 have all failed, we try test 4 by checking for intersections between the bounding edges of the two surfaces using line equations in the xy plane. As demonstrated in Fig., two surfaces may or may not intersect even though their coordinate extents overlap in the x, y, and z directions.

Should all four tests fail with a particular overlapping surface S', we interchange surfaces S and S' in the sorted list.

Two surfaces with overlapping bounding rectangles in the xy plane

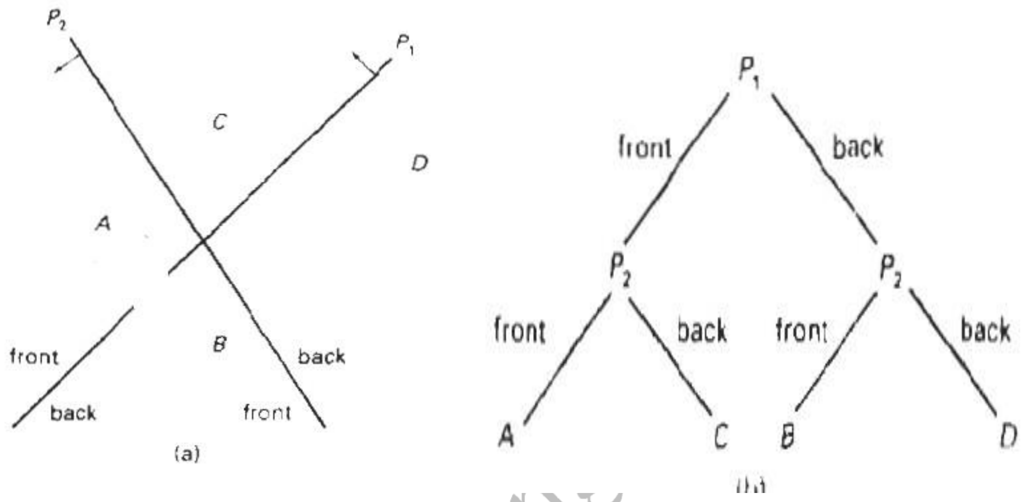


2.8.7 BSP-Tree Method

A binary space-partitioning (BSP) tree is an efficient method for determining object visibility by painting surfaces onto the screen from back to front, as in the painter's algorithm. The BSP tree is particularly useful when the view reference point changes, but the objects in a scene are at fixed positions.

Applying a BSP tree to visibility testing involves identifying surfaces that are "inside" and "outside" the partitioning plane at each step of the space subdivision, relative to the viewing direction. The figure(a) illustrates the basic concept in this algorithm.

A region of space (a) is partitioned with two planes P_1 and P_2 to form the BSP tree representation in (b)



With plane P_1,we first partition the space into two sets of objects. One set of objects is behind, or in back of, plane P_1, relative to the viewing direction, and the other set is in front of P_1. Since one object is intersected by plane P_1, we divide that object into two separate objects, labeled A and B.

Objects A and C are in front of P_1 and objects B and D are behind P_1. We next partition the space again with plane P_2 and construct the binary tree representation shown in Fig.(b).

In this tree, the objects are represented as terminal nodes, with front objects as left branches and back objects as right branches.

2.8.8 Area – Subdivision Method

This technique for hidden-surface removal is essentially an image-space method ,but object-space operations can be used to accomplish depth ordering of surfaces.

The area-subdivision method takes advantage of area coherence in a scene by locating those view areas that represent part of a single surface. We apply this method by successively dividing the total viewing area into smaller and smaller rectangles until each small area is the projection of part of a single visible surface or no surface at all.

To implement this method, we need to establish tests that can quickly identify the area as part of a single surface or tell us that the area is too complex to analyze easily. Starting with the total view, we apply the tests to determine whether we should subdivide the total area into smaller rectangles. If the tests indicate that the view is sufficiently complex, we subdivide it. Next. we apply the tests to each of the smaller areas, subdividing these if the tests indicate that visibility of a single surface is still uncertain. We continue this process until the subdivisions are easily analyzed as belonging to a single surface or until they are reduced to the size of a single pixel. An easy way to do this is to successively divide the area into four equal parts at each step.

Tests to determine the visibility of a single surface within a specified area are made by comparing surfaces to the boundary of the area. There are four possible relationships that a surface can have with a specified area boundary. We can describe these relative surface characteristics in the following way (Fig.):

- Surrounding surface-One that completely encloses the area.
- Overlapping surface-One that is partly inside and partly outside the area.
- Inside surface-One that is completely inside the area.
- Outside surface-One that is completely outside the area.

Possible relationships between polygon surfaces and a rectangular area

The tests for determining surface visibility within an area can be stated in terms of these four classifications. No further subdivisions of a specified area are needed if one of the following conditions is true:

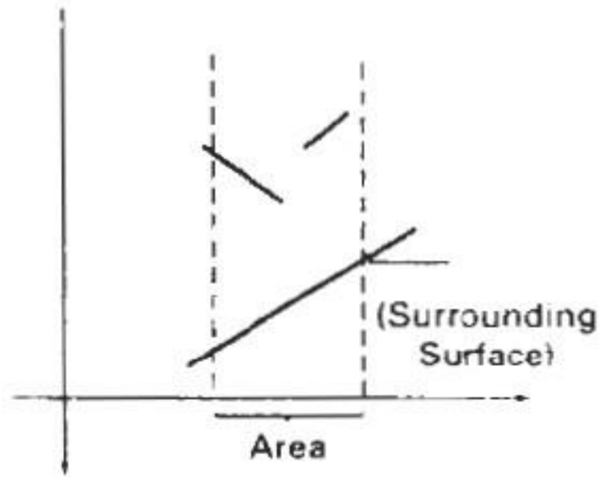
1. All surfaces are outside surfaces with respect to the area.
2. Only one inside, overlapping, or surrounding surface is in the area.
3. A surrounding surface obscures all other surfaces within the area boundaries.

Test 1 can be carrieded out by checking the bounding rectangles of all surfaces against the area boundaries.

Test 2 can also use the bounding rectangles in the xy plane to identify an inside surface

One method for implementing test 3 is to order surfaces according to their minimum depth from the view plane. For each surrounding surface, we then compute the maximum depth within the area under consideration. If the maximum depth of one of these surrounding surfaces is closer to the view plane than the minimum depth of all other surfaces within the area, test 3 is satisfied.

Within a specified area a surrounding surface with a maximum depth of Z_{max} obscures all surfaces that have a minimum depth beyond Z_{max}



Another method for carrying out test 3 that does not require depth sorting is to use plane equations to calculate depth values at the four vertices of the area for all surrounding, overlapping, and inside surfaces, If the calculated depths for one of the surrounding surfaces is less than the calculated depths for all other surfaces, test 3 is true. Then the area can be filled with the intensity values of thesurrounding surface.

For some situations, both methods of implementing test 3 will fail to identify correctly a surrounding surface that obscures all the other surfaces. It is faster to subdivide the area than to continue with more complex testing.

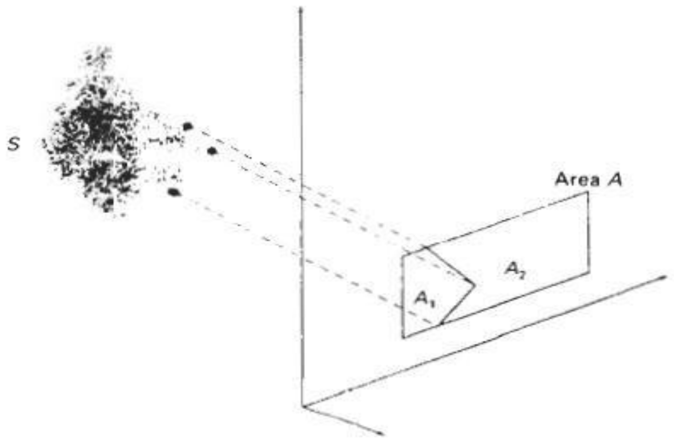
Once outside and surrounding surfaces have been identified for an area, they will remain outside and surrounding surfaces for all subdivisions of the area. Furthermore, some inside and overlapping surfaces can be expected to be eliminated as the subdivision process continues, so that the areas become easier to analyze.

In the limiting case, when a subdivision the size of a pixel is produced, we simply calculate the depth of each relevant surface at that point and transfer the intensity of the nearest surface to the frame buffer.

As a variation on the basic subdivision process, we could subdivide areas along surface boundaries instead of dividing them in half. The below Figure illustrates this method for subdividing areas. The projection of the boundary of surface S is used to partition the original area into the subdivisions A_1 and A2. Surface S is then a surrounding surface for A_1, and visibility tests 2 and 3 can be applied to determine whether further subdividing is necessary.

In general, fewer subdivisions are required using this approach, but more processing is needed to subdivide areas and to analyze the relation of surfaces to the subdivision boundaries.

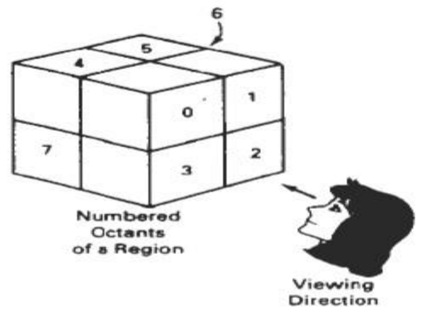
Area A is subdivided into A_1 and A_2, using the boundary of surface *S* on the view plane.



2.8.9 OCTREE METHODS

When an octree representation is used for the viewing volume, hidden-surface elimination is accomplished by projecting octree nodes onto the viewing surface in a front-to-back order.

In the below Fig. the front face of a region of space (the side toward the viewer) is formed with octants 0, 1, 2, and 3. Surfaces in the front of these octants are visible to the viewer. Any surfaces toward the re in the back octants (4,5,6, and 7) may be hidden by the front surfaces.



Back surfaces are eliminated, for the viewing directionby processing data elements in the octree nodes in the order 0, 1, 2,3,4, 5, 6, 7.

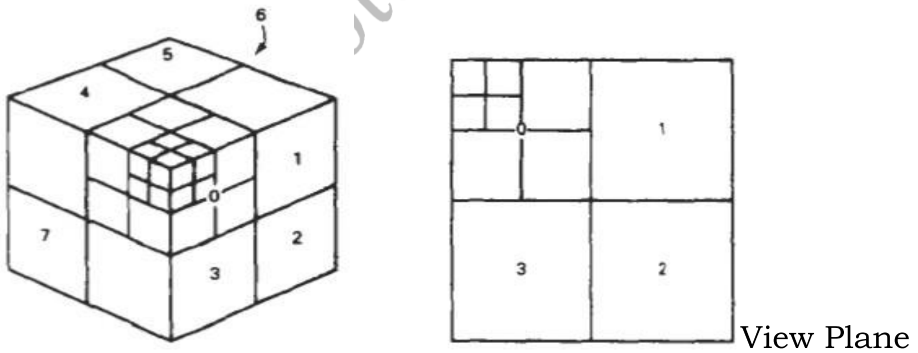
This results in a depth-first traversal of the octree, so that nodes representing octants 0, 1.2, and 3 for the entire region are visited before the nodes representing octants 4,5,6, and 7.

Similarly, the nodes for the front four suboctants of octant 0 are visited before the nodes for the four back suboctants. The traversal of the octree continues in this order for each octant subdivision.

When a color value is encountered in an octree node, the pixel area in the frame buffer corresponding to this node is assigned that color value only if no values have previously been stored in this area. In this way, only the front colors are loaded into the buffer. Nothing is loaded if an area is void. Any node that is found to be completely obscured is eliminated from further processing, so that its subtrees are not accessed.

Different views of objects represented as octrees can be obtained by applying transformations to the octree representation that reorient the object according to the view selected.

A method for displaying an octree is first to map the octree onto a quadtree of visible areas by traversing octree nodes from front to back in a recursive procedure. Then the quadtree representation for the visible surfaces is loaded into the frame buffer. The below Figure depicts the octants in a region of space and the corresponding quadrants on the view plane.



Contributions to quadrant 0 come from octants 0 and 4. Color values in quadrant 1 are obtained from surfaces in octants1 and 5, and values in each of the other two quadrants are generated from the pair of octants aligned with each of these quadrants.

In most cases, both a front and a back octant must be considered in determining the correct color values for a quadrant. But if the front octant is homogeneously filled with some color, we do not process the back octant. If the front is empty the, the rear octant is processed. Otherwise, two ,.recursive calls are made, one for the rear octant and one for the front octant.

```
typedef enum ( SOLID, MIXED } Status;
bdefine EMPTY -1
typedef struct tOctree (
int id;
Status status;
union (
int color;
struct tOctree  *children[8]:
) data;
}Octree:
typedef struct tQuadtree i
int id:
Status status;
union [
int color;
struct tQuadtree *children[4];
) data;
) Quadtree;
int nQuadtree = 0.
void octreeToQuadtree (Octree  *oTree. Quadtree *qTree)
(
Octree  *front. *back:
Quadtree  *newQuadtree;
int i, j;
if (oTree->status == SOLID) (
qTree->status = SOLID:
qTree->data.color = oTree->data color:
return:
)
qTree->status = MIXED:
/*Fill in each quad of the quadtree *I
for ( i = O ; i<4; i++)
 {
front = oTree->data.children[il;
back = oTree->data..children[i+4];
newQuadtree = (Quadtree *) malloc (sizeof (Quadtree)):
newQuadtree->id = nQuadtree++;
newQuadtree->status = SOLID;
qTree->data.childrenIil = newQuadtree;
if (front->status == SOLID)
if (front->data.color != EMPTY)
qTree->data.children[i]->data.color = front->data.color;
else
if (back->status == SOLID)
if (back->data.color != EMPTY)
qTree->data.children[i]->data.color = back->data.color;
else
qTree->data.children[il->data.color = EMPTY;
else ( / * back node is mixed * /
newQuadtree->status = MIXED;
octreeToQuadtree (back, newguadtree);
octreeToQuadtree (front, newQuadtree):
}
}
}
```

2.8.10 RAY CASTING METHOD

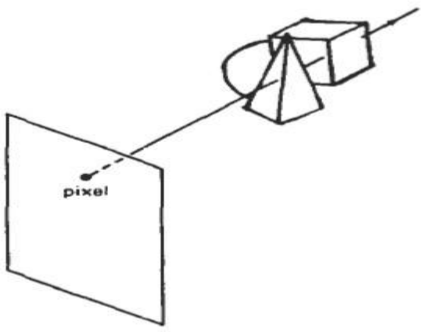
If we consider the line of sight from a pixel position on the view plane through a scene, as in the Fig. below, we can determine which objects in the scene intersect this line.

After calculating all ray-surface intersections, we identify the visible surface as the one whose intersection point is closest to the pixel. This visibility detection scheme uses ray-casting procedures.

Ray casting, as a visibility detection tool, is based on geometric optics methods, which trace the paths of light rays. Since there are an infinite number of light rays in a scene and we are interested only in those rays that pass through pixel positions, we can trace the light-ray paths backward from the pixels through the scene.

The ray-casting approach is an effective visibility-detection method for scenes with curved surfaces, particularly spheres.

A ray along a line of sight from a pixel position through a scene



We can think of ray casting as a variation on the depth-buffer method . In the depth-buffer algorithm, we process surfaces one at a time and calculate depth values for all projection points over the surface. The calculated surface depths are then compared to previously stored depths to determine visible surfaces at each pixel.

In ray casting, we process pixels one at a time and calculate depths for all surfaces along the projection path to that pixel. Ray casting is a special case of ray-tracing algorithms that trace multiple ray paths to pick up global reflection and refraction contributions from multiple objects in a scene. With ray casting, we only follow a ray out from each pixel to the nearest object.

2.8.11 Curved Surfaces

Effective methods for determining visibility for objects with curved surfaces include ray-casting and octree methods.

With ray casting, we calculate ray-surface intersections and locate the smallest intersection distance along the pixel ray.

With octree, once the representation has been established from the input definition of the objects, all visible surfaces are identified with the same processing procedures.

No special considerations need be given to different kinds of curved surfaces. We can also approximate a curved surface as a set of plane, polygon surfaces and use one of the other hidden-surface methods.With some objects, such as spheres, it can be more efficient as well as more accurate to use ray casting and the curved-surface equation.

Curved-Surface Representations

We can represent a surface with an implicit equation of the form $f(x, y, z) = 0$ or with a parametric representation .

Spline surfaces, for instance, are normally described with parametric equations.

In some cases, it is useful to obtain an explicit surface equation, as, for example, a height function over an xy ground plane:

$$z=f(x,y)$$

Many objects of interest, such as spheres, ellipsoids, cylinders, and cones, have quadratic representations.

Scan-line and ray-casting algorithms often involve numerical approximation techniques to solve the surface equation at the intersection point with a scan line or with a pixel ray. Various techniques, including parallel calculations and fast hardware implementations, have been developed for solving the curved-surface equations for commonly used objects.

Surface Contour Plots

For many applications in mathematics, physical sciences, engineering and other fields, it is useful to display a surface function with a set of contour lines that shows the surface shape. The surface may be described with an equation or with data tables.

With an explicit functional representation, we can plot the visible surface contour lines and eliminate those contour sections that are hidden by the visible parts of the surface.

To obtain an xy plot of a functional surface, we write the surface representation in the form

$$y=f(x,z) \qquad \text{----------------(1)}$$

A curve in the xy plane can then be plotted for values of z within some selected range, using a specified interval Δz. Starting with the largest value of z, we plot the curves from "front" to "back" and eliminate hidden sections.

We draw the curve sections on the screen by mapping an xy range for the function into an xy pixel screen range. Then, unit steps are taken in x and the corresponding y value for each x value is determined from Eq. (1) for a given value of z.

One way to identify the visible curve sections on the surface is to maintain a list of y_{min}, and y_{max}, values previously calculated for the pixel x coordinates on the screen.

As we step from one pixel x position to the next, we check the calculated y value against the stored range, y_{min}, and y_{max}, for the next pixel.

If $y_{min} <= y <= y_{max}$ that point on the surface is not visible and we do not plot it. But if the calculated y value is outside the stored y bounds for that pixel, the point is visible. We then plot the point and reset the bounds for that pixel.

2.8.12 WireFrame Methods

When only the outline of an object is to be displayed, visibility tests are applied to surface edges. Visible edge sections are displayed, and hidden edge sections can either be eliminated or displayed differently from the visible edges. For example,hidden edges could be drawn as dashed lines.

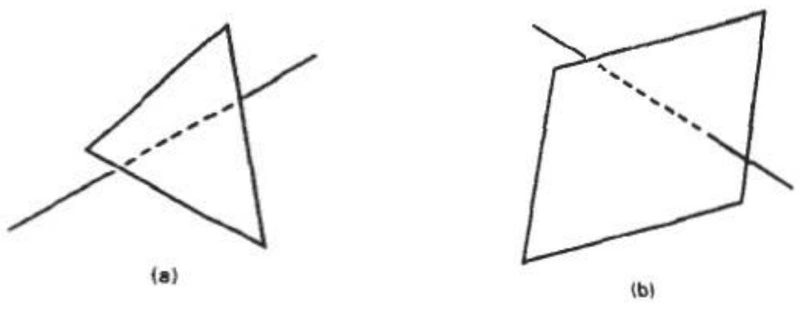
Procedures for determining visibility of object edges are referred to as wireframe-visibility methods. They are also called visible line detection methods or hidden-line detection methods.

A direct approach to identifying the visible lines in a scene is to compare each line to each surface, we now want to determine which sections of the lines are hidden by surfaces.

For each line, depth values are compared to the surfaces to determine which line sections are not visible. We can use coherence methods to identify hidden line segments without actually testing each coordinate position. If both line intersections with the projection of a surface boundary have greater depth than the surface at those points, the line segment between the intersections is completely hidden, as in Fig. (a).

This is the usual situation in a scene, but it is also possible to have lines and surfaces intersecting each other. When a line has greater depth at one boundary intersection and less depth than the surface at the other boundary intersection, the line must penetrate the surface interior, as in Fig. (b). In this case, we calculate the intersection point of the line with the surface using the plane equation and display only the visible sections.

Hidden line sections (dashed) for a line that (a) passes behind a surface and (b) penetrates a surface



Some visible-surface methods are readily adapted to wireframe visibility testing. Using a back-face method, we could identify all the back surfaces of an object and display only the boundaries for the visible surfaces. With depth sorting, surfaces can be painted into the refresh buffer so that surface interiors are in the background color, while boundaries are in the foreground color.

By processing the surfaces from back to front, hidden lines are erased by the nearer surfaces.An area-subdivision method can be adapted to hidden-line removal by displaying only the boundaries of visible surfaces. Scan-line methods can be used to display visible lines by setting points along the scan line that coincide with boundaries of visible surfaces.

2.8.13 VISIBILITY-DETECTION FUNCTIONS

Often, three-dimensional graphics packages accommodate several visible-surface detection procedures, particularly the back-face and depth-buffer methods.

A particular function can then be invoked with the procedure name, such as back-Face or depthBuffer.

UNIT III - GRAPHICS PROGRAMMING

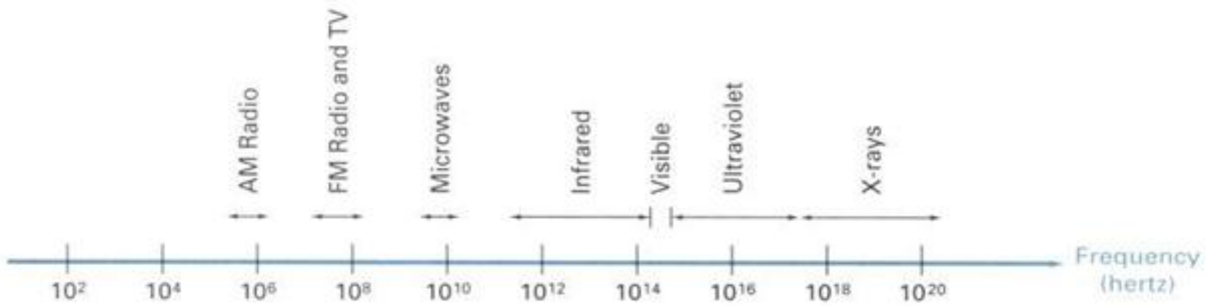
Color Models – RGB, YIQ, CMY, HSV – Animations – General Computer Animation, Raster, Keyframe - Graphics programming using OPENGL – Basic graphics primitives – Drawing three dimensional objects - Drawing three dimensional scenes

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 wave length λ of the wave.

- The wave length ad frequency of the monochromatic wave are inversely proportional to each other, with the proportionality constants as the speed of light C where C = λ f

- 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.
- The next property in the purity or saturation of the light.
    - 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.

Standard Primaries

XYZ Color Model

- The set of primaries is generally referred to as the XYZ or (X,Y,Z) color model where X,Y and Z represent vectors in a 3D, additive color space.

- Any color Cλ is expressed as

$$C\lambda = X\mathbf{X} + Y\mathbf{Y} + Z\mathbf{Z} \qquad \text{-------------(1)}$$

Where X,Y and Z designates the amounts of the standard primaries needed to match Cλ.

- It is convenient to normalize the amount in equation (1) against luminance (X + Y+ Z). Normalized amounts are calculated as,

$$x = X/(X+Y+Z), \quad y = Y/(X+Y+Z), \quad z = Z/(X+Y+Z)$$

  with $x + y + z = 1$

- Any color can be represented with just the x and y amounts. The parameters x and y are called the chromaticity values because they depend only on hue and purity.

- If we specify colors only with x and y, we cannot obtain the amounts X, Y and Z. so, a complete description of a color in given with the 3 values x, y and Y.

$$X = (x/y)Y, \qquad Z = (z/y)Y$$

  Where $z = 1-x-y$.

Intuitive Color Concepts

- Color paintings can be created by mixing color pigments with white and black pigments to form the various shades, tints and tones.

- Starting with the pigment for a „pure color‟ the color is added to black pigment to produce different shades. The more black pigment produces darker shades.

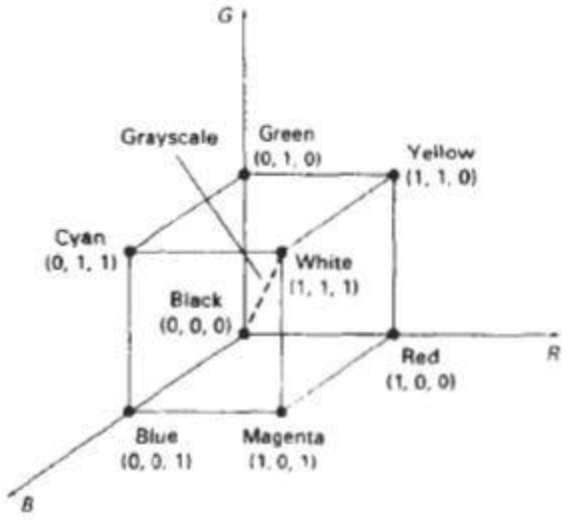
- Different tints of the color are obtained by adding a white pigment to the original color, making it lighter as more white is added.

- Tones of the color are produced by adding both black and white pigments.

RGB Color Model

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

2.5.5 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

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

- 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

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

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

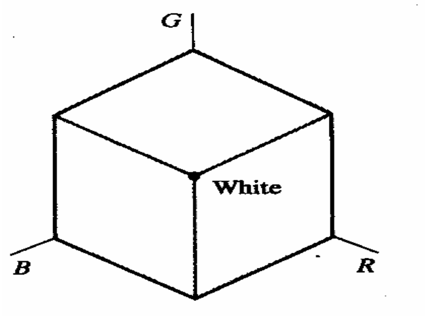
- Similarly the conversion of CMY to RGB representation is expressed as

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

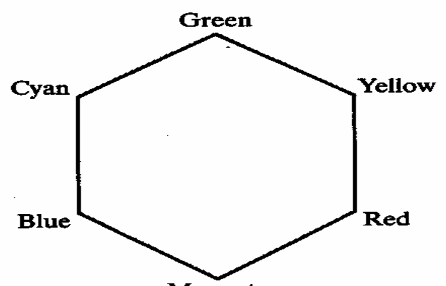
Where black is represented in the CMY system as the unit column vector.

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.

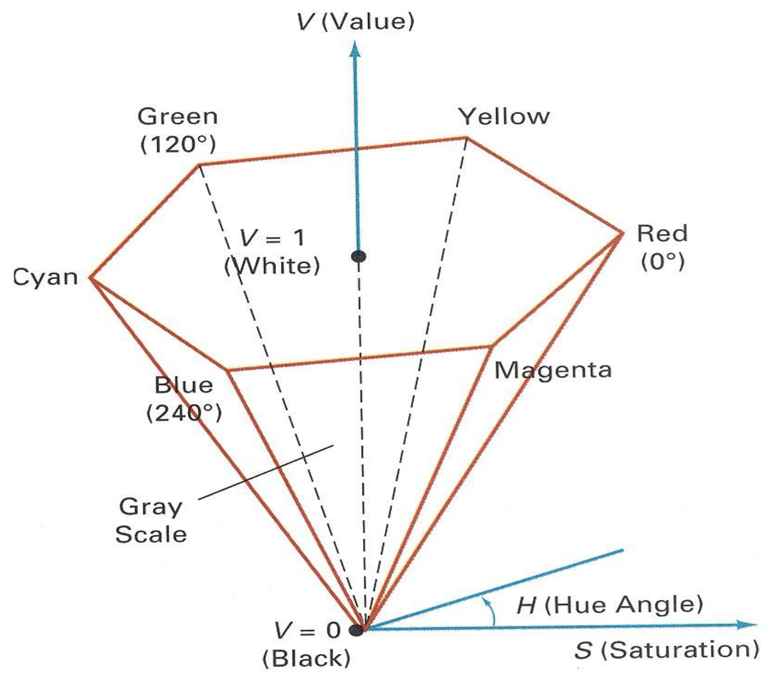


RGB Color Cube
(a)



Color Hexagon
(b)

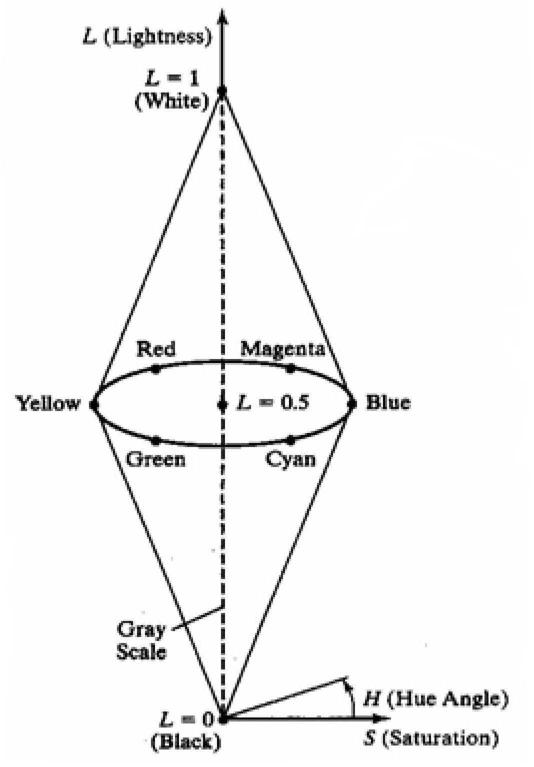
- The boundary of the hexagon represents the various hues, and it is used as the top of the HSV hexcone.
- In the hexcone, saturation is measured along a horizontal axis, and value is along a vertical axis through the center of the hexcone.
- Hue is represented as an angle about the vertical axis, ranging 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 hue. • 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 key frames 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,

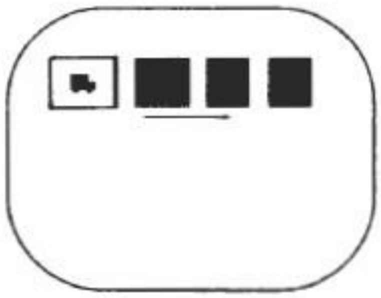
1. Object manipulation and rendering

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

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

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

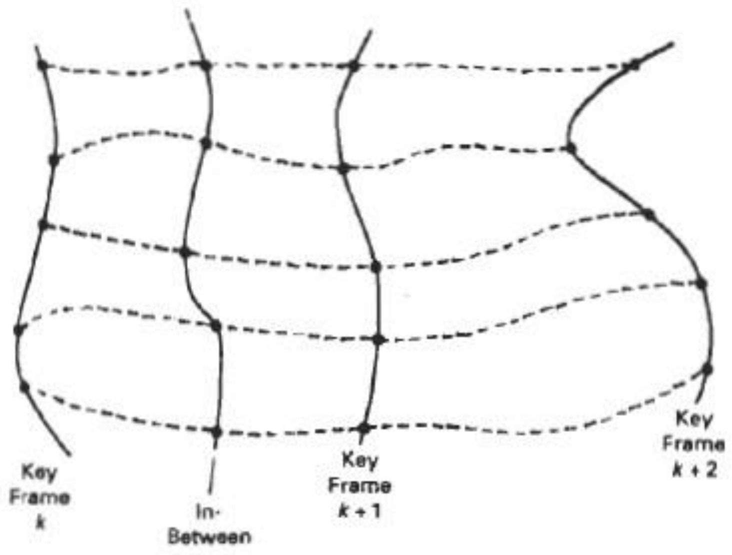
$$N_p = \text{int} ((V_{max} – 1)/(V_{min} – 1))$$

- 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

Δ= t2-t1/n+1

we can calculate the time for any in-between as

tBj = t1+j Δt, 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	Up Arrow
• GLUT_KEY_RIGHT	Right Arrow
• GLUT_KEY_DOWN	Down Arrow
• GLUT_KEY_PAGE_UP	Page Up
• GLUT_KEY_PAGE_DOWN	Page Down
• GLUT_KEY_HOME	Home
• GLUT_KEY_END	End
• GLUT_KEY_INSERT	Insert

4. glutMouseFunc(mymouse)

GLUT supports interaction with the computer mouse that is triggered when one of the three typical buttons is presses. A mouse callback fuction can be initiated when a given mouse button is pressed or released. The command glutMouseFunc() is used to specify the callback function to use when a specified button is is a given state at a certain location. This buttons are defined as either GL_LEFT_BUTTON, GL_RIGHT_BUTTON, or GL_MIDDLE_BUTTON and the states for that button are either GLUT_DOWN (when pressed) or GLUT_UP (when released). Finally, x and y callback parameters indicate the location (in window-relative coordinates) of the mouse at the time of the event.

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

GL_QUAD_STRIP
GL_POLYGON

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

glShadeModel : Sets the shading model. The mode parameter can be either GL_SMOOTH (the default) or GL_FLAT.

void glShadeModel (GLenum mode);

With flat shading, the color of one particular vertex of an independent primitive is duplicated across all the primitive‟s vertices to render that primitive. With smooth shading, the color at each vertex is treated individually.

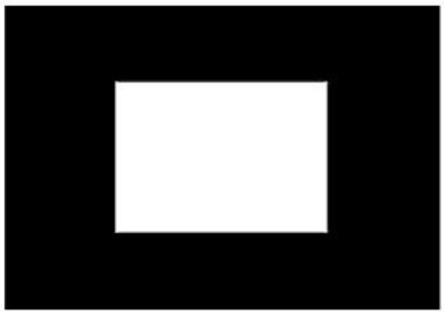
Example : OpenGL Program to draw three dots (2-Dimension)

```
#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;
}
```

Example : White Rectangle on a Black Background (3-Dimension co-ordinates)

```
#include "stdafx.h"
#include "gl/glut.h"
#include <gl/gl.h>

void Display(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glClear (GL_COLOR_BUFFER_BIT);
glColor3f (1.0, 1.0, 1.0);
glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
glBegin(GL_POLYGON);
glVertex3f (0.25, 0.25, 0.0);
glVertex3f (0.75, 0.25, 0.0);
glVertex3f (0.75, 0.75, 0.0);
glVertex3f (0.25, 0.75, 0.0);
glEnd();
glFlush();
}
int main (int argc, char **argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(640,480);
glutCreateWindow("Intro");
glClearColor(0.0,0.0,0.0,0.0);
glutDisplayFunc(Display);
glutMainLoop();
return 0;
}
```



Example : Big Dipper

```
#include "stdafx.h"
#include "gl/glut.h"
#include <gl/gl.h>
void myInit(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glColor3f (1.0, 1.0, 1.0);
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(289, 190);
glVertex2i(320, 128);
glVertex2i(239, 67);
glVertex2i(194, 101);
glVertex2i(129, 83);
glVertex2i(75, 73);
glVertex2i(74, 74);
glVertex2i(20, 10);
glEnd( );
glFlush();
}

int main (int argc, char **argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(640,480);
glutInitWindowPosition(100,150);
glutCreateWindow("Draw Big Dipper");
glutDisplayFunc(Display);
myInit();
glutMainLoop();
return 0;
}
```

Making Line Drawings

OpenGL makes it easy to draw a line: use GL_LINES as the argument to glBegin(), and pass it the two end points as vertices. Thus to draw a line between (40,100) and (202,96) use:

```
glBegin(GL_LINES); // use constant GL_LINES here
glVertex2i(40, 100);
glVertex2i(202, 96);
glEnd();
```

OpenGL provides tools for setting the attributes of lines.

A line"s color is set in the same way as for points, using glColor3f().

To draw thicker lines use glLineWidth(4.0). The default thickness is 1.0

To make stippled (dotted or dashed) lines, you use the command **glLineStipple()** to define the stipple pattern, and then we enable line stippling with **glEnable()**

```
glLineStipple(1, 0x3F07);
glEnable(GL_LINE_STIPPLE);
```

Drawing Polylines and Polygons

Polyline is a collection of line segments joined end to end. It is described by an ordered list of points,

$$p_0 = (x_0, y_0),\ p_1 = (x_1, y_1), \ldots,\ p_n = (x_n, y_n).$$

In OpenGL a polyline is called a "line strip", and is drawn by specifying the vertices in turn between glBegin(GL_LINE_STRIP) and glEnd().

For example, the code:

```
glBegin(GL_LINE_STRIP); // draw an open polyline
glVertex2i(20,10);
glVertex2i(50,10);
glVertex2i(20,80);
glVertex2i(50,80);
glEnd();
glFlush();


glBegin(GL_LINE_LOOP); // draw an polygon
glVertex2i(20,10);
glVertex2i(50,10);
glVertex2i(20,80);
glVertex2i(50,80);
glEnd();
glFlush();
```

Attributes such as color, thickness and stippling may be applied to polylines in the same way they are applied to single lines. If it is desired to connect the last point with the first point to make the polyline into a polygon simply replace GL_LINE_STRIP with GL_LINE_LOOP.

Polygons drawn using GL_LINE_LOOP cannot be filled with a color or pattern. To draw filled polygons we have to use glBegin(GL_POLYGON)

Drawing Aligned Rectangles.

A special case of a polygon is the **aligned rectangle,** so called because its sides are aligned with the coordinate axes.

OpenGL provides the ready-made function:

```
glRecti(GLint x1, GLint y1, GLint x2, GLint y2);

// draw a rectangle with opposite corners (x1, y1) and (x2, y2);
// fill it with the current color;

glClearColor(1.0,1.0,1.0,0.0); // white background
glClear(GL_COLOR_BUFFER_BIT); // clear the window
glColor3f(0.6,0.6,0.6); // bright gray
glRecti(20,20,100,70);
glColor3f(0.2,0.2,0.2); // dark gray
glRecti(70, 50, 150, 130);
```

aspect ratio = width/height;

Polygons

Polygons are the areas enclosed by single closed loops of line segments, where the line segments are specified by the vertices at their endpoints

Polygons are typically drawn by filling in all the pixels enclosed within the boundary, but you can also draw them as outlined polygons or simply as points at the vertices. A filled polygon might be solidly filled, or stippled with a certain pattern

OpenGL also supports filling more general polygons with a pattern or color.

To draw a convex polygon based on vertices (x0, y0), (x1, y1), …, (xn, yn) use the usual list of vertices, but place them between a glBegin(GL_POLYGON) and an glEnd():

```
glBegin(GL_POLYGON);
glVertex2f(x0, y0);
glVertex2f(x1, y1);
. . . . ..
glVertex2f(xn, yn);
glEnd();
```

The following list explains the function of each of the five constants:

GL_TRIANGLES: takes the listed vertices three at a time, and draws a separate triangle for each;

GL_QUADS: takes the vertices four at a time and draws a separate quadrilateral for each

GL_TRIANGLE_STRIP: draws a series of triangles based on triplets of vertices: *v*0, *v*1, *v*2, then *v*2, *v*1, *v*3, then *v*2, *v*3, *v*4, etc. (in an order so that all triangles are "traversed" in the same way;e.g. counterclockwise).

GL_TRIANGLE_FAN: draws a series of connected triangles based on triplets of vertices: *v*0, *v*1, *v*2, then *v*0, *v2*, *v*3, then *v*0, *v*3, *v*4, etc.

GL_QUAD_STRIP: draws a series of quadrilaterals based on foursomes of vertices: first v0, v1, v3, v2, then v2, v3, v5, v4, then v4, v5, v7, v6 (in an order so that all quadrilaterals are "traversed" in the same way; e.g. counterclockwise).

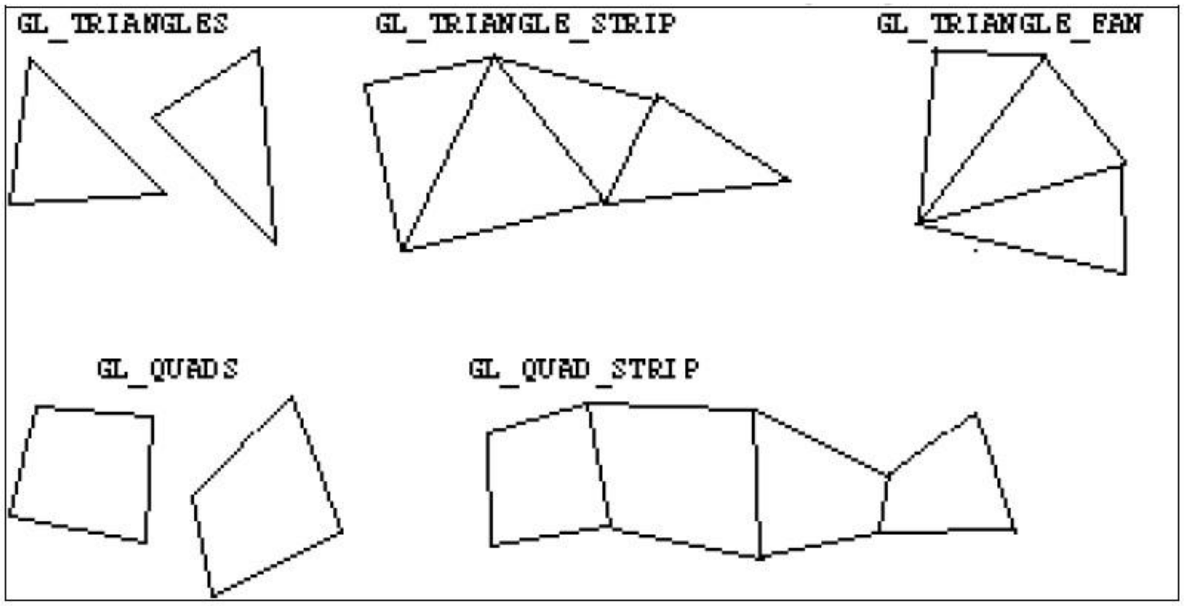


Figure 2.37. Other geometric primitive types.

Example to draw smooth shaded Trigangle with shades

```
#include "stdafx.h"
#include "gl/glut.h"
#include <gl/gl.h>
void init(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel (GL_SMOOTH);
gluOrtho2D (0.0, 640.0, 0.0, 480.0);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
}
void display(void)
```

```
{
glClear (GL_COLOR_BUFFER_BIT);
glBegin (GL_TRIANGLES);
glColor3f (1.0, 0.0, 0.0);
glVertex2f (50.0, 50.0);
glColor3f (0.0, 1.0, 0.0);
glVertex2f (250.0, 50.0);
glColor3f (0.0, 0.0, 1.0);
glVertex2f (50.0, 250.0);
glEnd();
glFlush ();
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (500, 500);
glutInitWindowPosition (100, 100);
glutCreateWindow ("Shade");
init ();
glutDisplayFunc(display);
glutMainLoop();
return 0;
}
```

Polygon Filling

A filled polygon might be **solidly filled**, or **stippled** with a certain pattern.

The pattern is specified with 128-byte array of data type GLubyte. The 128 bytes provides the bits for a mask that is 32 bits wide and 32 bits high.

GLubyte mask[]={0xff,0xfe…………128 entries}

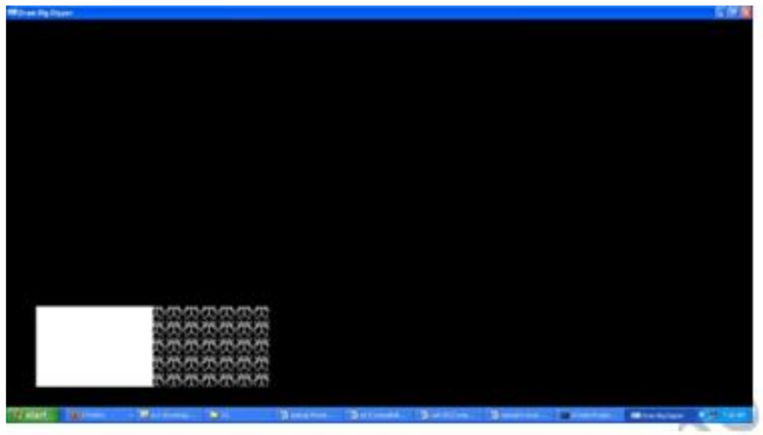
The first 4 bytes prescribe the 32 bits across the bottom row from left to right; the next 4 bytes give the next row up, etc..

Example

```
#include "stdafx.h"
#include "gl/glut.h"
#include <gl/gl.h>
```

```
        GLubyte mask[]={
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
   0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
   0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
   0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
   0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
   0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
   0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
   0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
   0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
   0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
   0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
   0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
   0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
   0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
   0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08};
void myInit(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glColor3f (1.0, 1.0, 1.0);
glPointSize(4.0);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}
void Display(void)
{
glClearColor(0.0,0.0,0.0,0.0); // white background
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 1.0, 1.0);
 glRectf(25.0, 25.0, 125.0, 125.0);
glEnable(GL_POLYGON_STIPPLE);
glPolygonStipple(mask);
glRectf (125.0, 25.0, 225.0, 125.0);
glDisable(GL_POLYGON_STIPPLE);
glFlush();
}

int main (int argc, char **argv)
{
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(640,480);
glutInitWindowPosition(100,150);
glutCreateWindow("Polygon Stipple");
glutDisplayFunc(Display);
myInit();
glutMainLoop();
return 0;
}
```



Simple Interaction with the mouse and keyboard

When the user presses or releases a mouse button, moves the mouse, or presses a keyboard key, an event occur. Using the OpenGL Utility Toolkit (GLUT) the programmer can register a callback function with each of these events by using the following commands:

glutMouseFunc(myMouse) which registers myMouse() with the event that occurs when the mouse button is pressed or released;

glutMotionFunc(myMovedMouse) which registers myMovedMouse() with the event that occurs when the mouse is moved while one of the buttons is pressed;

glutKeyboardFunc(myKeyboard) which registers myKeyBoard() with the event that occurs when a keyboard key is pressed.

Mouse interaction.

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

When a mouse event occurs the system calls the registered function, supplying it with values for these parameters. The value of button will be one of:

GLUT_LEFT_BUTTON,
GLUT_MIDDLE_BUTTON,
GLUT_RIGHT_BUTTON,

with the obvious interpretation, and the value of state will be one of: GLUT_UP or GLUT_DOWN. The values x and y report the position of the mouse at the time of the event.

Keyboard interaction.

As mentioned earlier, pressing a key on the keyboard queues a keyboard event. The callback function myKeyboard() is registered with this type of event through

glutKeyboardFunc(myKeyboard).

It must have prototype:

void myKeyboard(unsigned int key, int x, int y);

The value of key is the ASCII value12 of the key pressed. The values x and y report the position of the mouse at the time that the event occurred. (As before *y* measures the number of pixels down from the top of the window.)

```
void myKeyboard(unsigned char theKey, int mouseX, int mouseY)
{
GLint x = mouseX;
GLint y = screenHeight - mouseY; // flip the y value as always
switch(theKey)
{
case „p‟:
drawDot(x, y); // draw a dot at the mouse position
break;
case GLUT_KEY_LEFT: List[++last].x = x; // add a point
List[ last].y = y;
break;
case „E‟:
exit(-1); //terminate the program
default:
break; // do nothing
}
}
```

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_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();
        glPopMatrix();
glFlush();
glutSwapBuffers();
}
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;
}
```

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.

4.1 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

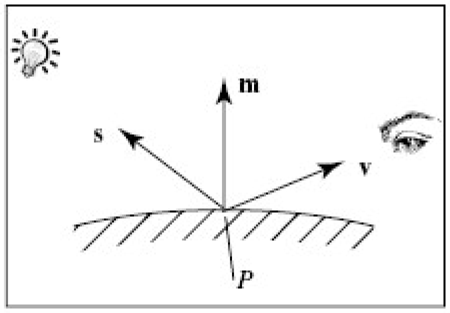
- Diffuse scattering occurs when some of the incident light slightly penetrates the surface and is re-radiated uniformly in all directions. Scattered light interacts strongly with the surface and so its color is usually affected by the nature of the surface material.
- Specular reflections are more mirrorlike 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.

4.1.1 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



1. The normal vector , m , to the surface at P.
2. The vector v from P to the viewer‟s eye.
3. 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.

4.1.2 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.

ae cross section of a point source illuminating a face S when m is aligned with s.

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.

4.1.3 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

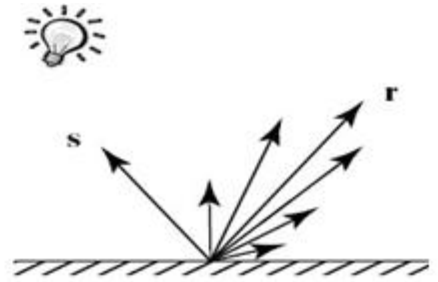
It is easy to apply and the highlights generated by the phong model given an plasticlike appearance , so the phong model is good when the object is made of shinny plastic or glass.

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

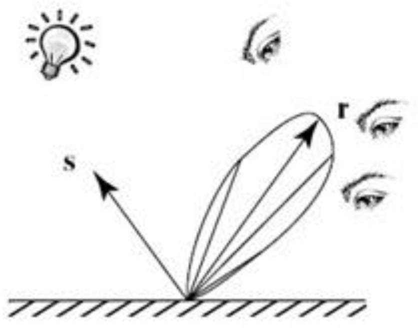
Fig a) shows a situation where light from a source impinges on a surface and is reflected in different directions.

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

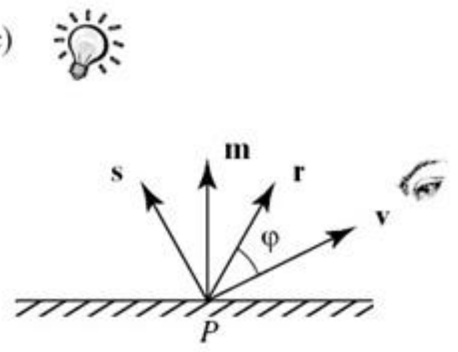
a)



b)



c)



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, according to:

$$r = -s + 2\frac{(s.m)}{|m|} m \text{ (the mirror – reflection direction)}$$

For surfaces that are shiny but are not true mirrors, the amount of light reflected falls off as the angle φ between r and v increases. In Phong model the φ is said to vary as some power f of the cosine of φ i.e., $(\cos(\varphi))^f$ in which f is chosen experimentally and usually lies between 1 and 200.

4.1.4 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.

4.1.5 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

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

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

4.1.6 To Add Color

Colored light can be constructed by adding certain amounts of red, green and blue light. When dealing with colored sources and surfaces we calculate each color component individually and simply add them to from the final color of the reflected light.

applied here

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

4.1.7 Shading and the Graphics Pipeline

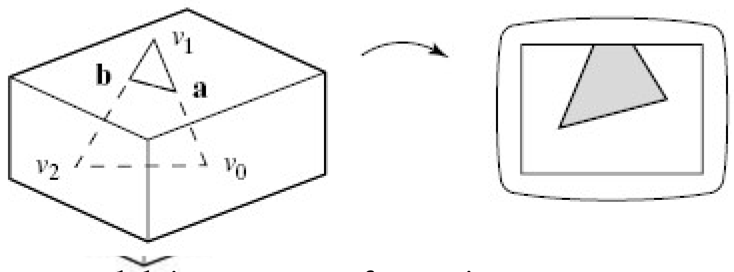
The key idea is that the vertices of a mesh are sent down the pipeline along with their associated vertex normals, and all shading calculations are done on vertices.

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 :

```
glBegin(GL_POLYGON);
        for( int i=0 ;i< 3; i++)
        {
            glNormal3f(m[i].x, m[i].y, m[i].z);
            glVertex3f(v[i].x, v[i].y, v[i].z);
        }
glEnd();
```

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.

Clipping a polygon against the view volume

The vertices are finally passed through the viewport transformation where they are mapped into the screen coordinates. The quadrilateral is then rendered.

4.1.8 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) : a local light source at the position (x,y,z)
(x,y,z,0) a vector to an infinitely remote light source in the direction (x,y,z)

te source
The above
fig,. shows a local source positioned at (0,3,3,1)
and a remote source "located" along vector
(3,3,0,0). Infinitely remote light sources are often
called "directional".

In OpenGL you can assign a different color to three types of light that a source emits : ambient , diffuse and specular. Arrays are used to hold the colors emitted by light sources and they are passed to glLightfv() through the following code:

```
GLfloat amb0[]={ 0.2 , 0.4, 0.6, 1.0 };          // define some colors
GLfloat diff0[]= { 0.8 ,0.9 , 0.5 ,1.0 };
GLfloat spec0[]= { 1.0 , 0.8 , 1.0, 1.0 };
glLightfv(GL_LIGHT0, GL_AMBIENT, amb0); //attach them to LIGHT0
glLightfv(GL_LIGHT0, GL_DIFFUSE, diff0);
glLightfv(GL_LIGHT0, GL_SPECULAR, spec0);
```

Colors are specified in RGBA format meaning red, green, blue and alpha. The alpha value is sometimes used for blending two colors on the screen. Light sources have various default values. For all sources:

Default ambient= (0,0,0,1);   dimmest possible :black

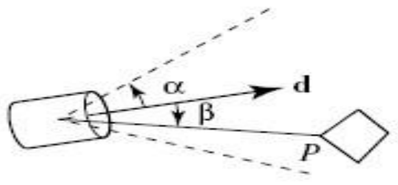
For light source LIGHT0:

Default diffuse= (1,1,1,1)   brightest possible:white
Default specular=(1,1,1,1)   brightest possible:white

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?

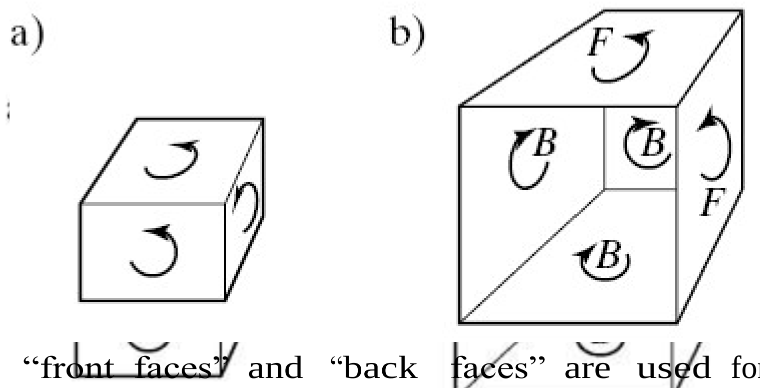
OpenGL computes specular reflection using halfway vector h= s + v. The true directions s and v are different at each vertex. If the light source is directional then s is constant but v varies from vertex to vertex. The rendering speed is increased if v is made constant for all vertices.

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.

OpenGL's definition of a front face

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.

4.1.9 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.

4.1.10 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.

4.2 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).

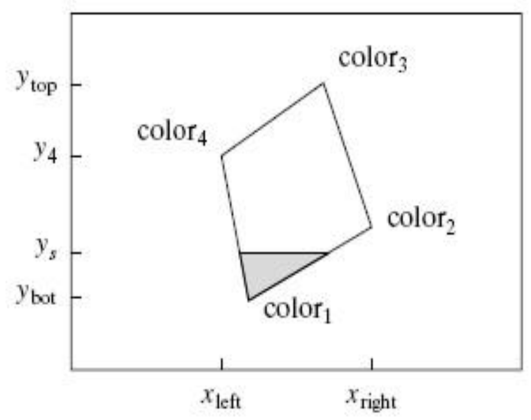
In both kinds of shading, the vertices are passed down the graphics pipeline, shading calculations are performed to attach a color to each vertex and the vertices are converted to screen coordinates and the face is "painted" pixel by pixel with the appropriate color.

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= $y_{bott}$ ; y<= $y_{top}$; y++)  // for each scan line
 {
     find $x_{left}$ and $x_{right}$
     for( int x= $x_{left}$ ; x<= $x_{right}$; 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.

4.2.1 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.

4.2.2 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= $y_{bott}$; y<=$y_{top}$ ; y++)          //for each scan line
{
   find $x_{left}$ and $x_{right}$
   find $color_{left}$ and $color_{right}$
   $color_{inc}$=( $color_{right}$ - $color_{left}$) / ($x_{right}$ - $x_{left}$);
   for(int x= $x_{left}$, c=$color_{left}$; x<=$x_{right}$; x++, c+=$color_{inc}$)
   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

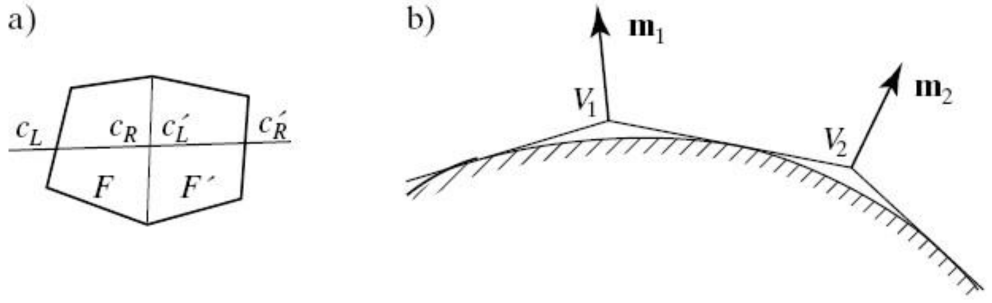


Fig.(a) shows two faces F and F‟ that share an edge. In rendering F, the colors C_L and C_R are used and in rendering F‟, the colors C‟$_L$ and C‟$_R$ are used. But since C_R equals C‟$_L$, there is no abrupt change in color at the edge along the scan line.

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.

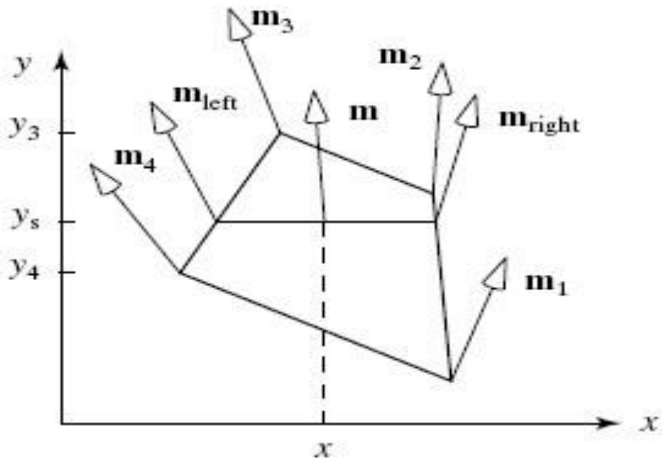
Phong Shading

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

4.3 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:

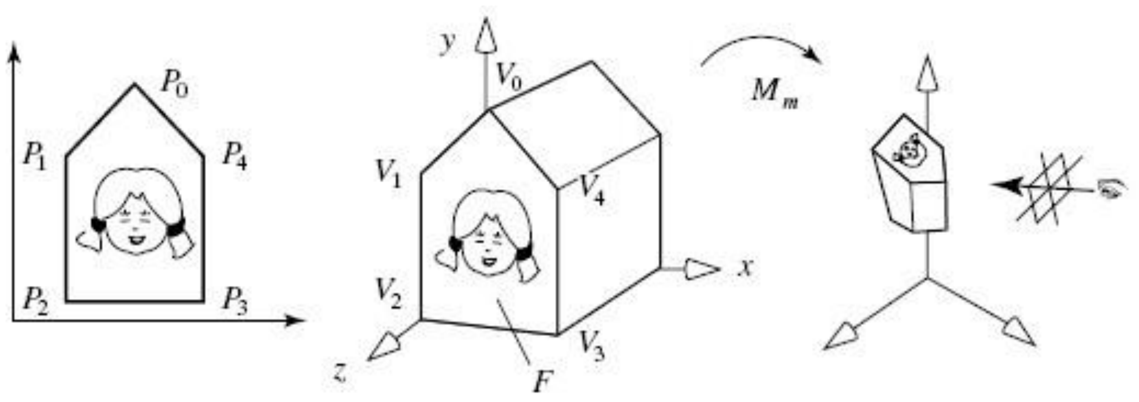
```
float fakesphere( float s, float t)
{
  float r= sqrt((s-0.5) * (s-0.5)+ (t-0.5) * (t-0.5));
  if (r < 0.3) return 1-r/0.3;        //sphere intensity
  else return 0.2;                    //dark background
}
```

This function varies from 1(white) at the center to 0 (black) at the edges of the sphere.

4.3.1 Painting the Textures onto a Flat Surface

Texture space is flat so it is simple to paste texture on a flat surface.

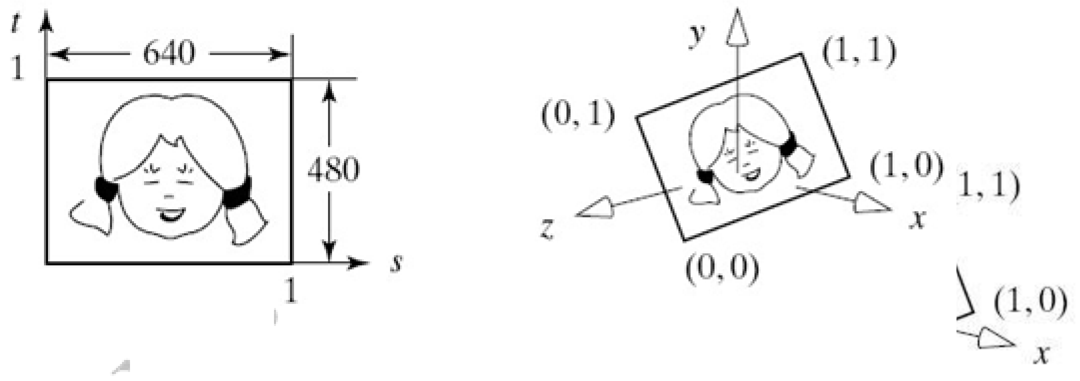
Mapping texture onto a planar polygon

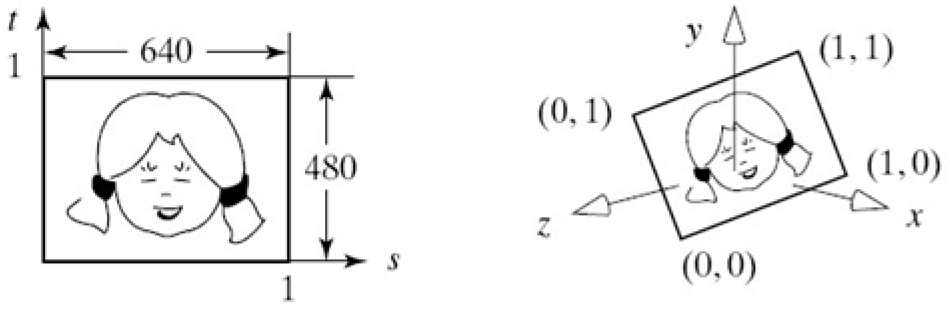


The fig. shows a texture image mapped to a portion of a planar polygon,F. We need to specify how to associate points on the texture with points on F.

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:



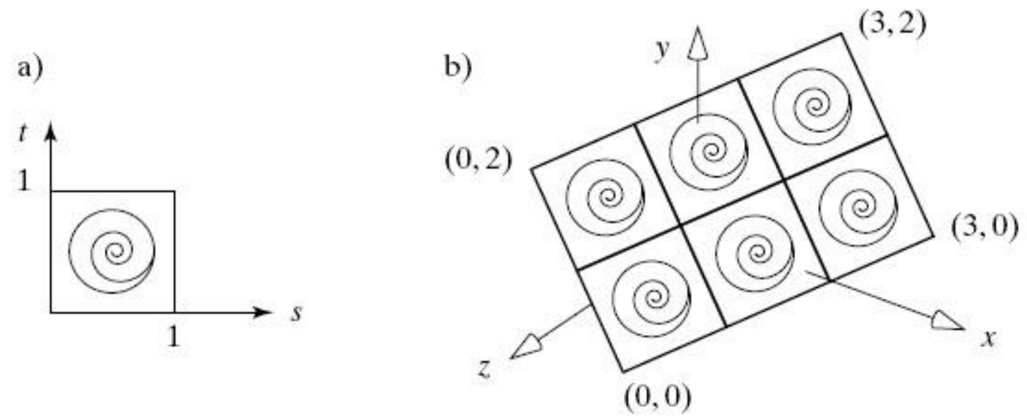


glEnd();

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.

Producing repeated textures



The fig. shows the use of texture coordinates , that tile the texture, making it to repeat. To do this some texture coordinates that lie outside the interval[0,1] are used. When rendering routine encounters a value of s and t outside the unit square, such as s=2.67, it ignores the integral part and uses only the fractional part 0.67. A point on a face that requires (s,t)=(2.6,3.77) is textured with texture (0.6,0.77).

The points inside F will be filled with texture values lying inside P, by finding the internal coordinate values (s,t) through the use of interpolation.

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;);

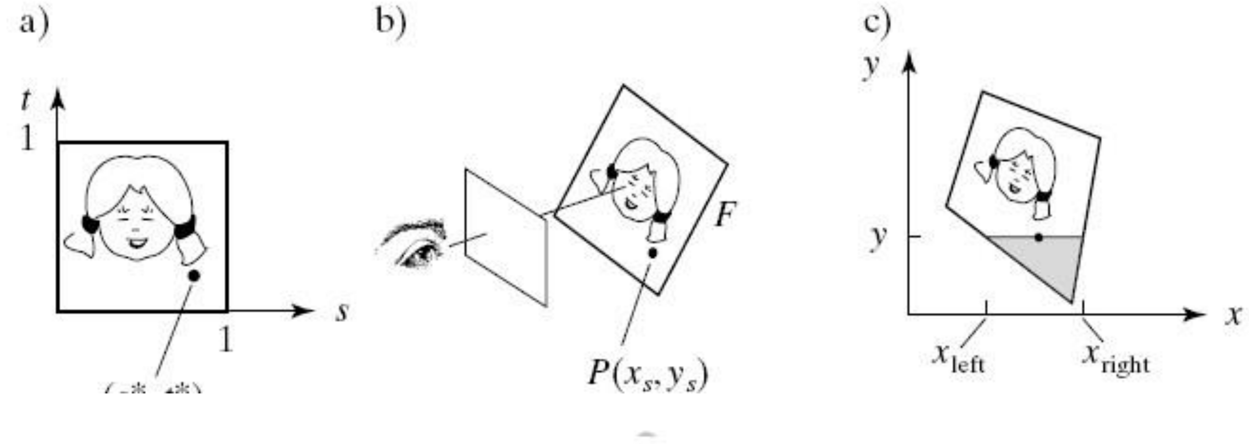
to hold all of the coordinate pairs of the mesh. The two important techniques to treat texture for an object are:

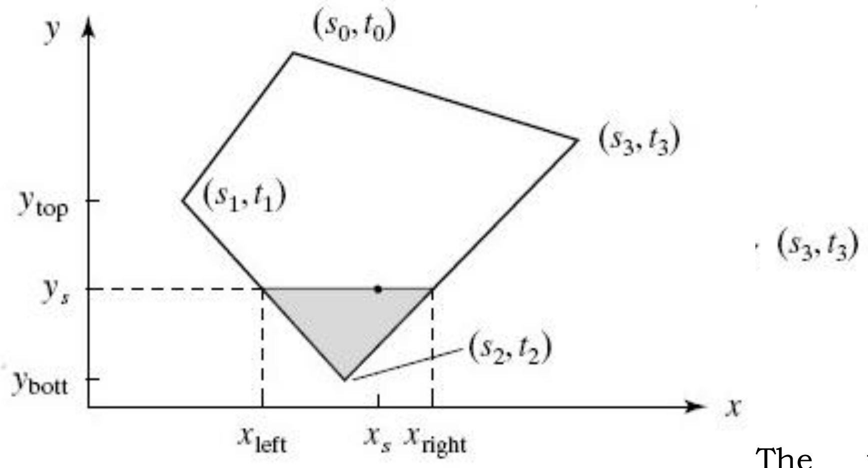
1. The mesh object consists of a small number of flat faces, and a different texture is to be applied to each. Each face has only a sigle normal vector, but its own list of texture coordinates. So the following data are associated with each face:
   - the number of vertices in the face.
   - the index 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.

4.3.2 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.

Rendering a face in a camera snapshot



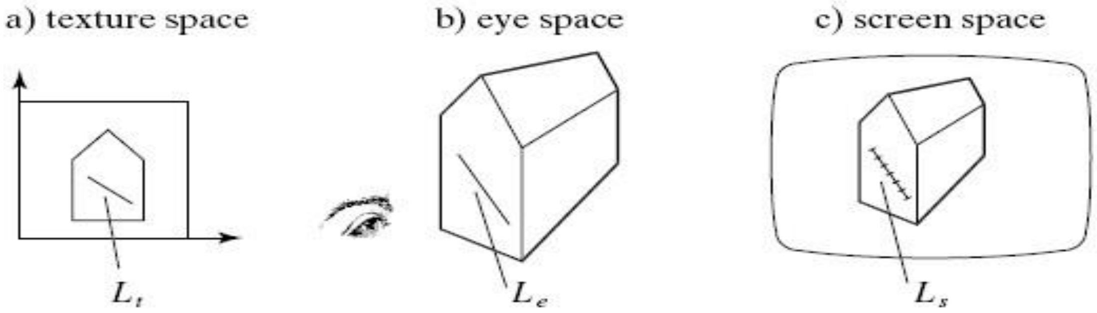


The fig shows the camera taking a snapshot of a face F with texture pasted onto it and the rendering in progress. The scan line y is being filled from x_{left} to x_{right}. For each x along this scan line, we compute the correct position on the face and from that , obtain the correct position (s^*, t^*) within the texture.

Incremental calculation of texture coordinates

We compute (s_{left},t_{left}) and (s_{right},t_{right}) for each scan line in a rapid incremental fashion and to interpolate between these values, moving across these scan lines. Linear interpolation produces some distortion in the texture. This distortion is disturbing in an animation when the polygon is rotating. Correct interpolation produces an texture as it should be. In an animation this texture would appear to be firmly attached to the moving or rotating face.

Lines in one space map to lines in another



Affine and projective transformations preserve straightness, so line L_e in eye space projects to line L_s in screen space, and similarly the texels we wish to draw on line L_s lie along the line L_t in texture spaces, which maps to L_e.

The question is : if we move in equal steps across L_s on the screen, how should we step across texels along L_t in texture space?

How does motion along corresponding lines operate?

The fig. shows a line AB in 3D being transformed into the line ab in

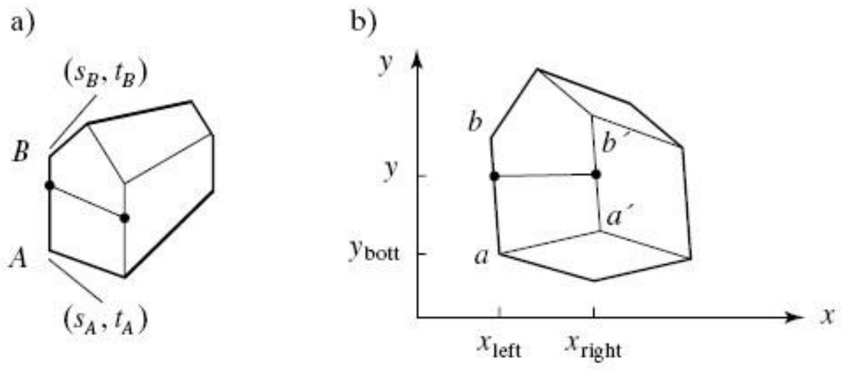
:

a, B maps to b. Consider the point R(g) that lies a fraction g of the way between A and B. This point maps to some point r(f) that lies a fraction f of the way from a to b. The fractions f and g are not the same. The question is, As f varies from 0 to 1, how exactly does g vary? How does motion along ab correspond to motion along AB?

Rendering Images Incrementally

We now find the proper texture coordinates (s,t) at each point on the face being rendered.

Rendering the texture on a face



The fig. shows the face of a barn. The left edge of the projected face has endpoints a and b. The face extends from x_{left} to x_{right} across scan line y. We need to find appropriate texture coordinates (s_{left}, t_{left}) and (s_{right}, t_{right}) to attach to x_{left} and x_{right}, which we can then interpolate across the scan line

Consider finding $s_{left}(y)$, the value of s_{left} at scan line y.We know that texture coordinate s_A is attached to point a and s_B is attached to point b. If the scan line at y is a fraction f of the way between y_{bott} and y_{top} so that $f=(y - y_{bott})/ (y_{top} - y_{bott})$, the proper texture coordinate to use is

$$s_{left}(y) = \left(lerp\left(\frac{s_A}{a_4}, \frac{s_B}{b_4}, f\right)\middle| lerp\left(\frac{1}{a_4}, \frac{1}{b_4}, f\right)\right.$$

and similarly for t_{left}.

Implications for the Graphics Pipeline

The shows a refinement of the pipeline. Each vertex V is associated with a texture pair (s,t) and a vertex normal. The vertex is transformed by the modelview matrix, producing vertex $A=(A_1, A_2, A_3)$ and a normal n'' in eye coordinates.

Shading calculations are done using this normal, producing the color $c=(c_r, c_g, c_b)$. The texture coordinates (s_A, t_A) are attached to A. Vertex A then goes perspective transformation, producing a $=(a_1,a_2, a_3,a_4)$. The texture coordinates and color c are not altered.

Next clipping against the view volume is done. Clipping can cause some vertices to disappear and some vertices to be formed. When a vertex D is created, we determine its position (d_1, d_2, d_3, d_4) and attach it to appropriate color and texture point. After clipping the face still consists of a number of verices, to each of which is attached a color and a texture point. For a point A, the information is stored in the array (a_1, a_2, a_3, a_4, s_A, t_A, c,1). A final term of 1 has been appended; this is used in the next step.

Perspective division is done, we need hyberbolic interpolation so we divide every term in the array that we wish to interpolate hyperbolically by a_4, to obtain the array (x, y, z, 1, s_A/a_4, t_4/a_4, c, $1/a_4$). The first three components of the array (x, y, z)=(a_1/a_4, a_2/a_4, a_3/a_4).

Finally, the rendering routine receives the array (x, y, z, 1, s_A/a_4, t_4/a_4, c, $1/a_4$) for each vertex of the face to be rendered.

4.3.3 What does the texture Modulate?

There are three methods to apply the values in the texture map in the rendering calculations

Creating a Glowing Object

This is the simplest method. The visibility intensity I is set equal to the texture value at each spot:

I=texture(s,t)

The object then appears to emit light or glow. Lower texture values emit less light and higher texture values emit more light. No additional lighting calculations are needed. OpenGL does this type of texturing using

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
          GL_REPLACE);
```

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

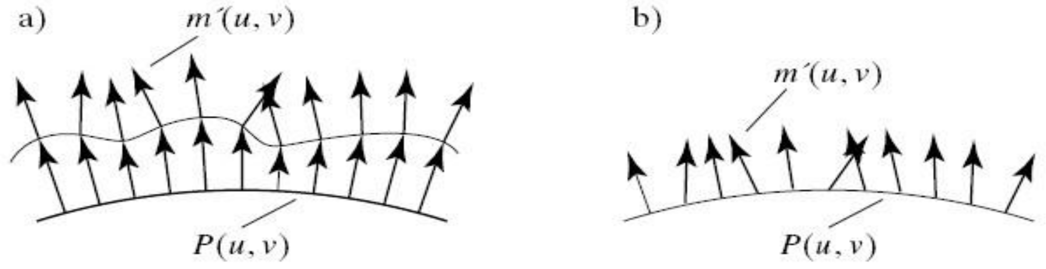
```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,
           GL_MODULATE);
```

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 X p_v) $texture_u$ – (m X 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.

4.3.4 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.

4.4 ADDING SHADOWS OF OBJECTS

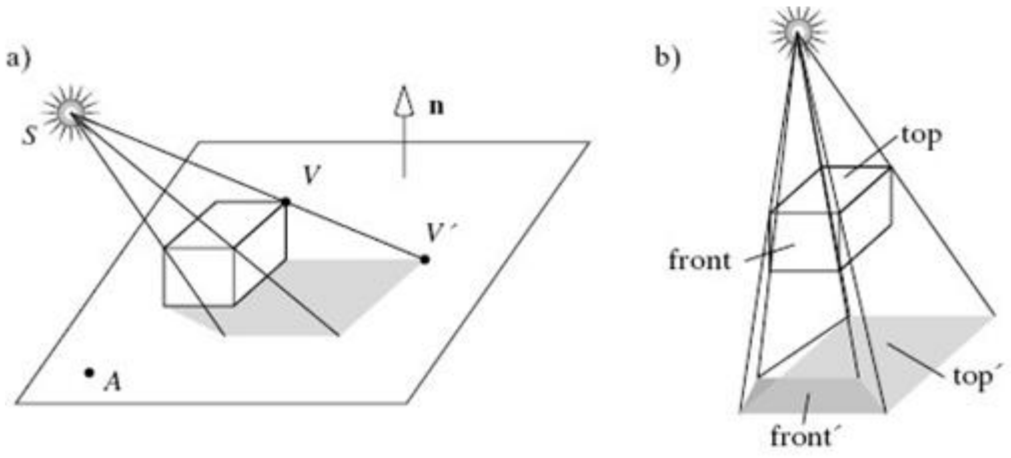
Shadows make an image more realistic. The way one object casts a shadow on another object gives important visual clues as to how the two objects are positioned with respect to each other. Shadows conveys lot of information as such, you are getting a second look at the object from the view point of the light source. There are two methods for computing shadows:

- Shadows as Texture
- Creating shadows with the use of a shadow buffer

4.4.1 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.

Buil**ding 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, this point is

$$V' = S + (V - S)\frac{n.(A - S)}{n.(V - S)}$$

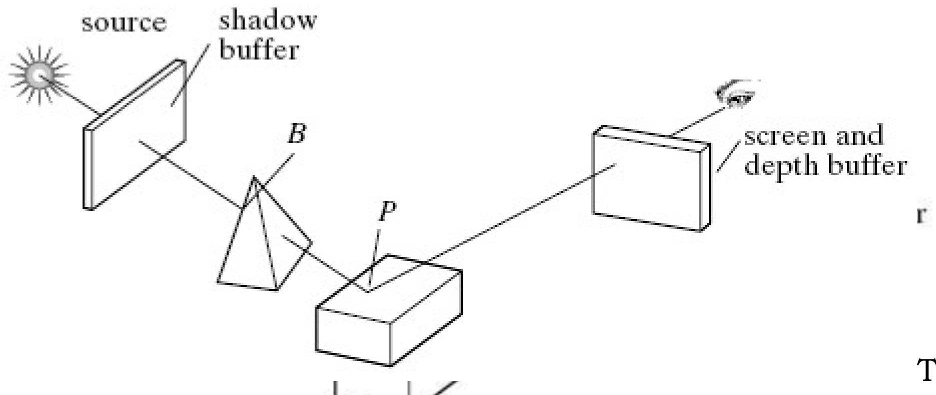
4.4.2 Creating Shadows with the use of a Shadow buffer

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 pseudodepth possible. Then through a camera positioned at the light source, each of the scene is rasterized but only the pseudodepth of the point on the face is tested. Each element of the shadow buffer keeps track of the smallest pseudodepth 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 pseudodepth to B; if the pyramid happens to be absent d[i][j] contains the pseudodepth 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 pseudodepth 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.

4.5 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, nearDist and farDist.

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

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

This matrix V accounts for the transformation of world points into camera coordinates. The utility routine computes the matrix V on the basis of current values of eye, u ,v and n and loads the matrix directly into the modelview matrix using glLoadMatrixf().

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.

4.5.1 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();
}
```

Rotating the Camera

Roll, pitch and yaw the camera , involves a rotation of the camera about one of its own axes.

To roll the camera we rotate it about its own n-axis. This means that both the directions u and v must be rotated as shown in fig.

Rolling the camera

Two new axes are formed u'' and v'' that lie in the same plane as u and v, and have been rotated through the angle α radians.

We form u'' as the appropriate linear combination of u and v and similarly for v'':

$u'' = \cos(\alpha)u + \sin(\alpha)v$;

$v'' = -\sin(\alpha)u + \cos(\alpha)v$

The new axes u'' and v'' then replace u and v respectively in the camera. The angles are measured in degrees.

Implementation of roll()

```
void Camera :: roll (float angle)
{ // roll the camera through angle degrees
        float cs = cos (3.14159265/180 * angle); float
        sn = sin (3.14159265/180 * angle); Vector3 t =
        u;                  //remember old u
        u.set(cs * t.x – sn * v.x , cs * t.y – sn * v.y, cs * t.z – sn * v.z);
        v.set(sn * t.x + cs * v.x , sn * t.y + cs * v.y, sn * t.z + cs * v.z);
        setModelViewMatrix();
}
```

Implementation of pitch()

```
void Camera :: pitch (float angle)
{ // pitch the camera through angle degrees around U
       float cs = cos(3.14159265/180 * angle);
       float sn = sin(3.14159265/180 * angle);
       Vector3 t(v); // remember old v
       v.set(cs*t.x - sn*n.x, cs*t.y - sn*n.y, cs*t.z - sn*n.z);
       n.set(sn*t.x + cs*n.x, sn*t.y + cs*n.y, sn*t.z + cs*n.z);
       setModelViewMatrix();
}
```

Implementation of yaw()

```
void Camera :: yaw (float angle)
{ // yaw the camera through angle degrees around V
        float cs = cos(3.14159265/180 * angle);
        float sn = sin(3.14159265/180 * angle);
        Vector3 t(n); // remember old v
        n.set(cs*t.x - sn*u.x, cs*t.y - sn*u.y, cs*t.z - sn*u.z);
        u.set(sn*t.x + cs*u.x, sn*t.y + cs*u.y, sn*t.z + cs*u.z);
        setModelViewMatrix();
}
```

The Camera class can be used with OpenGL to fly a camera through a scene. The scene consists of only a teapot. The camera is a global object and is set up in main(). When a key is pressed myKeyboard() is called and the camera is slid or rotated, depending on which key was pressed.

For instance, if P is pressed, the camera is pitched up by 1 degree. If CTRL F is pressed , the camera is pitched down by 1 degree. After the keystroke has been processed, glutPostRedisplay() causes myDisplay() to be called again to draw the new picture.

This application uses double buffering to produce a fast and smooth transition between one picture and the next. Two memory buffers are used to store the pictures that are generated. The display switches from showing one buffer to showing the other under the control of glutSwapBuffers().

Application to fly a camera around a teapot

```
#include "camera.h"
Camera cam;                //global camera object
//---------------------- myKeyboard-------------------------------
void myKeyboard(unsigned char key, int x, int y)
{
   switch(key)
     {
         //controls for the camera
           case „F‟:                          //slide camera forward
                    cam.slide(0, 0, 0.2);
                    break;
          case „F‟-64:                        //slide camera back
                    cam.slide(0, 0,-0.2);
                    break;
          case „P‟:
                    cam.pitch(-1.0);
                    break;
```

```
            case „P“-64:
                        cam.pitch(1.0);
                        break;
          //add roll and yaw controls
          }
               glutPostRedisplay();              //draw it again
}
//-------------------------myDisplay-----------------------------
void myDisplay(void)
{
          glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
          glutWireTeapot(1,0);            // draw the teapot
          glFlush();
          glutSwapBuffers();              //display the screen just made
}
//-------------------------main---------------------------
void main(int argc, char  **argv)
{
  glutInit(&argc, argv);
 glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);  //double buffering
 glutInitWindowSize(640, 480);
 glutInitWindowPosition(50, 50);
 glutCreateWindow("fly a camera around a teapot");
 glutKeyboardFunc(myKeyboard);
 glutDisplayFunc(myDisplay);
 glClearColor(1.0f, 1.0f, 1.0f, 1.0f);                    //background is white
 glColor3f(0.0f, 0.0f, 0.0f);                             //set color of stuff
 glViewport(0, 0, 640, 480);
 cam.set(4, 4, 4, 0, 0, 0, 0, 1, 0);              //make the initial camera
 cam.setShape(30.0f, 64.0f/48.0f, 0.5f, 50.0f);
 glutMainLoop();
}
```

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.

5.1 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.

5.1.1 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.

5.1.2 Drawing Koch Curves and Snowflakes

The Koch curves can be viewed in another way: Each generation consists of four versions of the previous generations. For instance K_2 consists of four versions of K_1 tied end to end with certain angles between them.

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 $K_{n-1}$;
            Turn left $60°$;

            Draw $K_{n-1}$;
            Turn right $120$ ;

            Draw $K_{n-1}$;
            Turn left $60$ ;
            Draw $K_{n-1}$;
           }
```

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().

5.3 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.

5.3.1 An Experimental Copier

We take an initial image I_0 and put it through a special photocopier that produces a new image I_1. I_1 is not a copy of I_0 rather it is a superposition of several reduced versions of I_0. We then take I_1 and feed it back into the copier again, to produce image I_2. This process is repeated , obtaining a sequence of images I_0, I_1, I_2,... called the orbit of I_0.

Making new copies from old

In general this copier will have N lenses, each of which perform an affine mapping and then adds its image to the output. The collection of the N affine transformations is called an “iterated function system”.

An iterated function system is a collection of N affine transformations T_i, for i=1,2,…N.

5.3.2 Underlying Theory of the Copying Process

Each lens in the copier builds an image by transforming every point in the input image and drawing it on the output image. A black and white image I can be described simply as the set of its black points:

I = set of all black points = { (x,y) such that (x,y) is colored black }

I is the input image to the copier. Then the ith lens characterized by transformation T_i, builds a new set of points we denote as $T_i(I)$ and adds them to the image being produced at the current iteration. Each added set $T_i(I)$ is the set of all transformed points I:

$T_i(I)$ = { (x’,y’) such that (x’,y’) = $T_i(P)$ for some point P in I }

Upon superposing the three transformed images, we obtain the output image as the union of the outputs from the three lenses:

Output image = $T_1(I)$ U $T_2(I)$ U $T_3(I)$

The overall mapping from input image to output image as W(.). It maps one set of points – one image – into another and is given by:

$W(.)=T_1(.)$ U $T_2(.)$ U $T_3(.)$

For instance the copy of the first image I_0 is the set $W(I_0)$.

Each affine map reduces the size of its image at least slightly, the orbit converge to a unique image called the attractor of the IFS. We denote the attractor by the set A, some of its important properties are:

1. The attractor set A is a fixed point of the mapping W(.), which we write as W(A)=A. That is putting A through the copier again produces exactly the same image A.

   The iterates have already converged to the set A, so iterating once more makes no difference.
2. Starting with any input image B and iterating the copying process enough times, we find that the orbit of images always converges to the same A.

   If $I_k = W^{(k)}(B)$ is the kth iterate of image B, then as k goes to infinity I_k becomes indistinguishable from the attractor A.

5.3.3 Drawing the kth Iterate

We use graphics to display each of the iterates along the orbit. The initial image I_0 can be set, but two choices are particularly suited to the tools developed:

- I_0 is a polyline. Then successive iterates are collections of polylines.
- I_0 is a single point. Then successive iterates are collections of points.

Using a polyline for I_0 has the advantage that you can see how each polyline is reduced in size in each successive iterate. But more memory and time are required to draw each polyline and finally each polyline is so reduced as to be indistinguishable from a point.

Using a single point for I_0 causes each iterate to be a set of points, so it is straight forward to store these in a list. Then if IFS consists of N affine maps, the first iterate I_1 consists of N points, image I_2 consists of N^2 points, I_3 consists of N^3 points, etc.

Copier Operation pseudocode(recursive version)

```
void superCopier( RealPolyArray pts, int k)
{ //Draw kth iterate of input point list pts for the IFS
        int i;
        RealPolyArray newpts;          //reserve space for new list
        if(k==0) drawPoints(pts);
        else for(i=1; i<=N; i++)       //apply each affine
        {
                newpts.num= N * pts.num;    //the list size grows fast
                for(j=0; j<newpts.num; j++)    //transforms the jth point
                     transform(affines[i], pts.pt[j], newpts.pt[j]);
                superCopier(newpts, k – 1);
              }
   }
```

- If k=0 it draws the points in the list
- If k>0 it applies each of the affine maps T_i, in turn, to all of the points, creating a new list of points, newpts, and then calls superCopier(newpts, k – 1);

To implement the algorithm we assume that the affine maps are stored in the global array Affine affines[N].

Drawbacks

- Inefficient
- Huge amount of memory is required.

5.3.4 The Chaos Game

The Chaos Game is a nonrecursive way to produce a picture of the attractor of an IFS.

The process for drawing the Sierpinski gasket

```
 Set corners of triangle :p[0]=(0,0), p[1]=(1,0), p[2]=(.5,1)
 Set P to one of these, chosen randomly;
  do {
        Draw a dot at P;
        Choose one of the 3 points at random;
        Set newPt to the midpoint of P and the chosen point;
        Set P= newPt;
     } while(!bored);
```

A point P is transformed to the midpoint of itself and one of the three fixed points: p[0], p[1] or p[2]. The new point is then drawn as a dot and the process repeats. The picture slowly fills in as the sequence of dots is drawn.

The key is that forming a midpoint based on P is in fact applying an affine transformation. That is

$$P = \frac{1}{2}\,(P + p[\ldots]) \qquad \text{(find the midpoint of P and p[..])}$$

Can be written as

$$P = P\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} + \frac{1}{2}\, p[..]$$

So that P is subjected to the affine map, and then the transformed version is written back into P. The offset for this map depends on which point p[i[is chosen.

Drawing the Sierpinski gasket

One of the three affine maps is chosen at random each time, and the previous point P is subjected to it. The new point is drawn and becomes the next point to be transformed.

Also listed with each map is the probability pr_i that the map is chosen at each iteration.

Starting with a point P_0, the sequence of iterations through this system produces a sequence of points P_0, P_1,.., which we call the orbit of the system with starting point P_0. This is a random orbit, different points are visited depending on the random choices made at each iteration.

The idea behind this approach is that the attractor consists of all points that are reachable by applying a long sequence of affines in the IFS. The randomness is invoked to ensure that the system is fully exercised, that every combination of affine maps is used somewhere in the process.

Pseudocode for playing the Chaos Game

```
void chaosGame(Affine aff[], double pr[], int N)
{
  RealPoint P = { 0,0 ,0,0};              //set some initial point
  int index;
  do {
          index = chooseAffine(pr , N);    // choose the next affine
          P = transform(aff[ondex], P);
          drawRealDot(P);                   // draw the dot
     } while (!bored);
}
```

The function chaosGame() plays the chaos game. The function draws the attractor of the IFS whose N transforms are stored in the array aff[]. The probabilities to be used are stored in an array pr[]. At each iteration one of the N affine maps is chosen randomly by the function chooseAffine() and is used to transform the previous point into the next point.

Adding Color

The pictures formed by playing the Chaos Game are bilevel, black dots on a white background. It is easy to extend the method so 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.

5.4 THE MANDELBROT SET

Graphics provides a powerful tool for studying a fascinating collection of sets that are the most complicated objects in mathematics.

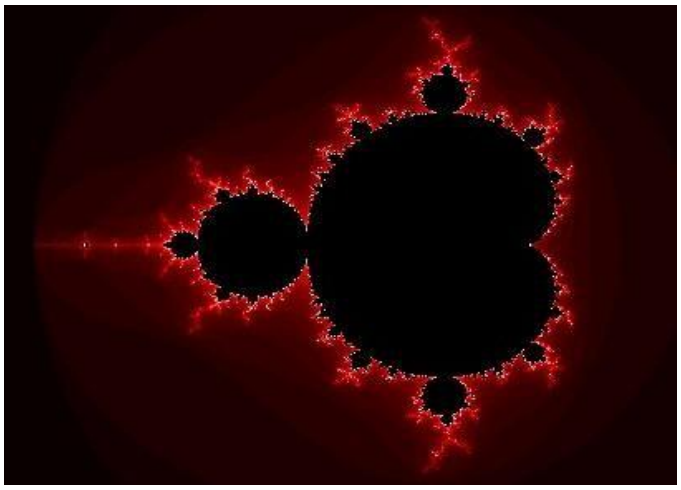
Julia and Mandelbrot sets arise from a branch of analysis known as iteration theory, which asks what happens when one iterates a function endlessly. Mandelbrot used computer graphics to perform experiments.

5.4.1 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

$d_1= (s)^2 + c$

$d_2= ((s)^2 + c)^2 + c$

$d_3= (((s)^2 + c)^2 + c)^2 + c$

$d_4= ((((s)^2 + c)^2 + c)^2 + c)^2 + c$ ------------------------------(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:

$(x + yi)^2 = (x^2 – y^2) + (2xy)^i$ ----------------------------------(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

$$p_+, p_- = \frac{1}{2} \pm \sqrt{\frac{1}{4} - c} \quad \text{--------------------------------(4)}$$

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.

5.4.2 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.

5.4.3 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).

A theorem from complex analysis states that if $|d_k|$ exceeds the value of 2, then the orbit will explode at some point. The number of iterations $|d_k|$ takes to exceed 2 is called the dwell of the orbit.

But if c lies in M, the orbit has an infinite dwell and we can't know this without it iterating forever. We set an upper limit Num on the maximum number of iterations we are willing to wait for.

A typical value is Num = 100. If $|d_k|$ has not exceeded 2 after Num iterates, we assume that it will never and we conclude that c is in M. The orbits for values of c just outside the boundary of M have a large dwell and if their dwell exceeds Num, we wrongly decide that they lie inside M. A drawing based on too small value of Num will show a Mandelbrot set that is slightly too large.

dwell() routine

```
int dwell (double cx, double cy)
{ // return true dwell or Num, whichever is smaller
        #define Num 100          // increase this for better pictures
        double tmp, dx=cx, dy=cy, fsq=cx *cx + cy * cy;
        for(int count=0; count<=Num && fsq <=4; count++)
        {
          tmp = dx;                          //save old real part
          dx = dx * dx – dy * dy +cx;        //new real part
          dy = 2.0 * tmp * dy + cy;          //new imaginary part
          fsq = dx * dx + dy * dy;
        }
       return count;                // number of iterations used
  }
```

For a given value of c = $c_x + c_y i$, the routine returns the number of iterations required for $|d_k|$ to exceed 2.

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.

5.4.4 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.

Assigning colors according to the orbit's dwell

The figure shows how color is assigned to a point having dwell d. For very small values of d only a dim blue component is used. As d approaches Num the red and green components are increased up to a maximum unity. This could be implemented in OpenGL using:

```
float v = d / (float)Num;
glColor3f(v * v, v*, v, 0.2);  // red & green at level v-squared
```

We need to see how to associate a pixel with a specific complex value of c. A simple approach is suggested in the following figure.

Establishing a window on M and a correspondence between points and pixels.

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

This specification determines the aspect ratio of the image :R= cols/rows. The user also chooses a portion of the complex plane to be displayed: a rectangular region having the same aspect ratio as the image. To do this the user specifies the region's upper left hand corner P and its width W. The rectangle's height is set by the required aspect ratio. The image is displayed in the upper left corner of the display.

To what complex value c= $c_x + c_y i$, does the center of the i, jth pixel correspond? Combining we get

$$c_{ij} = \left(P_x + \frac{i+\frac{1}{2}}{cols} W, P_y - \frac{j+\frac{1}{2}}{cols} W \right) \qquad \text{------------------------}(5)$$

for i = 0,…….,cols-1 and j=0,…..,rows-1.

The chosen region of the Mandelbrot set is drawn pixel by pixel. For each pixel the corresponding value of c is passed to dwell(), and the appropriate color associated with the dwell is found. The pixel is then set to this color.

Pseudocode for drawing a region of the Mandelbrot set

```
for(j=0; j<rows; j++)
    for(i=0; i<cols; i++)
        {
            find the corresponding c value in equation (5)
            estimate the dwell of the orbit
            find Color determined by estimated dwell
            setPixel( j , k, Color);
        }
```

A practical problem is to study close up views of the Mandelbrot set, numbers must be stored and manipulated with great precision.

Also when working close to the boundary of the set , you should use a larger value of Num. The calculation times for each image will increase as you zoom in on a region of the boundary of M. But images of modest size can easily be created on a microcomputer in a reasonable amount of time.

5.5 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.

5.5.1 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.

5.5.2 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.

5.5.3 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.

5.5.4 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.

Orbits that coincide at s

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 pseudocode is illustrative:

```
do {
    if ( coin flip is heads z= $\pm\sqrt{z-c}$ );
    else z = $-\sqrt{z-c}$ ;
        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.

5.6 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.

5.6.1 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

form 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.

5.6.2 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.

5.7 INTERSECTING RAYS WITH OTHER PRIMITIVES

First the ray is transformed into the generic coordinates of the object and then the various intersection with the generic object are computed.

1) Intersecting with a Square

The generic square lies in the z=0 plane and extends from -1 to 1 in both x and y.

The square can be transformed into any parallelogram positioned in space, so it is often used in scenes to provide this, flat surfaces such as walls and windows. The function hit(1) first finds where the ray hits the generic plane and then test whether this hit spot also lies within the square.

2) Intersecting with a Tapered Cylinder

The side of the cylinder is part of an infinitely long wall with a radius of L at z=0,and a small radius of S at z=1.This wall has the implicit form as

$$F(x, y, z)=x^2 + y^2- (1 + (S - 1) z)^2, \text{ for } 0 < z < 1$$

If S=1, the shape becomes the generic cylinder, if S=0 , it becomes the generic cone. We develop a hit () method for the tapered cylinder, which also provides hit() method for the cylinder and cone.

3) Intersecting with a Cube (or any Convex Polyhedron)

The convex polyhedron, the generic cube deserves special attention. It is centered at the origin and has corner at (±1, ±1, ±1) using all right combinations of +1 and -1.Thus,its edges are aligned with coordinates axes, and its six faces lie in the plan.
The generic cube is important for two reasons.

- A large variety of intersecting boxes can be modeled and placed in a scene by applying an affine transformation to a generic cube. Then, in ray tracing each ray can be inverse transformed into the generic cube's coordinate system and we can use a ray with generic cube intersection routine.
- The generic cube can be used as an extent for the other generic primitives in the sense of a bounding box. Each generic primitives, such as the cylinder, fits snugly inside the cube.

4) Adding More Primitives

To find where the ray S + ct intersects the surface, we substitute S + ct for P in F(P) (the explicit form of the shape)

$$d(t) = f(S + ct)$$

This function is

- positive at these values of t for which the ray is outside the object.
- zero when the ray coincides with the surface of the object and
- negative when the ray is inside the surface.

The generic torus has the implicit function as

$$F(P) = (\sqrt{(P_x^2+P_y^2)} - d)^2 + P_z^2 - 1$$

So the resulting equation d(t)=0 is quartic.

For quadrics such as the sphere, d(t) has a parabolic shape, for the torus, it has a quartic shape. For other surfaces d(t) may be so complicated that we have to search numerically to locate t's for which d(.) equals zero. The function for super ellipsoid is

$$d(t) = ((S_x + C_x t)^n + (S_y + C_y t)^n)^{m/n} + (S_y + C_y t)^m - 1$$

where n and m are constant that govern the shape of the surface.

5.8 ADDING SURFACE TEXTURE

A fast method for approximating global illumination effect is environmental mapping. An environment array is used to store background intensity information for a scene. This array is then mapped to the objects in a scene based on the specified viewing direction. This is called as environment mapping or reflection mapping.

To render the surface of an object, we project pixel areas on to surface and then reflect the projected pixel area on to the environment map to pick up the surface shading attributes for each pixel. If the object is transparent, we can also refract the projected pixel are also the environment map. The environment mapping process for reflection of a projected pixel area is shown in figure. Pixel intensity is determined by averaging the intensity values within the intersected region of the environment map.

A simple method for adding surface detail is the model structure and patterns with polygon facets. For large scale detail, polygon modeling can give good results. Also we could model an irregular surface with small, randomly oriented polygon facets, provided the facets were not too small.

Surface pattern polygons are generally overlaid on a larger surface polygon and are processed with the parent's surface. Only the parent polygon is processed by the visible surface algorithms, but the illumination parameters for the surfac3e detail polygons take precedence over the parent polygon. When fine surface detail is to be modeled, polygon are not practical.

5.8.1 Texture Mapping

A method for adding surface detail is to map texture patterns onto the surfaces of objects. The texture pattern may either be defined in a rectangular array or as a procedure that modifies surface intensity values. This approach is referred to as texture mapping or pattern mapping.

The texture pattern is defined with a rectangular grid of intensity values in a texture space referenced with (*s*,*t*) coordinate values. Surface positions in the scene are referenced with UV object space coordinates and pixel positions on the projection plane are referenced in *xy* Cartesian coordinates.

Texture mapping can be accomplished in one of two ways. Either we can map the texture pattern to object surfaces, then to the projection plane, or we can map pixel areas onto object surfaces then to texture space. Mapping a texture pattern to pixel coordinates is sometime called texture scanning, while the mapping from pixel coordinates to texture space is referred to as **pixel order scanning** or **inverse scanning** or **image order scanning**.

To simplify calculations, the mapping from texture space to object space is often specified with parametric linear functions

$$U=f_u(s,t)=a_u s+ b_u t + c_u$$
$$V=f_v(s,t)=a_v s+ b_v t + c_v$$

The object to image space mapping is accomplished with the concatenation of the viewing and projection transformations.

A disadvantage of mapping from texture space to pixel space is that a selected texture patch usually does not match up with the pixel boundaries, thus requiring calculation of the fractional area of pixel coverage. Therefore, mapping from pixel space to texture space is the most commonly used texture mapping method. This avoids pixel subdivision calculations, and allows anti aliasing procedures to be easily applied.

The mapping from image space to texture space does require calculation of the inverse viewing projection transformation m_{VP}^{-1} and the inverse texture map transformation m_T^{-1}.

5.8.2 Procedural Texturing Methods

Next method for adding surface texture is to use procedural definitions of the color variations that are to be applied to the objects in a scene. This approach avoids the transformation calculations involved transferring two dimensional texture patterns to object surfaces.

When values are assigned throughout a region of three dimensional space, the object color variations are referred to as solid textures. Values from texture space are transferred to object surfaces using procedural methods, since it is usually impossible to store texture values for all points throughout a region of space (*e.g*) Wood Grains or Marble patterns Bump Mapping.

Although texture mapping can be used to add fine surface detail, it is not a good method for modeling the surface roughness that appears on objects such as oranges, strawberries and raisins. The illumination detail in the texture pattern usually does not correspond to the illumination direction in the scene.

A better method for creating surfaces **bumpiness** is to apply a perturbation function to the surface normal and then use the perturbed normal in the illumination model calculations. This technique is called **bump mapping.**

If *P(u,v)* represents a position on a parameter surface, we can obtain the surface normal at that point with the calculation

$$N = P_u \times P_v$$

Where P_u and P_v are the partial derivatives of P with respect to parameters u and v.

To obtain a perturbed normal, we modify the surface position vector by adding a small perturbation function called a **bump function**.

$$P'(u,v) = P(u,v) + b(u,v)\, n.$$

This adds bumps to the surface in the direction of the unit surface normal n=N/|N|. The perturbed surface normal is then obtained as

$$N' = P_u' + P_v'$$

We calculate the partial derivative with respect to u of the perturbed position vector as

$$P_u' = \frac{\partial}{\partial u}(P + bn)$$

$$= P_u + b_u n + b n_u$$

Assuming the bump function b is small, we can neglect the last term and write

$$p_u' \approx p_u + b_u n$$

Similarly $p_v' = p_v + b_v n$.

and the perturbed surface normal is

$$N' = P_u + P_v + b_v (P_u \times n) + b_u (n \times P_v) + b_u b_v (n \times n).$$

But n x n =0, so that

$$N' = N + b_v (P_u \times n) + b_u (n \times P_v)$$

The final step is to normalize N' for use in the illumination model calculations.

5.8.3 Frame Mapping

Extension of bump mapping is frame mapping.

In frame mapping, we perturb both the surface normal N and a local coordinate system attached to N. The local coordinates are defined with a surface tangent vector T and a binormal vector B x T x N.

Frame mapping is used to model anisotrophic surfaces. We orient T along the grain of the surface and apply directional perturbations in addition to bump perturbations in the direction of N. In this way, we can model wood grain patterns, cross thread patterns in cloth and streaks in marble or similar materials. Both bump and directional perturbations can be obtained with table look-ups.

To incorporate texturing into a ray tracer, two principal kinds of textures are used.

- With image textures, 2D image is pasted onto each surface of the object.
- With solid texture, the object is considered to be carved out of a block of some material that itself has texturing. The ray tracer reveals the color of the texture at each point on the surface of the object.

5.8.4 Solid Texture

Solid texture is sometimes called as **3D texture**. We view an object as being carved out of some texture material such as marble or wood. A texture is represented by a function texture (x, y, z) that produces an (r, g, h) color value at every point in space. Think of this texture as a color or inkiness that varies with position, if u look at different points (x, y, z) you see different colors. When an object of some shape is defined in this space, and all the material outside the shape is chipped away to reveal the object’s surface the point (x, y, z) on the surface is revealed and has the specified texture.

5.8.5 Wood grain texture

The grain in a log of wood is due to concentric cylinders of varying color, corresponding to the rings seen when a log is cut. As the distance of the points from some axis varies, the function jumps back and forth between two values. This effect can be simulated with the modulo function.

Rings(r) = ((int) r)%2

where for rings about z-axis, the radius $r = \sqrt{x^2+y^2}$ .The value of the function rings () jumps between zero and unity as r increases from zero.

5.8.6 3D Noise and Marble Texture

The grain in materials such as marble is quite chaotic. Turbulent riverlets of dark material course through the stone with random whirls and blotches as if the stone was formed out of some violently stirred molten material. We can simulate turbulence by building a noise function that produces an apparently random value at each point (x,y,z) in space. This noise field is the stirred up in a well-controlled way to give appearance of turbulence.

5.8.7 Turbulence

A method for generating more interesting noise. The idea is to mix together several noise components: One that fluctuates slowly as you move slightly through space, one that fluctuates twice as rapidly, one that fluctuates four times rapidly, etc. The more rapidly varying components are given progressively smaller strengths

turb (s, x, y, z) = 1/2noise(s ,x, y, z) + 1/4noise(2s,x,y,z) +1/8 noise (4s,x,y,z).

The function adds three such components, each behalf as strong and varying twice as rapidly as its predecessor.

Common term of a turb () is a

$$\text{turb}(s, x, y, z) = 1/2 \sum_{k=0}^{m} 1/2^K \text{noise}(2^k s, x, y, z).$$

5.8.8 Marble Texture

Marble shows veins of dark and light material that have some regularity ,but that also exhibit strongly chaotic irregularities. We can build up a marble like 3D texture by giving the veins a smoothly fluctuating behavior in the z-direction and then perturbing it chantically using turb(). We start with a texture that is constant in x and y and smoothly varying in z.

Marble(x,y,z)=undulate(sin(2)).

Here undulate() is the spline shaped function that varies between some dark and some light value as its argument varies from -1 to 1.

5.9 REFLECTIONS AND TRANSPERENCY

The great strengths of the ray tracing method is the ease with which it can handle both reflection and refraction of light. This allows one to build scenes of exquisite realism, containing mirrors, fishbowls, lenses and the like. There can be multiple reflections in which light bounces off several shiny surfaces before reaching the eye or elaborate combinations of refraction and reflection. Each of these processes requires the spawnins and tracing of additional rays.

The figure 5.15 shows a ray emanating, from the eye in the direction dir and hitting a surface at the point P_h. when the surface is mirror like or transparent, the light I that reaches the eye may have 5 components

$$I=I_{amb} + I_{diff} + I_{spec} + I_{refl} + I_{tran}$$

The first three are the fan=miler ambient, diffuse and specular contributions. The diffuse and specular part arise from light sources in the environment that are visible at P_n. Iraft is the reflected light component ,arising from the light , Ik that is incident at P_n along the direction –r. This direction is such that the angles of incidence and reflection are equal,so

R=dir-2(dir.m)m

Where we assume that the normal vector m at P_h has been normalized.

Similarly I_{tran} is the transmitted light components arising from the light IT that is transmitted thorough the transparent material to Ph along the direction –t. A portion of this light passes through the surface and in so doing is bent, continuing its travel along –dir. The refraction direction + depends on several factors.

I is a sum of various light contributions, I_R and I_T each arise from their own fine components – ambient, diffuse and so on. I_R is the light that would be seen by an eye at Ph along a ray from P' to P_n. To determine I_R, we do in fact spawn a secondary ray from P_n in the direction r, find the first object it hits and then repeat the same computation of light component. Similarly IT is found by casting a ray in the direction t and seeing what surface is hit first, then computing the light contributions.

5.9.1 The Refraction of Light

When a ray of light strikes a transparent object, apportion of the ray penetrates the object. The ray will change direction from dir to + if the speed of light is different in medium 1 than in medium 2. If the angle of incidence of the ray is θ_1, Snell's law states that the angle of refraction will be

$$\frac{\sin(\theta_2)}{C_2} = \frac{\sin(\theta_1)}{C_1}$$

where C_1 is the spped of light in medium 1 and C_2 is the speed of light in medium 2. Only the ratio C_2/C_1 is important. It is often called the index of refraction of medium 2 with respect to medium 1. Note that if θ_1 ,equals zero so does θ_2 .Light hitting an interface at right angles is not bent.

In ray traving scenes that include transparent objects, we must keep track of the medium through which a ray is passing so that we can determine the value C_2/C_1 at the next intersection where the ray either exists from the current object or enters another one. This tracking is most easily accomplished by adding a field to the ray that holds a pointer to the object within which the ray is travelling.

Several design polices are used,

1) Design Policy 1: No two transparent object may interpenetrate.
2) Design Policy 2: Transparent object may interpenetrate.

5.10 COMPOUND OBJECTS: BOOLEAN OPERATIONS ON OBJECTS

A ray tracing method to combine simple shapes to more complex ones is known as constructive Solid Geometry(CSG). Arbitrarily complex shapes are defined by set operations on simpler shapes in a CSG. Objects such as lenses and hollow fish bowls, as well as objects with holes are easily formed by combining the generic shapes. Such objects are called compound, Boolean or CSG objects.

The Boolean operators: union, intersection and difference are shown in the figure 5.17.

Two compound objects build from spheres. The intersection of two spheres is shown as a lens shape. That is a point in the lens if and only if it is in both spheres. L is the intersection of the S_1 and S_2 is written as

$$L=S_1\cap S_2$$

The difference operation is shown as a bowl.A point is in the difference of sets A and B, denoted A-B,if it is in A and not in B.Applying the difference operation is analogous to removing material to cutting or carrying.The bowl is specified by

$$B=(S_1-S_2)-C.$$

The solid globe, S_1 is hollowed out by removing all the points of the inner sphere, S_2,forming a hollow spherical shell. The top is then opened by removing all points in the cone C.

A point is in the union of two sets A and B, denoted AUB, if it is in A or in B or in both. Forming the union of two objects is analogous to gluing them together.

The union of two cones and two cylinders is shown as a rocket.

$$R=C_1 \cup C_2 \cup C_3 \cup C_4.$$

Cone C_1 resets on cylinder C_2.Cone C_3 is partially embedded in C_2 and resets on the fatter cylinder C_4.

5.10.1 Ray Tracing CSC objects

Ray trace objects that are Boolean combinations of simpler objects. The ray inside lens L from t_3 to t_2 and the hit time is t_3.If the lens is opaque, the familiar shading rules will be applied to find what color the lens is at the hit spot. If the lens is mirror like or transparent spawned rays are generated with the proper directions and are traced as shown in figure 5.18.

Ray,first strikes the bowl at t_1,the smallest of the times for which it is in S_1 but not in either S_2 or C. Ray 2 on the other hand,first hits the bowl at t_5. Again this is the smallest time for which the ray is in S_1,but in neither the other sphere nor the cone.The hits at earlier times are hits with components parts of the bowl,but not with the bowl itself.

5.10.2 Data Structure for Boolean objects

Since a compound object is always the combination of two other objects say obj_1 OP Obj_2, or binary tree structure provides a natural description.

5.10.3 Intersecting Rays with Boolean Objects

We need to be develop a hit() method to work each type of Boolean object.The method must form inside set for the ray with the left subtree,the inside set for the ray with the right subtree,and then combine the two sets appropriately.

```
bool Intersection Bool::hit(ray in Intersection & inter)
{
   Intersection lftinter,rtinter;
   if (ray misses the extends)return false;
if (C) left −>hit(r,lftinter)||((right−>hit(r,rtinter)))
return false;
return (inter.numHits > 0);
}
```

Extent tests are first made to see if there is an early out.Then the proper hit() routing is called for the left subtree and unless the ray misses this subtree,the hit list rinter is formed.If there is a miss,hit() returns the value false immediately because the ray must hit dot subtrees in order to hit their intersection.Then the hit list rtInter is formed.

The code is similar for the union Bool and DifferenceBool classes. For UnionBool::hit(),the two hits are formed using

```
	if((!left-)hit(r,lftInter))**(|right-)hit(r,rtinter)))
	return false;
```

which provides an early out only if both hit lists are empty.
For differenceBool::hit(),we use the code

```
	if((!left−>hit(r,lftInter)) return false;
	if(!right−>hit(r,rtInter))
	{
		inter=lftInter;
		return true;
	}
```

which gives an early out if the ray misses the left subtree,since it must then miss the whole object.

5.10.4 Building and using Extents for CSG object

The creation of projection,sphere and box extend for CSG object. During a preprocessing step,the true for the CSG object is scanned and extents are built for each node and stored within the node itself. During raytracing,the ray can be tested against each extent encounted,with the potential benefit of an early out in the intersection process if it becomes clear that the ray cannot hit the object.