Computer Graphics Notes

Contributor: Abhishek Sharma

[Founder at TutorialsDuniya.com]

Computer Science Notes

COMPUTER GRAPHICS

UNIT –I

OUTPUT PRIMITIVES: POINTS AND LINES

Graphics programming packages provide functions to describe a scene in terms of these basic geometric structures, referred to as ***Output Primitives***, and to group sets of output primitives into more complex structures.

Each output primitive is specified with input coordinate data and other information about the way that objects is to be displayed.

Points and Straight Line segments are the simplest geometric components of pictures. Additional output primitives that can be used to construct a picture include circles and other conic sections, quadric surfaces, spline curves and surfaces, polygon color areas, and character strings.

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.

A Random-Scan (Vector) System stores point-plotting instructions in the display list, and coordinate values in these instructions are converted to deflection voltages that position the electron beam at the screen locations to be plotted during each refresh cycle.

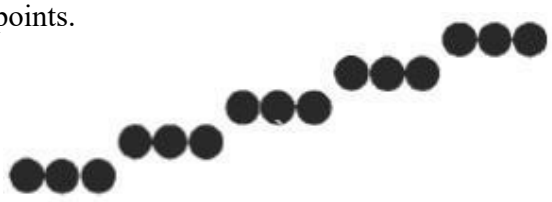
For a black-and-white raster system, a point is plotted by setting the bit value corresponding to a specified screen position within the frame buffer to 1. Then, as the electron beam sweeps across each horizontal scan line, it emits a burst of electrons (plots a point) whenever a value of 1 is encountered in the frame buffer.

With an RGB system, the frame buffer is loaded with the color codes for the intensities that are to be displayed at the screen pixel positions.

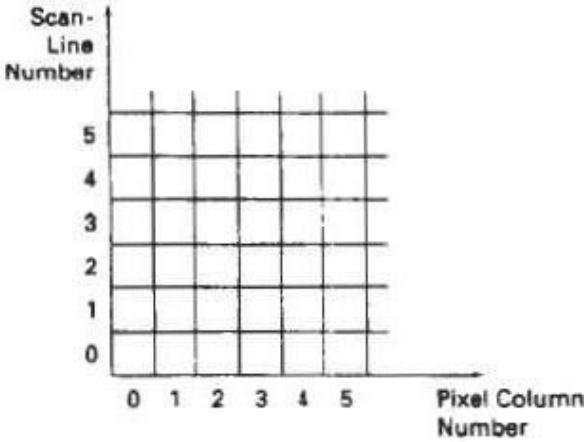
Line drawing is accomplished by calculating intermediate positions along the line path between two specified endpoint positions. An output device is then directed to fill in these positions between the endpoints.

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. Screen locations are referenced with integer values, so plotted positions may only approximate actual Line positions between two specified endpoints.

For example, a computed line position is (10.48, 20.51), it is rounded to (10, 21). This rounding of coordinate values to integers causes lines to be displayed with a stairstep appearance ("the jaggies"), as represented below. This stairstep shape is noticeable in low resolution systems.



For the raster-graphics device-level algorithms, object positions are specified directly in integer device coordinates.



To load a specified color into the frame buffer at a position corresponding to column ***x*** along scan line ***y***, we will assume we have available a low-level procedure of the form

```
setPixel (x, y)
```

Sometimes we want to retrieve the current frame-buffer intensity setting for a specified location. We accomplish this with the low-level function. We use,

```
getPixel (x, y)
```

LINE-DRAWING ALGORITHMS

The Cartesian slope-intercept equation for a straight line is

$$y = mx + b \longrightarrow (1)$$

with **m** representing the 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).**

We can determine the slope m and y intercept b with the following calculations:

$$m = \frac{(y_2 - y_1)}{(x_2 - x_1)} \longrightarrow (2)$$

$$b = y_1 - m \times 1 \longrightarrow (3)$$

Algorithms for displaying straight lines are based on the line equations (1) and the calculations given in equations (2) and (3).

For any given x interval Δx (X2-X1) along a line, we can compute the corresponding y interval Δy (Y2-Y1)from equation (2) as,

$$y = m\Delta x \longrightarrow (4)$$

Similarly, we can obtain the x interval Δx corresponding to a specified Δy as

$$\Delta x = \frac{\Delta y}{m} \longrightarrow (5)$$

These equations form the basis for determining deflection voltages in analog devices.

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 Equation 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 Equation 5.

For lines with $m = 1$, **Δx = Δy** and the horizontal and vertical deflections voltages are equal. In each case, a smooth line with slope m is generated between the specified endpoints.

DDA algorithm:

The **Digital Differential Analyzer (DDA)** is a Scan-Conversion line algorithm based calculating either Δy or Δx using equations (4) and (5).

Consider first a line with **positive slope, less than or equal to 1**, we sample at unit intervals (Δx=1) and compute each successive y value as

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

subscript k takes integer values starting from 1, for the first point, and increases by 1 until the final endpoints is reached.

Since m can be any real number between 0 & 1, the calculated y values must be rounded to the nearest integer.

For lines with a **positive slope greater than 1**, we reserve the roles of x & y. That is, we sample at unit y intervals (Δy=1) and calculate each succeeding x value as

$$x_{k+1} = x_k + \frac{1}{m} \longrightarrow (7)$$

Equations (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, so that the starting endpoint is that, then either we have Δx= -1 and

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

or (when the slope is greater than 1) we have△y = -1 with

$$x_{k+1} = x_k - \frac{1}{m} \longrightarrow (9)$$

Equations (6), (7), (8) and (9) can also be used to calculate pixel positions along a line with negative slope. 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 equation (6).

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

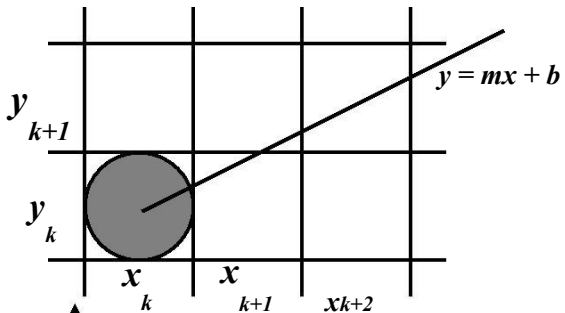
```
# define ROUND (a) ((int) (a+0.5))
void lineDDA (int xa, int ya, int xb, int yb)
{
     int dx = xb - xa, dy = yb - ya, steps, k;
     float xIncrement, yIncrement, x = xa, y =
     ya; if (abs (dx) > abs (dy))
          steps = abs (dx) ;
     else
          steps = abs (dy);
     xIncrement = dx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

The DDA algorithm is a faster method for calculating pixel position than the direct use of equation (1). We can improve the performance of the DDA algorithm by separating the increments m and 1/m into integer and fractional parts so that all calculations are reduced to integer operations.

Bresenham's Line Drawing Algorithm.

An accurate and efficient raster line-generating algorithm, developed by Bresenham, scans converts lines using only incremental integer calculations that can be adapted to display circles and other curves.

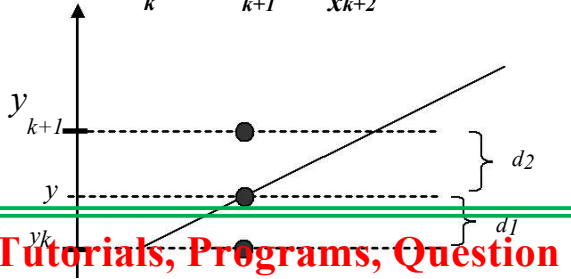
We first consider the scan-conversion process for lines with positive slope less than 1. Pixel positions along a line path are then determined by sampling at unit x intervals. Starting from the left endpoint (x_0, y_0) of a line, we step to each successive column (x position) and plot the pixel whose scan-line y value is closest to the line path.



The above figure demonstrates the k^{th} step in this process. Assuming we have determined that the pixel at (x_k, y_k) is to be displayed, we next need to decide which pixel to plot in column x_{k+1}. Our choices are the pixels at positions

(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 shown in the diagram. The y coordinate on the mathematical line at pixel column position x_k+1 is calculated as



$$y = m(x_k + 1) + b \longrightarrow (1)$$

Then

$$d_1 = y - y_k$$
$$= m(x_k + 1) + b - y_k$$

and

$$d_2 = (y_k+1) - y$$
$$= y_k + 1 - m(x_k + 1) - b$$

The difference between these two separations is

$$d_1 - d_2 = 2m(x_k + 1) - 2y_k + 2b - 1 \longrightarrow (2)$$

A decision parameter p_k for the kth step in the line algorithm can be obtained by rearranging equation (2) so that it involves only integer calculations. We accomplish this by substituting $\mathbf{m = \Delta y / \Delta x}$, where Δy and Δx are the vertical and horizontal separations of the endpoint positions, and defining:

$$p_k = \Delta x(d_1 - d_2)$$
$$= 2\Delta y.x_k - 2\Delta x \,.\, y_k + c \longrightarrow (3)$$

The sign of p_k is the same as sign of d_1-d_2, since $\Delta x > 0$ for our examples. Parameter c is constant and has the value $2\Delta y + \Delta x (2b-1)$, which is independent of 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(i.e., $d_1<d_2$), then decision parameter p_k is negative. In that case, we plot the lower pixel; otherwise, we plot the upper pixel.

Coordinate changes along the line occur in unit steps in either the x or y directions. Therefore, we can obtain the values of successive decision parameters using incremental integer calculations. At step k+1, the decision parameter is evaluated from equation (3) as

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

Substituting equation (3) from the preceding equation, we have

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

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

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x(y_{k+1}-y_k) \longrightarrow (4)$$

where the term $y_{k+1} - y_k$ is either 0 or 1, depending on the sign 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 3 at the starting pixel position (x_0,y_0) and with m evaluated as $\Delta y / \Delta x$:

$$p_0 = 2\Delta y - \Delta x \longrightarrow (5)$$

We can summarize Bresenham line drawing for a line with a positive slope less than 1 in the following listed steps:

1. *Input 2 endpoints, store left endpoint in* (x_0, y_0).
2. *Load* (x_0, y_0) *into frame buffer, i.e. plot the first point.*
3. *Calculate constants* $\Delta x, \Delta y, 2\Delta y, 2\Delta y - 2\,\Delta x$, *and initial value of decision parameter:* $p_0 = 2\Delta y - \Delta x$
4. *At each* x_k *along the line, start at* $k=0$, *test: if* $p_k < 0$, *plot* (x_{k+1}, y_k) *and*
   $$p_{k+1} = p_k + 2\Delta y$$
   else plot (x_{k+1}, y_{k+1}) *and*
   $$p_{k+1} = p_k + 2\Delta y - 2\Delta x$$
5. *Repeat step (4)* Δx *times.*

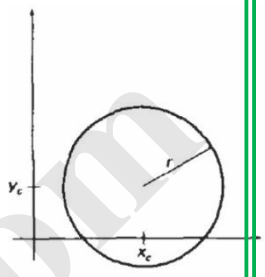
CIRCLE –GENERATING ALGORITHMS:

In general, a single procedure can be provided to display either ***Circular*** or ***Elliptical Curves.***

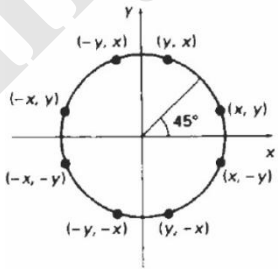
Properties of Circles:

A Circle is defined as the set of points that are all at a given distance r from a center position (x_c, y_c). This distance relationship is expressed by the Pythagorean theorem in Cartesian coordinates as,

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



symmetry of circles. The shape of the circle is similar in each quadrant. These symmetry conditions are illustrated in the above diagram, where a point at position ***(x , y)*** on a one eighth circle sector is mapped into the seven circle points inthe other octants of the **xy** plane. Taking advantage of the circle symmetry in this way we can generate all pixel positions around a circle by calculating only the points within the sector from ***x=0 to x=y.***



Midpoint Circle Algorithm:

For a given radius ***r*** and screen center position ***(x_c, y_c)*** we can first set up our algorithm to calculate pixel positions around a circle path centered at the coordinate origin ***(0, 0)***.

Then each calculated position ***(x, y)*** is moved to its proper screen position by adding x_c to **x** and y_c to **y**.

Along the circle section from ***x = 0*** to ***x = y*** in the first quadrant, the slope of the curve varies from ***0*** to ***-1***.

Positions in the other seven octants are then obtained by symmetry.

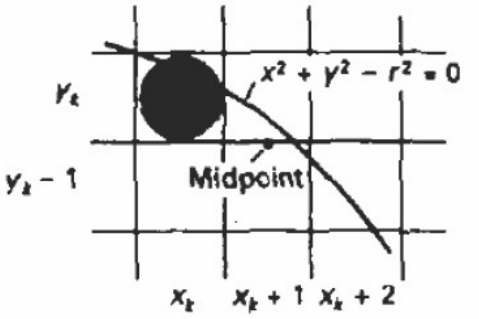
To apply the midpoint method, we define a circle function:

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.

To summarize, the relative position of any point ***(x, y)*** can be determined by checking the sign of the circle function:

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



The diagram shows the midpoint between the two candidate pixels at Sampling position x_k+1.
Assuming we have just plotted the pixel at (x_k, y_k), we 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 closer to the circle.
Decision Parameter is the circle function (4) evaluated at the midpoint between these two pixels:

$$p_k = f_{circle}\left(x_k + 1, y_k - \frac{1}{2}\right)$$

$$= (x_k + 1)^2 + \left(y_k - \frac{1}{2}\right)^2 - r^2$$

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 midposition is outside or on the circle boundary, and we select the pixel on scanline $y_k - 1$.
Successive decision parameters are obtained using incremental calculations. We can summarize the steps in the Midpoint algorithm as follows:
1. Input radius **r** and circle center (x_c, y_c), and obtain the first point on the circumference of a circle centered on the origin as

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

2. Calculate the initial value of the decision parameter as

$$p_0 = \frac{5}{4} - r$$

3. At each x_k position, 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

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1} \quad \text{where } 2x_{k+1} = 2x_k + 2 \text{ and } 2y_{k+1} = 2y_k - 2.$$

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

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

6. Repeat steps 3 through 5 until **$x \geq y$**.

ELLIPSE – GENERATING ALGORITHM:

An Ellipse is an elongated circle. So, ***Elliptical Curves*** can be generated by modifying circledrawing procedures to take into account the different dimensions along the major and minor axes.

Properties of Ellipses

An ***Ellipse*** is defined as the set of points such that the sum of the distances from two fixed positions(foci) is the same for all points. f the distances to the two foci from any point ***P = (x, y)*** on the ellipse are labeled **d_1** and **d_2,** then the general equation of an ellipse can be stated as,

$$d_1 + d_2 = constant \longrightarrow (1)$$

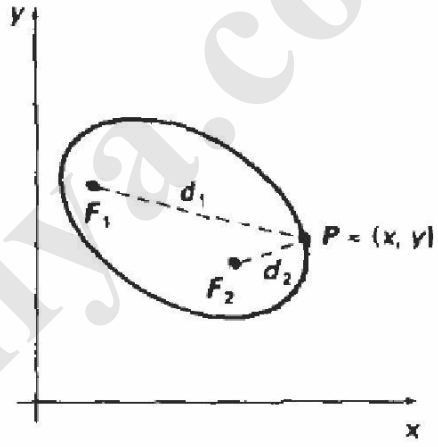
Expressing distances **d_1** and **d_2** in terms of the focal coordinates ***$F_1 = (x_1, y_1)$*** and ***$F_2 = (x_2, y_2)$,*** we have

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

We can rewrite the general ellipse equation in the form,

$$Ax^2 + By^2 + Cxy + Dx + Ey + F = 0 \longrightarrow (3)$$

where 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.

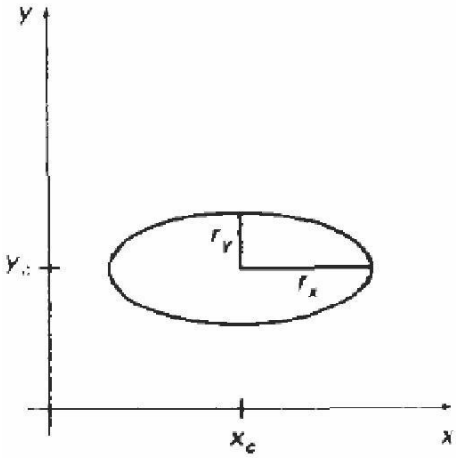


An interactive method for specifying an ellipse in an arbitrary orientation is to input the two foci and a point on the ellipse boundary.

In the diagram, we show an ellipse in "standard position" with major and minor axes oriented parallel to the ***x*** and y axes.

Parameter **r_x** labels the semimajor axis, and parameter **r_y** labels the semiminor axis. Using this the equation of the ellipse can be written as,

$$\frac{(x - x_c)^2}{r_x} + \frac{(y - y_c)^2}{r_y} = 1 \longrightarrow (4)$$

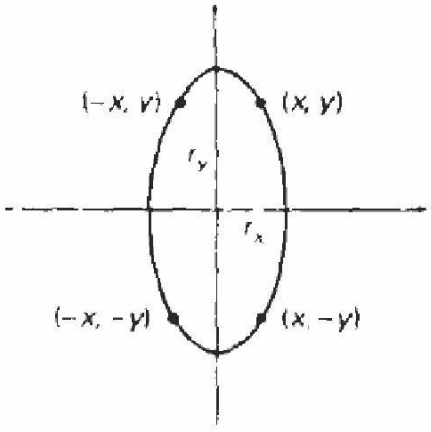


Using polar coordinates **r** and **θ**, we can also describe the ellipse in standard position with the parametric equations:

$$x = x_c + r_x \cos\vartheta$$
$$y = y_c + r_y \sin\vartheta \longrightarrow (5)$$

Symmetry considerations can be used to further reduce computations. An ellipse in standard position is symmetric between quadrants, but unlike a circle, it is not symmetric between the two octants of a quadrant.

Thus, we must calculate pixel positions along the elliptical arc throughout one quadrant, and then we obtain positions in the remaining three quadrants by symmetry as shown below: I



Midpoint Ellipse Algorithm:

Our approach here is similar *to* that used in displaying a raster circle. Given parameters $\mathbf{r_x}$, $\mathbf{r_y}$, and $\mathbf{(x_c, y_c)}$, we determine points ***(x, y)*** for an ellipse in standard position centered on the origin, and then we shift the points so the ellipse is centered at $\mathbf{(x_c, y_c)}$.

The midpoint ellipse method is applied throughout the first quadrant in two parts. The following diagram shows the division of the first quadrant according to the slope of an ellipse with $r_x < r_y$.

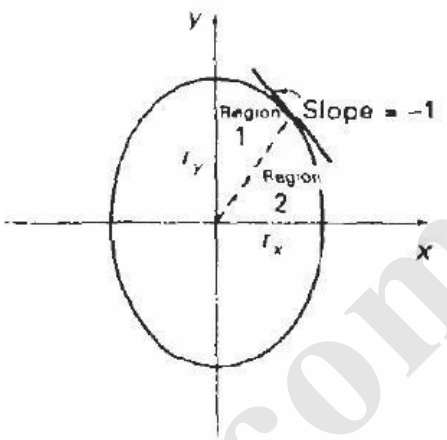
Regions 1 and 2 can be processed in various ways. We can start at position $\mathbf{(0, r_y)}$ and step clockwise along the elliptical path in the first quadrant, shifting from unit steps in ***x*** to unit steps in **y** when the slope becomes less than **-1.**

Alternatively, we could start at $(r_x, 0)$ and select points in a counterclockwise order, shifting from unit steps in **y** to unit steps in **x** when the slope becomes greater than **-1.**

With parallel processors, we could calculate pixel positions in the two regions simultaneously.

We define an Ellipse Function from equation (4) with $(x_c, y_c) = (0, 0)$ as

$$f_{ellipse}(x, y) = r_y^2 x^2 + r_x^2 y^2 - r_x^2 r_y^2$$



which has the following properties:

$f_{ellipse}(x, y)$ < 0, if (x, y) is inside the ellipse boundary,

$= 0$, if (x, y) is on the ellipse boundary

> 0, if (x, y) is outside the ellipse boundary

Thus the ellipse function serves as the decision parameter in the Midpoint 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,

   $$p1_0 = r_y^2 - r_x^2 r_y + \frac{1}{4} r_x^2$$

3. At each **x,** position in region 1, starting at $k = 3$, perform the following test:

   If $p1_k < 0$, the next point along the ellipse centered on (0, 0) is (x_{k+1}, y_k)

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

   Otherwise, the next point along the circle is (x_{k+1}, y_{k-1}) and

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

   with

   $$2r_y^2 x_{k+1} = 2r_y^2 x_k + 2r_y^2,$$

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

   and continue until $2r_y^2 x \geq 2r_x^2 y$

4. Calculate the initial value of the decision parameter in region **2** using the last point (x_0, y_0) calculated in region 1 as

   $$p2_0 = r_y^2 x_0 + \frac{1}{2}^2 + r_x^2 (y_0 - 1)^2 - r_x^2 r_y^2$$

5. At each y_k position in region **2,** starting at k = 0, perform the following test:

   If $p_k > 0$, the next point along the ellipse centered on **(0, 0)** is $(x_k, y_k - 1)$

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

   Otherwise, the next point along the circle is $(x_k + 1, y_k - 1)$ and $p2_k$

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

   using the same incremental calculations for x and y as in region 1.

6. Determine symmetry points in the other three quadrants.
7. Move each calculated 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 region 1 until $2r_y{}^2 x \geq 2r_x{}^2 y$.

ATTRIBUTES OF OUTPUT PRIMITIVES: LINE ATTRIBUTES

In general, any parameter that affects the way a primitive is to be displayed is referred to as an ***attribute parameter.*** Some attribute parameters, such as color and size, determine the fundamental characteristics of a primitive.

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:

Possible selections for the line-type attribute include *solid lines, dashed lines,* and *dotted lines*. We modify a line drawing algorithm to generate such lines by setting the length and spacing of displayed solid sections along the line path. A dashed line could be displayed by generating an interdash spacing that is equal to the length of the solid sections. Similar methods are used to produce other line-type variations.

To set line type attributes in a PHICS application program, a user invokes the function

```
setLinetype (lt)
```

where parameter 1 is assigned a positive integer value of 1, 2, 3, or 4 to generate lines that are, respectively, solid, dashed, dotted, or dash-dotted.

Raster line algorithms display line-type attributes by plotting pixel spans. For the various dashed, dotted, and dot-dashed pattern, the line-drawing procedure outputs sections of contiguous pixels along the line path, skipping over a number of intervening pixels between the solid spans.

Line Width:

Implementation of line-width options depends on the capabilities of the output device. A heavy line on the video monitor could be displayed as adjacent parallel lines, while a pen plotter might require pen changes.

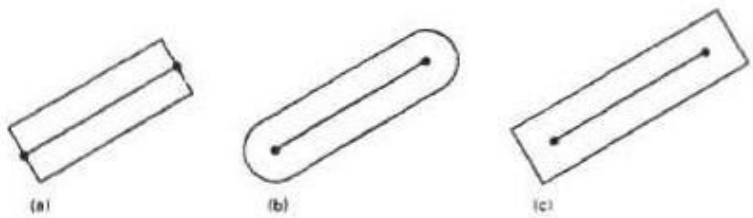
As with other PHIGS attributes, a line-width command is used to set the current line-width value in the attribute list.

We set the line-width attribute with the command:

```
setLinesidthScaleFactor (lw)
```

Line-width parameter **lw** is assigned a positive number to indicate the relative width of the line to be displayed. A value of 1 specifies a standard-width line.

For instance, on a pen plotter, 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.



(a) butt caps, (b) round caps, and (c) projecting square caps

Another problem with implementing width options using horizontal or vertical pixel spans is that the method produces lines whose ends are horizontal or vertical regardless of the slope of the line. This effect is more noticeable with very thick lines. We can adjust the shape of the line ends to give them a better appearance by adding ***line caps***.

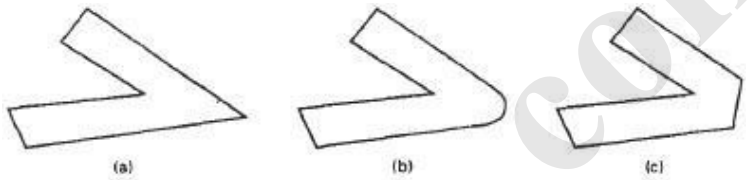
One kind of line cap is the ***butt cap*** obtained by adjusting the end positions of the component parallel lines so that the thick line is displayed with square ends that are perpendicular to the line path. If the specified line has slope m, the square end of the thick line has slope - 1 / m .

Another line cap is the ***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.

A third type of line cap is the ***projecting square cap***. Here, we simply extend the line and add butt caps that are positioned one-half of the line width beyond the specified endpoints.

Displaying thick lines using horizontal and vertical pixel spans, for example, leaves pixel gaps at the boundaries between lines of different slopes where there is a shift from horizontal spans to vertical spans. We can generate thick polylines that are smoothly joined at the cost of additional processing at the segment endpoints.

The following figure shows three possible methods for smoothly joining two line segments. A ***miter join*** is accomplished by extending the outer boundaries of each of the two lines until they meet. A ***round join*** is produced by capping the connection between the two segments with a circular boundary whose diameter is equal to the line width. And a ***bevel join*** is generated by displaying the line segments with butt caps and filling in the triangular gap where the segments meet.



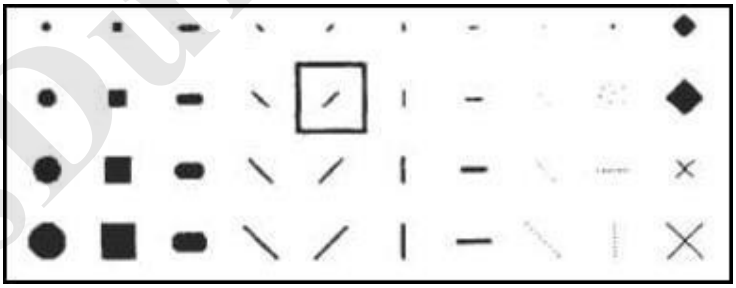
(a) miter join, (b) round join, and (c) bevel join

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 below.

These shapes can be stored in a pixel mask that identifies the array of pixel positions that are to be set along the line path.

To avoid setting pixels more than once in the frame buffer, we can simply accumulate the horizontal spans generated at each position of the mask and keep track of the beginning and ending *x* positions for the spans across each scan line.



Lines generated with pen (or brush) shapes can be displayed in various widths by changing the size of the mask. Also, lines can be displayed with selected patterns by superimposing the pattern values onto the pen or brush mask. An additional pattern option that can be provided in a paint package is the display of simulated brush strokes.

Line Color:

When a system provides color (or intensity) options, a parameter giving the current color index is included in the list of system-attribute values. A polyline 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 ***setpixel*** procedure. The number of color choices depends on the number of bits available per pixel in the frame buffer.

We set the line color value in PHIGS with the function,

```
SetPolylineColourIndex (lc)
```

Nonnegative integer values, corresponding to allowed color choices, are assigned to the line color parameter **lc**. A line drawn in the background color is invisible, and a user can erase a previously displayed line by respecifying it in the background color

CURVE ATTRIBUTES:

Parameters for curve attributes are the same as those for line segments. We can display curves with varying colors, widths, dotdash patterns, and available pen or brush options.

Raster curves of various widths can be displayed using the method of horizontal or vertical pixel spans. Where the magnitude of the curve slope is less than 1, we plot vertical spans; where the slope magnitude is greater than 1, we plot horizontal spans.

Another method for displaying thick curves is to fill in the area between two parallel curve paths, whose separation distance is equal to the desired width. We could do this using the specified curve path as one boundary and setting up the second boundary either inside or outside the original curve path. This approach shifts the original curve path either inward or outward, depending on which direction we choose for the second boundary. We can maintain the original curve position by setting the two boundary curves at a distance of one-half the width on either side of the specified curve path.

Although this method is accurate for generating thick circles, it provides only an approximation to the true area of other thick curves.

Curves drawn with pen and brush shapes can be displayed in different sizes and with superimposed patterns or simulated brush strokes.

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. General purpose raster-scan systems usually provide a wide range of colors, while random-scan monitors typically offer only a few color choices.

Color options are numerically coded with values ranging from 0 through the positive integers. For CRT monitors, these color codes are then converted to intensity level settings for the electron beams.

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, or

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 the Table.

Each of the three bit positions is used to control the intensity level (either on or off) of the corresponding electron gun in an RGB monitor. The leftmost bit controls the red gun, the middle bit controls the green gun, and the rightmost bit controls the blue gun. Adding more bits per pixel to the frame buffer increases the number of color choices.

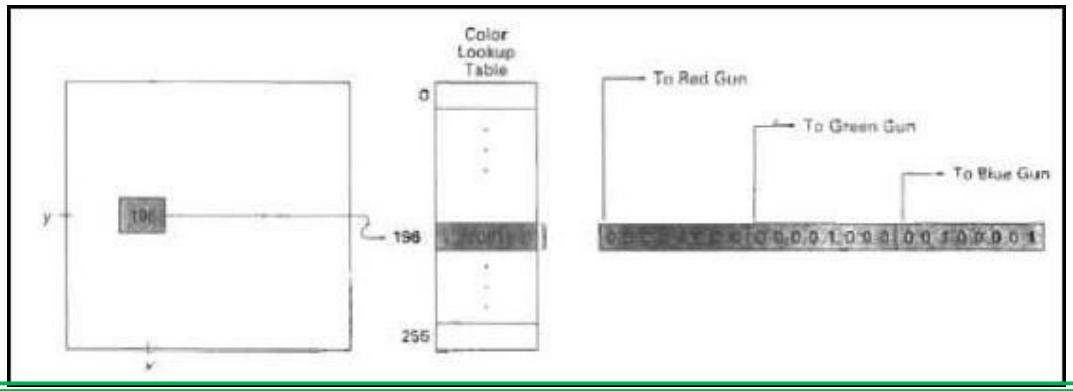
RGB system needs 3 megabytes of storage for the frame buffer. Color tables are an alternate means for providing extended color capabilities to a user without requiring large frame buffers. Lower-cost personal computer systems often use color tables to reduce frame-buffer storage requirements.

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:

The following figure illustrates a possible scheme for storing color values in a ***color lookup table*** (or ***video lookup table***), where frame-buffer values art- now used as indices into the color table.



TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

Systems employing this particular lookup table would allow a user to select any 256 colors for simultaneous display 17 million colors. Compared to a full-color system, this scheme reduces the number of simultaneous colors that can be displayed, but it also reduces the frame buffer storage requirements to 1 megabyte.

Advantages in storing color codes in a lookup table are,

Use of a color table can provide a "reasonable" number of simultaneous colors without requiring large frame buffers. For most applications, 256 or 512 different colors are sufficient for a single picture.

Table entries can be changed at any time, allowing a user to be able to experiment easily with different color combinations.

Visualization applications can store values for some physical quantity, such as energy, in the frame buffer and use a lookup table to try out various color encodings without changing the pixel values.

In visualization and image-processing applications, color tables are a convenient means for setting color thresholds so that all pixel values above or below a specified threshold can be set to the same color.

For these reasons, some systems provide both capabilities for color-code storage, so that a user can elect either to use color tables or to store color codes directly in the frame buffer.

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. This allows the intensity settings to be easily adapted to systems with differing grayscale capabilities.

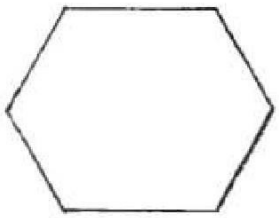
If additional bits per pixel are available in the frame buffer, the value of 0.33 would be mapped to the nearest level. With 3 bits per pixel, we can accommodate 8 gray levels; while 8 bits per pixel would give us 256 shades of gray.

An alternative scheme for storing the intensity information is to convert each intensity code directly to the voltage value that produces this grayscale level on the output device in use.

When multiple output devices are available at an installation, the same color-table interface may be used for all monitors. In this case, a color table for a monochrome monitor can be set up using a range of RGB values.

TABLE 4-2
INTENSITY CODES FOR A FOUR-LEVEL GRAYSCALE SYSTEM

Intensity Codes	Stored Intensity Values In The Frame Buffer (Binary Code)	Displayed Grayscale
0.0	0	
0.33	1	
0.67	2	
1.0	3	



AREA-FILL ATTRIBUTES:

Options for filling a defined region include a choice between a solid color or a patterned fill and choices for the particular colors and patterns. These fill options can be applied to polygon regions or to areas defined with curved boundaries, depending on the capabilities of the available package. In addition, areas can be painted using various brush styles, colors, and transparency parameters.

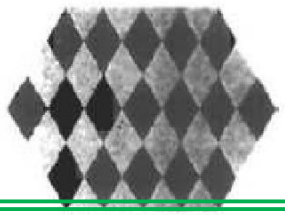
Fill Styles

Areas are displayed with three basic fill styles: ***hollow with a color border, filled with a solid color, or Wed with a specified pattern or design.***

A basic fill style is selected in a PHIGS program with the function

```
setInteriorStyle (fs)
```

values for the fill-style parameter fs include *hollow, solid,* and

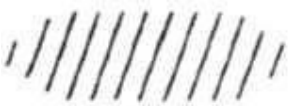


Another value for fill style is hatch, which is used to fill an area with selected hatching patterns- *parallel lines or crossed lines.*

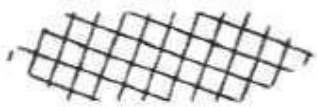
Fill selections for parameter fs are normally applied to polygon areas, but they can also be implemented to fill regions with curved boundaries.

Hollow areas are displayed using only the boundary outline, with the interior color the same as the background color. A ***solid fill*** is displayed in a single color up to and including the borders of the region. The color for a solid interior or for a hollow area outline is chosen with where fillcolor parameter fc is set to the desired color code.

We can display area edges dotted or dashed, fat or thin, and in any available color regardless of how we have filled the interior.



Diagonal
Hatch Fill



Diagonal
Cross-Hatch Fill

Pattern Fill

We select fill patterns with

```
setInteriorStyleIndex (pi)
```

where pattern index parameter pi specifies a table position.

Separate tables are set up for hatch patterns. If we had selected hatch fill for the interior style in this program segment, then the value assigned to parameter pi is an index to the stored patterns in the hatch table.

For fill style *pattern,* table entries can be created on individual output devices with

```
SetPatternRepresentation( ws, pi, nx, ny, cp)
```

Parameter pi sets the pattern index number for workstation code ws, and cp is a two-dimensional array of color codes with nx columns and ny rows.

When a color array cp is to be applied to fill a region, we need to specify the size of the area that is to be covered by each element of the array. We do this by setting the rectangular coordinate extents of the pattern:

```
setPatternSize (dx, dy)
```

where parameters dx and dy give the coordinate width and height of the array mapping.

A reference position for starting a *pattern* fill is assigned with the statement

```
setPatternReferencePoint (positicn)
```

Parameter *position* is a pointer to coordinates *(xp, yp)* that fix the lower left comer of the rectangular pattern.

A WORKSTATION PATTERN TABLE WITH TWO ENTRIES, USING THE COLOR CODES OF TABLE 4-1

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

Soft Fill:

Modified boundary-fill and flood-fill procedures that are applied to repaint areas so that the fill color is combined with the background colors are referred to as ***soft-fill*** or ***tint-fill*** algorithms.

One use for these fill methods is to soften the fill colors at object borders that have been blurred to antialias the edges.

Another is to allow repainting of a color area that was originally filled with a semitransparent brush, where the current color is then a mixture of the brush color and the background colors "behind" the area.

In either case, we want the new fill color to have the same variations over the area as the current fill color.

As an example of this type of fill, the linear soft-fill algorithm repaints an area that was originally painted by merging a foreground color F with a single background color B, where **F ≠ B**.

Assuming we know the values for F and B, we can determine how these colors were originally combined by checking the current color contents of the frame buffer. The current RGB color P of each pixel within the area to be refilled is some linear combination of F and B:

$$\mathbf{P = tF + (1 - t)B}$$

where the "transparency" factor t has a value betwe en 0 nad 1 for each pixel.

The above vector equation holds for each RGB component of the colors, with

$$P = (P_R, P_C, P_R),\ F = (F_R, Fc, F_R),\ B = (B_R, Bc, B_B)$$

We can thus calculate the value of parameter *f* using one of the RGB color components as

$$t = \frac{P - B}{kk}$$

where k = R, G, or B and $F_k \neq B_k$.

Similar soft-fill procedures can be applied to an area whose foreground color is to be merged with multiple background color areas, such as a checkerboard pattern.

CHARACTER ATTRIBUTES

The appearance of displayed characters 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:

There are a great many text options that can be made available to graphics programmers. First of all, there is the choice of font (or typeface), which is a set of characters with a particular design style such as New York, Courier, Helvetica, London, Times Roman, and various special symbol groups.

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

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

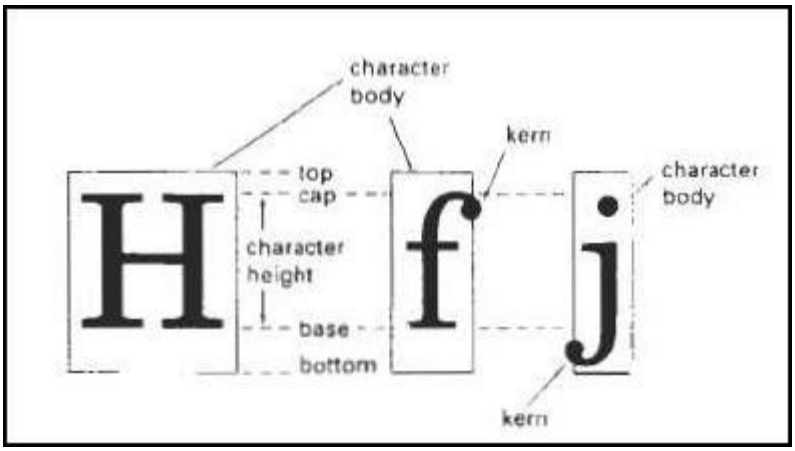
```
setTextFont (tf)
```

Color settings for displayed text are stored m the system attribute list and used by the procedures that load character definitions into the frame buffer. When a character string is to be displayed, the current color is used to set pixel values in the frame buffer corresponding to the character shapes and positions. 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.

We can adjust text size by scaling the overall dimensions (height and width) of characters or by scaling only the character width. Character size is specified by printers and compositors in *points,* where 1 point is 0.013837 inch (or approximately 1/72 inch).



The distance between the *bottomline* and the *topline* of the character body is the same for all characters in a particular size and typeface, but the body width may vary. *Proportionally spaced fonts* assign a smaller body width to narrow characters such as i, j, 1, and f compared to broad characters such as W or M. *Character height* is defined as the distance between the *baseline* and the *capline* of characters.

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

```
setCharacterHeight (ch)
```

The width only of text can be set with the function

```
setCharacterExpansionFactor (cw)
```

Marker Attribute:

A marker symbol is a single character that can he displayed in different colors and in different sizes. 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). Displayed marker types are centered on the marker coordinates. We set the marker size with

```
setMarkerSizeScaleFactor (ms)
```

with parameter marker size ms assigned a positive number.

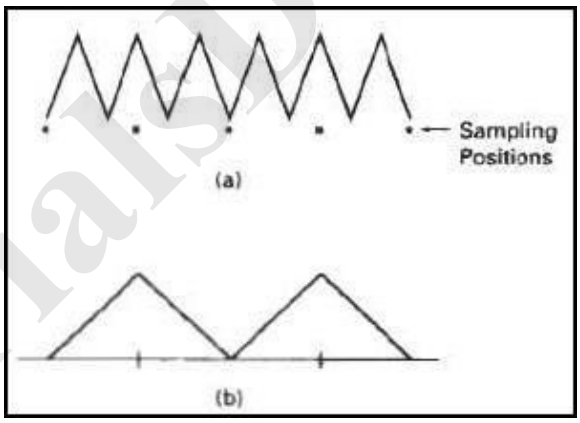
ANTIALIASING:

Displayed primitives generated by the raster algorithms have a jagged, or stairstep, appearance because the sampling process digitizes coordinate pints on an object to discrete integer pixel positions. This distortion of information due to low-frequency sampling (undersampling) is called aliasing.

We can improve the appearance of displayed raster lines by applying antialiasing methods that compensate for the undersampling process.

To avoid losing information from such periodic objects, we need to set the sampling frequency to at least twice that of the highest frequency occurring in the object, referred to as the ***Nyquist sampling frequency*** (or Nyquist sampling rate) f_s:

$$f_s = 2 f_{max}$$



Another way to state this is that the sampling interval should be no larger than one-half the cycle interval (called the Nyquist sampling interval). For x-interval sampling, the Nyquist sampling interval Ax, is

$$\Delta x_s = \frac{\Delta x_{cycle}}{2}$$

where $\Delta x_{cycle} = 1 / f_{max}$.

One way to increase sampling rate with raster systems is simply to display objects at higher resolution.

A straightforward antialiasing method is to increase sampling rate by treating the screen as if it were covered with a finer grid than is actually available. We can then use multiple sample points across this finer grid to determine an appropriate intensity level for each screen pixel. This technique of sampling object characteristics at a high resolution and displaying the results at a lower resolution is called ***supersampling*** (or ***postfiltering***, since the general method involves computing intensities, it subpixel grid positions, then combining the results to obtain the pixel intensities).

An alternative to supersampling is to determine pixel intensity by calculating the areas of overlap of each pixel with the objects to be displayed. Antialiasing by computing overlap areas is referred to as ***area sampling*** (or ***prefiltering***, since the intensity of the pixel as a whole is determined without calculating subpixel intensities). Pixel overlap areas are obtained by determining where object boundaries intersect individual pixel boundaries.

Raster objects can also be antialiased by shifting the display location of pixel areas. This technique, called ***pixel phasing***, is applied by "*micropositioning*" the electron beam in relation to object geometry.

Another advantage of supersampling with a finite-width line is that the total line intensity is distributed over more pixels.

2D TRANSFORMATIONS

BASIC TRANSFORMATIONS

Changes in orientation, size, and shape are accomplished with geometric transformations that alter the coordinate descriptions of objects. The basic geometric transformations are translation, rotation, and scaling. Other transformations that are often applied to objects include reflection and shear.

TRANSLATION

A ***translation*** is applied to an object by repositioning it along a straight-line path from one coordinate location to another. We translate a two-dimensional point by adding ***translation distances***, t_x and t_y, to the original coordinate position ***(x, y)*** to move the point to a new position ***(x', y')***.

$$x' = x + t_x \quad , \quad y' = y + t_y$$

The translation distance pair (t_x, t_y) is called a ***translation vector*** or ***shift vector***.

We can express the translation equations as a single matrix equation by using column vectors to represent coordinate positions and the translation vector:

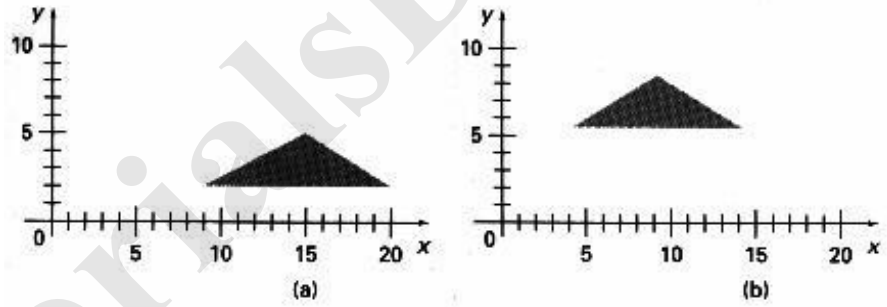
$$P = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad P' = \begin{bmatrix} x_1' \\ x_2' \end{bmatrix}, \qquad T = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

y

This allows us to write the two-dimensional translation equations in the matrix form:

$$P' = P + T$$

Sometimes matrix-transformation equations are expressed in terms of coordinate row vectors instead of column vectors. In this case, we would write the matrix representations as **P = [x y]** and **T = [t_x ty].**



Translation is a ***rigid-body transformation*** that moves objects without deformation, i.e., every point on the object is translated by the same amount.

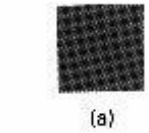
Polygons are translated by adding the translation vector to the coordinate position of each vertex and regenerating the polygon using the new set of vertex coordinates and the current attribute settings.

Similar methods are used to translate ***curved*** objects. To change the position of a circle or ellipse, we translate the center coordinates and redraw the figure in the new location. We translate other curves (splines) by displacing the coordinate positions defining the objects, and then we reconstruct the curve paths using the translated coordinate points.

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)*** of each vertex by ***scaling factors*** s_x and s_y to produce the transformed coordinates ***(x', y'):***

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



(a)

(b)

Scaling factor s_x, scales objects in the ***x*** direction, while s_y scales in the ***y*** direction. The transformation equations can be written in the matrix form as,

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

$$P' = S \cdot P$$

where S is the 2 by 2 scaling matrix.

Specifying a value of 1 for both s_x and s_y leaves the size of objects unchanged.

When s_x and s_y are assigned the same value, a ***uniform scaling*** is produced that maintains relative object proportions.

Unequal values for s_x and s_y result in a ***differential scaling*** that are often used in design applications, when pictures are constructed from a few basic shapes that can be adjusted by scaling and positioning transformations.

We can control the location of a scaled object by choosing a position, called the ***fixed point*** that is to remain unchanged after the scaling transformation.

Coordinates for the fixed point ***(xf , yf)*** can be chosen as one of the vertices, the object centroid, or any other position. A polygon is then scaled relative to the fixed point by scaling the distance from each vertex to the fixed point.

For a vertex with coordinates **(x, y)**, the scaled coordinates ***(x', y')*** are calculated as,

$$x' = x_f + (x - x_f)\, s_x \quad , \qquad y' = y_f + (y - y_f)\, s_y$$

We can rewrite these scaling transformations to separate the multiplicative and additive terms:

$$x' = x \cdot s_x + x_f (1 - s_x)$$

$$y' = y \cdot s_y + y_f (1 - s_y)$$

where the additive terms $x_f(1 - s_x)$ and $y_f(1 - s_y)$ are constant for all points in the object.

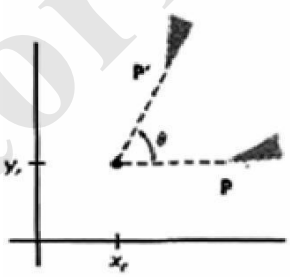
ROTATION

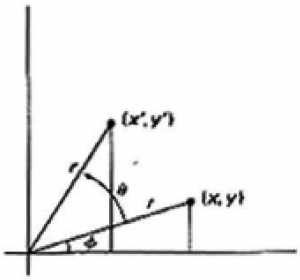
A two-dimensional rotation is applied to an object by repositioning it along a circular path in the **xy** plane. To generate a rotation, we specify a rotation angle 0 and the position ***(x, y)*** of the rotation point (or pivot point) about which the object is to be rotated.

Positive values for the rotation angle define counterclockwise rotations. Negative values rotate objects in the clockwise direction.

This transformation can also be described as a rotation about a rotation axis that is perpendicular to the **xy** plane and passes through the pivot point.

We first determine the transformation equations for rotation of a point position **P** when the pivot point is at the coordinate origin. The angular and coordinate relationships of the original and transformed point positions are shown in the diagram.





In this figure, r is the constant distance of the point from the origin, angle ϕ is the original angular position of the point from the horizontal, and θ is the rotation angle.

Using standard trigonometric identities, we can express the transformed coordinates in terms of angles θ and ϕ as

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

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

The original coordinates of the point in polar coordinates are,

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

Substituting expressions 2nd into 1st, we obtain the transformation equations for rotating a point at position ***(x, y)*** through an angle θ about the origin:

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

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

We can write the rotation equations in the matrix form:

$$P' = R \cdot P$$

where the rotation matrix is

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

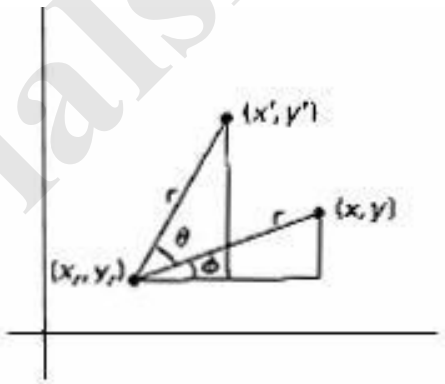
When coordinate positions are represented as row vectors instead of column vectors, the matrix product in rotation equation is transposed so that the transformed row coordinate vector [x' y'] calculated as,

$$P'^T = (R \cdot P)^T$$

$$= P^T \cdot R^T$$

where $\mathbf{P'^T}$ = **[x y],** and the transpose $\mathbf{R^T}$ of matrix R is obtained by interchanging rows and columns. For a rotation matrix, the transpose is obtained by simply changing the sign of the sine terms.

Rotation of a point about an arbitrary pivot position is illustrated in the following diagram.



Using the trigonometric relationships in this figure, we can generalize to obtain the transformation equations for rotation of a point about any specified rotation position **(x_r, y_r):**

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

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

As with translations, rotations are rigid-body transformations that move objects without deformation. Every point on an object is rotated through the same angle.

Polygons are rotated by displacing each vertex through the specified rotation angle and regenerating the polygon using the new vertices. Curved lines are rotated by repositioning the defining points and redrawing the curves.

A circle or an ellipse, for instance, can be rotated about a noncentral axis by moving the center position through the arc that subtends the specified rotation angle.

An ellipse can be rotated about its center coordinates by rotating the major and minor axes.

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 design and picture construction applications, we perform translations, rotations, and scalings to tit the picture components into their proper positions.

Each of the basic transformations can be expressed in the general *matrix form*

$$\mathbf{P' = M_1 . P + M_2}$$

with coordinate positions P and P' represented as column vectors.

Matrix **M1** is a 2 by 2 array containing multiplicative factors, and **M2** is a two-element column matrix containing translational terms.

For translation, **M1** is the identity matrix.

For rotation, **M2** contains the translational terms associated with the pivot point. For scaling, **M2** contains the translational terms associated with the fixed point.

To produce a sequence of transformations with these equations, such as scaling followed by rotation then translation, we must calculate the transformed coordinate one step at a time.

To express any two-dimensional transformation as a matrix multiplication, we represent each Cartesian coordinate position *(x, y)* with the homogeneous coordinate triple *(xh, yh, h)* where

$$x = \frac{x_h}{h}, \quad y = \frac{y_h}{h}$$

Thus, a general homogeneous coordinate representation can also be written as *(h.x, h.y, h)*.

For two-dimensional geometric transformations, we can choose the homogeneous parameter *h* to be any nonzero value. A convenient choice is simply to set *h = 1*.

Each two-dimensional position is then represented with homogeneous coordinates *(x, y, 1)*.

The term ***homogeneous coordinates*** is used in mathematics to refer to the effect of this representation on Cartesian equations.

When a Cartesian point *(x, y)* is converted to a homogeneous representation *(xh, yh, h)* equations containing *x* and *y* such as *f (x, y) = 0*, become homogeneous equations in the three parameters *xh, yh* and *h.*

Expressing positions in homogeneous coordinates allows us to represent all geometric transformation equations as matrix multiplications. Coordinates are represented with three-element column vectors, and transformation operations are written as 3 by 3 matrices.

For Translation,

we have

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

which we can write in the abbreviated form

$$\mathbf{P' = T}\,(t_x, t_y)\,\mathbf{.\ P}$$

with **T** (t_x, t_y) as the 3 by 3 translation matrix.

The inverse of the translation matrix is obtained by replacing the translation parameters t_x and t_y with their negatives $-t_x$ and $-t_y$.

Similarly, ***Rotation Transformation*** equations about the coordinate origin are written as

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

or as

P' = R (θ) . P

The rotation transformation operator R (θ) is the 3 by 3 matrix with rotation parameter θ. We get the inverse rotation matrix when θ is replaced with -θ.

A ***Scaling Transformation*** relative to the coordinate origin is now expressed as the matrix or

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

P' = S (s_x, s_y) . P

where **S (s_x, s_y)** is the 3 by 3 matrix with parameters s_x and s_y.

Replacing these parameters with their multiplicative inverses **(1/s_x** and ***1/s_y*)** yields the inverse scaling matrix.

Matrix representations are standard methods for implementing transformations in graphics systems. Rotations and Scalings relative to other reference positions are then handled as a succession of transformation operations.

An alternate approach in a graphics package is to provide parameters in the transformation functions for the scaling fixed-point coordinates and the pivot-point coordinates

COMPOSITE TRANSFORMATIONS

With the matrix representations, 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. Forming products of transformation matrices is often referred to as a ***concatenation***, or ***composition***, of matrices.

For column-matrix representation of coordinate positions, we form composite transformations by multiplying matrices in order from right to left, i.e., each successive transformation matrix premultiplies the product of the preceding transformation matrices.

Translations:

If two successive translation vectors **(t_{x1}, t_{y1})** and **(t_{x2}, t_{y2})** are applied to a coordinate position **P**, the final transformed location **P'** is calculated as

$$\mathbf{P'} = \mathbf{T}(t_{x2}, t_{y2}) \cdot \{\mathbf{T}(t_{x1}, t_{y1}) \cdot \mathbf{P}\}$$

$$= \{\mathbf{T}(t_{x2}, t_{y2}) \cdot \mathbf{T}(t_{x1}, t_{y1})\} \cdot \mathbf{P}\}$$

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

Also, the composite transformation matrix for this sequence of translations is

Or

$$\begin{bmatrix} 1 & 0 & t_{x2} \\ 0 & 1 & t_{y2} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{x1} \\ 0 & 1 & t_{y1} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{x1} + t_{x2} \\ 0 & 1 & t_{y1} + t_{y2} \\ 0 & 0 & 1 \end{bmatrix}$$

$\mathbf{T}(t_{x2}, t_{y2}) \cdot \mathbf{T}(t_{x1}, t_{y1}) = \mathbf{T}(t_{x1} + t_{x2}, t_{y1} + t_{y2})$ which demonstrates that two successive translations are additive.

x

Rotations:

Two successive rotations applied to point **P** produce the transformed position

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

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

$$\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1) = \mathbf{R}(\theta_1 + \theta_2)$$

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

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

Scaling:

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

$$\begin{bmatrix} s_{x2} & 0 & 0 \\ 0 & s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{x1} & 0 & 0 \\ 0 & s_{y1} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{x1} \cdot s_{x2} & 0 & 0 \\ 0 & s_{y1} \cdot s_{y2} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

or

$$\mathbf{S}(s_{x2}, s_{y2}) \cdot \mathbf{S}(s_{x1}, s) = \mathbf{S}(s_{x1} \cdot s_{x2}, s_{y1} \cdot s_{y2})$$

The resulting matrix in this case indicates that successive scaling operations are multiplicative.

General Pivot-Point Rotation:

With a graphics package that only provides a rotate function for revolving objects about the coordinate origin, we can generate rotations about any selected pivot point (x_r, y_r) by performing the following sequence of translate-rotate-translate operations:

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

This transformation sequence is illustrated in the following diagram. The composite transformation matrix for this sequence is obtained with the concatenation.

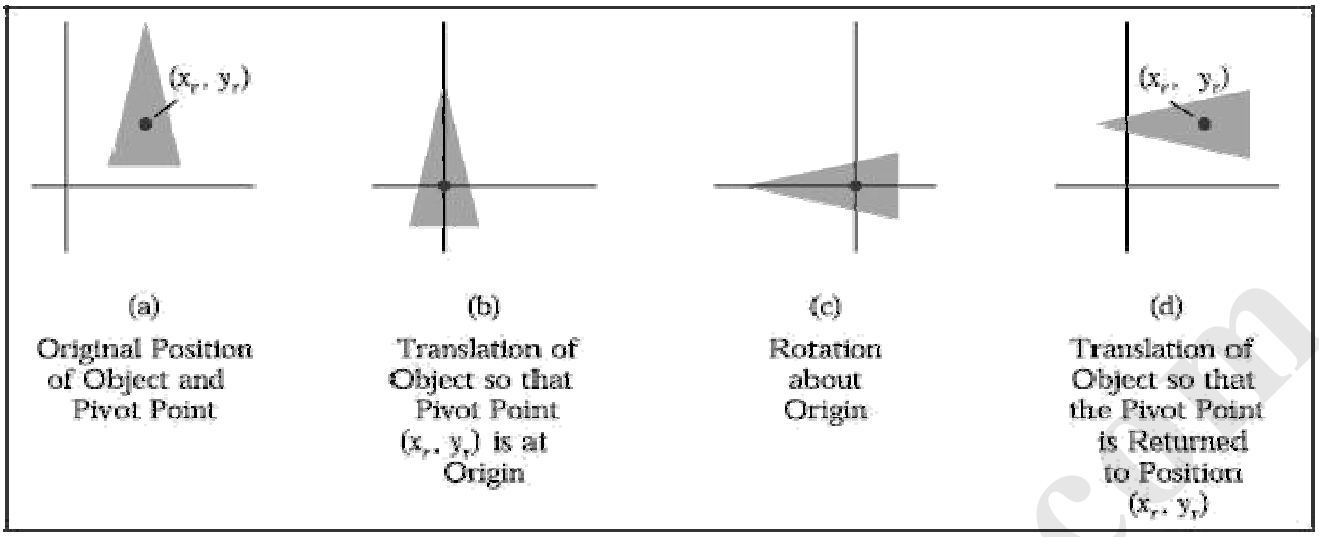
$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

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

which can be expressed in the form

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

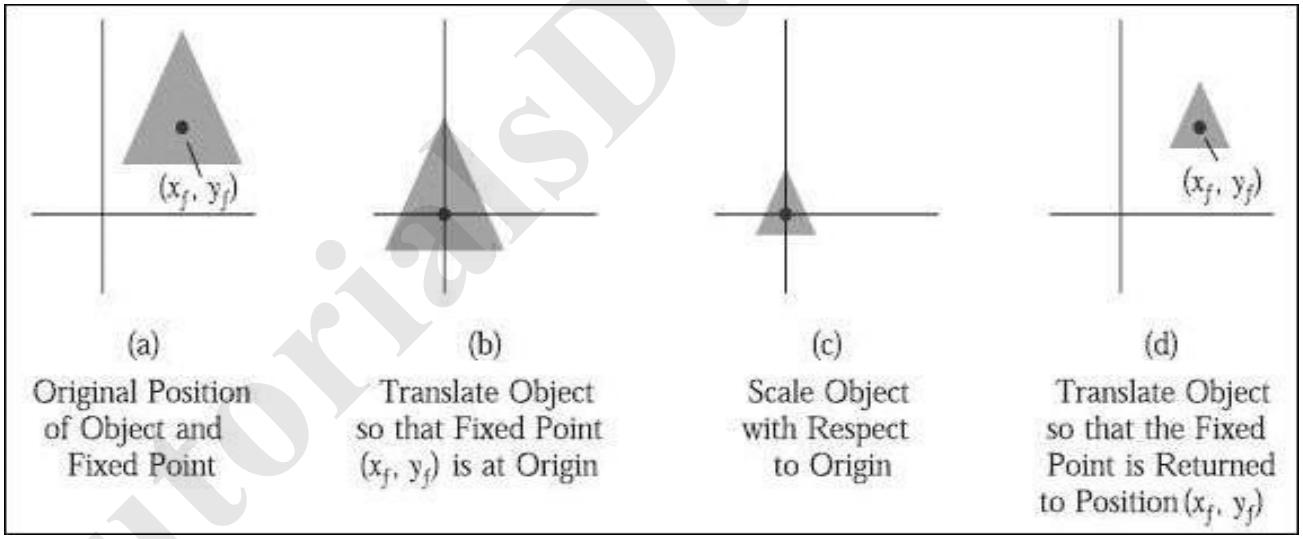
where $\mathbf{T}(-x_r, -y_r) = \mathbf{T}^{-1}(x_r, y_r)$



General Fixed-Point Scaling:

The following diagram illustrates a transformation sequence to produce scaling with respect to a selected fixed position ***(xf , yf)*** using a scaling function that can only scale relative to the coordinate origin.

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



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

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

or

$$\mathbf{T}(xf, yf) \cdot \mathbf{S}(sx, sy) \cdot \mathbf{T}(-xf, -yf) = \mathbf{S}(xf, yf, sx, sy)$$

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

OTHER TRANSFORMATIONS:

Some other additional transformations are ***reflection*** and ***shear***.

Reflection:

A ***reflection*** is a transformation that produces a mirror image of an object. The mirror image for a two-dimensional reflection is generated relative to an ***axis of reflection*** by rotating the object 180" about the reflection axis. Some common reflections are as follows:

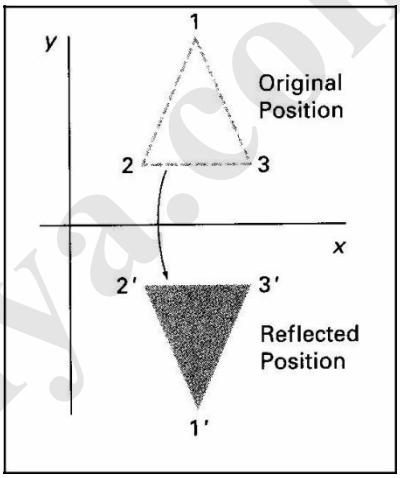
x-Reflection:

Reflection about the line ***y = 0***, the ***x*** axis, is accomplished with the transformation Matrix

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

This transformation keeps ***x*** values the same, but "flips" the ***y*** values of coordinate positions.

The resulting orientation of an object after it has been reflected about the ***x*** axis is shown in the diagram.



y-Reflection:

A reflection about the y axis flips ***x*** coordinates while keeping ***y*** coordinates the same.

The matrix for this transformation is,

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

The diagram illustrates the change in position of an object that has been reflected about the line ***x = 0***.

Origin-Reflection:

We flip both the ***x*** and ***y*** coordinates of a point by reflecting relative to an axis that is perpendicular to the ***xy*** plane and that passes through the coordinate origin.

This transformation, referred to as a reflection relative to the coordinate origin, has the matrix representation:

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

If we chose the reflection axis as the diagonal line $y = x$, the reflection matrix is

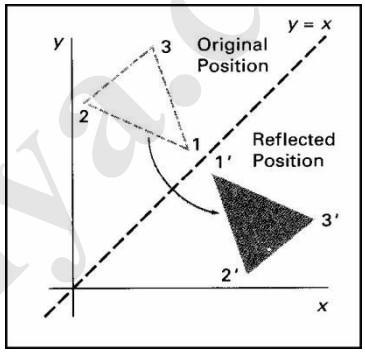
$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To obtain a transformation matrix for reflection about the diagonal $y = -x$, we could concatenate matrices for the transformation sequence:

(1) clockwise rotation by 45°,
(2) reflection about the y axis, and
(3) counterclockwise rotation by 45°.

The resulting transformation matrix is

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



Reflections about any line $y = mx + h$ in the xy plane can be accomplished with a combination of translate-rotate-reflect transformations.

Shear:

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

Two common shearing transformations are those that shift coordinate *x* values and those that shift *y* values.

x-Shearing:

An *x*-direction shear relative to the x axis is produced with the transformation matrix

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

which transforms coordinate positions as

$$x' = x + sh_x \cdot y, \; y' = y$$

In the following diagram, $sh_x = 2$, changes the square into a parallelogram.

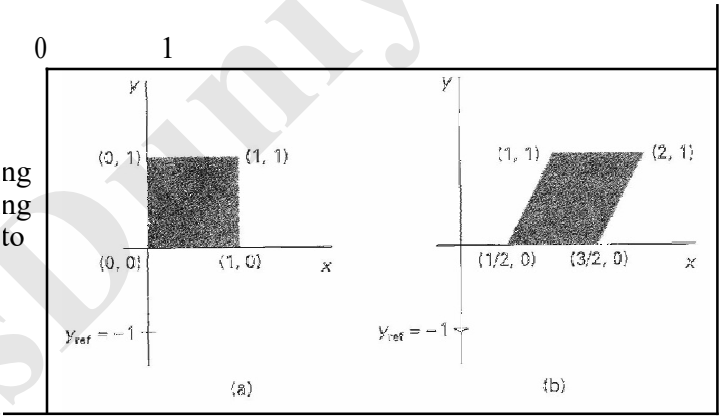
Negative values for sh_x shift coordinate positions to the left. We can generate x-direction shears relative to other reference lines with

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

with coordinate positions transformed as

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

An example of this shearing transformation is given in the following diagram for a shear parameter of ½ relative to the line $y_{ref} = -1$.



y-Shearing:

A y-direction shear relative to the line $x = x_{ref}$ is generated with the transformation matrix

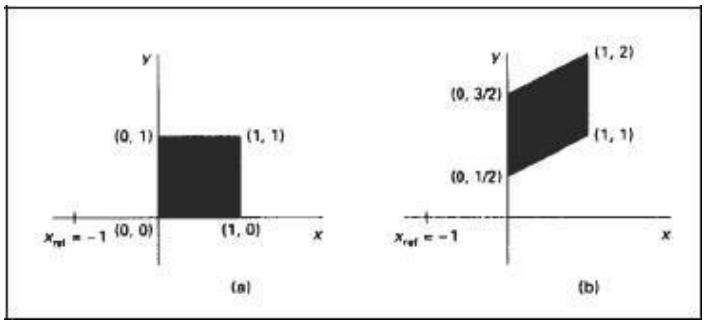
$$\begin{bmatrix} 1 & 0 & 0 \\ sh_y & 1 & -sh_y \cdot x_{ref} \\ 0 & 0 & 1 \end{bmatrix}$$

which generates transformed coordinate positions

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

This transformation shifts a coordinate position vertically by an amount proportional to its distance from the reference line $x = x_{ref}$.

The diagram shows the conversion of a square into a parallelogram with $sh_y = ½$ and $x_{ref} = -1$.



2D VIEWING AND CLIPPING

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 ***viewport***.

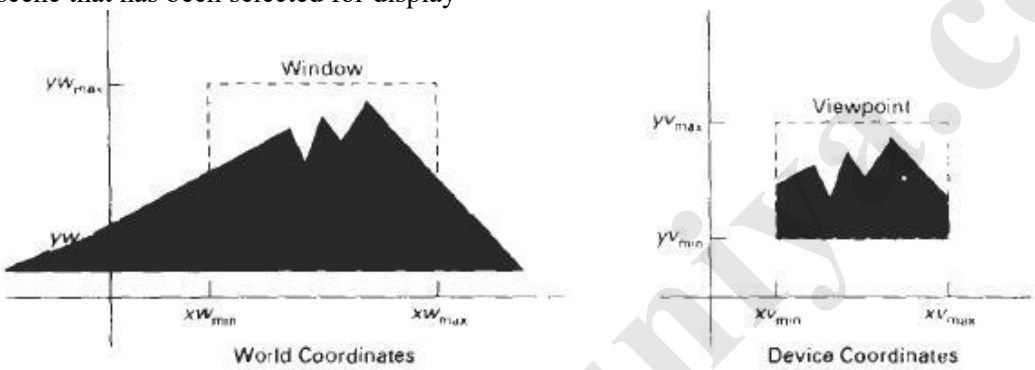
The window defines ***what*** is to be viewed.

The viewport defines ***where*** it is to be displayed.

Often, windows and viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes.

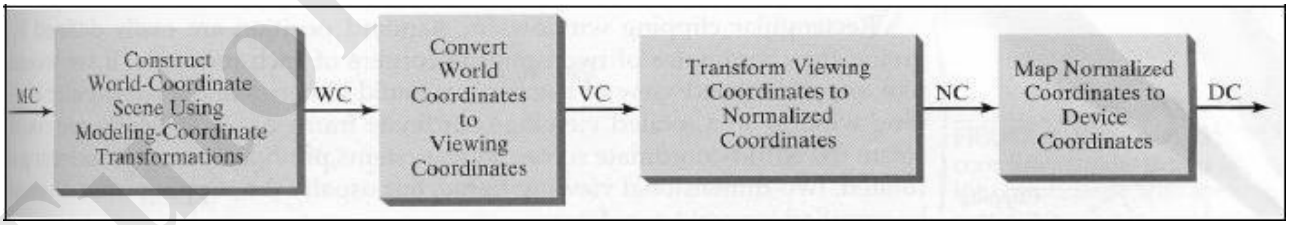
In general, the mapping of a part of a world-coordinate scene to device coordinates is referred to as a ***viewing transformation***.

Sometimes the two-dimensional viewing transformation is simply referred to as the ***window-to-viewport transformation*** or the ***windowing transformation.*** The term window to refer to an area of a world-coordinate scene that has been selected for display

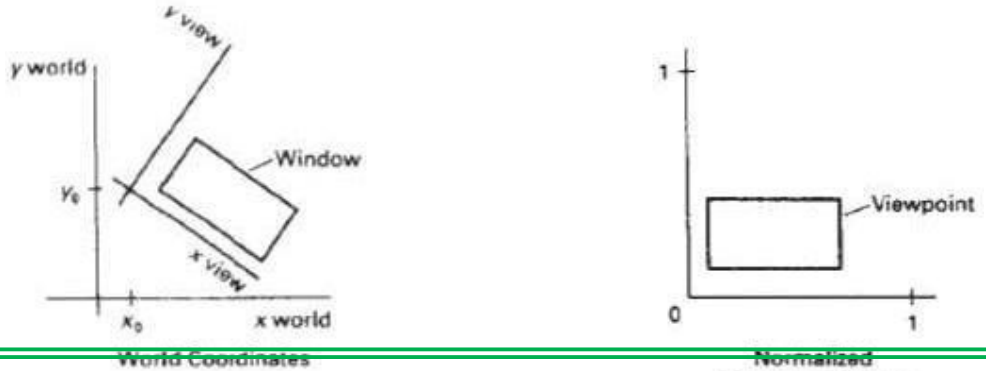


We carry out the viewing transformation in several steps, as indicated below.

1. First, we construct the scene in world coordinates using the output primitives and attributes.
2. 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.
3. 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.
4. 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.
5. 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.



The following diagram illustrates a rotated viewing-coordinate reference frame and the mapping to



normalized coordinates.

By changing the position of the viewport, we can view objects at different positions on the display area of an output device. Also, by varying the size of viewports, we can change the size and proportions of displayed objects. We achieve zooming effects by successively mapping different-sized windows on a fixed-size viewport.

Panning effects are produced by moving a fixed-size window across the various objects in a scene. When all coordinate transformations are completed, viewport clipping can be performed in normalized coordinates or in device coordinates. This allows us to reduce computations by concatenating the various transformation matrices.

VIEWING COORDINATE REFERENCE FRAME:

This coordinate system provides the reference frame for specifying the world coordinate window. First, a viewing-coordinate origin is selected at some world position: $P_0 = (x_0, y_0)$. Then we need to establish the orientation, or rotation, of this reference frame. One way to do this is to specify a world vector V that defines the viewing y_0, direction. Vector V is called the ***view up vector***.

Given V, we can calculate the components of unit vectors $v = (v_x, v_y)$ and $u = (u_x, u_y)$ for the viewing y_v and x_v axes, respectively. These unit vectors are used to form the first and second rows of the rotation matrix R that aligns the viewing $x_v y_v$ axes with the world $x_w y_w$ axes.

We obtain the matrix for converting world coordinate positions to viewing coordinates as a two-step composite transformation:

First, we translate the viewing origin to the world origin,
Then we rotate to align the two coordinate reference frames.

The composite 2D transformation to convert world coordinates to viewing coordinate is

$$M_{WC,VC} = R \cdot T$$

where T is the translation matrix that takes the viewing origin point P_0 to the world origin, and R is the rotation matrix that aligns the axes of the two reference frames.

A viewing-coordinate frame is moved into coincidence with the world frame in two steps: (a) translate the viewing origin to the world origin, then (b) rotate to align the axes of the two systems.

WINDOW-TO-VIEWPORT COORDINATE TRANSFORMATION

A point at position (xw, yw) in a designated window is mapped to viewport coordinates (xv, yv) so that relative positions in the two areas are the same.

Once object descriptions have been transferred to the viewing reference frame, we choose the window extents in viewing coordinates and select the viewport limits in normalized coordinates. Object descriptions are then transferred to normalized device coordinates. We do this using a transformation that maintains the same relative placement of objects in normalized space as they had in viewing coordinates. If a coordinate position is at the center of the viewing window, for instance, it will be displayed at the center of the viewport.

The above diagram illustrates the window-to-viewport mapping. A point at position ***(xw, yw)*** in the window is mapped into position ***(xv, yv)*** in the associated viewport. To maintain the same relative placement in the viewport as in the window, we require that,

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

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

Solving these expressions for the viewport position ***(xv, yv),*** we have

$$xv = xv_{min} + (xw - xw_{min})sx$$

$$yv = yv_{min} + (yw - yw_{min})sy$$

where the scaling factors are

$$sx = \frac{xv_{max} - xv_{min}}{xw_{max} - xw_{min}}$$

$$sy = \frac{yv_{max} - yv_{min}}{yw_{max} - yw_{min}}$$

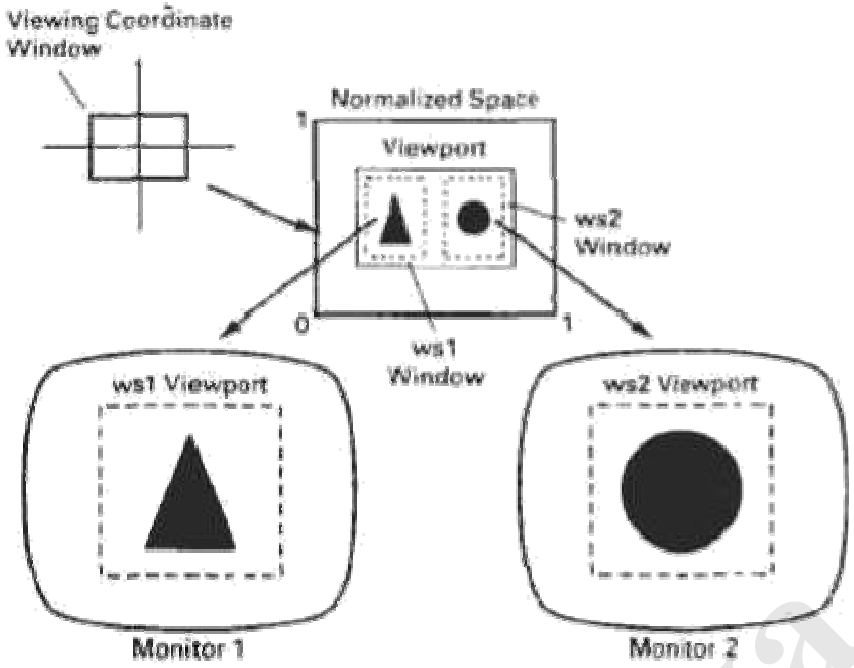
The Equations can also be derived with a set of transformations that converts the window area into the viewport area.

This conversion is performed with the following sequence of transformations:

1. Perform a scaling transformation using a fixed-point position of *(xw_{min}, yw_{min})* that scales the window area to the size of the viewport.
2. Translate the scaled window area to the position of the viewport.

Relative proportions of objects are maintained if the scaling factors are the same ***(sx = sy).*** Otherwise, world objects will be stretched or contracted in either the ***x*** or ***y*** direction when displayed on the output device.

The mapping, called the ***workstation transformation***, is accomplished by selecting a window area in normalized space and a viewport area in the coordinates of the display device.



Mapping selected parts of a scene in normalized coordinates to different video monitors with workstation transformations.

2D CLIPPING:

CLIPPING OPERATIONS

Generally, any procedure that identifies those portions of a picture that are either inside or outside of a specified region of space is referred to as a ***clipping algorithm***, or ***simply clipping***. The region against which an object is to clip is called a ***clip window.***

Applications of clipping include extracting part of a defined scene for viewing; identifying visible surfaces in three-dimensiona1 views; antialiasing line *seg*ments or object boundaries; creating objects using solid-modeling procedures; displaying a multiwindow environment; and drawing and painting operations that allow parts of a picture to be selected for copying, moving, erasing, or duplicating.

Clipping algorithms can be applied in world coordinates, so that only the contents of the window interior are mapped to device coordinates.

World-coordinate clipping removes those primitives outside the window from further consideration, thus eliminating the processing necessary to transform those primitives to device space.

Viewport clipping, can reduce calculations by allowing concatenation of viewing and geometric transformation matrices.

We consider algorithms for clipping the following primitive types

Point Clipping	Curve Clipping
Line Clipping (straight-line segments)	Text Clipping
Area Clipping (polygons)	

POINT CLIPPING

Assuming that the clip window is a rectangle in standard position, we save a point ***P = (x, y)*** for display if the following inequalities are satisfied:

$$xw_{min} \le x \le xw_{max}$$
$$yw_{min} \le y \le yw_{max}$$

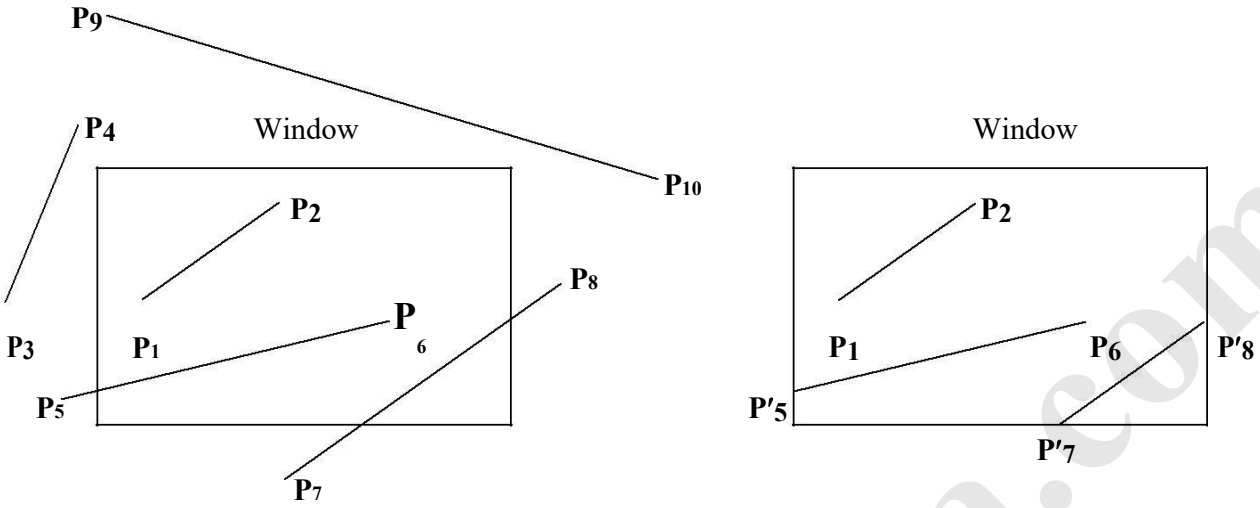
where the edges of the clip window ***(xwmin, xwmax, ywmin, ywmax)*** can be either the world-coordinate window boundaries or viewport boundaries. If any one of these four inequalities is not satisfied, the point is clipped (not saved for display).

LINE CLIPPING:

Explain the steps to perform Line Clipping. (Or) (5, 10 Marks)
Describe Cohen-Sutherland Line Clipping algorithm in detail.

Explain Liang-Barsky Line Clipping algorithm.

The following diagram illustrates the possible relationships between line positions and a standard rectangular clipping region.



(a) Before clipping *(b) After clipping*

A line clipping procedure involves several parts.

First, we can test a given line segment to determine 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 cannot identify a line as completely inside or completely outside, we must perform intersection calculations with one or more clipping boundaries.

We process lines through the "inside-outside" test by checking the line endpoints.

A line with both endpoints inside all clipping boundaries, such as the line from P_1 to P_2 is saved.

A line with both endpoints outside any one of the clip boundaries (line P_3P_4 in the diagram) is outside the window.

All other lines ***cross*** one or more clipping boundaries, and may require calculation of multiple intersection points.

For a line segment with endpoints (x_1, y_1) and (x_2, y_2) and one or both endpoints outside the clipping rectangle, the parametric representation

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

could be used to determine values of parameter ***u*** for intersections with the clipping boundary coordinates.

If the value of ***u*** for an intersection with a rectangle boundary edge is outside the range ***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 turn to determine whether any part of the line segment is to be displayed.

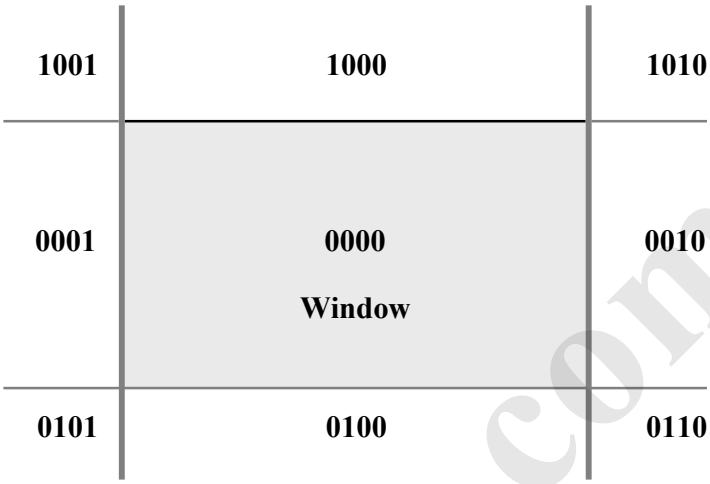
Cohen-Sutherland Line Clipping:

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

Every line end-point 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. Regions are set up in reference to the boundaries as shown below:.

Each bit position in the region code is used to indicate one of the four relative coordinate positions of the point with respect to the 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 can be correlated with the bit positions as

bit 1: left
bit 2: right
bit 3: below
bit 4: 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.**

The region-code bit values can be determined with the following two steps:

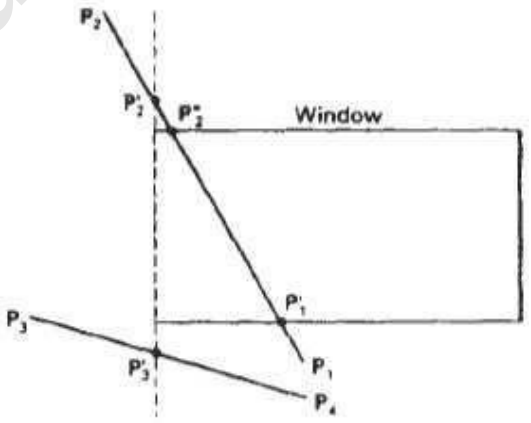
(1) Calculate differences between endpoint coordinates and clipping boundaries.

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

- Any lines that are completely contained within the window boundaries have a region code of 0000 for both endpoints, and we trivially 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 trivially reject these lines.
- 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.

.

To illustrate the specific steps in clipping lines against rectangular boundaries using the Cohen-Sutherland algorithm, we show how the lines in the above diagram could be processed.



Starting with the bottom endpoint of the line from **P1** to **P2,** we check **P1**, 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 **P1** to **P′1.** The line now has been reduced to the section from **P′1** to **P2.** Since **P2** 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.

Intersection points with a clipping boundary can be calculated using the slope-intercept form of the line equation. For a line with endpoint coordinates ***(x1 ,y1)*** and ***(x2 , y2),*** the *y* coordinate of the intersection point with a vertical boundary can be obtained with the calculation

$$y = y1 + m\ (x - x\ 1)$$

where the ***x*** value is set either to ***xwmin*** or to ***xwmax*** and the slope of the line is calculated as

$$m = (y2 - y1) / (x1 - x2).$$

Similarly, if we are looking for the intersection with a horizontal boundary, the ***x*** coordinate can be calculated as

$$x = x_1 + \frac{y - y_1}{m}$$

with y set either to y***wmin*** or to y***wmax*** .

Liang-Barsky Line Clipping:

Faster line clippers have been developed that are based on analysis of the parametric equation of a line segment, which we can write in the form

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

where $\Delta x = x_2 - x_1$, and $\Delta y = y_2 - y_1$.

Liang and Barsky independently devised an even faster parametric line-clipping algorithm. In this approach, we first write the point-clipping conditions in the parametric form:

$$xw_{min} \le x1 + u\,\Delta x \le xw_{max}$$
$$yw_{min} \le y1 + u\,\Delta y \le yw_{max}$$

Each of these four inequalities can be expressed as

$$u\,p_k \le q_k\,,\ k = 1, 2, 3, 4$$

where parameters p and q are defined as

$$p_1 = -\Delta x, \qquad q_1 = x_1 - xw_{min}$$
$$p_2 = \Delta x, \qquad q_2 = xw_{maz} - x_1$$
$$p_3 = -\Delta y, \qquad q_3 = y_1 - yw_{min}$$
$$p_4 = \Delta y, \qquad q_4 = yw_{maz} - y_1$$

Any line that is parallel to one of the clipping boundaries has $p_k = 0$ for the value of k corresponding to that boundary ($k = 1, 2, 3,$ and 4 correspond to the left, right, bottom, and top boundaries, respectively).

If, for that value of k, we also find $q_k < 0$, then the line is completely outside the boundary and can be eliminated from further consideration. If $q_k \ge 0$, the line is inside the parallel clipping boundary.

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

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

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

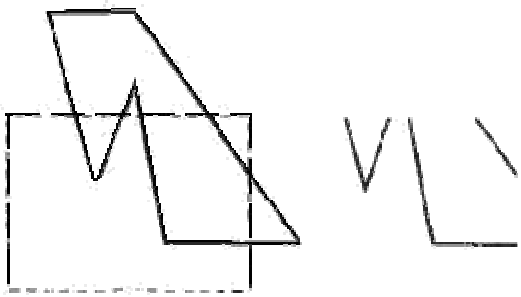
For each line, we can calculate values fur parameters u_1 and u_2 that define that part of the line that lies within the clip rectangle. The value of u_1 is determined by looking at the rectangle edges for which the line proceeds from the outside to the inside ($p < 0$). For these edges, we calculate $r_k = q_k / p_k$. The value of u_1 is taken as the largest of the set consisting of 0 and the various values of r. Conversely, the value of u_2 is determined by examining the boundaries for which the line proceeds from inside to outside ($p > 0$).

A value of r_k is calculated for each of these boundaries, and the value of **u**, is the minimum of the set consisting of 1 and the calculated r values. If $u_1 > u_2$, the line is completely outside the clip window and it can be rejected. Otherwise, the endpoints of the clipped line are calculated from the two values of parameter *u*.

POLYGON CLIPPING:

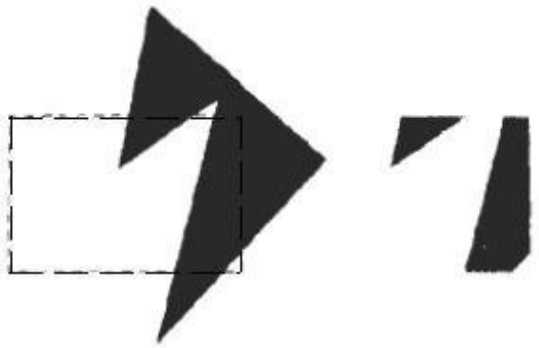
A polygon boundary processed with a line clipper may be displayed as a series of unconnected line segments (fig. (a)) depending on the orientation of the polygon to the clipping window. But we want to display a bounded area after clipping (fig. (b)).

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.



Before Clipping *After Clipping*

Fig. (a)



Before Clipping *After Clipping*

Fig. (b)

Sutherland-Hodgeman Polygon Clipping:

We can correctly clip a polygon 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 in turn.

Beginning with the initial set of polygon vertices, we could first clip the polygon against the left rectangle boundary to produce a new sequence of vertices. The new set of vertices could then k successively passed to a right boundary clipper, a bottom boundary clipper, and a top boundary clipper, as shown in the diagram below. At each step, a new sequence of output vertices is generated and passed to the next window boundary clipper.

Original Polygon *Clip Left* *Clip Right* *Clip Bottom* *Clip Top*

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

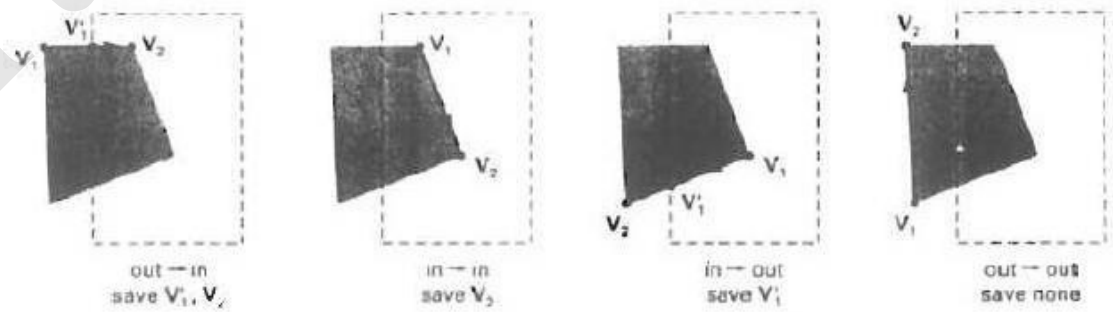
(1) If the first vertex is outside the window boundary and the second vertex is inside, both the intersection point of the polygon edge with the window boundary and the second vertex are added to the output vertex list.

(2) If both input vertices are inside the window boundary, only the second vertex is added to the output vertex list.

(3) If the first vertex is inside the window boundary and the second vertex is outside, only the edge intersection with the window boundary is added to the output vertex list.

(4) If both input vertices are outside the window boundary, nothing is added to the output list.

These four cases are illustrated in the following diagram for successive pairs of polygon vertices.



Once all vertices have been processed for one clip window boundary, the output list of vertices is clipped against the next window boundary.

Convex polygons are correctly clipped by this algorithm, but concave polygons may be displayed with extraneous lines as shown below. This occurs when the clipped polygon should have two or more separate sections.

CURVE CLIPPING:

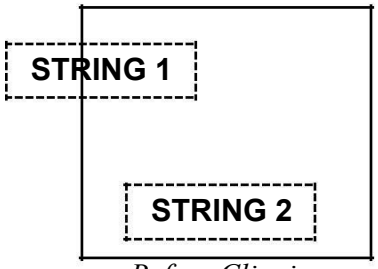
Curve-clipping procedures will involve nonlinear equations, however, and this requires more processing than for objects with linear boundaries.

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.

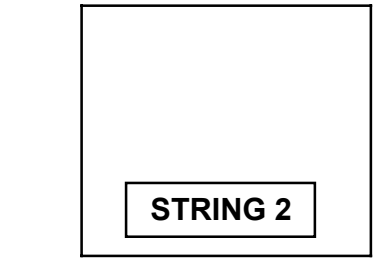
Similar procedures can be applied when clipping a curved object against a general polygon clip region. On the first pass, we can clip the bounding rectangle of the object against the bounding rectangle of the clip region. If the two regions overlap, we will need to solve the simultaneous line-curve equations to obtain the clipping intersection points.

TEXT CLIPPING:

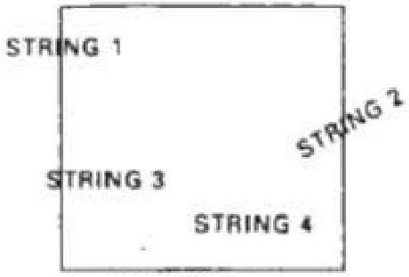
There are several techniques that can be used to provide text clipping in a graphics package. The clipping technique used will depend on the methods used to generate characters and the requirements of a particular application.



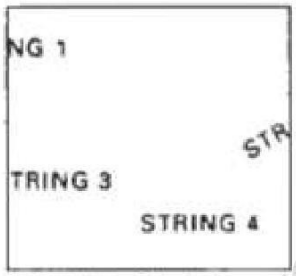
Before Clipping



After Clipping
fig. (1)

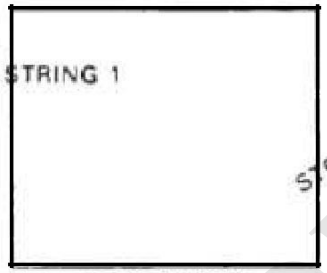


Before Clipping

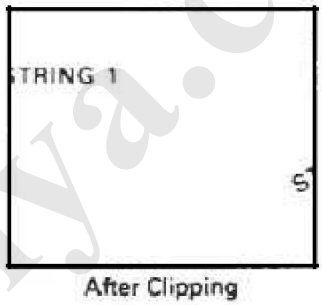


After Clipping

fig. (2)



Before Clipping



After Clipping

fig. (3)

The simplest method for processing character strings relative to a window boundary is to use the ***all-or-none string-clipping*** strategy shown in the diagram (fig. (1)). If all of the string is inside a clip window, we keep it. Otherwise, the string is discarded.

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 shown in fig. (2).

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 as in fig. (3).

EXTERIOR CLIPPING:

There are procedures to clip ***inside*** the region, to save the parts if the picture that are ***outside*** the region ***exterior clipping***.

A typical example of the application of exterior clipping is in multiple-window systems. To correctly display the screen windows, we often need to apply both internal and external clipping.

Objects within a window are clipped to the interior of that window. When other higher-priority windows overlap these objects, the objects are also clipped to the exterior of the overlapping windows.

Exterior clipping is used also in other applications that require overlapping pictures. Examples here include the design of page layouts in advertising or publishing applications or for adding labels or design patterns to a picture. The technique can also be used for combining graphs, maps, or schematics. For these applications, we can use exterior clipping to provide a space for an insert into a larger picture.

Procedures for clipping objects to the interior of concave polygon windows can also make use of external clipping.

End of Unit – I

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.

PROJECTIONS

After converting the description of objects from world co-ordinates to viewing co-ordinates, we can project the three dimensional objects onto the two dimensional view plane. There are two basic ways of projecting objects onto the view plane.

1. Parallel projection

2. Perspective projection

2.1. PARALLEL AND PERSPECTIVE PROJECTIONS

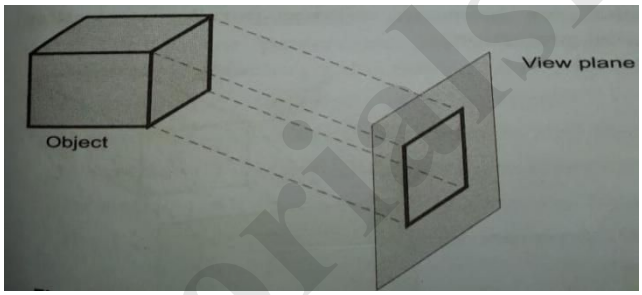
Parallel Projection:

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.



Types of parallel projections:

Parallel projections are basically categorized into two types.

Depending on the relation between the direction of projection and normal to the view plane.

- When the direction of projection is perpendicular to the view plane, we have an orthographic parallel projection.

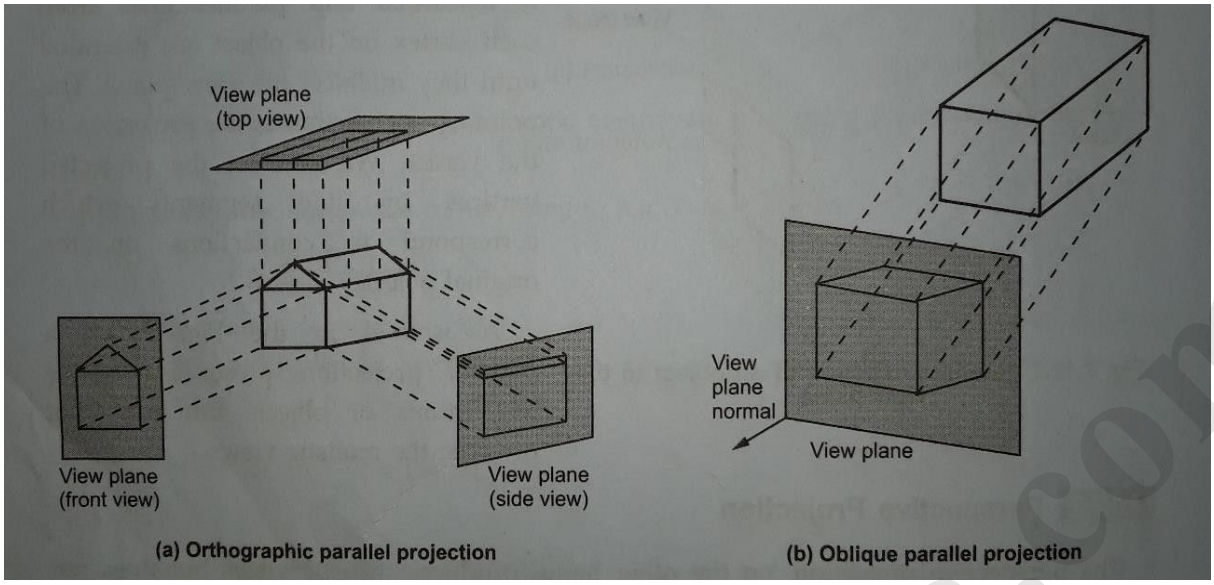
Otherwise, we have an oblique parallel projection.

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well



Orthographic projection:

• Orthographic projections are classified as **axonometric orthographic projection** and **multiview orthographic projection**.

Axonometric orthographic projection:

The most common types of orthographic projections are

➢the front projection,
➢top projection and
➢side projection. In all these,

❖the projection plane (view plane) is perpendicular to the principle axis.

These projections are often used in engineering drawing to depict machine parts, assemblies, buildings and so on.

❖ The orthographic projections can display more than one fact of an object. Such an orthographic projection is called **axonometric orthographic projection**.

The most commonly used axonometric orthographic projection is the **isometric** projection.

Types of Axonometric Projection:

Axonometric projections of three types:

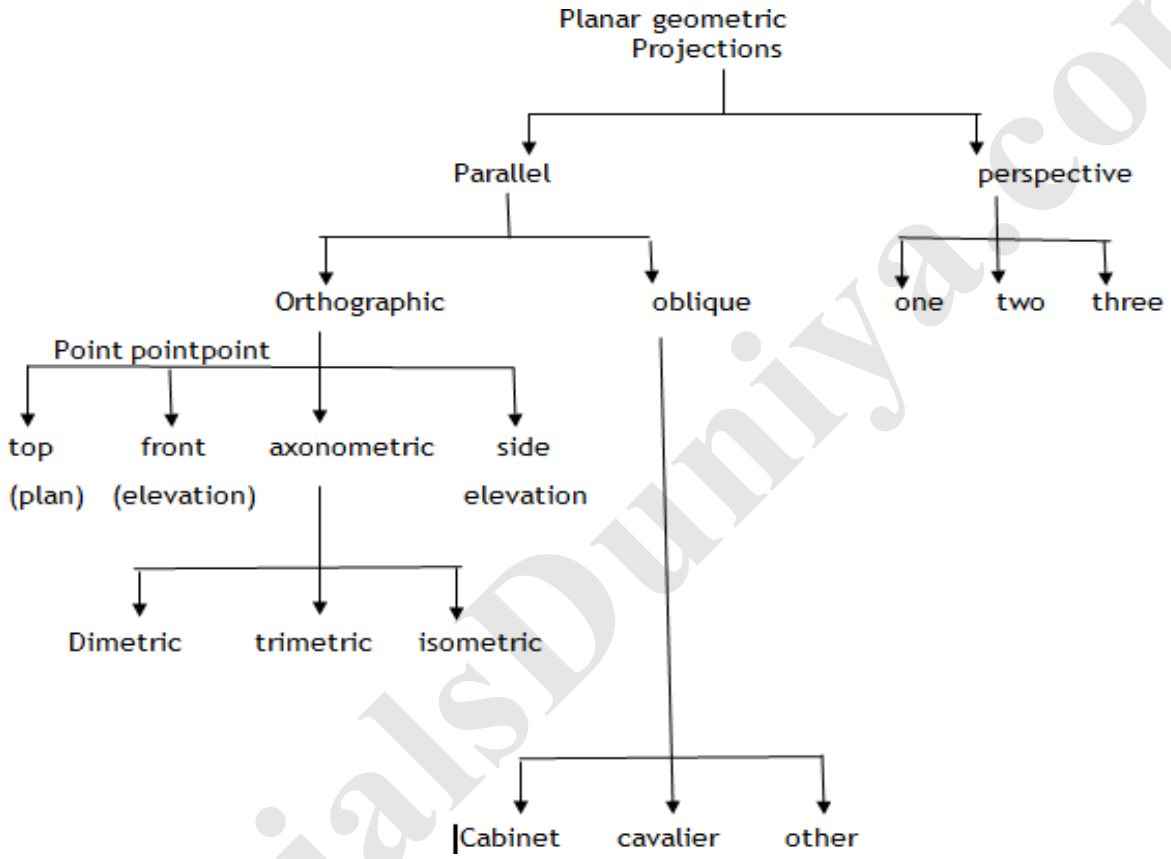
Isometric: All three principle axes are foreshortened equally.
Dimetric: Two principle axes are foreshortened equally.
Trimetric: All three principle axes are foreshortened unequally.

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.



2.3.POLYGON SURFACES

CONCEPT

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

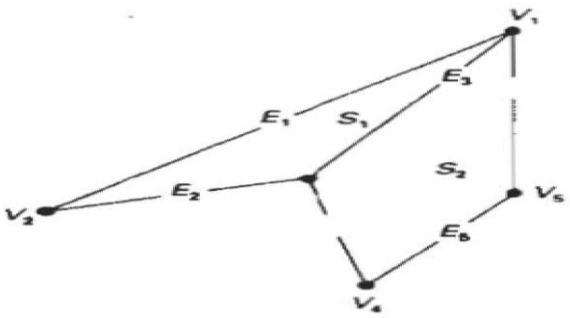
Polygon Tables

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.



Vertex table Edge Table Polygon surface table V1 : X1, Y1, Z1 E1 : V1, V2 S1 : E1, E2, E3 V2 : X2, Y2, Z2 E2 : V2, V3 S2 : E3, E4, E5, E6 V3 : X3, Y3, Z3 E3 : V3, V1 V4 : X4, Y4, Z4 E4 : V3, V4 V5 : X5, Y5, Z5 E5 : V4, V5 E6 : V5, V1

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, edgetable can be expanded to include forward points into the polygon table so that common edges betweenpolygons can be identified more rapidly.

vertices are input, we can calculate edge slopes and we can scan the coordinate values to identify theminimum 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 areciprocal 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 throughseveral 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---- (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.

Polygon Meshes

A single plane surface can be specified with a function such as fillArea. But when object surfaces are tobe 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.

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.

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.

$$x2 + y2 + z2 = r2$$

Ellipsoid

Ellipsoid surface is an extension of a spherical surface where the radius in three mutually perpendicular directions can have different values

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.
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

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.

Parametric Continuity Conditions

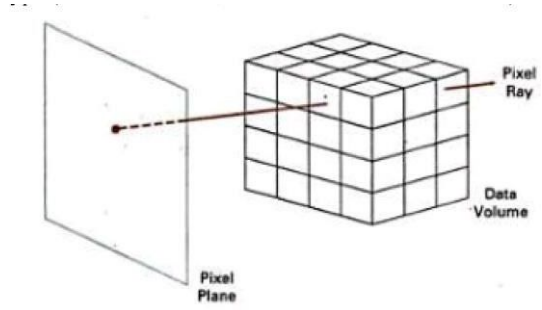
For a smooth transition from one section of a piecewise parametric curve to the next various **continuity conditions** are needed at the connection points.
If each section of a spline in described with a set of parametric coordinate functions or the form
x = x(u), y = y(u), z = z(u), u1<= u <= u2

Zero order parametric continuity referred to as C0 continuity, means that the curves meet. (i.e) the values of x,y, and z evaluated at u2 for the first curve section are equal. Respectively, to the value of x,y, and z evaluated at u1 for the next curve section.
First order parametric continuity referred to as C1 continuity means that the first parametric derivatives of the coordinate functions in equation (a) for two successive curve sections are equal at theirjoining 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.

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 G0 continuity means that the two curves sections must have the same coordinate position at the boundary point.
First order Geometric Continuity referred as G1 continuity means that the parametric first derivatives are proportional at the interaction of two successive sections.
Second order Geometric continuity referred as G2 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.

2.4.VISUALIZATION OF DATA SETS CONCEPT:
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 representation for Vector fields

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

(Vx , Vy,Vz,) one for each coordinate direction, and a two-dimensional vector has two components (Vx,Vy,). 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 forthe 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.

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.
SIGNIFICANCE:
The use of graphical methods as an aid in scientific and engineering analysis is commonly referred to as **scientific visualization**.

2.5 THREE DIMENSIONAL GEOMETRIC AND MODELING TRANSFORMATIONS:

CONCEPT:

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

Translation

In a three dimensional homogeneous coordinate representation, a point or an object is translated from position P = (x,y,z) to position $P^1 = (x^1, y^1, z^1)$ with the matrix operation.

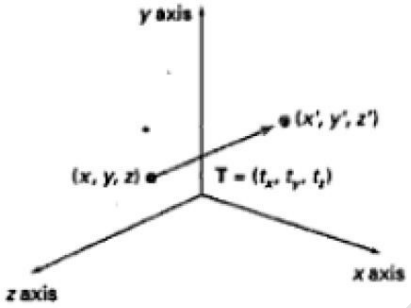
Rotation

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.

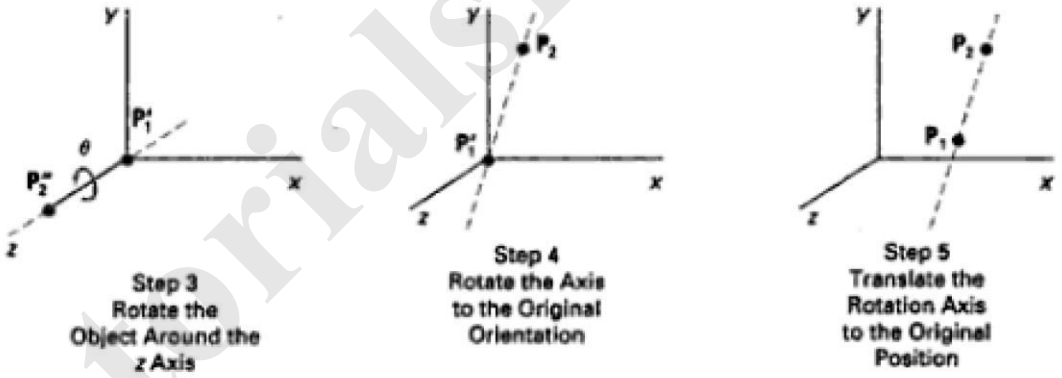
Co-ordinate Axes Rotations

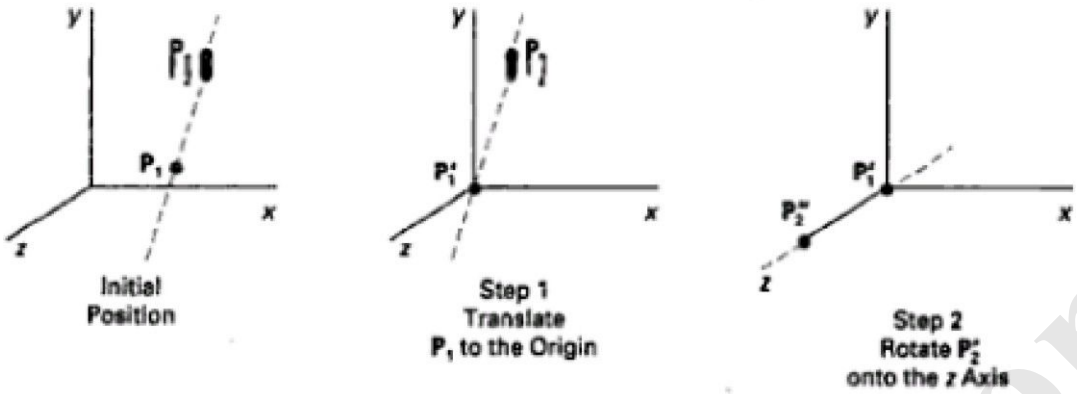
The 2D z axis rotation equations are easily extended to 3D.



$X^1 = x \cos\theta - y \sin\theta$

$Y^{1=X\cos}\theta + y \sin\theta$.





Scaling

The matrix expression for the scaling transformation of a position P = (x,y,.z)

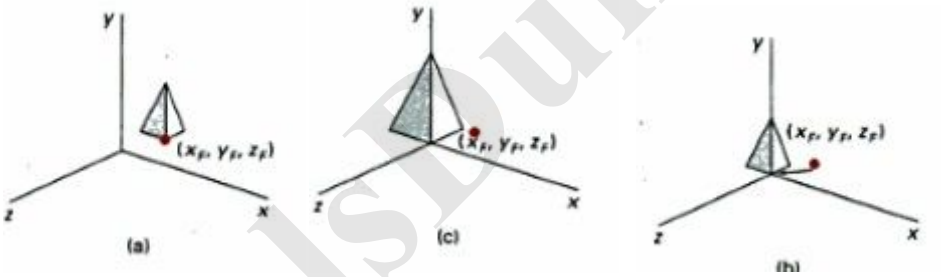
Scaling an object changes the size of the object and repositions the object relatives to the coordinate origin.

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 (sx = sy= sz) .

Scaling with respect to a selected fixed position (x f, yf, zf) 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



Other Transformations

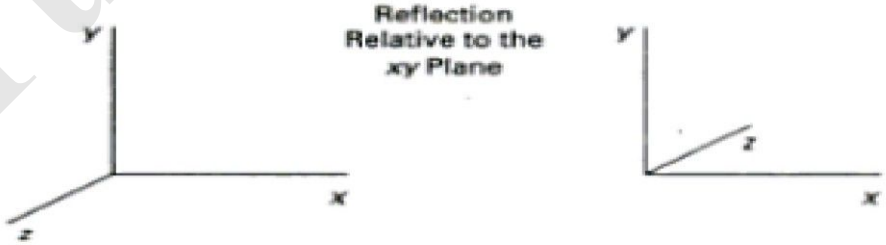
Reflections

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 1800 rotations about the axis.

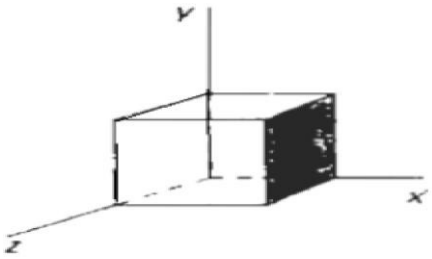
Reflection relative to a plane are equivalent to 1800 rotations in 4D space.

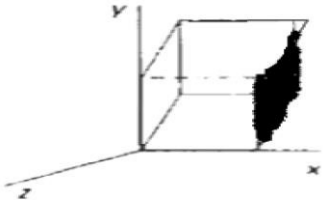
When the reflection plane in a coordinate plane (either xy, xz or yz) then the transformation can be a conversion between left-handed and right-handed systems.



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.





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.

Three 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.

2.6.THREE-DIMENSIONAL VIEWING

CONCEPT:

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

Viewing Pipeline:

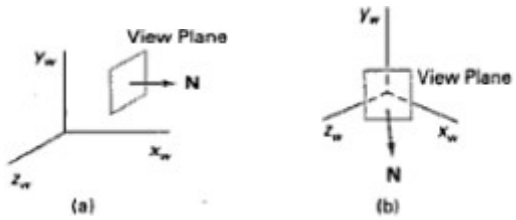
In the view of a three dimensional scene, to take a snapshot we need to do the following steps.
1. Positioning the camera at a particular point in space.
2. Deciding the camera orientation (i.e.,) pointing the camera and rotating it around the line of right to set up the direction for the picture.
3. When snap the shutter, the scene is cropped to the size of the „window of the camera and light from the visible surfaces is projected into the camera film.

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

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.

A **viewplane** or **projection plane** is set-up perpendicular to the viewing Zv 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 Zv 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.

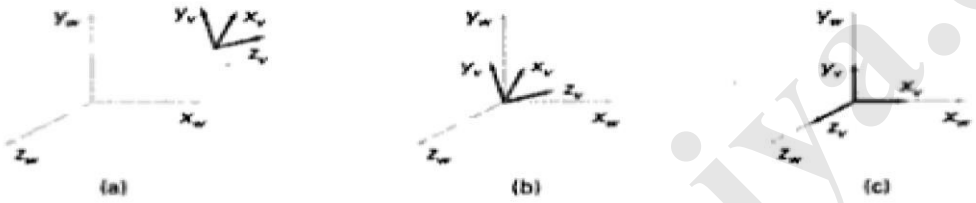


Transformation from world to viewing coordinates

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 xv, yv and zv axes with the world xw,yw and zw axes respectively.

If the view reference point is specified at world position(x0,y0,z0) this point is translated to the world origin with the matrix transformation.



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|) = (n1, n2, n3) u = (V*N) / (|V*N|) = (u1, u2, u3) v = n*u = (v1, v2, v3)

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

2.7.VISIBLE SURFACE IDENTIFICATION

CONCEPT

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.

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.

BACK-FACE DETECTION

A fast and simple object-space method for identifying the back faces of a polyhe dron is based on the "inside-outside" tests discussed in Chapter 10. A point ***(x,*** y, z) is "inside" a polygon surface with plane parameters A, B, C, and D if

$$Ax + By + Cz + D < 0$$

When an inside point is along the line of sight to the surface, the polygon must be a back face (we are inside that face and cannot see the front of it from our viewing position).

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 (or "camera") position, as shown in Fig. 13-1, then this polygon is a back face if

$$\mathbf{V} \cdot \mathbf{N} > 0$$

Furthermore, if object descriptions have been converted to projection coordinates and our viewing direction is parallel to the viewing z_v- axis, then $\mathbf{V} = (0, 0, V_z)$ and

$$\mathbf{V} \cdot \mathbf{N} = V_z C$$

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,, axis (Fig. 13-21, the polygon is a back face if C < 0. AIso, we cannot see any face whose normal has z component C=0, since our viewing direction is grazing that polygon. Thus, in general, we can label any polygon as a back face if its normal vector has a ztomponent value:

$$C \leq 0$$

Figure 13-1
Vector **V** in the viewing direction and a back-face normal vector **N** of a polyhedron

Figure 13-2
A polygon surface with plane parameter $C < 0$ in a right-handed viewing coordinate system is identified as a back face when the viewing direction is along the negative z_v axis.

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 onthe projection plane. This procedure is also referred to as the z-buffer method,since object depth is usually measured from the view plane along the z axis of aviewing system. Each surface of a scene is processed separately, one point at atime across the surface. The method is usually applied to scenes containing onlypolygon surfaces, because depth values can be computed very quickly **and** themethod is easy to implement. But the mcthod can be applied to nonplanar surfaces.

With object descriptions converted to projection coordinates, each ***(x,*** y, ***2)*** position on a polygon surface corresponds to the orthographic projection point***(x,*** y) on the view plane. Therefore, for each pixel pos~tion ***(x,*** y) on the viewplane, object depths can be compared by comparing z values.

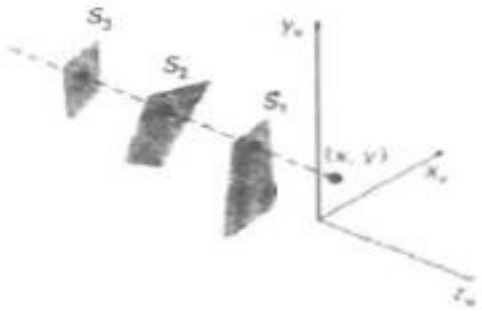


Figure 13-4
At view-plane position (x, y), surface S_1 has the smallest depth from the view plane and so is visible at that position.

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),

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

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 $z > \text{depth}(x, y)$, then set

$$\text{depth}(x, y) = z, \qquad \text{refresh}(x, y) = I_{\text{surf}}(x,y)$$

where I_{backgnd} is the value for the background intensity, and $I_{\text{surf}}(x,y)$ 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:

$$z = \frac{-Ax - By - D}{C} \qquad (13\text{-}4)$$

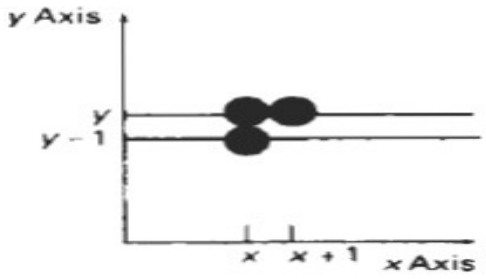


Figure 13-5
From position (x, y) on a scan line, the next position across the line has coordinates $(x + 1, y)$, and the position immediately below on the next line has coordinates $(x, y - 1)$.

For any scan line (Fig. 13-5), adjacent horizontal positions across the line differ by
1, 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. 13-4 as

$$z' = \frac{-A(x + 1) - By - D}{C} \qquad (13\text{-}5)$$

$$z' = z - \frac{A}{C} \qquad (13\text{-}6)$$

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

The ratio ***-A/C*** is constant for each surface, so succeeding depth values across a
scan line are obtained from precrd~ngv alues with a single addition.
On each scan line, we start by calculating the depth on a left edge of the polygon that intersects that scan line (Fig. 13-6). Depth values at each successive
position across the scan line are then calculated by Eq. 13-6.
We first determine the y-coordinate extents of each polygon, and process the surface from the topmost scan line to the bottom scan line, as shown in Fig.

13-6. Starting at a top vertex, we can recursively calculate x positions down a left
edge of the polygon as x' = ***x*** - **1/m,** where ***rn*** is the slope of the edge (Fig. 13-7).
Depth values down the edge are then obtained recursively as

$$z' = z + \frac{A/m + B}{C} \qquad (13\text{-}7)$$

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

$$z' = z + \frac{B}{C}$$

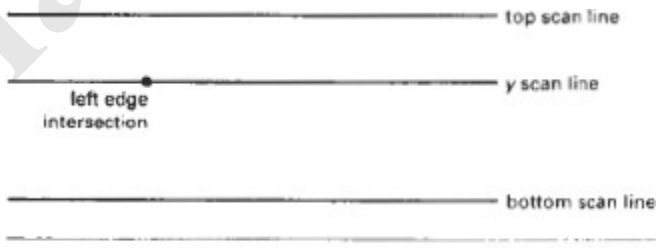


Figure 13-6
Scan lines intersecting a polygon surface.

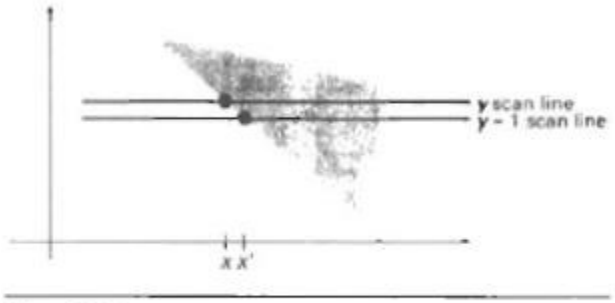


Figure 13-7
Intersection positions on successive scan lines along a left polygon edge.

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.

SCAN-LINE METHOD

This imagespace method for removing hidden surface5 is an extension of the scan-linealg&ithni for tilling polygon interiors. Instead of filling just one surface,we now deal with multiple surfaces. As each scan line is processed, all polygonsurfaces intersecting that line are examined to determine which are visible.

Across each scan line, d ~ p t hca lculations are made for each overlapping surfaceto determine which is nearest to the view plane. When the visible surface has been determined, the mtensity value for that position is entered into the refresh buffer.

We assume that tables are-set up for the various surfaces, as discussed in Chapter 10, 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 sur- face, intensity information for the surfaces, and possibly pointers into the edge table.

To facilitate the search for surfaces crossinga @ven 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 toright. At the leftmost boundary of a surface, the surface flag is turned on; and at the rightmost boundary, it is turned off.

Figure 13-10 illustrates the scan-line method for locating visible portions of

surfaces for pixel positions along the line. The active list for &an 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 **Sl** is on.

Therefo~n, o depth calculations are necessary, and intensity information for surfaceS, is entered from the polygon table into the refresh buffer. Similarly, betweenedges EH and FG, only the flag for surface S2 is on. NO other positionsalong scan line **1** intersect surfaces, so the intensity values in the other areas areset to the background intensity. The background intensity can be loaded throughoutthe buffer in an initialization routine.

For scan lines 2 and 3 in Fig. 13-10, the active edge l ~ scto ntains edges ***AD,*** *EH,* ***BC,*** and *FG.* Along scan line 2 from edge ***AD*** to edge *EH,* only the flag for

surface S, 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 **SI** is assumed to be less

than that of S,, so intensities for surface S, are loaded into the refresh buffer until

boundary BC is encountered. Then the flag for surface **SI** goes off, and intensities

for surface ***S2*** are stored until edge FG is passed.

We can take advantage of-coherence along the scan lines as we pass from one scan line to the next. In Fig. 13-10, scan line 3 has the same active list of edges

as scan line 2. Since no changes have occurred in line intersections, it is unnecessary

again to make depth calculations between edges *EH* and ***BC.*** The two surfaces must be in the same orientation as determined on scan line 2, so the intensities

for surface S, can be entered without further calculations.

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.

When these coherence methods are used, we need to be careful to keep track ofwhich surface section is visible on each scan line. This works only if surfaces donot cut through or otherwise cyclically overlap each other (Fig. 13-11). If any kindof cyclic overlap is present in a scene, we can divide the

surfaces to eliminate theoverlaps. The dashed lines in this figure indicate where planes could be subdividedto form two distinct surfaces, so that the cyclic overlaps are eliminated

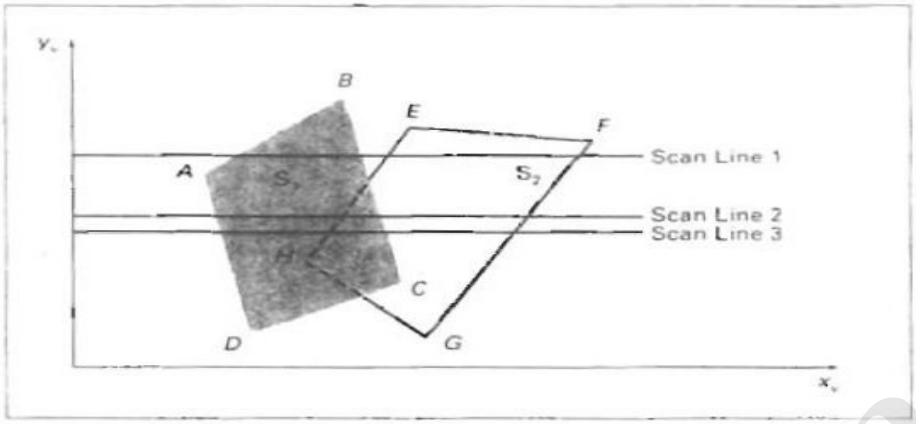


Figure 13-10
Scan lines crossing the projection of two surfaces, S_1 and S_2, in the view plane. Dashed lines indicate the boundaries of hidden surfaces.

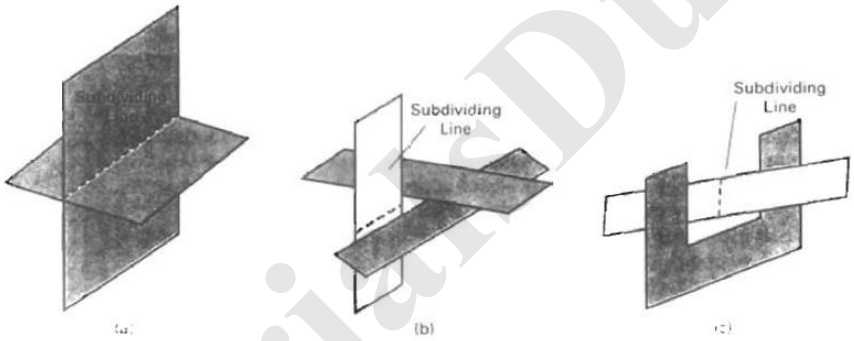
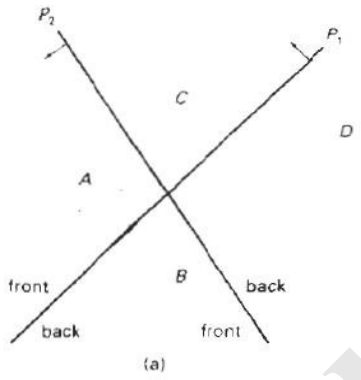


Figure 13-11
Intersecting and cyclically overlapping surfaces that alternately obscure one another.

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 fwed 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 subdivrsion,
relative to the viewing direction. Figure 13-19 illustrates the basic con. cept in this algorithm. With plane **PI,**w e firstpartition the space into two sets ofobjects. One set of objects is behind, or in back of, plane ***P,*** relative to the viewingdirection, and the other set is in front of PI. Since one object is intersected byplane PI, we divide that object into two separate objects, labeled ***A*** and ***B.*** Objects***A*** and C are in front of P,, and objects ***B*** and Dare behind PI. We next partitionthe space again with plane ***P2*** and construct the binary tree representationshown in Fig. **13-19(b).** In this tree, the objec?s are represented as terminalnodes, with front objects as left branches and back objects as right branches.



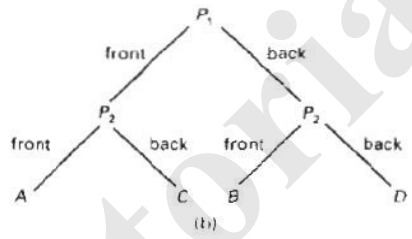


Figure 13-19
A region of space (a) is partitioned with two planes P_1 and P_2 to form the BSP tree representation in (b).

For objects dtascribed with polygon facets, we chose the partitioning planes to coincide with tne polygon planes. The polygon equations are then **used** to identify "inside" and "outside" polygons, and the tree is constructed with one partitioning plane for each polygon face. Any polygon intersected by a partitioningplane is split into two parts. When the BSP tree is complete, we process thetree by selecting :he surfaces for display in the order back to front, so that foregroundobjects are painted over the backgroundobjects. Fast hardware implementationsfor c.onstructing and processing DSP trees are used in some systems.

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 locatingthose view areas that represent part of a single surface. We apply thismethod by successively dividing the total viewing area into smaller and smallerrectangles until each small area is the projection of part of ***n*** single visible surfaceor no surface at all.

To implement this method, we need to establish tests tnat can quickly identifythe area as part of a single surface or tell us that the area is too complex to analyz eeasily. Starting with the total view, we apply the tests to determine whether we should subdivide the total area into smaller rectangles. If the tests indicatethat the view is sufficiently complex, we subdivide it. Next. we apply the tests toeach 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 areainto four equal parts at each step, as shown in Fig. 13-20. This approach is similarto that used in constructing a quadtree.

A viewing area with a resolution of 1024 by 1024 could be subdivided ten times in this way before a.subarea is reduced toa pint.

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 possibile relationships that a surface can have with a specified area boundary. We can describe these relative surface characteristics in the following way (Fig. 13-21):

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.

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 camed 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 iden* an inside surface. For other **types** of surfaces, the bounding mtangles can be **used** as an initial check. If a single bounding rectangle intersects the area in some way, additional checks are wd to determine whether the surface is surrounding, overlapping, or outside. Once a single inside, overlapping, or surrounding surface has been identified, its pixel intensities are t r a n s f e d to the appropriate area within the frame buffer.

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 maxi- mum 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.

Figure 13-22 shows an example of the conditions for this method.

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. Further testing could be carried out to identify the single surface that covers the area, butit is faster to subdivide the area than to continue with more complex testing.

Once outside and surrounding surfaces have been identified for an area, theywill remain outside and surrounding suriaces for all subdivisions of the area.

Furthermore, some ins~dea nd overlapping surfaces can be expected to be eliminatedas 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 thein&-nsity of the nearest surface to the frame buffer.

-

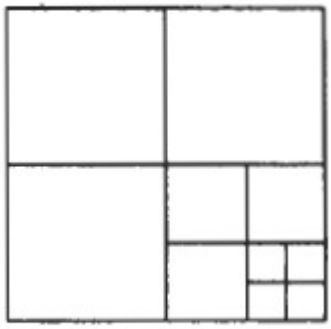


Figure 13-20
Dividing a square area into equal-sized quadrants at each step.

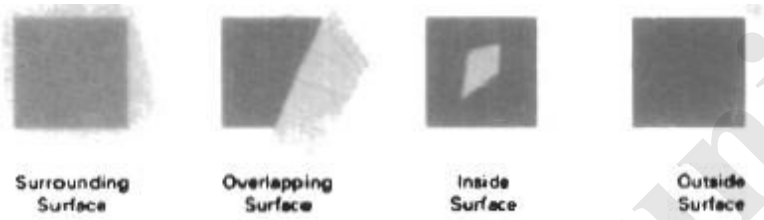


Figure 13-21
Possible relationships between polygon surfaces and a rectangular area.

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

3.1. COLOR MODELS CONCEPTS:

Color Model is a method for explaining the properties or behavior of color within some particularcontext. No single color model can explain all aspects of color, so we make use of different models tohelp 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 wavesand 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*104 Hz) and the highest frequency is a violet color (7.5 *10 14Hz)
Spectral colors range from the reds through orange and yellow at the low frequency end to greens, blues and violet at the high end.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.

3.2. 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
Cλ = X**X** + Y**Y** + Z**Z**---------- (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),
y = Y/(X+Y+Z),
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.

RGB Color Model:

Based on the tristimulus theory of 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.

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 produceother colors.

Each color point within the bounds of the cube can be represented as the triple (R,G,B) where values forR, G and B are assigned in the range from 0 to1.
The color Cλ is expressed in RGB component as Cλ = R**R** + G**G** + B

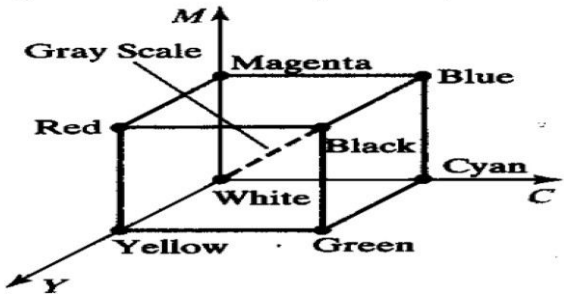
YIQ Color Model:

The National Television System Committee (NTSC) color model for forming the compositevideo signal in the YIQ model.
In the YIQ color model, luminance (brightness) information in contained in the Y parameter, chromaticityinformation (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 standardluminosity 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 abandwidth 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

CMY Color Model

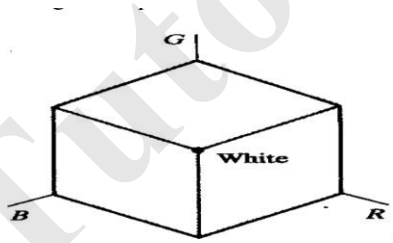
A color model defined with the primary colors cyan, magenta, and yellow (CMY) in useful for describing color output to hard copy devices.
It is a subtractive color model (i.e.,) cyan can be formed by adding green and blue light.
When white light is reflected from cyan-colored ink, the reflected light 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.



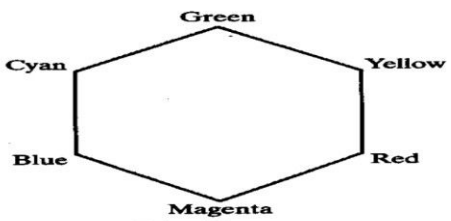
In CMY model, point (1,1,1) represents black because all components of the incident light are subtracted.
The origin represents white light.
Equal amounts of each of the primary colors produce grays along the main diagonal of the cube.
A combination of cyan and magenta ink produces blue light because 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.

HSV Color Model

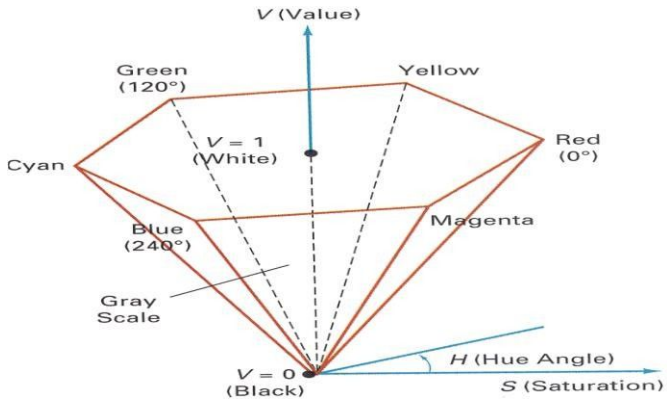
The HSV model uses color descriptions that have a more interactive appeal to a user.
Color parameters in this model are hue (H), saturation (S), and value (V).
The 3D representation of the HSV model is derived from the RGB cube. The outline of the cube has the hexagon shape.



RGB Color Cube
(a)

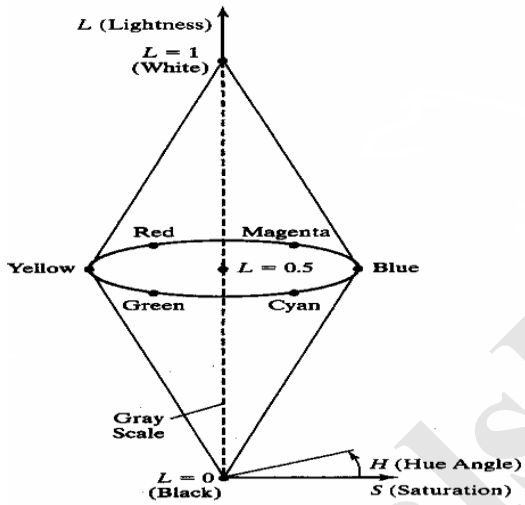


Color Hexagon
(b)



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).



3.3. 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.

DESIGN OF ANIMATION SEQUENCES

In general, an animation sequence is designed with the tollowing steps:

- Storyboard layout
- Object definitions

- Key-frame specititatlons
- Generation of in-between frames

This standard approach for animated cartoons is applied to other animation applications as well, although there are many special applications that do not follow this sequence. Real-time computer animations produced by Bight simulators,for instance, display motion sequences in response to settings on the air-craft controls.

And visualization applications are generated by the solutions of the numerical models. For 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.

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

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. In addition, the associated movements for each object are specified along with the shape.

A keyframe is a 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 great. More key frames are specified for intricate motions than for simple, slowly varing motions.

In-betweens are the intermediate frames between the key frames. The number of in-betweens 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 second. Typically, time intervals for the motion are set up so that there arr from three to five in-betweens for each pair of key frames. Depending on the speed specified for the motion, some key frames can be duplicated. For a I-minute film sequence with no duplication, we would need 1440 frames. With five in-betweens for each pair of key frames, we would need 288 key frames. If the motion is not too complicated, we could space the key frames a little farther apart.

There are several other tasks that may be required, depending on the application.

They include motion verification, editing, and production and synchronization of a soundtrack. Many of the functions needed to produce general animations are now computer-generated. Figures 16-1 and 16-2 show examples of computer-generated frames for animation sequences

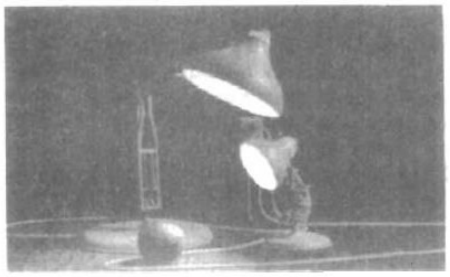


Figure 16-1
One frame from the award-winning computer-animated short film *Luxo Jr.* The film was designed using a key-frame animation system and cartoon animation techniques to provide lifelike actions of the lamps. Final images were rendered with multiple light sources and procedural texturing techniques. (*Courtesy of Pixar. © 1986 Pixar.*)

Figure 16-2
One frame from the short film *Tin Toy*, the first computer-animated film to win an Oscar. Designed using a key-frame animation system, the film also required extensive facial expression modeling. Final images were rendered using procedural shading, self-shadowing techniques, motion blur, and texture mapping. (*Courtesy of Pixar. © 1988 Pixar.*)

3.3.1 GENERAL COMPUTER-ANIMATION FUNCTIONS:

Some steps in the development of an animation sequence are well-suited to computer solution. These include object manipulations and rendering, camera motions,and the generation of in-betweens. Animation packages, such as Wavefront, for example, provide special functions for designing the animation and processing individual bjects. One function available in animation packages is provided to store and manage the object database. Object shapes and associated parameters are stored and updated in the database. Other object functions include those for motion generationand those for object rendering. Motions can be generated according to specified constraints using two-dimensional or three-dimensional transformations.
Standard functions can then be applied to identify visible surfaces and apply the rendering algorithms.
Another typical function simulates camera movements. Standard motions are zooming, panning, and tilting. Finally, given the specification for the key frames, the in-betweem can be automatically generated

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

RASTER ANIMATIONS:

On raster systems, we can generate real-time animation in limited applications using raster operations. As we have seen in Section 5-8, a simple method for translation in the xy plane is to transfer a rectangluar block of pixel values from one location to another. Twodimensional rotations in multiples of 90" are also simple to perform, although we can rotate rectangular blocks of pixels through arbitrary angles using antialiasing procedures. To rotate a block of pixels, we need to determine the percent of area coverage for those pixels that overlap the rotated block. Sequences of raster operations can be executed to produce real-time animation of either two-dimensional or three-dimensional objects, as long as we restrict the animation to motions in the projection plane. Then no viewing or visible-surface algorithms need be invoked.

We can also animate objects along two-dimensional motion paths using the color-table transformlions. Here we predefine the object at successive positions along the motion path, and set the successive blocks of pixel values to color-table entries. We set the pixels at the first position of the object to "on" values, and we set the pixels at the other object positions to the background color. The animation is then accomplished by changing the color-table values so that the object is "on" at successively positions along the ariimation path as the preceding posltion is set to the background intensity.

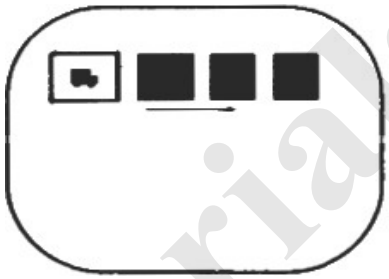


Figure 16-3
Real-time raster color-table animation.

KEY FRAME SYSTEMS:

We generate each set of in-betweens from the specification of two (or more) key frames. Motionpaths can be given with a kinematic description as a set of spline curves, or the motions can be physicdly based by specifying the forces acting on the objects to be animated.
For complex scenes, we can separate the frames into individual components or objects called cels (celluloid transparencies), an acronym from cartoon animation. Given the animation paths, we can interpolate the positions of individual objects between any two times.
With complex object transformations, the shapes of objects may change over time. Examples are clothes, facial features, magnified detail, evolving shapes, exploding or disintegrating objects, and transforming one object into another object. If all surfaces are described with polygon meshes, then the number of edges per polygon can change from one frame to the next. Thus, the total number of line segments can be different in different frames.

Morphing:

Transformation of object shapes from one form to another is called morphing, which is a shortened form of metamorphosis. Morphing methods can he applied to any motion or transition involving a change in shape.
Given two key frames for an object transformation, we first adjust the object specification in one of the frames so that the number of polygon edges (or the number of vertices) is the same for the two frames. This preprocessing step is illustrated in Fig. 16-6. A straight-line segment in key frame k is transformed into two line segments in key frame k+1. Since key frame k + 1 has an extra vertex, we add a vertex between 1 and 2 in key frame k to balance the number of vertices (and edges) In the two key frames. Using linear interpolation to generate the in-betweens. we transition the added vertex in key frame k into vertex 3' along the straight-line path shown in Fig. 16-7. An example of a triangle linearly expanding into ,a quadrilaterial is given In Fig. 16-8. Figures 16-9 and 16-10 show examples uf morphing in television advertising

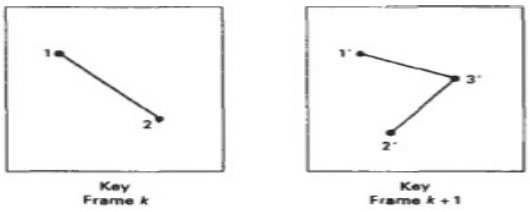


Figure 16-6
An edge with vertex positions 1 and 2 in key frame k evolves into two connected edges in key frame $k + 1$.

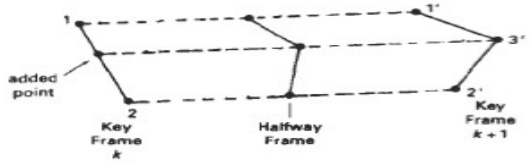


Figure 16-7
Linear interpolation for transforming a line segment in key frame k into two connected line segments in key frame $k + 1$.

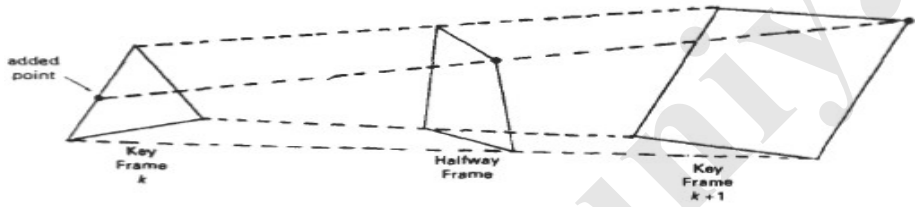


Figure 16-8
Linear interpolation for transforming a triangle into a quadrilateral.

We can state general preprocessing rules for equalizing key frames in terms of either the number of edges or the number of vertices to be added to a key frame. 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 then define

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

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

$$N_s = \text{int}\left(\frac{L_{max}}{L_{min}}\right)$$

Then the preprocessing is accomplished by
1. dividing N_e edges of keyframe_{min} into N_s+1sections
2. dividing the remaining lines of keyframe_{min} into N_s sections

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

If we equalize the vertex count, we can use parameters V_k and $V_{k+1}$to denote the number of vertices in the two consecutive frames. In this case, we define

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

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

$$N_p = \text{int}\left(\frac{V_{max} - 1}{V_{min} - 1}\right)$$

Preprocessing using vertex count is performed by
1. adding N_p points to N_{ls} line sections of keyframe,,
2. adding N_p - 1 points to the remaining edges of keyframe,,
For the triangle-toquadrilateral example, $V_k = 3$ and $V_{k+1} = 4$. Both N_{ls} and N_p are 1, so we would add one point to one edge of keyframek No points would be added to the remaining lines of keyframe$_{k+1}$.
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 16-11 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 t_1 and t_2 (Fig. 16-12). The time interval between key frames is then divided into n +1 subintervals, yielding an in-between spacing of

$$\Delta t = \frac{t_2 - t_1}{n + 1}$$

We can calculate the time for any in-between as

$$tB_j = t_1 + j\,\Delta t, \qquad j = 1, 2, \ldots, n$$

and determine the values for coordinate positions, color, and other physical parameters.
Nonzero accelerations are used to produce realistic displays of speed changes, particularly at the beginning and end of a motion sequence. We can model the start-up and slowdown portions of an animation path with spline or

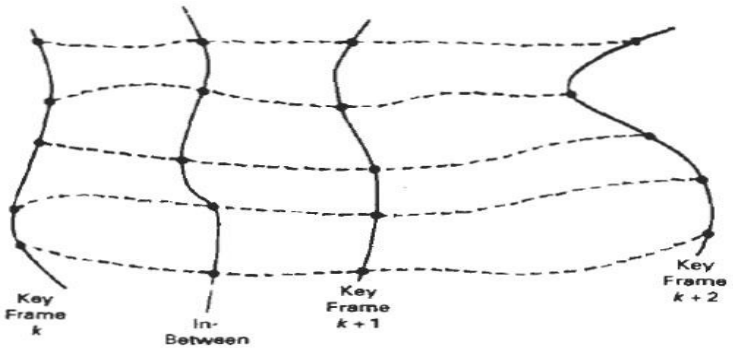


Figure 16-11
Fitting key-frame vertex positions with nonlinear splines.

trignometric functions. Parabolic and cubic time functions haw been applied to acceleration modeling, but trignometric functions are more commonly used in animation packages.

To model increasing speed (positive acceleration), we want the time spacing between frames to increase so that greater changes in position occur as the object moves faster. We can obtain an increasing interval size with the function $1 - \cos\theta, \quad 0 < \theta < \pi/2$

For n in-betweens, the time for the jth in-between would then be calculated as

$$tB_j = t_1 + \Delta t\left[1 - \cos\frac{j\pi}{2(n+1)}\right], \qquad j = 1, 2, \ldots, n$$

where Δt is the time difference between the two key frames. Figure 16-13 gives a plot of the higonometric acceleration function and the in-between spacing for n = 5.

We can model decreasing speed (deceleration) with sine in the range 0 < θ < π/2 . The time position of an in-between is now defined as

$$tB_j = t_1 + \Delta t \sin\frac{j\pi}{2(n+1)}, \qquad j = 1, 2, \ldots, n$$

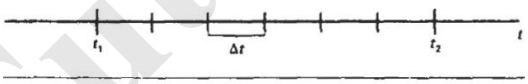


Figure 16-12
In-between positions for motion at constant speed.

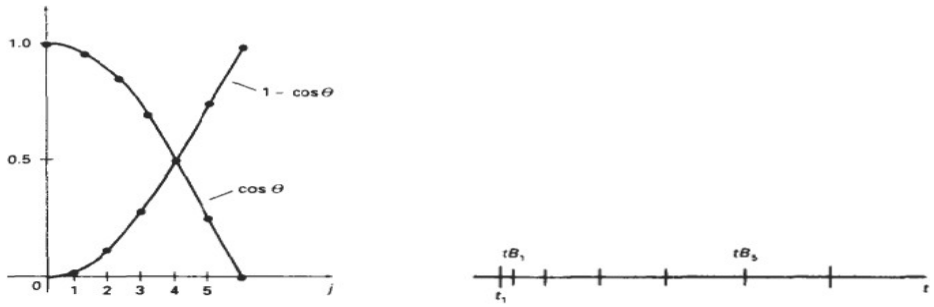


Figure 16-13
A trigonometric acceleration function and the corresponding in-between spacing for $n = 5$ and $\theta = j\pi/12$ in Eq. 16-7, producing increased coordinate changes as the object moves through each time interval.

A plot of this function and the decreasing size of the time intervals is shown in Fig. 16-14 for five in-betweens.

Often, motions contain both speed-ups and slow-downs. We can model a combination of increasing-decreasing speed by first increasing the in-between time spacing, then we decrease this sparing. A function to accomplish these time changes is

Figure 16-14
A trigonometric deceleration function and the corresponding in-between spacing for $n = 5$ and $\theta = j\pi/12$ in Eq. 16-8, producing decreased coordinate changes as the object moves through each time interval.

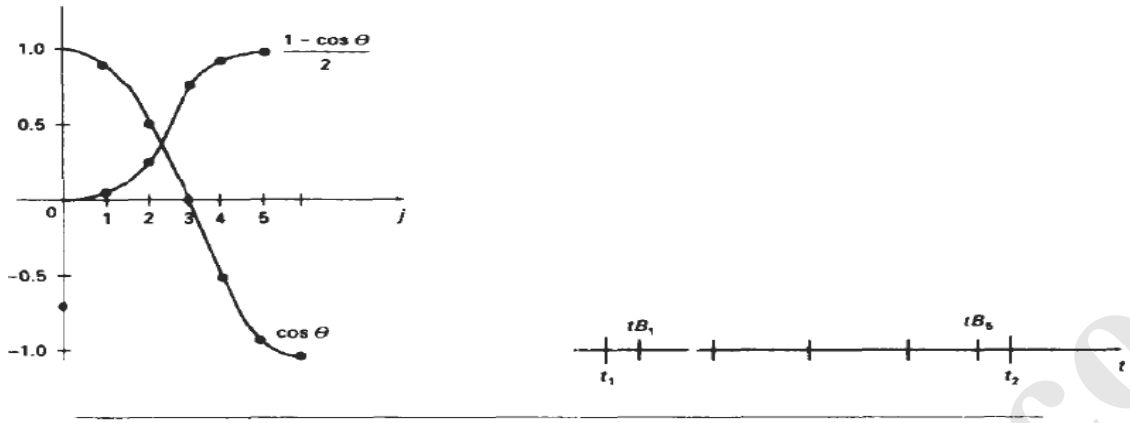


Figure 16-15
A trigonometric accelerate–decelerate function and the corresponding in-between spacing for $n = 5$ in Eq. 16-9.

$$\frac{1}{2}(1 - \cos\theta), \qquad 0 < \theta < \pi/2$$

The time for the jth in-between is now calculated as

$$tB_j = t_1 + \Delta t\left\{\frac{1 - \cos[j\pi/(n+1)]}{2}\right\}, \qquad j = 1, 2, \ldots, n \qquad (16\text{-}9)$$

with At denoting the time difference for the two key frames. Time intervals for the moving object first increase then the'time intervals decrease, as shown in Fig. 16-15.

Processing the in-betweens is simplified by initially modeling "skeleton" (wireframe)objects. This allows interactive adjustment of motion sequences. After the animation sequence is completely defined, objects can be fully rendered

3.5.GRAPHICS PROGRAMMING USING OPENGL

CONCEPT:

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

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), theprogrammer 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.

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

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

Drawing Aligned Rectangles.

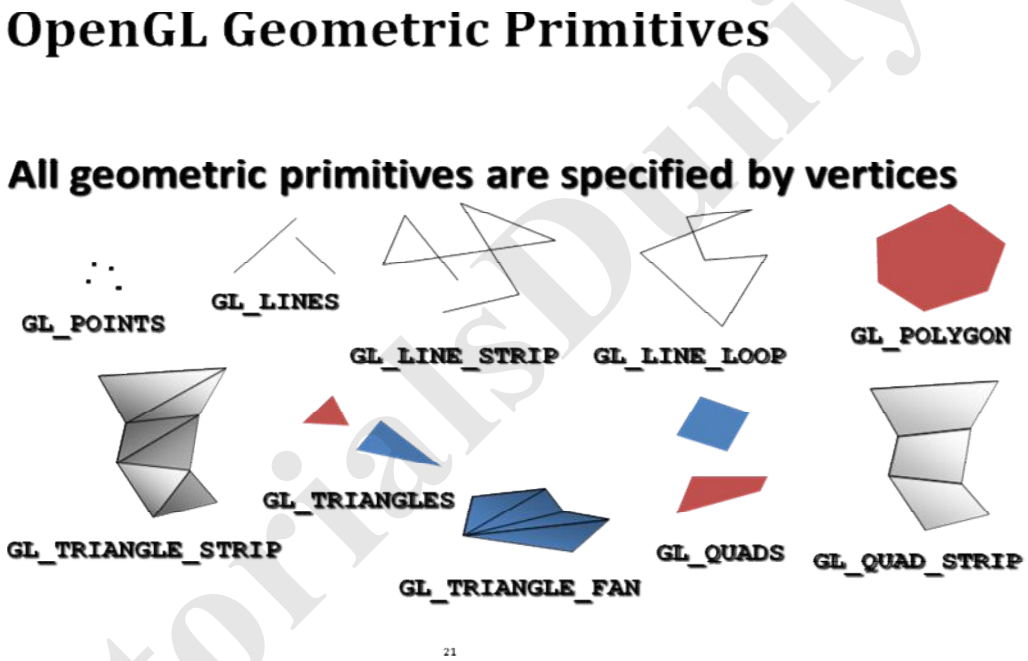
A special case of a polygon is the **aligned rectangle,** so called because its sides are aligned with the coordinate axes.

OpenGL provides the ready-made function:

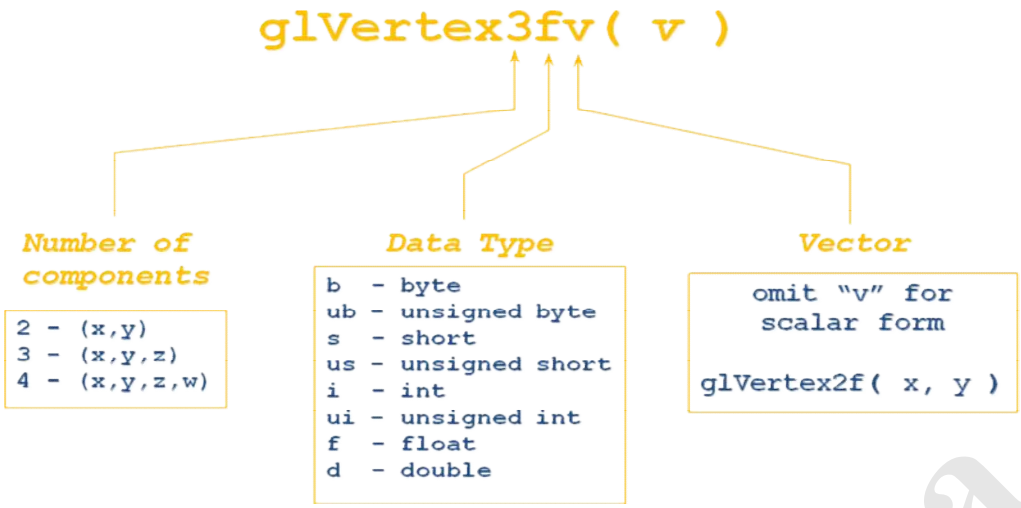
```
glRecti(GLint x1, GLint y1, GLint x2, GLint y2); // draw a rectangle with opposite corners (x1, y1) and
(x2, y2); // fill it with the current color;
glClearColor(1.0,1.0,1.0,0.0); // white background
glClear(GL_COLOR_BUFFER_BIT); // clear the window
glColor3f(0.6,0.6,0.6); // bright gray
glRecti(20,20,100,70); glColor3f(0.2,0.2,0.2); //
gray glRecti(70, 50, 150, 130);
```

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.

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, *v*2, *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 quadrilateralsare "traversed" in the same way; e.g. counterclockwise).



OpenGL Command Formats:



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(ρdr , ρdg ,ρdb) 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 reflectioncoefficients. 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 theemissive color of the surface. The emissive color of a face causes it to “glow” in the specified color, independently of any light source.

3.6 .DRAWING THREE DIMENSIONAL OBJECTS & DRAWING THREE DIMENSIONAL SCENES

CONCEPTS:

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**

// 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();
```

APPLICATIONS:

1. Implement a color models.
2. Implement a realistic scenes and objects
3. OPENGL is a easy way to make a real objects.

UNIT IV – RENDERING

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

Topic1:

Introduction to Shading Models

1) The mechanism of light reflection from an actual surface is very complicated it depends on many factors.
2) Some of these factors are geometric and others are related to the characteristics of the surface.
3) A shading model dictates how light is scattered or reflected from a surface.
4) 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
5) 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.
6) 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.
7) A shading model uses two types of light source to illuminate the objects in a scene : **point light sources** and **ambient light**.
8) 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.

- **Specular reflections** are more mirrorlike and highly directional. Incident light is directly reflected from its outer surface. This makes the surface looks shinny

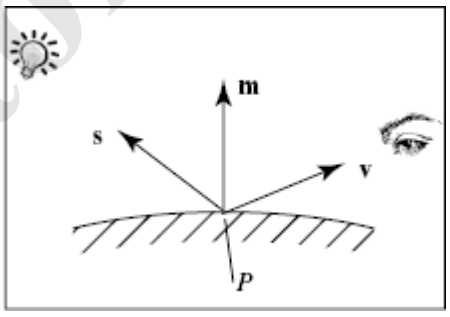
The total light reflected from the surface in a certain direction is the sum of the diffuse component and the specular component. For each surface point of interest we compute the size of each component that reaches the eye.

Computing Diffuse and Specular Components

Geometric Ingredients For Finding Reflected Light

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.

How to Compute the Diffuse Component

Some fraction of the re-radiated part reaches the eye, with an intensity denoted by **Id**.

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.

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

Therefore Id is independent of the angle between m and v.

The brightness depends on the area of the face that it sees

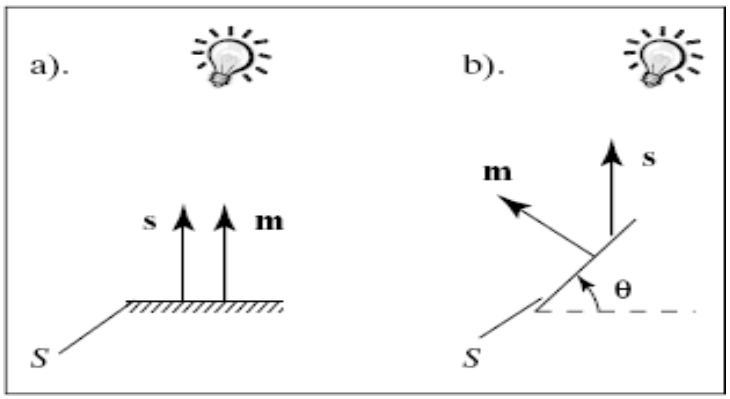


Fig (a) shows the cross section o-f a point source illuminating a face S when m is aligned with s.
Fig (b) the face is turned partially away from the light source through angle θ. The area subtended is now only cos(θ) , so that the brightness is reduced of S is reduced by this same factor. This relationship 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 coeffiecient.

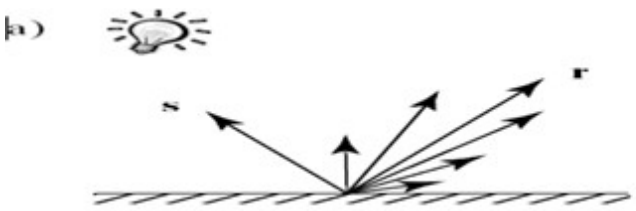
If the facet is aimed away from the eye this dot product is negative so we need to evaluate Id to 0.

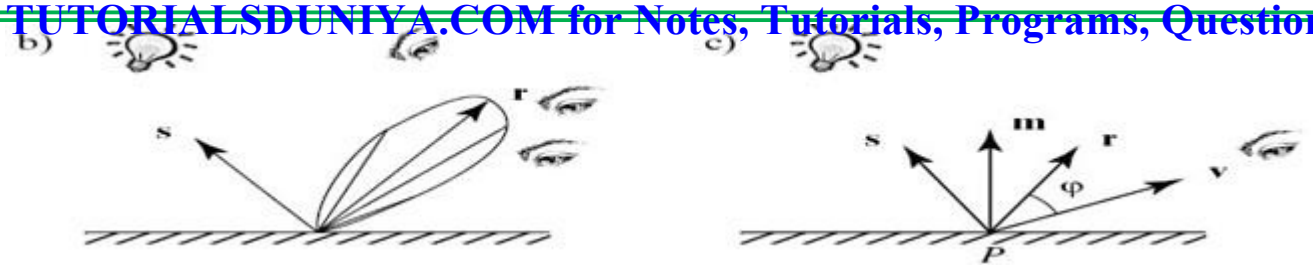
$$I_d = I_s \rho_d \max \frac{s.m}{|s||m|}, 0$$

The reflection coefficient ρ_d depends on the wavelength of the incident light , the angle θ and various physical properties of the surface.

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
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.





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

At the other nearby angles the amount 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.

(the mirror – reflection direction)

$$r = -s + \frac{(\square . \square)}{|\square|^2}\square$$

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 (φ))^f** in which f is chosen experimentally and usually lies between 1 and 200.

$I_{sp}=I_s\square(\frac{\square . \square}{|\square||\square|})^f$

ρ_s is the specular reflection coefficient
cos (φ) is the dot product between r and v

The Role of Ambient Light and Exploiting Human Perception

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 **Ia**. Each face in the model is assigned a value for its **ambient reflection coefficient ρa,** and the term **Ia ρa** is added to the diffuse and specular light that is reaching the eye from each point P on that face. **Ia** and **ρa** are found experimentally.

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

How to combine Light Contributions

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

I = ambient + diffuse + specular

$I= I_a \rho_a + I_d \rho_d \times lambert + I_{sp} \rho_s \times phong^f$ Where we define the values

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

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

To Add Color

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.

$I_r = I_{ar}\rho_{ar} + I_{dr}\rho_{dr} \times lambert + I_{spr}\rho_{sr} \times phong^f$

$I_g = I_{ag}\rho_{ag} + I_{dg}\rho_{dg} \times lambert + I_{spg}\rho_{sg} \times phong^f$

$I_b = I_{ab}\rho_{ab} + I_{db}\rho_{db} \times lambert + I_{spb}\rho_{sb} \times phong^f$ ---------- (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 =(Iar,Iag,Iab) , diffuse=(Idr,Idg,Idb) and specular=(Ispr,Ispg,Ispb).

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}

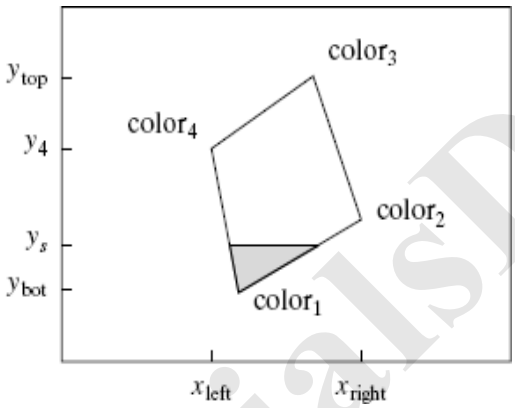
specular reflection coefficients: ρ_{sr} , ρ_{sg} and ρ_{sb}

Topic2:

FLAT SHADING AND SMOOTH SHADING

Painting a Face:

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 ybott and ytop. The tiler first fills in the row at y= ybott , then at ybott + 1, etc.

At each scan line ys, there is a leftmost pixel xleft and a rightmost pixel xright. The tiler moves from xleft to xright, placing the desired color in each pixel.

The tiler is implemented as a simple double loop:

```
for (int y= ybott ; y<= ytop; y++) // for each scan line
{
    find xleft and xright

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

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

Flat Shading

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

OpenGL offers a rendering mode in which the entire face is drawn with the same color **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);

Drawback : 1) MachBandEffect

When objects are rendered using flat shading. 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**.

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

Smooth Shading

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

The two types of smooth shading

Gouraud shading Phong shading

Gouraud Shading

Gouraud shading computes a different value of c for each pixel. For the scan line y_s in the fig. , it finds the color at the leftmost pixel, $color_{left}$, by linear interpolation of the colors at the top and bottom of the left edge of the polygon. For the same scan line the color at the top is color4, and that at the bottom is color1, so $color_{left}$ will be calculated as

$color_{left}$ = lerp(color1, color4,f),---------- (1) where the fraction

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

varies between 0 and 1 as y_s varies from y_{bott} to y4. 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:

$$c(x) = c\left(x, \frac{x - x_{left}}{x_{right} - x_{left}}\right)$$

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

$$C(x+1) = c(x) + \frac{color_{right} - color_{left}}{x_{right} - x_{left}}$$

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

```
For ( int y= ybott; y<=ytop ; y++)          //for each scan line
{
    find xleft and xright

    find colorleft and colorright

    colorinc=( colorright - colorleft) / (xright - xleft);
```

for(int x= xleft, c=colorleft; x<=xright; x++, c+=colorinc) put c into the pixel at (x,y)

}

Computationally Gouraud shading is more expensive than flat shading. Gouraud shading is established in OpenGL using the function:

glShadeModel(GL_SMOOTH);

Gouraud shading does not picture highlights well because colors are found by interpolation. Therefore in Gouraud shading the specular component of intensity is suppressed.

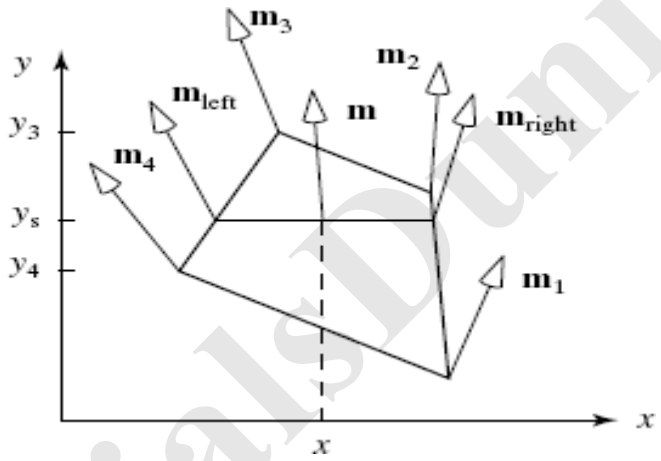
Phong Shading

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 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 ys, the vectors m left and m right are found by linear interpolation

$$m_{left} - lerp\left(m_4, m_3, \frac{y_s - y_4}{y_3 - y_4}\right)$$

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

Topic3:

Adding texture to faces

The realism of an image is greatly enhanced by adding surface texture to various faces of a mesh object. The basic technique begins with some texture function, **texture(s,t)** in **texture space** , which has two parameters s and t. The function texture(s,t) produces a color or intensity value for each value of s and t between 0(dark)and 1(light).

The two common sources of textures are

Bitmap Textures

Procedural Textures

Bitmap Textures Textures are formed from bitmap representations of images, such as digitized photo. Such a representation consists of an array **txtr[c][r]** of color values. If the array has C columns and R rows, the indices c and r vary from 0 to C-1 and R-1 resp.The function texture(s,t) accesses samples in the array as in the code:

```
Color3 texture (float s, float t)
{ return txtr[ (int) (s * C)][(int) (t * R)]; }
```

Where Color3 holds an RGB triple. Example: If R=400 and C=600, then the texture (0.261, 0.783) evaluates to txtr[156][313]. Note that a variation in s from 0 to 1 encompasses 600 pixels, the variation in t encompasses 400 pixels.

To avoid distortion during rendering , this texture must be mapped onto a rectangle with aspect ratio 6/4.

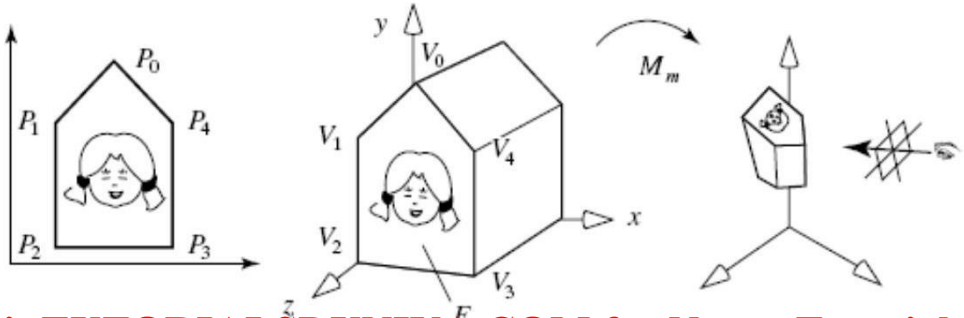
Procedural Textures Textures are defined by a mathematical function or procedure. For example a spherical shape could be generated by a function:

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

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

Painting the Textures onto a Flat Surface Texture space is flat so it is simple to paste texture on a flat surface.

Mapping texture onto a planar polygon



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

In OpenGL we use the function

glTexCoord2f()

to associate a point in texture space Pi=(si,ti) with each vertex Vi of the face. the function glTexCoord2f(s,t)sets the current texture coordinate to (s,y). All calls to **glVertex3f()**

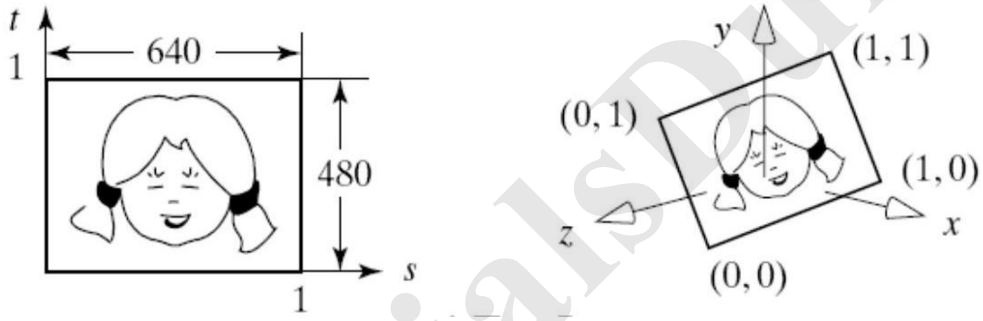
is called after a call to glTexCoord2f(), so each vertex gets a new pair of texture coordinates.

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:

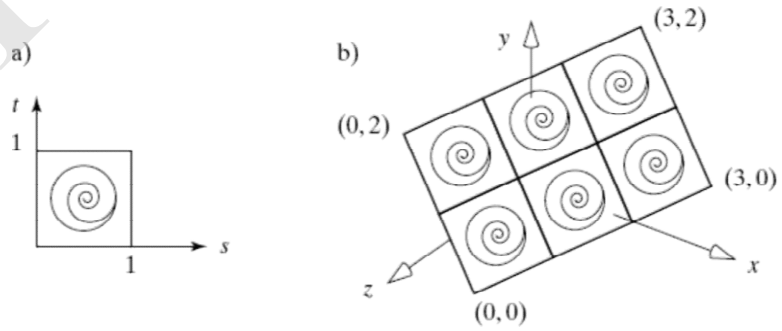
```
glBegin(GL_QUADS); //defines a quadrilateral face
glTexCoord2f(0.0 ,0.0); glVertex3f(1.0 ,2.5, 1.5);
glTexCoord2f(0.0 ,0.6);glVertex3f(1.0 , 3.7, 1.5);
glTexCoord2f(0.8 ,0.6);glVertex3f(2.0 , 3.7, 1.5);
glTexCoord2f(0.8 ,0.0);glVertex3f(2.0 , 2.5, 1.5);
glEnd();
```

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.

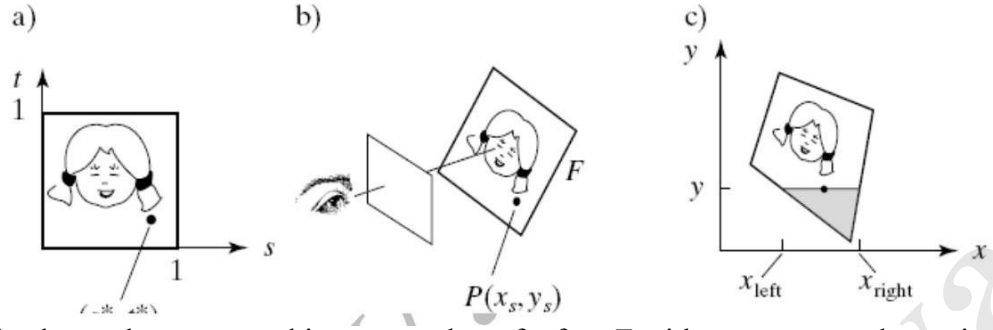
Producing repeated textures



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.

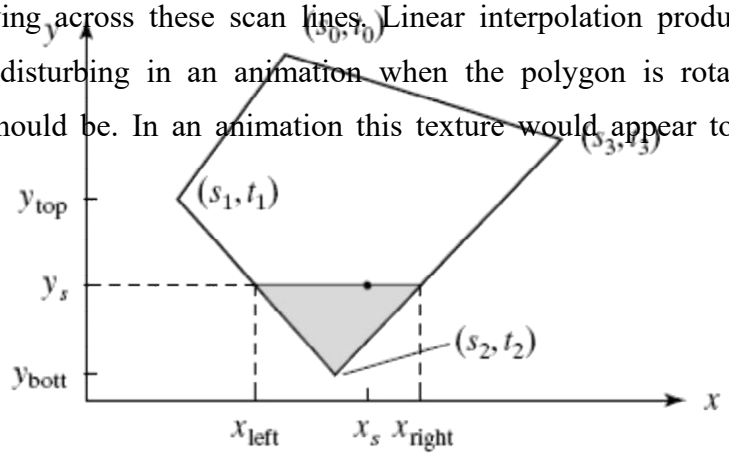
Rendering texture in a face F is similar to Gouraud Shading. It proceeds across the face pixel by pixel. For each pixel it must determine the corresponding texture coordinates (s,t), access the texture and set the pixel to the proper texture color. Finding the coordinates(s,t) should be done carefully.

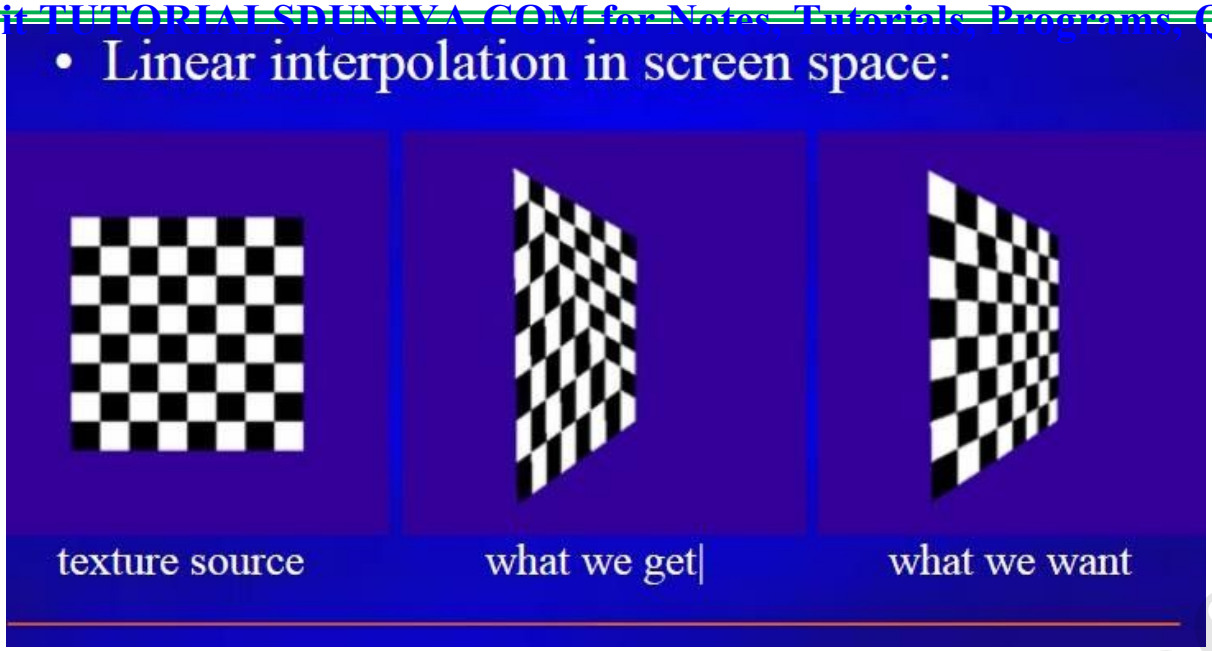
Rendering a face in a camera snapshot



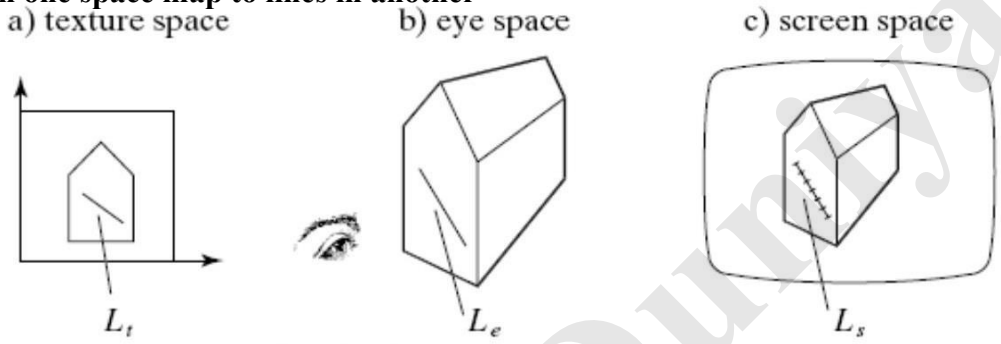
The fig shows the camera taking a snapshot of a face F with texture pasted onto it and the rendering in progress. The scan line y is being filled from xleft to xright. For each x along this scan line, we compute the correct position on the face and from that , obtain the correct position (s*, t*) within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

We compute (sleft,tleft) and (sright,tright) 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 Le in eye space projects to line Ls in screen space, and similarly the texels we wish to draw on line Ls lie along the line Lt in texture spaces, which maps to Le. The question is : if we move in equal steps across Ls on the screen, how should we step across texels along Lt in texture space? **How does motion along corresponding lines operate?**

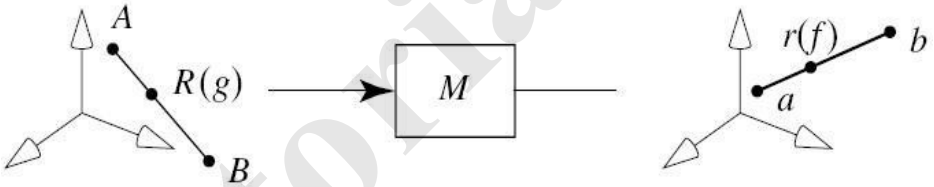


fig. shows a line AB in 3D being transformed into the line ab in 3D by the matrix M.

A maps to 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?

Topic4:

What is called a shadow buffer?

What does sliding means?

Write down the syntax for glFramebufferRenderbufferEXT(). What is the function of glCheckFramebufferStatusEXT()?

Write down the syntax for glGetRenderbufferParameteriveEXT().

List out some of the rules of FBO completeness.

Adding shadows of objects

Explain Adding shadows of objects

Shadows make an image more realistic. The way one object casts a shadow on another object gives important visual clues as to how the two objects are positioned with respect to each other. Shadows conveys lot of information as such, you are getting a second look at the object from the view point of the light source. There are two methods for computing shadows:

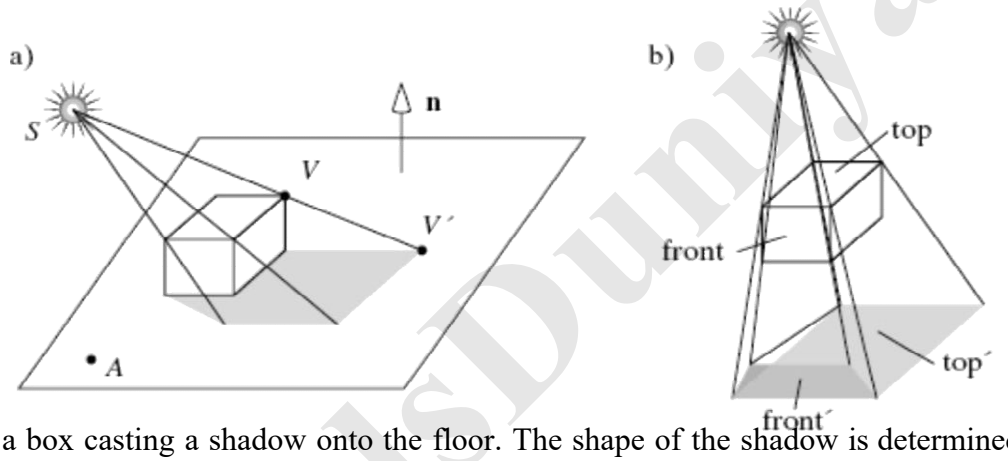
Shadows as Texture

Creating shadows with the use of a shadow buffer

Shadows as Texture

The technique of "painting" shadows as a texture works for shadows that are cast onto a flat surface by a point light source. The problem is to compute the shape of the shadow that is cast.

Computing the shape of a shadow



Fig(a) shows a box casting a shadow onto the floor. The shape of the shadow is determined by the projections of each of the faces of the box onto the plane of the floor, using the light source as the center of projection.

Fig(b) shows the superposed projections of two of the faces. The top faces projects to top" and the front face to front". This provides the key to drawing the shadow. After drawing the plane by the use of ambient, diffuse and specular light contributions, draw the six projections of the box"s faces on the plane, using only the ambient light. This technique will draw the shadow in the right shape and color. Finally draw the box.

Building the "Projected" Face

To make the new face F" produced by F, we project each of the vertices of F onto the plane. Suppose that the plane passes through point A and has a normal vector n. Consider projecting vertex V, producing V". V" is the point where the ray from source at S through V hits the plane, this point is

$$V' = S + (V - S)\frac{n.(A - S)}{n.(V - S)}$$

Creating Shadows with the use of a Shadow buffer

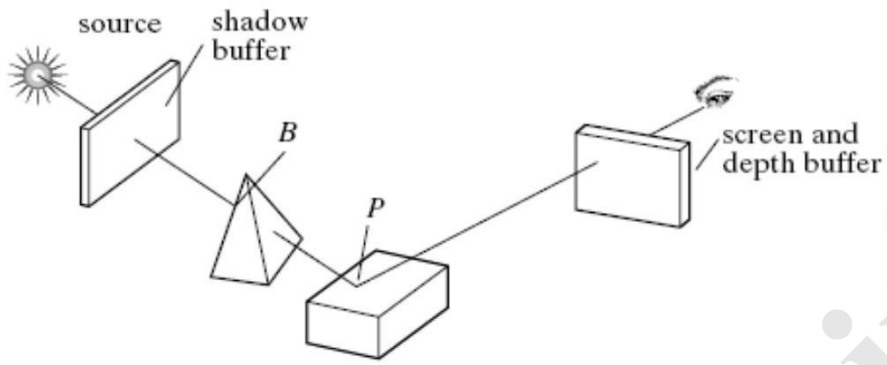
This method uses a variant of the depth buffer that performs the removal of hidden surfaces. An auxiliary second depth buffer called a shadow buffer is used for each light source. This requires lot of memory. This method is based on the principle that any points in a scene that are hidden from the light source must be in

shadow. If no object lies between a point and the light source, the point is not in shadow. The shadow buffer contains a depth picture of the

scene from the point of view of the light source. Each of the elements of the buffer records the distance from the source to the closest object in the associated direction. Rendering is done in two stages:

1) **Loading the shadow buffer** The shadow buffer is initialized with 1.0 in each element, the largest 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.

2) **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 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.

Topic5:

Write down and explain the details to build a camera in a program?

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

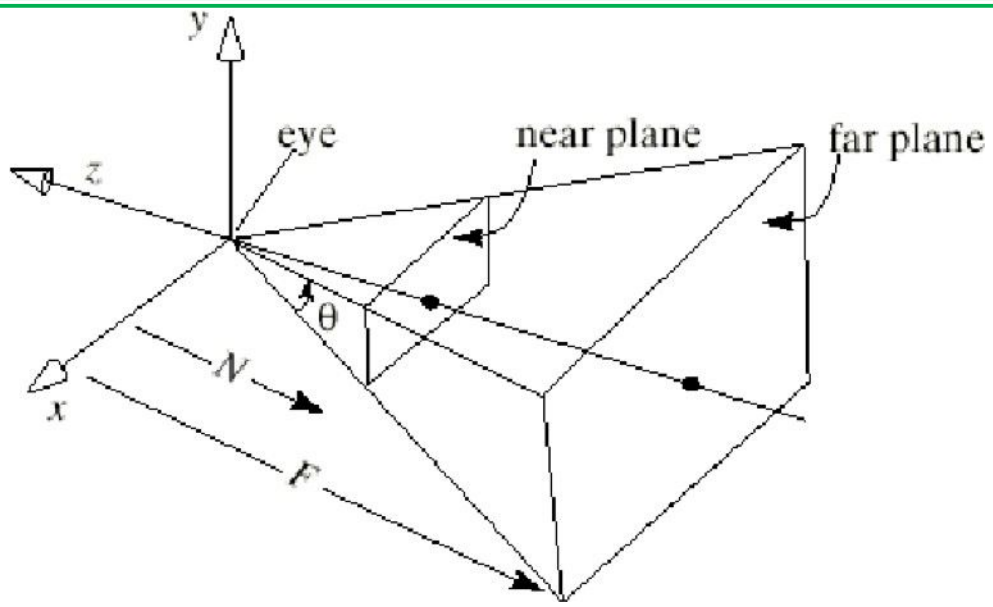
cam.slide(-1, 0, -2); //slide the camera forward and to the left

cam.roll(30); // roll it through 30 degree

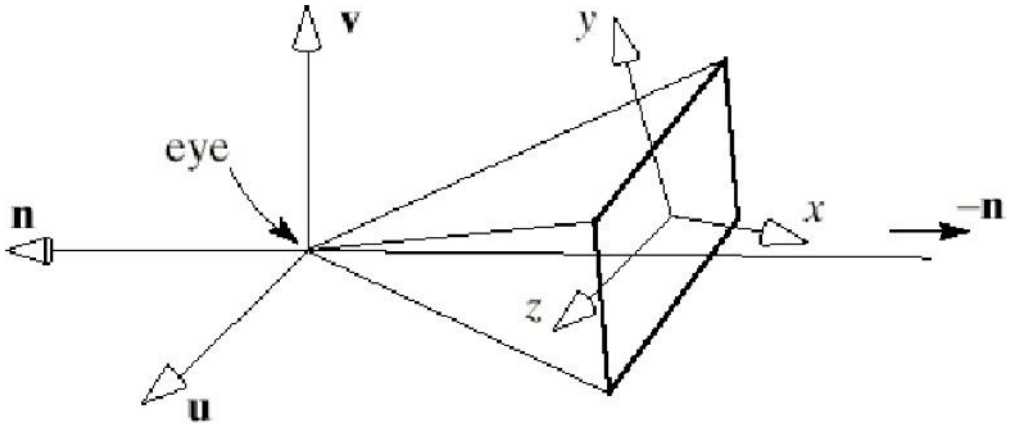
cam.yaw(20); // yaw it through 20 degree

We create a Camera class that does all things a camera does.

```
class Point3
{
public:
        float x,y,z;
       void set(float dx,float dy,float dz)
        { x=dx;y=dy;z=dz;}
   void set(Point3 &p){x=p.x;y=p.y;z=p.z;}
        Point3(float xx,float yy,float zz)
        {
        x=xx;y=yy;z=zz;
        }
        Point3()
        {
        x=y=z=0;
        }
};
class Vector3
{public:
float x,y,z;
Vector3()
        {
                x=y=z=0;
        }
Vector3(float xx,float yy,float zz)
        {
        x=xx;y=yy;z=zz;
        }
void set(float dx,float dy,float dz)
            {
                x=dx;y=dy;z=dz;
                }
void set(Vector3 &v)
                {
                x=v.x;y=v.y;z=v.z;
                        }

void normalize()
        {
          // Calculate the magnitude of our vector
          float magnitude = sqrt((x * x) + (y * y) + (z
* z));

          // As long as the magnitude isn't zero, divide
each element by the magnitude
          // to get the normalised value between -1
and +1
          if (magnitude != 0)
          {
            x /= magnitude;
            y /= magnitude;
            z /= magnitude;
          }
        }

void cross( Vector3 &vec1,  Vector3 &vec2)
        {
          x=vec1.y * vec2.z - vec1.z * vec2.y;
           y= vec1.z * vec2.x - vec1.x * vec2.z;
           z= vec1.x * vec2.y - vec1.y * vec2.x;
        }
float dot( Vector3 &vec1, Vector3 &vec2)
        {
          return vec1.x * vec2.x + vec1.y * vec2.y +
vec1.z * vec2.z;
        }

};
```




```
class camera1
{
private:
Point3 eye;
Vector3 u,v, n,Sample;
double viewAngle, aspect, nearDist, farDist; //view volume shape
void setModelViewMatrix();
public:
//default constructor
camera1(){}
void setShape(float vAng, float asp, float nearD, float farD);//set viewvolume
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
};
```



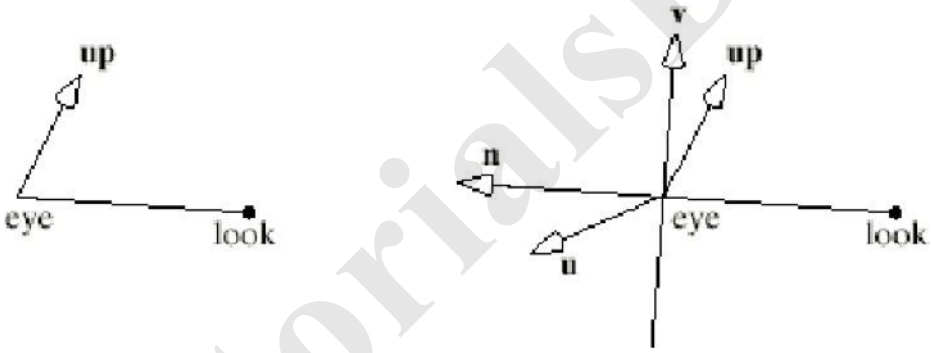
- It is useful to attach an explicit co-ordinate system to the camera as shown in the figure above.
- This co-ordinate system has its origin at the eye and has three axes, usually called the u-,v-, and n-axis which define the orientation of the camera

The General Camera with Arbitrary Orientation and Position

- The axes are pointed in directions given by the vectors u,v, and n
- Because, by default, the camera looks down the negative z-axis, we say in general that the camera looks down the negative n-axis in the direction -n
- The direction u points off "to the right of" the camera and the direction v points "upward"

Camera Control

set function



- What are the directions of **u**,**v** and **n** when we execute set() with given values for **eye**, **look** and **up**
- If given the locations of eye, look and up, we immediately know that **n** must be parallel to the vector **eye-look**, as shown above. so we can set **n=eye-look**
- We now need to find a **u** and a **v** that are perpendicular to **n** and to each other.
- The **u** direction points *"off to the side"* of a camera, so it is natural to make it perpendicular to up which the user has said is the *"upward"* direction.
- An easy way to build a vector that is perpendicular to two given vector is to form their cross product, so we set **u=up x n**
- With **u** and **n** formed it is easy to determine **v** as it must be perpendicular to both and is thus the cross product of **u** and **n** thus **v=n x u**
- Notice that **v** will usually not be aligned with up as **v** must be aimed perpendicular to **n** whereas the user provides up as a suggestion of *"upwardness"* and the only property of up that is used is its cross product with **n**

To summarise, given eye look and up, we form

n=eye−look

u=up×n

v=n×u

we then normalize to unit length

Viewing matrix

- The job of the View matrix is to convert world co-ordinates to camera co-ordinates it must transform the camera's coordinate system into the generic position for the camera
- An orthographic projection is specified by setting "clip" planes
- We do this using
  - near - far
  - left - right
  - top - bottom
- We can then create a matrix to scale and set the view of our object

cam.set(Point3(4, 4, 4), Point3(0, 0, 0) ,Vector3(0, 1, 0));

```
void camera1 :: set (Point3 Eye, Point3 look, Vector3 up)
{ 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.cross(up,n);
 u.set(u);          //make u= up X n
n.normalize(); // make them unit length
u.normalize();
v.cross(n,u);
v.set(v);          // make v= n X u
v.normalize();
setModelViewMatrix();          // tell OpenGL
}
```

setModelViewMatrix();

It is used only by member functions of the class and needs to be called after each change is made to the camera'sposition.Thematrix V transforms the world points into camera coordinates.

$$V = \begin{pmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

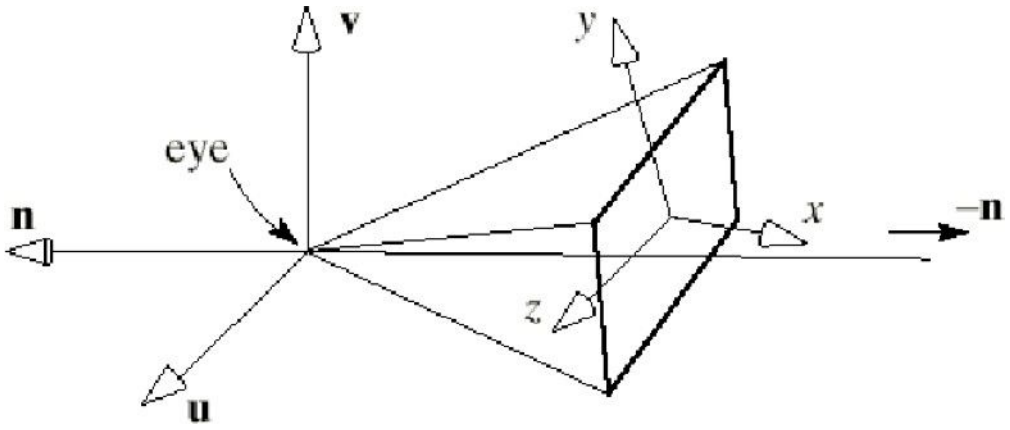
(d_x ,d_y, d_z)=(-eye.u,-eye.v,-eye.n)

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().

```
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);                    //loadOpenGL'smodelviewmatrix
}
```

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().

Setting the View Volume



- OpenGL 1.x provided a simple way to set the view volume in a program by setting the projection Matrix
- This was done using gluPerspective
- given in degrees and sets the angle between the top and bottom walls of the pyramid.
- The parameters w and h sets the aspect ratio of any window parallel to the xy-plane
- The value N is the distance from the eye to the near plane, and F is the distance from the eye to the far plane. N and F should be positive.
- gluPerspective(viewAngle, aspect, N, farDist);

cam.setShape(30.0f,64.0f/48.0f,0.5f,50.0f) ;

It is used only by member functions of the class and needs to be called after each change is made to the camera'sposition.Thematrix V transforms the world points into camera coordinates.

```
void camera1 :: setShape(float vAng, float asp, float nearD, float farD)
    {
        viewAngle = vAng;
        aspect = asp;
        nearDist = nearD;
        farDist = farD;
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();
        gluPerspective(viewAngle, aspect, nearDist, farDist);
    }
```

Example:

```
        cam.set(eye, look, up);              // initialize the camera
        cam.roll(30);                  // roll it through 30 degree
      cam.yaw(20);          // yaw it through 20 degree
```

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:

```
cam.slide(-1, 0, -2);                    //slide the camera forward and to the left
```

```
void camera1 :: 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

$$u' = \cos(a)u + \sin(a)v ;$$
$$v' = -\sin(a)u + \cos(a)v$$

```
void camera1 :: 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 camera1 :: 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 camera1 :: 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();
}
```

```
camera1 cam; //global camera object
void myKeyboard(unsigned char key, int x, int y) //---------------------- myKeyboard----------------
{
switch(key)
{
//controls for the camera
case 'F' :          //slide camera forward
 cam.slide(0, 0, 0.2);
break;
case 'B':          //slide camera back
cam.slide(0, 0,-0.2);
break;
case 'P':
cam.pitch(-1.0);
break;
case 'Q':
cam.pitch(1.0);
break;
case 'Y':
cam.yaw(1.0);
break;
case 'Z':
cam.yaw(-1.0);
break;
case 'R':
cam.roll(1.0);
break;
case 'S':
cam.roll(-1.0);
break;
}
glutPostRedisplay(); //draw it again
}
void initGL() {
  glClearColor(1.0f, 1.0f, 1.0f, 1.0f); // Set background color to black and opaquel
  glColor3f(0.0f,0.0f,0.0f);
  glViewport(0, 0, 640,480); // Set background depth to farthest
 }
/* Handler for window-repaint event. Called back when the window first appears and
  whenever the window needs to be re-painted. */
void display() {
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear color ,depth buffers
  glutWireTeapot(1.0);
  glFlush();
  glutSwapBuffers(); // Swap the front and back frame buffers (double buffering)
}
```

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

```
void reshape(GLsizei width, GLsizei height) { // GLsizei for non-negative integer
  // Compute aspect ratio of the new window
  if (height == 0) height = 1;                // To prevent divide by 0
  GLfloat aspect = (GLfloat)width / (GLfloat)height;
  // Set the viewport to cover the new window
  glViewport(0, 0, width, height);
   // Set the aspect ratio of the clipping volume to match the viewport
  glMatrixMode(GL_PROJECTION); // To operate on the Projection matrix
  glLoadIdentity();            // Reset
  // Enable perspective projection with fovy, aspect, zNear and zFar
  gluPerspective(45.0f, aspect, 0.1f, 100.0f);
}
//-------------------------main----------------------------
int main(int argc, char **argv)
{glutInit(&argc, argv);           // Initialize GLUT
glutInitDisplayMode(GLUT_DOUBLE); // Enable double buffered mode
glutInitWindowSize(640, 480); // Set the window's initial width & height
glutInitWindowPosition(50, 50); // Position the window's initial top-left corner
glutCreateWindow("xyz");
glutKeyboardFunc(myKeyboard); // Create window with the given title
glutDisplayFunc(display);       // Register callback handler for window re-paint event
//glutReshapeFunc(reshape);       // Register callback handler for window re-size event
initGL();
cam.set(Point3(4, 4, 4), Point3(0, 0, 0) ,Vector3(0, 1, 0));
cam.setShape(30.0f,64.0f/48.0f,0.5f,50.0f)   ;    // Our own OpenGL initialization
glutMainLoop();                // Enter the infinite event-processing loop
return 0;
}
```

Topic6

Creating shaded objects

Establishing a point lightsource in the scene and assigning various material properties , include ambient,diffuse and specular light provide key strokes that switches between flat and smooth shading

Step1:

To Use Light Sources in OpenGL

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

```
1)GLfloat myLightPosition[]={3.0 , 6.0,5.0,1.0 };
2)glLightfv(GL_LIGHT0, GL-POSITION, myLightPosition);
3)glEnable(GL_LIGHTING);        //enable lighting in general
4)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.

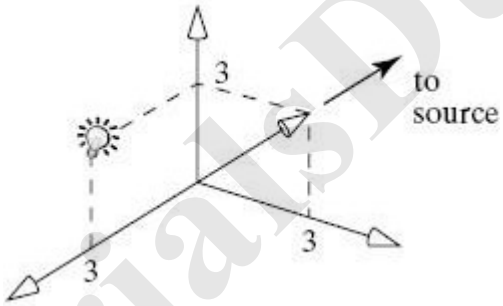
Some sources such as desk lamp are in the scene whereas like the sun are infinitely remote.

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)

A local source and an infinitely remote 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:

```
5)GLfloat amb0[]={ 0.2 , 0.4, 0.6, 1.0 };                    // define some colors
6)GLfloat diff0[]= { 0.8 ,0.9 , 0.5 ,1.0 };
7)GLfloat spec0[]= { 1.0 , 0.8 , 1.0, 1.0 };
8)glLightfv(GL_LIGHT0, GL_AMBIENT, amb0); //attach them to LIGHT0
9)glLightfv(GL_LIGHT0, GL_DIFFUSE, diff0); 10)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

Step2:

This version of glMaterialfv takes vector-based color coordinates. The GL_AMBIENT_AND_DIFFUSE parameter specifies that colorBlue will be applied to both ambient and diffuse components of the material.

```
// Evaluate the reflective properties of the material
   11)float colorBlue[] = { 0.0f, 0.0f, 1.0f, 1.0f };
   12) glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, colorBlue);
```

The first parameter of the glMaterialfv command described above (GL_FRONT) indicates which face of the polygon should reflect the light specified by colorBlue.

now, draw polygon as its material properties will be affected by the glMaterialfv call.

Step3:

```
   // Draw a polygon with current material properties being set
   13)glBegin(GL_TRIANGLES);
   glVertex3f(-1.0f, 0.0f, 0.0f);
   glVertex3f(0.0f, -1.0f, 0.0f);
   glVertex3f(1.0f, 0.0f, 0.0f);
   glEnd();
```

Step4:

FlatShading

```
14)glShadeModel(GL_FLAT);
```

SmoothShading

```
15)SmoothShading(GL_SMOOTH);
```

Topic7:Rendering Textures

Applying Textures

```
#include "RGBpixmap.h"
################globals################
RGBpixmap pix[6]; //make six(empty)pixmaps
Float xspeed=0, yspeed=0, xAngle=0.0,yAngle=0.0;
//<<<<<<<<<<<<<<myinit>>>>>>>>>>>>>>>>>>
Void myinit(void)
{
 glClearColor(1.0f,1.0f,1.0f,1.0f); //background is white
  glEnable(GL_DEPTH_TEST);
 glEnable(GL_TEXTURE_2D);
  pix[0].makeCheckerboard();     //make pixmap procedurally
 pix[0].setTexture(2001);          // create texture
  pix[1].readBMPFile("Mandrill.bmp");    //make pixmap from image
  pix[1].setTexture(2002);         //create texture
___similarly for other four textures____
 glViewport(0.0,640,480);       //set up the viewing system
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
 gluPerspective(60.0,640,0/480,1.0,30.0); //set camera shape
  glMatrixMode(GL_MODELVIEW);
 glLoadidentity();
 glTranslated(0.0,0.0.-4); // move camera back
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<display>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Void display(void)
{
 glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
  glTexEnv f(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,GL_DECAL);
  glPushMatrix();
  glRoatated(xAngle,1.0,0.0,0.0); glRotated(yAngle,0.0,1.0,0.0);
  glBindTexture(GL_TEXTURE_2D,2001);    // top face:'fake' checkerboard
 glBegin(GL_QUADS);
 glTexCoord2f(-1.0,-1.0);    glVertex3f(-1.0f,1.0f,-1.0f);
 glTexCoord2f(-1.0,2.0);    glVertex3f(-1.0f,1.0f,1.0f);
glTexCoord2f(2.0,2.0);    glVertex3f(1.0f,1.0f,1.0f);
glTexCoord2f(2.0,-1.0);    glVertex3f(1.0f,1.0f,-1.0f);
glEnd();
  glBindTexture(GL_TEXTURE_2D,2002);    // right face:mandrill
 glBegin(GL_QUADS);
 glTexCoord2f(0.0,0.0);    glVertex3f(1.0f,-1.0f,-1.0f);
 glTexCoord2f(0.0,2.0);    glVertex3f(1.0f,-1.0f,-1.0f);
 glTexCoord2f(2.0,2.0);    glVertex3f(1.0f,1.0f,-1.0f);
 glTexCoord2f(2.0,0.0);    glVertex3f(1.0f,1.0f,1.0f);
.....similarly for other four faces.....
glFlush();
glPopMatrix();
glutSwapBuffers();
```

The code uses a number of OpenGL functions to establish the six textures out of which 5 bitmap textures 1 procedural texture and to attach them to the walls of the cube.

Step1: OpenGL uses textures that are stored in **pixel maps**, or pixmaps for short.

pixmap is a simple array of pixel values, each pixel value is red, green, and blue color values of texture image so for six texture images we need six pixmaps

The method readBMPFile() reads a (24-bit) BMP file and stores the pixel values in its pixmap object;

- Our example OpenGL application will use six textures. To create them we first make an RGBpixmap object for each:
  RGBpixmap pix[6]; // create six (empty) pixmaps

- **Making a texture from a stored image.**
- OpenGL offers no support for reading an image file and creating the pixel map in memory.
- The method readBMPFile(), available on the book's companion website, provides a simple way to read a BMP image into a pixmap. For instance,
  pix[1].readBMPFile("mandrill.bmp");
  reads the file mandrill.bmp and creates the pixmap in pix[1].
- Once the pixel map has been created, pix[1].setTexture() is used to pass the pixmap to OpenGL to make a texture.

- We then load the desired texture image into each one.
-
- Finally each one is passed to OpenGL to define a texture.

- **Making a procedural texture.**
- We create a checkerboard texture using the method makeCheckerboard() and store it in pix[0]:
  pix[0].makeCheckerboard();
- glTexImage2D() to create the actual texture for OpenGL.

Step2:

- Once we have a pixel map, we must bind it to a unique integer (its name) to refer to it in OpenGL.
- We arbitrarily assign the names 2001, 2002, …, 2006 to our six textures.
  - OpenGL can supply unique names: If we need six unique names we can build an array to hold them: GLuint name[6]; and then call glGenTextures(6,name). OpenGL places six unique integers in name[0],…,name[5], and we subsequently refer to the i^{th} texture using name[i].
- The texture is created by making certain calls to OpenGL, which we encapsulate in the method:
  void RGBpixmap :: setTexture(GLuint textureName)

The intensity I is **replaced** by the value of the texture. (In color, the replacement is for each of the R, G, B components.)

In OpenGL,for I = texture (s,t) we use
glTexEnvf (GL_TEXTURE_ENV, GL_TEX_ENV_MODE, GL_REPLACE);
GL_REPLACE and GL_DECAL are equivalent.

Modulating Reflection Coefficient

- Color of an object is color of its diffuse light component; vary diffuse reflection coefficient.
- I = texture (s,t) *[$I_a\rho_a$ + $I_d\rho_d$ x lambert] + $I_{sp}\rho_s$ x phong.
  - The specular component is the color of the light, not the object.
- In OpenGL, use glTexEnvf (GL_TEXTURE_ENV, GL_TEX_ENV_MODE, GL_MODULATE);

```
void RGBpixmap :: setTexture(GLuint textureName)
{
        glBindTexture(GL_TEXTURE_2D, textureName); glTexParameteri(GL_TEXTURE_2D,
        GL_TEXTURE_MAG_FILTER,GL_NEAREST); glTexParameteri(GL_TEXTURE_2D,
        GL_TEXTURE_MIN_FILTER,GL_NEAREST);
   glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, nCols, nRows, 0, GL_RGB,
GL_UNSIGNED_BYTE, pixel);
}
```

- The call to glBindTexture() binds the given name to the texture being formed. When this call is made at a later time, it will make this texture the active texture.
- The calls to glTexParameteri() specify that a pixel should be filled with the texel whose coordinates are nearest the center of the pixel, for both enlarged or reduced versions. This is fast but can lead to aliasing effects.
- Finally, the call to glTexImage2D() associates the pixmap with this current texture. This call describes the texture as 2D consisting of RGB byte-triples, gives its width, height, and the address in memory (pixel) of the first byte of the bitmap.

- Texture mapping must also be enabled with glEnable(GL_TEXTURE_2D).
- The call glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST) is used to request that OpenGL render the texture properly (using hyperbolic interpolation), so that it appears correctly attached to faces even when a face rotates relative to the viewer in an animation.
- Complete code is in Fig. 8.49. The texture creation, enabling, and hinting is done once, in an initialization routine.
- In display() the cube is rotated through angles xAngle, and yAngle, and the faces are drawn. The appropriate texture must be bound to the face, and the texture coordinates and 3D positions of the face vertices be specified inside a glBegin()/glEnd() pair.
- Once the rendering (off screen) of the cube is complete, glutSwapBuffers() is called to make the new frame visible.
- Animation is controlled by the callback “idle” function spinner(). Whenever there is no user input, spinner is called; it alters the rotation angles of the cube slightly, and calls display().

UNIT V

FRACTALS

Fractals and Self similarity – Peano curves – Creating image by iterated functions –Mandelbrot sets – Julia Sets – Random Fractals

A French/American mathematician Dr Benoit Mandelbrot iscovered Fractals. The word fractal was derived from a
Latin word fractus which means broken.

- Fractals are very complex pictures generated by a computer from a single formula.
- They are created using iterations. This means one formula is repeated with slightly different values over and over again, taking into account the results from the previous iteration.
- Fractals are used in many areas such as −

Astronomy − For analyzing galaxies, rings of Saturn, etc.
Biology/Chemistry − For depicting bacteria cultures, Chemical reactions, human anatomy, molecules, plants,
Others − For depicting clouds, coastline and borderlines, data compression, diffusion, economy, fractal art, fractal music, landscapes, special effect, etc.

Examples of Fractals

- Clouds
- Grass
- Fire
- Modeling mountains (terrain)
- Coastline
- Branches of a tree
- Surface of a sponge
- Cracks in the pavement
- Designing antennae

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.

Types: Fractals can also be classified according to their self-similarity

Exactly self-similar:
if a region is enlarged the enlargement looks exactly like the original.

Statistically self-similar:
A **coastline** or a **seashore** is the area where land meets

the sea or ocean, or a line that forms the boundary between the land and the ocean .

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

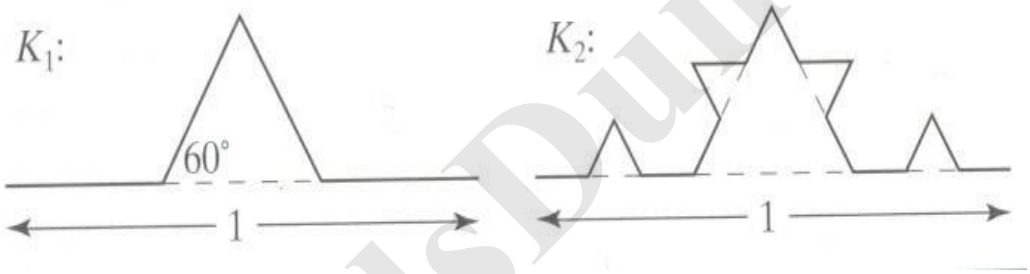
Koch Curves:

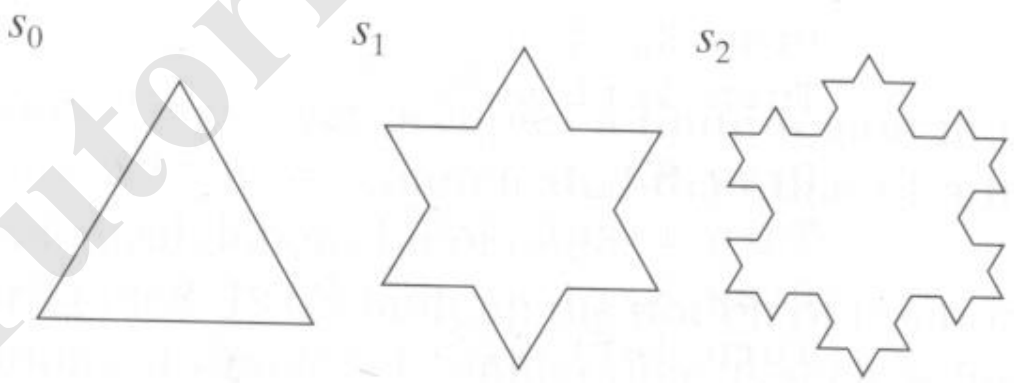
Discovered in 1904 by Helge von Koch Start with straight line of length 1
Recursively:

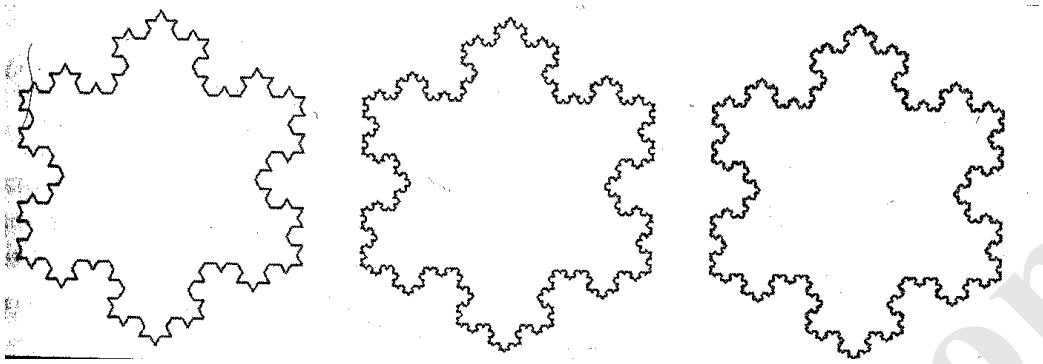
Divide line into 3 equal parts
Replace middle section with triangular bump with sides of length 1/3
New length = 4/3

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.







- The curve produces an infinitely long line within a region of finite area.
  Successive generations of the Koch curve are denoted K0,K1,K2….
  The zeroth generation shape K0 is a horizontal line of length unity.
- To create K1 , divide the line K0 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 K2, is formed by building a bump on each of the four line segments of K1.
  To form Kn+1 from Kn:
- Subdivide each segment of Kn 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 Ki has total length of **$(4/3)^i$** , which increases as i increases.
As i tends to infinity, the length of the curve becomes infinite.

Can form Koch snowflake by joining three Koch curves
Perimeter of snowflake grows as:

where P_i is the perimeter of the I*th* snowflake iteration

- However, area grows slowly and $S_\infty = 8/5$!!
- Self-similar:
  - zoom in on any portion
  - If *n* is large enough, shape still same
  - On computer, smallest line segment > pixel spacing

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);
         }
  }
```

We call n the order of the curve Kn, and we say the order –n Koch curve consists of four versions of the order (n-1) Koch curve.

To draw K2 we draw a smaller version of K1 , then turn left 60 , draw K1 again, turn right 120 , draw K1 a third time.

For snowflake this routine is performed just three times, with a 120 turn in between.

5.3 Peano curves

The Peano curves are amongst the first known fractals curves. They were described the first time in 1890 by the italian mathematician Guiseppe Peano.

Construction:

- Start from a octogon drawn inside a square.
- Divide the octogon in nine smaller octogons.
- Link the outer octogons two by two and then link all those couples to the entral octogon.

- Applying the same procedure to the nine small octogon gives rise to the drawing showed here:

Properties :

- **Fractal Dimension**

The fractal dimension is computed using the Hausdorff-esicovitch equation (self-similarity method):

D = log (N) / log (r)

Replacing r by three (as each segment is divided by three on each iteration) and N by nine (as the drawing process yields 9 smaller octogons) in the Hausdorff-Besicovitch equation gives:

D = log(9) / log(3) = 2

- **Self-Similarity**

  -Looking at two successive iterations of the drawing process provides graphical evidence that this property is also shared by this curve.

- The Peano curve is a closed curve. The inner space can be filled to increase the contrast with the surrounding background

5.3 Creating An Image By Means of Iterative Function Systems

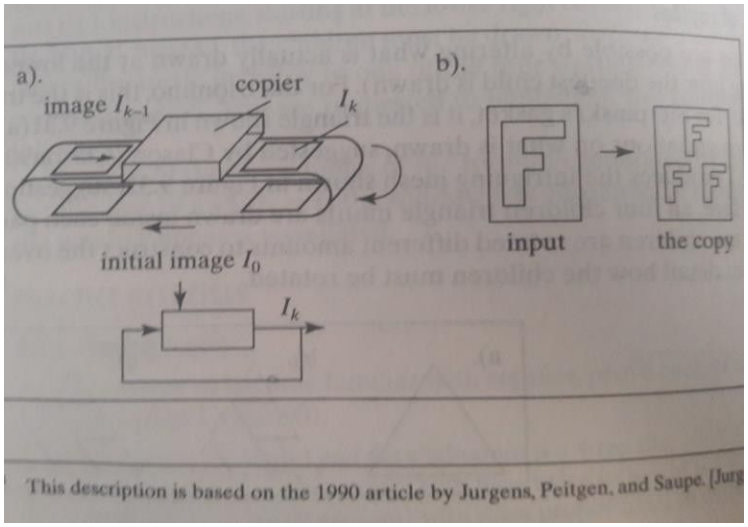
Another way to approach infinity is to apply a transformation to a picture again and again and examine the results. This technique also provides an another method to create fractal shapes.

5.3.1 An Experimental Copier

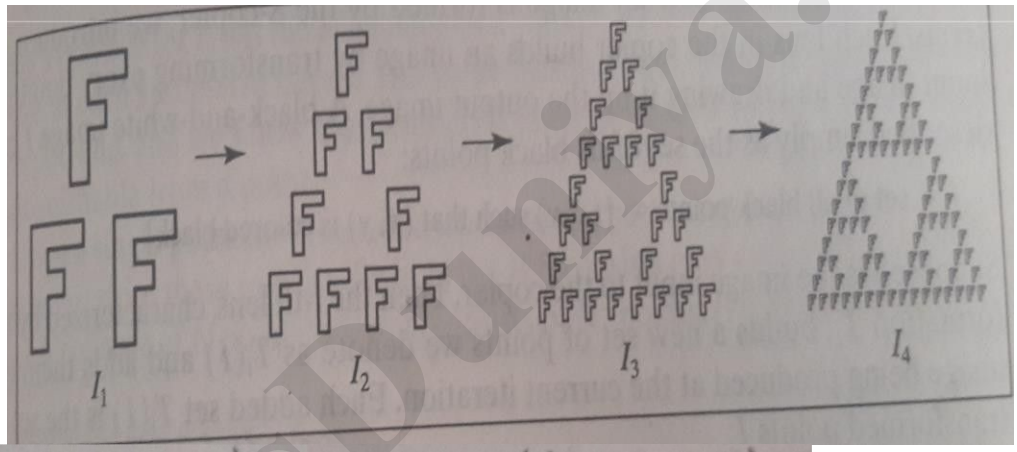
We take an initial image I_0 and put it through a special photocopier that produces a new image I_1. I_1 is not a copy of I_0 rather it is a superposition of several reduced versions of I_0. We then take I_1 and feed it back into the copier again, to produce image I_2. This process is repeated , obtaining a sequence of images I_0, I_1, I_2,... called the **orbit of Io**.

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.



This description is based on the 1990 article by Jurgens, Peitgen, and Saupe. [Jurg



$$M_1 = \begin{pmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad M_2 = \begin{pmatrix} \frac{1}{2} & 0 & 0 \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

$$M_3 = \begin{pmatrix} \frac{1}{2} & 0 & \frac{1}{4} \\ 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

Affine Transformation :
T ={m11,m12,m21,m22,m13,m23}
The first four terms contain the elements that performs scalling and rotation.

The final 2 terms perform Translation
Thus the three maps of the s-copiers are
T1={ ½,0,0, ½,0,0}
T2={ ½,0,0, ½,½,0}
T3={ ½,0,0, ½ , ¼,¼}

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^1,y^1) | (x^1,y^1) = $T_i(P)$ 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 = $T_1(I) \cup T_2(I) \cup T_3(I)$

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:

$W(.)=T_1(.) \cup T_2(.) \cup T_3(.)$

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.

  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.

5.4 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 Iterated function systems for Julia and Mandelbrot sets

The IFS uses the simple function

$$f(z) = z^2 + c \qquad \text{----------(1)}$$

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

$$p_+, p_- = \frac{1}{2} \quad \frac{1}{4} \quad c \qquad \text{----------(4)}$$

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

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 inequation
            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.

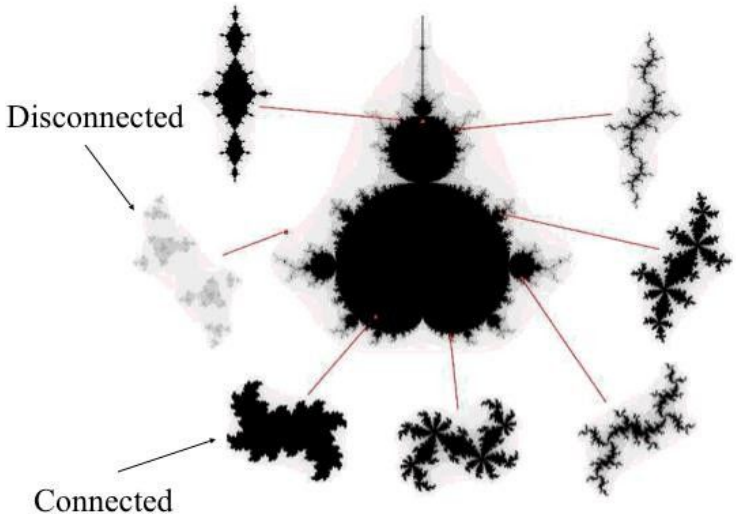
5.5 JULIA SETS

The Julia Set(s) are a slightly different than the Mandelbrot Set as I will explain. For every POINT in the complex plane, there exists a set called the Julia Set each of which has a different shape and a different dynamic associated with it. In other words, every point in the complex plane generates a different Julia Set "curve" or image.

Also, it has been shown that the Julia Sets generated from the points on the INSIDE of the Mandelbrot Set form Connected Sets, and the Julia Sets that are generated from the points on the OUTSIDE of the M-Set form disconnected sets also known as Cantor dusts.

- Julia sets are extremely complicated sets of points in the complex plane.
- There is a different Julia set, denoted Jc for each value of c.
  A closely related variation is the filled-in Julia set,
- denoted by Kc, which is easier to define.

The Julia Set (s)



5.5.1 The Filled-In Julia Set K_c

- The filled-in Julia set at c, Kc, is the set of all starting points whose orbits are finite.
- When studying Kc, one choosesa single value for c and considers different starting points.
- Kc 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.

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well

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 $f(.) = (.)^2 + c$.

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

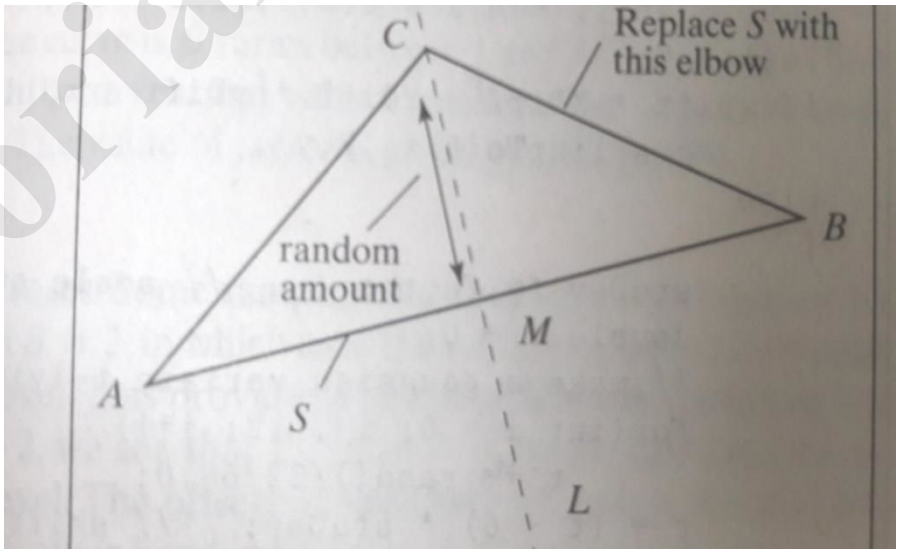
5.6 RANDOM FRACTALS

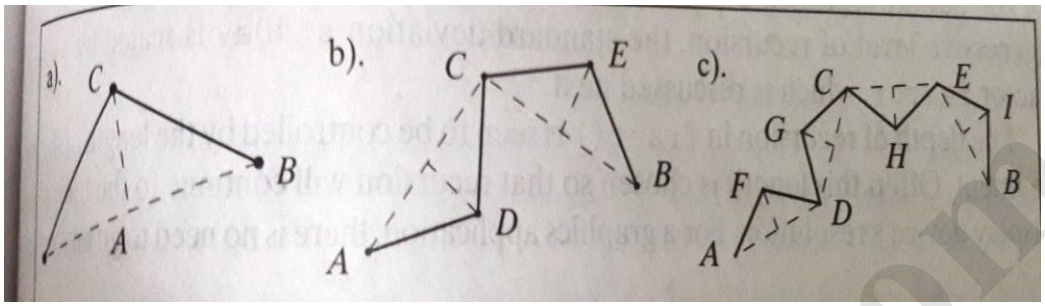
Fractal is the term associated with randomly generated curves and surfaces that exhibit a degree of self-similarity. These curves are used to provide "naturalistic" shapes for representing objects such as coastlines, rugged mountains, grass and fire.

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.





- 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 final stage, the new points F…..I are added.

To perform afractalization 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:

$$C(t) = M + (B-A)^{\perp} t$$

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 to0 and sc
C.x = 0.5 *(A.x +B.x) – t * (B.y – A.y);
C.y = 0.5 *(A.y +B.y) – t * (B.x –A.x);
          fract(A, C, stdDev);
           fract(C, B, stdDev);
}
```

The routine fract() generates curves that approximate actual fractals. The routine recursively replaces each segment in a random elbow with a smaller random elbow.

The stopping criteria used is: When the length of the segment is small enough, the segment is drawn using cvs.lineTo(), where cvs is a Canvas object.

The variable t is made to be approximately Gaussian in its distribution by summing together 12 uniformly distributed random values lying between 0 and 1.

The result has a mean value of 6 and a variance of 1. The mean value is then shifted to 0 and the variance is scaled as necessary.The depth of recursion in fract() is controlled by the length of the line segment.

Controlling the Spectral Density of the Fractal Curve:

The fractal dimension of such processes is:

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.

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:
factor = $2^{(1-\beta/2)}$
The factor decreases as β increases.

factor	β	D	
1.4	1	2	1/f noise
1.1	1.6	1.7	
1.0	2	1.5	Brownian motion
0.9	2.4	1.3	
0.7	3	1	

Figure 9.60. In this routine, `factor` is computed

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); }
```

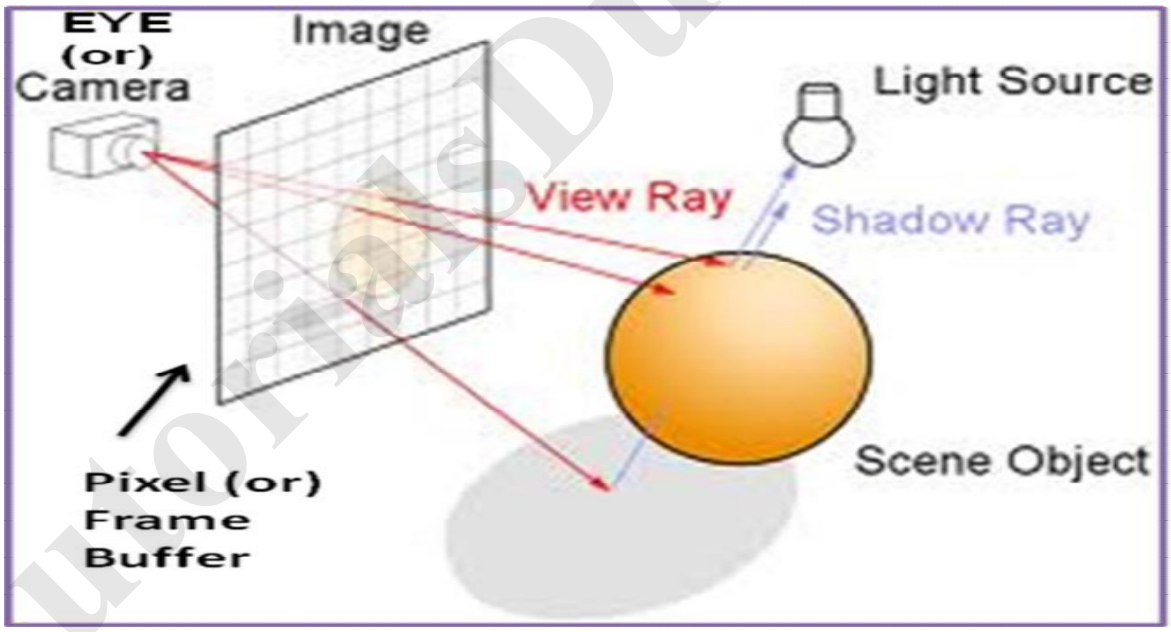
UNIT 6

Overview of Ray Tracing Intersecting rays with other primitives – Adding Surface texture –Reflections and Transparency – Boolean operations on Objects.

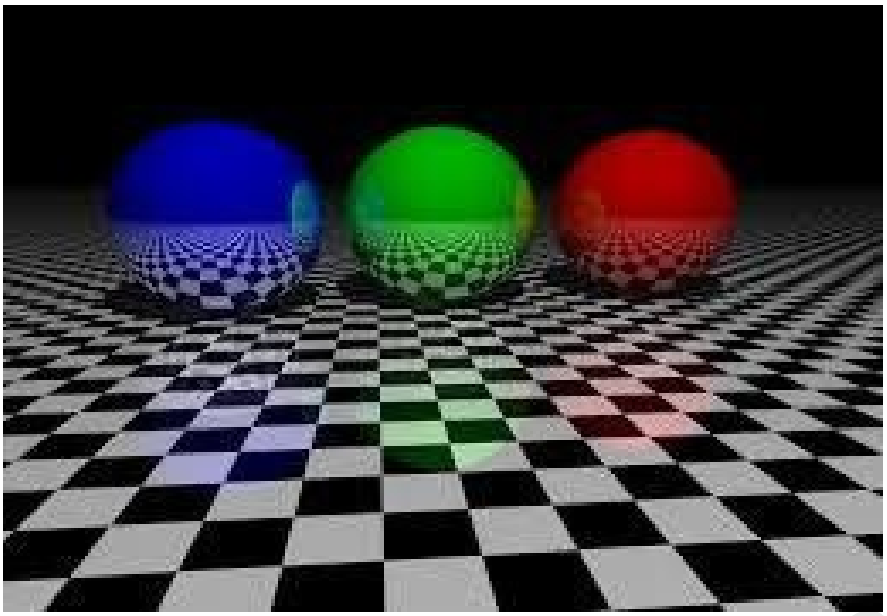
6. 1 OVERVIEW OF RAY TRACING:

- Ray tracing is a technique for generating an image by tracing the path of light through pixels in an image plane and simulating the effects of its encounters with virtual objects.
- The technique is capable of producing a very high degree of visual realism, usually higher than that of typical scan line rendering methods, but at a greater computational cost.

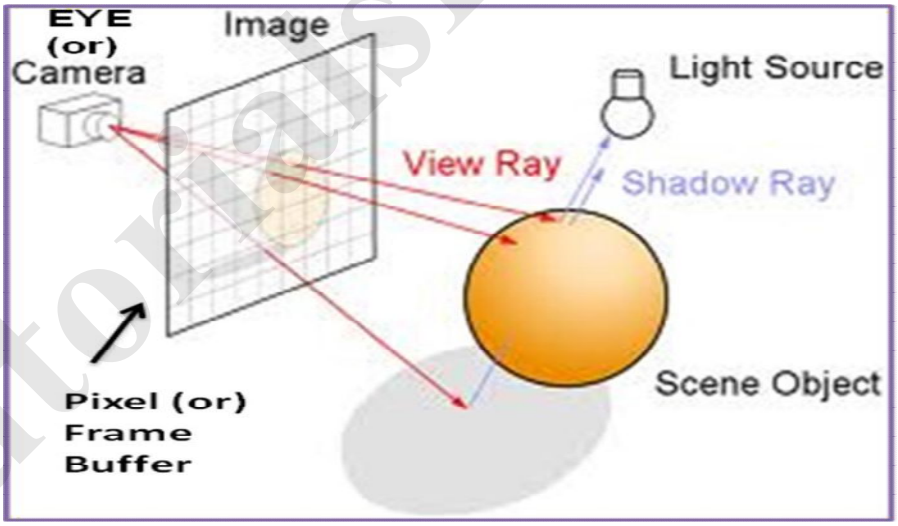
- Ray tracing Provides a related, but even more powerful, approach to rendering scenes.
- A Ray is cast from the eye through the center of the pixel is traced to see what object it hits first and at what point.



- The Resulting color is then displayed at the pixel, the path of a ray traced through the scene, interesting visual effects such as shadowing, reflection and refraction are easy to incorporate and producing dazzling images.



- Ray tracing can create realistic images.
- In addition to the high degree of realism, ray tracing can simulate the effects of a camera due to depth of field and aperture shape
- Ray tracing is capable of simulating a wide variety of optical effects, such as reflection and refraction, scattering,and dispersion phenomena (such as chromatic aberration).
- Descriptions of all then Objects are stored in an **object list.**
- The ray that interacts the Sphere and the Cylinder.
- The hit spot ($\mathbf{P_{HIT}}$) is easily found wit the ray itself.
  The ray of Equation at the Hit time $\mathbf{t_{hit}}$: $\mathbf{P_{HIT}=eye + dir_{r,c}t_{hit}}$



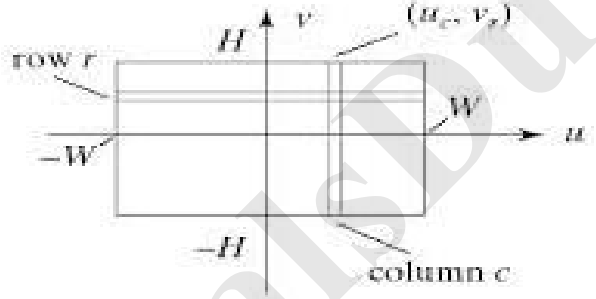
Pseudocode of a Ray Tracer

```
define the objects and light sources in the scene set up the camera
   for(int r=0 ; r < nRows ; r++)
   for(int c=0 ; c < nCols ; c++)
   {
      1. Build the rc-th ray.
      2. Find all interactions of the rc-th ray with objects in the
scene.
      3. Identify the intersection that lies closest to and infront of
the eye.
      4. Compute the Hit Point.
      5. Find the color of the light returning to the eye along the ray
from the point of intersection.
      6. Place the color in the rc-th pixel.
}
```

6.2 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.

PLANE	NAME	EQUATION	OUTWARD NORMAL	SPOT
0	TOP	Y=1	(0,1,0)	(0,1,0)
1	BOTTOM	Y=-1	(0,-1,0)	(0,-1,0)
2	RIGHT	X=1	(1,0,0)	(1,0,0)
3	LEFT	X=-1	(-1,0,0)	(-1,0,0)
4	FRONT	Z=1	(0,0,1)	(0,0,1)
5	BACK	Z=-1	(0,0,-1)	(0,0,-1)

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 $d(t) = ((S_X + C_Xt)^n + (S_Y + C_Yt)^n)^{m/n} + (S_Y + C_Yt)^m - 1$

where n and m are constant that govern the shape of the surface.

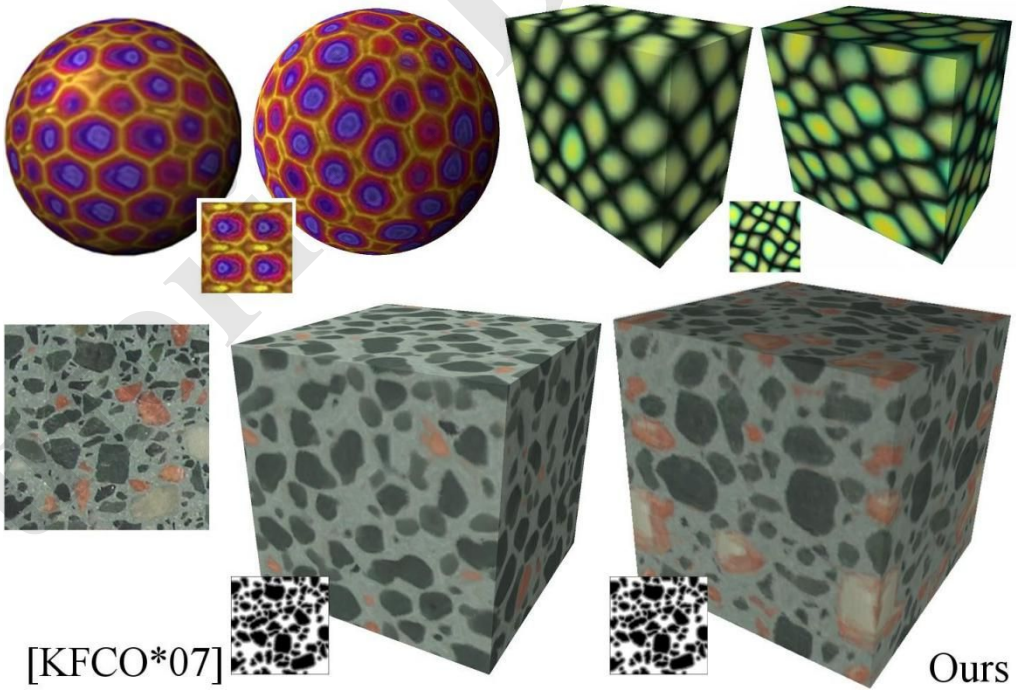
6.3 ADDING SURFACE TEXTURE

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.



6.3.1 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 mVP^{-1} and the inverse texture map transformation mT^{-1} .

6.3.2 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

$$P_u' = \frac{\partial}{\partial u}(P + bn)$$
$$= P_u + b_u n + bn_u$$

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.

6.3.3 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.

6.3.4 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.

6.3.5 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

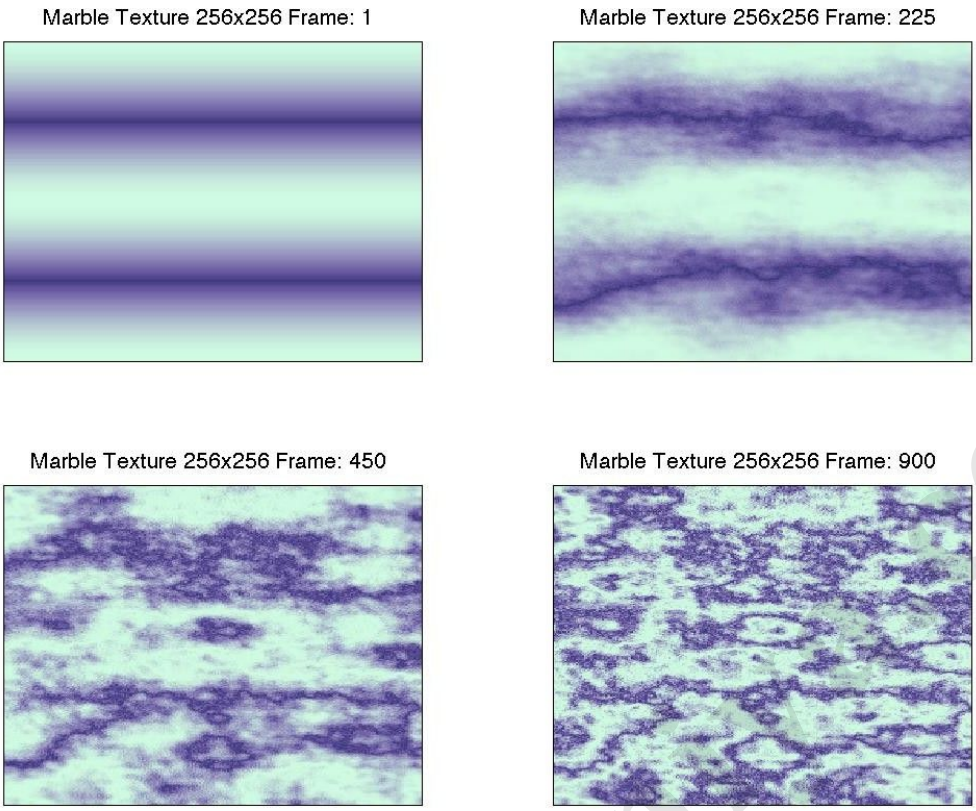
where for rings about z-axis,

the radius $r = \sqrt{x^2+y^2}$.

The value of the function rings () jumps between zero and unity as r increases from zero.

6.3.6 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.



6.3.7 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, etc.

The more rapidly varying components are given progressively smaller strengths

The function

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

$$turb(s, x, y, z) = 1/2 \sum_{k=0}^{m} 1/2^{K} noise(2^{k}s, x, y, z).$$

6.3.8 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.

6.4 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.

6.4.1 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.

6.5 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

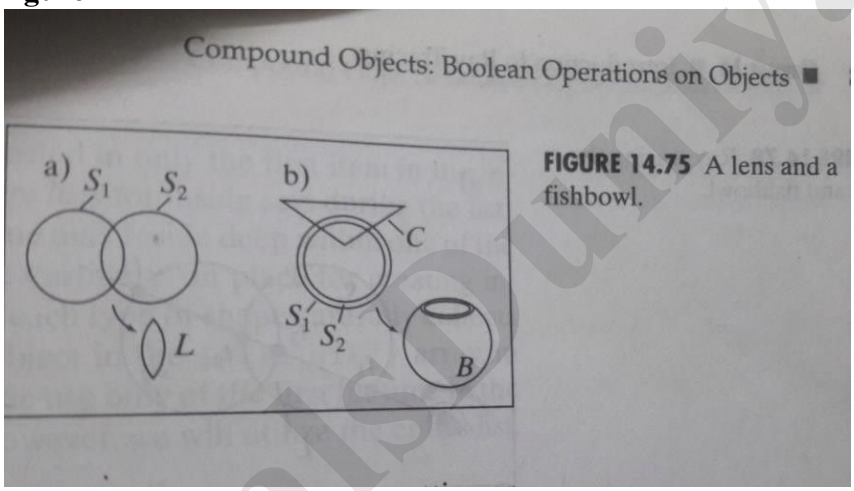


FIGURE 14.75 A lens and a fishbowl.

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 arocket.

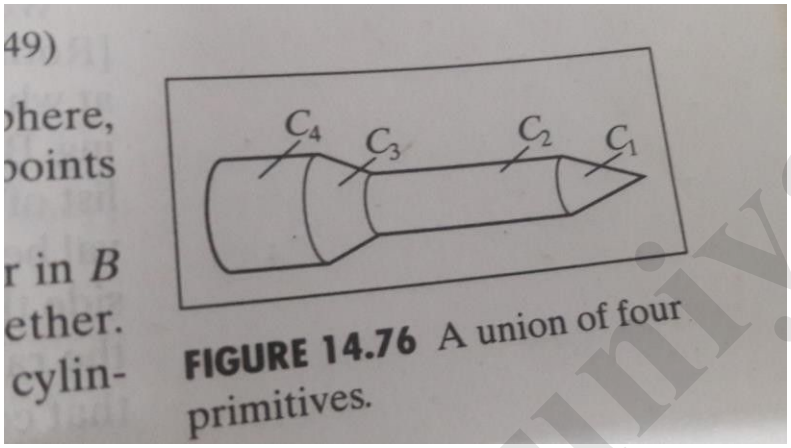


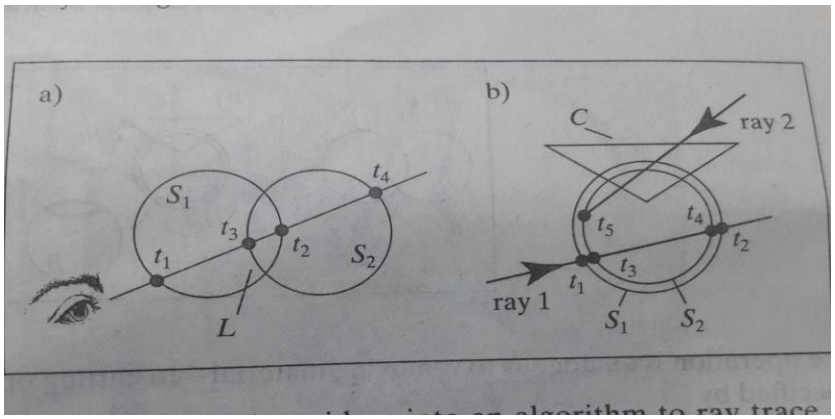
FIGURE 14.76 A union of four primitives.

$R=C_1 \cup C_2 \cup C_3 \cup C_4$.

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.

6.5.1 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



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.

6.5.2 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.

6.5.3 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.

6.5.4 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.

TutorialsDuniya.com

Download FREE Computer Science Notes, Programs, Projects, Books PDF for any university student of BCA, MCA, B.Sc, B.Tech CSE, M.Sc, M.Tech at https://www.tutorialsduniya.com

- Algorithms Notes
- Artificial Intelligence
- Android Programming
- C & C++ Programming
- Combinatorial Optimization
- Computer Graphics
- Computer Networks
- Computer System Architecture
- DBMS & SQL Notes
- Data Analysis & Visualization
- Data Mining
- Data Science
- Data Structures
- Deep Learning
- Digital Image Processing
- Discrete Mathematics
- Information Security
- Internet Technologies
- Java Programming
- JavaScript & jQuery
- Machine Learning
- Microprocessor
- Operating System
- Operational Research
- PHP Notes
- Python Programming
- R Programming
- Software Engineering
- System Programming
- Theory of Computation
- Unix Network Programming
- Web Design & Development

Please Share these Notes with your Friends as well