RGPVNOTES.IN

Program : **B.Tech**

Subject Name: **Computer Graphics & Multimedia**

Subject Code: **IT-601**

Semester: **6th**

Department of Information Technology

Subject Name: Computer Graphics and Multimedia **Subject Code:IT 601**

Subject Notes

Syllabus:2D & 3D Co-ordinate system, Translation, Rotation, Scaling, Reflection Inverse transformation, Composite transformation, world coordinate system, screen coordinate system, parallel and perspective projection, Representation of 3D object on 2D screen, Point Clipping, Line Clipping Algorithms, Polygon Clipping algorithms, Introduction to Hidden Surface elimination, Basic illumination model, diffuse reflection, specular reflection, phong shading, Gourand shading ray tracing, color models like RGB, YIQ, CMY, HSV.

Unit-III

2D & 3D Co-ordinate system:

A two-dimensional Cartesian coordinate system is formed by two mutually perpendicular axes. The axes intersect at the point, which is called the origin. In the right-handed system, one of the axes (x-axis) is directed to the right, the other y-axis is directed vertically upwards. The coordinates of any point on the xy-plane are determined by two real numbers x and y, which are orthogonal projections of the points on the respective axes. The x-coordinate of the point is called the abscissa of the point, and the y-coordinate is called its ordinate.

A three-dimensional Cartesian coordinate system is formed by a point called the origin and a basis consisting of three mutually perpendicular vectors. These vectors define the three coordinate axes: the x–, y–, and z–axis. They are also known as the abscissa, ordinate and applicate axis, respectively. The coordinates of any point in space are determined by three real numbers: x, y, z.

2-D Transformation

In many applications, changes in orientations, size, and shape are accomplished with geometric transformations that alter the coordinate descriptions of objects.

Basic geometric transformations are:

Translation, Rotation, Scaling

Other transformations: Reflection, Shear

Translation:

We translate a 2D point by adding translation distances, tx and ty, to the original coordinate position (x,y):

$$x' = x + tx$$
$$y' = y + ty$$

Alternatively, translation can also be specified by the following transformation matrix:

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

Then we can rewrite the formula as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scaling:

We scale a 2D point by multiplying scaling factor, Sx and Sy, to the original coordinate position (x,y):

$$x' = xSx$$
$$y' = ySy$$

Alternatively, scaling can also be specified by the following transformation matrix:

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

Then we can rewrite the formula as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Scaling is performed about the origin (0,0) not about the centre of the line or polygon.

- Scale > 1 enlarge the object and move it away from the origin
- Scale = 1 leave the object alone
- Scale< 1 shrink the object and move it towards the origin.

Uniform scaling: Sx = Sy

Differential scaling Sx != Sy -> alters proportions

Rotation:

To rotate an object about the origin (0,0), we specify the rotation angle. Positive and negative values for the rotation angle define counter clockwise and clockwise rotations respectively. The following is the computation of this rotation for a point:

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

Alternatively, this rotation can also be specified by the following transformation matrix:

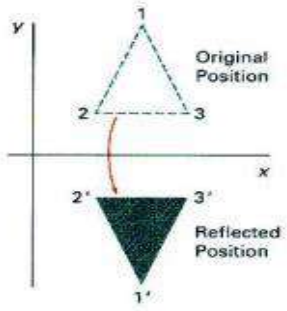
$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Then we can rewrite the formula as:

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

Reflection:

Reflection about the x axis:

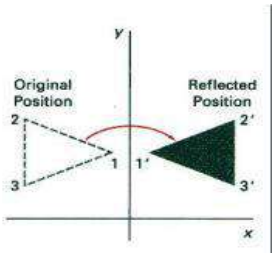


$$x' = x$$
$$y' = -y$$

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

Figure 3.1 Reflection about the x axis

Reflection about the y axis:

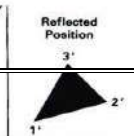


$$x' = -x$$
$$y' = y$$

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

Figure 3.2 Reflection about the y axis

Flipping both x and y coordinates of a point relative to the origin:

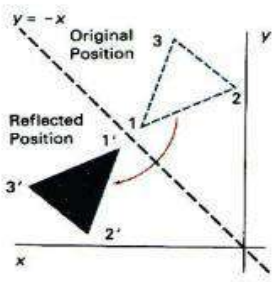


$$x' = -x$$
$$y' = -y$$

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

Figure 3.3 Reflection about the origin

Reflection about the diagonal line y=x:

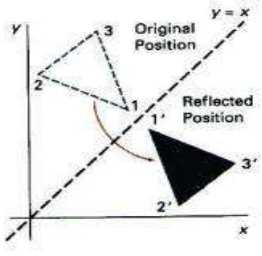


$$x' = y$$
$$y' = x$$

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

Figure 3.4 Reflection about the line y=x

Reflection about the diagonal line y=-x:

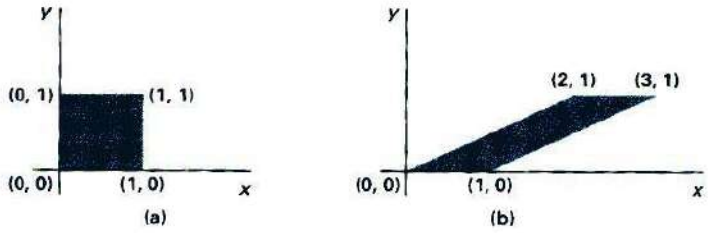


$$x' = -y$$
$$y' = -x$$

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

Figure 3.5 Reflection about the y=-x

Shearing:X-direction shear, with a shearing parameter shx, relative to the x-axis:



$$x' = x + y.\,Shx$$
$$y' = x$$
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & Shx & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Figure 3.6 Shearing about the x axis

Similarly, we can find a y-direction shear, with a shearing parameter shy, relative to the y-axis.

Inverse Transformation

Each geometric transformation has an inverse, which is described by the opposite operation performed by the original transformation. Any transformation followed by its inverse transformation keeps an object unchanged in position, orientation, size and shape. We will use inverse transformation to nullify the effects of already applied transformation.

Homogeneous Coordinates in 2 Dimensions

Scaling and rotations are both handled using matrix multiplication, which can be combined as we will see above. The translations cause a difficulty, however, since they use addition instead of multiplication.

We want to be able to treat all 3 transformations (translation, scaling, and rotation) in the same way - as multiplications.

The solution is to give each point a third coordinate (X, Y, W), which will allow translations to be handled as a multiplication also.

Two triples (X,Y,W) and (X',Y',W') represent the same point if they are multiples of each other e.g. (1,2,3) and (2,4,6).

At least one of the three coordinates must be nonzero.

If W is 0 then the point is at infinity. This situation will rarely occur in practice in computer graphics.

If W is nonzero we can divide the triple by W to get the Cartesian coordinates of X and Y which will be identical for triples representing the same point (X/W, Y/W, 1). This step can be considered as mapping the point from 3-D space onto the plane W=1.

Conversely, if the 2-D Cartesian coordinates of a point are known as (X, Y), then the homogenous coordinates can be given as (X, Y, 1).

Homogeneous co-ordinates for Translation

The homogeneous co-ordinates for translation are given as $T = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$

Homogeneous co-ordinates for Scaling

The homogeneous co-ordinates for Scaling are given as $S = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Homogeneous co-ordinates for Rotation

The homogeneous co-ordinates for Rotation are given as $R = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Composite Transformation

There are many situations in which the final transformation of a point is a combination of several (often many) individual transformations.

Translations

By common sense, if we translate a shape with 2 successive translation vectors: (tx1, ty1) and (tx2, ty2), it is equal to a single translation of (tx1+ tx2, ty1+ ty2).

This additive property can be demonstrated by composite transformation matrix:

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

This demonstrates that 2 successive translations are additive.

Scaling

By common sense, if we scale a shape with 2 successive scaling factors: (sx1, sy1) and (sx2, sy2), with respect to the origin, it is equal to a single scaling of (sx1* sx2, sy1* sy2) with respect to the origin. This multiplicative property can be demonstrated by composite transformation matrix:

$$\begin{bmatrix} Sx1 & 0 & 0 \\ 0 & Sy1 & 0 \\ 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} Sx2 & 0 & 0 \\ 0 & Sy2 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} Sx1.Sx2 & 0 & 0 \\ 0 & Sy1.Sy2 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

This demonstrates that 2 successive scaling with respect to the origin are multiplicative.

Rotations

By common sense, if we rotate a shape with 2 successive rotation angles: θ1 and θ2, about the origin, it is equal to rotating the shape once by an angle θ1+ θ2 about the origin.

Similarly, this additive property can be demonstrated by composite transformation matrix:

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

This demonstrates that 2 successive rotations are additive.

World Coordinate System & Screen Coordinate System

The world coordinate system is used to define the position of objects in the natural world. This system does not depend on the screen coordinate system, so the interval of number can be anything (positive, negative or decimal). Sometimes the complete picture of object in the world coordinate system is too large and complicate to clearly show on the screen, and we need to show only some part of the object. The capability that show some part of object internal a specify window is called windowing and a rectangular region in a world coordinate system is called window. Before going into clipping, you should understand the differences between window and a viewport.

A Window is a rectangular region in the world coordinate system. This is the coordinate system used to locate an object in the natural world. The world coordinate system does not depend on a display device, so the units of measure can be positive, negative or decimal numbers.

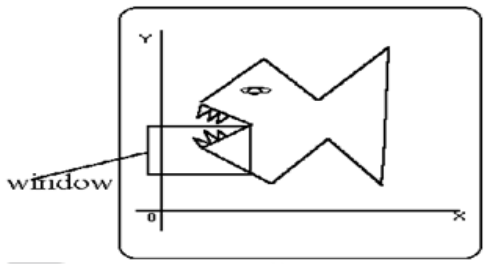


Figure 3.7 Graphical Object in World Coordinate System

A Viewport is the section of the screen where the images encompassed by the window on the world coordinate system will be drawn. A coordinate transformation is required to display the image, encompassed by the window, in the viewport. The viewport uses the screen coordinate system so this transformation is from the world coordinate system to the screen coordinate system.

When a window is "placed" on the world, only certain objects and parts of objects can be seen. Points and lines which are outside the window are "cut off" from view. This process of "cutting off" parts of the image of the world is called Clipping. In clipping, we examine each line to determine whether or not it is completely inside the window, completely outside the window, or crosses a window boundary. If inside the window, the line is displayed. If outside the window, the lines and points are not displayed. If a line crosses the boundary, we must determine the point of intersection and display only the part which lies inside the window.

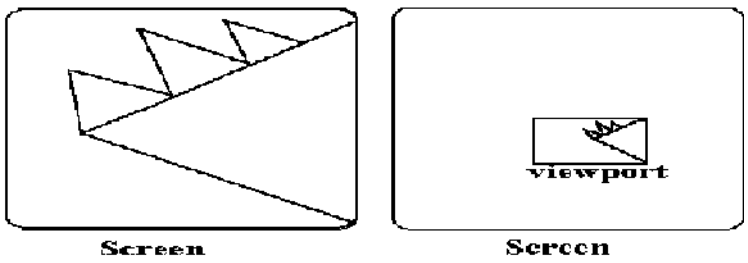


Figure 3.8 Picture in Viewing Coordinate System

Viewing Transformation

An image representing a view often becomes part of a larger image, like a photo on an album page, which models a computer monitor's display area. Since album pages vary and monitor sizes differ from one system to another, we want to introduce a device-independent tool to describe the display area. This tool is called the normalized device coordinate system (NDCS) in which a unit (1 x 1) square whose lower left corner is at the origin of the coordinate system defines the

display area of a virtual display device. A rectangular viewport with its edges parallel to the axes of the NDCS is used to specify a sub-region of the display area that embodies the image.

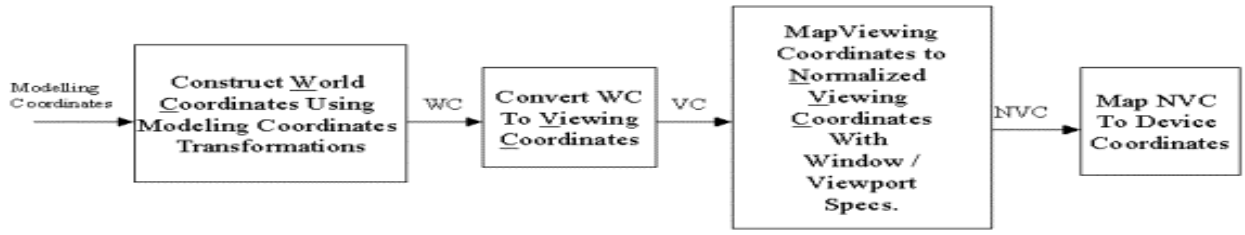


Figure 3.9 2D Viewing-Transformation Pipeline

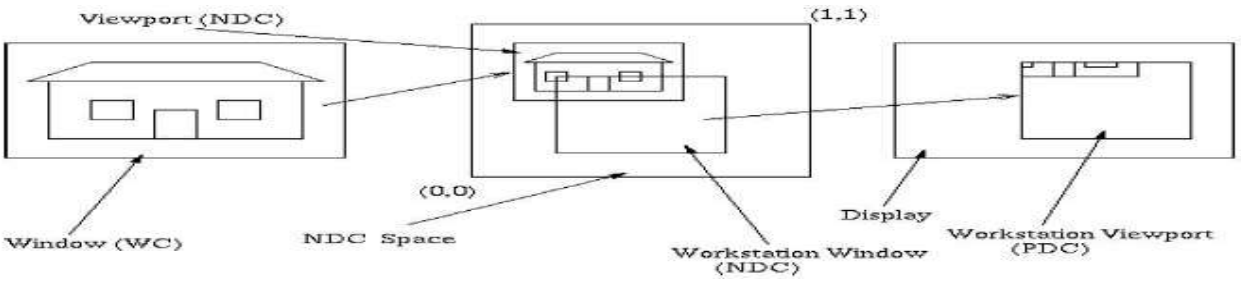


Figure 3.10 Viewing Transformation

The process that converts object coordinates in WCS to normalized device coordinate is called window–to– viewport mapping or normalization transformation. The process that maps normalized device coordinates to Physical Device Co-ordinates (PDC) / image coordinates is called work, station transformation, which is essentially a second window-to-viewport mapping, with a workstation window in the normalized device coordinate system and a workstation viewport in the device coordinate window in the normalized device coordinate system and a workstation viewport in the device coordinate system. Collectively, these two coordinate mapping operations are referred to as viewing transformation.

Workstation transformation is dependent on the resolution of the display device/frame buffer. When the whole display area of the virtual device is mapped to a physical device that does not have a 1/1 aspect ratio, it may be mapped to a square sub-region so as to avoid introducing unwanted geometric distortion.

Parallel & Perspective Projections

Projection operations convert theviewing-coordinate description (3D) to coordinatepositions on the projection plane (2D). Thereare2 basic projection methods:

1. Parallel Projection transforms object positions to the view planealongparallel lines.

A parallel projection preserves relative proportionsof objects. Accurate views ofthe various sides of an object areobtained with a parallel projection. But nota realistic representation

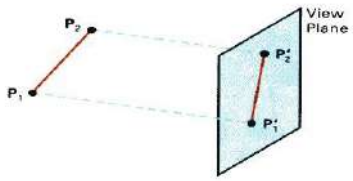


Figure 3.11 Parallel Projection

2. PerspectiveProjectiontransforms object positions to the view planewhile convergingto a center pointof projection. Perspectiveprojection produces realisticviews but doesnot preserve relative proportions. Projections of distant objects aresmaller than the projections of objects of the same sizethat arecloserto the projection plane.

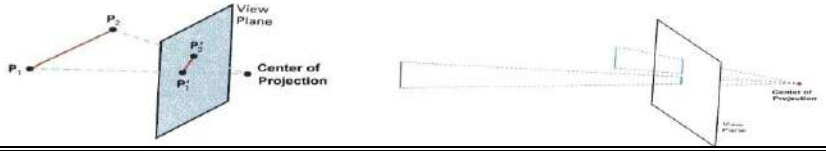


Figure 3.12 Perspective Projection

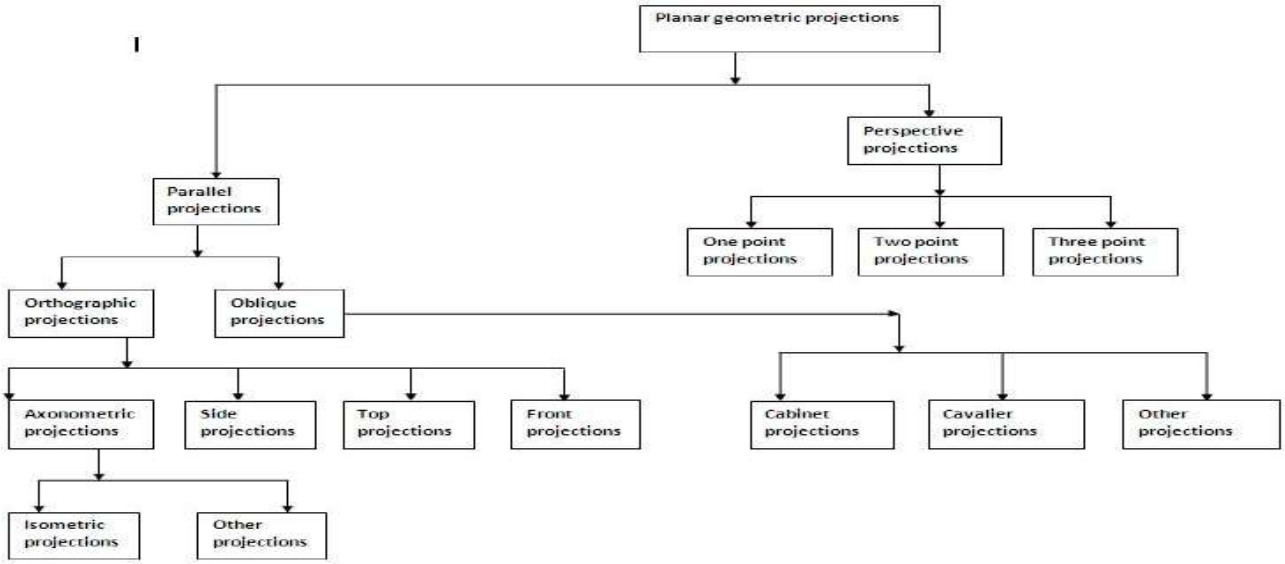


Figure 3.13 Projection Hierarchy

Parallel Projection Classification:

Orthographic Parallel Projection and Oblique Projection:

Figure 3.14 Orthographic Parallel Projection and Oblique Parallel Projection

Orthographic parallel projectionsaredonebyprojectingpoints alongparallel lines that are perpendicular to theprojection plane.

Axonometric: The required transformation is produced in two stages, firstly the direction of projection is rotated until it aligns with one of the co-ordinate axes and then an orthographic projection along that axis is carried out. For example in the isometric case the direction of projection must be symmetric with respect to the three co-ordinate directions to allow equal foreshortening on each axis. This is obviously achieved by using a direction of projection (1,1,1).

Some special Orthographic Parallel Projections involve Plan View (Top projection), SideElevations, andIsometricProjection.

- Front Projection
- Top Projection
- Side Projection

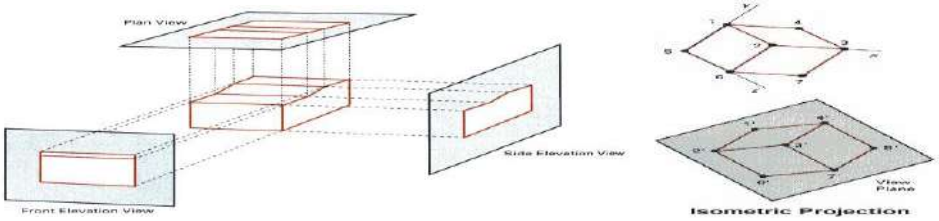


Figure 3.15 Front View, Top View, Side View and Isometric Projection

Oblique projections

Oblique projections areobtained byprojectingalongparallel lines that arenot perpendicular to the projection plane.

The following results canbeobtained from obliqueprojections of acube:

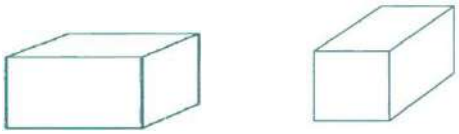


Figure 3.16 Oblique Projection

Main Types of Oblique Projections:

Cavalier: Angle between projectors and projection plane is 45°. Perpendicular faces are projected at full scale.

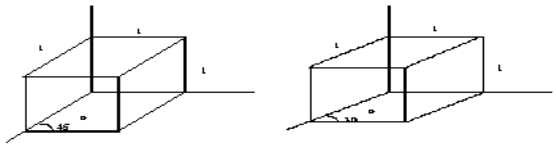


Figure 3.17 Cavalier Projection of Unit Cube

Cabinet: Angle between projectors and projection plane is arc tan (2) = 63.4°. Perpendicular faces are projected at 50% scale.

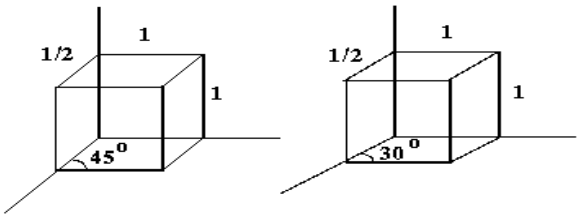


Figure 3.18 Cabinet Projection of Unit Cube

Perspective Projection

In perspective projection, the distance from the center of projection to project plane is finite and the size of the object varies inversely with distance which looks more realistic.

The distance and angles are not preserved and parallel lines do not remain parallel. Instead, they all converge at a single point calledcenter of projection or projection reference point. There are 3 types of perspective projections which are shown in the following chart.

- One-point perspective projection is simple to draw.
- Two-point perspective projection gives better impression of depth.
- Three-point perspective projection is most difficult to draw.

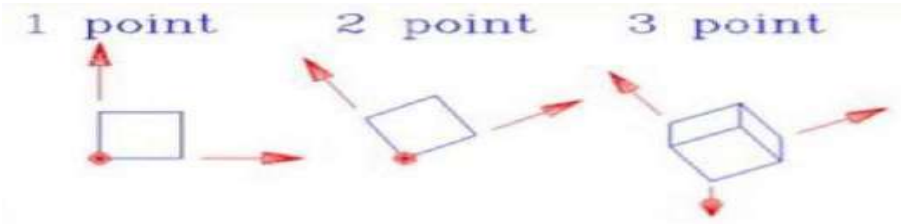


Figure 3.19 Type of Perspective Projection

Representation of 3D Object on 2D Screen (Three–Dimensional Transformations)

With respect to some three- dimensional coordinate system, an object Obj is considered as a set of points.

$$Obj = \{ P (x,y,z)\}$$

If the object moved to a new position, we can regard it as a new object Obj', all of whose coordinate points P' (x', y', z') can be obtained from the original coordinate points P(x,y,z) of Obj through the application of a geometric transformation.

Translation

An object is displaced a given distance and direction from its original position. The direction and displacement of the translation is prescribed by a vector

$$V = aI + bJ + cK$$

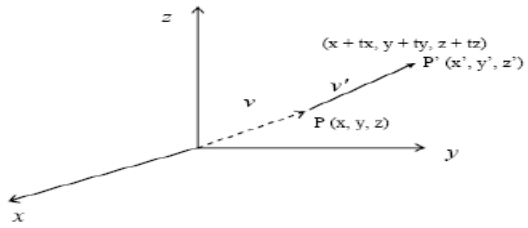
The new coordinates of a translated point can be calculated by using the transformation

$$x'=x+a$$

$$y'=y+b$$

$$z'=z+c$$

In order to represent this transformation as a matrix transformation, we need to use homogeneous coordinate. The required homogeneous matrix transformation can then be expressed as



$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Figure 3.20 Three-Dimensional Translation

Rotation

Rotation in three dimensions is considerably more complex then rotation in two dimensions. In 2-D, a rotation is prescribed by an angle of rotation θ and a center of rotation, say P.

In two dimensions, a rotation is prescribed by an angle of rotations require the prescription of an angle of rotation and an axis of rotation. The canonical rotations are defined when one of the positive x, y or z coordinate axes is chosen as the axis of rotation. Then the construction of the rotation transformation proceeds just like that of a rotation in two dimensions about the origin. The Corresponding matrix transformations are

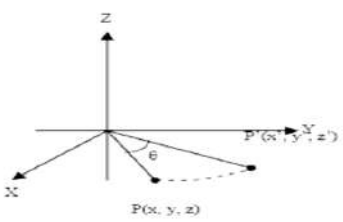


Figure 3.21 Three-Dimensional Rotation

Rotation about the z axis $\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Rotation about the y axis $\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Rotation about the x axis $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Note that the direction of a positive angle of rotation is chosen in accordance to the right-hand rule with respect to the axis of rotation.

Point Clipping

Point clipping is essentially the evaluation of the following inequalities:

$$x_{min} \le x \le x_{max} \text{ and } y_{min} \le y \le y_{max}$$

Where, xmin, xmax, ymin and ymax define the clipping window. A point (x,y) is considered inside the window when the inequalities all evaluate to true.

Line Clipping

Lines that do not intersect the clipping window are either completely inside the window or completely outside the window. On the other hand, a line that intersects the clipping window is divided by the intersection points into segments that are either inside or outside the window. The following algorithms provide efficient ways to decide the spatial relationship between an arbitrary line and the clipping window and to find intersection points.

Cohen-Sutherland Line Clipping

In this algorithm we divide the line clipping process into two phases: (1) identify those lines which intersect the clipping window and so need to be clipped and (2) perform the clipping.

All lines fall into one of the following clipping categories:

1. Visible – both endpoints of the line within the window
2. Not visible – the line definitely lies outside the window. This will occur if the line from (x1,y1,) to (x2,y2) satisfies any one of the following four inequalities:

x1,x2>xmax y1,y2>ymax

x1,x2<xmin y1,y2<ymin

3. Clipping candidate – the line is in neither category 1 and 2

In fig., line l1 is in category 1 (visible); lines l2 and l3 are in category 2 (not visible); and lines l4 and l5 are in category 3 (clipping candidate).

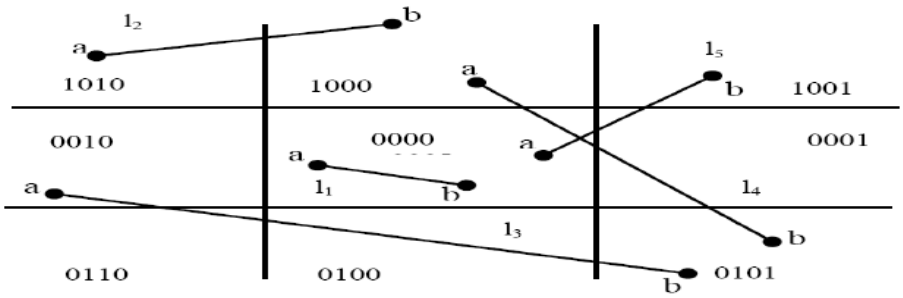


Figure 3.22 Representations of Line codes in Cohen Sutherland Algorithm

The algorithm employs an efficient procedure for finding the category of a line. It proceeds in two steps:

1. Assign a 4-bit region code to each endpoint of the line. The code is determined according to which of the following nine regions of the plane the endpoint lies in

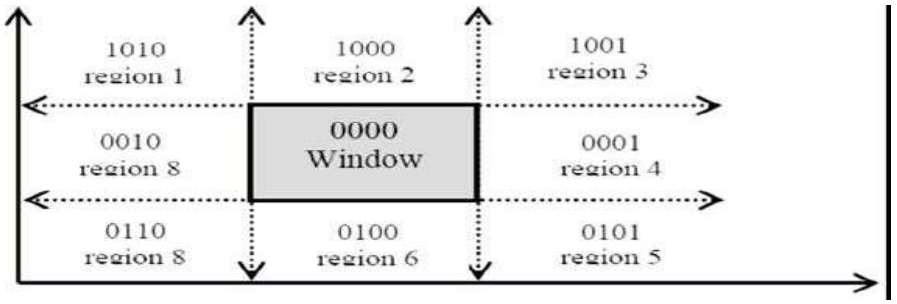


Figure 3.23 Region Code Assignments

Starting from the leftmost bit, each bit of the code is set to true (1) or false (0) according to the scheme

Bit 1: endpoint is above the window = sign (y – ymax)

Bit 2: endpoint is below the window = sign (ymin-y)

Bit 3: endpoint is to the right of the window = sign (x–xmax)

Bit 4: endpoint is to the left of the window = sign (xmin –x)

We use the convention that sign (a) =1 if a is positive, 0 otherwise. Of course, a point with code 0000 is inside the window.

2. The line is visible if both region codes are 0000, and not visible if the bitwise logical AND of the codes is not 0000, and a candidate for clipping if the bitwise logical AND of the region codes is 0000.

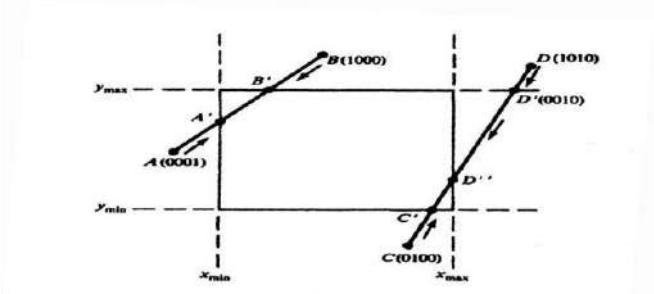


Figure 3.24 Example of Cohen Sutherland Algorithm

For a line in category 3 we proceed to find the intersection point of the line with one of the boundaries of the clipping window, or to be exact, with the infinite extension of one of the boundaries. We choose an endpoint of the line, say (x1, y1), that is outside the window, i.e., whose region code is not 0000. We then select an extended boundary line by observing that those boundary lines that are candidates for intersection are of ones for which the chosen endpoint must be "pushed across" so as to change a "1" in its code to a "0" (see Fig. above). This means:

If bit 1 is 1, intersect with line y = ymax.

If bit 2 is 1, intersect with line y = ymin.

If bit 3 is 1, intersect with line x = xmax.

If bit 4 is 1, intersect with line y = ymin.

Consider line CD is Fig. If endpoint C is chosen, then the bottom boundary line y=ymin is selected for computing intersection. On the other hand, if endpoint D is chosen, then either the top boundary line y=ymax or the right boundary line x = xmax is used. The coordinates of the intersection point are

xi=xmin or xmax // if the boundary line is vertical

yi=y1+m(xi-x1)

or

xi=x1+(yi-y1)/m // if the boundary line is horizontal

yi=ymin or ymax

Where m = (y2 – y1)/(x2 – x1) is the slope of the line.

Now we replace endpoint (x1, y1) with the intersection point (x1, yi) effectively eliminating the portion of the original line that is on the outside of the selected window boundary. The new endpoint is then assigned an updated region code and the clipped line re-categorized and handled in the same way. This iterative process terminates when we finally reach a clipped line that belongs to either category 1 (visible) or category 2 (not visible).

Liang-Barsky Line Clipping

The ideas for clipping line of Liang-Barsky and Cyrus-Beck are the same. The only difference is Liang-Barsky algorithm has been optimized for an upright rectangular clip window. Liang and Barsky have created an algorithm that uses floating-point arithmetic but finds the appropriate end points with at most four computations. This algorithm uses the parametric equations for a line and solves four inequalities to find the range of the parameter for which the line is in the viewport.

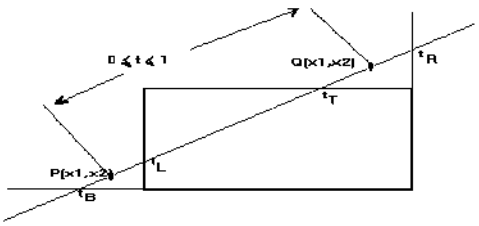


Figure 3.25 Example of Liang-Barsky Clipping

Let P(x1,y1) , Q(x2,y2)be the line which we want to study. The parametric equation of the line segment from gives x-values and y-values for every point in terms of a parameter that ranges from 0 to 1. The equations are

$$x=x1+(x2-x1)*t=x1+dx*t$$

$$\& \; y=y1+(y2-y1)*t=y1+dy*t$$

We can see that when t = 0, the point computed is P(x1,y1); and when t = 1, the point computed is Q(x2,y2).

Algorithm

1. Set $t_{min} = 0$ and $t_{max} = 1$
2. Calculate the values of tL, tR, tT, and tB (tvalues).
   - if $t < t_{min}$ or $t > t_{max}$ ignore it and go to the next edge
   - otherwise classify the tvalue as entering or exiting value (using inner product to classify)
   - if t is entering value set $t_{min} = t$; if t is exiting value set $t_{max} = t$
3. If $t_{min} < t_{max}$ then draw a line from (x1 + dx*tmin, y1 + dy*tmin) to (x1 + dx*tmax, y1 + dy*tmax)
4. If the line crosses over the window, you will see (x1 + dx*tmin, y1 + dy*tmin) and (x1 + dx*tmax, y1 + dy*tmax) are intersection between line and edge.

Midpoint Subdivision

An alternative way to process a line in category 3 is based on binary search. The line is divided at its midpoint into two shorter line segments. The clipping categories of the two new line segments are then determined by their region codes. Each segment in category 3 is divided again into shorter segments and categorized. This bisection and categorization process continue until each line segment that spans across a window boundary (hence encompasses an intersection point) reaches a threshold for line size and all other segments are either in category 1 (visible) or in category 2 (invisible). The midpoint coordinates (xm,ym) of a line joining (x1,y1) and (x2,y2) are given by

$$xm=x1+x2/2 \qquad ym=y1+y2/2$$

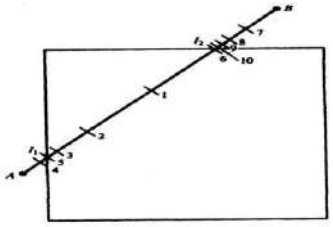


Figure 3.26 Mid-point Subdivision Algorithm

The example in above figure illustrates how midpoint subdivision is used to zoom in onto the two intersection points and with 10 bisections. The process continues until we reach two-line segments that are, say, pixel – sized. i.e. mapped to one single pixel each in the image space. If the maximum number of pixels in a line is M, this method will yield a pixel-sized line segment in N subdivisions, where 2iI2IN = M or N = log2 M. For instance, when M = 1024 we sneed at most N = log2 1024 = 10 subdivisions.

Polygon Clipping

An algorithm that clips a polygon must deal with many different cases. The case is particularly noteworthy in that the concave polygon is clipped into two separate polygons. All in all, the task of clipping seems rather complex. Each edge of the polygon must be tested against each edge of the clip rectangle; new edges must be added, and existing edges must be discarded, retained, or divided. Multiple polygons may result from clipping a single polygon. We need an organized way to deal with all these cases.

Sutherland - Hodgman Polygon Clipping

Sutherland and Hodgman's polygon-clipping algorithm uses a divide-and-conquer strategy: It solves a series of simple and identical problems that, when combined, solve the overall problem. The simple problem is to clip a polygon against a single infinite clip edge. Four clip edges, each defining one boundary of the clip rectangle, successively clip a polygon against a clip rectangle.

Note the difference between this strategy for a polygon and the Cohen-Sutherland algorithm for clipping a line: The polygon clipper clips against four edges in succession, whereas the line clipper tests the out code to see which edge is crossed, and clips only when necessary.

Figure 3.27 Clipping Using Sutherland-Hodgeman Polygon Clipping

This figure represents a polygon (the large, solid, upward pointing arrow) before clipping has occurred.

The following figures show how this algorithm works at each edge, clipping the polygon.

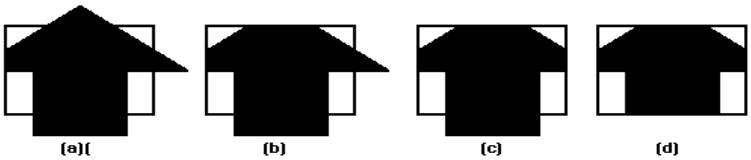


Figure 3.28 Clipping Using Sutherland-Hodgeman Polygon Clipping

a. Clipping against the left side of the clip window.

b. Clipping against the top side of the clip window.

c. Clipping against the right side of the clip window.

d. Clipping against the bottom side of the clip window.

Four Types of Edges

As the algorithm goes around the edges of the window, clipping the polygon, it encounters four types of edges. All four edge types are illustrated by the polygon in the following figure. For each edge type, zero, one, or two vertices are added to the output list of vertices that define the clipped polygon.

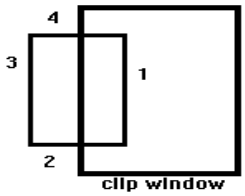


Figure 3.29 Concept of Sutherland-Hodgeman Polygon Clipping

The four types of edges are:

1. Edges that are totally inside the clip window. - add the second inside vertex point
2. Edges that are leaving the clip window. - add the intersection point as a vertex
3. Edges that are entirely outside the clip window. - add nothing to the vertex output list
4. Edges that are entering the clip window. - save the intersection and inside points as vertices

How to Calculate Intersections

Assume that we're clipping a polygon's edge with vertices at (x1,y1) and (x2,y2) against a clip window with vertices at (xmin, ymin) and (xmax,ymax).

The location (IX, IY) of the intersection of the edge with the left side of the window is:

i. IX = xmin

ii. IY = slope*(xmin-x1) + y1, where the slope = (y2-y1)/(x2-x1)

The location of the intersection of the edge with the right side of the window is:

i. IX = xmax

ii. IY = slope*(xmax-x1) + y1, where the slope = (y2-y1)/(x2-x1)

The intersection of the polygon's edge with the top side of the window is:

i. IX = x1 + (ymax - y1) / slope

ii. IY = ymax

Finally, the intersection of the edge with the bottom side of the window is:

i. IX = x1 + (ymin - y1) / slope

ii. IY = ymin

Weiler-Atherton Algorithm

- General clipping algorithm for concave polygons with holes
- Produces multiple polygons (with holes)
- Make linked list data structure
- Traverse to make new polygons

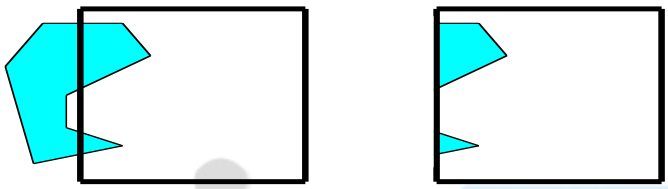


Figure 3.30 Clipping Using Weiler-Atherton Polygon Clipping

- Given polygons A and B as linked list of vertices (counter-clockwise order)
- Find all edge intersections & place in list
- Insert as “intersection” nodes
- Nodes point to A & B
- Determine in/out status of vertices

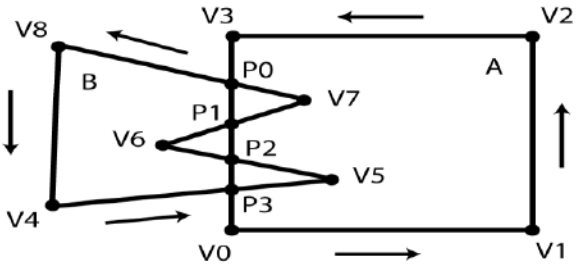


Figure 3.31 Example of Weiler-Atherton Polygon Clipping

- If “intersecting” edges are parallel, ignore
- Intersection point is a vertex
  - Vertex of A lies on a vertex or edge of B
  - Edge of A runs through a vertex of B
  - Replace vertex with an intersection node

Weiler-Atherton Algorithm: Union

- Find a vertex of A outside of B
- Traverse linked list
- At each intersection point switch to other polygon
- Do until return to starting vertex
- All visited vertices and nodes define union’ed polygon

Weiler-Atherton Algorithm: Intersection

- Start at intersection point
  - If connected to an "inside" vertex, go there
  - Else step to an intersection point
  - If neither, stop
- Traverse linked list
- At each intersection point switch to other polygon and remove intersection point from list
- Do until return to starting intersection point
- If intersection list not empty, pick another one
- All visited vertices and nodes define and'ed polygon

If polygons don't intersect

- Union
  - If one inside the other, return polygon that surrounds the other
  - Else, return both polygons
- Intersection
  - If one inside the other, return polygon inside the other

Else, return no polygons

Point P Inside a Polygon?

- Connect P with another point P` that you know is outside polygon
- Intersect segment PP` with polygon edges
- Watch out for vertices!
- If # intersections are even (or 0): Outside
- If odd: Inside

Introduction to Hidden surface elimination

1. One of the most challenging problems in computer graphics is the removal of hidden parts from images of solid objects.
2. In real life, the opaque material of these objects obstructs the light rays from hidden parts and prevents us from seeing them.
3. In the computer generation, no such automatic elimination takes place when objects are projected onto the screen coordinate system.
4. Instead, all parts of every object, including many parts that should be invisible are displayed.
5. To remove these parts to create a more realistic image, we must apply a hidden line or hidden surface algorithm to set of objects.
6. The algorithm operates on different kinds of scene models, generate various forms of output or cater to images of different complexities.
7. All use some form of geometric sorting to distinguish visible parts of objects from those that are hidden.
8. Just as alphabetical sorting is used to differentiate words near the beginning of the alphabet from those near the ends.
9. Geometric sorting locates objects that lie near the observer and are therefore visible.
10. Hidden line and Hidden surface algorithms capitalize on various forms of coherence to reduce the computing required to generate an image.
11. Different types of coherence are related to different forms of order or regularity in the image.

Types of hidden surface detection algorithms

1. Object space methods

2. Image space methods

Object space methods: In this method, various parts of objects are compared. After comparison visible, invisible or hardly visible surface is determined. These methods generally decide visible surface. In the wireframe model, these are used to determine a visible line. So these algorithms are line based instead of surface based. Method proceeds by determination of parts of an object whose view is obstructed by other object and draws these parts in the same color.

Image space methods: Here positions of various pixels are determined. It is used to locate the visible surface instead of a visible line. Each point is detected for its visibility. If a point is visible, then the pixel is on, otherwise off. So the object close to the viewer that is pierced by a projector through a pixel is determined. That pixel is drawn is appropriate color.

Algorithms used for hidden line surface detection

1. Back Face Removal Algorithm
2. Z-Buffer Algorithm
3. Painter Algorithm
4. Scan Line Algorithm
5. Subdivision Algorithm
6. Floating horizon Algorithm

Back-Face Detection

In a solid object, there are surfaces which are facing the viewer (front faces) and there are surfaces which are opposite to the viewer (back faces). These back faces contribute to approximately half of the total number of surfaces. Since we cannot see these surfaces anyway, to save processing time, we can remove them before the clipping process with a simple test. Each surface has a normal vector. If this vector is pointing in the direction of the centre of projection, it is a front face and can be seen by the viewer. If it is pointing away from the centre of projection, it is a back face and cannot be seen by the viewer. The test is very simple, if the z component of the normal vector is positive, then, it is a back face. If the z component of the vector is negative, it is a front face. Note that this technique only caters well for non-overlapping convex polyhedral. For other cases where there are concave polyhedral or overlapping objects, we still need to apply other methods to further determine where the obscured faces are partially or completely hidden by other objects (eg. Using Depth-Buffer Method or Depth-sort Method).

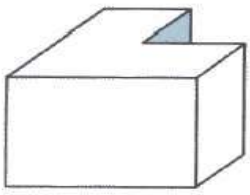


Figure 3.32 Representation of Object

The Z-buffer Algorithm

This method is developed by Cutmull. It is an image-space approach. The basic idea is to test the Z-depth of each surface to determine the closest (visible) surface.

In this method, each surface is processed separately one pixel position at a time across the surface. The depth values for a pixel are compared and the closest (smallest z) surface determines the color to be displayed in the frame buffer.

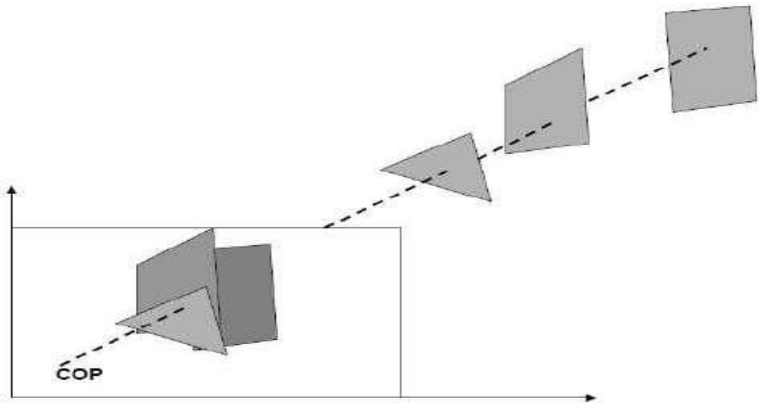


Figure 3.33 Z-Buffer Algorithm

It is applied very efficiently on surfaces of polygon. Surfaces can be processed in any order. To override the closer polygons from the far ones, two buffers named frame buffer and depth buffer, are used.

Depth buffer is used to store depth values for (x, y) position, as surfaces are processed (0 ≤ depth ≤ 1).

The frame buffer is used to store the intensity value of color value at each position (x, y).

The z-coordinates are usually normalized to the range [0, 1]. The 0 value for z-coordinate indicates back clipping pane and 1 value for z-coordinates indicates front clipping pane.

Algorithm

Step-1 − Set the buffer values −

Depthbuffer (x, y) = 0

Framebuffer (x, y) = background color

Step-2 − Process each polygon (One at a time)

For each projected (x, y) pixel position of a polygon, calculate depth z.

If Z > depthbuffer (x, y)

Compute surface color,

set depthbuffer (x, y) = z,

framebuffer (x, y) = surfacecolor (x, y)

Advantages

- It is easy to implement.
- It reduces the speed problem if implemented in hardware.
- It processes one object at a time.

Disadvantages

- It requires large memory.
- It is time consuming process.

Back-Face Detection

A fast and simple object-space method for identifying the back faces of a polyhedron is based on the "inside-outside" tests. A point (x, y, z) is "inside" a polygon surface with plane parameters A, B, C, and D if 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, then this polygon is a back-face if

V.N > 0

Furthermore, if object descriptions are converted to projection coordinates and your viewing direction is parallel to the viewing z-axis, then −

$$V = (0, 0, V_z) \text{ and } V.N = V_z C$$

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 ZVZV axis, the polygon is a back face if C < 0. Also, we cannot see any face whose normal has z component C = 0, since your viewing direction is towards that polygon. Thus, in general, we can label any polygon as a back face if its normal vector has a z component value −

C <= 0

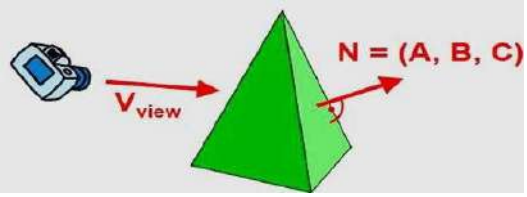


Figure 3.34 Back Face Detection

Similar methods can be used in packages that employ a left-handed viewing system. In these packages, plane parameters A, B, C and D can be calculated from polygon vertex coordinates specified in a clockwise direction (unlike the counter clockwise direction used in a right-handed system).

Also, back faces have normal vectors that point away from the viewing position and are identified by C >= 0 when the viewing direction is along the positive ZvZv axis. By examining parameter C for the different planes defining an object, we can immediately identify all the back faces.

Depth Sorting Method

Depth sorting method uses both image space and object-space operations. The depth-sorting method performs two basic functions −

- First, the surfaces are sorted in order of decreasing depth.
- Second, the surfaces are scan-converted in order, starting with the surface of greatest depth.

The scan conversion of the polygon surfaces is performed in image space. This method for solving the hidden-surface problem is often referred to as the **painter's algorithm**. The following figure shows the effect of depth sorting −

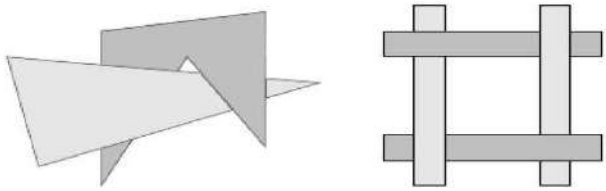


Figure 3.35 Depth Sorting Algorithm

The algorithm begins by sorting by depth. For example, the initial “depth” estimate of a polygon may be taken to be the closest z value of any vertex of the polygon.

Let us take the polygon P at the end of the list. Consider all polygons Q whose z-extents overlap P’s. Before drawing P, we make the following tests. If any of the following tests is positive, then we can assume P can be drawn before Q.

- Do the x-extents not overlap?
- Do the y-extents not overlap?
- Is P entirely on the opposite side of Q’s plane from the viewpoint?
- Is Q entirely on the same side of P’s plane as the viewpoint?
- Do the projections of the polygons not overlap?

If all the tests fail, then we split either P or Q using the plane of the other. The new cut polygons are inserting into the depth order and the process continues. Theoretically, this partitioning could generate $O(n^2)$ individual polygons, but in practice, the number of polygons is much smaller.

Binary Space Partition (BSP) Trees

Binary space partitioning is used to calculate visibility. To build the BSP trees, one should start with polygons and label all the edges. Dealing with only one edge at a time, extend each edge so that it splits the plane in two. Place the first edge in the tree as root. Add subsequent edges based on whether they are inside or outside. Edges that span the extension of an edge that is already in the tree are split into two and both are added to the tree.

- From the above figure, first take **A** as a root.
- Make a list of all nodes in figure (a).
- Put all the nodes that are in front of root **A** to the left side of node **A** and put all those nodes that are behind the root **A** to the right side as shown in figure (b).
- Process all the front nodes first and then the nodes at the back.
- As shown in figure (c), we will first process the node **B**. As there is nothing in front of the node **B**, we have put NIL. However, we have node **C** at back of node **B**, so node **C** will go to the right side of node **B**.
- Repeat the same process for the node **D**.

The Z-buffer algorithm is one of the most commonly used routines. It is simple, easy to implement, and is often found in hardware.

The idea behind it is uncomplicated: Assign a z-value to each polygon and then display the one (pixel by pixel) that has the smallest value.

There are some advantages and disadvantages to this:

Advantages:

- Simple to use
- Can be implemented easily in object or image space
- Can be executed quickly, even with many polygons
- Disadvantages:
- Takes up a lot of memory
- Can't do transparent surfaces without additional code

Consider these two polygons (right: edge-on left: head-on).

The computer would start (arbitrarily) with Polygon 1 and put its depth value into the buffer. It would do the same for the next polygon, P2. It will then check each overlapping pixel and check to see which one is closer to the viewer, and display the appropriate color.

This is a simplistic example, but the basic ideas are valid for polygons in any orientation and permutation (this algorithm will properly display polygons piercing one another, and polygons with conflicting depths, such as:

Visible Surface Determination: Painter's Algorithm

The painter's algorithm is based on depth sorting and is a combined object and image space algorithm. It is as follows:

1. Sort all polygons according to z value (object space); Simplest to use maximum z value
2. Draw polygons from back (maximum z) to front (minimum z)

This can be used for wireframe drawings as well by the following:

1. Draw solid polygons using Poly scan (in the background color) followed by Polyline (polygon color).
2. Poly scan erases Polygons behind it then Polyline draws new Polygon.

Problems with simple Painter's algorithm

Look at cases where it doesn't work correctly. S has a greater depth than S' and so will be drawn first. But S' should be drawn first since it is obscured by S. We must somehow reorder S and S'.:

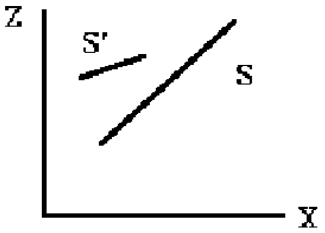


Figure 3.36 Representation of Lines

We will perform a series of tests to determine, if two polygons need to be reordered. If the polygons fail a test, then the next test must be performed. If the polygons fail all tests, then they are reordered. The initial tests are computationally cheap, but the later tests are more expensive.

So look at revised algorithm to test for possible reordering could store Zmax, Zmin for each Polygon.

- sort on Zmax
- start with polygon with greatest depth (S)
- compare S with all other polygons (P) to see if there is any depth overlap (Test 0)

If S.Zmin <= P.Zmax then have depth overlap (as in above and below figures)

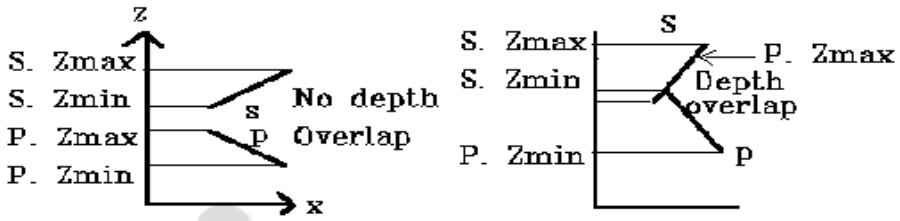


Figure 3.37 Simple Painter's Algorithm

If have depth overlap (failed Test 0) we may need to reorder polygons.

Next (Test 1) check to see if polygons overlap in xy plane (use bounding rectangles)

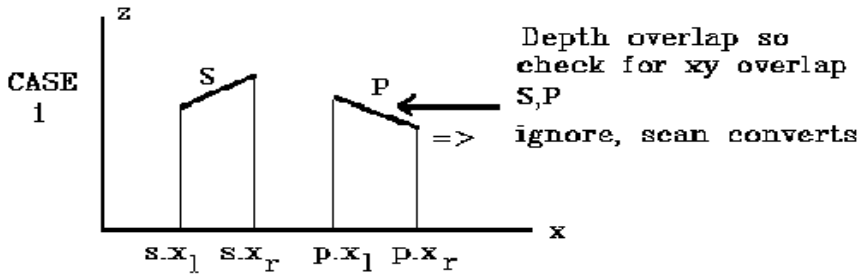


Figure 3.38 Case I

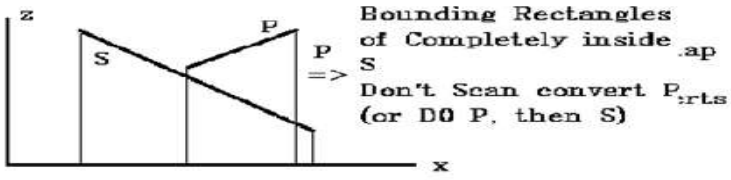


Figure 3.39 Case II

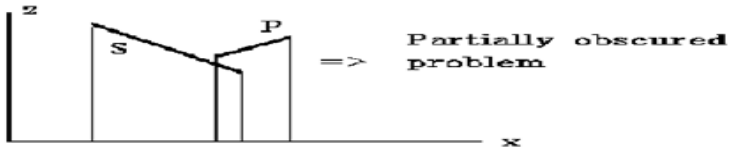


Figure 3.40 Case III

Do above tests for x and y.

If have case 1 or 2 then we are done (passed Test 1) but for case 3 we need further testing failed Test 1)
Next test (Test 2) to see if polygon S is "outside" of polygon P (relative to view plane)
Remember: a point (x, y, z) is "outside" of a plane if we put that point into the plane equation and get:
Ax + By + Cz + D > 0
So to test for S outside of P, put all vertices of S into the plane equation for P and check that all vertices give a result that is > 0.
i.e. Ax' + By' + Cz' + D > 0 x', y', z' are S vertices
A, B, C, D are from plane equation of P (choose normal away from view plane since define "outside" with respect to the view plane)

Basic Illumination Model:

A simple illumination model (called a lighting model in OpenGL) can be based on three components: ambient reflection, diffuse reflection and specular reflection. The model we look at is known as the Phong model. The OpenGL lighting model has the components of the Phong model, but is more elaborate.

Ambient Reflection

Imagine a uniform intensity background light which illuminates all objects in the scene equally from all directions. This is known as ambient light. The ambient light model attempts to capture the effect of light which results from many object-object reflections without detailed calculations, that is, achieve computational efficiency. Those detailed calculations are performed in some rendering techniques such as radiosity which attempt to achieve higher quality images, but not real-time.

Diffuse Reflection

Objects illuminated by only ambient light have equal intensity everywhere. If there are light sources in a scene then different objects should have different intensities based on distance and orientation with respect to the light source and the viewer.

A point on a diffuse surface appears equally bright from all viewing positions because it reflects light equally in all directions. That is, its intensity is independent of the position of the viewer.

Whilst independent of the viewer, the intensity of a point on a diffuse surface does depend on the orientation of the surface with respect to the light source and the distance to the light source.

A simple model for diffuse reflection is Lambertian reflection.

Assuming the following geometry

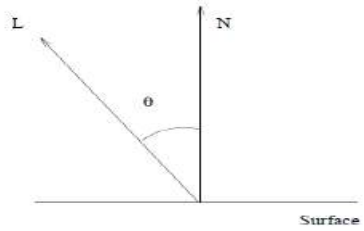


Figure 3.41 The direction of light is measured from the surface normal

Then the intensity of the diffuse reflection from a point light source is

$$I = L_d k_d \cos\theta$$

Where Ld is the intensity of the (point) light source, kd is the diffuse reflection coefficient of the object's material, and θ is the angle between the normal to the surface and the light source direction vector.

If the normal vector to the surface N and the light source direction vector L are both normalized then the above equation can be simplified to

$$I = L_d k_d (N.L)$$

If a light source is an infinite distance from the object then L will be the same for all points on the object — the light source becomes a directional light source. In this case less computation can be performed.

Adding the ambient reflection and diffuse reflection contributions together we get

$$I = L_a k_a + L_d k_d (N.L)$$

Specular Reflection

Specular reflection occurs on hard, shiny surfaces and is seen as highlights. Specular highlights are strongly directional. The approach to specular reflection in the Phong model is that the specular reflection intensity drops off as the cosine of the angle between the normal and the specular reflection direction raised to some power n which indicates the shininess of the surface. The higher the power of n the smaller and brighter the highlight.

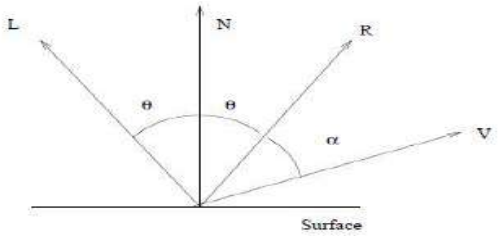


Figure 3.42 Specular Reflection

The specular component of the illumination model may thus be given as

$$I = f_{att} L_{s\lambda} k_{s\lambda} cos^n \alpha$$

If the direction of (specular) reflection R and the viewpoint direction V are normalized then the equation becomes

$$I = f_{att} L_{s\lambda} k_{s\lambda} (\mathbf{R}.\mathbf{V})^n$$

The full lighting equation becomes

$$I_\lambda = L_{a\lambda} k_{a\lambda} + f_{att}[L_{d\lambda} k_{d\lambda} (\mathbf{N}.\mathbf{L}) + L_{s\lambda} k_{s\lambda} (\mathbf{R}.\mathbf{V})^n]$$

Phong Shading Algorithm

Using highlights avoids surfaces that look dull, lifeless, boring, blah. One cool thing about highlights is that, in addition to being all bright and shiny, they change as the object moves. In this way, highlights provide useful visual information about shape and motion.

The simplest model for approximating surface highlights is the Phong model, originally developed by Bui-Tong Phong. In this model, we think of the interaction between light and a surface as having three distinct components:

Ambient, Diffuse, Specular

The ambient component is usually given a dim constant value, such as 0.2. It approximates light coming from a surface due to all the non-directional ambient light that is in the environment. In general, you'll want this to be some tinted color, rather than just gray. $[r_a,g_a,b_a]$. For example, a slightly greenish object might have an ambient color of [0.1,0.3,0.1].

The diffuse component is that dot product **n•L** that we discussed in class. It approximates light, originally from light source **L**, reflecting from a surface which is diffuse, or non-glossy. One example of a non-glossy surface is paper. In general, you'll also want this to have a non-gray color value, so this term would in general be a color defined as: $[r_d,g_d,b_d]$(**n•L**).

Finally, the Phong model has a provision for a highlight, or specular, component, which reflects light in a shiny way. This is defined by $[r_s,g_s,b_s]$(**R•L**)^p, where **R** is the mirror reflection direction vector we discussed in class (and also used for ray tracing), and where p is a specular power. The higher the value of p, the shinier the surface.

The complete Phong shading model for a single light source is:

$[r_a,g_a,b_a] + [r_d,g_d,b_d]max_0$(**n•L**) + $[r_s,g_s,b_s]max_0$(**R•L**)^p

If you have multiple light sources, the effect of each light source $\mathbf{L_i}$ will geometrically depend on the normal, and therefore on the diffuse and specular components, but not on the ambient component. Also, each light might have its own [r,g,b] color. So the complete Phong model for multiple light sources is:

$[r_a,g_a,b_a] + \Sigma_i([L_r,L_g,L_b] ([r_d,g_d,b_d]max_0(\mathbf{n•L_i}) + [r_s,g_s,b_s]max_0(\mathbf{R•L_i})^p))$

Below you can see three different shaders for a sphere. In all cases, the sphere is lit by two light sources: a dim source from the rear left (-1,0,-.5) and a bright source from the front upper right (1,1,.5). The spheres have, respectively, no highlight, a highlight with an exponent of p = 4, and a highlight with an exponent of p = 16.

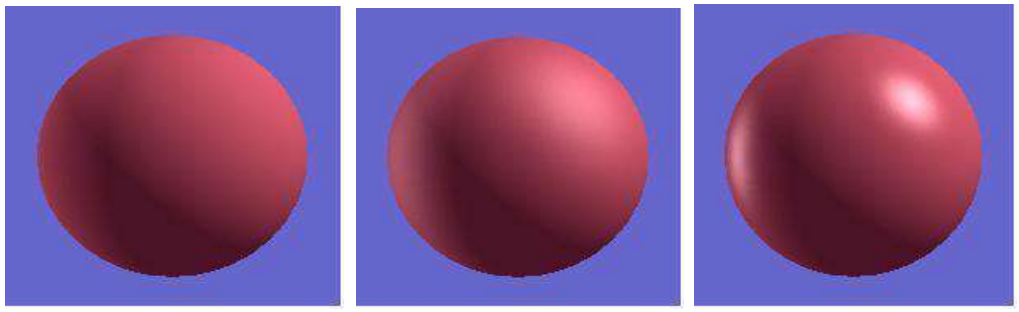


Figure 3.43 Graphical Object Representation Using Phong Shadding

Gouraud Shading Model

The second shading model, Gouraud shading, computes intensity for each vertex and then interpolates the computed intensities across the polygons. Gouraud shading performs a bi-linear interpolation of the intensities down and then across scan lines. It thus eliminates the sharp changes at polygon boundaries

The algorithm is as follows:

1. Compute a normal N for each vertex of the polygon.
2. From N compute intensity I for each vertex of the polygon.
3. From bi-linear interpolation compute intensity Ii for each pixel.
4. Paint pixel to shade corresponding to Ii.

How do we compute N for a vertex? Let N = the average of the normal of the polygons which include the vertex. Note that 4 sided polygons have 4 neighbours and triangles have 6 neighbours.

We can find the neighbours for a particular vertex by searching the polygons and including any polygons which include the vertex. Now look at performing the bi-linear intensity interpolation. This is the same as the bi-linear interpolation for the z depth of a pixel (used with the z buffer visible surface algorithm).

Advantages of Gouraud shading:

Gouraud shading gives a much better image than faceted shading (the facets no longer visible). It is not too computationally expensive: one floating point addition (for each colour) for each pixel. (the mapping to actual display registers requires a floating-point multiplication)

Disadvantages to Gouraud shading:

It eliminates creases that you may want to preserve, e.g. in a cube. We can modify data structure to prevent this by storing each physical vertex 3 times, i.e. a different logical vertex for each polygon. here is a data structure for a cube that will keep the edges:

Ray Tracing

Ray tracing is a technique for rendering three-dimensional graphics with very complex light interactions. This means you can create pictures full of mirrors, transparent surfaces, and shadows, with stunning results. We discuss ray tracing in this introductory graphics article because it is a very simple method to both understand and implement. It is based on the idea that you can model reflection and refraction by recursively following the path that light takes as it bounces through an environment.

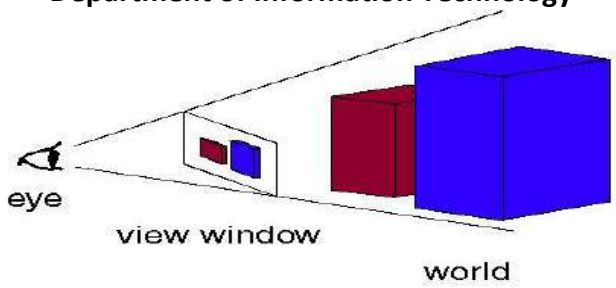


Figure 3.44 The eye, view window, and world

Ray tracing is so named because it tries to simulate the path that light rays take as they bounce around within the world - they are traced through the scene. The objective is to determine the color of each light ray that strikes the view window before reaching the eye. A light ray can best be thought of as a single photon (although this is not strictly accurate because light also has wave properties - but I promised there would be no theory).

The name ``ray tracing'' is a bit misleading because the natural assumption would be that rays are traced starting at their point of origin, the light source, and towards their destination, the eye. This would be an accurate way to do it, but unfortunately it tends to be very difficult due to the sheer numbers involved. Consider tracing one ray in this manner through a scene with one light and one object, such as a table. We begin at the light bulb, but first need to decide how many rays to shoot out from the bulb. Then for each ray we have to decide in what direction it is going. There is really infinity of directions in which it can travel - how do we know which to choose? Let's say we've answered these questions and are now tracing a number of photons. Some will reach the eye directly, others will bounce around some and then reach the eye and many, and many more will probably never hit the eye at all. For all the rays that never reach the eye, the effort tracing them was wasted.

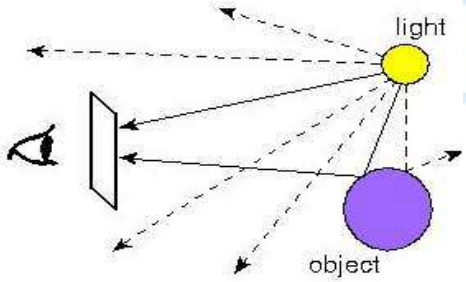


Figure 3.45 Tracing rays from the light source to the eye

In order to save ourselves this wasted effort, we trace only those rays that are guaranteed to hit the view window and reach the eye. It seems at first that it is impossible to know beforehand which rays reach the eye. After all, any given ray can bounce around the room many times before reaching the eye. However, if we look at the problem backwards, we see that it has a very simple solution. Instead of tracing the rays starting at the light source, we trace them backwards, starting at the eye.

Consider any point on the view window whose colour we're trying to determine. Its color is given by the color of the light ray that passes through that point on the view window and reaches the eye. We can just as well follow the ray backwards by starting at the eye and passing through the point on its way out into the scene. The two rays will be identical, except for their direction: if the original ray came directly from the light source, then the backwards ray will go directly to the light source; if the original bounced off a table first, the backwards ray will also bounce off the table. You can see this by looking at Figure again and just reversing the directions of the arrows. So the backwards method does the same thing as the original method, except it doesn't waste any effort on rays that never reach the eye.

This, then, is how ray tracing works in computer graphics. For each pixel on the view window, we define a ray that extends from the eye to that point. We follow this ray out into the scene and as it bounces off of different objects. The final color of the ray (and therefore of the corresponding pixel) is given by the color of the objects hit by the ray as it travels through the scene.

Just as in the light-source-to-eye method it might take a very large number of bounces before the ray ever hits the eye, in backwards method it might take many bounces before the ray every hits the light. Since we need to establish some limit on the number of bounces to follow the ray on, we make the following approximation: every time a ray hits an object, we follow a single new ray from the point of intersection directly towards the light source.

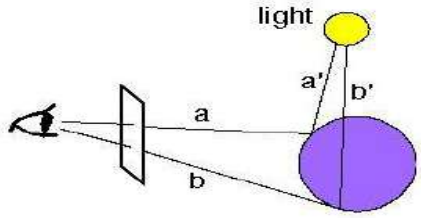


Figure 3.46 We trace a new ray from each ray-object intersection directly towards the light source

In the figure we see two rays, **a** and **b,** which intersect the purple sphere. To determine the color of **a,** we follow the new ray **a'** directly towards the light source. The color of **a** will then depend on several factors, discussed in Color and Shading below. As you can see, **b** will be shadowed because the ray **b'** towards the light source is blocked by the sphere itself. Ray **a** would have also been shadowed if another object blocked the ray **a'**.

Color Model

A color model is an abstract mathematical model describing the way colors can be represented as tuples of numbers, typically as three or four values or color components. When this model is associated with a precise description of how the components are to be interpreted (viewing conditions, etc.), the resulting set of colors is called color space.

CIE (Commission International de l'Eclairage - International Color Commission) organisation produced two models for defining color:

• 1931: Measured on 10 subjects (!) on samples subtending 2 (!) degrees of the field of view

• 1964: Measured on larger number of subjects subtending 10 degrees of field of view

• The CIE 1931 model is the most commonly used

• It defines three primary “colors” X, Y and Z that can be used to describe all visible colors, as well as a standard white, called C.

• The range of colors that can be described by combinations of other colors is called a color gamut.

- Since it is impossible to find three colors with a gamut containing all visible colors, the CIE’s three primary colors are imaginary. They cannot be seen, but they can be used to define other visible colors.

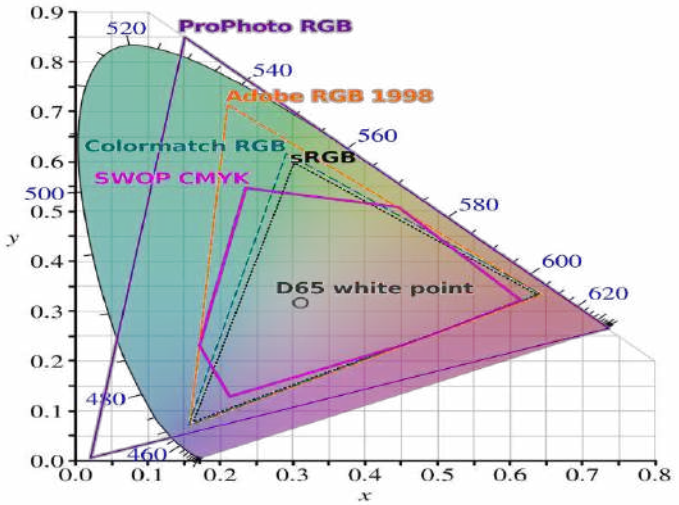


Figure 3.47 CIE Chromaticity Diagram

RGB Color Model

Media that transmit light (such as television) use additive color mixing with primary colors of red, green, and blue, each of which stimulates one of the three types of the eye's color receptors with as little stimulation as possible of the other two. This is called "RGB" color space. Mixtures of light of these primary colors cover a large part of the human color space and thus produce a large part of human color experiences. This is why color television sets or color computer monitors need only produce mixtures of red, green and blue light. See Additive color.

Other primary colors could in principle be used, but with red, green and blue the largest portion of the human color space can be captured. Unfortunately there is no exact consensus as to what loci in the chromaticity diagram the red, green, and blue colors should have, so the same RGB values can give rise to slightly different colors on different screens.

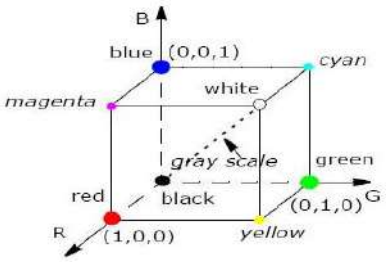


Figure 3.48RGB Color Model

YIQ Color Model

YIQ is the color space used by the NTSC color TV system, employed mainly in North and Central America, and Japan. I stand for in-phase, while Q stands for quadrature, referring to the components used in quadrature amplitude modulation. Some forms of NTSC now use the YUV color space, which is also used by other systems such as PAL. The Y component represents the luma information, and is the only component used by black-and-white television receivers. I and Q represent the chrominance information. In YUV, the U and V components can be thought of as X and Y coordinates within the color space. I and Q can be thought of as a second pair of axes on the same graph, rotated 33°; therefore, IQ and UV represent different coordinate systems on the same plane.

This model was designed to separate chrominance from luminance. This was a requirement in the early days of color television when black-and-white sets still were expected to pick up and display what were originally color pictures. The Y-channel contains luminance information (sufficient for black-and-white television sets) while the I and Q channels (in-phase and in-quadrature) carried the color information. A color television set would take these three channels, Y, I, and Q, and map the information back to R, G, and B levels for display on a screen.

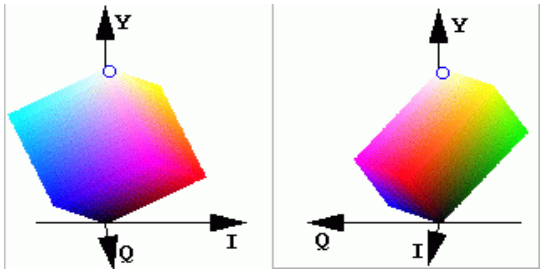


Figure 3.49YIQ Color Model

CMYK Color Model

It is possible to achieve a large range of colors seen by humans by combining cyan, magenta, and yellow transparent dyes/inks on a white substrate. These are the subtractive primary colors. Often a fourth ink, black, is added to improve reproduction of some dark colors. This is called "CMY" or "CMYK" color space.

The cyan ink absorbs red light but transmits green and blue, the magenta ink absorbs green light but transmits red and blue, and the yellow ink absorbs blue light but transmits red and green. The white substrate reflects the transmitted light back to the viewer. Because in practice the CMY inks suitable for printing also reflect a little bit of color, making a deep and neutral black impossible, the K (black ink) component, usually printed last, is needed to compensate for their deficiencies. Use of a separate black ink is also economically driven when a lot of black content is expected, e.g. in text

media, to reduce simultaneous use of the three colored inks. The dyes used in traditional color photographic prints and slides are much more perfectly transparent, so a K component is normally not needed or used in those media.

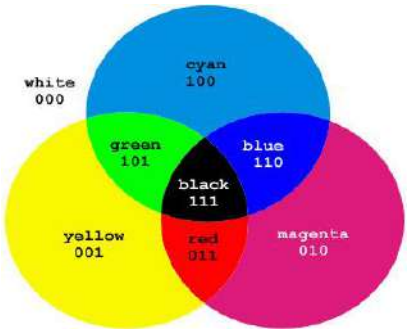


Figure 3.50 CMYK Color Model

HSV Color Model

Recognizing that the geometry of the RGB model is poorly aligned with the color-making attributes recognized by human vision, computer graphics researchers developed two alternate representations of RGB, HSV and HSL (hue, saturation, value and hue, saturation, lightness), in the late 1970s. HSV and HSL improve on the color cube representation of RGB by arranging colors of each hue in a radial slice, around a central axis of neutral colors which ranges from black at the bottom to white at the top. The fully saturated colors of each hue then lie in a circle, a color wheel.

HSV models itself on paint mixture, with its saturation and value dimensions resembling mixtures of a brightly colored paint with, respectively, white and black. HSL tries to resemble more perceptual color models such as NCS or Munsell. It places the fully saturated colors in a circle of lightness ½, so that lightness 1 always implies white, and lightness 0 always implies black.

HSV and HSL are both widely used in computer graphics, particularly as color pickers in image editing software. The mathematical transformation from RGB to HSV or HSL could be computed in real time, even on computers of the 1970s, and there is an easy-to-understand mapping between colors in either of these spaces and their manifestation on a physical RGB device

Recognizing that the geometry of the RGB model is poorly aligned with the color-making attributes recognized by human vision, computer graphics researchers developed two alternate representations of RGB, HSV and HSL (hue, saturation, value and hue, saturation, lightness), in the late 1970s. HSV and HSL improve on the color cube representation of RGB by arranging colors of each hue in a radial slice, around a central axis of neutral colors which ranges from black at the bottom to white at the top. The fully saturated colors of each hue then lie in a circle, a color wheel.

HSV models itself on paint mixture, with its saturation and value dimensions resembling mixtures of a brightly colored paint with, respectively, white and black. HSL tries to resemble more perceptual color models such as NCS or Munsell. It places the fully saturated colors in a circle of lightness ½, so that lightness 1 always implies white, and lightness 0 always implies black.

HSV and HSL are both widely used in computer graphics, particularly as color pickers in image editing software. The mathematical transformation from RGB to HSV or HSL could be computed in real time, even on computers of the 1970s, and there is an easy-to-understand mapping between colors in either of these spaces and their manifestation on a physical RGB device.

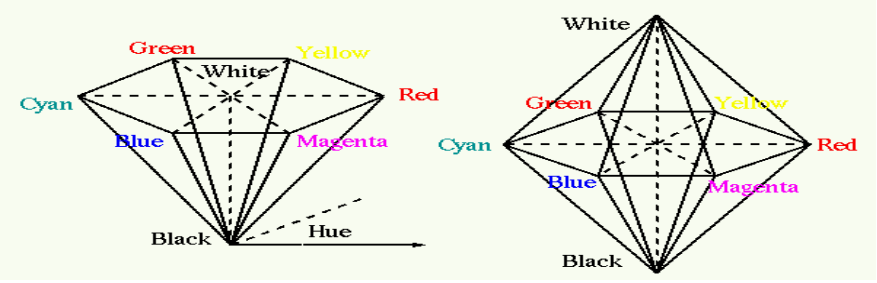


Figure 3.51HSV Color Model

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in