Acknowledgements to

Donald Hearn & Pauline Baker: Computer Graphics with OpenGL Version,3rd / 4th Edition, Pearson Education,2011

Edward Angel: Interactive Computer Graphics- A Top Down approach with OpenGL, 5th edition. Pearson Education, 2008

M M Raiker, Computer Graphics using OpenGL, Filip learning/Elsevier

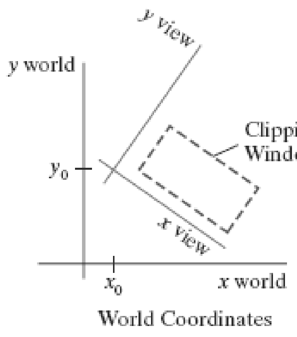
3.1 Clipping:

3.1.1 The Clipping Window

- We can design our own clipping window with any shape, size, and orientation we choose.
- But clipping a scene using a concave polygon or a clipping window with nonlinear boundaries requires more processing than clipping against a rectangle.
- Rectangular clipping windows in standard position are easily defined by giving the coordinates of two opposite corners of each rectangle

<u>Viewing-Coordinate Clipping Window</u>

✓ A general approach to the two-dimensional viewing transformation is to set up a *viewing-coordinate system* within the world-coordinate frame



✓ We choose an origin for a two-dimensional viewing-coordinate frame at some world position $\mathbf{P}_0 = (x_0, y_0)$, and we can establish the orientation using a world vector **V** that defines the *y*view direction.

✓ Vector **V** is called the two-dimensional **view up vector.**

- ✓ An alternative method for specifying the orientation of the viewing frame is to give a rotation angle relative to either the *x* or *y* axis in the world frame.
- ✓ The first step in the transformation sequence is to translate the viewing origin to the world origin.
- ✓ Next, we rotate the viewing system to align it with the world frame.
- ✓ Given the orientation vector **V**, we can calculate the components of unit vectors **v** = (v_x, v_y) and **u** = (u_x, u_y) for the *y*view and *x*view axes, respectively.

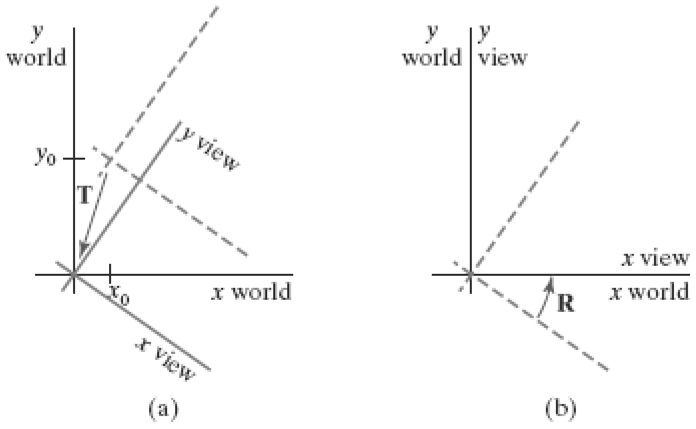
$$\mathbf{M}_{WC,VC} = \mathbf{R} \cdot \mathbf{T}$$

Where,

T is the translation matrix,

R is the rotation matrix

- ✓ A viewing-coordinate frame is moved into coincidence with the world frame is shown in below figure

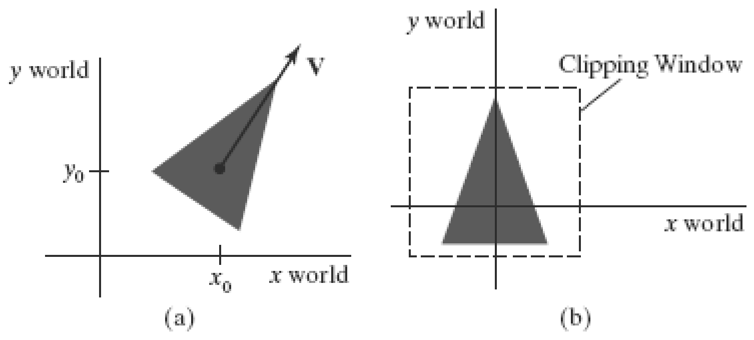


(a) applying a translation matrix **T** to move the viewing origin to the world origin, then (b) applying a rotation matrix **R** to align the axes of the two systems.

World-Coordinate Clipping Window

- ➢ A routine for defining a standard, rectangular clipping window in world coordinates is typically provided in a graphics-programming library.
- ➢ We simply specify two world-coordinate positions, which are then assigned to the two opposite corners of a standard rectangle.

- Once the clipping window has been established, the scene description is processed through the viewing routines to the output device.
- Thus, we simply rotate (and possibly translate) objects to a desired position and set up the clipping window all in world coordinates.



A triangle

(a), with a selected reference point and orientation vector, is translated and rotated to position

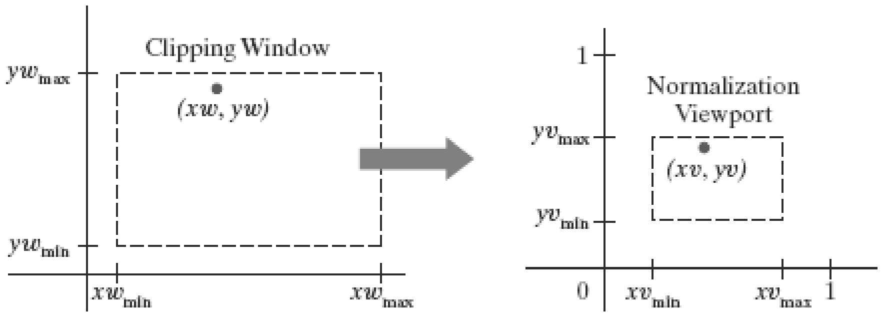
(b) within a clipping window.

3.1.2 Normalization and Viewport Transformations

- The viewport coordinates are often given in the range from 0 to 1 so that the viewport is positioned within a unit square.
- After clipping, the unit square containing the viewport is mapped to the output display device

Mapping the Clipping Window into a Normalized Viewport

- ✓ We first consider a viewport defined with normalized coordinate values between 0 and 1.
- ✓ Object descriptions are transferred to this normalized space using a transformation that maintains the same relative placement of a point in the viewport as it had in the clipping window Position (x_w, y_w) in the clipping window is mapped to position (x_v, y_v) in the associated viewport.



- ✓ To transform the world-coordinate point into the same relative position within the viewport, 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 (x_v, y_v), we have

$x_v = s_x x_w + t_x$

$y_v = s_y y_w + t_y$

Where the scaling factors are

$$s_x = \frac{xv_{\max} - xv_{\min}}{xw_{\max} - xw_{\min}}$$
$$s_y = \frac{yv_{\max} - yv_{\min}}{yw_{\max} - yw_{\min}}$$

and the translation factors are

$$t_x = \frac{xw_{\max} xv_{\min} - xw_{\min} xv_{\max}}{xw_{\max} - xw_{\min}}$$
$$t_y = \frac{yw_{\max} yv_{\min} - yw_{\min} yv_{\max}}{yw_{\max} - yw_{\min}}$$

- ✓ We could obtain the transformation from world coordinates to viewport coordinates with the following sequence:

  1. Scale the clipping window to the size of the viewport using a fixed-point position of (xw_{min}, yw_{min}).

  2. Translate (xw_{min}, yw_{min}) to (xv_{min}, yv_{min}).

- ✓ The scaling transformation in step (1) can be represented with the two dimensional Matrix

$$\mathbf{S} = \begin{bmatrix} s_x & 0 & xw_{\min}(1 - s_x) \\ 0 & s_y & yw_{\min}(1 - s_y) \\ 0 & 0 & 1 \end{bmatrix}$$

- ✓ The two-dimensional matrix representation for the translation of the lower-left corner of the clipping window to the lower-left viewport corner is

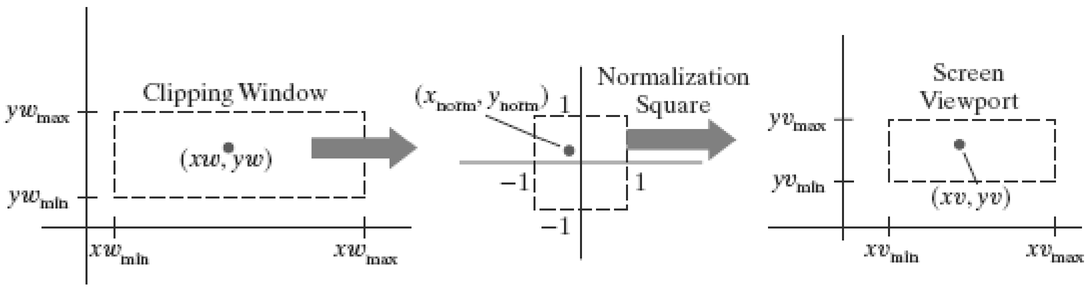
$$\mathbf{T} = \begin{bmatrix} 1 & 0 & xv_{\min} - xw_{\min} \\ 0 & 1 & yv_{\min} - yw_{\min} \\ 0 & 0 & 1 \end{bmatrix}$$

- ✓ And the composite matrix representation for the transformation to the normalized viewport is

$$\mathbf{M}_{\text{window, normviewp}} = \mathbf{T} \cdot \mathbf{S} = \begin{bmatrix} s_x & 0 & t_x \\ 0 & s_y & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

<u>Mapping the Clipping Window into a Normalized Square</u>

- ✓ Another approach to two-dimensional viewing is to transform the clipping window into a normalized square, clip in normalized coordinates, and then transfer the scene description to a viewport specified in screen coordinates.
- ✓ This transformation is illustrated in Figure below with normalized coordinates in the range from −1 to 1



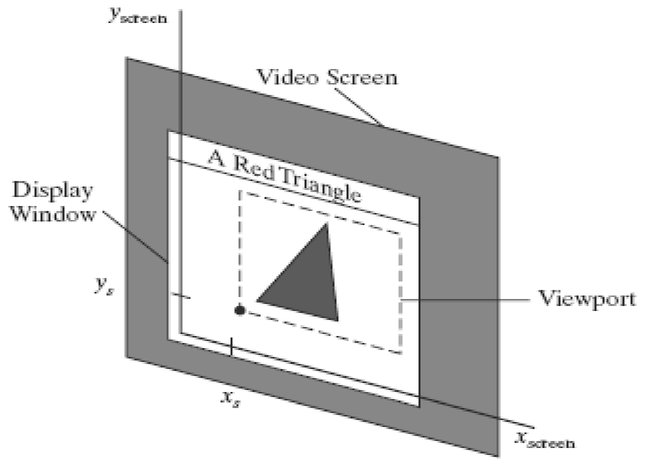
- ✓ The matrix for the normalization transformation is obtained by substituting −1 for $xv_{\min}$ and $yv_{\min}$ and substituting +1 for $xv_{\max}$ and $yv_{\max}$.

$$\mathbf{M}_{\text{window, normsquare}} = \begin{bmatrix} \frac{2}{xw_{\max} - xw_{\min}} & 0 & -\frac{xw_{\max} + xw_{\min}}{xw_{\max} - xw_{\min}} \\ 0 & \frac{2}{yw_{\max} - yw_{\min}} & -\frac{yw_{\max} + yw_{\min}}{yw_{\max} - yw_{\min}} \\ 0 & 0 & 1 \end{bmatrix}$$

- ✓ Similarly, after the clipping algorithms have been applied, the normalized square with edge length equal to 2 is transformed into a specified viewport.
- ✓ This time, we get the transformation matrix by substituting −1 for $xw_{\min}$ and $yw_{\min}$ and substituting +1 for $xw_{\max}$ and $yw_{\max}$

$$\mathbf{M}_{\text{normsquare, viewport}} = \begin{bmatrix} \frac{xv_{\max} - xv_{\min}}{2} & 0 & \frac{xv_{\max} + xv_{\min}}{2} \\ 0 & \frac{yv_{\max} - yv_{\min}}{2} & \frac{yv_{\max} + yv_{\min}}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

- ✓ Typically, the lower-left corner of the viewport is placed at a coordinate position specified relative to the lower-left corner of the display window. Figure below demonstrates the positioning of a viewport within a display window.



Display of Character Strings

- ✓ Character strings can be handled in one of two ways when they are mapped through the viewing pipeline to a viewport.
- ✓ The simplest mapping maintains a constant character size.

- ✓ This method could be employed with bitmap character patterns.
- ✓ But outline fonts could be transformed the same as other primitives; we just need to transform the defining positions for the line segments in the outline character shape

<u>Split-Screen Effects and Multiple Output Devices</u>

- ✓ By selecting different clipping windows and associated viewports for a scene, we can provide simultaneous display of two or more objects, multiple picture parts, or different views of a single scene.
- ✓ It is also possible that two or more output devices could be operating concurrently on a particular system, and we can set up a clipping-window/viewport pair for each output device.
- ✓ A mapping to a selected output device is sometimes referred to as a **workstation transformation**

3.1.3 Clipping Algorithms

- ➔ Any procedure that eliminates those portions of a picture that are either inside or outside a specified region of space is referred to as a **clipping algorithm** or simply **clipping.**
- ➔ The most common application of clipping is in the viewing pipeline, where clipping is applied to extract a designated portion of a scene (either two-dimensional or three-dimensional) for display on an output device.
- ➔ Different objects clipping are
  1. Point clipping
  2. Line clipping (straight-line segments)
  3. Fill-area clipping (polygons)
  4. Curve clipping
  5. Text clipping

<u>Two-Dimensional Point Clipping</u>

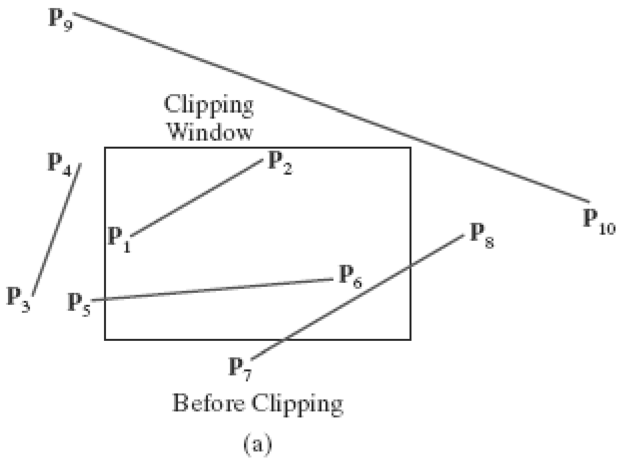
- ❖ For a clipping rectangle in standard position, we save a two-dimensional point **P** = (x, y) for display if the following inequalities are satisfied:

$$xw_{\min} \leq x \leq xw_{\max} \quad \text{and} \quad yw_{\min} \leq y \leq yw_{\max}$$

❖ If any of these four inequalities is not satisfied, the point is clipped

Two-Dimensional Line Clipping

✓ Clipping straight-line segments using a standard rectangular clipping window.



✓ A line-clipping algorithm processes each line in a scene through a series of tests and intersection calculations to determine whether the entire line or any part of it is to be saved.

✓ The expensive part of a line-clipping procedure is in calculating the intersection positions of a line with the window edges.

✓ Therefore, a major goal for any line-clipping algorithm is to minimize the intersection calculations.

✓ To do this, we can first perform tests to determine whether a line segment is completely inside the clipping window or completely outside.

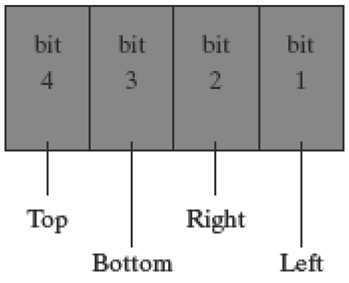
✓ It is easy to determine whether a line is completely inside a clipping window, but it is more difficult to identify all lines that are entirely outside the window.

✓ One way to formulate the equation for a straight-line segment is to use the following parametric representation, where the coordinate positions (x_0, y_0) and (x_{end}, y_{end}) designate the two line endpoints:

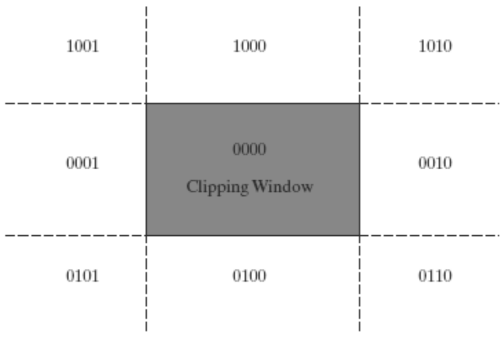
$$\begin{aligned} x &= x_0 + u(x_{\text{end}} - x_0) \\ y &= y_0 + u(y_{\text{end}} - y_0) \qquad 0 \le u \le 1 \end{aligned}$$

Cohen-Sutherland Line Clipping

- ✓ Processing time is reduced in the Cohen-Sutherland method by performing more tests before proceeding to the intersection calculations.
- ✓ Initially, every line endpoint in a picture is assigned a four-digit binary value, called a **region code,** and each bit position is used to indicate whether the point is inside or outside one of the clipping-window boundaries.



- ✓ A possible ordering for the clipping window boundaries corresponding to the bit positions in the Cohen-Sutherland endpoint region code.
- ✓ Thus, for this ordering, the rightmost position (bit 1) references the left clipping-window boundary, and the leftmost position (bit 4) references the top window boundary.
- ✓ A value of 1 (or *true*) in any bit position indicates that the endpoint is outside that window border. Similarly, a value of 0 (or *false*) in any bit position indicates that the endpoint is not outside (it is inside or on) the corresponding window edge.
- ✓ Sometimes, a region code is referred to as an "**out**" **code** because a value of 1 in any bit position indicates that the spatial point is outside the corresponding clipping boundary.
- ✓ The nine binary region codes for identifying the position of a line endpoint, relative to the clipping-window boundaries.



- ✓ Bit values in a region code are determined by comparing the coordinate values (x, y) of an endpoint to the clipping boundaries.

- ✓ Bit 1 is set to 1 if $x < xw_{\min}$, and the other three bit values are determined similarly.
- ✓ To determine a boundary intersection for a line segment, we can use the slopeintercept form of the line equation.
- ✓ For a line with endpoint coordinates (x_0, y_0) and (x_{end}, y_{end}), the y coordinate of the intersection point with a vertical clipping border line can be obtained with the calculation

$$y = y_0 + m(x - x_0)$$

Where the x value is set to either $xw_{\min}$ or $xw_{\max}$, and the slope of the line is calculated as

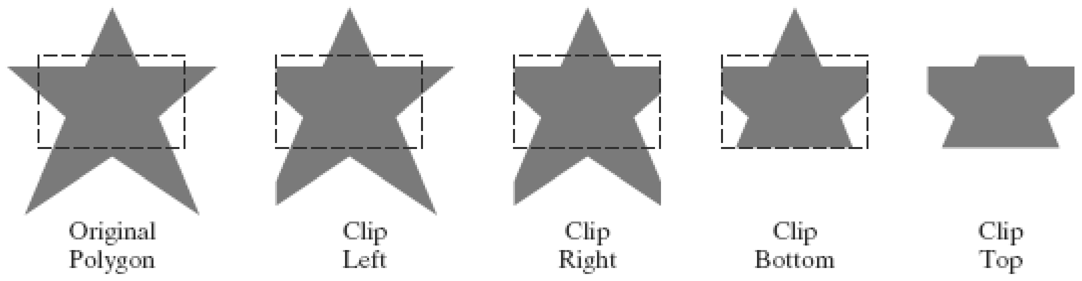
$$m = (y_{\text{end}} - y_0)/(x_{\text{end}} - x_0).$$

- ✓ Similarly, if we are looking for the intersection with a horizontal border, the x coordinate can be calculated as

$x = x_0 + y - y_0/m$, with y set either to $yw_{\min}$ or to $yw_{\max}$.

Polygon Fill-Area Clipping

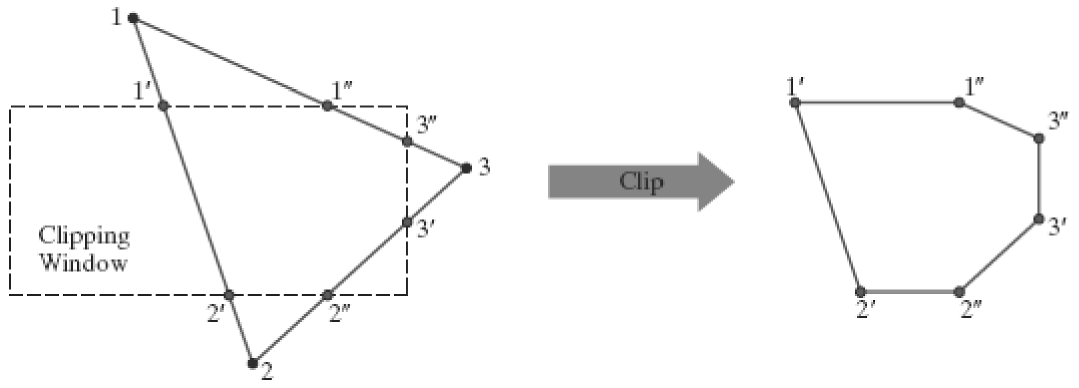
- ➔ To clip a polygon fill area, we cannot apply a line-clipping method to the individual polygon edges directly because this approach would not, in general, produce a closed polyline.
- ➔ We can process a polygon fill area against the borders of a clipping window using the same general approach as in line clipping.
- ➔ We need to maintain a fill area as an entity as it is processed through the clipping stages.
- ➔ Thus, we can clip a polygon fill area by determining the new shape for the polygon as each clipping-window edge is processed, as demonstrated



- ➔ When we cannot identify a fill area as being completely inside or completely outside the clipping window, we then need to locate the polygon intersection positions with the clipping boundaries.

➔ One way to implement convex-polygon clipping is to create a new vertex list at each clipping boundary, and then pass this new vertex list to the next boundary clipper.

➔ The output of the final clipping stage is the vertex list for the clipped polygon



Sutherland--Hodgman Polygon Clipping

❖ An efficient method for clipping a convex-polygon fill area, developed by Sutherland and Hodgman, is to send the polygon vertices through each clipping stage so that a single clipped vertex can be immediately passed to the next stage.

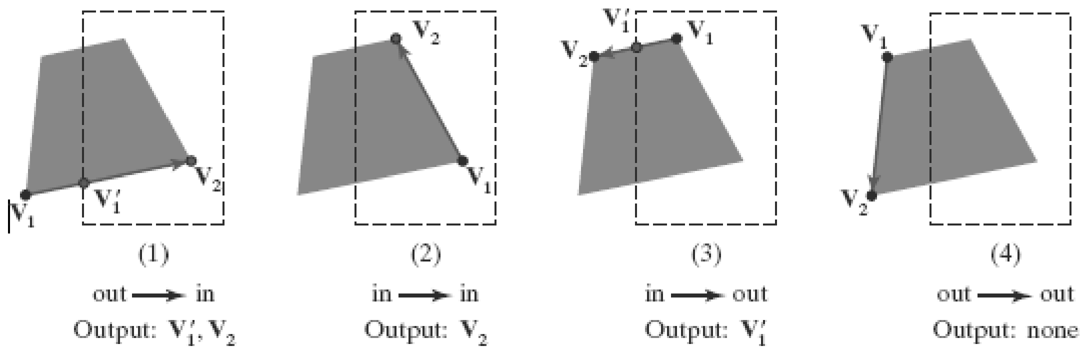
❖ The final output is a list of vertices that describe the edges of the clipped polygon fill area the basic Sutherland-Hodgman algorithm is able to process concave polygons when the clipped fill area can be described with a single vertex list.

❖ The general strategy in this algorithm is to send the pair of endpoints for each successive polygon line segment through the series of clippers (left, right, bottom, and top)

❖ There are four possible cases that need to be considered when processing a polygon edge against one of the clipping boundaries.

1. One possibility is that the first edge endpoint is outside the clipping boundary and the second endpoint is inside.
2. Or, both endpoints could be inside this clipping boundary.
3. Another possibility is that the first endpoint is inside the clipping boundary and the second endpoint is outside.
4. And, finally, both endpoints could be outside the clipping boundary

❖ To facilitate the passing of vertices from one clipping stage to the next, the output from each clipper can be formulated as shown in Figure below



The selection of vertex edge of intersection for each clipper is given as follows

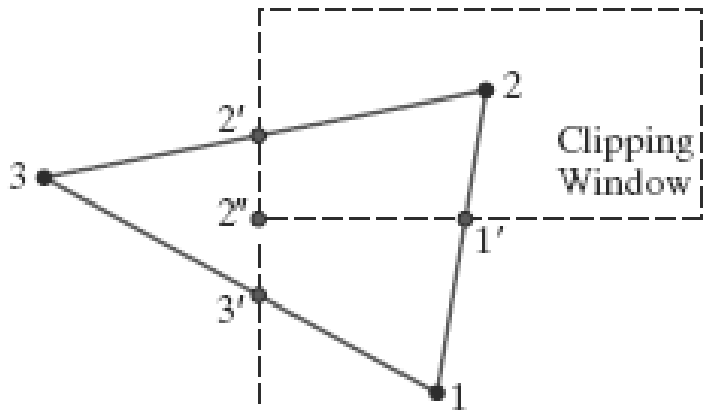
1. If the first input vertex is outside this clipping-window border and the second vertex is inside, both the intersection point of the polygon edge with the window border and the second vertex are sent to the next clipper.

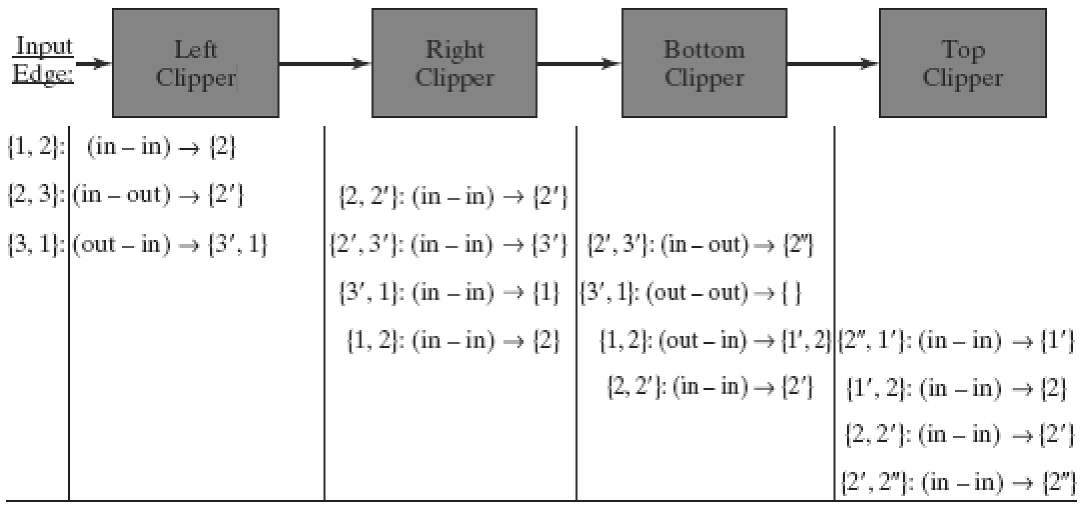
2. If both input vertices are inside this clipping-window border, only the second vertex is sent to the next clipper.

3. If the first vertex is inside this clipping-window border and the second vertex is outside, only the polygon edge-intersection position with the clipping-window border is sent to the next clipper.

4. If both input vertices are outside this clipping-window border, no vertices are sent to the next clipper.

Example





Input Edge:	Left Clipper	Right Clipper	Bottom Clipper	Top Clipper
{1, 2}:	(in – in) → {2}			
{2, 3}:	(in – out) → {2′}	{2, 2′}: (in – in) → {2′}		
{3, 1}:	(out – in) → {3′, 1}	{2′, 3′}: (in – in) → {3′}	{2′, 3′}: (in – out) → {2″}	
		{3′, 1}: (in – in) → {1}	{3′, 1}: (out – out) → { }	
		{1, 2}: (in – in) → {2}	{1, 2}: (out – in) → {1′, 2}	{2″, 1′}: (in – in) → {1′}
			{2, 2′}: (in – in) → {2′}	{1′, 2}: (in – in) → {2}
				{2, 2′}: (in – in) → {2′}
				{2′, 2″}: (in – in) → {2″}

- ❖ When a concave polygon is clipped with the Sutherland-Hodgman algorithm, extraneous lines may be displayed.
- ❖ This occurs when the clipped polygon should have two or more separate sections. But since there is only one output vertex list, the last vertex in the list is always joined to the first vertex.
- ❖ There are several things we can do to display clipped concave polygons correctly.
- ❖ For one, we could split a concave polygon into two or more convexpolygons and process each convex polygon separately using the Sutherland- Hodgman algorithm
- ❖ Another possibility is to modify the Sutherland- Hodgman method so that the final vertex list is checked for multiple intersection points along any clipping-window boundary.
- ❖ If we find more than two vertex positions along any clipping boundary, we can separate the list of vertices into two or more lists that correctly identify the separate sections of the clipped fill area.
- ❖ A third possibility is to use a more general polygon clipper that has been designed to process concave polygons correctly

3.2 3DGeometric Transformations:

3.2.1 Three-Dimensional Geometric Transformations

- Methods for geometric transformations in three dimensions are extended from two dimensional methods by including considerations for the *z* coordinate.
- A three-dimensional position, expressed in homogeneous coordinates, is represented as a four-element column vector

3.2.2 Three-Dimensional Translation

➢ A position **P** = *(x, y, z)* in three-dimensional space is translated to a location **P'**= *(x', y', z')* by adding translation distances *tx*, *ty*, and *tz* to the Cartesian coordinates of **P**:

$$x' = x + t_x, \qquad y' = y + t_y, \qquad z' = z + t_z$$

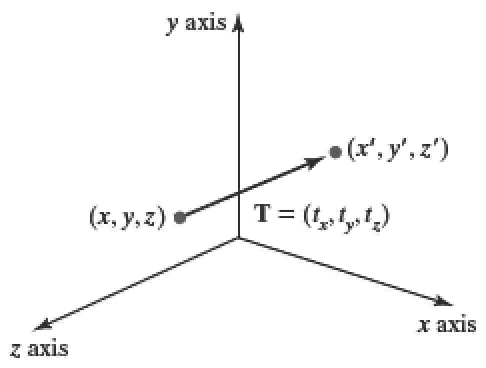
➢ We can express these three-dimensional translation operations in matrix form

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

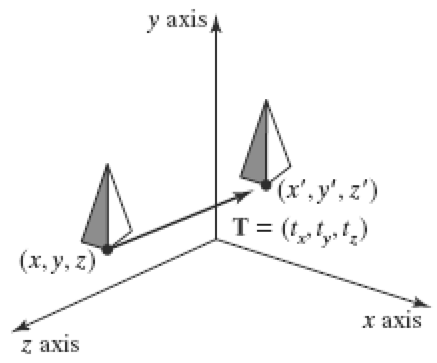
or

$$\mathbf{P}' = \mathbf{T} \cdot \mathbf{P}$$

➢ Moving a coordinate position with translation vector T = (tx , ty , tz) .



- Shifting the position of a three-dimensional object using translation vector **T**.



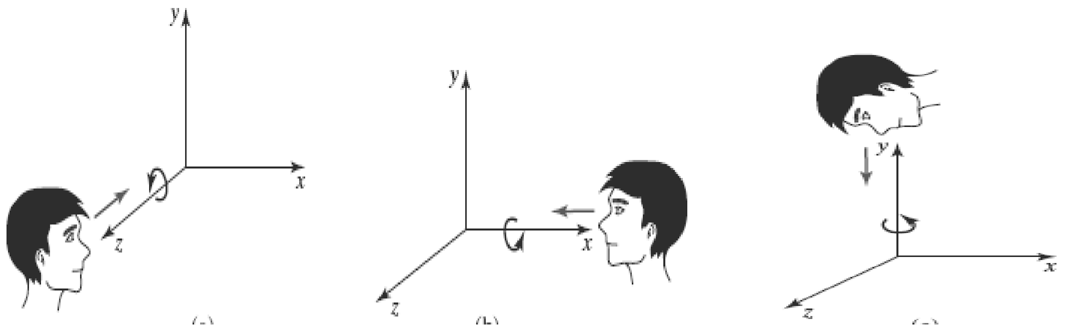
CODE:

```
typedef GLfloat Matrix4x4 [4][4];
/* Construct the 4 x 4 identity matrix. */
void matrix4x4SetIdentity (Matrix4x4 matIdent4x4)
{
        GLint row, col;
        for (row = 0; row < 4; row++)
        for (col = 0; col < 4 ; col++)
        matIdent4x4 [row][col] = (row == col);
}
void translate3D (GLfloat tx, GLfloat ty, GLfloat tz)
{
        Matrix4x4 matTransl3D;
        /* Initialize translation matrix to identity. */
        matrix4x4SetIdentity (matTransl3D);
        matTransl3D [0][3] = tx;
        matTransl3D [1][3] = ty;
        matTransl3D [2][3] = tz;
}
```

- An inverse of a three-dimensional translation matrix is obtained by negating the translation distances *tx*, *ty*, and *tz*

3.2.3 Three-Dimensional Rotation

- ✓ By convention, positive rotation angles produce counterclockwise rotations about a coordinate axis.
- ✓ Positive rotations about a coordinate axis are counterclockwise, when looking along the positive half of the axis toward the origin.



<u>Three-Dimensional Coordinate-Axis Rotations</u>

Along z axis:

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

- ✓ In homogeneous-coordinate form, the three-dimensional *z*-axis rotation equations are

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

- ✓ Transformation equations for rotations about the other two coordinate axes can be obtained with a cyclic permutation of the coordinate parameters *x*, *y*, and *z*

$$x \rightarrow y \rightarrow z \rightarrow x$$

Along x axis

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

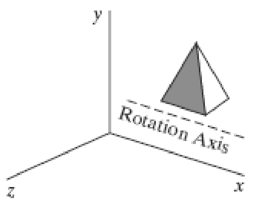
Along y axis

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

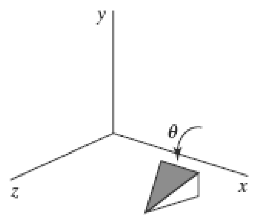
- ✓ An inverse three-dimensional rotation matrix is obtained in the same by replacing θ with $-\theta$.

General Three-Dimensional Rotations

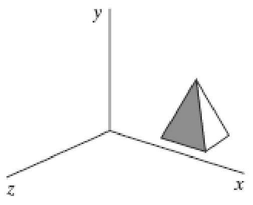
- ✓ A rotation matrix for any axis that does not coincide with a coordinate axis can be set up as a composite transformation involving combinations of translations and the coordinate-axis rotations the following transformation sequence is used:



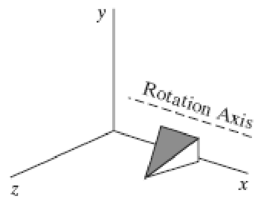
(a)
Original Position of Object



(c)
Rotate Object Through Angle θ



(b)
Translate Rotation Axis onto *x* Axis



(d)
Translate Rotation
Axis to Original Position

1. Translate the object so that the rotation axis coincides with the parallel coordinate axis.

2. Perform the specified rotation about that axis.

3. Translate the object so that the rotation axis is moved back to its original position.

- ✓ A coordinate position **P** is transformed with the sequence shown in this figure as

$$\mathbf{P}' = \mathbf{T}^{-1} \cdot \mathbf{R}_x(\theta) \cdot \mathbf{T} \cdot \mathbf{P}$$

Where the composite rotation matrix for the transformation is

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{R}_x(\theta) \cdot \mathbf{T}$$

- ✓ When an object is to be rotated about an axis that is not parallel to one of the coordinate axes, we must perform some additional transformations we can accomplish the required rotation in five steps:

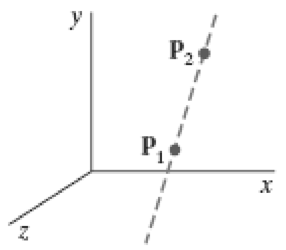
1. Translate the object so that the rotation axis passes through the coordinate origin.

2. Rotate the object so that the axis of rotation coincides with one of the coordinate axes.

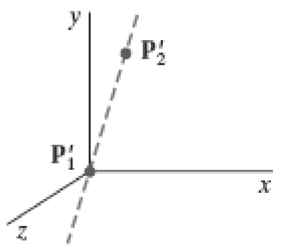
3. Perform the specified rotation about the selected coordinate axis.

4. Apply inverse rotations to bring the rotation axis back to its original orientation.

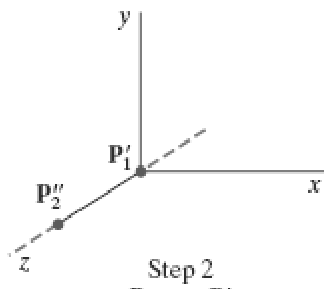
5. Apply the inverse translation to bring the rotation axis back to its original spatial position.



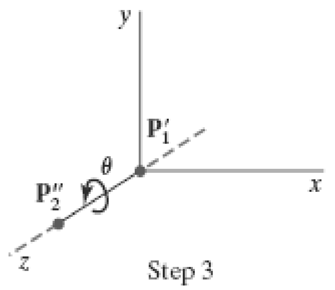
Initial Position



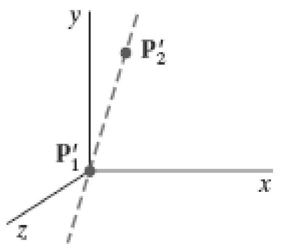
Step 1 Translate $\mathbf{P}_1$ to the Origin



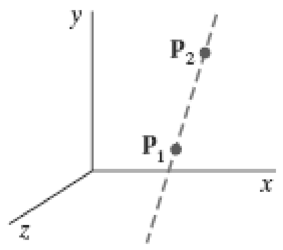
Step 2 Rotate $\mathbf{P}_2'$ onto the z Axis



Step 3 Rotate the Object Around the z Axis



Step 4 Rotate the Axis to Its Original Orientation



Step 5 Translate the Rotation Axis to Its Original Position

- Components of the rotation-axis vector are then computed as

$$\mathbf{V} = \mathbf{P}2 - \mathbf{P}1$$
$$= (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

- The unit rotation-axis vector **u** is

$$\mathbf{u} = \frac{\mathbf{V}}{|\mathbf{V}|} = (a, b, c)$$

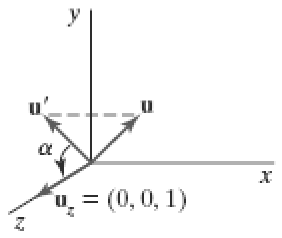
Where the components *a*, *b*, and *c* are the direction cosines for the rotation axis

$$a = \frac{x_2 - x_1}{|\mathbf{V}|}, \qquad b = \frac{y_2 - y_1}{|\mathbf{V}|}, \qquad c = \frac{z_2 - z_1}{|\mathbf{V}|}$$

- The first step in the rotation sequence is to set up the translation matrix that repositions the rotation axis so that it passes through the coordinate origin.
- Translation matrix is given by

$$\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & -x_1 \\ 0 & 1 & 0 & -y_1 \\ 0 & 0 & 1 & -z_1 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Because rotation calculations involve sine and cosine functions, we can use standard vector operations to obtain elements of the two rotation matrices.
- A vector dot product can be used to determine the cosine term, and a vector cross product can be used to calculate the sine term.
- Rotation of u around the x axis into the x z plane is accomplished by rotating u' (the projection of u in the y z plane) through angle α onto the z axis.



- If we represent the projection of **u** in the *yz* plane as the vector **u**'= (0, *b*, *c*), then the cosine of the rotation angle α can be determined from the dot product of **u**' and the unit vector **u***z* along the *z* axis:

$$\cos\alpha = \frac{\mathbf{u}' \cdot \mathbf{u}_z}{|\mathbf{u}'|\,|\mathbf{u}_z|} = \frac{c}{d}$$

where *d* is the magnitude of **u**'

$$d = \sqrt{b^2 + c^2}$$

- The coordinate-independent form of this cross-product is

$$\mathbf{u}' \times \mathbf{u_z} = \mathbf{u}_x\, |\mathbf{u}'|\, |\mathbf{u_z}| \sin\alpha$$

- and the Cartesian form for the cross-product gives us

$$\mathbf{u}' \times \mathbf{u_z} = \mathbf{u}_x \cdot b$$

- Equating the above two equations

$$d \sin\alpha = b$$

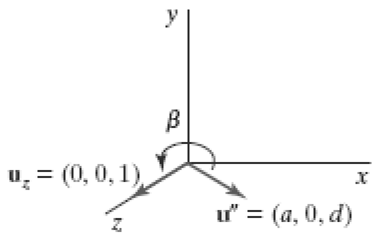
or

$$\sin\alpha = \frac{b}{d}$$

- We have determined the values for cos *α* and sin *α* in terms of the components of vector **u**, the matrix elements for rotation of this vector about the *x* axis and into the *xz* plane

$$\mathbf{R}_x(\alpha) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \frac{c}{d} & -\frac{b}{d} & 0 \\ 0 & \frac{b}{d} & \frac{c}{d} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- Rotation of unit vector u" (vector u after rotation into the x z plane) about the y axis. Positive rotation angle *β* aligns u" with vector uz .



- We can determine the cosine of rotation angle *β* from the dot product of unit vectors **u**'' and **u***z*. Thus,

$$\cos\beta = \frac{\mathbf{u}'' \cdot \mathbf{u}_z}{|\mathbf{u}''|\,|\mathbf{u}_z|} = d$$

- Comparing the coordinate-independent form of the cross-product

$$\mathbf{u}'' \times \mathbf{u}_z = \mathbf{u}_y\,|\mathbf{u}''|\,|\mathbf{u}_z|\sin\beta$$

with the Cartesian form

$$\mathbf{u}'' \times \mathbf{u}_z = \mathbf{u}_y \cdot (-a)$$

- we find that

$$\sin\beta = -a$$

- The transformation matrix for rotation of **u**” about the *y* axis is

$$\mathbf{R}_y(\beta) = \begin{bmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- The specified rotation angle *θ* can now be applied as a rotation about the *z* axis as follows:

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- The transformation matrix for rotation about an arbitrary axis can then be expressed as the composition of these seven individual transformations:

$$\mathbf{R}(\theta) = \mathbf{T}^{-1} \cdot \mathbf{R}_x^{-1}(\alpha) \cdot \mathbf{R}_y^{-1}(\beta) \cdot \mathbf{R}_z(\theta) \cdot \mathbf{R}_y(\beta) \cdot \mathbf{R}_x(\alpha) \cdot \mathbf{T}$$

- The composite matrix for any sequence of three-dimensional rotations is of the form

$$\mathbf{R} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- The upper-left 3 × 3 submatrix of this matrix is orthogonal

$$\mathbf{R}\cdot\begin{bmatrix} r_{11}\\ r_{12}\\ r_{13}\\ 1\end{bmatrix}=\begin{bmatrix}1\\0\\0\\1\end{bmatrix},\qquad \mathbf{R}\cdot\begin{bmatrix} r_{21}\\ r_{22}\\ r_{23}\\ 1\end{bmatrix}=\begin{bmatrix}0\\1\\0\\1\end{bmatrix},\qquad \mathbf{R}\cdot\begin{bmatrix} r_{31}\\ r_{32}\\ r_{33}\\ 1\end{bmatrix}=\begin{bmatrix}0\\0\\1\\1\end{bmatrix}$$

- Assuming that the rotation axis is not parallel to any coordinate axis, we could form the following set of local unit vectors

$$\mathbf{u}'_z=\mathbf{u}$$
$$\mathbf{u}'_y=\frac{\mathbf{u}\times\mathbf{u}_x}{|\mathbf{u}\times\mathbf{u}_x|}$$
$$\mathbf{u}'_x=\mathbf{u}'_y\times\mathbf{u}'_z$$

- If we express the elements of the unit local vectors for the rotation axis as

$$\mathbf{u}'_x=(u'_{x1},u'_{x2},u'_{x3})$$
$$\mathbf{u}'_y=(u'_{y1},u'_{y2},u'_{y3})$$
$$\mathbf{u}'_z=(u'_{z1},u'_{z2},u'_{z3})$$

- Then the required composite matrix, which is equal to the product **R***y(β)* · **R***x(α)*, is

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

<u>Quaternion Methods for Three-Dimensional Rotations</u>

- ✓ A more efficient method for generating a rotation about an arbitrarily selected axis is to use a quaternion representation for the rotation transformation.
- ✓ Quaternions, which are extensions of two-dimensional complex numbers, are useful in a number of computer-graphics procedures, including the generation of fractal objects.
- ✓ One way to characterize a quaternion is as an ordered pair, consisting of a *scalar part* and a *vector part:*

$$q=(s,\mathbf{v})$$

- ✓ *A rotation about any axis passing through the coordinate origin is accomplished by first setting up a unit quaternion with the scalar and vector parts as follows:*

$$s=\cos\frac{\theta}{2},\qquad \mathbf{v}=\mathbf{u}\sin\frac{\theta}{2}$$

- ✓ Any point position **P** that is to be rotated by this quaternion can be represented in quaternion notation as

$$\mathbf{P} = (0, \mathbf{p})$$

- ✓ Rotation of the point is then carried out with the quaternion operation

$$\mathbf{P}' = q\mathbf{P}q^{-1}$$

where $q^{-1} = (s, -\mathbf{v})$ is the inverse of the unit quaternion q

- ✓ This transformation produces the following new quaternion:

$$\mathbf{P}' = (0, \mathbf{p}')$$

- ✓ The second term in this ordered pair is the rotated point position **p'**, which is evaluated with vector dot and cross-products as

$$\mathbf{p}' = s^2\mathbf{p} + \mathbf{v}(\mathbf{p} \cdot \mathbf{v}) + 2s(\mathbf{v} \times \mathbf{p}) + \mathbf{v} \times (\mathbf{v} \times \mathbf{p})$$

- ✓ Designating the components of the vector part of q as $\mathbf{v} = (a, b, c)$, we obtain the elements for the composite rotation matrix

$$\mathbf{M}_R(\theta) = \begin{bmatrix} 1 - 2b^2 - 2c^2 & 2ab - 2sc & 2ac + 2sb \\ 2ab + 2sc & 1 - 2a^2 - 2c^2 & 2bc - 2sa \\ 2ac - 2sb & 2bc + 2sa & 1 - 2a^2 - 2b^2 \end{bmatrix}$$

- ✓ Using the following trigonometric identities to simplify the terms

$$\cos^2\frac{\theta}{2} - \sin^2\frac{\theta}{2} = 1 - 2\sin^2\frac{\theta}{2} = \cos\theta, \qquad 2\cos\frac{\theta}{2}\sin\frac{\theta}{2} = \sin\theta$$

we can rewrite Matrix as

$$\mathbf{M}_R(\theta) =$$
$$\begin{bmatrix} u_x^2(1-\cos\theta) + \cos\theta & u_xu_y(1-\cos\theta) - u_z\sin\theta & u_xu_z(1-\cos\theta) + u_y\sin\theta \\ u_yu_x(1-\cos\theta) + u_z\sin\theta & u_y^2(1-\cos\theta) + \cos\theta & u_yu_z(1-\cos\theta) - u_x\sin\theta \\ u_zu_x(1-\cos\theta) - u_y\sin\theta & u_zu_y(1-\cos\theta) + u_x\sin\theta & u_z^2(1-\cos\theta) + \cos\theta \end{bmatrix}$$

3.2.4 Three-Dimensional Scaling

- ✓ The matrix expression for the three-dimensional scaling transformation of a position **P** = *(x, y, z) is given by*

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

✓ The three-dimensional scaling transformation for a point position can be represented as

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P}$$

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

✓ Explicit expressions for the scaling transformation relative to the origin are

$$x' = x \cdot s_x, \qquad y' = y \cdot s_y, \qquad z' = z \cdot s_z$$

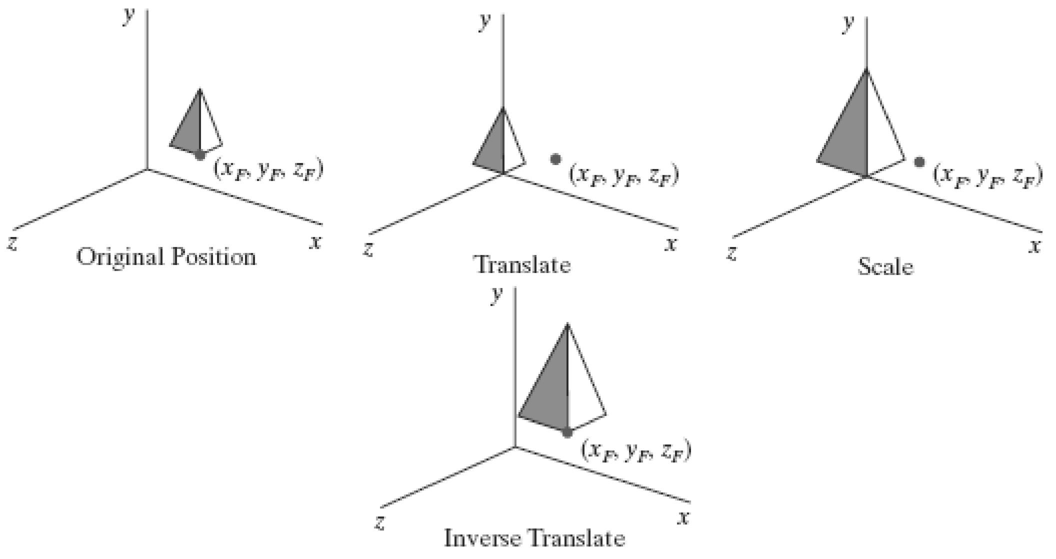
✓ Because some graphics packages provide only a routine that scales relative to the coordinate origin, we can always construct a scaling transformation with respect to any selected *fixed position* (x_f, y_f, z_f) using the following transformation sequence:

1. Translate the fixed point to the origin.

2. Apply the scaling transformation relative to the coordinate origin

3. Translate the fixed point back to its original position.

✓ This sequence of transformations is demonstrated



$$\mathbf{T}(x_f, y_f, z_f) \cdot \mathbf{S}(s_x, s_y, s_z) \cdot \mathbf{T}(-x_f, -y_f, -z_f) = \begin{bmatrix} s_x & 0 & 0 & (1 - s_x)x_f \\ 0 & s_y & 0 & (1 - s_y)y_f \\ 0 & 0 & s_z & (1 - s_z)z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

CODE:

```
class wcPt3D
{
        private:
        GLfloat x, y, z;
        public:
        /* Default Constructor:
        * Initialize position as (0.0, 0.0, 0.0).
        */
        wcPt3D ( ) {
                x = y = z = 0.0;
        }
        setCoords (GLfloat xCoord, GLfloat yCoord, GLfloat zCoord) {
                x = xCoord;
                y = yCoord;
                z = zCoord;
        }
        GLfloat getx ( ) const {
                return x;
        }
        GLfloat gety ( ) const {
                return y;
        }
        GLfloat getz ( ) const {
                return z;
        }
};
typedef float Matrix4x4 [4][4];
void scale3D (GLfloat sx, GLfloat sy, GLfloat sz, wcPt3D fixedPt)
{
        Matrix4x4 matScale3D;
        /* Initialize scaling matrix to identity. */
        matrix4x4SetIdentity (matScale3D);
        matScale3D [0][0] = sx;
        matScale3D [0][3] = (1 - sx) * fixedPt.getx ( );
        matScale3D [1][1] = sy;
        matScale3D [1][3] = (1 - sy) * fixedPt.gety ( );
        matScale3D [2][2] = sz;
        matScale3D [2][3] = (1 - sz) * fixedPt.getz ( );
}
```

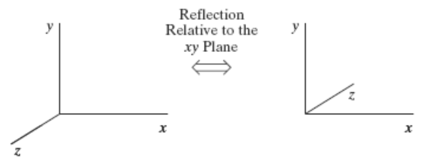
3.2.5 Composite Three-Dimensional Transformations

- We form a composite threedimensional transformation by multiplying the matrix representations for the individual operations in the transformation sequence.
- We can implement a transformation sequence by concatenating the individual matrices from right to left or from left to right, depending on the order in which the matrix representations are specified

3.2.6 Other Three-Dimensional Transformations

Three-Dimensional Reflections

- A reflection in a three-dimensional space can be performed relative to a selected *reflection axis* or with respect to a *reflection plane*.
- Reflections with respect to a plane are similar; when the reflection plane is a coordinate plane (x_y, x_z, or y_z), we can think of the transformation as a 180◦ rotation in four-dimensional space with a conversion between a left-handed frame and a right-handed frame
- An example of a reflection that converts coordinate specifications froma right handed system to a left-handed system is shown below



➔ The matrix representation for this reflection relative to the *xy* plane is

$$M_{\text{zreflect}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Three-Dimensional Shears

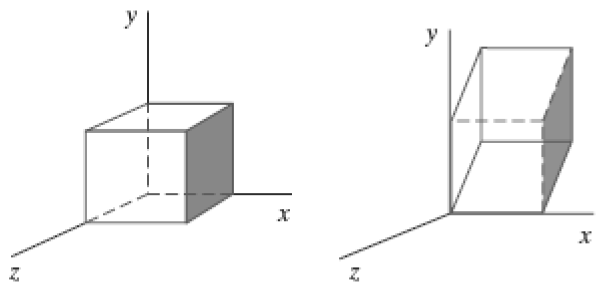
➔ These transformations can be used to modify object shapes.

➔ For three-dimensional we can also generate shears relative to the *z* axis.

➔ A general *z*-axis shearing transformation relative to a selected reference position is produced with the following matrix:

$$M_{\text{zshear}} = \begin{bmatrix} 1 & 0 & \text{sh}_{zx} & -\text{sh}_{zx} \cdot z_{\text{ref}} \\ 0 & 1 & \text{sh}_{zy} & -\text{sh}_{zy} \cdot z_{\text{ref}} \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

➔ The Below figure shows the shear transformation of a cube



A unit cube (a) is sheared relative to the origin (b) by Matrix 46, with $\text{sh}_{zx} = \text{sh}_{zy} = 1$.

3.2.7 Affine Transformations

❖ A coordinate transformation of the form

$$x' = a_{xx}x + a_{xy}y + a_{xz}z + b_x$$
$$y' = a_{yx}x + a_{yy}y + a_{yz}z + b_y$$
$$z' = a_{zx}x + a_{zy}y + a_{zz}z + b_z$$

is called an **affine transformation**

- ❖ Affine transformations (in two dimensions, three dimensions, or higher dimensions) have the general properties that parallel lines are transformed into parallel lines, and finite points map to finite points.
- ❖ Translation, rotation, scaling, reflection,andshear are examples of affine transformations.
- ❖ Another example of an affine transformation is the conversion of coordinate descriptions for a scene from one reference system to another because this transformation can be described as a combination of translation and rotation

3.2.8 OpenGL Geometric-Transformation Functions

OpenGL Matrix Stacks

glMatrixMode:

- ❖ used to select the modelview composite transformation matrix as the target of subsequent OpenGL transformation calls
- ❖ four modes: modelview, projection, texture, and color
- ❖ the top matrix on each stack is called the "current matrix"**.**
- ❖ for that mode. the **modelview matrix stack** is the 4 × 4 composite matrix that combines the viewing transformations and the various geometric transformations that we want to apply to a scene.
- ❖ OpenGL supports a modelview stack depth of at least 32,

glGetIntegerv (GL_MAX_MODELVIEW_STACK_DEPTH, stackSize);

- ❖ determine the number of positions available in the modelview stack for a particular implementation of OpenGL.
- ❖ It returns a single integer value to array **stackSize**
- ❖ **other OpenGL symbolic constants:** GL_MAX_PROJECTION_STACK_DEPTH, GL_MAX_TEXTURE_STACK_DEPTH, or GL_MAX_COLOR_STACK_DEPTH**.**
- ❖ We can also find out how many matrices are currently in the stack with

  glGetIntegerv (GL_MODELVIEW_STACK_DEPTH, numMats);

We have two functions available in OpenGL for processing the matrices in a stack

glPushMatrix ( );

Copy the current matrix at the top of the active stack and store that copy in the second stack position

glPopMatrix ( );

which destroys the matrix at the top of the stack, and the second matrix in the stack becomes the current matrix

3.3 Illumination and Color

3.3.1 Illumination Models

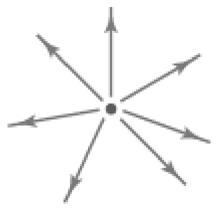
- ✓ An **illumination model,** also called a **lighting model** (and sometimes referred to as a *shading model*), is used to calculate the color of an illuminated position on the surface of an object

3.3.2 Light Sources

- ✓ Any object that is emitting radiant energy is a **light source** that contributes to the lighting effects for other objects in a scene.
- ✓ We can model light sources with a variety of shapes and characteristics, and most emitters serve only as a source of illumination for a scene.
- ✓ A light source can be defined with a number of properties. We can specify its position, the color of the emitted light, the emission direction, and its shape.
- ✓ We could set up a light source that emits different colors in different directions.
- ✓ We assign light emitting properties using a single value for each of the red, green, and blue (RGB) color components, which we can describe as the amount, or the "intensity," of that color component.

<u>Point Light Sources</u>

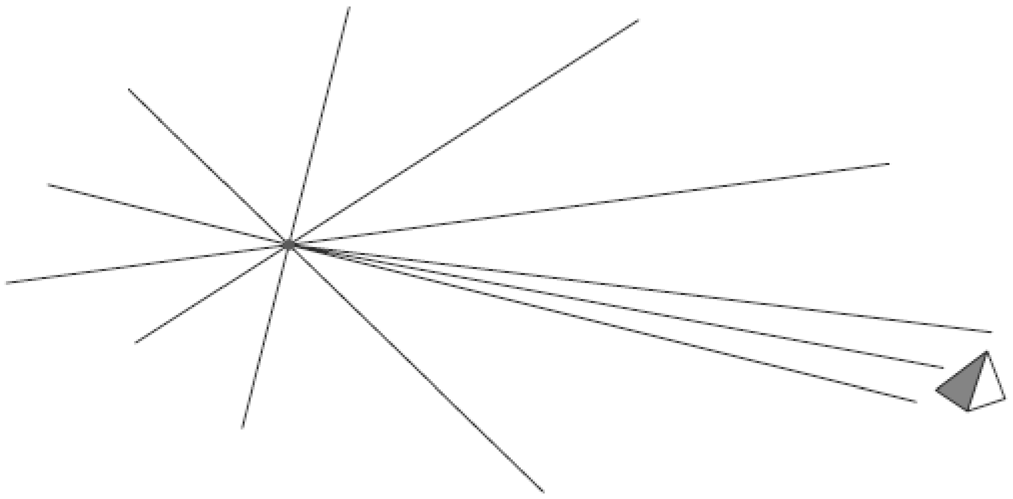
- ❖ The simplest model for an object that is emitting radiant energy is a **point light source** with a single color, specified with three RGB components



- ❖ A point source for a scene by giving its position and the color of the emitted light. light rays are generated along radially diverging paths from the single-color source position.
- ❖ This light-source model is a reasonable approximation for sources whose dimensions are small compared to the size of objects in the scene

Infinitely Distant Light Sources

- ❖ A large light source, such as the sun, that is very far from a scene can also be approximated as a point emitter, but there is little variation in its directional effects.
- ❖ The light path from a distant light source to any position in the scene is nearly constant



- ❖ We can simulate an infinitely distant light source by assigning it a color value and a fixed direction for the light rays emanating from the source.
- ❖ The vector for the emission direction and the light-source color are needed in the illumination calculations, but not the position of the source.

Radial Intensity Attenuation

- ➢ As radiant energy from a light source travels outwards through space, its amplitude at any distance d_l from the source is attenuated by the factor $1/d^2$ a surface close to the light source receives a higher incident light intensity from that source than a more distant surface.

- However, using an attenuation factor of $1/d_l^2$ with a point source does not always produce realistic pictures.
- The factor $1/d_l^2$ tends to produce too much intensity variation for objects that are close to the light source, and very little variation when d_l is large
- We can attenuate light intensities with an inverse quadratic function of *dl* that includes a linear term:

$$f_{\text{radatten}}(d_l) = \frac{1}{a_0 + a_1 d_l + a_2 d_l^2}$$

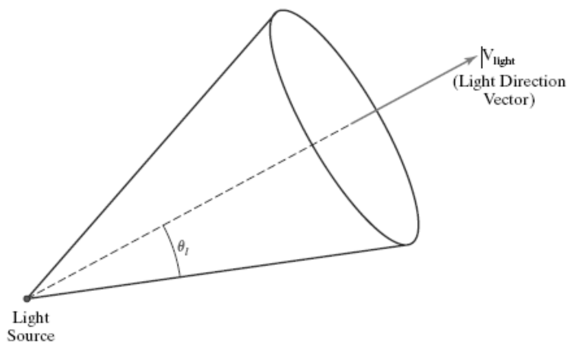
- The numerical values for the coefficients, a_0, a_1, and a_2, can then be adjusted to produce optimal attenuation effects.
- We cannot apply intensity-attenuation calculation 1 to a point source at "infinity," because the distance to the light source is indeterminate.
- We can express the intensity-attenuation function as

$$f_{l,\text{radatten}} = \begin{cases} 1.0, & \text{if source is at infinity} \\ \dfrac{1}{a_0 + a_1 d_l + a_2 d_l^2}, & \text{if source is local} \end{cases}$$

<u>Directional Light Sources and Spotlight Effects</u>

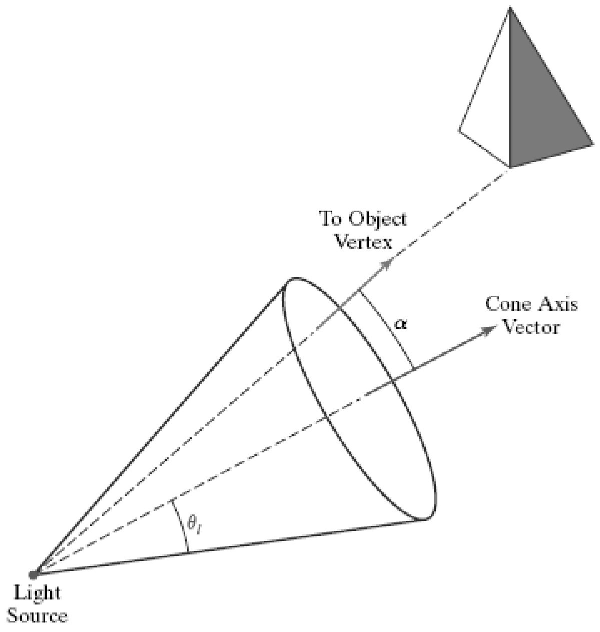
- A local light source can be modified easily to produce a directional, or spotlight, beam of light.
- If an object is outside the directional limits of the light source, we exclude it from illumination by that source
- One way to set up a directional light source is to assign it a vector direction and an angular limit θ_l measured from that vector direction, in addition to its position and color
- We can denote$\mathbf{V}_{\text{light}}$ as the unit vector in the light-source direction and$\mathbf{V}_{\text{obj}}$ as the unit vector in the direction from the light position to an object position.

Then $\mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}} = \cos\alpha$



where angle α is the angular distance of the object from the light direction vector.

➔ If we restrict the angular extent of any light cone so that $0° < \theta_l \leq 90°$, then the object is within the spotlight if $\cos\alpha \geq \cos\theta_l$, as shown



➔ . If $\mathbf{V}_{obj} \cdot \mathbf{V}_{light} < \cos\theta_l$, however, the object is outside the light cone.

Angular Intensity Attenuation

- For a directional light source, we can attenuate the light intensity angularly about the source as well as radially out from the point-source position
- This allows intensity decreasing as we move farther from the cone axis.
- A commonly used angular intensity-attenuation function for a directional light source is

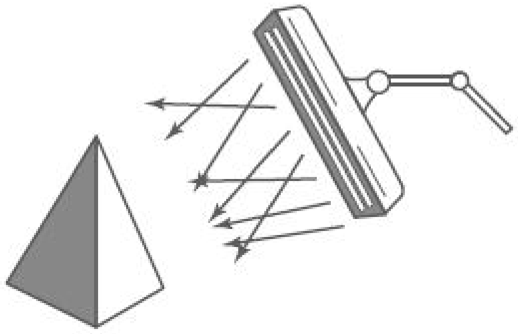
$$f_{\text{angatten}}(\phi) = \cos^{a_l} \phi, \qquad 0^\circ \le \phi \le \theta$$

- Where the attenuation exponent *al* is assigned some positive value and angle *φ* is measured from the cone axis.
- The greater the value for the attenuation exponent *al* , the smaller the value of the angular intensity-attenuation function for a given value of angle*φ* > 0◦.
- There are several special cases to consider in the implementation of the angular-attenuation function.
- There is no angular attenuation if the light source is not directional (not a spotlight).
- We can express the general equation for angular attenuation as

$$f_{l,\text{angatten}} = \begin{cases} 1.0, & \text{if source is not a spotlight} \\ 0.0, & \text{if } \mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}} = \cos\alpha < \cos\theta_l \\ & \quad \text{(object is outside the spotlight cone)} \\ (\mathbf{V}_{\text{obj}} \cdot \mathbf{V}_{\text{light}})^{a_l}, & \text{otherwise} \end{cases}$$

Extended Light Sources and the Warn Model

- ✓ When we want to include a large light source at a position close to the objects in a scene, such as the long neon lamp, we can approximate it as a lightemitting surface



- ✓ One way to do this is to model the light surface as a grid of directional point emitters.
- ✓ We can set the direction for the point sources so that objects behind the light-emitting surface are not illuminated.
- ✓ We could also include other controls to restrict the direction of the emitted light near the edges of the source

- ✓ The **Warn model** provides a method for producing studio lighting effects using sets of point emitters with various parameters to simulate the barn doors, flaps, and spotlighting controls employed by photographers.
- ✓ Spotlighting is achieved with the cone of light discussed earlier, and the flaps and barn doors provide additional directional control

3.3.3 Basic Illumination Models

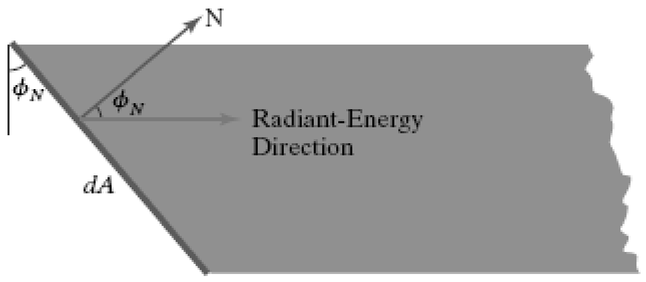
- ➢ Light-emitting objects in a basic illumination model are generally limited to point sources many graphics packages provide additional functions for dealing with directional lighting (spotlights) and extended light sources.

Ambient Light

- ➢ This produces a uniform ambient lighting that is the same for all objects, and it approximates the global diffuse reflections from the various illuminated surfaces.
- ➢ Reflections produced by ambient-light illumination are simply a form of diffuse reflection, and they are independent of the viewing direction and the spatial orientation of a surface.
- ➢ However, the amount of the incident ambient light that is reflected depends on surface optical properties, which determine how much of the incident energy is reflected and how much is absorbed

Diffuse Reflection

- ➢ The incident light on the surface is scattered with equal intensity in all directions, independent of the viewing position.
- ➢ Such surfaces are called **ideal diffuse reflectors** They are also referred to as **Lambertian reflectors,** because the reflected radiant light energy fromany point on the surface is calculated with **Lambert's cosine law.**
- ➢ This law states that the amount of radiant energy coming from any small surface area dAin a direction φN relative to the surface normal is proportional to cos φN



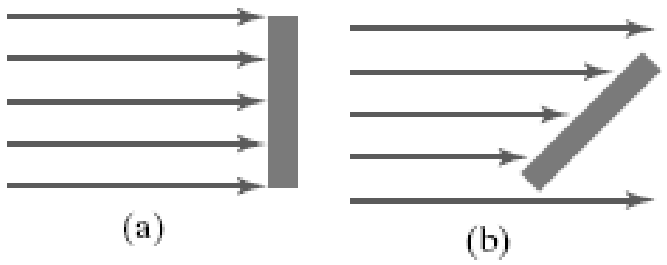
- The intensity of light in this direction can be computed as the ratio of the magnitude of the radiant energy per unit time divided by the projection of the surface area in the radiation direction:

$$\begin{aligned}\text{Intensity} &= \frac{\text{radiant energy per unit time}}{\text{projected area}} \\ &\propto \frac{\cos\phi_N}{dA\cos\phi_N} \\ &= \text{constant}\end{aligned}$$

- Assuming that every surface is to be treated as an ideal diffuse reflector (Lambertian), we can set a parameter *kd* for each surface that determines the fraction of the incident light that is to be scattered as diffuse reflections.
- This parameter is called the **diffuse-reflection coefficient** or the **diffuse reflectivity. T**he ambient contribution to the diffuse reflection at any point on a surface is simply

$$I_{\text{ambdiff}} = k_d I_a$$

- The below figure illustrates this effect, showing a beam of light rays incident on two equal-area plane surface elements with different spatial orientations relative to the illumination direction from a distant source



A surface that is perpendicular to the direction of the incident light (a) is more illuminated than an equal-sized surface at an oblique angle (b) to the incoming light direction.

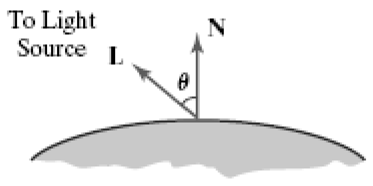
- We can model the amount of incident light on a surface from a source with intensity I_l as

$$I_{l,\text{incident}} = I_l \cos\theta$$

- We can model the diffuse reflections from a light source with intensity I_l using the calculation

$$\begin{aligned} I_{l,\text{diff}} &= k_d I_{l,\text{incident}} \\ &= k_d I_l \cos\theta \end{aligned}$$

- At any surface position, we can denote the unit normal vector as **N** and the unit direction vector to a point source as **L**,



- The diffuse reflection equation for single point-source illumination at a surface position can be expressed in the form

$$I_{l,\text{diff}} = \begin{cases} k_d I_l (\mathbf{N} \cdot \mathbf{L}), & \text{if } \mathbf{N} \cdot \mathbf{L} > 0 \\ 0.0, & \text{if } \mathbf{N} \cdot \mathbf{L} \leq 0 \end{cases}$$

- The unit direction vector **L** to a nearby point light source is calculated using the surface position and the light-source position:

$$\mathbf{L} = \frac{\mathbf{P}_{\text{source}} - \mathbf{P}_{\text{surf}}}{|\mathbf{P}_{\text{source}} - \mathbf{P}_{\text{surf}}|}$$

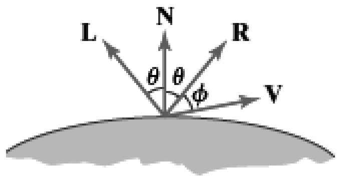
- We can combine the ambient and point-source intensity calculations to obtain an expression for the total diffuse reflection at a surface position
- Using parameter k_a , we can write the total diffuse-reflection equation for a single point source as

$$I_{\text{diff}} = \begin{cases} k_a I_a + k_d I_l (\mathbf{N} \cdot \mathbf{L}), & \text{if } \mathbf{N} \cdot \mathbf{L} > 0 \\ k_a I_a, & \text{if } \mathbf{N} \cdot \mathbf{L} \leq 0 \end{cases}$$

- Where both k_a and k_d depend on surface material properties and are assigned values in the range from 0 to 1.0 for monochromatic lighting effects

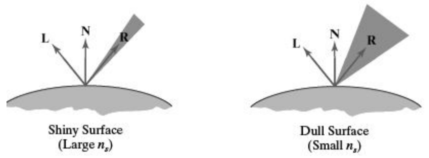
Specular Reflection and the Phong Model

- ✓ The bright spot, or specular reflection, that we can see on a shiny surface is the result of total, or near total, reflection of the incident light in a concentrated region around the **specular-reflection angle**.
- ✓ The below figure shows the specular reflection direction for a position on an illuminated surface

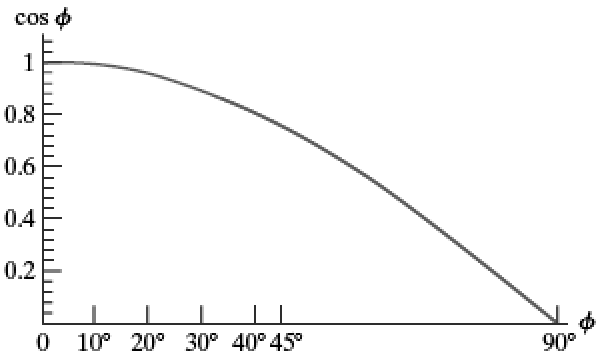


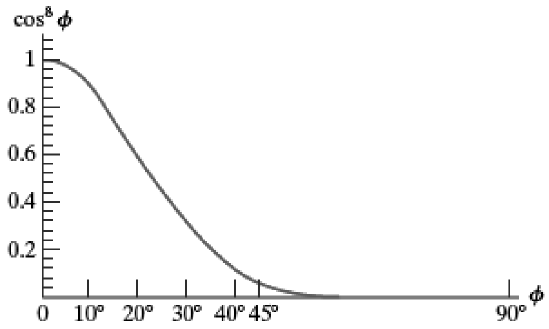
1. **N** represents: unit normal surface vector The specular reflection angle equals the angle of the incident light, with the two angles measured on opposite sides of the unit normal surface vector **N**
2. **R**represents the unit vector in the direction of ideal specular reflection,
3. **L** is the unit vector directed toward the point light source, and
4. **V**is the unit vector pointing to the viewer fromthe selected surface position.

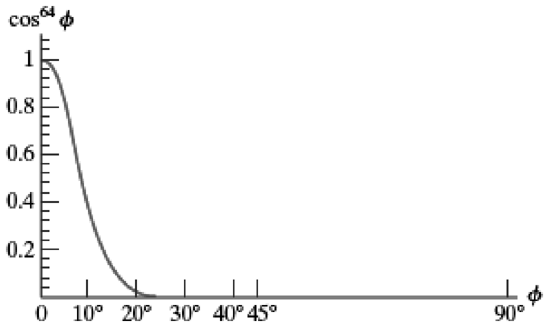
- ✓ Angle φ is the viewing angle relative to the specular-reflection direction **R**
- ✓ An empirical model for calculating the specular reflection range, developed by Phong Bui Tuong and called the **Phong specular-reflection model** or simply the **Phon G model,** sets the intensity of specular reflection proportional to $\cos^{ns} \varphi$
- ✓ **Angle φ** can be assigned values in the range 0◦ to 90◦, so that cos φ varies from 0 to 1.0.
- ✓ The value assigned to the specular-reflection exponent *ns* is determined by the type of surface that we want to display.
- ✓ A very shiny surface is modeled with a large value for *ns* (say, 100 or more), and smaller values (down to 1) are used for duller surfaces.
- ✓ For a perfect reflector, *ns* is infinite. For a rough surface, such as chalk or cinderblock, *ns* is assigned a value near 1.

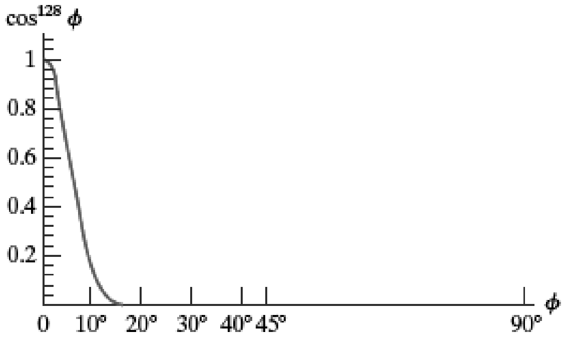


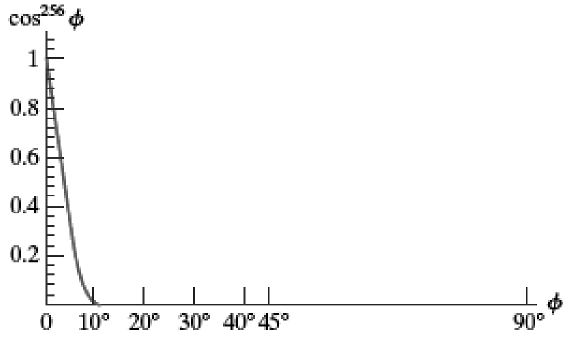
- ✓ Plots of cosns *φ* using five different values for the specular exponent n_s .



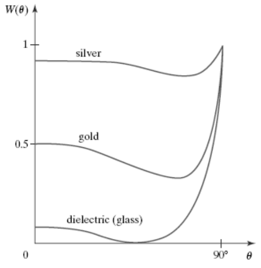








- ✓ We can approximately model monochromatic specular intensity variations using a **specular-reflection coefficient,***W(θ)*, for each surface.
- ✓ In general, *W(θ)* tends to increase as the angle of incidence increases. At *θ* = 90◦, all the incident light is reflected (*W(θ)* = 1).



- ✓ Using the spectral-reflection function $W(\theta)$, we can write the Phong specular-reflection model as

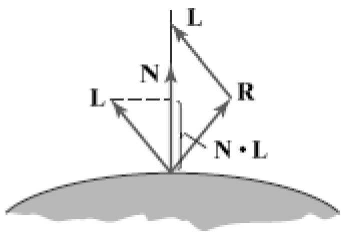
$$I_{l,\text{spec}} = W(\theta) I_l \cos^{n_s} \phi$$

where I_l is the intensity of the light source, and φ is the viewing angle relative to the specular-reflection direction **R**.

- ✓ Because**V**and **R** are unit vectors in the viewing and specular-reflection directions, we can calculate the value of cos φ with the dot product **V·R**.
- ✓ In addition, no specular effects are generated for the display of a surface if **V** and **L** are on the same side of the normal vector **N** or if the light source is behind the surface
- ✓ We can determine the intensity of the specular reflection due to a point light source at a surface position with the calculation

$$I_{l,\text{spec}} = \begin{cases} k_s I_l (\mathbf{V} \cdot \mathbf{R})^{n_s}, & \text{if } \mathbf{V} \cdot \mathbf{R} > 0 \quad \text{and} \quad \mathbf{N} \cdot \mathbf{L} > 0 \\ 0.0, & \text{if } \mathbf{V} \cdot \mathbf{R} \le 0 \quad \text{or} \quad \mathbf{N} \cdot \mathbf{L} \le 0 \end{cases}$$

- ✓ The direction for **R**, the reflection vector, can be computed from the directions for vectors **L** and **N**.



- ✓ The projection of **L** onto the direction of the normal vector has a magnitude equal to the dot product**N·L**, which is also equal to the magnitude of the projection of unit vector **R** onto the direction of **N**.
- ✓ Therefore, from this diagram, we see that

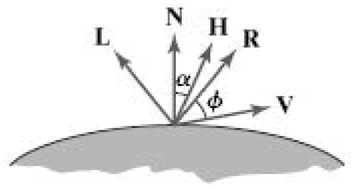
$$\mathbf{R} + \mathbf{L} = (2\mathbf{N}\cdot\mathbf{L})\mathbf{N}$$

and the specular-reflection vector is obtained as

$$\mathbf{R} = (2\mathbf{N}\cdot\mathbf{L})\mathbf{N} - \mathbf{L}$$

- ✓ A somewhat simplified Phong model is obtained using the **halfway vector H** between **L** and **V** to calculate the range of specular reflections.

- ✓ If we replace **V·R** in the Phong model with the dot product**N·H**, this simply replaces the empirical cos φ calculation with the empirical cos α calculation



- ✓ The halfway vector is obtained as

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{|\mathbf{L} + \mathbf{V}|}$$

- ✓ For nonplanar surfaces, **N·H** requires less computation than **V·R** because the calculation of **R** at each surface point involves the variable vector **N**.

3.3.4 OpenGL Illumination Functions

<u>OpenGL Point Light-Source Function</u>

glLight* (lightName, lightProperty, propertyValue);

- ➔ A suffix code of **i** or **f** is appended to the function name, depending on the data type of the property value
- ➔ **lightName:** GL_LIGHT0, GL_LIGHT1, GL_LIGHT2, . . . , GL_LIGHT7
- ➔ **lightProperty:** must be assigned one of the OpenGL symbolic property constants

glEnable (lightName);➔ turn on that light with the command

glEnable (GL_LIGHTING);➔ activate the OpenGL lighting routines

<u>Specifying an OpenGL Light-Source Position and Type</u>

GL_POSITION:

- ➔ specifies light-source position
- ➔ this symbolic constant is used to set two light-source properties at the same time: the light-source position and the *light-source type*

Example:

GLfloat light1PosType [ ] = {2.0, 0.0, 3.0, 1.0};

GLfloat light2PosType [ ] = {0.0, 1.0, 0.0, 0.0};

glLightfv (GL_LIGHT1, GL_POSITION, light1PosType);

glEnable (GL_LIGHT1);

glLightfv (GL_LIGHT2, GL_POSITION, light2PosType);

glEnable (GL_LIGHT2);

Specifying OpenGL Light-Source Colors

- ➔ Unlike an actual light source, an OpenGL light has three different color properties the symbolic color-property constants **GL_AMBIENT, GL_DIFFUSE,** and **GL_SPECULAR**

Example:

GLfloat blackColor [ ] = {0.0, 0.0, 0.0, 1.0};

GLfloat whiteColor [ ] = {1.0, 1.0, 1.0, 1.0};

glLightfv (GL_LIGHT3, GL_AMBIENT, blackColor);

glLightfv (GL_LIGHT3, GL_DIFFUSE, whiteColor);

glLightfv (GL_LIGHT3, GL_SPECULAR, whiteColor);

Specifying Radial-Intensity Attenuation Coefficients

- ➔ For an OpenGL Light Source we could assign the radial-attenuation coefficient values as

  glLightf (GL_LIGHT6, GL_CONSTANT_ATTENUATION, 1.5);

  glLightf (GL_LIGHT6, GL_LINEAR_ATTENUATION, 0.75);

  glLightf (GL_LIGHT6, GL_QUADRATIC_ATTENUATION, 0.4);

OpenGL Directional Light Sources (Spotlights)

- ➔ There are three OpenGL property constants for directional effects: **GL_SPOT_DIRECTION, GL_SPOT_CUTOFF**, and **GL_SPOT_EXPONENT**

  GLfloat dirVector [ ] = {1.0, 0.0, 0.0};

  glLightfv (GL_LIGHT3, GL_SPOT_DIRECTION, dirVector);

  glLightf (GL_LIGHT3, GL_SPOT_CUTOFF, 30.0);

  glLightf (GL_LIGHT3, GL_SPOT_EXPONENT, 2.5);

OpenGL Global Lighting Parameters

glLightModel* (paramName, paramValue);

- ➔ We append a suffix code of i or f, depending on the data type of the parameter value.
- ➔ In addition, for vector data, we append the suffix code v.
- ➔ Parameter paramName is assigned an OpenGL symbolic constant that identifies the global property to be set, and parameter paramValue is assigned a single value or set of values.

```
globalAmbient [ ] = {0.0, 0.0, 0.3, 1.0);
glLightModelfv (GL_LIGHT_MODEL_AMBIENT, globalAmbient);
```

glLightModeli (GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);

- ➔ turn off this default and use the actual viewing position (which is the viewing-coordinate origin) to calculate V

Texture

- ➔ patterns are combined only with the nonspecular color, and then the two colors are combined.
- ➔ We select this two-color option with

glLightModeli (GL_LIGHT_MODEL_COLOR_CONTROL,
GL_SEPARATE_SPECULAR_COLOR);

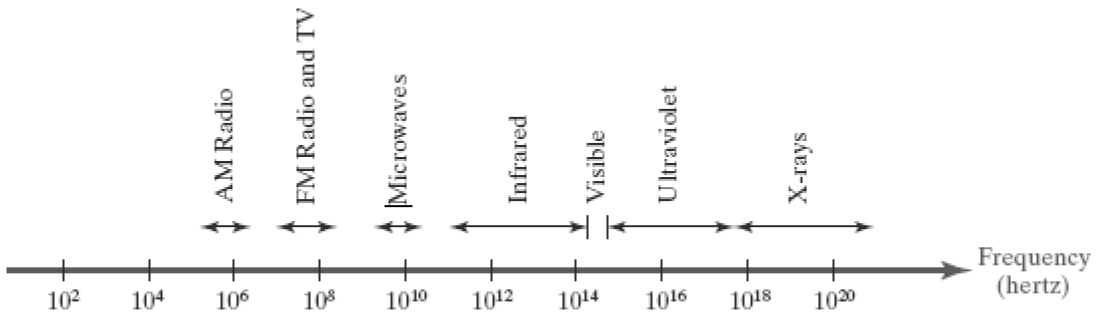
Color Models

3.3.5 Properties of Light

- ✓ We can characterize light as radiant energy, but we also need other concepts to describe our perception of light.

The Electromagnetic Spectrum

- ✓ Color is electromagnetic radiation within a narrow frequency band.
- ✓ Some of the other frequency groups in the electromagnetic spectrum are referred to as radio waves, microwaves, infrared waves, and X-rays. The frequency is shown below



- ✓ Each frequency value within the visible region of the electromagnetic spectrum corresponds to a distinct **spectral color.**
- ✓ At the low-frequency end (approximately 3.8×1014 hertz) are the red colors, and at the high-frequency end (approximately 7.9 × 1014 hertz) are the violet colors.
- ✓ In the wave model of electromagnetic radiation, light can be described as oscillating transverse electric and magnetic fields propagating through space.
- ✓ The electric and magnetic fields are oscillating in directions that are perpendicular to each other and to the direction of propagation.
- ✓ For one spectral color (a monochromatic wave), the wavelength and frequency are inversely proportional to each other, with the proportionality constant as the speed of light (*c*):

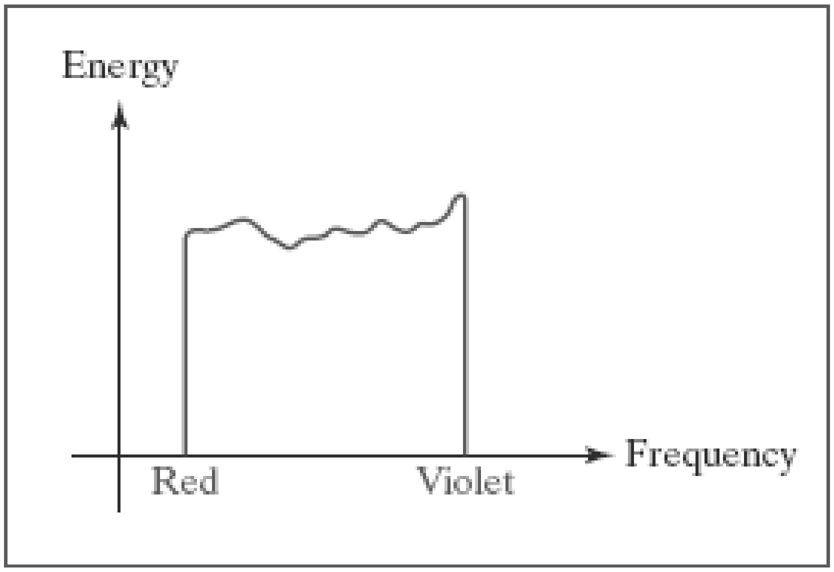
$$c = \lambda f$$

- ✓ A light source such as the sun or a standard household light bulb emits all frequencies within the visible range to produce white light.

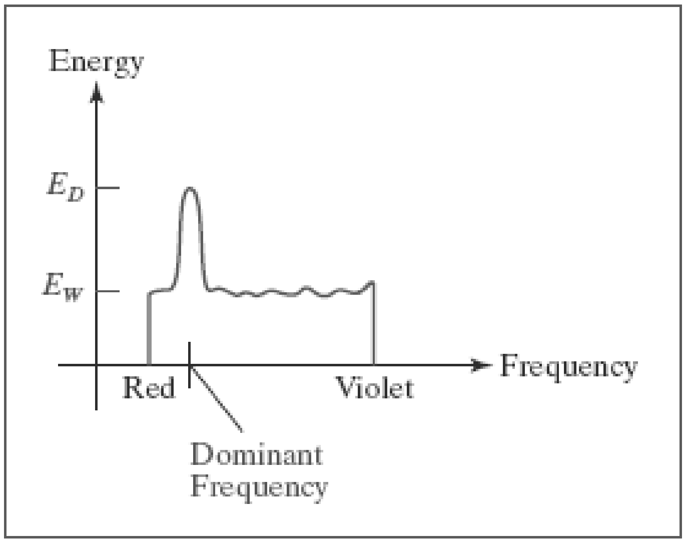
- ✓ When white light is incident upon an opaque object, some frequencies are reflected and some are absorbed.
- ✓ If low frequencies are predominant in the reflected light, the object is described as red. In this case, we say that the perceived light has a **dominant frequency** (or **dominant wavelength**) at the red end of the spectrum.
- ✓ The dominant frequency is also called the **hue,** or simply the **color,** of the light.

Psychological Characteristics of Color

- ➢ Other properties besides frequency are needed to characterize our perception of Light
- ➢ **Brightness:** which corresponds to the total light energy and can be quantified as the luminance of the light.
- ➢ **Purity**, or the **saturation** of the light: Purity describes how close a light appears to be to a pure spectral color, such as red.
- ➢ **chromaticity**, is used to refer collectively to the two properties describing color characteristics: purity and dominant frequency (hue).
- ➢ We can calculate the brightness of the source as the area under the curve, which gives the total energy density emitted.
- ➢ Purity (saturation) depends on the difference between *ED* and *EW*
- ➢ Below figure shows *Energy distribution for a white light source*



- Below figure shows, Energy distribution for a light source with a dominant frequency near the red end of the frequency range.



3.3.7 Color Models

- Any method for explaining the properties or behavior of color within some particular context is called a **color model.**

Primary Colors

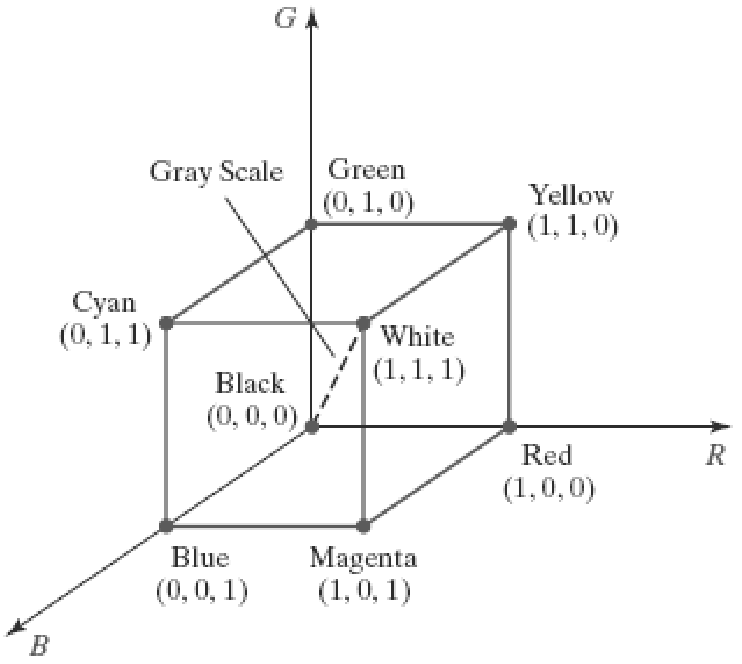
- The hues that we choose for the sources are called the **primary colors,** and the **color gamut** for the model is the set of all colors that we can produce from the primary colors.
- Two primaries that produce white are referred to as **complementary colors.**
- Examples of complementary color pairs are red and cyan, green and magenta, and blue and yellow

Intuitive Color Concepts

- An artist creates a color painting by mixing color pigments with white and black pigments to form the various shades, tints, and tones in the scene.
- Starting with the pigment for a “pure color” (“pure hue”), the artist adds a black pigment to produce different **shades** of that color.
- **Tones** of the color are produced by adding both black and white pigments.

<u>The RGB Color Model</u>

- ❖ According to the *tristimulus theory* of vision, our eyes perceive color through the stimulation of three visual pigments in the cones of the retina.
- ❖ One of the pigments is most sensitive to light with a wavelength of about 630 nm (red), another has its peak sensitivity at about 530 nm (green), and the third pigment is most receptive to light with a wavelength of about 450 nm (blue).
- ❖ The three primaries red, green, and blue, which is referred to as the *RGB color model.*
- ❖ We can represent this model using the unit cube defined on *R*, *G*, and *B* axes, as shown in Figure



- ❖ The origin represents black and the diagonally opposite vertex, with coordinates (1, 1, 1), is white the RGB color scheme is an additive model.
- ❖ Each color point within the unit cube can be represented as a weighted vector sum of the primary colors, using unit vectors **R**, **G**, and **B**:

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

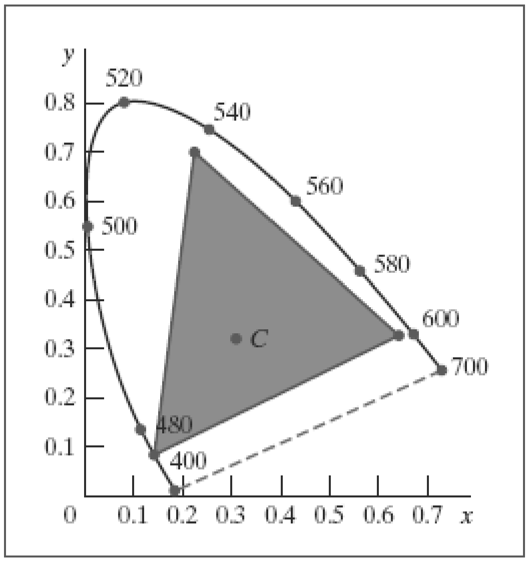
where parameters *R*, *G*, and *B* are assigned values in the range from 0 to 1.0

- ❖ Chromaticity coordinates for the National Television System Committee (NTSC) standard RGB phosphors are listed in Table

RGB (*x*, *y*) Chromaticity Coordinates

	NTSC Standard	CIE Model	Approx. Color Monitor Values
R	(0.670, 0.330)	(0.735, 0.265)	(0.628, 0.346)
G	(0.210, 0.710)	(0.274, 0.717)	(0.268, 0.588)
B	(0.140, 0.080)	(0.167, 0.009)	(0.150, 0.070)

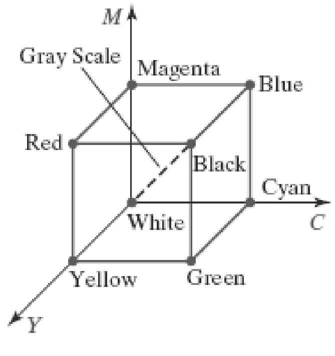
- ❖ Below figure shows the approximate color gamut for the NTSC standard RGB primaries



The CMY and CMYK Color Models

The CMY Parameters

- ❖ A subtractive color model can be formed with the three primary colors cyan, magenta, and yellow
- ❖ A unit cube representation for the CMY model is illustrated in Figure



- ❖ In the CMY model, the spatial position (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 shades of gray 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.
- ❖ Similarly, a combination of cyan and yellow ink produces green light, and a combination of magenta and yellow ink yields red light.
- ❖ The CMY printing process often uses a collection of four ink dots, which are arranged in a close pattern somewhat as an RGB monitor uses three phosphor dots.
- ❖ Thus, in practice, the CMY color model is referred to as the CMYK model, where *K* is the black color parameter.
- ❖ One ink dot is used for each of the primary colors (cyan, magenta, and yellow), and one ink dot is black

<u>Transformations Between CMY and RGB Color Spaces</u>

- ❖ We can express the conversion from an RGB representation to a CMY representation using the following matrix transformation:

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

- ❖ Where the white point in RGB space is represented as the unit column vector.
- ❖ And we convert from a CMY color representation to an RGB representation using the matrix transformation

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