Viewing Coordinate Clipping Window:

- A general approach to the two-dimensional viewing transformation is to set up a viewing-coordinate system within the world-coordinate frame.
- This viewing frame provides a reference for specifying a rectangular clipping window with any selected orientation and position as shown in the following figure.

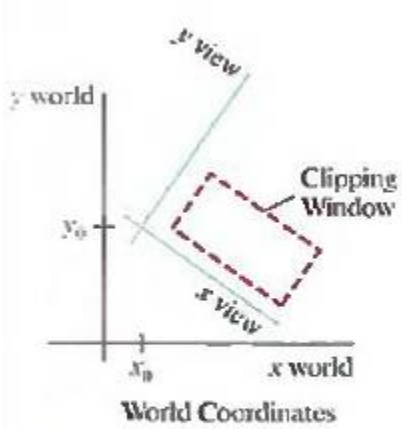


FIGURE 6-4 A rotated clipping window defined in viewing coordinates.

- First, a viewing-coordinate origin is selected at some world position: $P_o = (x_o, y_o)$.
- Then we need to establish the orientation, or rotation, of this reference frame. One way to do this is to specify a world vector V that defines the viewing y_{view} direction. Vector V is called **the view up vector.**
- 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 viewing y_{view} and x_{view} axes, respectively. These unit vectors are used to form the first and second rows of the rotation matrix R that aligns the viewing $x_{view}y_{view}$ axes with the world x_wy_w axes.
- The composite two-dimensional transformation to convert world coordinates to viewing coordinate is

$$\mathbf{M_{WC,VC} = R.T}$$

where T is the translation matrix that takes the viewing origin point Po to the world origin, and R is the rotation matrix that rotates the viewing frame of reference into coincidence with the world-coordinate system.

Figure 6-4
A viewing-coordinate frame is moved into coincidence with the world frame in two steps: (a) translate the viewing origin to the world origin, then (b) rotate to align the axes of the two systems.

World Coordinate Clipping Window:

- A routine for defining a standard, rectangular clipping window in world coordinates is 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.

Normalization and Viewport Transformations:

Mapping the Clipping Window into a Normalized Viewport:

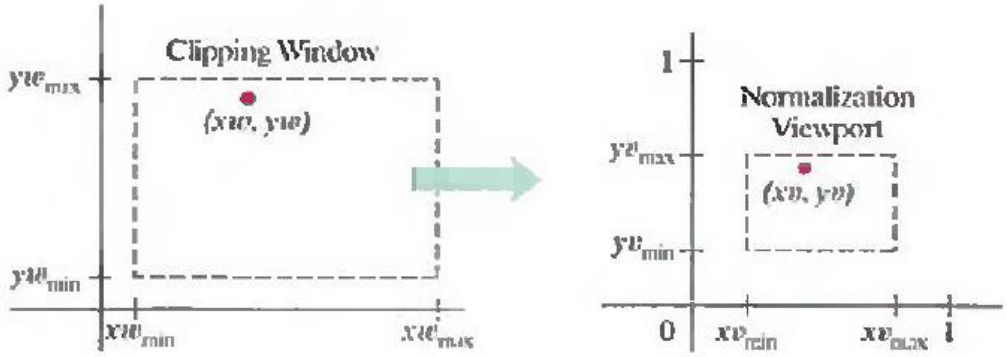


FIGURE 6-7 A point (xw, yw) in a world-coordinate clipping window is mapped to viewport coordinates (xv, yv), within a unit square, so that the relative positions of the two points in their respective rectangles are the same.

The above figure illustrates the window-to-viewport mapping. A point at position (x*w*,y*w*) in the window is mapped into position (x*v*, y*v*) in the associated viewport.
To maintain the same relative placement in the viewport as in the window, we require that

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

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

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

$$xv = s_x xw + t_x$$
$$yv = s_y yw + t_y$$

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

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

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

Since we are simply mapping world-coordinate positions into a viewport that is positioned near the world origin, we can also perform any transformation sequence that converts the rectangle for the clipping window into the viewport rectangle.
For example, we could obtain the transformation from world coordinates to viewport coordinates with the sequence

1. Scale the clipping window to the size of the viewport using s 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 representation for the transformation to the normalized viewport is

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

- The window-to-viewport transformation maintains the relative placement of object descriptions.
- An object inside the clipping window is mapped to a corresponding position inside the viewport. Similarly, an object outside the clipping window is outside the viewport.
- Relative proportions of objects are maintained only if the aspect ratio of the viewport is the same as the aspect ratio of the clipping window.

Mapping the Clipping Window into a Normalized Square:

In this approach of two dimensional viewing, we 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 the following figure:

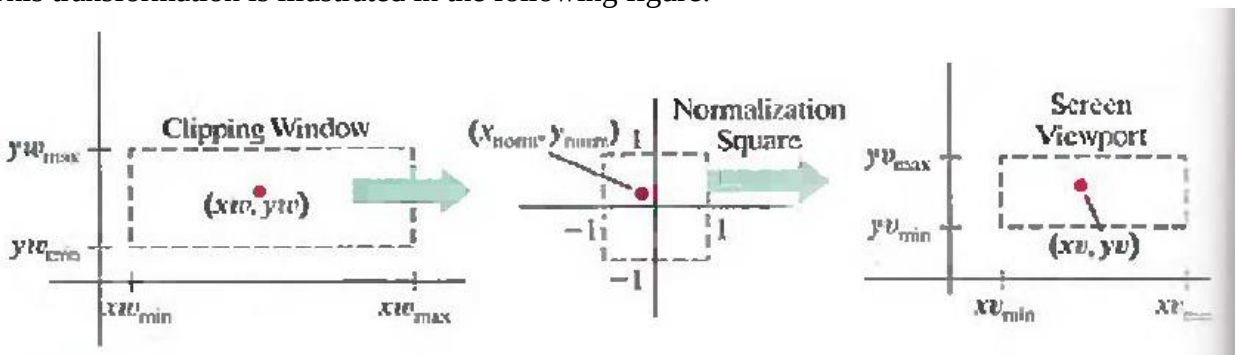


FIGURE 6-8 A point (xw, yw) in a clipping window is mapped to a normalized coordinate position (x_{norm}, y_{norm}), then to a screen-coordinate position (xv, yv) in a viewport. Objects are clipped against the normalization square before the transformation to viewport coordinates.

We transfer the contents of the clipping window into the normalization square using the same procedures as in the window-to-viewport transformation.

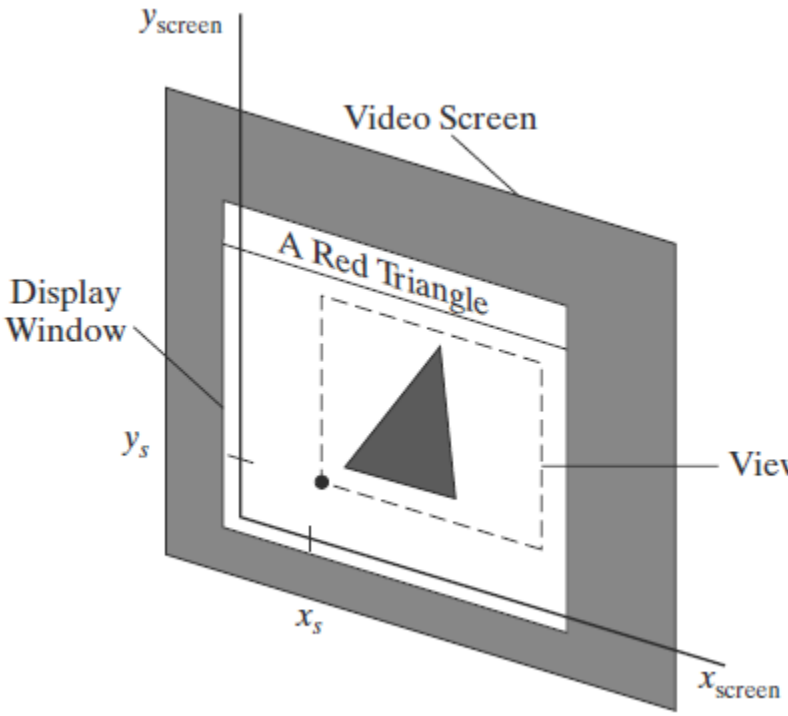
The matrix for the normalization transformation is given as given below, by substituting -1 for xv_{min} and yv_{min} and substituting +1 for xv_{max} and yv_{max}.

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

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

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

The last step in viewing process is to position the viewport area in the display window. Typically, the lower-left corner of the viewport is placed at the coordinate position specified relative to the lower-left corner of the display window.



Display of Character Strings:

- Character strings can be handled in two ways when they are mapped to a viewport.
- The simplest mapping maintains a constant character size. This method could be employed with bitmap character patterns.
- But, outline onts could be transformed the same as other primitives; we just need to transform the defining positions for the line segments in the outline character shapes.

Split-Screen Effects and Multiple Output Devices:

- 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 screen.
- We can position these views in different parts of a single display window or in multiple display windows on the screen.
- In a design application, for example, we can display a wire-frame view of an object in one viewport, while also displaying a fully rendered view of the object in another viewport.
- It is also possible that two or more output devices could be operating con-currently 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.**

OpenGL Two-Dimensional Viewing Functions:

- Actually, the basic OpenGL library has no functions specifically for two dimensional viewing because it is designed primarily for three-dimensional applications.
- But we can adapt the three-dimensional viewing routines to a two-dimensional scene, and the core library contains a viewport function.
- In addition, the GLU library provides a function for specifying a two-dimensional clipping window, and we have GLUT library functions for handling display windows. Therefore, we can use these two-dimensional routines, along with the OpenGL viewport function, for all the viewing operations we need.

OpenGL Projection Mode

- Before we select a clipping window and a viewport in OpenGL, we need to establish the appropriate mode for constructing the matrix to transform from world coordinates to screen coordinates.
- With OpenGL, we cannot set up a separate two-dimensional viewing-coordinate system and we
- must set the parameters for the clipping window as part of the projection transformation. Therefore, we must first select the projection mode. We do this with the same function we used to set the modelview mode for the geometric transformations.
- Subsequent commands for defining a clipping window and viewport will then be applied to the projection matrix.

  glMatrixMode (GL_PROJECTION);
- This designates the projection matrix as the current matrix, which is originally set to the identity matrix.
- However, if we are going to loop back through this statement for other views of a scene, we can also set the initialization as

  glLoadIdentity ( );
- This ensures that each time we enter the projection mode, the matrix will be reset to the identity matrix so that the new viewing parameters are not combined with the previous ones.

GLU Clipping-Window Function:

To define a two-dimensional clipping window, we can use the GLU function:

gluOrtho2D (xwmin, xwmax, ywmin, ywmax);

- Coordinate positions for the clipping-window boundaries are given as double precision numbers. This function specifies an orthogonal projection for mapping the scene to the screen.
- For a three-dimensional scene, this means that objects would be projected along parallel lines that are perpendicular to the two-dimensional xy display screen.
- But for a two-dimensional application, objects are already defined in the xy plane. Therefore, the orthogonal projection has no effect on our two-dimensional scene other than to convert object positions to normalized coordinates.

- Nevertheless, we must specify the orthogonal projection because our two-dimensional scene is processed through the full three dimensional OpenGL viewing pipeline.
- In fact, we could specify the clipping window using the three-dimensional OpenGL core-library version of the **gluOrtho2D** function.
- Normalized coordinates in the range from −1 to 1 are used in the OpenGL clipping routines. And the gluOrtho2D function sets up a three-dimensional version of transformation matrix for mapping objects within the clipping window to normalized coordinates.
- Objects outside the normalized square (and outside the clipping window) are eliminated from the scene to be displayed.
- If we do not specify a clipping window in an application program, the default coordinates are (xwmin, ywmin) = (−1.0, −1.0) and (xwmax, ywmax) = (1.0, 1.0).
- Thus the default clipping window is the normalized square centered on the coordinate origin with a side length of 2.0.

OpenGL Viewport Function:

We specify the viewport parameters with the OpenGL function

glViewport (xvmin, yvmin, vpWidth, vpHeight);

where all parameter values are given in integer screen coordinates relative to the display window.

- Parameters **xvmin** and **yvmin** specify the position of the lowerleft corner of the viewport relative to the lower-left corner of the display window, and the pixel width and height of the viewport are set with parameters **vpWidth and vpHeight.**
- If we do not invoke the glViewport function in a program, the default viewport size and position are the same as the size and position of the display window.
- After the clipping routines have been applied, positions within the normalized square are transformed into the viewport rectangle.
- Coordinates for the upper-right corner of the viewport are calculated for this transformation matrix in terms of the viewport width and height:

xvmax = xvmin + vpWidth, yvmax = yvmin + vpHeight

- For the final transformation, pixel colors for the primitives within the viewport are loaded into the refresh buffer at the specified screen locations.
- Multiple viewports can be created in OpenGL for a variety of applications
- We can obtain the parameters for the currently active viewport using the query function

glGetIntegerv (GL_VIEWPORT, vpArray);

where vpArray is a single-subscript, four-element array.

- This Get function returns the parameters for the current viewport to vpArray in the order xvmin, yvmin, vpWidth, and vpHeight.
- In an interactive application, for example, we can use this function to obtain parameters for the viewport that contains the screen cursor.

Creating a GLUT Display Window:

Because the GLUT library interfaces with any window-management system, we use the GLUT routines for creating and manipulating display windows so that our example programs will be independent of any specific machine.

To access these routines, we first need to initialize GLUT with the following function:

glutInit (&argc, argv);

- Parameters for this initialization function are the same as those for the main procedure, and we can use **glutInit** to process command-line arguments.
- We have three functions in GLUT for defining a display window and choosing its dimensions and position:

  glutInitWindowPosition (xTopLeft, yTopLeft);
  glutInitWindowSize (dwWidth, dwHeight);
  glutCreateWindow ("Title of Display Window");

- The first of these functions gives the integer, screen-coordinate position for the top-left corner of the display window, relative to the top-left corner of the screen. If either coordinate is negative, the display-window position on the screen is determined by the window-management system.
- With the second function, we choose a width and height for the display window in positive integer pixel dimensions.
- If we do not use these two functions to specify a size and position, the default size is 300 by 300 and the default position is (−1, −1), which leaves the positioning of the display window to the window-management system.
- In any case, the display-window size and position specified with GLUT routines might be ignored, depending on the state of the window-management system or the other requirements currently in effect for it.
- Thus, the window system might position and size the display window differently. The third function creates the display window, with the specified size and position, and assigns a title, although the use of the title also depends on the windowing system.
- At this point, the display window is defined but not shown on the screen until all the GLUT setup operations are complete.

Setting the GLUT Display-Window Mode and Color;

Various display-window parameters are selected with the GLUT function

glutInitDisplayMode (mode);

We use this function to choose a color mode (RGB or index) and different buffer combinations, and the selected parameters are combined with the logical or operation.

The default mode is single buffering and the RGB (or RGBA) color mode, which is the same as setting this mode with the statement

glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);

The color mode specification GLUT RGB is equivalent to GLUT RGBA.

A background color for the display window is chosen in RGB mode with the OpenGL routine

glClearColor (red, green, blue, alpha);

In color-index mode, we set the display-window color with

glClearIndex (index);

where parameter index is assigned an integer value corresponding to a position within the color table.

GLUT Display-Window Identifier:

Multiple display windows can be created for an application, and each is assigned a positive-integer display-window identifier, starting with the value 1 for the first window that is created. At the time that we initiate a display window, we can record its identifier with the statement

windowID = glutCreateWindow ("A Display Window");

Once we have saved the integer display-window identifier in variable name windowID, we can use the identifier number to change display parameters or to delete the display window.

Deleting a GLUT Display Window:

The GLUT library also includes a function for deleting a display window that we have created. If we know the display window's identifier, we can eliminate it with the statement

glutDestroyWindow (windowID);

Current GLUT Display Window:

When we specify any display-window operation, it is applied to the current display window, which is either the last display window that we created or the one we select with the following command:

glutSetWindow (windowID);

In addition, at any time, we can query the system to determine which window is the current display window:

currentWindowID = glutGetWindow ( );

A value of 0 is returned by this function if there are no display windows or if the current display window was destroyed.

Relocating and Resizing a GLUT Display Window:

We can reset the screen location for the current display window with

glutPositionWindow (xNewTopLeft, yNewTopLeft);

where the coordinates specify the new position for the upper-left display-window corner, relative to the upper-left corner of the screen.

Similarly, the following function resets the size of the current display window:

glutReshapeWindow (dwNewWidth, dwNewHeight);

With the following command, we can expand the current display window to fill the screen:

glutFullScreen ( );

The exact size of the display window after execution of this routine depends on the window-management system.
A subsequent call to either **glutPositionWindow** or **glutReshapeWindow** will cancel the request for an expansion to full-screen size. Whenever the size of a display window is changed, its aspect ratio may

glutReshapeFunc (winReshapeFcn);

This GLUT routine is activated when the size of a display window is changed, and the new width and height are passed to its argument: the function winReshapeFcn, in this example.
Thus, winReshapeFcn is the "callback function" for the "reshape event." We can then use this callback function to change the parameters for the viewport so that the original aspect ratio of the scene is maintained.
In addition, we could also reset the clipping-window boundaries, change the display-window color, adjust other viewing parameters, and perform any other tasks.

Managing Multiple GLUT Display Windows:

The GLUT library also has a number of routines for manipulating a display window in various ways.
These routines are particularly useful when we have multiple display windows on the screen and we want to rearrange them or locate a particular display window.
We use the following routine to convert the current display window to an icon in the form of a small picture or symbol representing the window:

glutIconifyWindow ( );

The label on this icon will be the same name that we assigned to the window, but we can change this with the following command:

glutSetIconTitle ("Icon Name");

We also can change the name of the display window with a similar command:

glutSetWindowTitle ("New Window Name");

With multiple display windows open on the screen, some windows may overlap or totally obscure other display windows.
We can choose any display window to be in front of all other windows by first designating it as the current window, and then issuing the "pop-window" command:

```
glutSetWindow (windowID);
glutPopWindow ( );
```

In a similar way, we can "push" the current display window to the back so that it is behind all other display windows.
This sequence of operations is

```
glutSetWindow (windowID);
glutPushWindow ( );
```

We can also take the current window off the screen with

glutHideWindow ( );

In addition, we can return a "hidden" display window, or one that has been converted to an icon, by designating it as the current display window and then invoking the function

glutShowWindow ( );

GLUT Subwindows:

Within a selected display window, we can set up any number of second-level display windows, which are called subwindows.
This provides a means for partitioning display windows into different display sections.We create a subwindow with the following function:

glutCreateSubWindow (windowID, xBottomLeft, yBottomLeft, width, height);

- Parameter windowID identifies the display window in which we want to set up the subwindow.
- With the remaining parameters, we specify the subwindow's size and the placement of its lower-left corner relative to the lower-left corner of the display window.
- Subwindows are assigned a positive integer identifier in the same way that first-level display windows are numbered, and we can place a subwindow inside another subwindow.
- Also, each subwindow can be assigned an individual display mode and other parameters. We can even reshape, reposition, push, pop, hide, and show subwindows, just as we can with first-level display windows.
- But we cannot convert a GLUT subwindow to an icon.

Selecting a Display-Window Screen-Cursor Shape:

We can use the following GLUT routine to request a shape for the screen cursor that is to be used with the current window:

glutSetCursor (shape);

- The possible cursor shapes that we can select are an arrow pointing in a chosen direction, a bidirectional arrow, a rotating arrow, a crosshair, a wristwatch, a question mark, or even a skull and crossbones.
- For example, we can assign the symbolic constant GLUT_CURSOR_UP_DOWN to parameter shape to obtain an up-down arrow.
- A rotating arrow is chosen with GLUT_CURSOR_CYCLE, a wristwatch shape is selected with GLUT_CURSOR_WAIT, and a skull and crossbones is obtained with the constant GLUT_CURSOR_DESTROY.
- A cursor shape can be assigned to a display window to indicate a particular kind of application, such as an animation.
- However, the exact shapes that we can use are system dependent.

Viewing Graphics Objects in a GLUT Display Window:

- After we have created a display window and selected its position, size, color, and other characteristics, we indicate what is to be shown in that window.
- If more than one display window has been created, we first designate the one we want as the current display window.

Then we invoke the following function to assign something to that window:

glutDisplayFunc (pictureDescrip);

- The argument is a routine that describes what is to be displayed in the current window.
- This routine, called pictureDescrip for this example, is referred to as a callback function because it is the routine that is to be executed whenever GLUT determines that the display-window contents should be renewed.
- Routine pictureDescrip usually contains the OpenGL primitives and attributes that define a picture, although it could specify other constructs such as a menu display.
- If we have set up multiple display windows, then we repeat this process for each of the display windows or subwindows.
- Also, we may need to call **glutDisplayFunc** after the **glutPopWindow** command if the display window has been damaged during the process of redisplaying the windows.
- In this case, the following function is used to indicate that the contents of the current display window should be renewed:
  glutPostRedisplay ( );

This routine is also used when an additional object, such as a pop-up menu, is to be shown in a display window.

Executing the Application Program

When the program setup is complete and the display windows have been created and initialized, we need to issue the final GLUT command that signals execution of the program:

glutMainLoop ( );

At this time, display windows and their graphic contents are sent to the screen.
The program also enters the GLUT processing loop that continually checks for new "events," such as interactive input from a mouse or a graphics tablet.

Other GLUT Functions:

Sometimes it is convenient to designate a function that is to be executed when there are no other events for the system to process.
We can do that with

glutIdleFunc (function);

The parameter for this GLUT routine could reference a background function or a procedure to update parameters for an animation when no other processes are taking place.
Finally, we can use the following function to query the system about some of the current state parameters:

glutGet (stateParam);

This function returns an integer value corresponding to the symbolic constant we select for its argument.
For example, we can obtain the x-coordinate position for the top-left corner of the current display window, relative to the top-left corner of the screen, with the constant GLUT WINDOW X; and we can retrieve the current display-window width or the screen width with GLUT WINDOW WIDTH or GLUT SCREEN WIDTH.

OpenGL Two-Dimensional Viewing Program Example:

```
#include <GL/glut.h>
class wcPt2D {
public:
GLfloat x, y;
};

void init (void)
{
        /* Set color of display window to white. */
        glClearColor (1.0, 1.0, 1.0, 0.0);
        /* Set parameters for world-coordinate clipping window. */
        glMatrixMode (GL_PROJECTION);
        gluOrtho2D (-100.0, 100.0, -100.0, 100.0);
        /* Set mode for constructing geometric transformation matrix. */
        glMatrixMode (GL_MODELVIEW);
}

void triangle (wcPt2D *verts)
{
        GLint k;
        glBegin (GL_TRIANGLES);
        for (k = 0; k < 3; k++)
        glVertex2f (verts [k].x, verts [k].y);
        glEnd ( );
}

void displayFcn (void)
{
        /* Define initial position for triangle. */
        wcPt2D verts [3] = { {-50.0, -25.0}, {50.0, -25.0}, {0.0, 50.0} };
        glClear (GL_COLOR_BUFFER_BIT); // Clear display window.
        glColor3f (0.0, 0.0, 1.0); // Set fill color to blue.
        glViewport (0, 0, 300, 300); // Set left viewport.
        triangle (verts); // Display triangle.
        /* Rotate triangle and display in right half of display window. */
        glColor3f (1.0, 0.0, 0.0); // Set fill color to red.
        glViewport (300, 0, 300, 300); // Set right viewport.
```

```
        glRotatef (90.0, 0.0, 0.0, 1.0); // Rotate about z axis.
        triangle (verts); // Display red rotated triangle.
        glFlush ( );
}

void main (int argc, char ** argv)
{
        glutInit (&argc, argv);
        glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
        glutInitWindowPosition (50, 50);
        glutInitWindowSize (600, 300);
        glutCreateWindow ("Split-Screen Example");
        init ( );
        glutDisplayFunc (displayFcn);
        glutMainLoop ( );
}
```

Clipping Algorithms:

- Generally, 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.
- Usually a clipping region is a rectangle in standard position, although we could use any shape for a clipping application.
- 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.
- Clipping methods are also used to antialias object boundaries, to construct objects using solid-modeling methods, to manage a multi-window environment, and to allow parts of a picture to be moved, copied, or erased in drawing and painting programs.
- Clipping algorithms are applied in two-dimensional viewing procedures to identify those parts of a picture that are within the clipping window. Everything outside the clipping window is then eliminated from the scene description that is transferred to the output device for display.
- An efficient implementation of clipping in the viewing pipeline is to apply the algorithms to the normalized boundaries of the clipping window.
- This reduces calculations, because all geometric and viewing transformation matrices can be concatenated and applied to a scene description before clipping is carried out.
- The clipped scene can then be transferred to screen coordinates for final processing.

In the following sections, we explore two-dimensional algorithms for

- Point clipping
- Line clipping (straight-line segments)
- Fill-area clipping (polygons)
- Curve clipping
- Text clipping

- Point, line, and polygon clipping are standard components of graphics packages. But similar methods can be applied to other objects, particularly conics, such as circles, ellipses, and spheres, in addition to spline curves and surfaces.
- Usually, however, objects with nonlinear boundaries are approximated with straight-line segments or polygon surfaces to reduce computations.
- Unless otherwise stated, we assume that the clipping region is a rectangular window in standard position, with boundary edges at coordinate positions xwmin, xwmax, ywmin, and ywmax.
- These boundary edges typically correspond to a normalized square, in which the x and y values range either from 0 to 1 or from −1 to 1.

Two-Dimensional Point Clipping:

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

xwmin ≤ x ≤ xwmax
ywmin ≤ y ≤ ywmax

If any of these four inequalities is not satisfied, the point is clipped (not saved for display).
Although point clipping is applied less often than line or polygon clipping, it is useful in various situations, particularly when pictures are modeled with particle systems.
For example, point clipping can be applied to scenes involving clouds, sea foam, smoke, or explosions that are modeled with "particles," such as the center coordinates for small circles or spheres.

Two-Dimensional Line Clipping:

Figure 9 illustrates possible positions for straight-line segments in relationship to a standard clipping window.

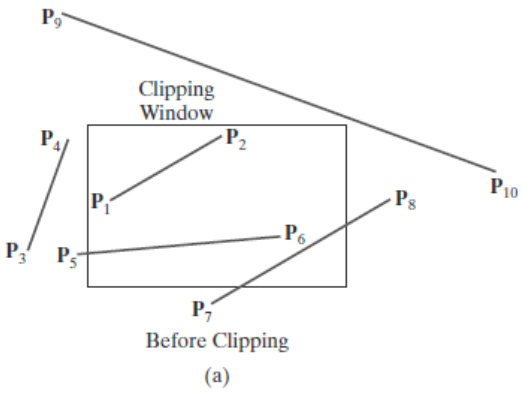


FIGURE 9
Clipping straight-line segments using a standard rectangular clipping window.

- 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.
- If we are unable to identify a line as completely inside or completely outside a clipping rectangle, we must then perform intersection calculations to determine whether any part of the line crosses the window interior.

- We test a line segment to determine if it is completely inside or outside a selected clipping-window edge by applying the point-clipping tests of the previous section.
- When both endpoints of a line segment are inside all four clipping boundaries, such as the line from**P**1 to **P**2 in Figure 9, the line is completely inside the clipping window and we save it.
- And when both endpoints of a line segment are outside any one of the four boundaries (as with line **P**3**P**4 in Figure 9), that line is completely outside the window and it is eliminated from the scene description.
- But if both these tests fail, the line segment intersects at least one clipping boundary and it may or may not cross into the interior of the clipping window.
- One way to formulate the equation for a straight-line segment is to use the following parametric representation, where the coordinate positions ($x0$, $y0$) and ($x$end, $y$end) designate the two line endpoints:

$$x = x_0 + u(x_{end} - x_0)$$
$$y = y_0 + u(y_{end} - y_0) \quad 0 \le u \le 1$$

- We can use this parametric representation to determine where a line segment crosses each clipping-window edge by assigning the coordinate value for that edge to either x or y and solving for parameter u.
- For example, the left window boundary is at position x_{wmin}, so we substitute this value for x, solve for u, and calculate the corresponding y-intersection value.
- If this value of u is outside the range from 0 to 1, the line segment does not intersect that window border line.
- However, if the value of u is within the range from 0 to 1, part of the line is inside that border.
- We can then process this inside portion of the line segment against the other clipping boundaries until either we have clipped the entire line or we find a section that is inside the window.
- Processing line segments in a scene using the simple clipping approach described in the preceding paragraph is straightforward, but not very efficient.
- It is possible to reformulate the initial testing and the intersection calculations to reduce processing time for a set of line segments, and a number of faster line clippers have been developed.
- Some of the algorithms are designed explicitly for two-dimensional pictures and some are easily adapted to sets of three-dimensional line segments.

Cohen-Sutherland Line Clipping:

- This is one of the earliest algorithms to be developed for fast line clipping, and variations of this method are widely used.
- 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.

- We can reference the window edges in any order, and Figure 10 illustrates one possible ordering with the bit positions numbered 1 through 4 from right to left.

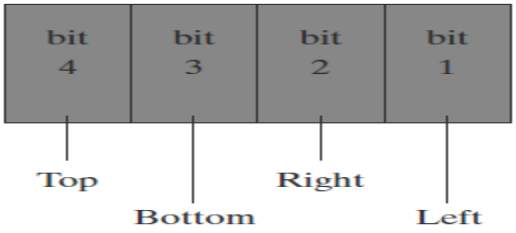


FIGURE 10
A possible ordering for the clipping-window boundaries corresponding to the bit positions in the Cohen-Sutherland endpoint region code.

- 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
- Each clipping-window edge divides two-dimensional space into an inside half space and an outside half space.
- Together, the four window borders create nine regions, and Figure 11 lists the value for the binary code in each of these regions.

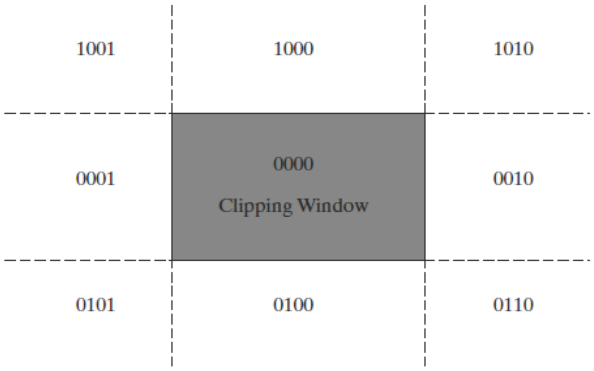


FIGURE 11
The nine binary region codes for identifying the position of a line endpoint, relative to the clipping-window boundaries.

- Thus, an endpoint that is below and to the left of the clipping window is assigned the region code 0101, and the region-code value for any endpoint inside the clipping window is 0000.
- 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 $\mathbf{x} < \mathbf{xw}_{\mathbf{min}}$, and the other three bit values are determined similarly. Instead of using inequality testing, we can determine the values for a region-code more efficiently using bit-processing operations and the following two steps:

(1) Calculate differences between endpoint coordinates and clipping boundaries.
(2) Use the resultant sign bit of each difference calculation to set the corresponding value in the region code.

- For the ordering scheme shown in Figure 10, bit 1 is the sign bit of $\mathbf{x - xw_{min}}$; bit 2 is the sign bit of $\mathbf{xw_{max} - x}$; bit 3 is the sign bit of $\mathbf{y - yw_{min}}$; and bit 4 is the sign bit of $\mathbf{yw_{max} - y}$.
- Once we have established region codes for all line endpoints, we can quickly determine which lines are completely inside the clip window and which are completely outside.
- Any lines that are completely contained within the window edges have a region code of 0000 for both endpoints, and we save these line segments.
- Any line that has a region-code value of 1 in the same bit position for each endpoint is ***completely outside the clipping rectangle,*** and we eliminate that line segment.
- As an example, a line that has a region code of 1001 for one endpoint and a code of 0101 for the other endpoint is completely to the left of the clipping window, as indicated by the value of 1 in the first bit position of each region code.
- We can perform the inside-outside tests for line segments using logical operators.
- When the or operation between two endpoint region codes for a line segment is false (0000), the line is inside the clipping window.
- Therefore, we save the line and proceed to test the next line in the scene description. When the and operation between the two endpoint region codes for a line is true (not 0000), the line is completely outside the clipping window, and we can eliminate it from the scene description.
- Lines that cannot be identified as being completely inside or completely outside a clipping window by the region-code tests are next checked for intersection with the window border lines.
- As shown in Figure 12, line segments can intersect clipping boundary lines without entering the interior of the window.

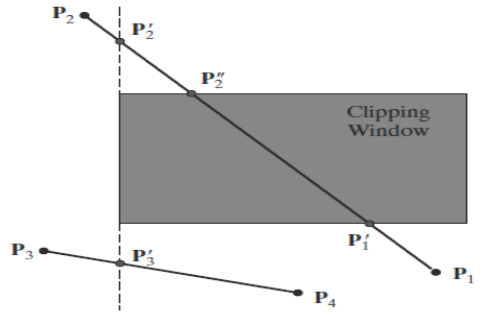


FIGURE 12
Lines extending from one clipping-window region to another may cross into the clipping window, or they could intersect one or more clipping boundaries without entering the window.

- Therefore, several intersection calculations might be necessary to clip a line segment, depending on the order in which we process the clipping boundaries.
- As we process each clipping-window edge, a section of the line is clipped, and the remaining part of the line is checked against the other window borders.
- We continue eliminating sections until either the line is totally clipped or the remaining part of the line is inside the clipping window.
- Let us assume that the window edges are processed in the following order: left, right, bottom, top.

- To determine whether a line crosses a selected clipping boundary, we can check corresponding bit values in the two endpoint region codes.
- If one of these bit values is 1 and the other is 0, the line segment crosses that boundary.

Figure 12 illustrates two line segments that cannot be identified immediately as completely inside or completely outside the clipping window. The region codes for the line from P_1 to P_2 are 0100 and 1001. Thus, P_1 is inside the left clipping boundary and P_2 is outside that boundary. We then calculate the intersection position $\mathbf{P}'_2$, and we clip off the line section from P_2 to $\mathbf{P}'_2$.

The remaining portion of the line is inside the right border line, and so we next check the bottom border. Endpoint P_1 is below the bottom clipping edge and $\mathbf{P}'_2$ is above it, so we determine the intersection position at this boundary ($\mathbf{P}'_1$).

We eliminate the line section from P_1 to $\mathbf{P}'_1$and proceed to the top window edge. There we determine the intersection position to be $\mathbf{P}''_2$. The final step is to clip off the section above the top boundary and save the interior segment from $\mathbf{P}'_1$ to $\mathbf{P}''_2$.

For the second line, we find that point P_3 is outside the left boundary and P_4 is inside. Thus, we calculate the intersection position P'_3 and eliminate the line section from P_3 to P'_3. By checking region codes for the endpoints $P'_3$and P_4, we find that the remainder of the line is below the clipping window and can be eliminated as well. It is possible, when clipping a line segment using this approach, to calculate an intersection position at all four clipping boundaries, depending on how the line endpoints are processed and what ordering we use for the boundaries.

Figure 13 shows the four intersection positions that could be calculated for a line segment that is processed against the clipping-window edges in the order left, right, bottom, top. Therefore, variations of this basic approach have been developed in an effort to reduce the intersection calculations.

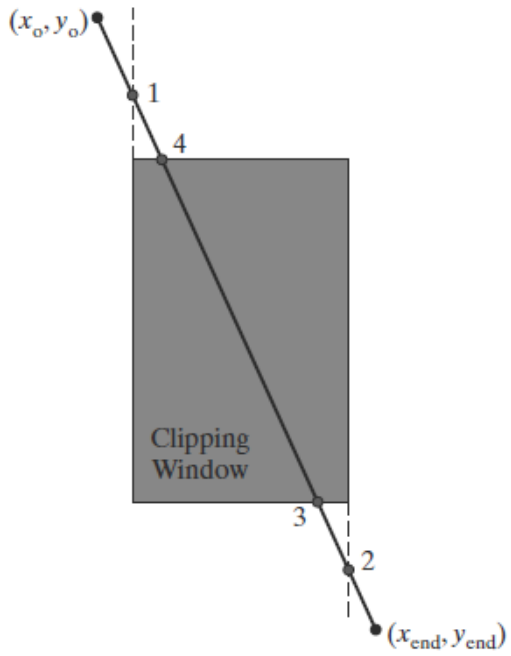


FIGURE 13
Four intersection positions (labeled from 1 to 4) for a line segment that is clipped against the window boundaries in the order left, right, bottom, top.

To determine a boundary intersection for a line segment, we can use the slope intercept 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 $m = (y_{end} - y_0)/(x_{end} - x_0)$.
Similarly, if we are looking for the intersection with a horizontal border, the x coordinate can be calculated as

$$x = x_0 + \frac{y - y_0}{m}$$

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

REFER THE LAB PROGRAM FOR IMPLEMENTATION COHENSUTHERLAND LINE CLIPPING ALGORITHM.

Polygon Fill-Area Clipping:

Graphics packages typically support only fill areas that are polygons, and often only convex polygons. 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. Instead, a line clipper would often produce a disjoint set of lines with no complete information about how we might forma closed boundary around the clipped fill area. Figure 19 illustrates a possible output from a line-clipping procedure applied to the edges of a polygon fill area.

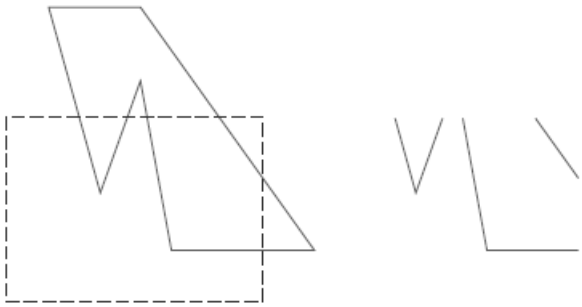


FIGURE 19
A line-clipping algorithm applied to the line segments of the polygon boundary in (a) generates the unconnected set of lines in (b).

What we require is a procedure that will output one or more closed polylines for the boundaries of the clipped fill area, so that the polygons can be scan-converted to fill the interiors with the assigned color or pattern, as in Figure 20.

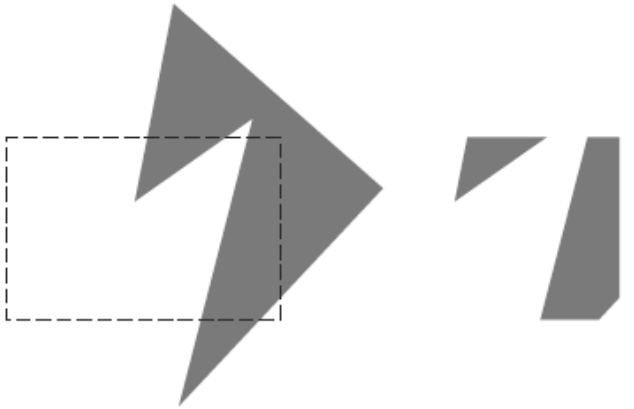


FIGURE 20
Display of a correctly clipped polygon fill area.

We can process a polygon fill area against the borders of a clipping window using the same general approach as in line clipping. A line segment is defined by its two endpoints, and these endpoints are processed through a line-clipping procedure by constructing a new set of clipped endpoints at each clipping-window boundary.

Similarly, 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 in Figure 21. Of course, the interior fill for the polygon would not be applied until the final clipped border had been determined.

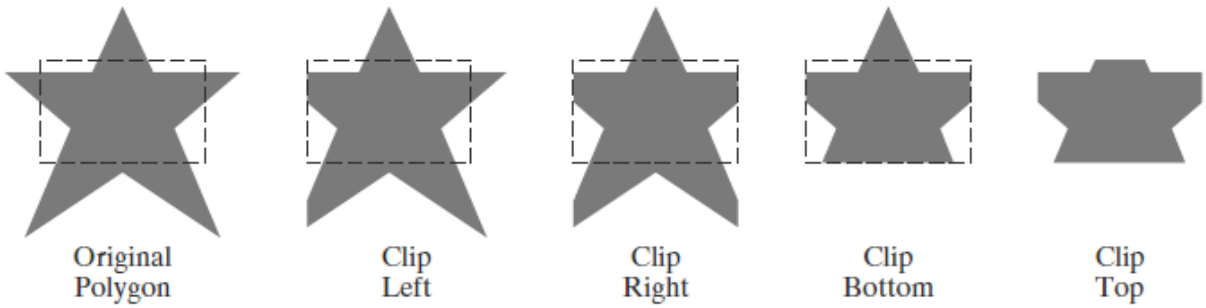


FIGURE 21
Processing a polygon fill area against successive clipping-window boundaries.

Just as we first tested a line segment to determine whether it could be completely saved or completely clipped, we can do the same with a polygon fill area by checking its coordinate extents. If the minimum and maximum coordinate values for the fill area are inside all four clipping boundaries, the fill area is saved for further processing. If these coordinate extents are all outside any of the clipping-window borders, we eliminate the polygon from the scene description (Figure 22).

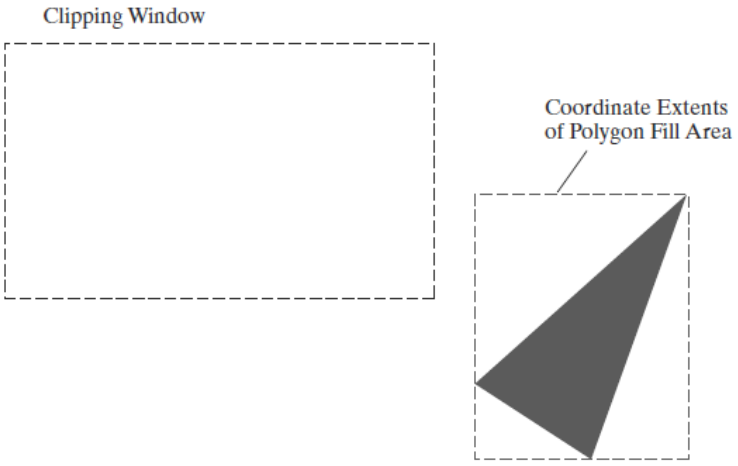


FIGURE 22
A polygon fill area with coordinate extents outside the right clipping boundary.

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 (Figure 23). For concave-polygon clipping, we would need to modify this basic approach so that multiple vertex lists could be generated.

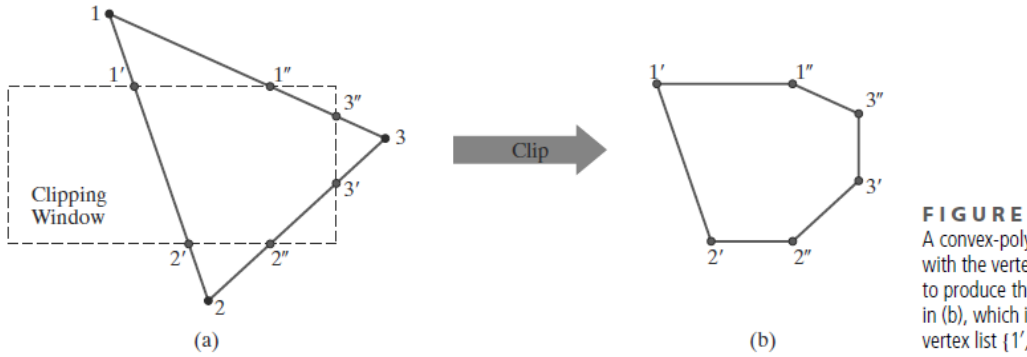


FIGURE 23
A convex-polygon fill area (a), defined with the vertex list {1, 2, 3}, is clipped to produce the fill-area shape shown in (b), which is defined with the output vertex list {1′, 2′, 2″, 3′, 3″, 1″}.

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.

This eliminates the need for an output set of vertices at each clipping stage, and it allows the boundary-clipping routines to be implemented in parallel. 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). As soon as a clipper completes the processing of one pair of vertices, the clipped coordinate values, if any, for that edge are sent to the next clipper. Then the first clipper processes the next pair of endpoints. In this way, the individual boundary clippers can be operating in parallel.

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

One possibility is that the first edge endpoint is outside the clipping boundary and the second endpoint is inside. Or, both endpoints could be inside this clipping boundary.

Another possibility is that the first endpoint is inside the clipping boundary and the second endpoint is outside. 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 24.

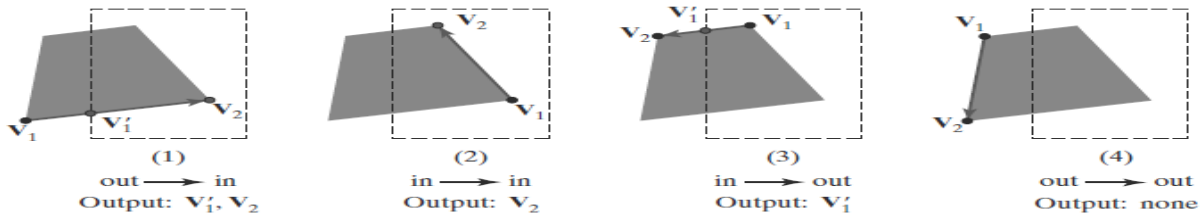


FIGURE 24
The four possible outputs generated by the left clipper, depending on the position of a pair of endpoints relative to the left boundary of the clipping window.

As each successive pair of endpoints is passed to one of the four clippers, an output is generated for the next clipper according to the results of the following tests:

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.

The last clipper in this series generates a vertex list that describes the final clipped fill area.

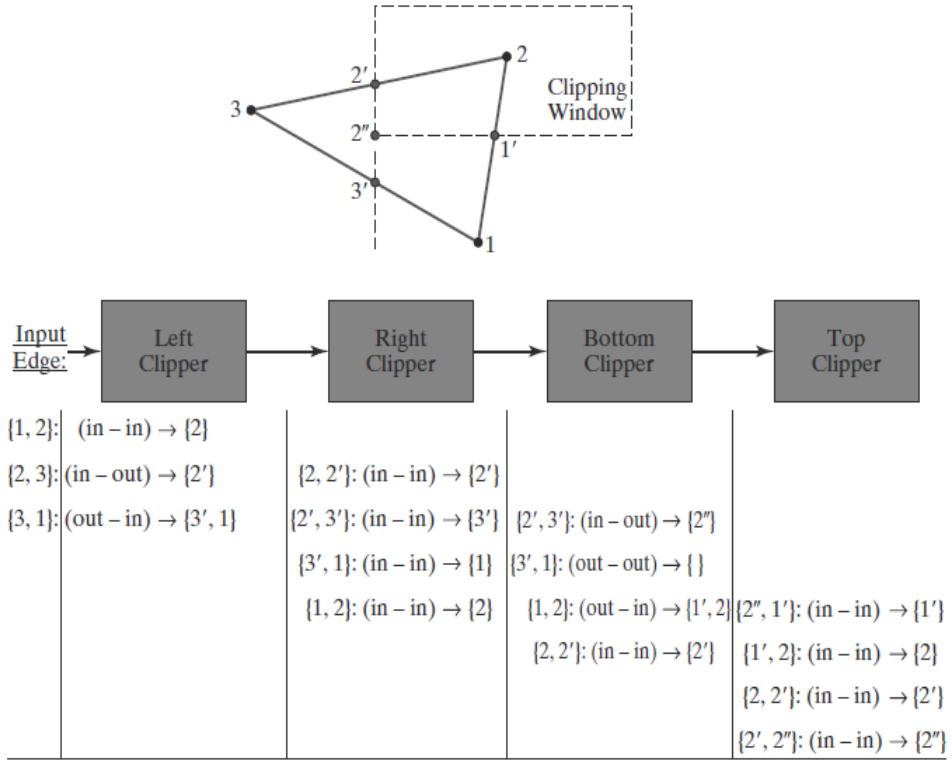


FIGURE 25
Processing a set of polygon vertices, {1, 2, 3}, through the boundary clippers using the Sutherland-Hodgman algorithm. The final set of clipped vertices is {1′, 2, 2′, 2″}.

Figure 25 provides an example of the Sutherland-Hodgman polygon clipping algorithm for a fill area defined with the vertex set {1, 2, 3}. As soon as a clipper receives a pair of endpoints, it determines the appropriate output using the tests illustrated in Figure 24. These outputs are passed in succession from the left clipper to the right, bottom, and top clippers. The output from the top clipper is the set of vertices defining the clipped fill area. For this example, the output vertex list is {**1′, 2, 2′, 2″**}.

Three-Dimensional Geometric Transformations:

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 t_x, t_y, and t_z to the Cartesian coordinates of P:

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

Figure 1 illustrates three-dimensional point translation.

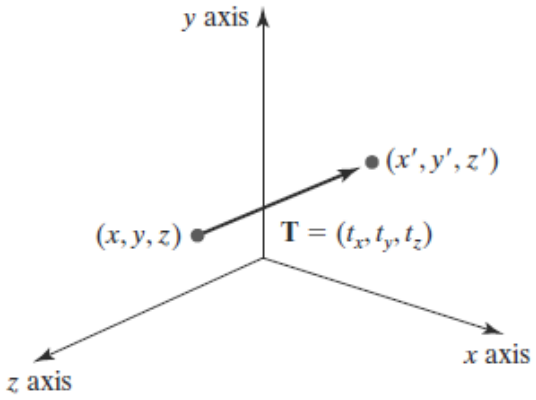


FIGURE 1
Moving a coordinate position with translation vector $\mathbf{T} = (t_x, t_y, t_z)$.

We can express these three-dimensional translation operations in matrix form. But now the coordinate positions, P and P', are represented in homogeneous coordinates with four-element column matrices, and the translation operator T is a 4 × 4 matrix:

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

or

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

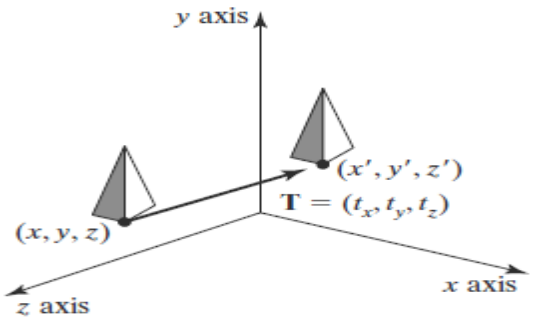


FIGURE 2
Shifting the position of a three-dimensional object using translation vector **T**.

An object is translated in three dimensions by transforming each of the defining coordinate positions for the object, then reconstructing the object at the new location. For an object represented as a set of polygon surfaces, we translate each vertex for each surface (Figure 2) and redisplay the polygon facets at the translated positions.
The following program segment illustrates construction of a translation matrix, given an input set of translation parameters.

```
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 using the same procedures that we applied in a two-dimensional translation. That is, we negate the translation distances t_x, t_y, and t_z. This produces a translation in the opposite direction, and the product of a translation matrix and its inverse is the identity matrix.

Three-Dimensional Rotation:

We can rotate an object about any axis in space, but the easiest rotation axes to handle are those that are parallel to the Cartesian-coordinate axes. Also, we can use combinations of coordinate-axis rotations (along with appropriate translations) to specify a rotation about any other line in space.

By convention, positive rotation angles produce counterclockwise rotations about a coordinate axis, assuming that we are looking in the negative direction along that coordinate axis. This agrees with our earlier discussion of rotations in two dimensions, where positive rotations in the xy plane are counterclockwise about a pivot point (an axis that is parallel to the z axis).

Three-Dimensional Coordinate-Axis Rotations:

The two-dimensional **z-axis rotation** equations are easily extended to three dimensions, as follows:

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

Parameter θ specifies the rotation angle about the z axis, and z-coordinate values are unchanged by this transformation. In homogeneous-coordinate form, the three-dimensional z-axis rotation equations are

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

which we can write more compactly as

$$\mathbf{P}' = \mathbf{R}_z(\theta) \cdot \mathbf{P}$$

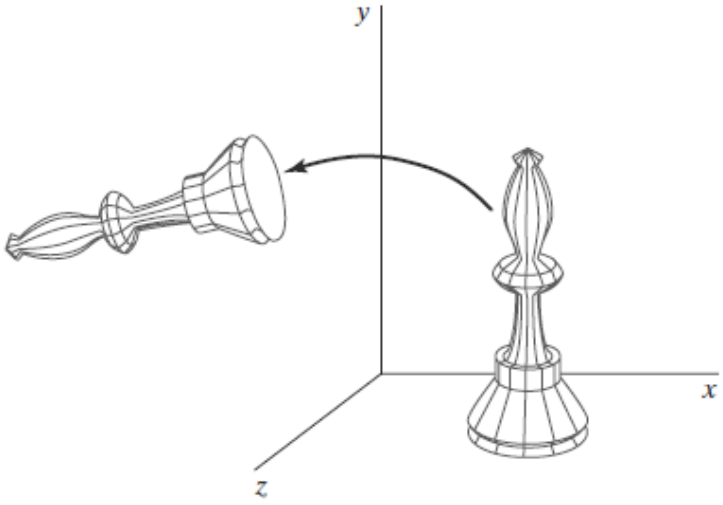


FIGURE 4
Rotation of an object about the *z* axis.

Figure 4 illustrates rotation of an object about the z axis.
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 :

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

Thus, to obtain the x-axis and y-axis rotation transformations, we cyclically replace x with y, y with z, and z with x, as illustrated in Figure 5.

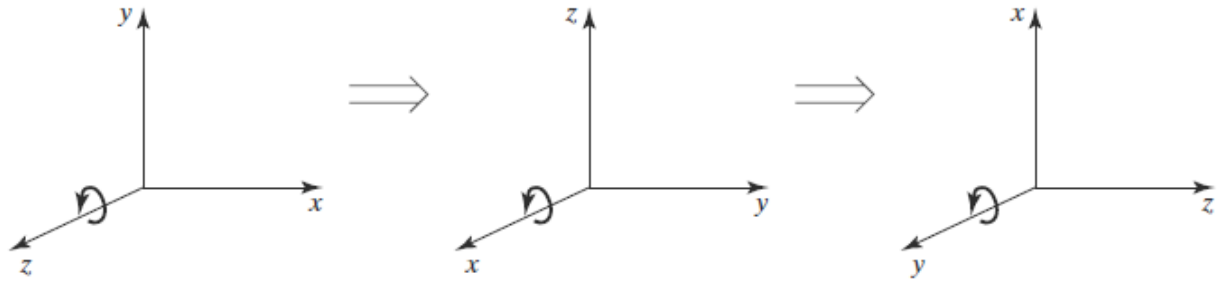


FIGURE 5
Cyclic permutation of the Cartesian-coordinate axes to produce the three sets of coordinate-axis rotation equations.

We get the equations for an **x-axis rotation**:

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

Rotation of an object around the x axis is demonstrated in Figure 6.

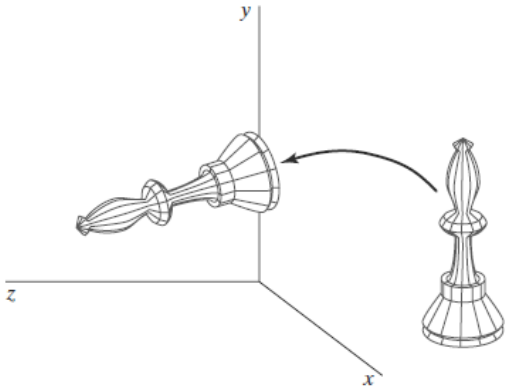


FIGURE 6
Rotation of an object about the *x* axis.

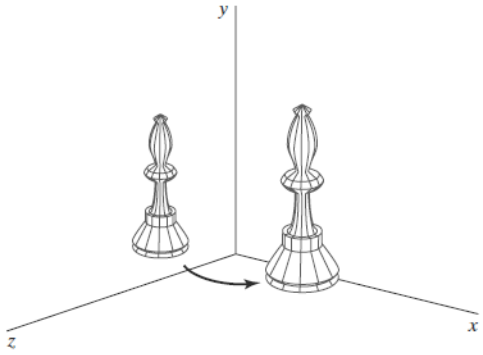


FIGURE 7
Rotation of an object about the *y* axis.

A cyclic permutation of coordinates in Equations 8 gives us the transformation equations for a **y-axis rotation:**

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

An example of y-axis rotation is shown in Figure 7.

An inverse three-dimensional rotation matrix is obtained in the same way as the inverse rotations in two dimensions. We just replace the angle θ with −θ.

Negative values for rotation angles generate rotations in a clockwise direction, and the identity matrix is produced when we multiply any rotation matrix by its inverse. Because only the sine function is affected by the change in sign of the rotation angle, the inverse matrix can also be obtained by interchanging rows and columns. That is, we can calculate the inverse of any rotation matrix R by forming its transpose ($R^{-1} = R^T$).

General Three-Dimensional Rotations:

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

We first move the designated rotation axis onto one of the coordinate axes. Then we apply the appropriate rotation matrix for that coordinate axis. The last step in the transformation sequence is to return the rotation axis to its original position.

In the special case where an object is to be rotated about an axis that is parallel to one of the coordinate axes, we attain the desired rotation with the following transformation sequence:

1. Translate the object so that the rotation axis coincides with the parallel coordinate axis.
2. Perform the specified rotation about that axis.
3. Translate the object so that the rotation axis is moved back to its original position.

The steps in this sequence are illustrated in Figure 8.

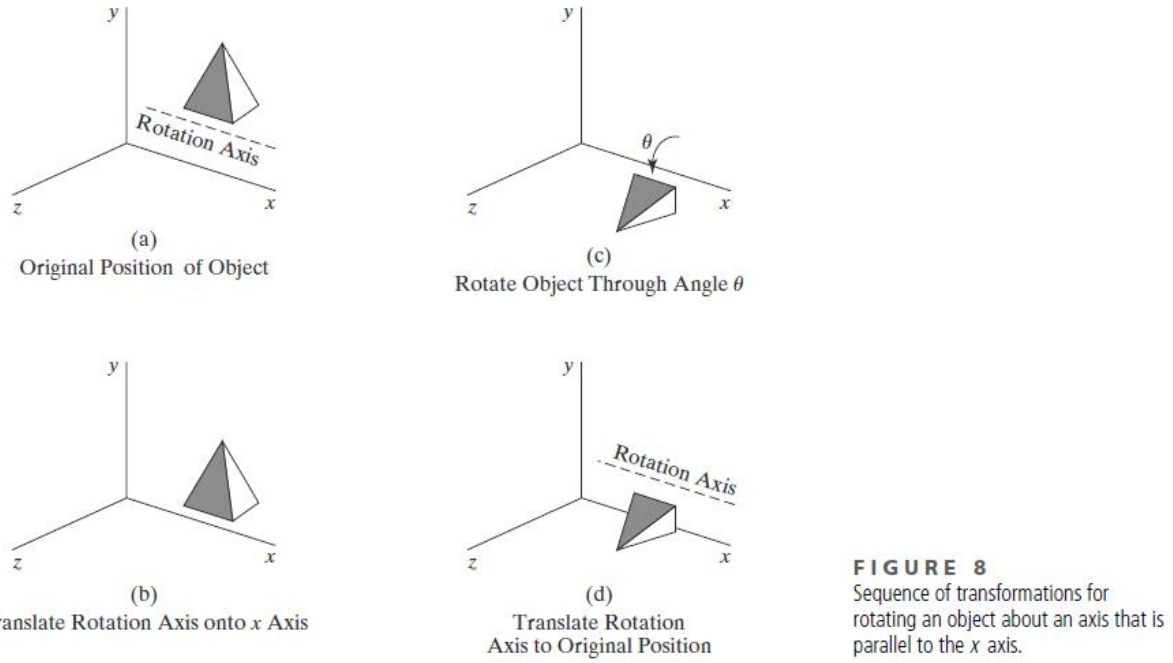


FIGURE 8
Sequence of transformations for rotating an object about an axis that is parallel to the *x* axis.

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

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

This composite matrix is of the same form as the two-dimensional transformation sequence for rotation about an axis that is parallel to the z axis (a pivot point that is not at the coordinate origin).

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. In this case, we also need rotations to align the rotation axis with a selected coordinate axis and then to bring the rotation axis back to its original orientation.

Given the specifications for the rotation axis and the rotation angle, we can accomplish the required rotation in five steps:

1. Translate the object so that the rotation axis passes through the coordinate origin.
2. Rotate the object so that the axis of rotation coincides with one of the coordinate axes.
3. Perform the specified rotation about 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.

We can transform the rotation axis onto any one of the three coordinate axes. The z axis is often a convenient choice, and we next consider a transformation sequence using the z-axis rotation matrix (Figure 9).

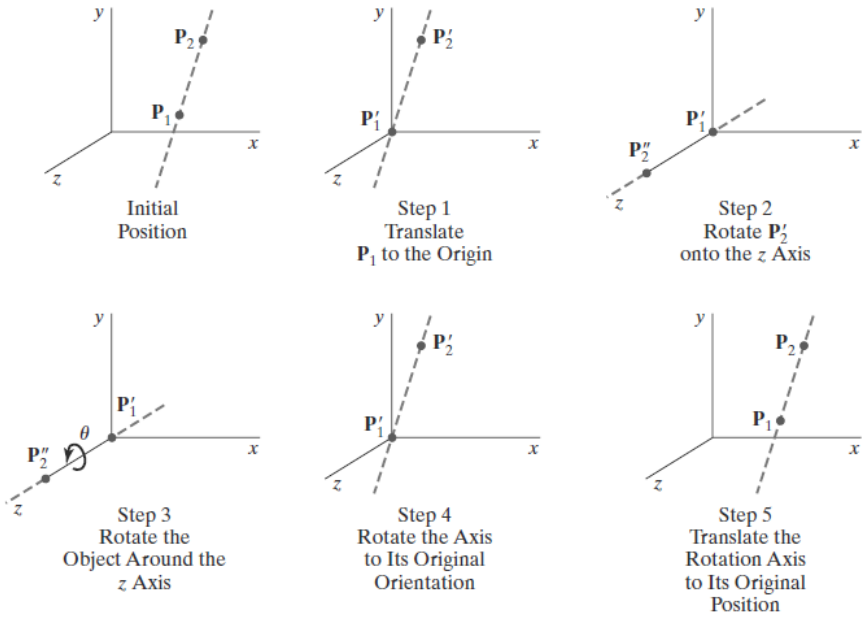


FIGURE 9
Five transformation steps for obtaining a composite matrix for rotation about an arbitrary axis, with the rotation axis projected onto the z axis.

A rotation axis can be defined with two coordinate positions, as in Figure 10, or with one coordinate point and direction angles (or direction cosines) between the rotation axis and two of the coordinate axes.

We assume that the rotation axis is defined by two points, as illustrated, and that the direction of rotation is to be counterclockwise when looking along the axis from P_2 to P_1. The components of the rotation-axis vector are then computed as

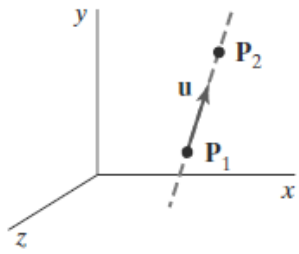


FIGURE 10
An axis of rotation (dashed line) defined with points P_1 and P_2. The direction for the unit axis vector **u** is determined by the specified rotation direction.

$$\mathbf{V} = \mathbf{P}_2 - \mathbf{P}_1 = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

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:

$$a = \frac{x_2 - x_1}{|\mathbf{V}|}, \qquad b = \frac{y_2 - y_1}{|\mathbf{V}|}, \qquad c = \frac{z_2 - z_1}{|\mathbf{V}|} \tag{14}$$

If the rotation is to be in the opposite direction (clockwise when viewing from P_2 to P_1), then we would reverse axis vector V and unit vector u so that they point in the direction from P_2 to P_1.

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.

Because we want a counterclockwise rotation when viewing along the axis from P_2 to P_1 (Figure 10), we move the point P_1 to the origin. (If the rotation had been specified in the opposite direction, we would move P_2 to the origin.)

This translation matrix is

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

which repositions the rotation axis and the object as shown in Figure 11.

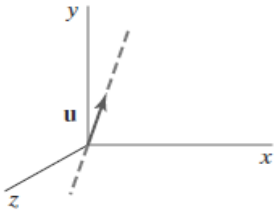


FIGURE 11
Translation of the rotation axis to the coordinate origin.

Next, we formulate the transformations that will put the rotation axis onto the z axis. We can use the coordinate-axis rotations to accomplish this alignment in two steps, and there are a number of ways to perform these two steps.

For this example, we first rotate about the x axis, then rotate about the y axis. The x-axis rotation gets vector u into the xz plane, and the y-axis rotation swings u around to the z axis. These two rotations are illustrated in Figure 12 for one possible orientation of vector u.

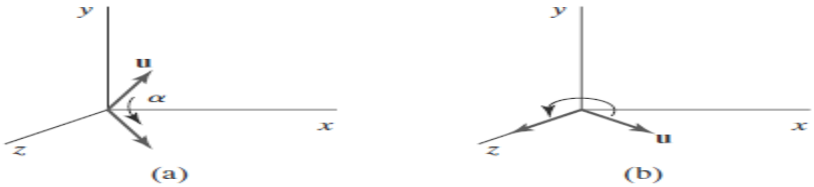


FIGURE 12
Unit vector **u** is rotated about the x axis to bring it into the xz plane (a), then it is rotated around the y axis to align it with the z axis (b).

We establish the transformation matrix for rotation around the x axis by determining the values for the sine and cosine of the rotation angle necessary to get u into the xz plane.

This rotation angle is the angle between the projection of u in the yz plane and the positive z axis (Figure 13).

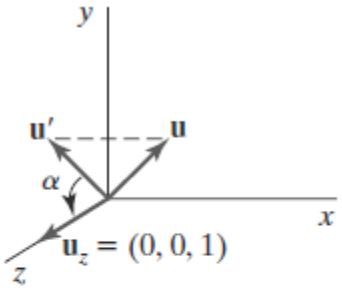


FIGURE 13
Rotation of **u** around the *x* axis into the *xz* plane is accomplished by rotating **u′** (the projection of **u** in the *yz* plane) through angle α onto the *z* axis.

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 uz 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 $\mathbf{u}'$:

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

Similarly, we can determine the sine of α from the cross-product of u` and uz. 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 right sides of Equations 18 and 19, and noting that $|\mathbf{u}| = 1_z$ and $|\mathbf{u}'| = d$, we have

$$d\sin\alpha = b$$

or

$$\sin\alpha = \frac{b}{d} \qquad (20)$$

Now that we have determined the values for **cosα** and **sinα** in terms of the components of vector **u**, we can set up the matrix elements for rotation of this vector about the x axis and into the xz plane:

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

The next step in the formulation of the transformation sequence is to determine the matrix that will swing the unit vector in the xz plane counterclockwise around the y axis onto the positive z axis. Figure 14 shows the orientation of the unit vector in the xz plane, resulting from the rotation about the x axis.

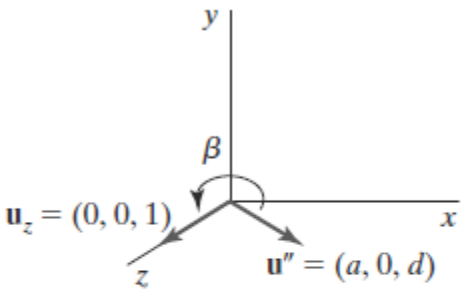


FIGURE 14
Rotation of unit vector $\mathbf{u}''$ (vector $\mathbf{u}$ after rotation into the xz plane) about the y axis. Positive rotation angle β aligns $\mathbf{u}''$ with vector $\mathbf{u}_z$.

This vector, labeled u``, has the value a for its x component, because rotation about the x axis leaves the x component unchanged. Its z component is d (the magnitude of u`), because vector u` has been rotated onto the z axis. Also, the y component of u`` is 0 because it now lies in the xz plane. Again, we can determine the cosine of rotation angle β from the dot product of unit vectors u`` and uz. Thus,

$$\cos\beta = \frac{\mathbf{u}'' \cdot \mathbf{u}_z}{|\mathbf{u}''|\,|\mathbf{u}_z|} = d \qquad (22)$$

because $|\mathbf{u}_z| = |\mathbf{u}''| = 1$. Comparing the coordinate-independent form of the cross-product

$$\mathbf{u}'' \times \mathbf{u}_z = \mathbf{u}_y\,|\mathbf{u}''|\,|\mathbf{u}_z|\sin\beta \qquad (23)$$

with the Cartesian form

$$\mathbf{u}'' \times \mathbf{u}_z = \mathbf{u}_y \cdot (-a) \tag{24}$$

we find that

$$\sin\beta = -a \tag{25}$$

Therefore, the transformation matrix for rotation of $\mathbf{u}''$ about the y axis is

$$\mathbf{R}_y(\beta) = \begin{bmatrix} d & 0 & -a & 0 \\ 0 & 1 & 0 & 0 \\ a & 0 & d & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{26}$$

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

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

To complete the required rotation about the given axis, we need to transform the rotation axis back to its original position. The transformation matrix for rotation about an arbitrary axis can then be expressed as the composition of these seven individual transformations:

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

Quaternion Methods for Three-Dimensional Rotations:

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.

They require less storage space than 4 × 4 matrices, and it is simpler to write quaternion procedures for transformation sequences. This is particularly important in animations, which often require complicated motion sequences and motion interpolations between two given positions of an object.

One way to characterize a quaternion is as an ordered pair, consisting of a scalar part and a vector part:

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

We can also think of a quaternion as a higher-order complex number with one real part (the scalar part) and three complex parts (the elements of vector 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:

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

where u is a unit vector along the selected rotation axis and θ is the specified rotation angle about this axis (Figure 16).

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

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

with the coordinates of the point as the vector part p = (x, y, z). The rotation of the point is then carried out with the quaternion operation

$$\mathbf{P^1} = \mathbf{qPq^{-1}}$$

where $\mathbf{q^{-1}} = (\mathbf{s}, -\mathbf{v})$ is the inverse of the unit quaternion q with the scalar and vector parts
This transformation produces the following new quaternion:

$$\mathbf{P^1} = (0, \mathbf{p^1})$$

The second term in this ordered pair is the rotated point position p^1, which is evaluated with vector dot and cross-products as

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

We can evaluate the terms in the above equation using the definition for quaternion multiplication.
Also, designating the components of the vector part of q as v = (a, b, c) , we obtain the elements for the composite rotation matrix $\mathbf{R^{-1}x(\alpha)\cdot R^{-1}y(\beta)\cdot Rz(\theta)\cdot Ry(\beta)\cdot Rx(\alpha)}$ in a 3 × 3 form as

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

The calculations involved in this matrix can be greatly reduced by substituting explicit values for parameters a, b, c, and s, and then using the following trigonometric identities to simplify the terms:

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

Thus, we can rewrite Matrix as

$$\mathbf{M}_R(\theta) =$$

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

where u_x, u_y, and u_z are the components of the unit axis vector **u**.

To complete the transformation sequence for rotating about an arbitrarily placed rotation axis, we need to include the translations that move the rotation axis to the coordinate axis and return it to its original position. Thus, the complete quaternion rotation expression, is

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

Three-Dimensional Scaling:

The matrix expression for the three-dimensional scaling transformation of a position P = (x, y, z) relative to the coordinate origin is a simple extension of two-dimensional scaling. We just include the parameter for z-coordinate scaling in the transformation matrix:

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

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

$$\mathbf{P}^1 = \mathbf{S} \cdot \mathbf{P}$$

where scaling parameters sx, sy, and sz are assigned any positive values. Explicit expressions for the scaling transformation relative to the origin are

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

Scaling an object changes the position of the object relative to the coordinate origin. A parameter value greater than 1 moves a point farther from the origin in the corresponding coordinate direction.

Similarly, a parameter value less than 1 moves a point closer to the origin in that coordinate direction. Also, if the scaling parameters are not all equal, relative dimensions of a transformed object are changed. We preserve the original shape of an object with a uniform scaling: $s_x = s_y = s_z$. The result of scaling an object uniformly, with each scaling parameter set to 2, is illustrated in Figure 17

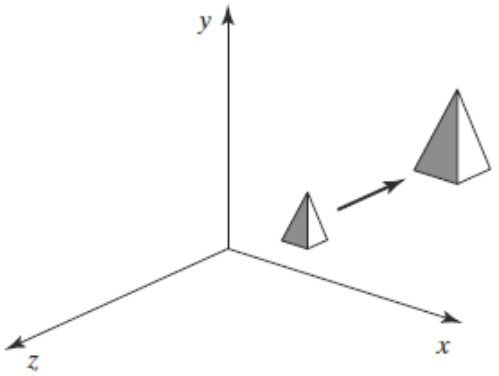


FIGURE 17
Doubling the size of an object with transformation 41 also moves the object farther from the origin.

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 in Figure 18.

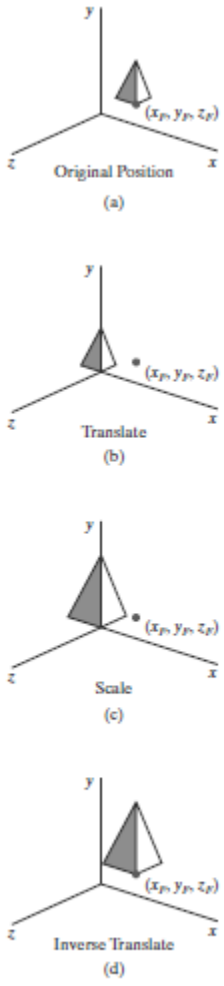


FIGURE 18
A sequence of transformations for scaling an object relative to a selected fixed point, using Equation 41.

The matrix representation for an arbitrary fixed-point scaling can then be expressed as the concatenation of these translate-scale-translate transformations:

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

An inverse, three-dimensional scaling matrix is set up by replacing each scaling parameter (s_x, s_y, and s_z)with its reciprocal.
However, this inverse transformation is undefined if any scaling parameter is assigned the value 0. The inverse matrix generates an opposite scaling transformation, and the concatenation of a three-dimensional scaling matrix with its inverse yields the identity matrix.

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.
- In general, three-dimensional reflection matrices are set up similarly to those for two dimensions.
- Reflections relative to a given axis are equivalent to 180◦ rotations about that axis.
- Reflections with respect to a plane are similar; when the reflection plane is a coordinate plane (xy, xz, or yz), 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 from a right-handed system to a left-handed system(or vice versa) is shown in Figure 19.

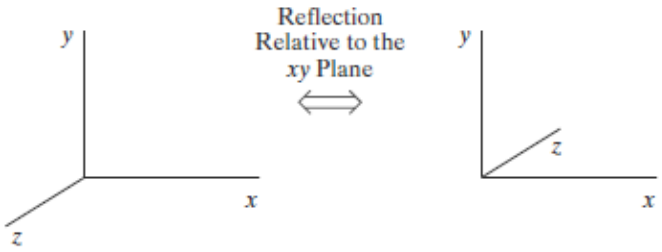


FIGURE 19
Conversion of coordinate specifications between a right-handed and a left-handed system can be carried out with the reflection transformation 45.

- This transformation changes the sign of z coordinates, leaving the values for the x and y coordinates unchanged. The matrix representation for this reflection relative to the xy plane is

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

Transformation matrices for inverting x coordinates or y coordinates are defined similarly, as reflections relative to the yz plane or to the xz plane, respectively. Reflections about other planes can be obtained as a combination of rotations and coordinate-plane reflections.

Three-Dimensional Shears:

These transformations can be used to modify object shapes, just as in two-dimensional applications. They are also applied in three-dimensional viewing transformations for perspective projections.

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:

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

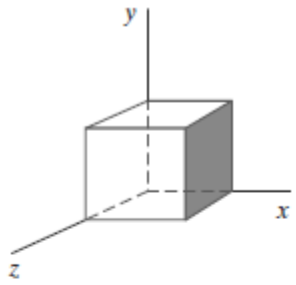
Shearing parameters sh_{zx} and sh_{zy} can be assigned any real values.

The effect of this transformation matrix is to alter the values for the x and y coordinates by an amount that is proportional to the distance from zref, while leaving the z coordinate unchanged.

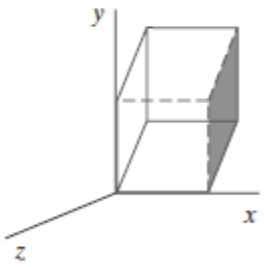
Plane areas that are perpendicular to the z axis are thus shifted by an amount equal to z − zref.

An example of the effect of this shearing matrix on a unit cube is shown in Figure 20 for shearing values $sh_{zx} = sh_{zy} = 1$ and a reference position zref = 0.

Three-dimensional transformation matrices for an x-axis shear and a y-axis shear are similar to the two-dimensional matrices. We just need to add a row and a column for the z coordinate shearing parameters.



(a)



(b)

FIGURE 20
A unit cube (a) is sheared relative to the origin (b) by Matrix 46, with $sh_{zx} = sh_{zy} = 1$.

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.

- Each of the transformed coordinates x`, y`, and z` is a linear function of the original coordinates x, y, and z, and parameters aij and bk are constants determined by the transformation type.

- 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, and shear are examples of affine transformations.
- We can always express any affine transformation as some composition of these five 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.
- An affine transformation involving only translation, rotation, and reflection preserves angles and lengths, as well as parallel lines.
- For each of these three transformations, line lengths and the angle between any two lines remain the same after the transformation.

OpenGL Geometric Transformation Functions:

Basic OpenGL Geometric Transformations

A 4× 4 translation matrix is constructed with the following routine:

glTranslate* (tx, ty, tz);

Translation parameters **tx**, **ty**, and **tz** can be assigned any real-number values, and the single suffix code to be affixed to this function is either **f** (float) or **d** (double).

For two-dimensional applications, we set **tz** = 0.0; and a two-dimensional position is represented as a four-element column matrix with the z component equal to 0.0. The translation matrix generated by this function is used to transform positions of objects defined after this function is invoked.

For example, we translate subsequently defined coordinate positions 25 units in the *x* direction and −10
units in the *y* direction with the statement

glTranslatef (25.0, -10.0, 0.0);

Similarly, a 4 × 4 rotation matrix is generated with

glRotate* (theta, vx, vy, vz);

where the vector **v** = (**vx**, **vy**, **vz**) can have any floating-point values for its components.

This vector defines the orientation for a rotation axis that passes through the coordinate origin. If **v** is not specified as a unit vector, then it is normalized automatically before the elements of the rotation matrix are computed. The suffix code can be either **f** or **d**, and parameter **theta** is to be assigned a rotation angle in degrees, which the routine converts to radians for the trigonometric calculations.

The rotation specified here will be applied to positions defined after this function call. Rotation in two-dimensional systems is rotation about the *z* axis, specified as a unit vector with *x* and *y* components of zero, and a *z* component of 1.0. For example, the statement

glRotatef (90.0, 0.0, 0.0, 1.0);
sets up the matrix for a 90◦ rotation about the *z* axis. We should note here that internally, this function generates a rotation matrix using *quaternions*.

This method is more efficient when rotation is about an arbitrarily-specific axis.

We obtain a 4 × 4 scaling matrix with respect to the coordinate origin with the following routine:

glScale* (sx, sy, sz);

The suffix code is again either **f** or **d**, and the scaling parameters can be assigned any real-number values. Scaling in a two-dimensional system involves changes in the *x* and *y* dimensions, so a typical two-dimensional scaling operation has a *z* scaling factor of 1.0 (which causes no change in the *z* coordinate of positions).

Because the scaling parameters can be any real-number value, this function will also generate reflections when negative values are assigned to the scaling parameters.For example, the following statement produces a matrix that scales by a factor of 2 in the *x* direction, scales by a factor of 3 in the *y* direction, and reflects with respect to the *x* axis:

glScalef (2.0, -3.0, 1.0);

A zero value for any scaling parameter can cause a processing error because an inverse matrix cannot be calculated. The scale-reflect matrix is applied to subsequently defined objects.

It is important to note that internally OpenGL uses composite matrices to hold transformations. As a result, transformations are cumulative—that is, if we apply a translation and then apply a rotation, objects whose positions are specified after that will have both transformations applied to them. If that is not the behavior we desired, we must be able to remove the effects of previous transformations. This requires additional functions for manipulating the composite matrices.

OpenGL Matrix Operations:

The **glMatrixMode** routine is used to set the *projection mode* , which designates the matrix that is to be used for the projection transformation.
This transformation determines how a scene is to be projected onto the screen. We use the same routine to set up a matrix for the geometric transformations.
In this case, however, the matrix is referred to as the *modelview matrix*, and it is used to store and combine the geometric transformations.
It is also used to combine the geometric transformations with the transformation to a viewing-coordinate system.

We specify the *modelview mode* with the statement

glMatrixMode (GL_MODELVIEW);
which designates the 4×4 modelview matrix as the **current matrix.**

The OpenGL transformation routines discussed in the previous section are all applied to whatever composite matrix is the current matrix, so it is important to use **glMatrixMode** to change to the modelview matrix before applying geometric transformations.

Following this call, OpenGL transformation routines are used to modify the modelview matrix, which is then applied to transform coordinate positions in a scene. Two other modes that we can set with the **glMatrixMode** function are the *texture mode* and the *color mode*.
The texture matrix is used for mapping texture patterns to surfaces, and the color matrix is used to convert from one color model to another.

The default argument for the **glMatrixMode** function is **GL MODELVIEW**.

Once we are in the modelview mode (or any other mode), a call to a transformation routine generates a matrix that is multiplied by the current matrix for that mode.
In addition, we can assign values to the elements of the current matrix, and there are two functions in the OpenGL library for this purpose.With the following function, we assign the identity matrix to the current matrix:

glLoadIdentity ( );

Alternatively, we can assign other values to the elements of the current matrix using

glLoadMatrix* (elements16);

A single-subscripted, 16-element array of floating-point values is specified with parameter **elements16**, and a suffix code of either **f** or **d** is used to designate the data type.
The elements in this array must be specified in *column-major* order. That is, we first list the four elements in the first column, and then we list the four elements in the second column, the third column, and finally the fourth column. To illustrate this ordering, we initialize the modelview matrix with the following code:

```
glMatrixMode (GL_MODELVIEW);

GLfloat elems [16];
GLint k;

for (k = 0; k < 16; k++)
   elems [k] = float (k);
glLoadMatrixf (elems);
```

which produces the matrix

$$\mathbf{M} = \begin{bmatrix} 0.0 & 4.0 & 8.0 & 12.0 \\ 1.0 & 5.0 & 9.0 & 13.0 \\ 2.0 & 6.0 & 10.0 & 14.0 \\ 3.0 & 7.0 & 11.0 & 15.0 \end{bmatrix}$$

We can also concatenate a specified matrix with the current matrix:

glMultMatrix*(otherElements16);

The suffix code is either f or d, and parameter otherElements16 is a 16-element , single subscripted array that lists the elements of some other matrix in column-major order.

The current matrix is postmultiplied by the matrix specified in glMultMatrix, and this product replaces the current matrix. Thus, assuming that the current matrix is the modelview matrix, which we designate as M, then the updated modelview matrix is compted as

$$\mathbf{M = M.M^1}$$

OpenGL Matrix Stacks:

- For each of the four modes (modelview, projection, texture and color) that we can select with the glMatrixMode function, OpenGL maintains a matrix stack.
- Initially, each stack contains only the identity matrix.
- At any time during the processing of a scene, the top matrix on each stack is called the "current matrix" for that mode.
- After we specify the viewing and geometric transformations, the top of the modelview matrix stack is the 4 by 4 composite matrix that combines the viewing transformations and the various geometric transformations that we want to apply to a scene.
- We can determine the number of positions available in the modelview stack for a particular implementation of OpenGL with

  glGetIntegerv(GL_MAX_MODELVIEW_STACK_DEPTH, stackSize);

  which returns a single integer value to array stackSize

Other OpenGL symbolic constants that can be used are:
GL_MAX_PROJECTION_STACK_DEPTH,

GL_MAX_TEXTURE_STACK_DEPTH

GL_MAX_COLOR_STACK_DEPTH

We can also find out how many matrices are currently in the stack with

glGetIntegerv(GL_MAX_MODELVIEW_STACK_DEPTH, numMats);

We have two functions available in OpenGL for processing the matrices in a stack. These stack-processing functions are more efficient than manipulating the stack matrices individually.

With the following function, we copy the current matrix at the top of the active stack and store the copy in the second stack position.

glPushMatrix();

This gives us duplicate matrices at the top two positions of the stack. The other stack function is

glPopMatrix();

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

To "pop" the top of the stack, there must be at least two matrices in the stack. Otherwise, we generate an error.

Color Models:

Properties of Light:

The Electromagnetic Spectrum

In physical terms, 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.

Figure 1 shows the approximate frequency ranges for these various aspects of electromagnetic radiation.

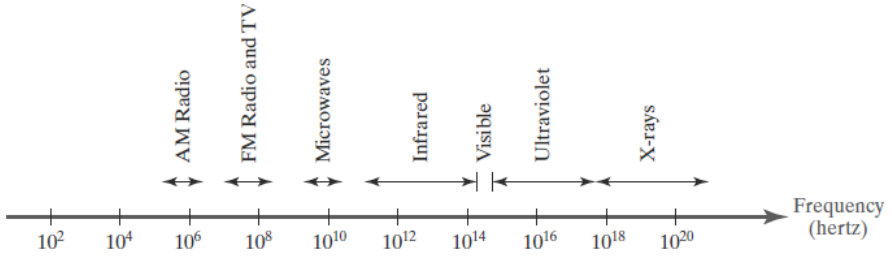


FIGURE 1
Electromagnetic spectrum.

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.

Actually, the human eye is sensitive to some frequencies into the infrared and ultraviolet bands. Spectral colors range from shades of red through orange and yellow, at the low-frequency end, to shades of green, blue, and violet at the high end.

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 each spectral color, the rate of oscillation of the field magnitude is given by the frequency f . Figure 2 illustrates the time-varying oscillations for the magnitude of the electric field within one plane.

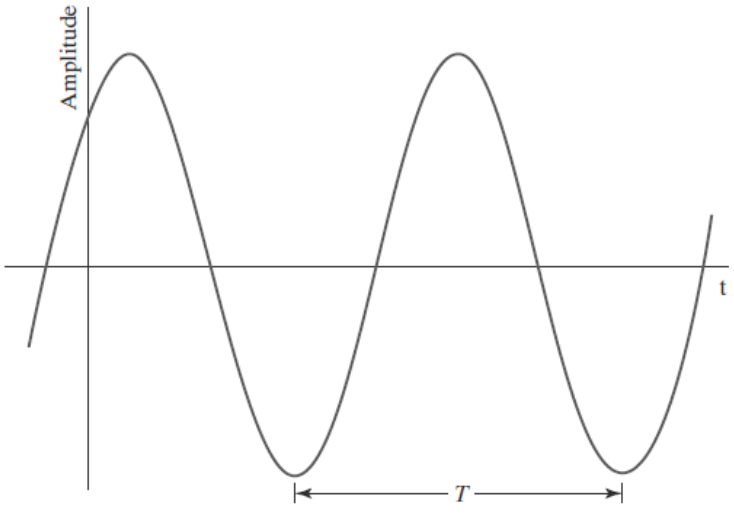


FIGURE 2
Time variations for the amplitude of the electric field for one frequency component of a plane-polarized electromagnetic wave. The time between two consecutive amplitude peaks or two consecutive amplitude minimums is called the period of the wave.

The time between any two consecutive positions on the wave that have the same amplitude is called the period (T) of the wave, which is the inverse of the frequency (i.e., T =1/ f). And the distance that the wave has traveled from the beginning of one oscillation to the beginning of the next oscillation is called the wavelength (λ).

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):

$$c = \lambda f$$

Frequency for each spectral color is a constant for all materials, but the speed of light and the wavelength are material dependent. In a vacuum, the speed of light is very nearly $c = 3\times10^{10}$ cm/sec.

Light wavelengths are very small, so length units for designating spectral colors are usually given in angstroms ($1 Å = 10^{-8}$ cm) or in nanometers ($1nm = 10^{-7}$ cm).

An equivalent term for nanometer is milli-micron. Light at the low-frequency end of the spectrum (red) has a wavelength of approximately 780 nanometers (nm), and the wavelength at the other end of the spectrum (violet) is about 380 nm. Because wavelength units are somewhat more convenient to deal with than frequency units, spectral colors are typically specified in terms of the wavelength values in a vacuum.

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.

The combination of frequencies present in the reflected light determines what we perceive as the color of the object. If low frequencies are predominant in the reflected light, the object is described as red. In this case, 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.

When we view a source of light, our eyes respond to the color (or dominant frequency) and two other basic sensations. One of these we call the brightness, which corresponds to the total light energy and can be quantified as the luminance of the light.

The third perceived characteristic is called the **purity**, or the **saturation**, of the light. Purity describes how close a light appears to be to a pure spectral color, such as red.

Pastels and pale colors have low purity (low saturation) and they appear to be nearly white. Another term, **chromaticity**, is used to refer collectively to the two properties describing color characteristics: purity and dominant frequency (hue).

Radiation emitted by a white light source has an energy distribution that can be represented over the visible frequencies as in Figure 3.

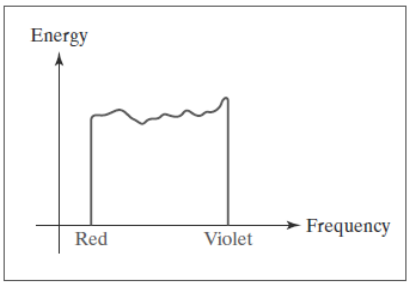


FIGURE 3
Energy distribution for a white light source.

Each frequency component within the range from red to violet contributes more or less equally to the total energy, and the color of the source is described as white.

When a dominant frequency is present, the energy distribution for the source takes a form such as that in Figure 4.

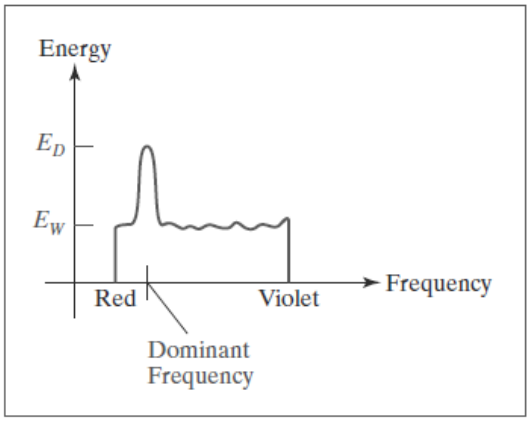


FIGURE 4
Energy distribution for a light source with a dominant frequency near the red end of the frequency range.

We would describe this light as a red color (the dominant frequency), with a relatively high value for the purity. The energy density of the dominant light component is labeled as E_D in this figure, and the contributions from the other frequencies produce white light of energy density E_W.

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 E_D and E_W.

The larger the energy E_D of the dominant frequency compared to the white-light component E_W, the higher the purity of the light.

We have a purity of 100 percent when $E_W = 0$ and a purity of 0 percent when $E_W = E_D$.
of the light. The third perceived characteristic is called the purity, or the saturation,
of the light.

Color Models

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

No single model can explain all aspects of color, so we make use of different models to help describe different color characteristics.

Primary Colors

When we combine the light from two or more sources with different dominant frequencies, we can vary the amount (intensity) of light from each source to generate a range of additional colors. This represents one method for forming a color model.

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. No finite set of real primary colors can be combined to produce all possible visible colors.

Nevertheless, three primaries are sufficient for most purposes, and colors not in the color gamut for a specified set of primaries can still be described using extended methods.

Given a set of three primary colors, we can characterize any fourth color using color-mixing processes. Thus, a mixture of one or two of the primaries with the fourth color can be used to match some combination of the remaining primaries.

In this extended sense, a set of three primary colors can be considered to describe all colors. Figure 5 shows a set of color-matching functions for three primaries and the amount of each needed to produce any spectral color.

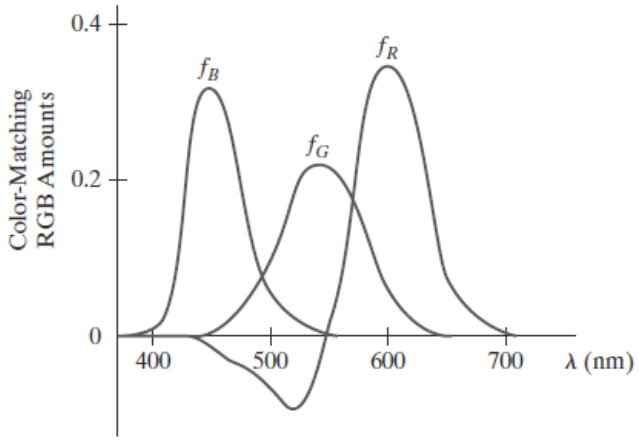


FIGURE 5
Three color-matching functions for displaying spectral frequencies within the approximate range from 400 nm to 700 nm.

The curves plotted in Figure 5 were obtained by averaging the judgments of a large number of observers. Colors in the vicinity of 500 nm can be matched only by "subtracting" an amount of red light from a combination of blue and green lights. This means that a color around 500 nm is described only by combining that color with an amount of red light to produce the blue-green combination specified in the diagram. Thus, an RGB color monitor cannot display colors in the neighborhood of 500 nm.

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. The more black pigment, the darker the shade.

Similarly, different tints of the color are obtained by adding a white pigment to the original color, making it lighter as more white is added. Tones of the color are produced by adding both black and white pigments.

To many, these color concepts are more intuitive than describing a color as a set of three numbers that give the relative proportions of the primary colors. It is generally much easier to think of creating a pastel red color by adding white to pure red and producing a dark blue color by adding black to pure blue. Therefore, graphics packages providing color palettes to a user often employ two or more color models. One model provides an intuitive color interface for the user, and the others describe the color components for the output devices.

The RGB Color Model:

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).

By comparing intensities in a light source, we perceive the color of the light. This theory of vision is the basis for displaying color output on a video monitor using 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 11.

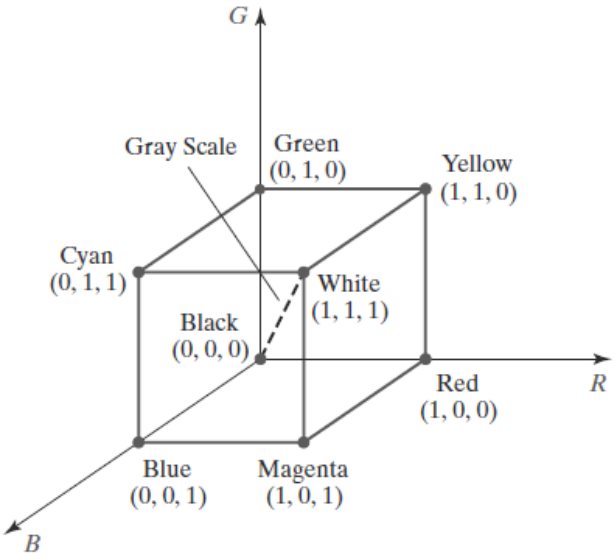


FIGURE 11
The RGB color model. Any color within the unit cube can be described as an additive combination of the three primary colors.

The origin represents black and the diagonally opposite vertex, with coordinates (1, 1, 1), is white.

Vertices of the cube on the axes represent the primary colors, and the remaining vertices are the complementary color points for each of the primary colors.

As with the XYZ color system, 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:

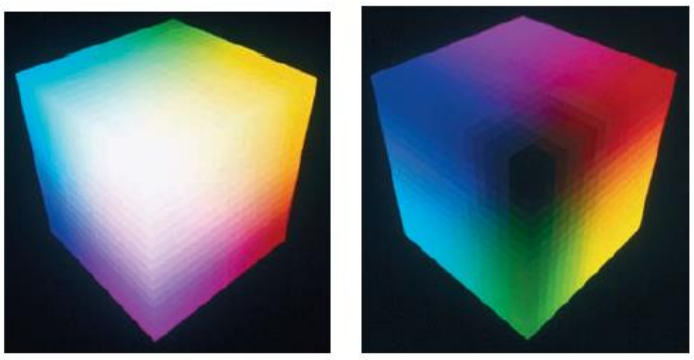
$$\mathbf{C(\lambda) = (R, G, B) = RR + GG + BB}$$

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

For example, the magenta vertex is obtained by adding maximum red and blue values to produce the triple (1, 0, 1), and white at (1, 1, 1) is the sum of the maximum values for red, green, and blue.

Shades of gray are represented along the main diagonal of the cube from the origin (black) to the white vertex. Points along this diagonal have equal contributions from each primary color, and a gray shade halfway between black and white is represented as (0.5, 0.5, 0.5).

The color graduations along the front and top planes of the RGB cube are illustrated in Color Plate 22



Color Plate 22
Two views of the RGB color cube. View (a) is along the gray-scale diagonal from white to black, and view (b) is along the gray-scale diagonal from black to white.

The CMY and CMYK Color Models:

A video monitor displays color patterns by combining light that is emitted from the screen phosphors, which is an additive process. However, hard-copy devices, such as printers and plotters, produce a color picture by coating a paper with color pigments. We see the color patterns on the paper by reflected light, which is a subtractive process.

The CMY Parameters:

A subtractive color model can be formed with the three primary colors cyan, magenta, and yellow.

As we have noted, cyan can be described as a combination of green and blue. Therefore, when white light is reflected from cyan colored ink, the reflected light contains only the green and blue components, and the red component is absorbed, or subtracted, by the ink.

Similarly, magenta ink subtracts the green component from incident light, and yellow subtracts the blue component. A unit cube representation for the CMY model is illustrated in Figure 13.

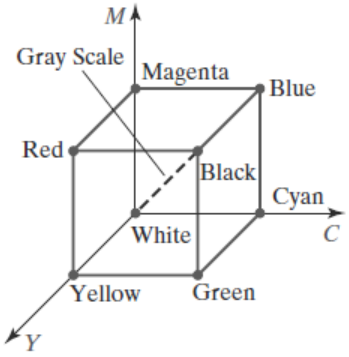


FIGURE 13
The CMY color model. Positions within the unit cube are described by subtracting the specified amounts of the primary colors from white.

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. A black dot is included because reflected light from the cyan, magenta, and yellow inks typically produce only shades of gray. Some plotters produce different color combinations by spraying the ink for the three primary colors over each other and allowing them to mix before they dry. For black-and-white or grayscale printing, only the black ink is used.

Transformations Between CMY and RGB Color Spaces

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

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

In this transformation, the unit column vector represents the black point in the CMY color space.

For the conversion from RGB to the CMYK color space, we first set K = max(R, G, B). Then K is subtracted from each of C, M, and Y .

Similarly, for the transformation from CMYK to RGB, we first set K =min(R,G, B). Then K is subtracted from each of R, G, and B in Equation 12.

In practice, these transformation equations are often modified to improve the printing quality for a particular system.

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.

We refer to the model for calculating the light intensity at a single surface point as an illumination model or a lighting model

Illumination models in computer graphics are often approximations of the physical laws that describe surface-lighting effects.

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. In some applications, however, we may want to create an object that is both a light source and a light reflector.

For example, a plastic globe surrounding a light bulb both emits and reflects light from the surface of the globe. We could also model the globe as a semitransparent surface around a light source. However, for some objects, such as a large fluorescent light panel, it might be more convenient to describe the surface simply as a combination emitter and reflector.

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.

If the source is also to be a light-reflecting surface, we need to give its reflectivity properties. In addition, we could set up a light source that emits different colors in different directions.

For example, we could define a light source that emits a red light on one side and a green light on the other side.

In most applications, and particularly for real-time graphics displays, a simple light-source model is used to avoid excessive computations. 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.

Point Light Sources:

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.

We define a point source for a scene by giving its position and the color of the emitted light.

FIGURE 1
Diverging ray paths from a point light source.

As shown in Figure 1, 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.

We can also simulate larger sources as point emitters if they are not too close to a scene. We use the position of a point source in an illumination model to determine which objects in the scene are illuminated by that source and to calculate the light direction to a selected object surface position.

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.

In contrast to a light source in the middle of a scene, which illuminates objects on all sides of the source, a remote source illuminates the scene from only one direction.

The light path from a distant light source to any position in the scene is nearly constant, as illustrated in Figure 2.

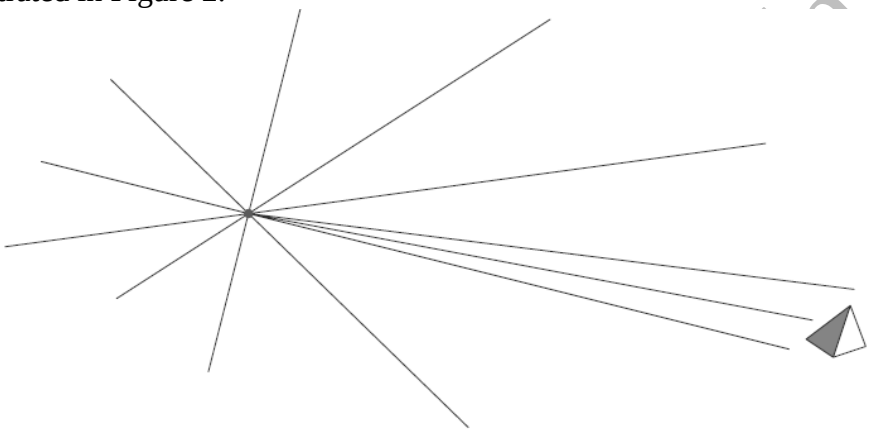


FIGURE 2
Light rays from an infinitely distant light source illuminate an object along nearly parallel light paths.

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 $1/d^2_l$.

This means that a surface close to the light source receives a higher incident light intensity from that source than a more distant surface. Therefore, to produce realistic lighting effects, we should take this intensity attenuation into account. Otherwise, all surfaces are illuminated with the same intensity from a light source, and undesirable display effects can result.

For example, if two surfaces with the same optical parameters project to overlapping positions, they would be indistinguishable from one another. Thus, regardless of their relative distances from the light source, the two surfaces would appear to be one surface.

In practice, however, using an attenuation factor of $1/d^2_l$ with a point source does not always produce realistic pictures. The factor $1/d^2_l$ 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. This is

because actual light sources are not infinitesimal points, and illuminating a scene with point emitters is only a simple approximation of true lighting effects.

To generate more realistic displays using point sources, 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, $\mathbf{a_0}$, $\mathbf{a_1}$, and $\mathbf{a_2}$, can then be adjusted to produce optimal attenuation effects. For instance, we can assign a large value to $\mathbf{a_0}$ when $\mathbf{d_l}$ is very small to prevent $\mathbf{f_{radatten}(d_l)}$ from becoming too large.
As an additional option, often available in graphics packages, a different set of values for the attenuation coefficients could be assigned to each point light source in the scene.

We cannot apply the above intensity-attenuation calculation to a point source at "infinity," because the distance to the light source is indeterminate. Also, all points in the scene are at a nearly equal distance from a far-off source. To accommodate both remote and local light sources, we can express the intensity-attenuation function as

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

Directional Light Sources and Spotlight Effects:

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. This defines a conical region of space with the light-source vector direction along the axis of the cone (Figure 3).

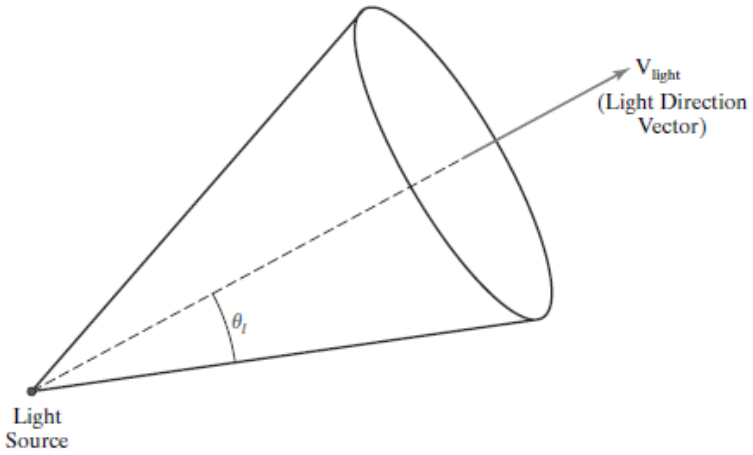


FIGURE 3
A directional point light source. The unit light-direction vector defines the axis of a light cone, and angle θ_l defines the angular extent of the circular cone.

multiple direction vectors and a different emission color for each direction.

We can denote $\mathbf{V}_{light}$ as the unit vector in the light-source direction and $\mathbf{V}_{obj}$ as the unit vector in the direction from the light position to an object position. Then

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

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 $0^\circ < \theta_l \leq 90^\circ$, then the object is within the spotlight if $\cos\alpha \geq \cos\theta_l$, as shown in Figure 4. If $\mathbf{V}_{obj} \cdot \mathbf{V}_{light} < \cos\boldsymbol{\theta_l}$, however, the object is outside the light cone.

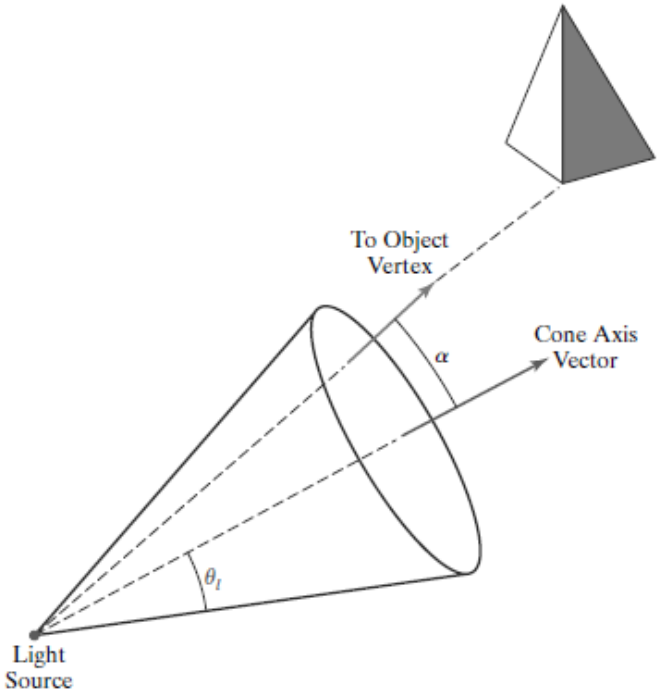


FIGURE 4
An object illuminated by a directional point light source.

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 us to simulate a cone of light that is most intense along the axis of the cone, with the intensity decreasing as we move farther from the cone axis.

A commonly used angular intensity-attenuation function for a directional light source is

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

where the attenuation exponent al is assigned some positive value and angle φ is measured from the cone axis. Along the cone axis, $\varphi = 0^\circ$ and $\boldsymbol{f_{angatten}(\varphi)} = 1.0$.

The greater the value for the attenuation exponent $\boldsymbol{a_l}$, the smaller the value of the angular intensity-attenuation function for a given value of angle $\varphi > 0^\circ$.

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). Also, an object is not illuminated by the light source if it is anywhere outside the cone of the spotlight.

To determine the angular attenuation factor along a line from the light position to a surface position in a scene, we can compute the cosine of the direction angle from the cone axis using the dot product calculation in the following Equation.

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

We designate $\mathbf{V}_{light}$ as the unit vector in the light-source direction (along the cone axis) and $\mathbf{V}_{obj}$ as the unit vector in the direction from the light source to an object position. Using these two unit vectors and assuming that $0° < \theta_l \leq 90°$, we can express the general equation for angular attenuation as

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

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 in Figure 5, we can approximate it as a light emitting surface.

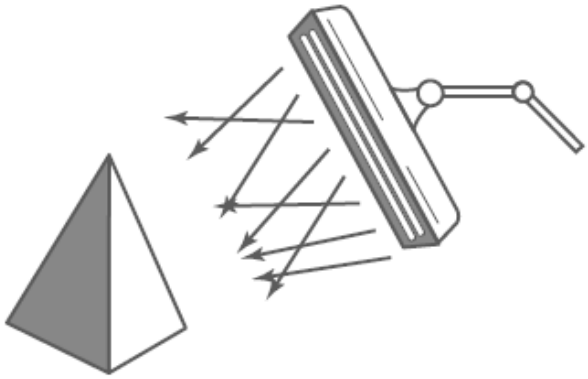


FIGURE 5
An object illuminated by a large nearby light source.

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. For instance, two flaps can be set up for each of the x, y, and z directions to further restrict the path of the emitted light rays. This light-source simulation is implemented in some graphics packages.

Basic Illumination Models:

Accurate surface lighting models compute the results of interactions between incident radiant energy and the material composition of an object.

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

Ambient Light:

In our basic illumination model, we can incorporate background lighting by setting a general brightness level for a scene. 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.

Assuming that we are describing only monochromatic lighting effects, such as shades of gray, we designate the level for the ambient light in a scene with an intensity parameter $\mathbf{I_a}$.

Each surface in the scene is then illuminated with this background light. 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:

We can model diffuse reflections from a surface by assuming that the incident light 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 from any 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 ***dA*** in a direction $\boldsymbol{\varphi_N}$ relative to the surface normal is proportional to cos φN (Figure 9).

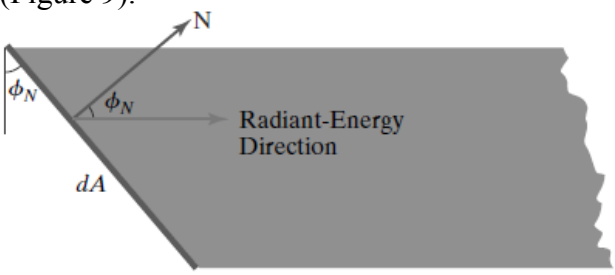


FIGURE 9
Radiant energy from a surface area element dA in direction ϕ_N relative to the surface normal direction is proportional to $\cos\phi_N$.

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:

$$\text{Intensity} = \frac{\text{radiant energy per unit time}}{\text{projected area}}$$

$$\propto \frac{\cos\phi_N}{dA\cos\phi_N}$$

$$= \text{constant}$$

Thus, for Lambertion reflection, the intensity of light is the same over all viewing directions.

Assuming that every surface is to be treated as an ideal diffuse reflector (Lambertian), we can set a parameter k_d 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**.

The diffuse reflection in any direction is then a constant, which is equal to the incident light intensity multiplied by the diffuse-reflection coefficient.

For a monochromatic light source, parameter k_d is assigned a constant value in the interval 0.0 to 1.0, according to the reflecting properties we want the surface to have.

If we want a highly reflective surface, we set the value of k_d near 1.0. This produces a brighter surface with the intensity of the reflected light near that of the incident light. If we want to simulate a surface that absorbs most of the incident light, we set the reflectivity to a value near 0.0.

For the background lighting effects, we can assume that every surface is fully illuminated by the ambient light $\mathbf{I_a}$ that we assigned to the scene. Therefore, the ambient contribution to the diffuse reflection at any point on a surface is simply

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

When a surface is illuminated by a light source with an intensity I_l , the amount of incident light from the source depends on the orientation of the surface relative to the light source direction.

A surface that is oriented nearly perpendicular to the illumination direction receives more light from the source than a surface that is tilted at an oblique angle to the direction of the incoming light. This illumination effect can be observed on a white sheet of paper or smooth cardboard that is placed parallel to a sunlit window.

As the sheet is slowly rotated away from the window direction, the surface appears less bright. Figure 10 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 (parallel incoming rays).

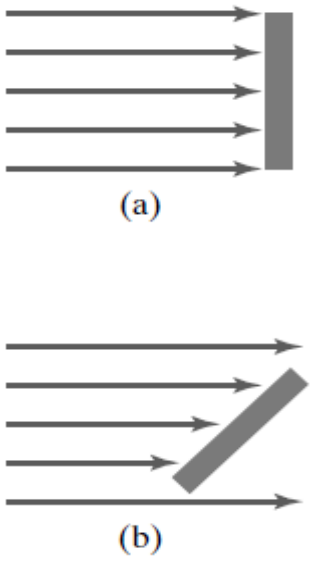


FIGURE 10

From Figure 10, we see that the number of light rays intersecting a surface element is proportional to the area of the surface projection perpendicular to the incident light direction.

If we denote the angle of incidence between the incoming light direction and the surface normal as θ (Figure 11), then the projected area of a surface element perpendicular to the light direction is proportional to **cosθ**. Therefore, we can model the amount of incident light on a surface from a source with intensity $\boldsymbol{I_l}$ as

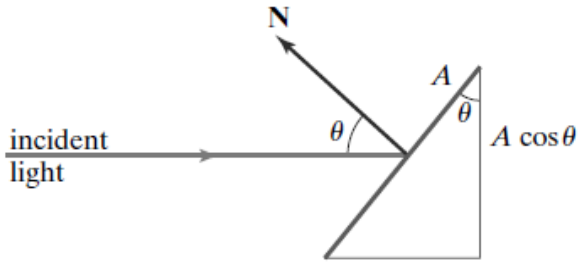


FIGURE 11
An illuminated area A projected perpendicular to the path of incoming light rays. This perpendicular projection has an area equal to $A \cos\theta$.

$$I_{l,\text{incident}} = I_l \cos\theta \tag{8}$$

Using Equation 8, we can model the diffuse reflections from a light source with intensity I_l using the calculation

$$\begin{aligned} I_{l,\text{diff}} &= k_d\, I_{l,\text{incident}} \\ &= k_d\, I_l \cos\theta \end{aligned} \tag{9}$$

When the incoming light from the source is perpendicular to the surface at a particular point, θ = 90◦ and $\boldsymbol{I_{l,diff} = k_d\, I_l}$.

As the angle of incidence increases, the illumination from the light source decreases. Furthermore, a surface is illuminated by a point source only if the angle of incidence is in the range 0° to 90° (cosθ is in the interval from 0.0 to 1.0). When cosθ < 0.0, the light source is behind the surface.

At any surface position, we can denote the unit normal vector as N and the unit direction vector to a point source as **L**, as in Figure 12. Then, **cosθ = N· L** and the diffuse reflection equation for single point-source illumination at a surface position can be expressed in the form

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

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

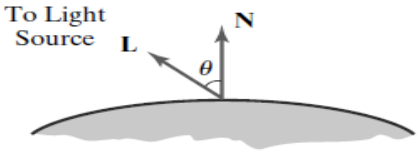


FIGURE 12
Angle of incidence θ between the unit light-source direction vector **L** and the unit normal vector **N** at a surface position.

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

A light source at "infinity," however, has no position, only a propagation direction. In that case, we use the negative of the assigned light-source emission direction for the direction of vector L.

We can combine the ambient and point-source intensity calculations to obtain an expression for the total diffuse reflection at a surface position. In addition, many graphics packages introduce an ambient-reflection coefficient k_a that can be assigned to each surface to modify the ambient-light intensity I_a . This simply provides us with an additional parameter for adjusting the lighting effects in our empirical model.

Using parameter k_a , we can write the total diffuse-reflection equation for a single point source as

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

where both $\mathbf{k_a}$ and $\mathbf{k_d}$ 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.

Figure 13 shows the specular reflection direction for a position on an illuminated surface. 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**.

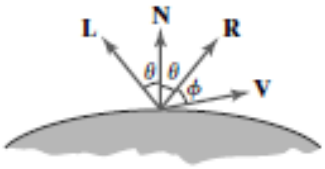


FIGURE 13
Specular reflection angle equals angle of incidence θ.

In this figure, **R** represents the unit vector in the direction of ideal specular reflection, **L** is the unit vector directed toward the point light source, and **V** is the unit vector pointing to the viewer from the selected surface position.

Angle **φ** is the viewing angle relative to the specular-reflection direction **R**. For an ideal reflector (a perfect mirror), incident light is reflected only in the specular-reflection direction, and we would see reflected light only when vectors **V** and **R** coincide (φ = 0).

Objects other than ideal reflectors exhibit specular reflections over a finite range of viewing positions around vector **R**. Shiny surfaces have a narrow specular reflection range, and dull surfaces have a wider reflection range.

An empirical model for calculating the specular reflection range, developed by *Phong Bui Tuong* and called the **Phong specular-reflection model** or simply the **Phong model**, sets the intensity of specular reflection proportional to $\mathbf{cos^{ns}\ \varphi}$.

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. Figures 14 show the effect of ns on the angular range for which we can expect to see specular reflections.

FIGURE 14
Modeling specular reflections (shaded area) with parameter n_s.

The intensity of specular reflection depends on the material properties of the surface and the angle of incidence, as well as other factors such as the polarization and color of the incident light.

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). The variation of specular intensity with angle of incidence is described by ***Fresnel's Laws of Reflection.***

Using the spectral-reflection function , we can write the Phong specular-reflection model as

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

where $\boldsymbol{I_l}$ is the intensity of the light source, and φ is the viewing angle relative to the specular-reflection direction R.

We then simply set ***specular-reflection coefficient*** $\boldsymbol{k_s}$ equal to some value in the range from 0 to 1.0 for each surface. 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. Thus, assuming the specular-reflection coefficient is a constant for any material, we can determine the intensity of the specular reflection due to a point light source at a surface position with the calculation

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

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

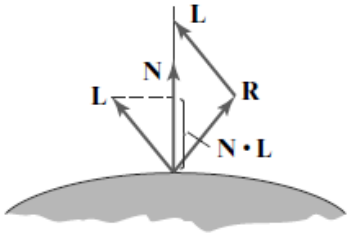


FIGURE 17
The projection of either **L** or **R** onto the direction of the normal vector **N** has a magnitude equal to **N · L**.

As seen in Figure 17, 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

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

and the specular-reflection vector is obtained as

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

We calculate **V** using the surface position and the viewing position, in same way that we obtained the unit vector **L**. But if a fixed viewing direction is to be used for all positions in a scene, we can set **V = (0.0, 0.0, 1.0)**, which is a unit vector in the positive z direction.

Specular calculations take less time to calculate using a constant **V**, but the displays are not as realistic.

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 (Figure 18).

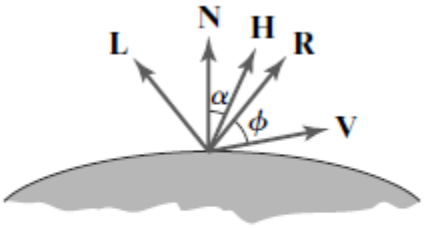


FIGURE 18
Halfway vector **H** along the bisector of the angle between **L** and **V**.

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**. Also, if both the viewer and the light source are sufficiently far from the surface, vectors **V** and **L** are each constants, and thus **H** is also constant for all surface points.

If the angle between **H** and **N** is greater than 90°, **N·H** is negative and we set the specular-reflection contribution to 0.0.

Halfway vector **H** along the bisector of the angle between **L** and **V**. Vector **H** is the orientation direction for the surface that would produce maximum specular reflection in the viewing direction, for a given position of a point light source. For this reason, **H** is sometimes referred to as the surface orientation direction for maximum highlights.

Also, if vector **V** is coplanar with vectors **L** and **R** (and thus **N**), angle α has the value $\varphi/2$.

When **V, L,** and **N** are not coplanar, $\alpha > \varphi/2$, depending on the spatial relationship of the three vectors.

Combined Diffuse and Specular Reflections:

For a single point light source, we can model the combined diffuse and specularreflections from a position on an illuminated surface as

$$I = I_{\text{diff}} + I_{\text{spec}}$$
$$= k_a I_a + k_d I_l(\mathbf{N} \cdot \mathbf{L}) + k_s I_l(\mathbf{N} \cdot \mathbf{H})^{n_s}$$

The surface is illuminated only with ambient light if the light source is behind the surface, and there are no specular effects if **V** and **L** are on the same side of the normal vector **N**.

Diffuse and Specular Reflections from Multiple Light Sources:

We can place any number of light sources in a scene. For multiple point light sources, we compute the diffuse and specular reflections as a sum of the contributions from the various sources, as follows:

$$I = I_{\text{ambdiff}} + \sum_{l=1}^{n} [I_{l,\text{diff}} + I_{l,\text{spec}}]$$

$$= k_a I_a + \sum_{l=1}^{n} I_l [k_d (\mathbf{N} \cdot \mathbf{L}) + k_s (\mathbf{N} \cdot \mathbf{H})^{n_s}]$$

Surface Light Emissions:

Some surfaces in a scene could be emitting light, as well as reflecting light from their surfaces.

For example, a room scene can contain lamps or overhead lighting, and outdoor night scenes could include streetlights, store signs, and automobile headlights.

We can empirically model surface light emissions by simply includingan emission term $I_{surfemission}$ in the illumination model in the same way that we simulated background lighting using an ambient light level.

This surface emission is then added to the surface reflections resulting from the light-source and the background-lighting illumination.

To illuminate other objects from a light-emitting surface, we could position a directional light source behind the surface to produce a cone of light through the surface. Alternatively, we could simulate the emission with a set of point light sources distributed over the surface.

In general, however, an emitting surface is usually not used in the basic illumination model to illuminate other surfaces because of the added calculation time. Rather, surface emissions are used as a simple means for approximating the appearance of the surface of an extended light-source. This produces a glowing effect for the surface.

Basic Illumination Model with Intensity

Attenuation and Spotlights

We can formulate a general, monochromatic illumination model for surface reflections that includes multiple point light sources, attenuation factors, directional light effects (spotlight), infinite sources, and surface emissions as

$$I = I_{\text{surfemission}} + I_{\text{ambdiff}} + \sum_{l=1}^{n} f_{l,\text{radatten}} f_{l,\text{angatten}} (I_{l,\text{diff}} + I_{l,\text{spec}})$$

For each light source, we calculate the diffuse reflection from a surface point as

$$I_{l,\text{diff}} = \begin{cases} 0.0, & \text{if } \mathbf{N} \cdot \mathbf{L}_l \le 0.0 \text{ (light source behind object)} \\ k_d I_l (\mathbf{N} \cdot \mathbf{L}_l), & \text{otherwise} \end{cases}$$

The specular reflection term, due to a point-source illumination, is calculated with similar expressions:

$$I_{l,\text{spec}} = \begin{cases} 0.0, & \text{if } \mathbf{N} \cdot \mathbf{L}_l \le 0.0 \\ & \text{(light source behind object)} \\ k_s I_l \max\{0.0, (\mathbf{N} \cdot \mathbf{H}_l)^{n_s}\}, & \text{otherwise} \end{cases}$$

To ensure that any pixel intensity does not exceed the maximum allowable value, we can apply some type of normalization procedure.

A simple approach is to set a maximum magnitude for each term in the intensity equation. If any calculated term exceeds the maximum, we simply set it to the maximum value.

Another way to compensate for intensity overflow is to normalize the individual terms by dividing each by the magnitude of the largest term.

A more complicated procedure is to calculate all pixel intensities for the scene, then scale this set of intensities onto the intensity range from 0.0 to 1.0.

Also, the values for the coefficients in the radial attenuation function, and the optical surface parameters for a scene, can be adjusted to prevent calculated intensities from exceeding the maximum allowable value. This is an effective method for limiting intensity values when a single light source illuminates a scene.

In general, however, calculated intensities are never allowed to exceed the value 1.0, and negative intensity values are adjusted to the value 0.0.

RGB Color Considerations

For an RGB color description, each intensity specification in the illumination model is a three-element vector that designates the red, green, and blue components of that intensity. Thus, for each light source, $\boldsymbol{I_l = (I_{lR}, I_{lG}, I_{lB})}$.

Similarly, the reflection coefficients are also specified with RGB components: $\mathbf{k_a = (k_{aR}, k_{aG},)}$, $\mathbf{k_d = (k_{dR}, k_{dG}, k_{dB})}$, and $\mathbf{k_s = (k_{sR}, k_{sG}, k_{sB})}$. Each component of the surface color is then calculated with a separate expression.

For example, the blue component of the diffuse and specular reflections for a point source are computed as

$$I_{lB,\text{diff}} = k_{dB} I_{lB} (\mathbf{N} \cdot \mathbf{L}_l)$$

and

$$I_{lB,\text{spec}} = k_{sB} I_{lB} \max\{0.0, (\mathbf{N} \cdot \mathbf{H}_l)^{n_s}\}$$

Surfaces are most often illuminated with white light sources, but, for special effects or indoor lighting, we might use other colors for the light sources. We then set the reflectivity coefficients to model a particular surface color.

For example, if we want an object to have a blue surface, we select a nonzero value in the range from 0.0 to 1.0 for the blue reflectivity component, $\mathbf{k_{dB}}$, while the red and green reflectivity components are set to zero ($\mathbf{k_{dR} = k_{dG} = 0.0}$).

Any nonzero red or green components in the incident light are absorbed, and only the blue component is reflected.

In his original specular-reflection model, Phong set parameter $\mathbf{k_s}$ to a constant value independent of the surface color. This produces specular reflections that are the same color as the incident light (usually white), which gives the surface a plastic appearance.

For a nonplastic material, the color of the specular reflection is actually a function of the surface properties and may be different from both the color of the incident light and the color of the diffuse reflections.

We can approximate specular effects on such surfaces by making the specular-reflection coefficient color-dependent using the above equation.

Another method for setting surface color is to specify the components of diffuse and specular color vectors for each surface, while retaining the reflectivity coefficients as single-valued constants.

For an RGB color representation, for instance, the components of these two surface-color vectors could be denoted as (S_{dR}, S_{dG}, S_{dB}) and (S_{sR}, S_{sG}, S_{sB}).

The blue component of the diffuse reflection is then calculated as

$$I_{lB,\text{diff}} = k_d S_{dB} I_{lB} (\mathbf{N} \cdot \mathbf{L}_l) \qquad (24)$$

This approach provides somewhat greater flexibility, because surface color parameters and reflectivity values can be set independently.

In some graphics packages, additional lighting parameters are supplied by allowing a light source to be assigned multiple colors, where each color contributes to one of the surface lighting effects.

For example, one of the colors can be used as a contribution to the general background lighting in a scene. Similarly, another light-source color can be used as the light intensity for the diffuse reflection calculations, and a third light-source color can be used in the specular reflection calculations.

Other Color Representations:

We can describe colors using a variety of models other than the RGB representation.
For example, a color can be represented using cyan, magenta, and yellow components, or a color could be described in terms of a particular hue along with the perceived brightness and saturation of the color.

We can incorporate any of these representations, including color specifications with more than three components, into an illumination model. As an example, Equation 24 can be expressed in terms of any spectral color with wavelength λ as

$$I_{l\lambda,\text{diff}} = k_d S_{d\lambda} I_{l\lambda}(\mathbf{N} \cdot \mathbf{L}_l)$$

Luminance:

Another characteristic of color is luminance, which is sometimes also called luminous energy. Luminance provides information about the lightness or darkness level of a color, and it is a psychological measure of our perception of brightness that varies with the amount of illumination we are viewing.

Physically, color is described in terms of the frequency range for visible radiant energy (light), and luminance is calculated as a weighted sum of the intensity components in a particular illumination.

Because any actual illumination contains a continuous range of frequencies, a luminance value is computed as

$$\text{luminance} = \int_{\text{visible } f} p(f)\, I(f)\, df$$

Parameter ***I(f)*** in this calculation represents the intensity of the light component with a frequency f that is radiating in a particular direction.
Parameter ***p(f)*** is an experimentally determined proportionality function that varies with both frequency and illumination level. The integration is performed for all intensities over the frequency range contained in the light.

For grayscale and monochromatic displays, we need only the luminance values to describe object lighting. And some graphics packages do allow the lighting parameters to be expressed in terms of luminance.

Green components of a light source contribute most to the luminance, and blue components contribute least. Therefore, the luminance of an RGB color source is typically computed as

$$\text{luminance} = 0.299R + 0.587G + 0.114B$$

The luminance parameter is most often represented with the symbol Y, which corresponds to the Y component in the XYZ color model.